Artificial Intelligence

UNIT I INTRODUCTION

Introduction–Definition - Future of Artificial Intelligence – Characteristics of Intelligent Agents– Typical Intelligent Agents – Problem Solving Approach to Typical AI problems.

UNIT II PROBLEM SOLVING METHODS

Problem solving Methods - Search Strategies- Uninformed - Informed - Heuristics - Local Search Algorithms and Optimization Problems - Searching with Partial Observations – Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search - Game Playing – Optimal Decisions in Games – Alpha - Beta Pruning - Stochastic Games.

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.

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.

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.

TEXT BOOKS:

1 S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach||, Prentice Hall, Third Edition, 2009.

2 I. Bratko, —Prolog: Programming for Artificial Intelligence||, Fourth edition, Addison-Wesley Educational Publishers Inc., 2011.

REFERENCES:

1. M. Tim Jones, —Artificial Intelligence: A Systems Approach(Computer Science)||, Jones and Bartlett Publishers, Inc.; First Edition, 2008

2. Nils J. Nilsson, —The Quest for Artificial Intelligence||, Cambridge University Press, 2009.
3. William F. Clocksin and Christopher S. Mellish,|| Programming in Prolog: Using the ISO Standard||, Fifth Edition, Springer, 2003.
4. Gerhard Weiss, —Multi Agent Systems||, Second Edition, MIT Press, 2013.
5. David L. Poole and Alan K. Mackworth, —Artificial Intelligence: Foundations of Computational Agents||, Cambridge University Press, 2010

UNIT I INTRODUCTION

Introduction–Definition - Future of Artificial Intelligence – Characteristics of Intelligent Agents–Typical Intelligent Agents – Problem Solving Approach to Typical AI problems.

Artificial Intelligence – An Introduction

What is AI?

Artificial intelligence is the study of how to make computers do things which, at the moment people do better.

Some definitions of artificial intelligence, organized into four categories

I. Systems that think like humans

1. "The exciting new effort to make computers think machines with minds, in the full and literal sense." **(Haugeland, 1985)**

2. "The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning" **(Bellman, 1978)**

II. Systems that act like humans

3. "The art of creating machines that performs functions that require intelligence when performed by people." **(Kurzweil, 1990)**

4. "The study of how to make computers do things at which, at the moment, people are better." **(Rich and Knight, 1991)**

III. Systems that think rationally

5. "The study of mental faculties through the use of computational models." **(Chamiak and McDermott, 1985)**

6. "The study of the computations that make it possible to perceive, reason, and act." **(Winston, 1992)**

IV. Systems that act rationally

7. "Computational Intelligence is the study of the design of intelligent agents." **(Poole et al., 1998)**

8. "AI is concerned with intelligent behavior in artifacts." **(Nilsson, 1998)**

The definitions on the 1, 2, 3, 4 measure success in terms of human performance, whereas the ones on the 5, 6, 7, 8 measure against an ideal concept of intelligence.

A system is **rational** if it does the "right thing," given what it knows.

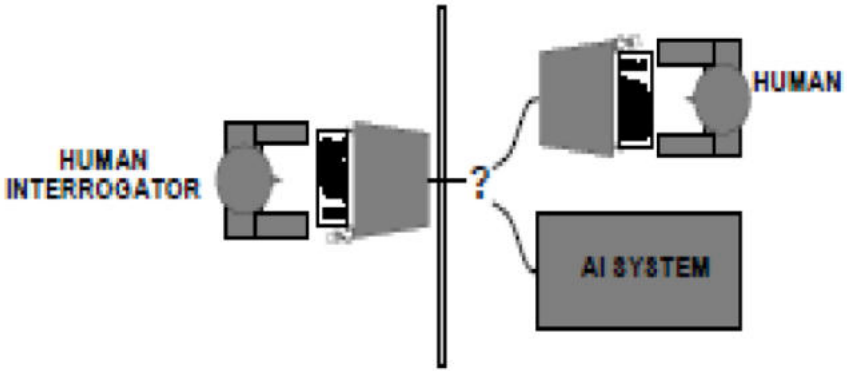
The term AI is defined by each author in its own perceive, leads to four important categories

i. Acting humanly: The Turing Test approach
ii. Thinking humanly: The cognitive modeling approach
iii. Thinking rationally: The "laws of thought" approach
iv. Acting rationally: The rational agent approach

(i) Acting humanly: The Turing Test approach

To conduct this test, we need two people and the machine to be evaluated. One person plays the role of the interrogator, who is in a separate room from the

computer and the other person. The interrogator can ask questions of either the person or the computer but typing questions and receiving typed responses. However, the interrogator knows them only as A and B and aims to determine which the person is and which is the machine.



The goal of the machine is to fool the interrogator into believing that is the person. If the machine succeeds at this, then we will conclude that the machine is acting humanly. But programming a computer to pass the test provides plenty to work on, to possess the following capabilities.

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

Total Turing Test: the test which includes a video so that the interrogator can test the perceptual abilities of the machine. To undergo the total Turing test, the computer will need

- computer vision to perceive objects, and
- robotics to manipulate objects and move about

(ii) Thinking humanly: The cognitive modeling approach

To construct a machines program to think like a human, first it requires the knowledge about the actual workings of human mind. After completing the study about human mind it is possible to express the theory as a computer program.

If the program's inputs/output and timing behavior matched with the human behavior then we can say that the program's mechanism is working like a human mind.

Example: General Problem Solver (GPS) – A problem solvers always keeps track of human mind regardless of right answers. The problem solver is contrast to other researchers, because they are concentrating on getting the right answers regardless of the human mind.

An Interdisciplinary field of cognitive science uses computer models from AI and experimental techniques from psychology to construct the theory of the working of the human mind.

(iii) Thinking rationally: The "laws of thought" approach

Laws of thought were supposed to govern the operation of the mind and their study initiated the field called **logic**

Example 1:"Socrates is a man; All men are mortal; therefore, Socrates is mortal."

Example 2:"Ram is a student of III year CSE; All students are good in III year CSE; therefore, Ram is a good student"

Syllogisms : A form of deductive reasoning consisting of a major premise, a minor premise, and a conclusion

Syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises

There are two main obstacles to this approach.

1. It is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less.

2. There is a big difference between being able to solve a problem "in principle" and doing so in practice

(iv) Acting rationally: The rational agent approach

An **agent** is just something that acts. A rational **agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. The study of rational agent has two advantages.

1. Correct inference is selected and applied
2. It concentrates on scientific development rather than other methods.

Foundation of Artificial Intelligence

AI derives the features from Philosophy, Mathematics, Psychology, Computer Engineering, Linguistics topics.

Philosophy(428 B.C. – present)

Aristotle (384-322 B.C.) was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which allowed one to generate conclusions mechanically, given initial premises.

Mathematics (c. 800-present)

- What are the formal rules to draw valid conclusions?

- What can be computed?
- How do we reason with uncertain information?

Philosophers staked out most of the important ideas of k1, but the leap to a formal science required a level of mathematical formalization in three fundamental areas: logic, computation, and probability

Economics (1776-present)

- How should we make decisions so as to maximize payoff?
- How should we do this when others may not go along?

The science of economics got its start in 1776, when Scottish philosopher Adam Smith (1723-1790) published An Inquiry into the Nature and Causes of the Wealth of Nations. While the ancient Greeks and others had made contributions to economic thought, Smith was the first to treat it as a science, using the idea that economies can be thought of as consisting of individual agents maximizing their own economic well-being

Neuroscience (1861-present)

- How do brains process information?

Neuroscience is the study of the nervous system, particularly the brain. The exact way in which the brain enables thought is one of the great mysteries of science. It has been appreciated for thousands of years that the brain is somehow involved in thought, because of the evidence that strong blows to the head can lead to mental incapacitation

	Computer	**Human Brain**

Computational units	1 CPU,10^8 gates	10^{11} neurons
Storage units	1010 bits RAM	10^{11} neurons
	1011 bits disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	109	10^{14}
Comparison of the raw computational resources and brain.		

Psychology (1879 – present)

The origin of scientific psychology are traced back to the wok if German physiologist Hermann von Helmholtz(1821-1894) and his student Wilhelm Wundt(1832 – 1920). In 1879, Wundt opened the first laboratory of experimental psychology at the University of Leipzig. In US,the development of computer modeling led to the creation of the field of **cognitive science**. The field can be said to have started at the workshop in September 1956 at MIT.

Computer engineering (1940-present)

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice.**A1** also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs

Control theory and Cybernetics (1948-present)

Ktesibios of Alexandria (c. 250 B.c.) built the first self-controlling machine: a water clock with a regulator that kept the flow of water running through it at a constant, predictable

pace. Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an **objective function** over time.

Linguistics (1957-present)

Modem linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called **computational linguistics** or **natural language processing.**

History of Artificial Intelligence

The gestation of artificial intelligence (1943-1955)

There were a number of early examples of work that can be characterized as AI, but it was Alan Turing who first articulated a complete vision of A1 in his 1950 article "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning.

The birth of artificial intelligence (1956)

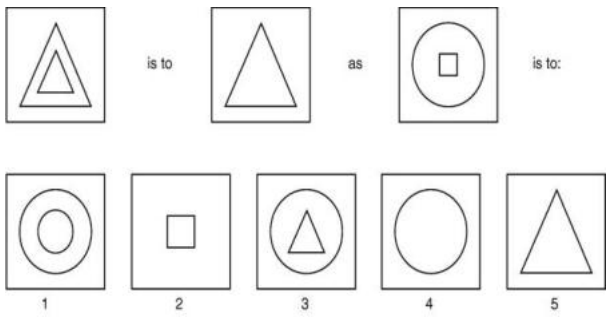
McCarthy convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at Dartmouth in the summer of 1956. Perhaps the longest-lasting thing to come out of the workshop was an agreement to adopt McCarthy's new name for the field: **artificial intelligence.**

Early enthusiasm, great expectations (1952-1969)

The early years of A1 were full of successes-in a limited way. **General Problem Solver** (**GPS**) was a computer program created in 1957 by Herbert Simon and Allen Newell to build a universal problem solver machine. The order in which the program considered

subgoals and possible actions was similar to that in which humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach. At IBM, Nathaniel Rochester and his colleagues produced some of the first A1 programs. Herbert Gelernter (1959) constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky.

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). In 1963, McCarthy started the AI lab at Stanford. Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests, such as the one in Figure



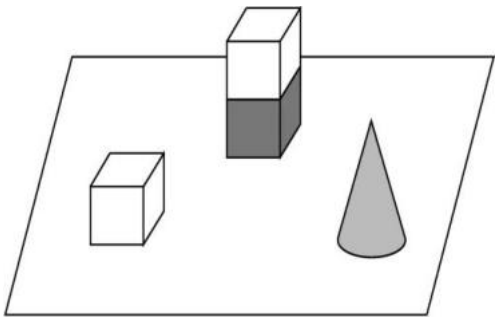


Fig : The Tom Evan's ANALOGY program could solve geometric analogy problems as shown.

A dose of reality (1966-1973)

From the beginning, AI researchers were not shy about making predictions of their coming successes. The following statement by Herbert Simon in 1957 is often quoted:

"It is not my aim to surprise or shock you-but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until-in a visible future-the range of problems they can handle will be coextensive with the range to which the human mind has been applied.

Knowledge-based systems: The key to power? (1969-1979)

Dendral was an influential pioneer project in artificial intelligence (AI) of the 1960s, and the computer software **expert system** that it produced. Its primary aim was to help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry. It was done at Stanford University by Edward Feigenbaum, Bruce Buchanan, Joshua Lederberg, and Carl Djerassi.

AI becomes an industry (1980-present)

In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog. Overall, the A1 industry boomed from a few million dollars in 1980 to billions of dollars in 1988.

The return of neural networks (1986-present)

Psychologists including David Rumelhart and Geoff Hinton continued the study of neural-net models of memory.

AI becomes a science (1987-present)

In recent years, approaches based on **hidden Markov models** (HMMs) have come to dominate the area. Speech technology and the related field of handwritten character recognition are already making the transition to widespread industrial and consumer applications.

The **Bayesian network** formalism was invented to allow efficient representation of, and rigorous reasoning with, uncertain knowledge.

The emergence of intelligent agents (1995-present)

One of the most important environments for intelligent agents is the Internet.

Sample Applications

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. 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 (Garry Kasparov) in a chess match. The value of IBM's stock increased by $18 billion.

Autonomous control: The ALVINN computer vision system was trained to steer a car to keep it following a lane. The computer-controlled minivan used to navigate across the United States-for 2850 miles and it was in control of steering the vehicle 98% of the time. A human took over the other 2%, mostly at exit ramps.

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 Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART 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

Robotics: Many surgeons now use robot assistants in microsurgery

Language understanding and problem solving: PROVERB 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.

Typical problems to which AI methods are applied

Pattern recognition, Optical character recognition , Handwriting recognition , Speech recognition , Face recognition, Computer vision, Virtual reality and Image processing , Diagnosis , Game theory and Strategic planning , Natural language processing, Translation and Chatterboxes , Nonlinear control and Robotics, Artificial life, Automated reasoning , Automation , Biologically inspired computing ,Concept mining , Data mining , Knowledge representation , Semantic Web , E-mail spam filtering, Robotics, ,Cognitive , Cybernetics , Hybrid intelligent system, Intelligent agent ,Intelligent control.

INTELLIGENT AGENTS

Introduction - Agents and Environments

An agent is anything that can be viewed as perceiving its environment through sensors andacting upon that environment through actuators.

Different types of agents

1. A **human agent** has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.

2. A **robotic agent** might have cameras and infrared range finders for sensors and various motors for actuators.

3. 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.

4. **Generic agent** – A general structure of an agent who interacts with the environment.

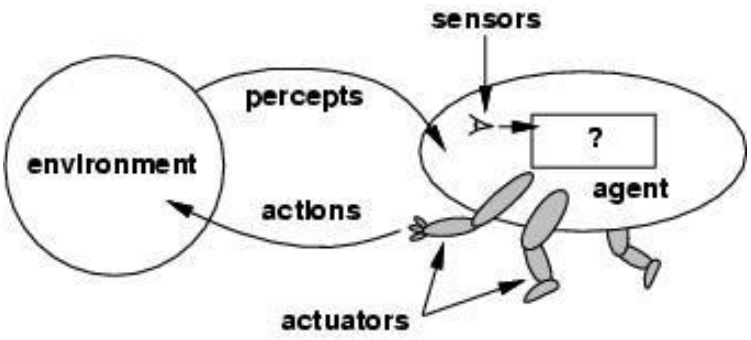


Fig : Agents interact with environments through sensors and effectors (accuators)

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

PERCEPT SEQUENCE: Agent's percept sequence is the complete history of everything the agent has ever perceived.

An agent's behavior is described by the agent function that maps any given percept sequence to an action.

AGENT PROGRAM : The agent function for an artificial agent will be implemented by an agent program.

Example : The vacuum-cleaner 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 move to the other square.

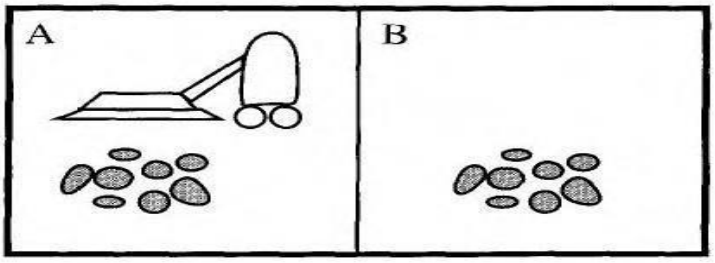


Fig : A vacuum-cleaner world with just two locations

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

- Percepts: location and status, e.g., [A,Dirty]
- Actions: Left, Right, Suck, NoOp

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

[B, Dirty]	Suck

The agent program for a simple agent in the two-state vacuum environment for above tabulation

function VACUUM-AGENT([location,status]) if status = Dirty then return Suck else if location = A then return Right else if location = B then return Left **Concept of Rationality**

A rational agent is one that does the right 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

Rational at any given time depends on four things:

1. The performance measure that defines the criterion of success.
2. The agent's prior knowledge of the environment.
3. The actions that the agent can perform.
4. The agent's percept sequence to date.

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. A rational agent should be autonomous

Definition of an omniscient agent:

An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.

Autonomy

A rational agent should be autonomous-it should learn what it can to compensate for partial or incorrect prior knowledge.

Information Gathering

Doing actions in order to modify future percepts is called as information gathering.

Specifying the task environment

In the discussion of the rationality of any agent, we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these together under the heading of the task environment and we call this as **PEAS (Performance, Environment, Actuators, Sensors)** or **PAGE (Percept, Action, Goal, Environment)** description. In designing an agent, the first step must always be to specify the task environment.

Example : PEAS description of the task environment for agents

Agent Type	Performance Measure	Environment	Actuators	Sensors
Automated Taxi Driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, traffic, pedestrian customers	Steering accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer engine sensors, keyboard
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Screen display (question tests, diagnoses treatment referrals)	Keyboard (entry of symptoms, findings, patient's answers)
Part-Picking Robot	Percentage of parts in correct	Conveyor belt with parts, bins	Jointed arm and hand	Camera, joint angle sensors

	bin			
Interactive English tutor	Maximize student's score on test	Set of students	Screen display (exercises)	Keyboard
robot soccer player	amount of goals scored	soccer match field	legs	cameras, sonar or infrared
Satellite Image Analysis	Correct Image Categorization	Downlink from satellite	Display categorizati on of scene	Color pixel arrays
Refinery controller	Maximum purity, safety	Refinery operators	Valves, pumps, heaters,	Temperature, pressure, chemical sensors
			displays	
Vacuum Agent	minimize energy consumption, maximize dirt pick up	two squares	Left, Right, Suck, NoOp	Sensors to identify the dirt

Properties of task environments (Environment Types)

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 chess playing system is an example of a system that operates in a fully observable environment.

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. A bridge playing program is an example of a system operating in a partially observable environment.

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

Image analysis systems are examples of deterministic. The processed image is determined completely by the current image and the processing operations.

Taxi driving is clearly stochastic in this sense, because one can never predict the behavior of traffic exactly;

Episodic vs. sequential

An episodic environment means that subsequent episodes do not depend on what actions occurred in previous episodes.

In a sequential environment, the agent engages in a series of connected episodes. In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential.

Static vs. dynamic

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Taxi driving is clearly dynamic. Crossword puzzles are static.

Discrete vs. continuous

If the number of distinct percepts and actions is limited, the environment is discrete, otherwise it is continuous. Taxi driving is a continuous state and continuous-time problem. Chess game has a finite number of distinct states.

Single agent vs. Multi agent

The distinction between single-agent and multi agent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment.

Chess is a competitive multi agent environment. Taxi-driving environment is a partially cooperative multi agent environment.

<u>Environment Characteristics</u>

Examples of task environments and their characteristics

Task Environment	**Observable**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Agent**
Crosswordpuzz	Fully	Deterministic	Sequentia	Stati	Discrete	Sing

le			l	c		le
Chess with a clock	Fully	Stochastic	Sequential	Semi	Discrete	Multi
Poker	Partially	Stochastic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxidnving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diag	Partially	Stochastic	Sequen	Dyn	Continu	Sin

nosis			tial	amic	ous	gle
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tuto	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

r						

- The simplest environment is
- Fully observable, deterministic, episodic, static, discrete and singleagent.
- Most real situations are:
- Partially observable, stochastic, sequential, dynamic, continuous and multi-agent.

Structure of the Agents

The job of AI is to design the agent program that implements the agent function mapping percepts to actions.

Intelligent agent = Architecture + Agent program

Agent programs

Agent programs take the current percept as input from the sensors and return an action to the actuators

The agent program takes the current percept as input, and the agent function takes the entire percept history

Architecture is a computing device used to run the agent program.

The agent programs will use some internal data structures that will be updated as new percepts arrive. The data structures are operated by the agents decision making procedures to generated an action choice, which is then passed to the architecture to be executed. Two types of agent programs are

1. A Skeleton Agent 2. A Table Lookup Agent

Skeleton Agent

The agent program receives only a single percept as its input.
If the percept is a new input then the agent updates the memory with the new percept

function SKELETON-AGENT(*percept*) **returns** action **static**: *memory*, the agent's memory of the world *memory* <- UPDATE-MEMORY(*memory, percept*) *action* <- CHOOSE-BEST-ACTION(*memory*) *memory* <- UPDATE-MEMORY(*memory, action*) **return** *action*

Table-lookup agent

A table which consists of indexed percept sequences with its corresponding action

The input percept checks the table for the same

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 of table lookup agent

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

Four basic types in order of increasing generality

- Simple reflex agents
- Model-based reflex agents
- Goal-based agents
- Utility-based agents

Simple reflex agents

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.

This agent describes about how the condition – action rules allow the agent to make the connection from percept to action

It acts according to a rule whose condition matches the current state, as defined by the percept.

Condition – action rule : if condition then action

Example : condition-action rule: if car-in-front-is-braking then initiatebraking

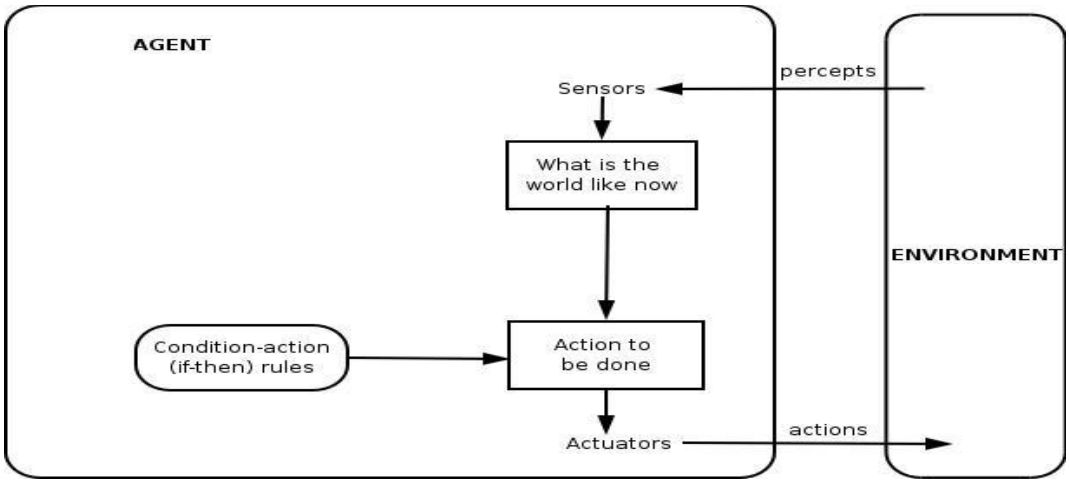


Fig : Schematic diagram of a simple reflex agent.

Rectangles - to denote the current internal state of the agent's decision process

Ovals - to represent the background information used in the process.

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

state < - I N T E R P R E T - I N P U T (percept)
rule <- R U L E - M A T C H (state, rules), action <- R U L E - A C T I O N [rule]
return action

- **INTERPRET-INPUT** function generates an abstracted description of the current state from the percept
- **RULE-MATCH** function returns the first rule in the set of rules that matches the given state description
- **RULE - ACTION** – the selected rule is executed as action of the given percept

Example : Medical Diagnosis System

If the patient has reddish brown spots then start the treatment for measles.

Model based Reflex Agents

An agent which combines the current percept with the old internal state to generate updated

function REFLEX-AGENT-WITH-STATE(percept)returns action static: state, a description of the current world state rules, a set of condition-action rules action, the most recent action, initially none state <- UPDATE-STATE(state, action, percept) rule <- RULE - MATCH (state, rules) action <- RULE-ACTION [rule]

description of the current state.

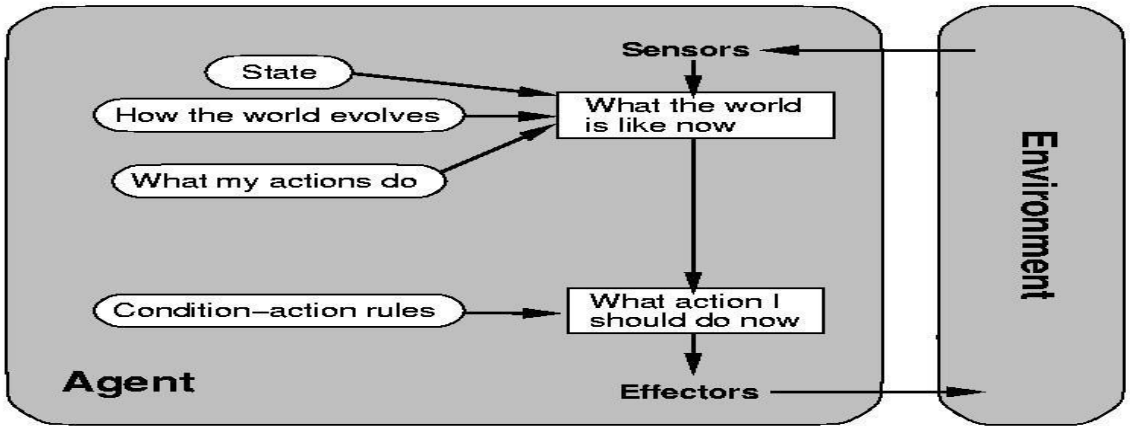
UPD ATE-

STATE - is responsible for creating the new internal state description

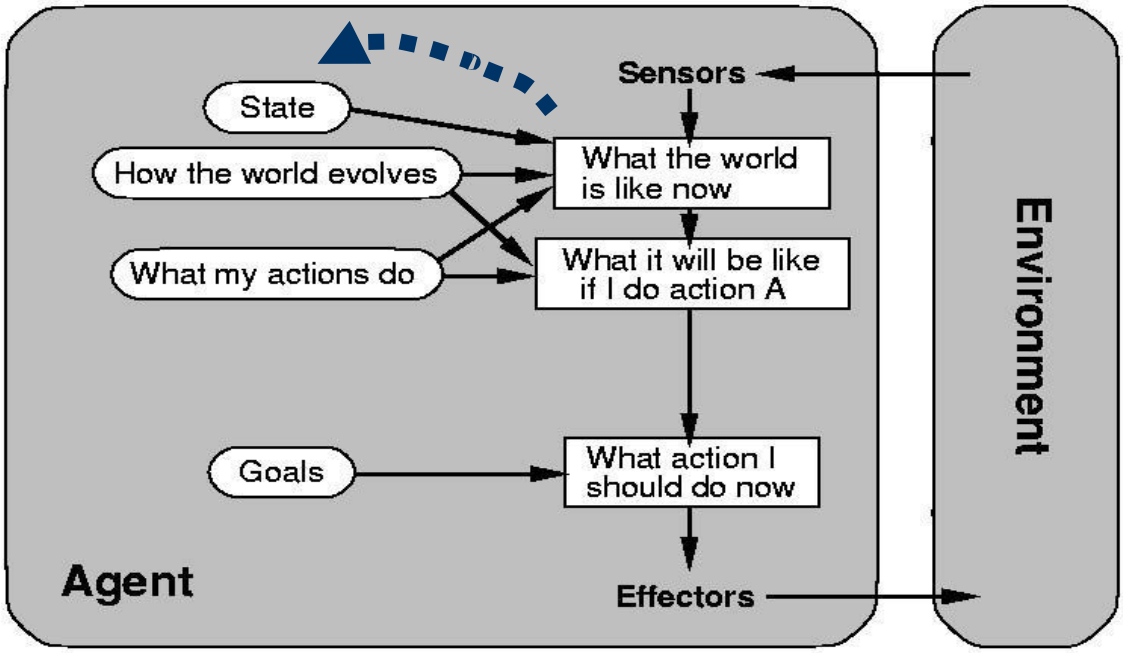
Example: Medical Diagnosis system

If the Patient has spots then check the internal state (i. e) any change in the environment may lead to cause spots on the patient. From this internal state the current state is updated and the corresponding action is executed.

Goal based Agents



An Agent knows the description of current state as well as goal state. The action matches with the current state is selected depends on the goal state.



Example : Medical diagnosis system

If the name of disease is identified for the patient then the treatment is given to the patient to recover from him from the disease and make the patient healthy is the goal to be achieved

Utility base agents

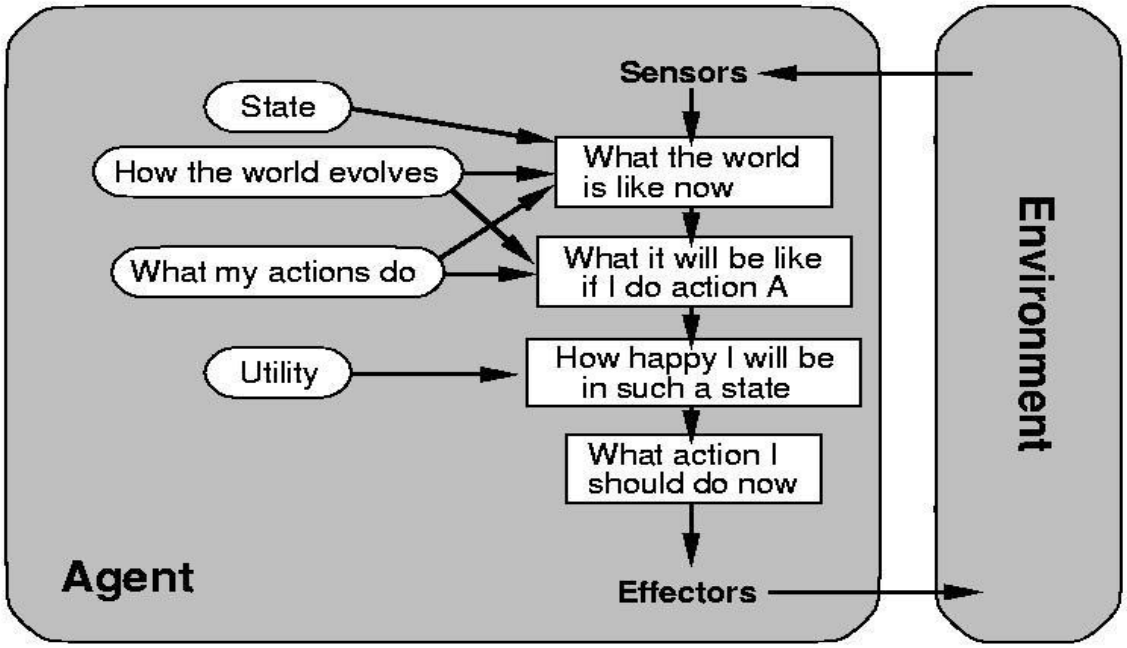
An agent which generates a goal state with high – quality behavior (i.e) if more than one sequence exists to reach the goal state then the sequence with more reliable, safer, quicker and cheaper than others to be selected.

Utility is a function that maps a state onto a real number, which describes the associated degree of happiness

The utility function can be used for two different cases :

1. When there are conflicting goals, only some of which can be achieved (for example, speed and safety)

2. When there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed up against the importance of the goal



Example : Medical diagnosis System

If the patient disease is identified then the sequence of treatment which leads to recover the patient with all utility measure is selected and applied

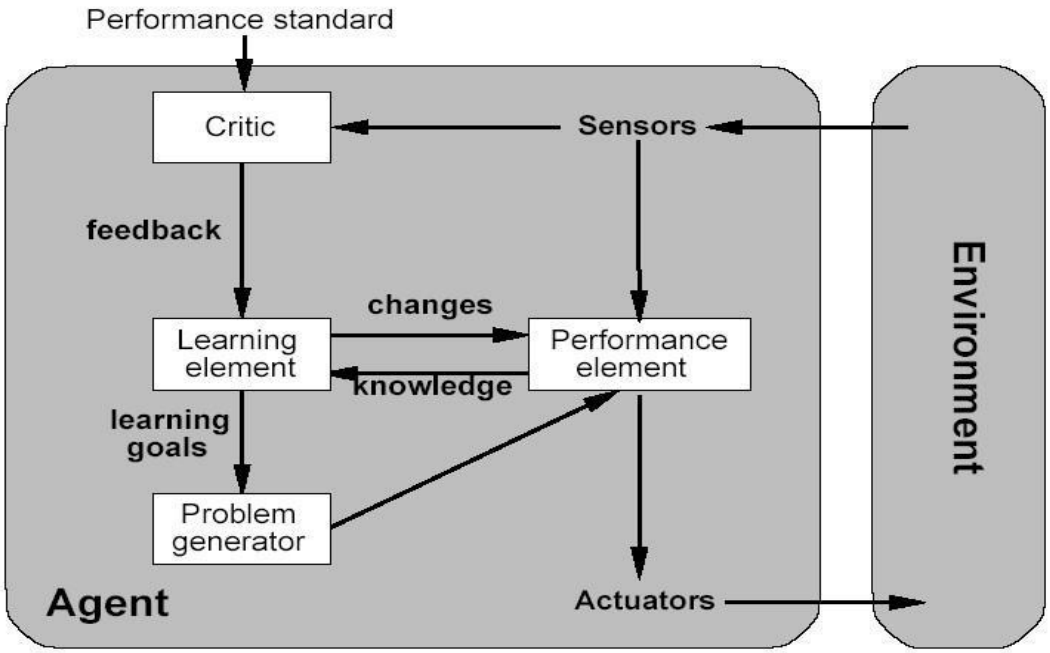
Learning agent

All agents can improve their performance through Learning

The learning task allows the agent to operate in unknown environments initially and then become more competent than its initial knowledge.

A learning agent can be divided into four conceptual components:

1. Learning element
2. performance element
3. Critic
4. Problem generator



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. Learning element is also responsible for making improvements

Performance element is to select external action and it is equivalent to agent

The **critic** tells the learning element how well the agent is doing with respect to a fixed performance standard

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.

Problem solving – Introduction

Search is one of the operational tasks that characterize AI programs best. Almost every AI program depends on a search procedure to perform its prescribed functions. Problems are typically defined in terms of state, and solution corresponds to goal states.

Problem solving using search technique performs two sequence of steps:

(i) **Define the problem** - Given problem is identified with its required initial and goal state.
(ii) **Analyze the problem** - The best search technique for the given: problem is chosen from different an AI search technique which derives one or more goal state in minimum number of states.

Types of problem

In general the problem can be classified under anyone of the following four types which depends on two important properties. They are

(i) Amount of knowledge, of the agent on the state and action description. (ii) How the agent is connected to its environment through its percepts and actions?

The four different types of problems are:

(i) Single state problem
(ii) Multiple state problem
(iii) Contingency problem
(iv) Exploration problem

Problem solving Agents

Problem solving agent is one kind of goal based agent, where the agent decides what to do by finding sequence of actions that lead to desirable states. The complexity arises

here is the knowledge about the formulation process, (from current state to outcome action) of the agent.

If the agent understood the definition of problem, it is relatively straight forward to construct a search process for finding solutions, which implies that problem solving agent should be an intelligent agent to maximize the performance measure.

The sequence of steps done by the intelligent agent to maximize the performance measure:

i) Goal formulation - based on current situation is the first step in problem solving. Actions that result to a failure case can be rejected without further consideration.

(ii)Problem formulation - is the process of deciding what actions and states to consider and follows goal formulation.

(iii) Search - is the process of finding different possible sequence of actions that lead to state of known value, and choosing the best one from the states. **(iv) Solution** - a search algorithm takes a problem as input and returns a solution in the form of action sequence.

(v) Execution phase - if the solution exists, the action it recommends can be carried out.

A simple problem solving agent

function SIMPLE-PROBLEM-SOLVING-AGENT(p) **returns** an action **input :** *p,* a percept
static: *s,* an action sequence, initially empty *state,* some description of the current world state g, a goal initially null *problem,* a problem formulation *state* <- UPDATE-STATE(state, *p)* if *s* is empty then *g* <- *FORMULATE-GOAL(state) problem* <- FORMULATE-PROBLEM(state,*g)* s <- *SEARCH(problem) action* <- RECOMMENDATION(s, *state) s* <- *REMAINDER(s, state)* **return** *action*

Note :

RECOMMENDATION - first action in the sequence

REMAINDER - returns the rest

SEARCH - choosing the best one from the sequence of actions

FORMULATE-PROBLEM - sequence of actions and states that lead to goal state.

UPDATE-STATE - initial state is forced to next state to reach the goal state

Well-defined problems and solutions

A problem can be defined formally by four components:

1. initial state 2. successor function 3. goal test 4. **path cost**

The **initial state** that the agent starts in.

Successor function (S) - Given a particular state x, S(x) returns a set of states reachable from x by any single action.

The **goal test,** which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

A **path cost** function that assigns a numeric cost to each path. The problemsolving agent chooses a cost function that reflects its own performance measure.

A **solution** to a problem is a path from the initial state to a goal state

Operator - The set of possible actions available to the agent.

State space (or) state set space - The set of all possible states reachable from the initial state by any sequence of actions.

Path (state space) - The sequence of action leading from one state to another

The effectiveness of a search can be measured using three factors. They are:

1 Solution is identified or not?

2. Is it a good solution? If yes, then path cost to be minimum.

3. Search cost of the problem that is associated with time and memory required to find a solution.

For Example

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the goal of getting to Bucharest. The agent's task is to find out which sequence of actions will get it to a goal state.

This process of looking for such a sequence is called search.

A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.

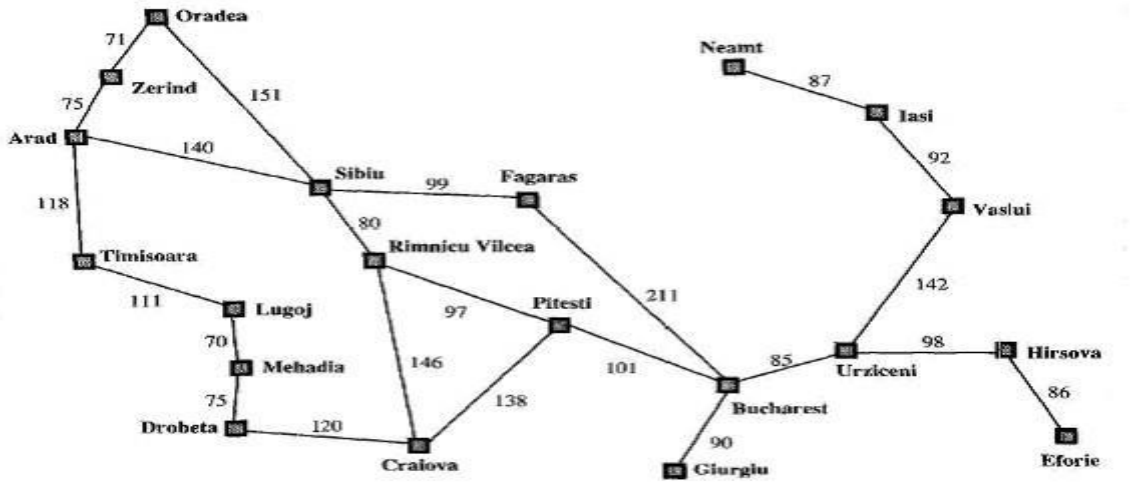
Formulating problems

Initial state : the initial state for our agent in Romania might be described as In(Arad)

Successor function : Given a particular state x, SUCCESSOR-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(Sibzu),In(Sibiu)), (Go(Timisoara), In(Tzmisoara)), (Go(Zerznd),In(Zerind)))

Goal test : The agent's goal in Romania is the singleton set {In(Bucharest)).

Path cost : The **step cost** of taking action a to go from state x to state y is denoted by c(x, a, y).



Example Problems

The problem-solving approach has been applied to a vast array of task environments.

A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description. It can be used easily by different researchers to compare the performance of algorithms

A real-world problem is one whose solutions people actually care about.

Some list of best known toy and real-world problems

Toy Problems

i) Vacuum world Problem

States: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are $2 * 2^2 = 8$ possible world states.

Initial state: Any state can be designated as the initial state.

Successor function: three actions (Left, **Right,** and **Suck).**

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

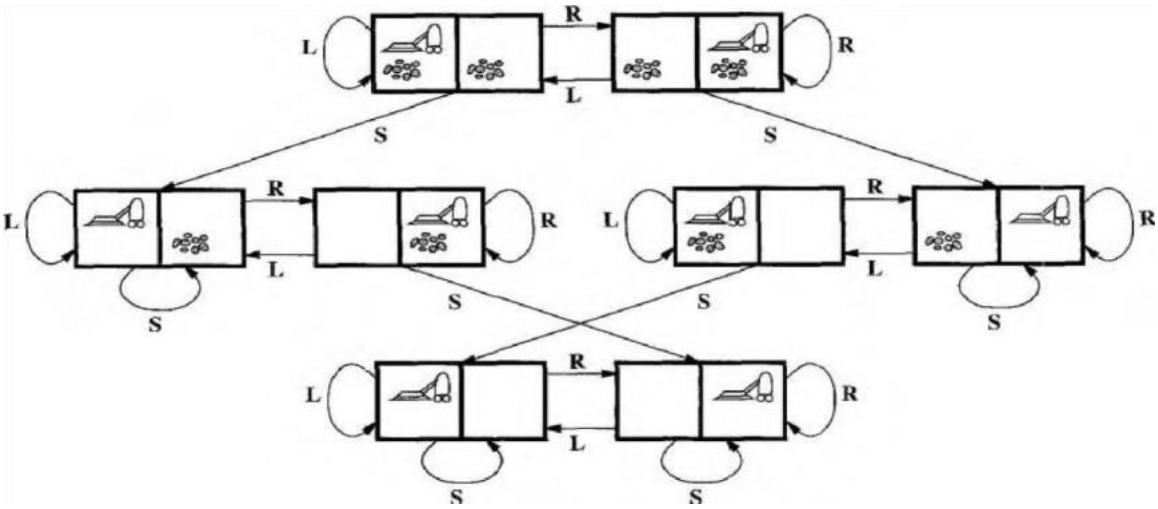


Fig : The complete state space for Vacuum World

ii) 8-puzzle Problem

The 8-puzzle problem consists of a 3 x 3 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 a specified goal state

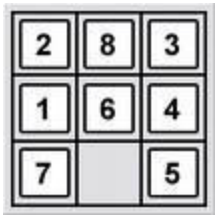
States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

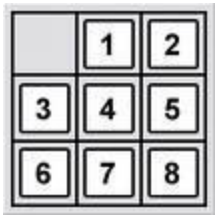
Initial state: Any state can be designated as the initial state.

Successor function: This generates the legal states that result from trying the four actions (blank moves Left, Right, Up, or Down).

Goal test: This checks whether the state matches the goal configuration (Other goal configurations are possible.)

Path cost: Each step costs 1, so the path cost is the number of steps in the path.





Initial State **Goal State**

iii) 8-queens problem

The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.

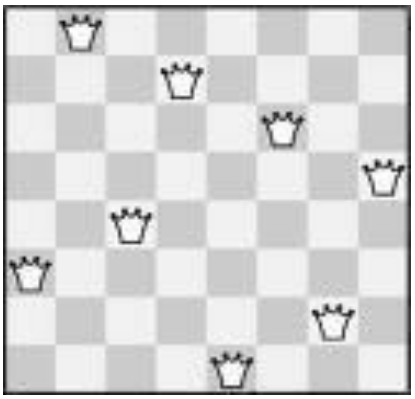
States: Any arrangement of 0 to 8 queens on the board is a state.

Initial state: No queens on the board.

Successor function: Add a queen to any empty square.

Goal test: 8 queens are on the board, none attacked.

Path cost : Zero (search cost only exists)



solution to the 8-queens problem.

iv) Crypt arithmetic Problem

In crypt arithmetic problems letters stand for digits and the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, each letter stand for a different digit

Rules

There should be no more than 10 distinct characters
The summation should be the longest word
The summation can not be too long
There must be a one-to-one mapping between letters and digits The leftmost letter can't be zero in any word.

States: A crypt arithmetic puzzle with some letters replaced by digits
Initial state: No digits is assigned to the letters
Successor function: Replace all occurrences of a letter with a digit not already appearing in the puzzle
Goal test: Puzzle contains only digits and represents a correct sum
Path cost : Zero

Example 1:

```
    S E N D
+ M O R E
---------      M O N E Y
```

Solution : S=9 , E = 5, N = 6, D=7, M= 1, O= 0, R = 8, Y=2

Example 2:

```
   FORTY
   +TEN
   +TEN
-------  SIXTY
-------
```

Solution : F=2, O=9, R=7, T=8 , Y=6, E=5, N=0

v) Missionaries and cannibals problem

Three missionaries and three cannibals are on one side of a river, along with a oat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place out numbers by the cannibals in that place

Assumptions :

1. Number of trips is not restricted
2. Both the missionary and cannibal can row the boat

States: A state consists of an ordered sequence of two numbers representing the number of missionaries and cannibals

Example : (i,j) = (3,3) three missionaries and three cannibals

Initial state: (i,j) = (3,3) in one side of the river

Successor function: The possible move across the river are:

1. One Missionary and One Cannibal
2. Two Missionaries
3. Two Cannibals
4. One Missionary
5. One Cannibal

Rule No.	Explanation

(i)	(i, j) : One missionary and one cannibal can cross the river only when ((i-1) >= (j-1)) in one side of the river and ((i+1) >= (j+ 1)) in the other side of the river.
(ii)	(i,j) : Two missionaries can cross the river only when ((i-2)>=j) in one side of the river and ((i+2)>=j) in the other side of the river.
(iii)	(i,j) : Two cannibals can cross the river only when ((j-2)<= i) in one side of the river and ((j+2)<= i) in the other side of the river.
(iv)	(i,j) : One missionary can cross the river only when ((i-1)>=j)) in one side of the river and ((i-1)>=j)) in the other side of the river.
(v)	(i,j) : One cannibal can cross the river only when (((j-l)<=i) in one side of the river and (((j+l)<=i)in the other side of the river.

Initial state : (i.j) = (3,3) in one side of the river.

Goal test: (i,j) = (3,3) in the other side of the river.

Path cost : Number of crossings between the two sides of the river.

Solution:

Bank 1		Boat		Bank 2	Rule Applied
(i,j)=(3,3)	->		<-	(i,j)=(0,0)	
(3,1)	<-	(0,2)	->	(0,2)	(iii)
(3,2)	->	(0,1)	<-	(0,1)	(v)
(3,0)	<-	(0,2)	->	(0,3)	(iii)
(3,1)	->	(0,1)	<-	(0,2)	(v)
(1,1)	<-	(2,0)	->	(2,2)	(ii)
(2,2)	->	(1,1)	<-	(1,1)	(i)
(0,2)	<-	(2	->	(3,1)	(ii)

		,0)					
(0,3)	-> <- ->	(0,1)	-> <- ->	(3,0)		(v)	
(0,1)		(0,2)		(3,2)		(iii)	
(0,2)		(0,1)		(3,1)		(v)	
(0,0)		(0,2)		(3,3)		(iii)	

Real-world problems

Airline travel problem

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 (perhaps 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 pre specified time?

Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequentflyer mileage awards, and so on.

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 airline travel planning systems

The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.

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 comes after the logical design phase, and is usually split into two parts: **cell layout** and **channel routing.** In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells.

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) 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 must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). In assembly problems, the aim is to find an order in which to assemble the parts of some object. **If** the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.

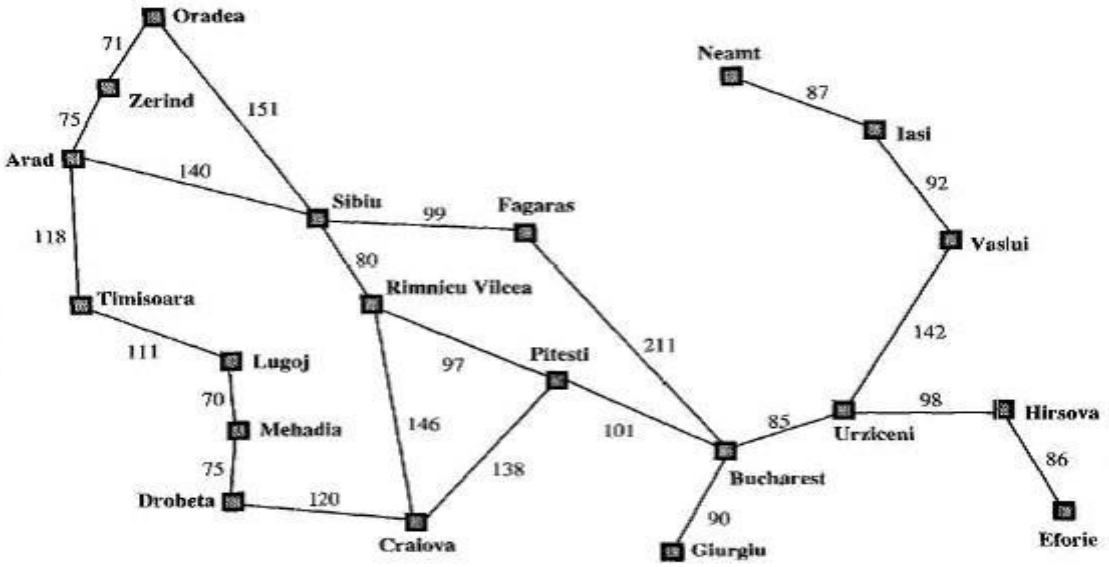
UNIT II- PROBLEM SOLVING METHODS

Problem solving Methods - Search Strategies- Uninformed - Informed - Heuristics - Local Search Algorithms and Optimization Problems - Searching with Partial Observations - Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search - Game Playing - Optimal Decisions in Games – Alpha - Beta Pruning - Stochastic Games.

Searching for Solutions

Search techniques use an explicit **search tree** that is generated by the initial state and the successor function that together define the state space. In general, we may have a search graph rather than a search tree, when the same state can be reached from multiple paths

Example Route finding problem



The root of the search tree is a **search node** corresponding to the initial state, In(Arad).

The first step is to test whether this is a goal state.

Apply the successor function to the current state, and **generate** a new set of states

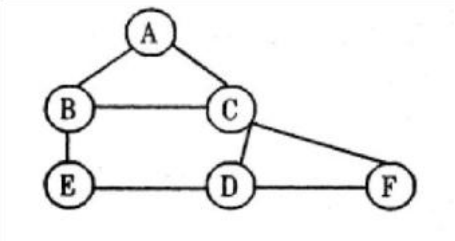
In this case, we get three new states: In(Sibiu),In(Timisoara), and In(Zerind). Now we must choose which of these three possibilities to consider further.

Continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded.

The choice of which state to expand is determined by the **search strategy**

Tree Search algorithm

Task : Find a path to reach F from A



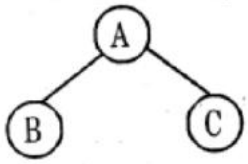
1. Start the sequence with the initial state and check whether it is a goal state or not.

a, If it is a goal state return success.

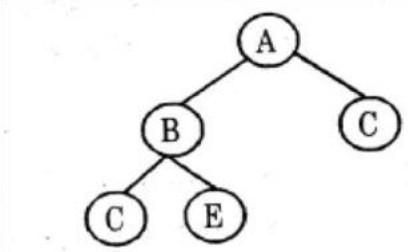
b. Otherwise perform the following sequence of steps

From the initial state (current state) generate and expand the new set of states. The collection of nodes that have been generated but not expanded is called as fringe. Each element of the fringe is a leaf node, a node with no successors in the tree.

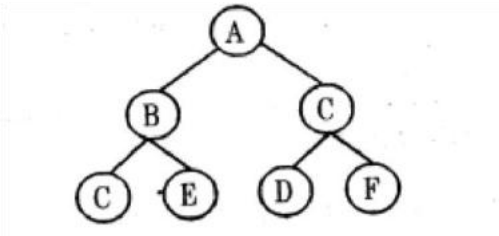
Expanding A



Expanding B



Expanding C



Sequence of steps to reach the goal state F from (A = A - C - F)

2. Search strategy: In the above example we did the sequence of choosing, testing and expanding until a solution is found or until there are no more states to be expanded. The choice of which state to expand first is determined by search strategy.

3. Search tree: The tree which is constructed for the search process over the state space.

4. Search node: The root of the search tree that is the initial state of the problem.

The general tree search algorithm

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

There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

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

The collection of nodes represented in the search tree is defined using set or queue representation.

Set : The search strategy would be a function that selects the next node to be expanded from the set

Queue: Collection of nodes are represented, using queue. The queue operations are defined as:

MAKE-QUEUE(elements) - creates a queue with the given elements
EMPTY(queue)-returns true only if there are no more elements in the queue. **REM0VE-FIRST(queue)** - removes the element at the front of the queue and returns it
INSERT ALL (elements, queue) - inserts set of elements into the queue and returns the resulting queue.
FIRST (queue) - returns the first element of the queue.
INSERT (element, queue) - inserts an element into the queue and returns the resulting queue

The general tree search algorithm with queue representation

function *TREE-SEARCH(problem,fringe)* **returns** a solution, or failure
fringe <- INSERT(MAKE-*NODE(INITIAL-STATE[problem]), fringe)*
loop do
if EMPTY?*(fringe)* **then return** failure *node* <- *REMOVE-FIRST(fringe)*
ifGOAL-TEST*[problenl]*applied to *STATE[node]* succeeds **then return** *SOLUTION(node) fringe* <- INSERT-*ALL(EXPAND(node, problem),fringe)*

function *EXPAND(node, problem)* returns a set of nodes *successors* <- the empty set

for each *<action, result>* in *SUCCESSOR-FN [problem](STATE[node])**do***

S <- a new NODE

STATE[s] <- *result*

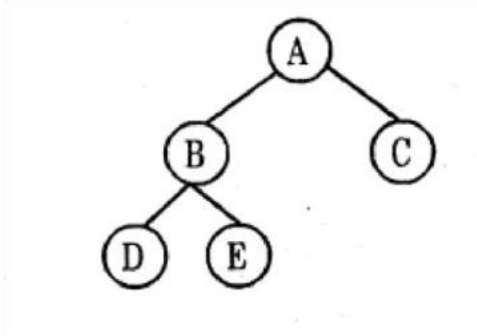
PARENT-NODE[s] <- *node*

ACTION[s] <- *action*

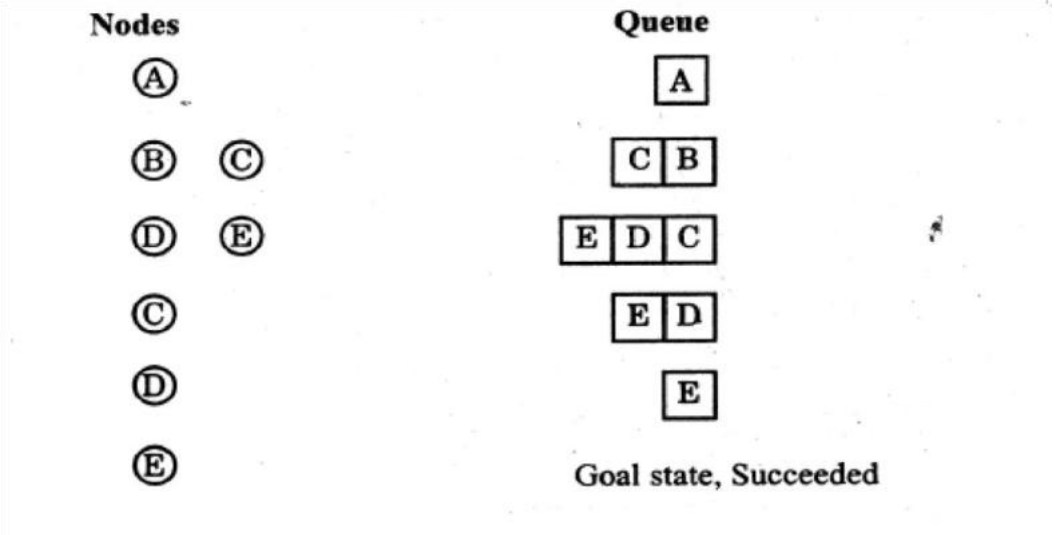
PATH-COST[*s]* <- PATH-COST[*node*]+STEP-COST(*node,action,s)*

DEPTH[s] <- DEPTH[node] + 1 add *s* to *successors* **return** *successors*

Example: Route finding problem



Task. : Find a path to reach E using Queuing function in general tree search algorithm



Measuring problem solving performance

The search strategy algorithms are evaluated depends on four important criteria's. They are:

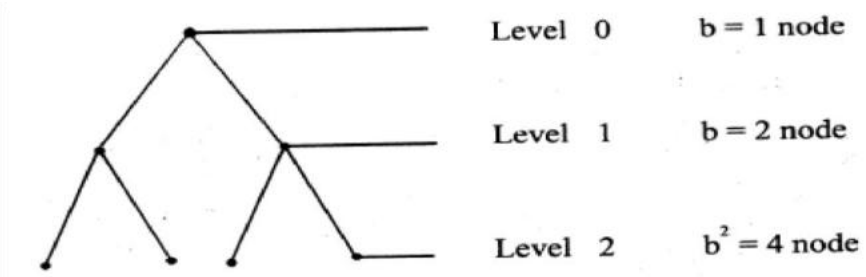
(i) Completeness : The strategy guaranteed to find a solution when there is one.

(ii) Time complexity : Time taken to run a solution

(iii) Space complexity : Memory needed to perform the search.

(iv) Optimality : If more than one way exists to derive the solution then the best one is Selected

Definition of branching factor (b): The number of nodes which is connected to each of the node in the search tree. Branching factor is used to find space and time complexity of the search strategy



Solving Problems by Searching

The searching algorithms are divided into two categories

1. Uninformed Search Algorithms (Blind Search) 2. Informed Search Algorithms (Heuristic Search)

There are six Uninformed Search Algorithms

1. Breadth First Search 2. Uniform-cost search 3. Depth-first search 4. Depth-limited search 5. Iterative deepening depth-first search 6. Bidirectional Search

There are three Informed Search Algorithms

1. **Best First Search 2. Greedy Search 3. A* Search Blind search Vs Heuristic search .**

Blind search	**Heuristic search**
No information about the number of steps (or) path cost from current state to goal state	The path cost from the current state to the goal state is calculated, to select the minimum path cost as the next state.
Less effective in search method	More effective in search method
Problem to be solved with the given information	Additional information can be added as assumption to solve the problem

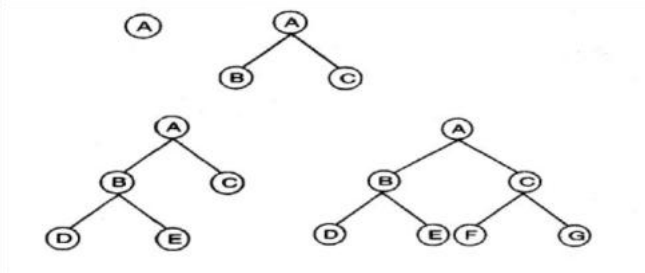
Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their 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.

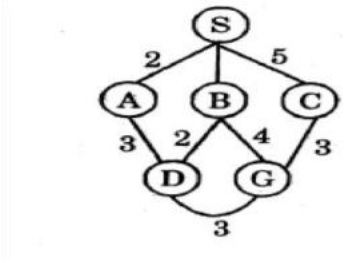
Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first.

In other words, calling **TREE-SEARCH(Problem, FIFO-QUEUE())**results in a 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

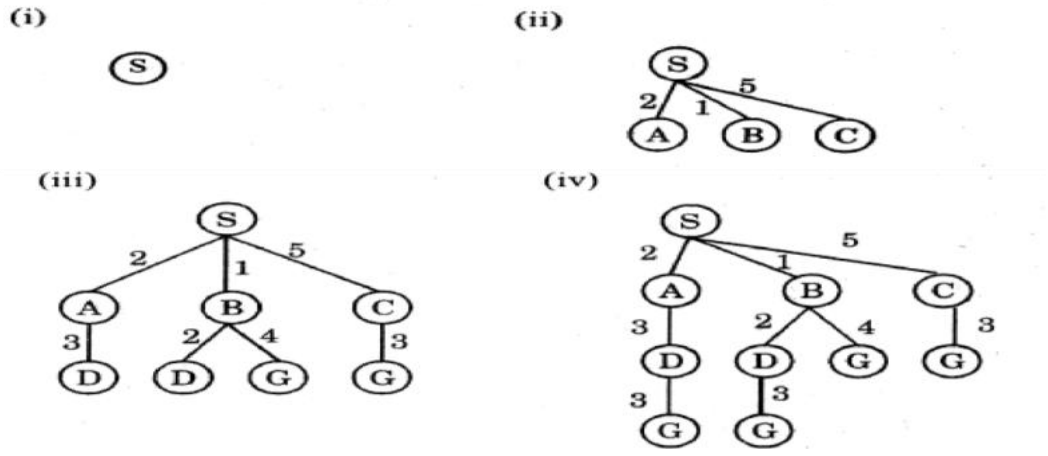
Breadth first search trees after node expansions



Example: Route finding problem



Task: Find a ,path from. S to G using BFS



The path in the 2nd depth level is selected, (i.e) SBG{or) SCG.

Algorithm :

```
function BREADTH-FIRST-SEARCH(problem)
returns a solution, or failure node ←a node with
STATE = problem.INITIAL-STATE, PATH-COST = 0 if
problem.GOAL-TEST(node.STATE) then return
SOLUTION(node) frontier ←a FIFO queue with node
as the only element explored ←an empty set loop
do
if EMPTY?( frontier ) then return failure
node←POP( frontier ) /* chooses the shallowest
node in frontier */ add node.STATE to explored
for each action in problem.ACTIONS(node.STATE)
do child ←CHILD-NODE(problem, node, action) if
child.STATE is not in explored or frontier then
if problem.GOAL-TEST(child.STATE) then return
SOLUTION(child) frontier ←INSERT(child, frontier )
```

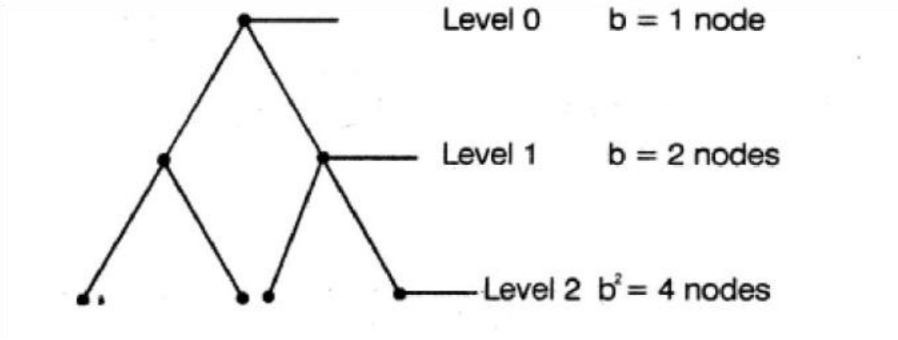
Time and space complexity:

Example:

Time complexity

$= 1 + b + b^2 + \ldots\ldots + b^d$

$= \mathbf{O(b^d)}$



The **space complexity** is same as time complexity because all the leaf nodes of the tree must be maintained in memory at the same time = **$O(b^d)$**

Completeness: Yes

Optimality: Yes, provided the path cost is a non decreasing function of the depth of the node

Advantage: Guaranteed to find the single solution at the shallowest depth level

Disadvantage: Suitable for only smallest instances problem (i.e.) (number of levels to be minimum (or) branching factor to be minimum) ')

Uniform-cost search

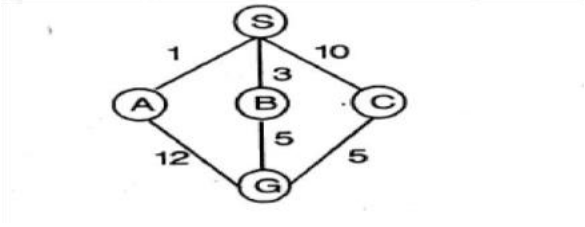
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or
failure node ←a node with STATE = problem.INITIAL-STATE, PATH-
COST = 0 frontier ←a priority queue ordered by PATH-COST, with node
as the only element explored ←an empty set
loop do
if EMPTY?( frontier ) then return failure node←POP( frontier ) /*
chooses the lowest-cost node in frontier */ if problem.GOAL-
TEST(node.STATE) then return SOLUTION(node) add node.STATE to
```

explored for each action in problem.ACTIONS(node.STATE) do child ←CHILD-NODE(problem, node, action) if child.STATE is not in explored or frontier then frontier ←INSERT(child, frontier) else if child.STATE is in frontier with higher PATH-COST then replace that frontier node with child

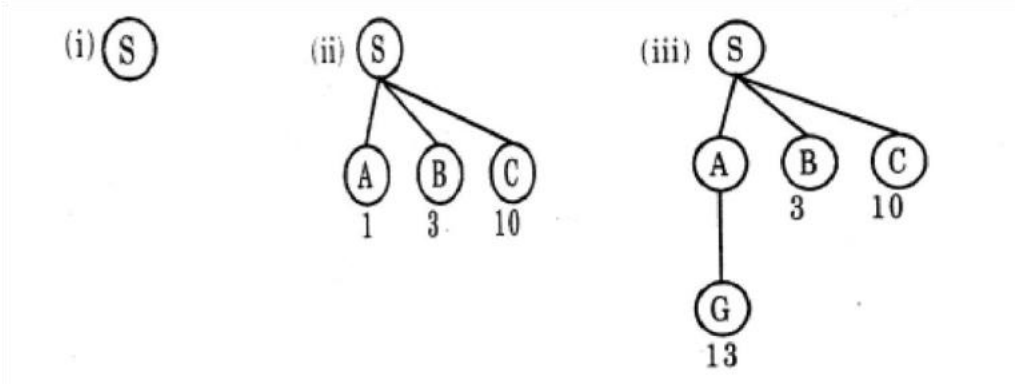
Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost. Note that if all step costs are equal, this is identical to breadth-first search.

Uniform-cost search does not care about the number of steps a path has, but only about their total cost.

Example: Route finding problem

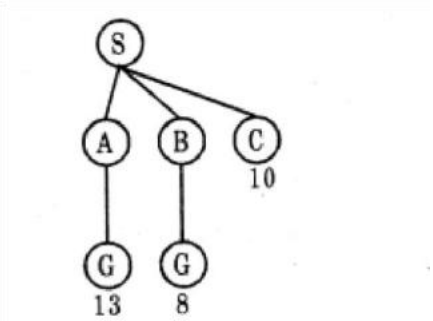


Task : Find a minimum path cost from S to G



Since the value of A is less it is expanded first, but it is not optimal.

B to be expanded next



SBG is the path with minimum path cost.

No need to expand the next path SC, because its path cost is high to reach C from S, as well as goal state is reached in the previous path with minimum cost.

Time and space complexity:

Time complexity is same as breadth first search because instead of depth level the minimum path cost is considered.

Time complexity: **$O(b^d)$** **Space complexity**: **$O(b^d)$**

Completeness: Yes **Optimality:** Yes

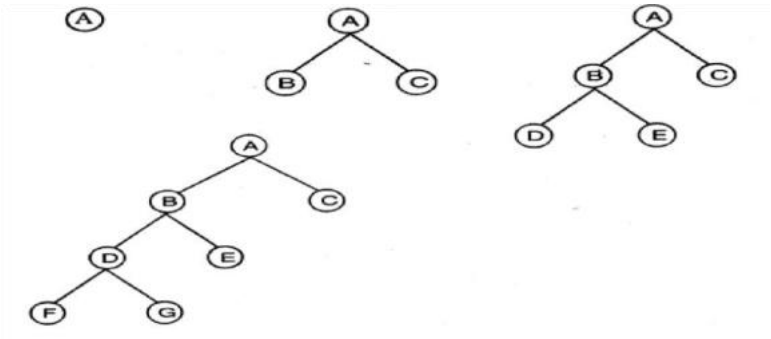
Advantage: Guaranteed to find the single solution at minimum path cost.

Disadvantage: Suitable for only smallest instances problem.

Depth-first search

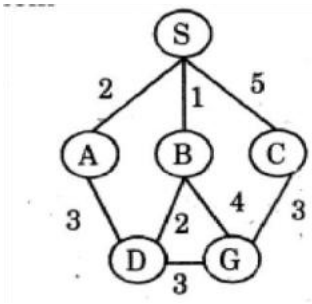
Depth-first search always expands the deepest node in the current fringe of the search tree

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 TREESEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

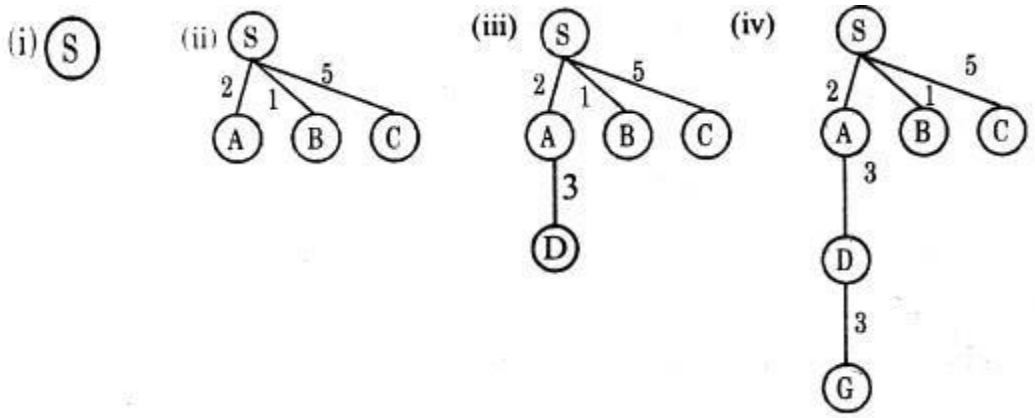


Depth first search tree with 3 level expansion

Example: Route finding problem



Task: Find a path from S to G using DFS



The path in the 3rd depth level is selected. (i.e. S-A-D-G

Algorithm:

function *DFS(problem)* **return** a solution or failure
TREE-SEARCH*(problem,* LIFO-QUEUE())

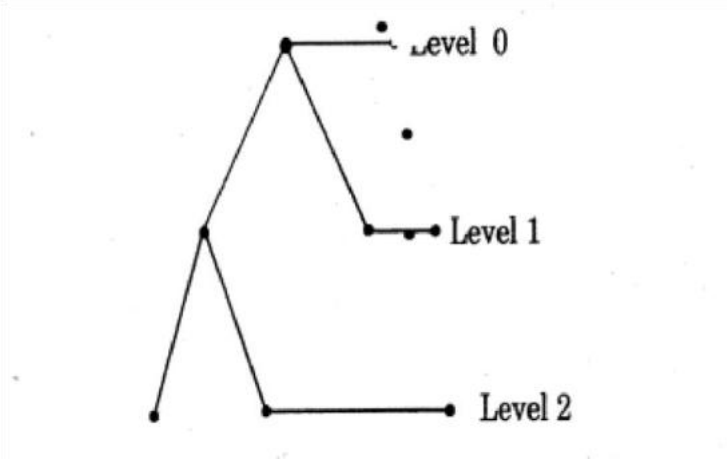
Time and space complexity:

In the worst case depth first search has to expand all the nodes

Time complexity : O(b^m).

The nodes are expanded towards one particular direction requires memory for only that nodes.

Space complexity : O(bm)



b=2

m = 2 :. bm=4

Completeness: No

Optimality: No

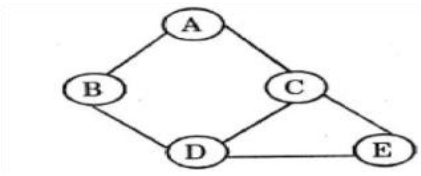
Advantage: If more than one solution exists (or) number of levels is high then DFS is best because exploration is done only in a small portion of the whole space.

Disadvantage: Not guaranteed to find a solution

Depth - limited search

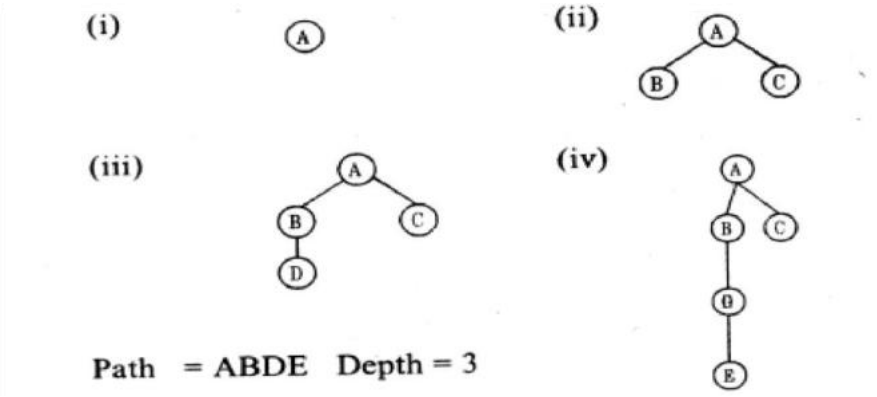
1. Definition: A cut off (maximum level of the depth) is introduced in this search technique to overcome the disadvantage of depth first search. The cutoff value depends on the number of states.

Example: Route finding problem



The number of states in the given map is 5. So, it is possible to get the goal state at a maximum depth of 4. Therefore the cutoff value is 4

Task : Find a path from A to E.



A recursive implementation of depth-limited search

function *DEPTH-LIMITED-SEARCH(problem, limit)* **returns** a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE [*problem*]), *problem, limit)*

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

Time and space complexity:

The worst case time complexity is equivalent to BFS and worst case DFS.

Time complexity : $O(b^l)$

The nodes which is expanded in one particular direction above to be stored.

Space complexity : O(bl)

Optimality: No, because not guaranteed to find the shortest solution first in the search technique.

Completeness : Yes, guaranteed to find the solution if it exists.

Advantage: Cut off level is introduced in the DFS technique

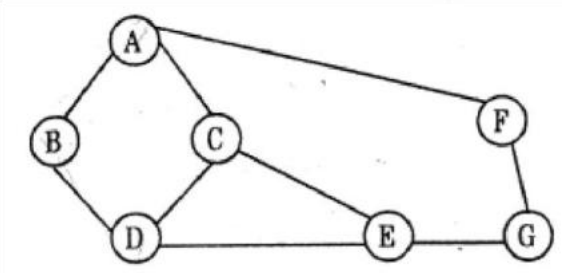
Disadvantage : Not guaranteed to find the optimal solution.

Iterative deepening search

Iterative deepening search

Definition: Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits.

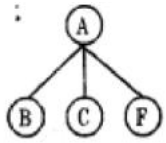
Example: Route finding problem



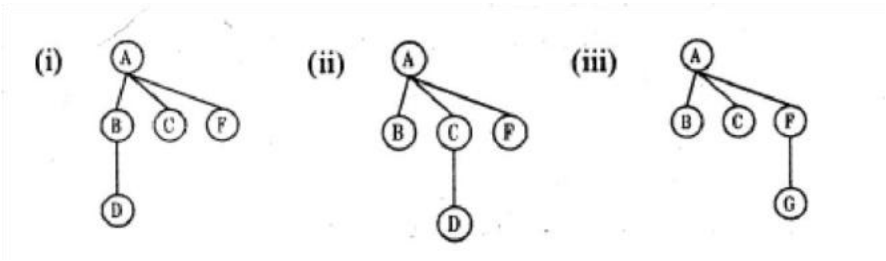
Task: Find a path from A to G

Limit = 0

Limit = 1



Limit = 2



Solution: The goal state G can be reached from A in four ways. They are:

1. A – B – D - E – G ------- Limit 4 2. A - B - D - E - G ------- Limit 4 3. A - C - E - G ------- Limit 3 4. **A - F - G ------ Limit2**

Since it is a iterative deepening search it selects lowest depth limit (i.e.) A-F-G is selected as the solution path.

The iterative deepening search algorithm :

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

Time and space complexity :

Iterative deepening combines the advantage of breadth first search and depth first search (i.e) expansion of states is done as BFS and memory requirement is equivalent to DFS.

Time complexity : $O(b^d)$

Space Complexity : O(bd)

Optimality: Yes, because the order of expansion of states is similar to breadth first search.

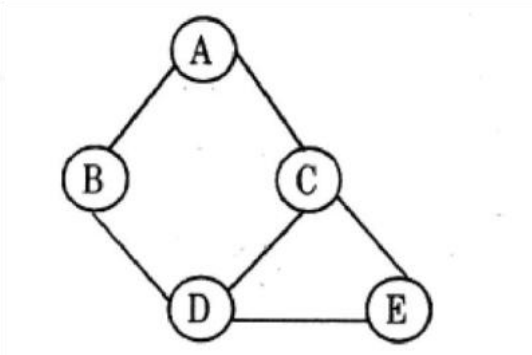
Completeness: yes, guaranteed to find the solution if it exists.
Advantage: This method is preferred for large state space and the depth of the search is not known.

Disadvantage : Many states are expanded multiple times
Example : The state D is expanded twice in limit 2

Bidirectional search

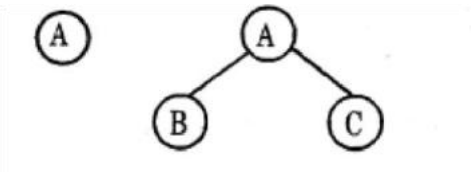
Definition : Bidirectional search is a strategy that simultaneously search both the directions (i.e.) forward from the initial state and backward from the goal, and stops when the two searches meet in the middle.

Example: Route finding problem

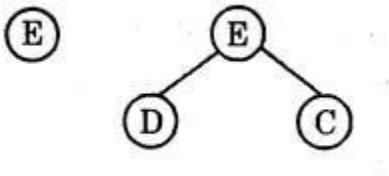


Task : Find a path from A to E.

Search from forward (A) :



Search from backward (E) :



Time and space complexity:

The forward and backward searches done at the same time will lead to the solution in **$O(2b^{d/2}) = O(b^{d/2})$**step, because search is done to go only halfway If the two searches meet at all, the nodes of at least one of them must all be retained in memory requires **$O(b^{d/2})$** space.

Optimality: Yes, because the order of expansion of states is done in both the directions.

Completeness: Yes, guaranteed to find the solution if it exists.

Advantage : Time and space complexity is reduced.

Disadvantage: If two searches (forward, backward) does not meet at all, complexity arises in the search technique. In backward search calculating predecessor is difficult task. If more than one goal state 'exists then explicit, multiple state search is required

Comparing uninformed search strategies

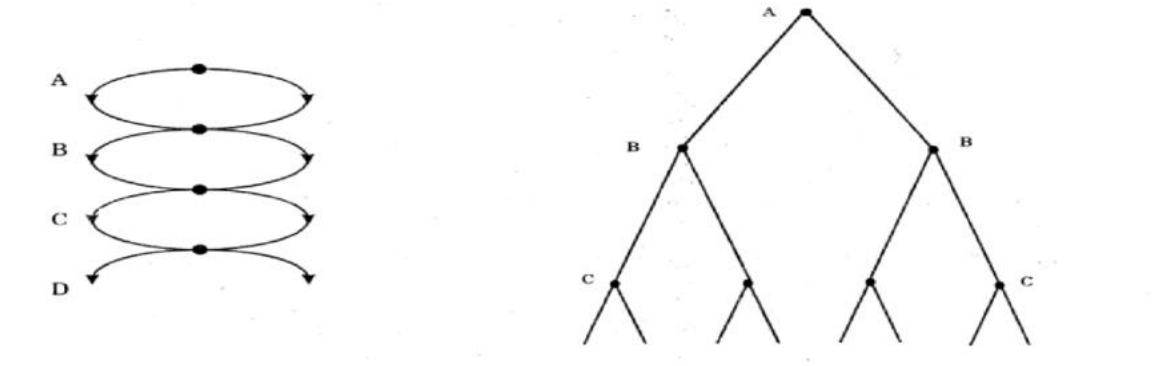
Criterion	**Breadth First**	**Uniform Cost**	**Depth First**	**Depth Limited**	**Iterative Deepening**	**Bidirection**
Complete	**Yes**	**Yes**	**No**	**No**	**Yes**	**Yes**
Time	**$O(b^d)$**	**$O(b^d)$**	**$O(bm)$**	**$O(b^l)$**	**$O(b^d)$**	**$O(b^{d/2})$**
Space	**$O(b^d)$**	**$O(b^d)$**	**$O(bm)$**	**$O(bl)$**	**O(bd)**	**$O(b^{d/2})$**

Optimal	**Yes**	**Yes**	**No**	**No**	**Yes**	**Yes**

Avoiding Repeated States

The most important complication of search strategy is expanding states that have already been encountered and expanded before on some other path

A state space and its exponentially larger search tree



The repeated states can be avoided using three different ways. They are:

1. Do not return to the state you just came from (i.e) avoid any successor that is the same state as the node's parent.
2. Do not create path with cycles (i.e) avoid any successor of a node that is the same as any of the node's ancestors.
3. Do not generate any state that was ever generated before.

The general TREE-SEARCH algorithm is modified with additional data structure, such as :

Closed list - which stores every expanded node.
Open list - fringe of unexpanded nodes.

If the current node matches a node on the closed list, then it is discarded and it is not considered for expansion. This is done with GRAPH-SEARCH algorithm. This algorithm is efficient for problems with many repeated states

```
function GRAPH-SEARCH (problem, fringe) returns a solution, or
failure closed <- an empty set
fringe <- INSERT (MAKE-NODE(INITIAL-STATE[problem]),  fringe)
loop do
if EMPTv?(fringe) then return failure node <- REMOVE-FIKST (fringe)
if GOAL-TEST [problem ](STATE[node]) then return SOLUTION
(node)
if STATE [node] is not in closed then add STATE [node] to closed
fringe <- INSERT-ALL(EXPAND(node, problem), fringe)
```

The worst-case time and space requirements are proportional to the size of the state space, this may be much smaller than $O(b^d)$

Informed search and exploration

Uninformed search strategies can find solutions to problems by systematically generating new states and testing them against the goal. These strategies are inefficient in most cases.

An informed search Strategy uses problem-specific knowledge and it can find solutions more efficiently.

Informed Heuristic Search Strategies

An **informed search** strategy uses problem-specific knowledge beyond the definition of the problem itself and it can find solutions more efficiently than an uninformed strategy.

The general approach is best first search that uses an evaluation function in **TREE-SEARCH or GRAPH-SEARCH**.

Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function,** f(n)

The node with the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal.

Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order of f –values

Implementation of Best-first search using general search algorithm

function BEST-FIRST-*SEARCH(problem, EVAL-FN)* returns a solution sequence **inputs:** *problem,* a problem *EVAL-FN,* an evaluation function QUEUEING -FN<- a function that orders nodes by *EVAL-FN* **return** *TREE-SEARCH(problem,* QUEUEING-FN)

The key component of these algorithms is a heuristic functions denoted h(n)

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

One constraint: if n is a goal node, then h(n) = 0

The two types of evaluation functions are:

(i) Expand the node closest to the goal state using estimated cost as the evaluation is called **greedy best first search.**

(ii) Expand the node on the least cost solution path using estimated cost and actual cost as the evaluation function is called **A*search**

Greedy best first search (Minimize estimated cost to reach a goal)

Definition : A best first search that uses h(n) to select next node to expand is called greedy search

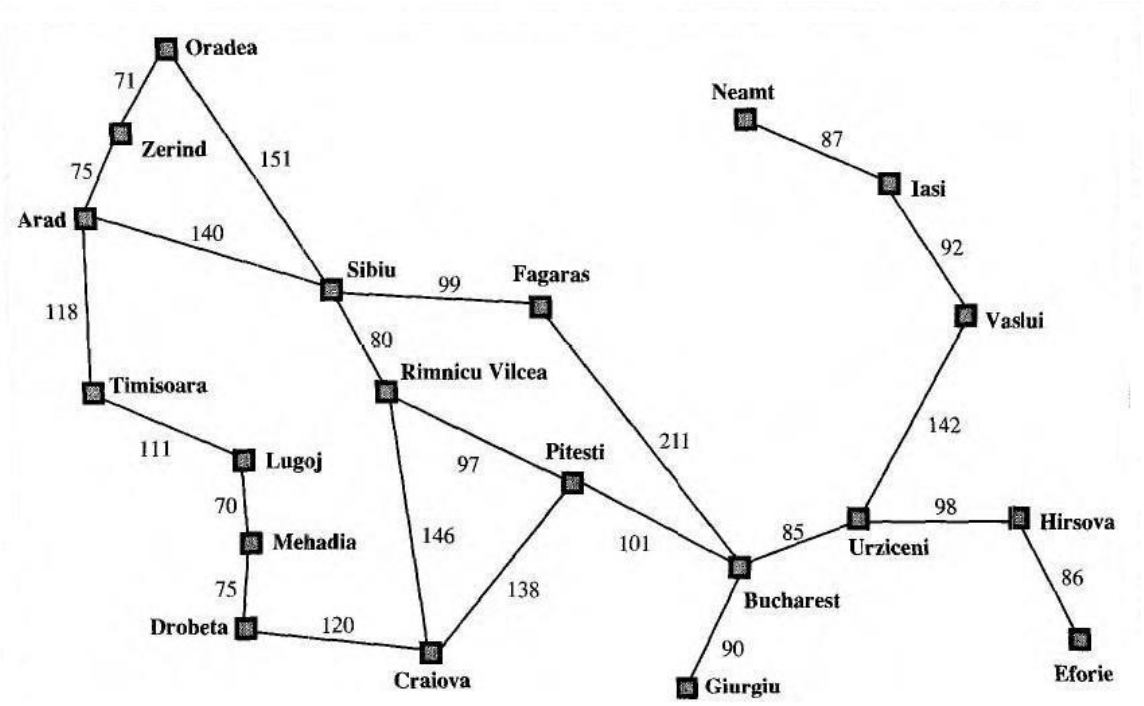
Evaluation function : The estimated cost to reach the goal state, denoted by the letter h(n)

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

Algorithm :

Function GREEDY-BEST-FIRST *SEARCH (problem)* returns a solution or failure **return** BEST-FIRST-SEARCH *(problem,* h)

Example 1 : Route Finding Problem



Problem : Route finding Problem from Arad to Burcharest

Heuristic function : A good heuristic function for route-finding problems is Straight-Line Distance to the goal and it is denoted as $h_{SLD}(n)$.

$h_{SLD}(n)$ = Straight-Line distance between n and the goal locatation

Note : The values of $h_{SLD}(n)$ cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience

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

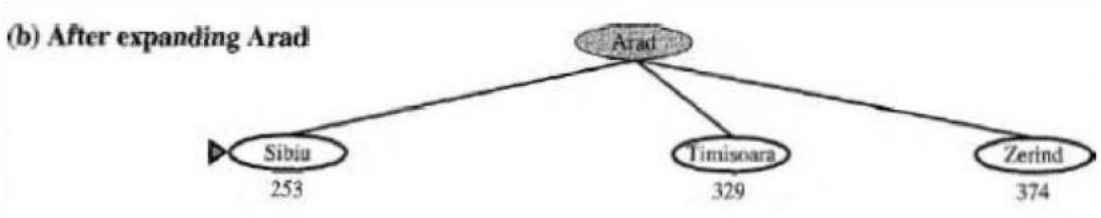
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

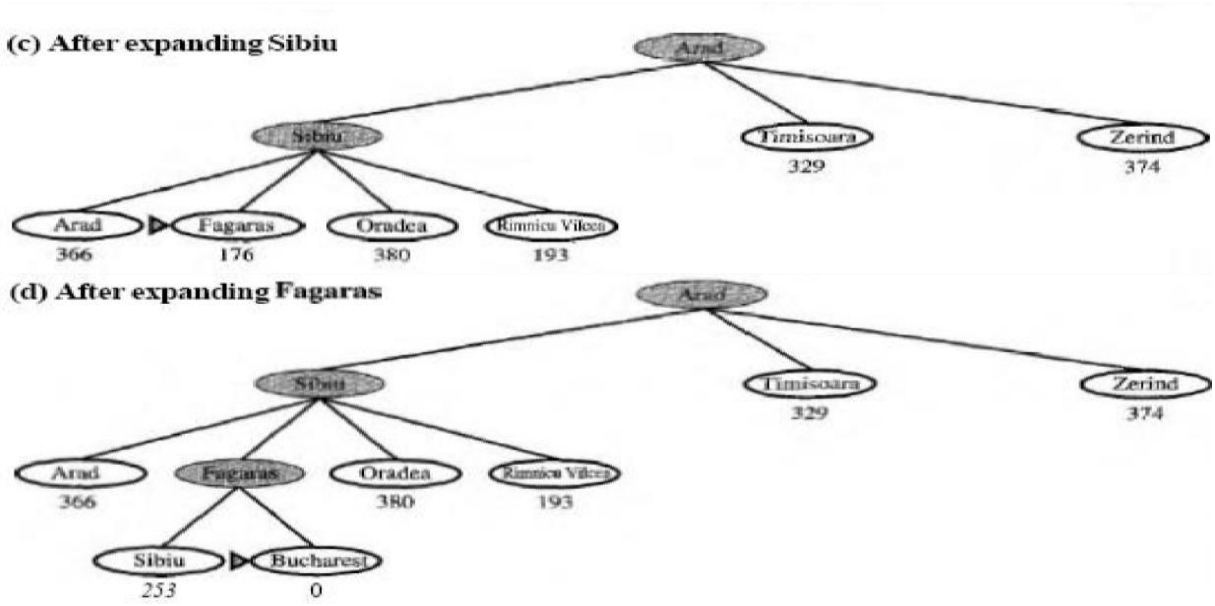
Solution :

From the given graph and estimated cost, the goal state is identified as B u c h a r e s t from Arad. Apply the evaluation function h (n) to find a path from Arad to Burcharest from A to B

(a) The initial state

(b) After expanding Arad





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.

For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called "greedy'-at each step it tries to get as close to the goal as it can.

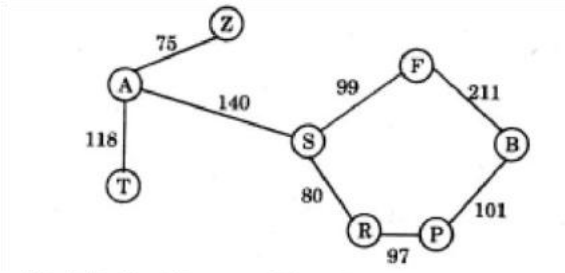
Minimizing h(n) is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, because it is closest to Fagaras, but it is a dead end. The solution is to go first to Vaslui-a step that is actually farther from the goal according to the heuristic-and then to continue to Urziceni, Bucharest, and Fagaras.

Time and space complexity : Greedy search resembles depth first search, since it follows one path to the goal state, backtracking occurs when it finds a dead end. The worst case time complexity is equivalent to depth first search, that is $O(b^m)$, where m is the maximum depth of the search space. The greedy search retains all nodes in memory, therefore the space complexity is also $O(b^m)$ The time and space complexity can be reduced with good heuristic function.

Optimality : It is not optimal, because the next level node for expansion is selected only depends on the estimated cost and not the actual cost.

Completeness : No, because it can start down with an infinite path and never return to try other possibilities.

Example 2 : Finding the path from one node to another node



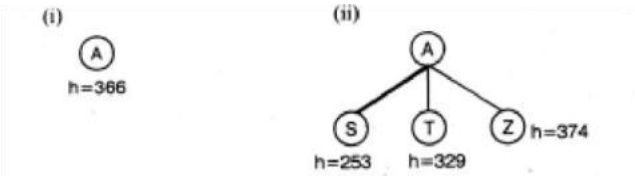
Straight - line distance to B from A:

A - 366
B - 0
F - 178
P - 98
R - 193
S - 253
T - 329
Z - 374

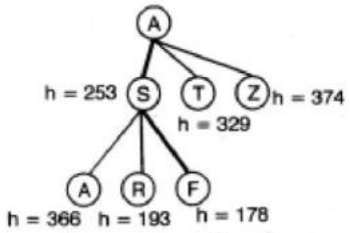
Solution :

From the given graph and estimated cost, the goal state IS identified as B from A.

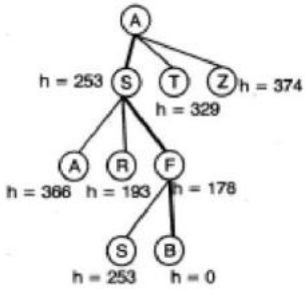
Apply the evaluation function h(n) to find a path from A to B



(iii) S is selected for next level of expansion, since *h(n)* is minimum from S, when comparing to T and Z



(iv) F is selected for next level of expansion, since *h(n)* is minimum from F.



From F, goal state B is reached. Therefore the path from A to Busing greedy search is A - S - F - B = 450 (i.e) (140 + 99 + 211)

A* search (Minimizing the total estimated solution cost)

The most widely-known form of best-first search is called **A*** search (pronounced "A-star search"). **A*** search is both complete and optimal.

It evaluates nodes by combining g(n), the cost to reach the node, and h(n.),the cost to get from the node to the goal

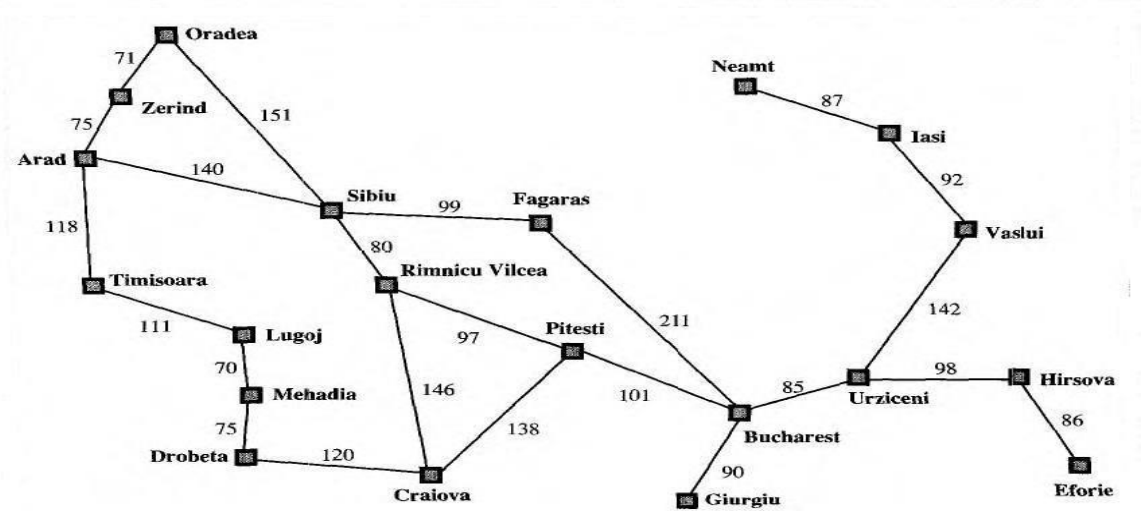
f(n) =g(n) + h(n)

g(n) - path cost from the start node to node n h(n) - estimated cost of the cheapest path from n to the goal f (n) - estimated cost of the cheapest solution through n

A* Algorithm

function A* *SEARCH(problem)* returns a solution or failure **return** BEST-FIRST-SEARCH *(problem,* g+h)

Example 1 : Route Finding Problem



Problem : Route finding Problem from Arad to Burcharest

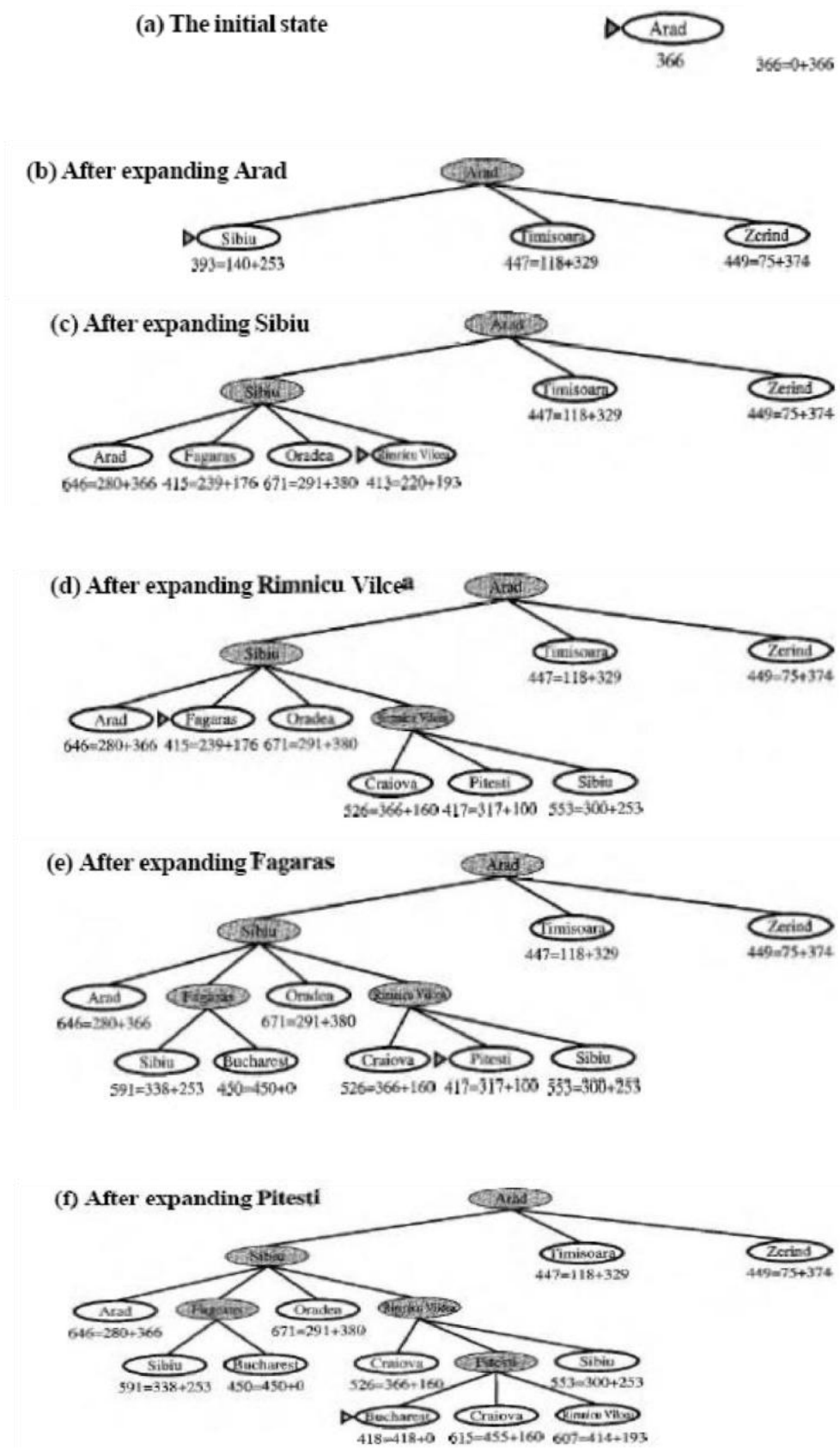
Heuristic function : A good heuristic function for route-finding problems is Straight-Line Distance to the goal and it is denoted as $h_{SLD}(n)$.

$h_{SLD}(n)$ = Straight-Line distance between n and the goal locatation

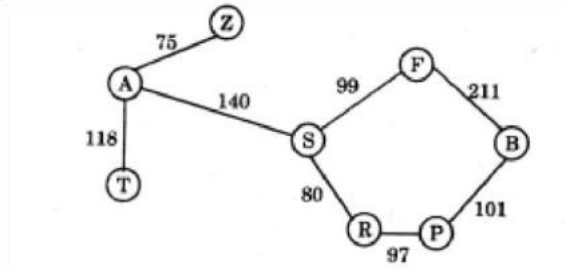
Values of h_{SLD}-straight-line distances to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Stages in an A* search for Bucharest. Nodes are labeled with f (n) = g (n) + h(n)



Example 2 : Finding the path from one node to another node

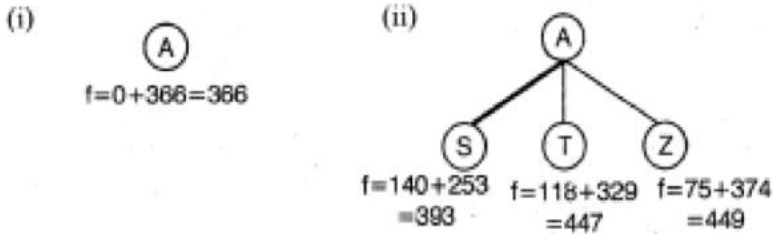


Straight – line distance to B from A:

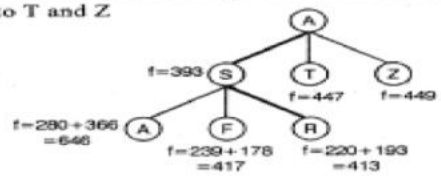
A - 366
B - 0
F - 178
P - 98
R - 193
S - 253
T - 329
Z - 374

Solution:

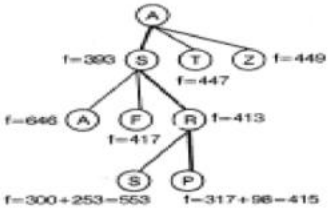
From the given graph and estimated cost, the goal state is identified as B from A Apply the evaluation function f(n) = g(n) +h(n) to find a path from A to B



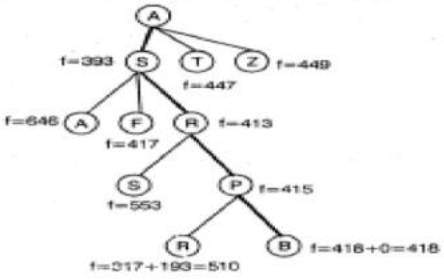
(iii) S is selected for next level of expansion, since f(S) is minimum from S, when comparing to T and Z



(iv) R is selected for next level of expansion, since f(R) is minimum when comparing to A and F.



(v) P is selected for next level of expansion. since f(P) is minimum.



From P, goal state B is reached. Therefore the path from A to B using A* search is A – S - R - P -B : 418 (ie) {140 + 80 + 97 + 101), that the path cost is less than Greedy search path cost.

Time and space complexity: Time complexity depends on the heuristic function and the admissible heuristic value. Space complexity remains in the exponential order.

The behavior of A* search

Monotonicity (Consistency)

In search tree any path from the root, the f- cost never decreases. This condition is true for almost all admissible heuristics. A heuristic which satisfies this property is called monotonicity(**consistency**).

A heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

If the heuristic is non-monotonic, then we have to make a minor correction that restores monotonicity.

Example for monotonic

Let us consider two nodes n and n', where n is the parent of n'

For example

g(n) = 3 and h(n) = 4. then f(n) = g(n) + h(n) = 7. g(n') = 54 and h(n') = 3. then f(n') = g(n') + h(n') = 8

Example for Non-monotonic

Let us consider two nodes n and n', where n is the parent of n'. For example

g(n) = 3 and h(n) = 4. then f(n) = g(n) + h(n) = 7. g(n') = 4 and h(n') = 2. then f(n') = g(n') + h(n') = 6.

To reach the node n the cost value is 7, from there to reach the node n' the value of cost has to increase as per monotonic property. But the above example does not satisfy this property. So, it is called as non-monotonic heuristic.

How to avoid non-monotonic heuristic?

We have to check each time when we generate anew node, to see if its f-cost is less that its parent's f-cost; if it is we have to use the parent's f- cost instead.

Non-monotonic heuristic can be avoided using path-max equation.

f(n') = max (f{n), g(n') + h(n'))

Optimality

A* search is complete, optimal, and optimally efficient among all algorithms

A* using GRAPH-SEARCH is optimal if h(n) is consistent.

Completeness

A* is complete on locally finite graphs (graphs with a finite branching factor) provided there is some constant d such that every operator costs at least d.

Drawback

A* usually runs out of space because it keeps all generated nodes in memory

Memory bounded heuristic search

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterativedeepening A" (IDA*) algorithm.

The memory requirements of A* is reduced by combining the heuristic function with iterative deepening resulting an IDA* algorithm.

Iterative Deepening A* search (IDA*)

Depth first search is modified to use an f-cost limit rather than a depth limit for IDA* algorithm.

Each iteration in the search expands all the nodes inside the contour for the current f-cost and moves to the next contour with new f - cost.

Space complexity is proportional to the longest path of exploration that is bd is a good estimate of storage requirements

Time complexity depends on the number of different values that the heuristic function can take on

Optimality: yes, because it implies A* search.

Completeness: yes, because it implies A* search.

Disadvantage: It will require more storage space in complex domains (i.e) Each contour will include only one state with the previous contour. To avoid this, we increase the f-cost
limit by a fixed amount □ on each iteration, so that the total number of iteration is proportional to 1/□. Such an algorithm is called □ admissible.

The two recent memory bounded algorithms are:

- Recursive Best First Search (RBfS)
- Memory bounded A* search (MA*)

Recursive Best First Search (RBFS)

A recursive algorithm with best first search technique uses only linear space. It is similar to recursive depth first search with an inclusion (i.e.) keeps track of the f-value of the best alternative path available from any ancestor of the current node.

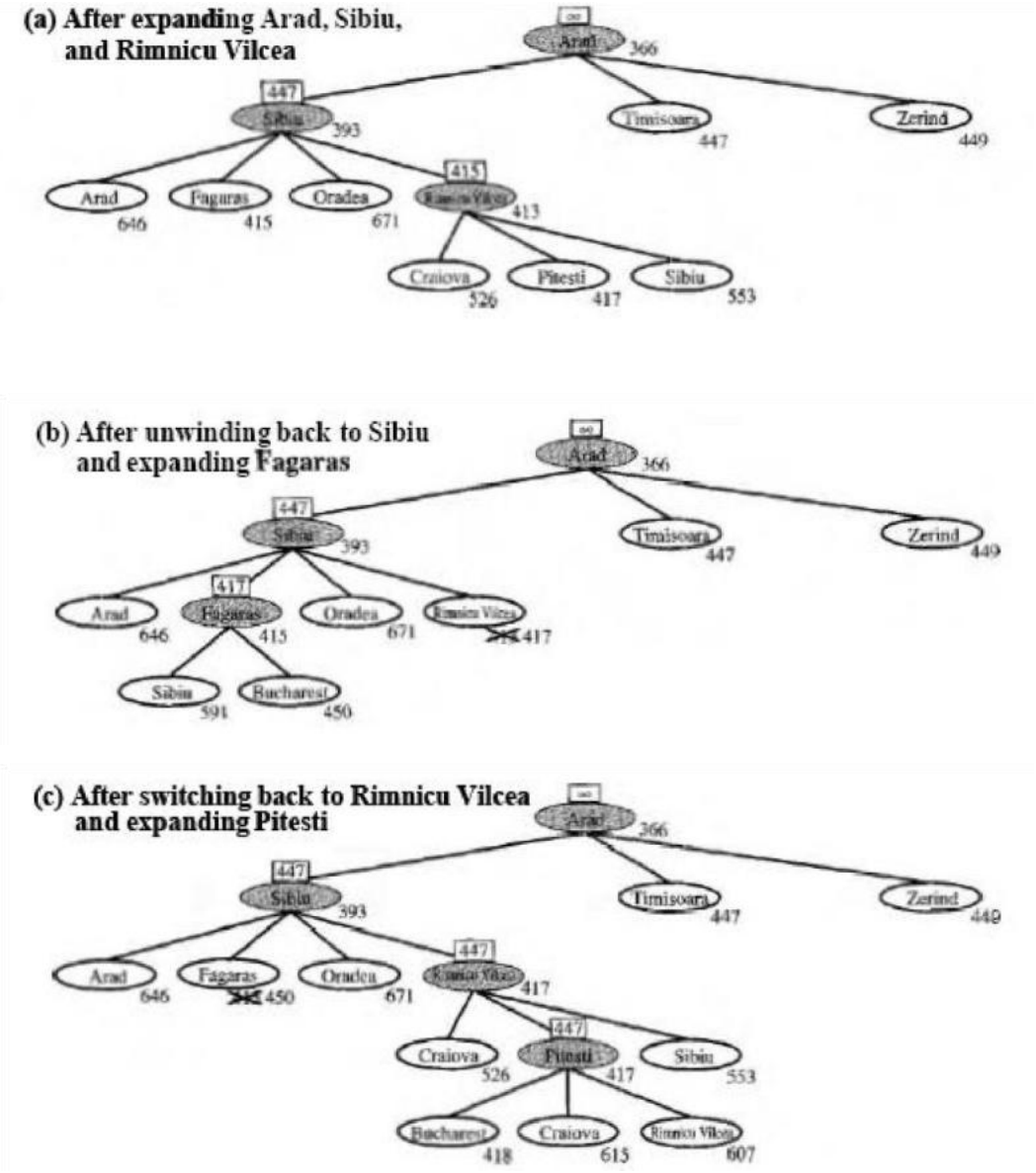
If the current node exceeds this limit, the recursion unwinds back to the alternative path and replaces the f-value of each node along the path with the best f-value of its children.

The main idea lies in to keep track of the second best alternate node (forgotten node) and decides whether it's worth to reexpand the subtree at some later time.

Algortihm For Recursive Best-First Search

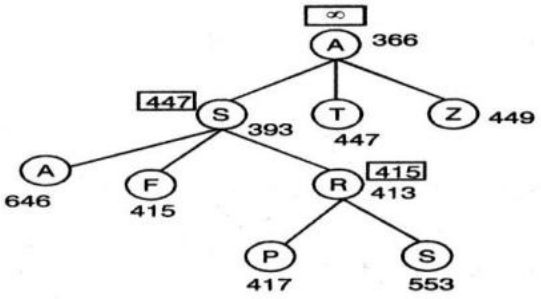
function RECURSIVE-BEST-FIRST-*SEARCH(problem)* returns a solution, or failure
*RBFS(problem,*MAKE-*NODE(INITIAL-STATE[problem]),* □)
function, *RBFS(problem, n*ode, f_*limit)* returns a
solution, or failure and a new *f-cost* limit if GOAL-*TEST[problem](state)* then return n*ode successors* <- EXPAND(node, *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*

<u>Stages in an RBFS search for the shortest route to Bucharest</u>.



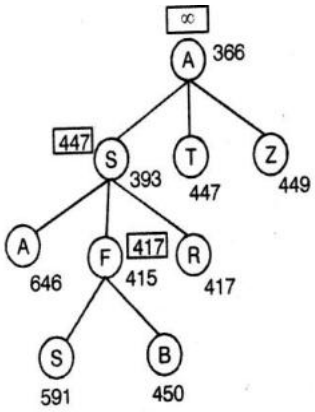
Example:

After expanding A, S, and R, the current best leaf(P) has a value that is worse than the best alternative path (F)



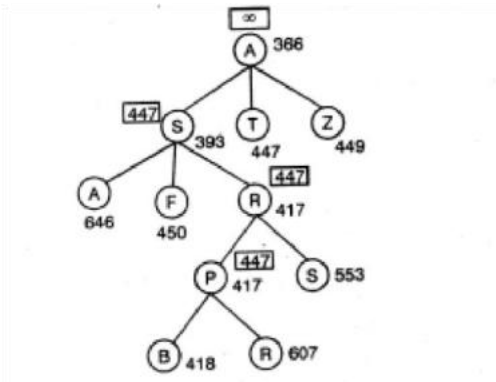
f-limit value of each recursive call is shown on top of each current node. After expanding R, the condition f[best] >f-limit (417 > 415) is true and returns f[best] to that node.

a) After unwinding back to and expanding F



Here the f[best] is 450 and which is greater than the f-limit of 417. Therefore if returns and unwinds with f[best] value to that node.

b) After switching back to Rand expanding P.



The best alternative path through T costs at least 447, therefore the path through R and P is considered as the best one.

Time and space complexity : RBFS is an optimal algorithm if the heuristic function h(n) is admissible. Its time complexity depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. Its space complexity is O(bd), even though more is available.

A search techniques (algorithms) which uses all available memory are:

a. MA* (Memory - bounded A*)
b. SMA* (Simplified MA*)

Simplified Memory - bounded A* search (SMA*)

SMA* algorithm can make use of all available memory to carry out the search.

Properties of SMA* algorithm:

(a) It will utilize whatever memory is made available to it.
(b) It avoids repeated states as far as its memory allows.

It is **complete** if the available memory is sufficient to store the deepest solution path.
It is optimal if enough memory is available to store the deepest solution path. Otherwise, it returns the best solution that can be reached with the available memory.

Advantage: SMA* uses only the available memory.

Disadvantage: If enough memory is not available it leads to unoptimal solution.

Space and Time complexity: depends on the available number of node.

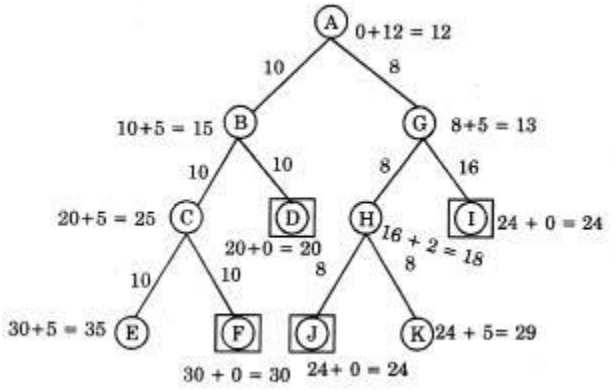
The SMA* Algorithm

function SMA*(*problem*) returns a solution sequence inputs: *problem,* a problem

local variables: *Queue,* a queue of nodes ordered by *f-cost*

Queue<-MAKE-QUEUE(*{MAKE-NODE(INITIAL-STATE[problem])})* loop do
if *Queue* is empty then return failure n<-deepest least-f-cost node in *Queue* if *GOAL-TEST(n)* then return success s<-NEXT-*SUCCESSOR(n)*
if *s* is not a goal and is at maximum depth then f{s)<- ? else
f{s)<- MAX(f(n), g(s)+h(s))
if all of *n's* successors have been generated then update *n's* f-cost and those of its ancestors if necessary if *SUCCESSORS(n)* all in the memory then remove *n* from *Queue*
if memory is full then
delete shallowest, highest-f-cost node in *Queue* remove it from its parent's successor list
insert its parent on *Queue* if necessary insert *s* on *Queue* end

Example:



The values at the nodes are given as per the A* function i.e. g+h=f

From the Figure we identified that the goal states are D,F,J,I because the h value of these nodes are zero (marked as a square)

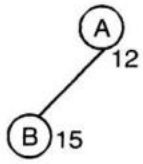
Available memory - 3 nodes of storage space.

Task: Find a optimal path from A to anyone of the goal state.

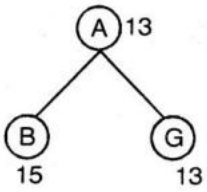
Solution:

(i) 

Initial state

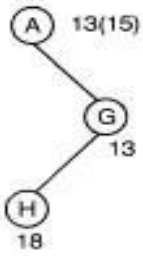
(ii) 

One successor to be added at each stage, therefore B is added to A.

(iii) 

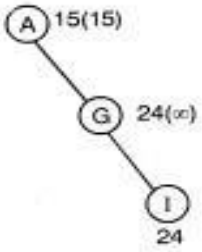
Now, all the children of A is expanded, update the f – cost of root node to the minimum of its children, that is f(A) is 13, the memory is now full.

(iv)



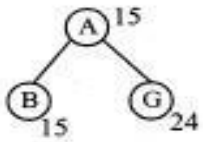
Now, expand the node G to its next successor, but already the memory is full. B is discarded from the queue and it is added as a forgotten descendent of A, which is shown in the parenthesis. The node H is added to G, with $f(H)=18$ but H is not a goal state and it uses all the available memory. Hence there is no way to find a solution through H, make $f(H) = \infty$

(v)



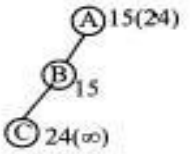
G is expanded again. We drop H and add I (i.e) $f(I) = 24$. The value of G is updated with minimum of H and I and the forgotten value in the parenthesis (i.e) $f(G) = 24$ (∞) Again the value of A is also updated with minimum of $f(B) = 15$ and $f(G) = 24$ and the forgotten value in the parenthesis (i.e) $f(A) = 15(15)$. Notice that I is a goal node, but it might not be the best solution because A's f–cost is only 15, it leads a path in another direction.

(vi)



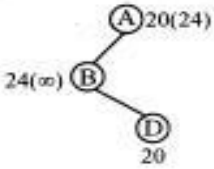
The path from the root (A) leads to B because $f(A) = 15$ in the previous step. Therefore B is generated for the second time.

(vii)



From B, C is generated, which is a non goal state (i.e) $f(C) = \infty$

(viii)



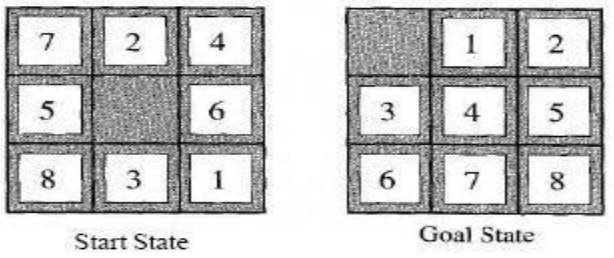
Drop C, add the another successor (D) to B (i.e) $f(D) = 20$, and this value is inherited by B and A.

Now, the deepest, lowest f - cost node is D and it is a goal state also, the search terminates with D as the goal state path=A–B–D.

HEURISTIC FUNCTIONS

The 8-puzzle was one of the earliest heuristic search problems.

Given :



Task : Find the shortest solution using heuristic function that never over estimates the number of steps to the goal.

Solution : To perform the given task two candidates are required, which are named as h_1 and h_2

h_1 = the number of misplaced tiles.

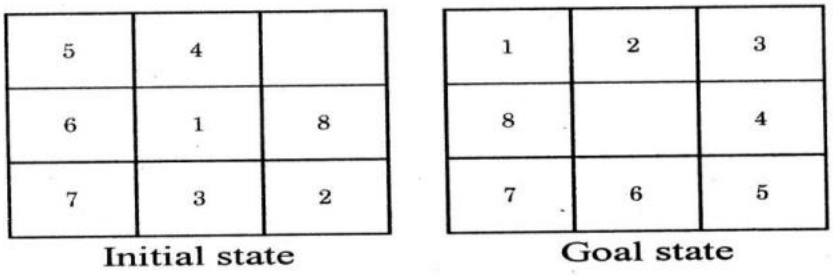
All of the eight tiles are out of position in the above figure, so the start state would have h_l = 8. h_l is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is called as the **city block distance** or **Manhattan distance.** h_2 is also admissible, because any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

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

True solution cost is h1 + h2 = 26

Example :



h_1=7

h_2 = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18

True Solution Cost is $h_1 + h_2$ = 25

Effective branching factor(b*)

In the search tree, if the total number of nodes expanded 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 N nodes. Thus:

$N= l+ b^* + (b^*)^2 + (b^*)^3 + +(b^*)^d$

Example:

For example, if **A*** finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

Depth = 5

N = 52

Effective branching factor is 1.92.

Relaxed problem

A problem with less restriction on the operators is called a relaxed problem. If the given problem is a relaxed problem then it is possible to produce good heuristic function. **Example:** 8 puzzle problem, with minimum number of operators.

Local Search Algorithms And Optimization Problems

In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.
The best state is identified from the objective function or heuristic cost function. In such cases, we can use local search algorithms (ie) keep only a single current state, try to improve it instead of the whole search space explored so far

For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.

Local search algorithms operate a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained.

They have two key advantages:

(1) They use very little memory-usually a constant amount; (2) They can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

The local search problem is explained with the state space land scape. A landscape has:

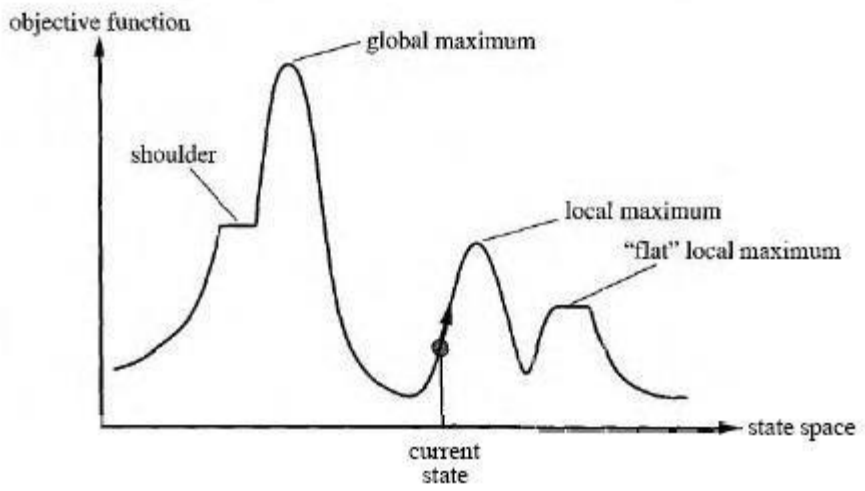
Location - defined by the state

Elevation - defined by the value of the heuristic cost function or objective function, if elevation corresponds to cost then the lowest valley (global minimum) is achieved. If

elevation corresponds to an objective function, then the highest peak (global maximum) is achieved.

A complete local search algorithm always finds a goal if one exists, an optimal algorithm always finds a global minimum/maximum.

A one-dimensional state space landscape in which elevation corresponds to the objective function.



Applications

Integrated - circuit design
Factory - floor layout
Job-shop scheduling
Automatic programming
Vehicle routing
Telecommunications network Optimization

Advantages

- Constant search space. It is suitable for online and offline search
- The search cost is less when compare to informed search
- Reasonable solutions are derived in large or continuous state space for which systematic algorithms are unsuitable.

Some of the local search algorithms are:

1. Hill climbing search (Greedy Local Search)
2. Simulated annealing
3. Local beam search
4. Genetic Algorithm (GA)

Hill Climbing Search (Greedy Local Search)

The **hill-climbing** search algorithm is simply a loop that continually moves in the direction of increasing value. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value. At each step the current node is replaced by the best neighbor;

Hill-**climbing** search algorithm

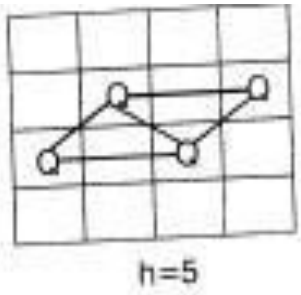
> **function** HILL-CLIMBING(problem) **returns** a state that is a local maximum
> **inputs:** problem, a problem
> **local variables:**current, a node and neighbor, a node current <- **MAKE-NODE(INITIAL-STATE[problem]) loop do**
> neighbor <- a highest-valued successor of current **if** VALUE[neighbor]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

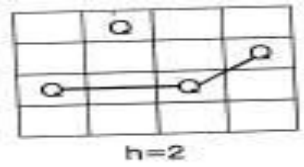
To illustrate hill-climbing, we will use the 8-queens, where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors).

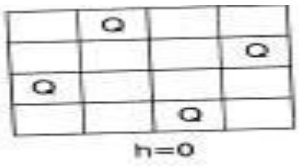
Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.

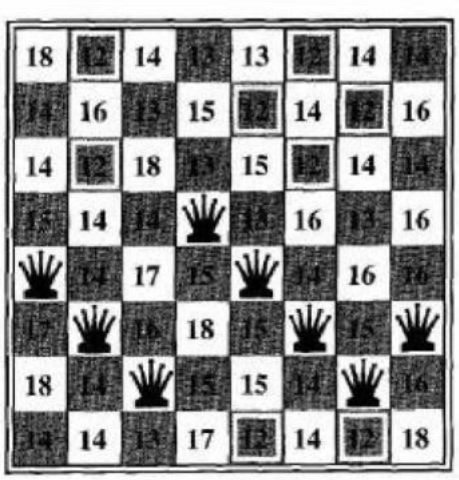
The global minimum of this function is zero, which occurs only at perfect solutions.



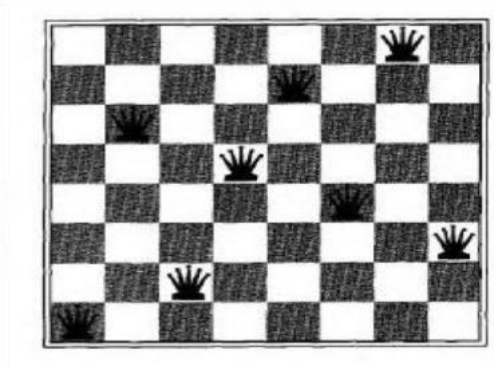




An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked.



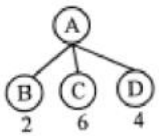
A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.



Hill climbing often gets stuck for the following reasons:

Local maxima or foot hills : a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum

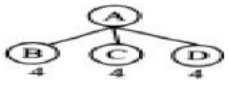
Example :



The evaluation function value is maximum at C and from their there is no path exist for expansion. Therefore C is called as local maxima. To avoid this state, random node is selected using back tracking to the previous node.

Plateau or shoulder: a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum.

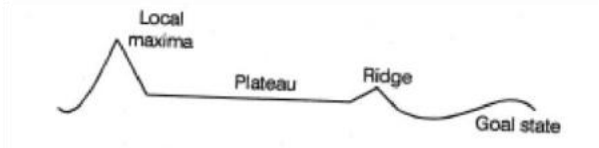
Example :



The evaluation function value of B C D are same, this is a state space of plateau. To avoid this state, random node is selected or skip the level (i.e) select the node in the next level

Ridges: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate. But the disadvantage is more calculations to be done function

Structure of hill climbing drawbacks



Variants of hill-climbing

Stochastic hill climbing - Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.

First-choice hill climbing - First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state

Random-restart hill climbing - Random-restart hill climbing adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial state, stopping when a goal is found.

Simulated annealing search

An algorithm which combines hill climbing with random walk to yield both efficiency and completeness

In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them

When the search stops at the local maxima, we will allow the search to take some down Hill steps to escape the local maxima by allowing some "bad" moves but gradually decrease their size and frequency. The node is selected randomly and it checks whether it is a best move or not. If the move improves the situation, it is executed. ΔE variable is introduced to calculate the probability of worsened. A Second parameter T is introduced to determine the probability.

The simulated annealing search algorithm

function SIMULATED-ANNEALING(problem, schedule) **returns** a solution state
inputs: problem, a problem schedule, a mapping from time to "temperature" **local variables:** current, a node
next, a node
T, a "variable" controlling the probability of downward steps
current <- **MAKE-NODE(INITIAL-STATE[problem]) for t<- l to ∞ do** T <- schedule[t]
if T = 0 **then return** current next <- a randomly selected successor of current
ΔE <- VALUE[next] - VALUE[current] **if** ΔE > 0 **then** current <- next **else** current <- next only with probability $e^{\Delta E/T}$

Property of simulated annealing search

T decreases slowly enough then simulated annealing search will find a global optimum with probability approaching one

Applications

VLSI layout

Airline scheduling

Local beam search

Local beam search is a variation of beam search which is a path based algorithm. It uses K states and generates successors for K states in parallel instead of one state and its successors in sequence. The useful information is passed among the K parallel threads.

The sequence of steps to perform local beam search is given below:

- Keep track of K states rather than just one.
- Start with K randomly generated states.
- At each iteration, all the successors of all K states are generated.
- If anyone is a goal state stop; else select the K best successors from the complete list and repeat.

This search will suffer from lack of diversity among K states.

Therefore a variant named as stochastic beam search selects K successors at random, with the probability of choosing a given successor being an increasing function of its value.

Genetic Algorithms (GA)

A genetic algorithm (or **GA)** is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state

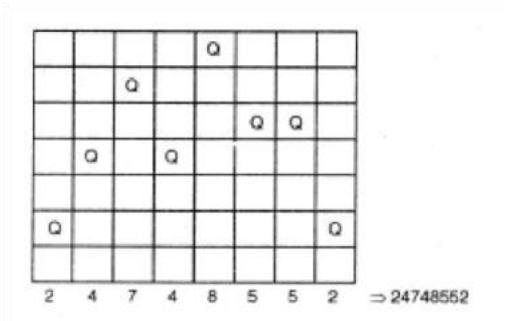
GA begins with a set of k randomly generated states, called the **population.** Each state, or **individual,** is represented as a string over a finite alphabet.

For Example an 8 queen's state could be represented as 8 digits, each in the range from 1 to 8.

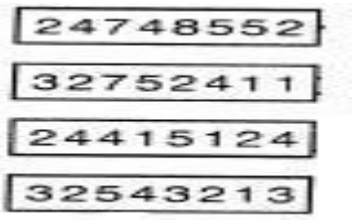
Initial population: K randomly generated states of 8 queen problem

Individual (or) state: Each string in the initial population is individual (or) state. In one state, the position of the queen of each column is represented.

Example: The state with the value 24748552 is derived as follows:



The Initial Population (Four randomly selected States) are :



Evaluation function (or) Fitness function: A function that returns higher values for better State. For 8 queens problem the number of non attacking pairs of queens is defined as fitness function

Minimum fitness value is 0

Maximum fitness value is : 8*7/2 = 28 for a solution

The values of the four states are 24, 23, 20, and 11.

The probability of being chosen for reproduction is directly proportional to the fitness score, which is denoted as percentage.

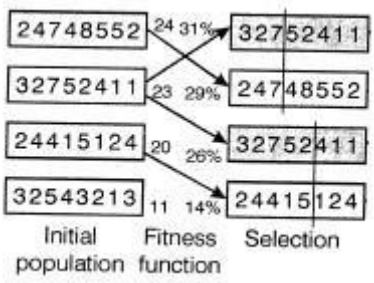
24 / (24+23+20+11) = 31%

23 / (24+23+20+11) = 29%

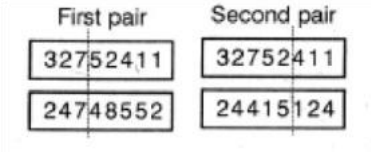
20 / (24+23+20+11) = 26%

11 / (24+23+20+11) = 14%

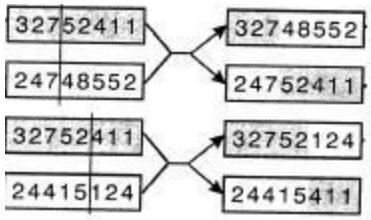
Selection : A random choice of two pairs is selected for reproduction, by considering the probability of fitness function of each state. In the example one state is chosen twice probability of 29%) and the another one state is not chosen (Probability of 14%)



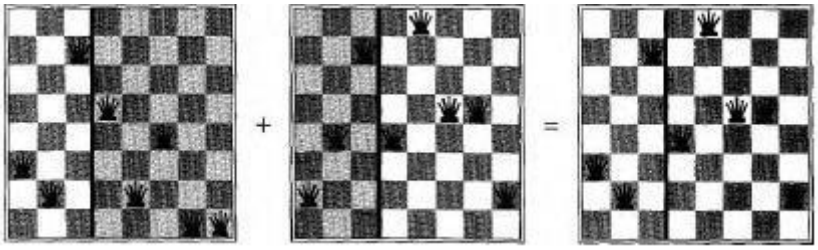
Cross over: Each pair to be mated, a crossover point is randomly chosen. For the first pair the crossover point is chosen after 3 digits and after 5 digits for the second pair.



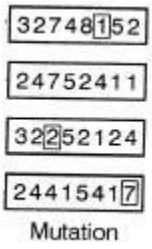
offspring : Offspring is created by crossing over the parent strings in the crossover point. That is, the first child of the first pair gets the first 3 digits from the first parent and the remaining digits from the second parent. Similarly the second child of the first pair gets the first 3 digits from the second parent and the remaining digits from the first parent.



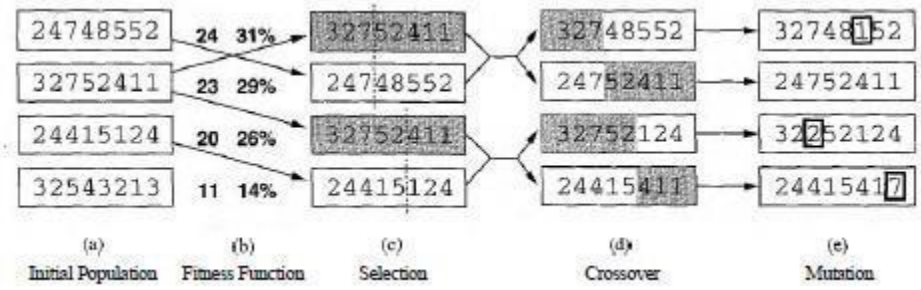
The 8-queens states corresponding to the first two parents



Mutation : Each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring



Production of Next Generation of States



The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in(e).

function GENETIC-ALGORITHM(population, *FITNESS-FN)* **returns** *an* individual
inputs: *population,* a set of individuals
FITNESS-FN, a function that measures the fitness of an individual **repeat**
new-population **<-** empty set
loop for *i* **from 1 to** *SIZE(population) d***o** *x <-* ***RANDOM-SELECTION(Population,****FITNESS-FN) y <-* ***RANDOM-SELECTION(Population,****FITNESS-FN) child <- REPRODUCE(yX),*
if (small random probability) **then** *child MUTATE(chi1d)* add *child* to *new-population*
population <- new-population
until some individual is fit enough, or enough time has elapsed
return the best individual in *population,* according to *FITNESS-FN*
function *REPRODUCE(x,y),* **returns** an individual **inputs:** *x, y,* parent individuals *n <- LENGTH(x)*
c <- random number from 1 to *n*
return *APPEND(SUBSTRING(x,1,c),SUBSTRING(y, c + 1, n))*

The sequence of steps to perform GA is summarized below:

- A successor state is generated by combining two parent states.
- Start with K randomly generated states population
- Each state or individual is represented as a string over a finite alphabet (often a string of O's and 1's)
- Evaluation function (fitness function) is applied to find better states with higher values.
- Produce the next generation of states by selection, crossover and mutation

Local Search In Continuous Spaces

Local Search in continuous space is the one that deals with the real world problems.

- One way to solve continuous problems is to discretize the neighborhood of each state.

- Stochastic hill climbing and simulated annealing are applied directly in the continuous space
- Steepest - ascent hill climbing is applied by updating the formula of current state

$x \leftarrow x + \alpha \nabla f(x)$

α- small constant

$\nabla f(x)$ - magnitude & direction of the steepest slope.

- Empirical gradient, line search, Newton-Raphson method can be applied in this domain to find the successor state.
- Local search methods in continuous space may also lead to local maxima, ridges and plateau. This situation is avoided by using random restart method.

Online Search Agents and Unknown Environments

Online search agent operates by interleaving computation and action, that is first it takes an action, then it observes the environment and computes the next action, whereas ,the offline search computes complete solution (problem solving agent) before executing the problem solution.

online search agents suits well for the following domains.

- ❖ Dynamic or Semi dynamic domain
- ❖ Stochastic domain

Online search is a necessary idea for an exploration problem, where the states and actions are unknown to the agent. For example, consider a newborn baby for exploration problem and the baby's gradual discovery of how the world works is an online search process

Online search problems

An online search problem can be solved by an agent executing actions rather than by a computational process. The agent knows the following terms to do the search in the given environment

- ACTIONS(S) - which returns a list of actions allowed in state s ;
- c(s, a, s') - The step-cost function known to the agent when it reaches s'
- GOAL-TEST(S).

Searching With Partial Information

When the knowledge of the states or actions is incomplete about the environment, then only partial information is known to the agent. This incompleteness lead to three distinct problem types. They are:

(i) **Sensorless problems (conformant problems) :** If the agent has no sensors at all, then it could be in one of several possible initial states, and each action might therefore lead to one of possible successor states.

(ii) **Contigency problems:** If the environment is partially observable or if actions are uncertain, then the agent's percepts provide new information after each action. A problem is called adversarial if the uncertainty is caused by the actions of another agent. To handle the situation of unknown circumstances the agent needs a contigency plan.

(iii) **Exploration problem:** It is an extreme case of contigency problems, where the states and actions of the environment are unknown and the agent must act to discover them.

CONSTRAINT SATISFACTION PROBLEMS(CSP)

Constraint satisfaction problems (CSP) are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria. A constraint is a restriction of the feasible solutions in an optimization problem.

Some examples for CSP's are:

The n-queens problem
A crossword puzzle
A map coloring problem
The Boolean satisfiability problem
A cryptarithmetic problem

All these examples and other real life problems like time table scheduling, transport scheduling, floor planning etc. are instances of the same pattern,

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

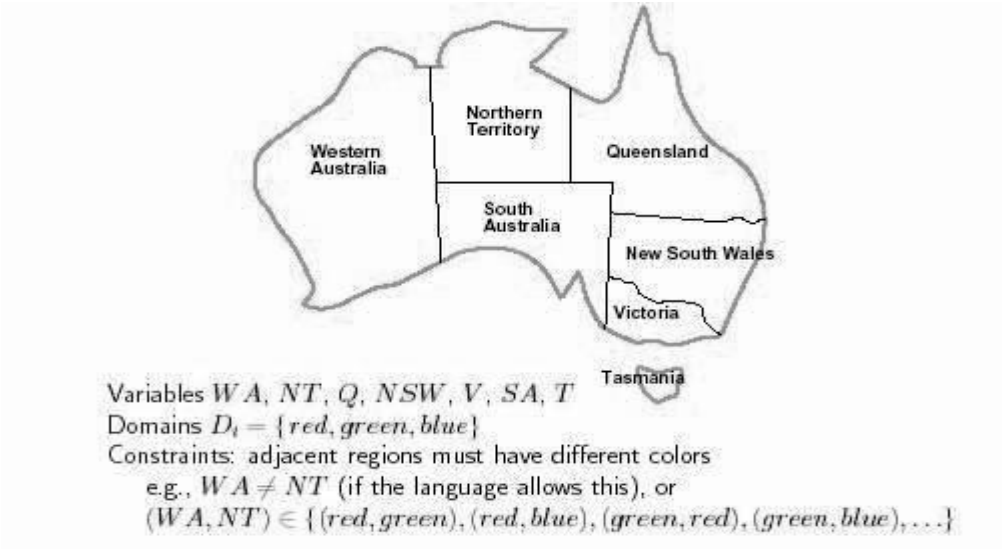
A **State** of the problem is defined by an **assignment** of values to some or all of the variables,$\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.**

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

Example for Constraint Satisfaction Problem :

The map coloring problem. The task of coloring each region red, green or blue in such a way that no neighboring regions have the same color.

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 must not 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)}.

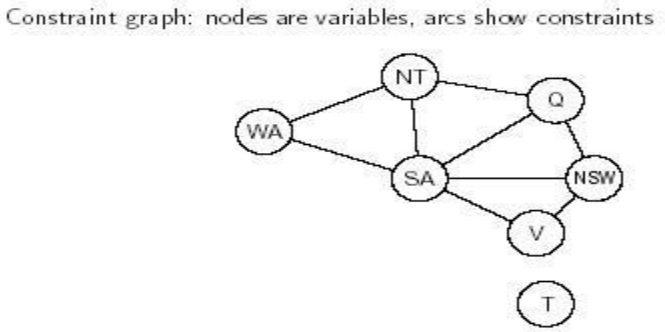
(The constraint can also be represented as the inequality WA □ NT)

There are many possible solutions, such as

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

Constraint Graph : A CSP is usually represented as an undirected graph, called constraint graph where the nodes are the variables and the edges are the binary 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. So Depth first search algorithms are popular for CSPs.

Varieties of CSPs

Discrete variables

Discrete variables can have

- Finite Domains
- Infinite domains

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,.....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. For example, if Jobl, which takes five days, must precede Jobs, then we would need a constraint language of algebraic inequalities such as

$$Startjob_1 + 5 <= Startjob_3.$$

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 :

Unary constraints – Which restricts a single variable.
Example : SA ^ green

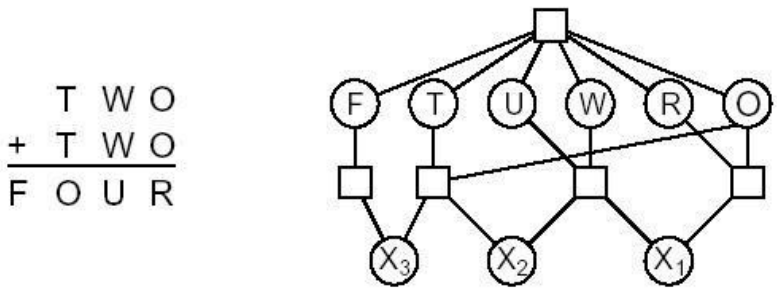
Binary constraints - relates pairs of variables.
Example : SA ^ WA

Higher order constraints involve 3 or more variables.

Example : cryptarithmetic puzzles. 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.

Constraint graph for the cryptarithmetic 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.

Alldiff constraint can be broken down into binary constraints - F ≠ T, F≠ U, and so on.
The addition constraints on the four columns of the puzzle also involve several variables and can be written as

$O + O = R + 10 . X_1$

$X_1 + W + W = U + 10 . X_2$

$X_2 + T + T = O + 10 . X_3$

$X_3 = F$

Where X_1, X_2, and X_3 are **auxiliary variables** representing the digit (0 or 1) carried over into the next column.

Real World CSP's : Real world problems involve read-valued variables,

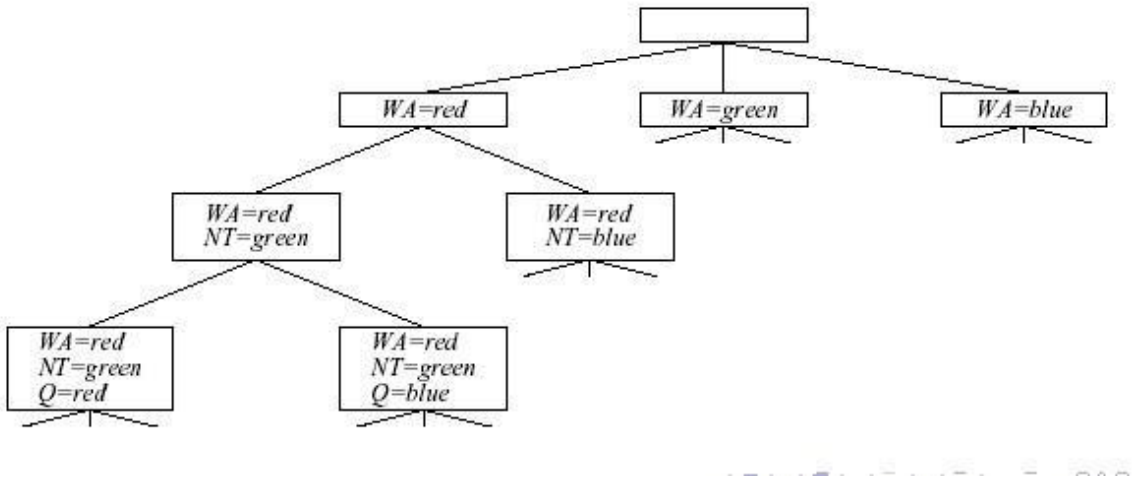
- Assignment problems Example : who teaches what class.
- Timetabling Problems Example : Which class is offered when & where?
- Transportation Scheduling
- Factory Scheduling

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.

```
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
```

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



Improving backtracking efficiency is done with general purpose methods, which can give huge gains in speed.

- Which variable should be assigned next and what order should be tried?
- What are the implications of the current variable assignments for the other unassigned variables?
- Can we detect inevitable failure early?

Variable & value ordering: In the backtracking algorithm each unassigned variable is chosen from minimum Remaining Values (MRV) heuristic, that is choosing the variable with the fewest legal values. It also has been called the "most constrained variable" or "fail-first" heuristic.

If the tie occurs among most constrained variables then most constraining variable is chosen (i.e.) choose the variable with the most constraints on remaining variable. Once a variable has been selected, choose the least constraining value that is the one that rules out the fewest values in the remaining variables.

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-UNASSIGNED-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 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

In 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. NT and SA cannot be blue

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.

Constraint propagation repeatedly enforces constraints locally to detect inconsistencies. This propagation can be done with different types of consistency techniques. They are:

Node consistency (one consistency)

Arc consistency (two consistency)

Path consistency (K-consistency)

Node consistency

- Simplest consistency technique
- The node representing a variable V in constraint graph is node consistent if for every value X in the current domain of V, each unary constraint on V is satisfied.
- The node inconsistency can be eliminated by simply removing those values from the domain D of each variable V that do not satisfy unary constraint on V.

Arc Consistency

The idea of **arc consistency** provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, 'arc' refers to a directed arc in the constraint graph, such as the arc from SA to NSW. Given the current domains of SA and NSW, the arc is consistent if, for every value x of SA, there is some value y of NSW that is consistent with x.

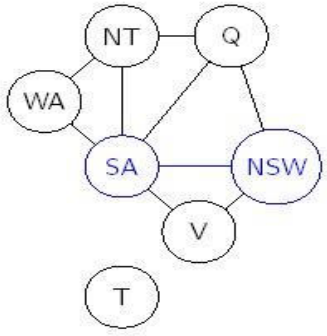


Figure: Australian Territories

In the constraint graph, binary constraint corresponds to arc. Therefore this type of consistency is called arc consistency.

Arc (v_i, v_j) is arc consistent if for every value X the current domain of v_i there is some value Y in the domain of v_j such v_i =X and v_j=Y is permitted by the binary constraint between v_i and v_j

Arc-consistency is directional ie if an arc ($v_{i,}$ v_j) is consistent than it does not automatically mean that ($v_{j,}$ v_i) is also consistent.

An arc ($v_{i,}$ v_j) can be made consistent by simply deleting those values from the domain of D_i for which there is no corresponding value in the domain of Dj such that the binary constraint between V_i and v_j is satisfied - It is an earlier detection of inconstency that is not detected by forward checking method.

The different versions of Arc consistency algorithms are exist such as AC-I, AC2,AC-3, AC-4, AC-S; AC-6 & AC-7, but frequently used are AC-3 or AC-4.

AC - 3 Algorithm

In this algorithm, queue is used to cheek the inconsistent arcs.

When the queue is not empty do the following steps:

- Remove the first arc from the queue and check for consistency.
- If it is inconsistent remove the variable from the domain and add a new arc to the queue
- Repeat the same process until queue is empty

function AC-3(csp) **returns** the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables {X_1, X_2, . . . , X_n}
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty **do**
(Xi, Xj) <- REMOVE-FIRST(queue) **if REMOVE-INCONSISTENT-VALUEXS(x_i,x_j) then for each X_k in** NEIGHBORS[X_j] **do** add (X_k , X_i)to queue

function REMOVE-INCONSISTENT-VALUEXS(x_i,x_j) returns true iff we remove a value removed <-false
for each x **in** DOMAIN[x_i] **do**
if no value y in DOMAIN[x_j] allows (x, y) to satisfy the constraint between X_i and X_j
then delete **x** from DOMAIN[X_i]removed <- true **return** removed

k-Consistency (path Consistency)

A CSP is k-consistent if, for any set of k - 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable

- 1-consistency means that each individual variable by itself is consistent; this is also called node consistency.
- 2-consistency is the same as arc consistency.
- 3-consistency means that any pair of adjacent variables can always be extended to a third neighboring variable; this is also called **path consistency**.

Handling special constraints

Alldiff constraint - All the variables involved must have distinct values.

Example: Crypt arithmetic problem

The inconsistency arises in Alldiff constraint when m>n (i.e.) m variables are involved in the constraint and n possible distinct values are there. It can be avoided by selecting the variable in the constraint that has a singleton. domain and delete the variable's value from the domain of remaining variables, until the singleton variables are-exist. This simple algorithm will resolve the inconsistency of the problem.

Resource constraint (Atmost Constraint) - Higher order constraint or atmost constraint, in which consistency is achieved by deleting the maximum value of any domain if it is not consistent with minimum values; of the other domains.

Intelligent backtracking

Chronological backtracking : When a branch of the search fails, back up to the preceding variable and try a different value for it (i.e.) the most recent decision point is revisited. This will lead to inconsistency in real world problems (map coloring problem) that can't be resolved. To overcome the disadvantage of chronological backtracking, an intelligence backtracking method is proposed.

Conflict directed backtracking: When a branch of the search fails, backtrack to one of the set of variables that caused the failure-conflict set. The conflict set for variable X is the set of previously assigned variables that are connected to X
by constraints. A backtracking algorithm that was conflict sets defined in this way
is called conflict directed backtracking

Local Search for CSPs

- Local search method is effective in solving CSP's, because complete state formulation is defined.
- Initial state - assigns a value to every variable.
- Successor function - works by changing the value of each variable

Advantage : useful for online searching when the problem changes.
Ex : Scheduling problems

The MIN-CONFLICTS algorithm for solving CSPs by local search.

function *MIN-CONFLICTS (CSP, max-steps)* **returns** a solution or failure
inputs: *csp,* a constraint satisfaction problem *max-steps,* the number of steps allowed before giving up *current* <- an initial complete assignment for *csp* **for *i*** = **1** to *max-steps* **do**
if *current* is a solution for *csp* **then return** *current var* <- a randomly chosen, conflicted variable from *VARIABLES[CSP]*
value <- the value *v* for *var* that minimizes
CONFLICTS(var, v, current, csp) set *var* = *value* in *current* **return** *failure*

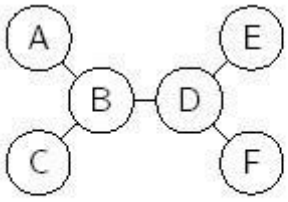
A two-step solution for an &-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square.

The structure of problems

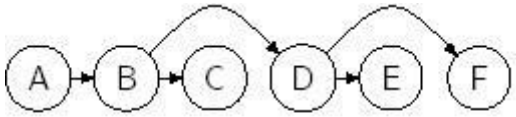
The complexity of solving a CSP-is strongly related to the structure of its constraint graph. If CSP can be divided into independent sub problems, then each sub problem is solved independently then the solutions are combined. When the n variables are divided as n/c subproblems, each will take d^c work to solve. Hence the total work is $O(d^c n/c)$. If n=lO, c=2 then 5 problems are reduced and solved in less time.

Completely independent sub problems are rare, in most cases sub problems of a CSP are connected

The way how to convert the constraint graph of a tree structure CSP into linear ordering of the variables consistent with the tree is shown in Figure. Any two variables are connected by atmost one path in the tree structure



Tree-Structured CSP



Linear ordering

If the constraint graph of a CSP forms a tree structure then it can be solved in linear time number of variables). The algorithm has the following steps.

1. Choose any variable as the root of the tree and order the variables from the root to the leaves in such a way that every node's parent in the tree preceeds it in the ordering label the variables X_l .... X_n in order, every variable except the root has exactly one parent variable.

2. For j from n down 2, apply arc consistency to the arc (X_i , X_j), where X_i is the parent of X_j removing values from Domain [X_i] as necessary.

3. For j from 1 to n, assign any value for Xj consistent with the value assigned for X_i, where X_i is the parent of X_j Keypoints of this algorithm are as follows:

Step-(2), CSP is arc consistent so the assignment of values in step (3) requires no backtracking.

Step-(2), the arc consistency is applied in reverse order to ensure the consistency of arcs that are processed already.

General constraint graphs can be reduced to trees on two ways. They are:

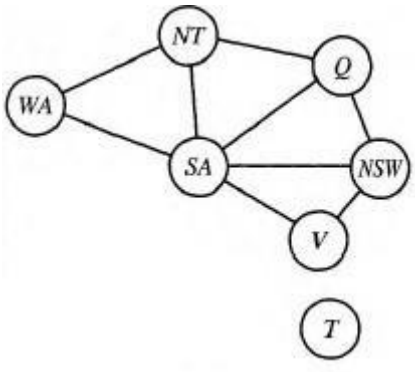
(a) Removing nodes - Cutset conditioning
(b) Collapsing nodes together - Tree decomposition.

(a) Removing nodes - Cutset conditioning

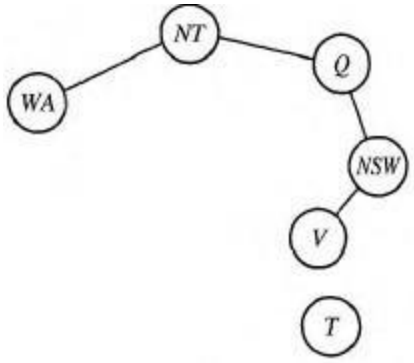
- Assign values to some variables so that the remaining variables form a tree.
- Delete the value assigned variable from the list and from the domains of the other variables any values that are inconsistent with the value chosen for the variable.
- This works for binary CSP's and not suitable for higher order constraints.
- The remaining problem (tree structure) is solved in linear order time variables.

Example: In the constraint graph of map coloring problem, the region SA is assigned with a value and it is removed to make the problem in the form of tree structure, then it is solvable in linear time

The original constraint graph



The constraint graph after the removal of SA



- If the value chosen for the variable to be deleted for tree structure is wrong, then the following algorithm is executed.

(i) Choose a subset S from VARIABLES[CSP] such that the constraint graph becomes **a tree after removal of S-cycle cutset.**

(ii) For each variable on S with assignment satisfies all constraints on S.

* Remove from the deomains of the remaining variables any values that are inconsistent with the assignment for S.
* If the remaining CSP has a solution, return it with the assignment for S.

(b) Collapsing nodes together-Tree decomposition

- Construction of tree decomposition the constraint graph is divided into a set of subproblems, solved independently and the resulting solutions are combined.

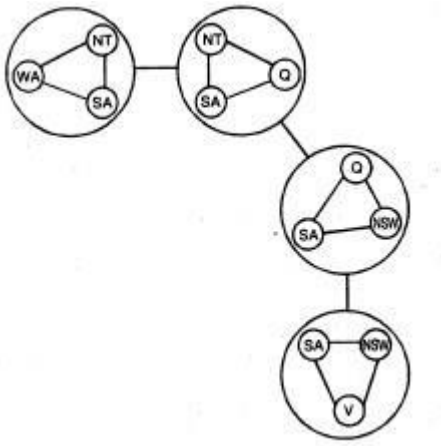
  - Works well, when the subproblem is small.

  - Requirements of this method are:

- Every variable in the base problem should appear in atleast one of the subproblem.
- If the binary constraint is exist, then the same constraint must appear in atleast one of the subproblem. .

- If the variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems, that is the variable should be assigned with same value and constraint in every subproblem.

A tree decompositon of the constraint graph



Solution

- If any subproblem has no solution then the entire problem has no solution.
- If all the sub problems are solvable then a global solution is achieved.

Adversarial Search

Competitive environments, in which the agent's goals are in conflict, give rise to **adversarial search** problems-often known as **games.**

In our terminology, games means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess (+1), the other player necessarily loses (-1).

There are two types of games

1. Perfect Information (Example : chess, checkers) 2. Imperfect Information (Example : Bridge, Backgammon)

In game playing to select the next state, search technique is required. Game playing itself is considered as a type of search problems. But, how to reduce the search time to make on a move from one state to another state.

The **pruning technique** allows us to ignore positions of the search tree that make no difference to the final choice.

Heuristic evaluation function allows us to find the utility (win, loss, and draw) of a state without doing a complete search.

Optimal Decisions in Games

A Game with two players - Max and Min.

- Max, makes a move first by choosing a high value and take turns moving until the game is over
- Min, makes a move as a opponent and tries to minimize the max player score, until the game is over.

At the end of the game (goal state or time), points are awarded to the winner.

The components of game playing are :

Initial state - Which includes the board position and an indication of whose move and identifies the player to the move.

Successor function - which returns a list of (move, state) pairs, each indicating a legal move and the resulting state.

Terminal test - Which determines the end state of the game. States where the game has ended are called **terminal states.**

Utility function (also called an objective function or payoff function), - which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1, -1, or 0. Some games have a wider , variety of possible outcomes; the payoffs in backgammon range from +192 to -192.

The initial state and the legal moves for each side define the **game tree** for the game

Example : Tic – Tac – Toe (Noughts and Crosses)

From the initial state, MAX has nine possible moves. Play alternates between MAX'S placing an **X** and MIN'S placing an **O** until we reach leaf nodes corresponding to ter.mina1 states such that one player has three in a row or all the squares are filled.

Initial State : Initial Board Position

Successor Function : Max placing X's in the empty square

Min placing O's in the empty square

Goal State : We have three different types of goal state, any one to be reached.

i) If the O's are placed in one column, one row (or) in the diagonal continuously then, it is a goal state of min player. (Won by Min Player)

ii) If the X's are placed in one column, one row (or) in the diagonal continuously then, it is a goal state of min player. (Won by Max Player) iii) If the all the nine squares are filled by either X or O and there is no win condition by Max and Min player then it is a state of Draw between two players.

Some terminal states

Won by Min Players

X	O	X
	O	
	O	

Won by Max Players

X	O	X
	X	
X	0	O

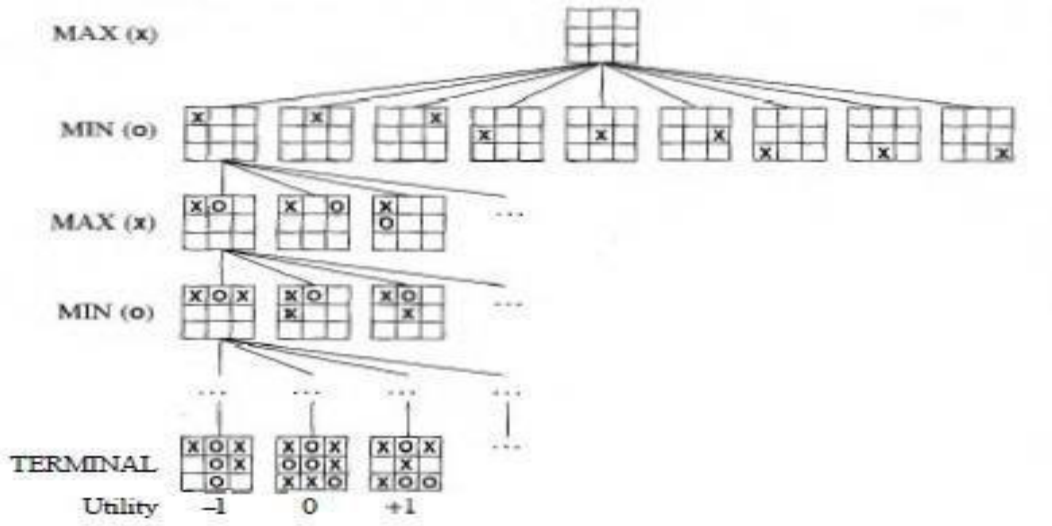
Draw between two Players

X	O	X
O	O	X
X	O	O

Utility function

Win = 1 Draw =0 Loss = -1

<u>A (partial) search tree for the game of tic-tac-toe</u>



Optimal strategies

In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state-a terminal state that is a win. In a game, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent

Given a game tree, the optimal strategy can be determined by examining the **minimax value** of each node, which we write as MINIMAX- VALUE(n).

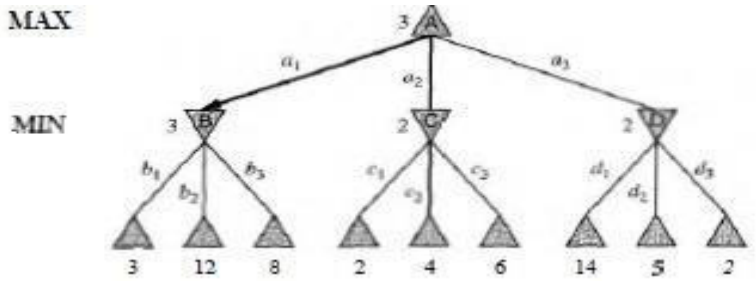
MINIMAX- VALUE(n) =

UTILITY(n)	if n is a terminal state
$Max_{s \in successors(n)}$ MAX-VALUE(s)	if *n* is a MAX node
$Min_{s \in Successors(n)}$ MAX-VALUE(s)	if *n* is a MIN node

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree.

The possible moves for MAX at the root node are labeled a_l, a_2, and a_3. The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on.

A Two Ply Game Tree



- Moves by Max Player
- Moves by Min Player

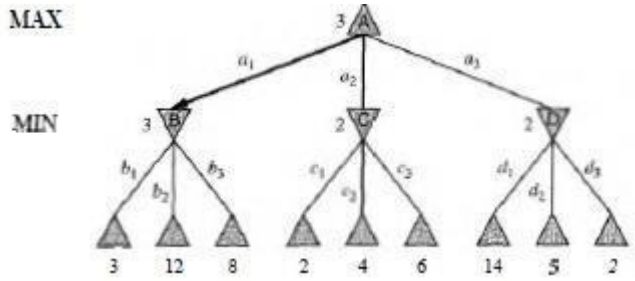
- The terminal nodes show the utility values for MAX; The other nodes are labeled with their minimax values.
- MAX'S best move at the root is a_1, because it leads to the successor with the highest minimax value
- MIN'S best reply is b_1, because it leads to the successor with the lowest minimax value.
- The root node is a MAX node; its successors have minimax values 3, 2, and 2; so it has a minimax value of 3.
- The first MIN node, labeled B, has three successors with values 3, 12, and 8, so its minimax value is 3**3**

The minimax algorithm

The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the

leaves of the tree, and then the minimax values are **backed** up through the tree as the recursion unwinds.

For Example



The algorithm first recurses down to the three bottom left nodes, and uses the UTILITY function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally, we take the maximum of 3,2, and 2 to get the backed-up value of 3 for the root node.

An algorithm for minimax decision

function MINIMAX-DECISION *(state)* returns *an action* inputs: state, current state in game *v* <- MAX-VALUE(state) return the action in SUCCESSORS*(state)* with value v

An algorithm for minimax decision

```
function MINIMAX-DECISION (state) returns an action
     inputs: state, current state in game
v <- MAX-VALUE(state)
return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v<- -∞
for a, s in SUCCESSORS(state) do
v <- MAX(v, MIN-VALUE(s))
return v
```

```
function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
V <- ∞
for a, s in SUCCESSORS(state)do'
v <- MIN(v, MAX-VALUE(s))
return v
```

- Generate the whole game tree, all the way down to the terminal state.
- Apply the utility function to each terminal state to get its value.
- Use utility functions of the terminal state one level higher than the current value to determine Max or Min value.
- Minimax decision maximizes the utility under the assumption that the opponent will play perfectly to minimize the max player score.

Complexity : If the maximum depth of the tree is m, and there are b legal moves at each point then the time complexity of the minimax algorithm is $O(b^m)$. This algorithm is a

depth first search, therefore the space requirements are linear in m and b. For real games the calculation of time complexity is impossible, however this algorithm will be a good basis for game playing.

Completeness : It the tree is finite, then it is complete.

Optimality : It is optimal when played against an optimal opponent

ALPHA - BETA PRUNING

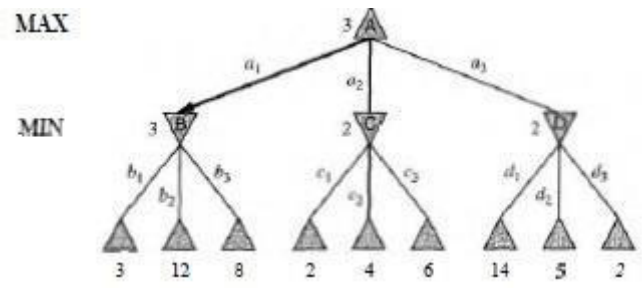
Pruning - The process of eliminating a branch of the search tree from consideration without examining is called pruning.

The two parameters of pruning technique are:

Alpha (α) : Best choice for the value of MAX along the path (or) lower bound on the value that on maximizing node may be ultimately assigned.

Beta (β) : Best choice for the value of MIN along the path (or) upper bound on the value that a minimizing node may be ultimately assigned.

Alpha - Beta pruning : The α and β values are applied to a minimax tree, it returns the same move as minimax, but prunes away branches that cannot possibly influence the final decision is called **Alpha - Beta pruning (or) Cutoff** Consider again the two-ply game tree from



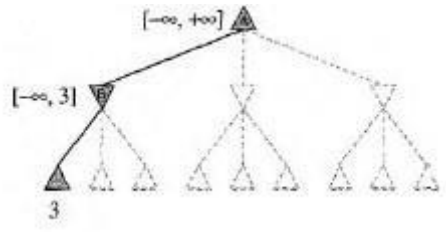
Let the two unevaluated successors of node C have values x and y and let z be the minimum of x and y. The value of the root node is given by

MINIMAX-VALUE(ROOT)=max((min(3,12,8),min(2,x,y),min(l4,5,2))
= max(3, min(2, x, y), 2) = max(3, z, 2) where z <=2 = **3.**

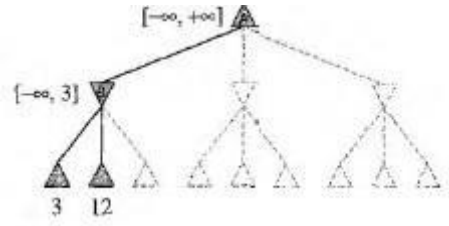
In other words, the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.

Stages in the calculation of the optimal decision for the game tree

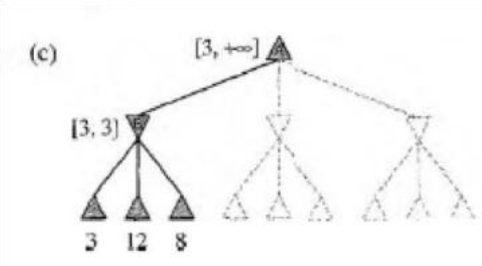
(a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3



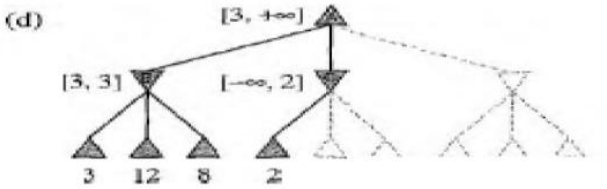
(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3



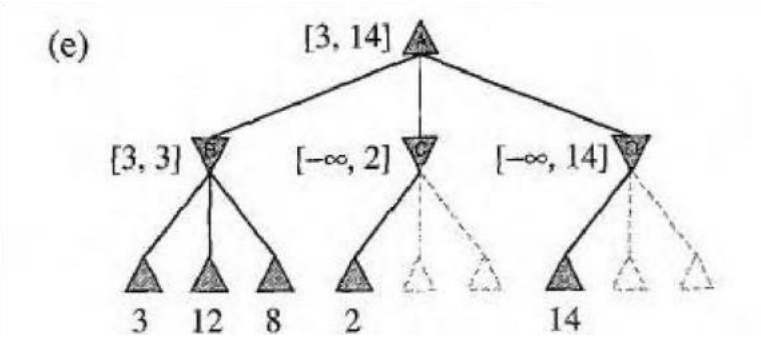
c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.



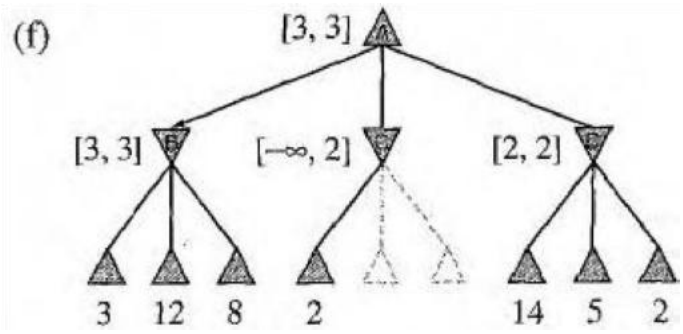
(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha-beta pruning.



(e) The first leaf below D has the value 14, so D is worth atmost 14. This is still higher than MAX'S best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.



(f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX'S decision at the root is to move to B, giving a value of 3.



The alpha-beta search algorithm

```
function ALPHA-BETA-SEARCH (returns an ) action
inputs: state, current state in game
v<-  M A X -V A L U E (State,- α , +α )
return the action in SUCCESSORS(state) with value v
```

```
function MAX - VALUE ( state, α,β)  returns α utility
value
inputs: state, current state in game
α , the value of the best alternative for MAX along the
path to state
β, the value of the best alternative for MIN along the
path to state
if TERMINAL-TEST(state)  then  return) UTILITY( s t a t e
)
v<- -∞
for α, S in S U C C E S S O R S (state) do
v <-  MAX(v,MIN-VALUE(S, α, β ))
if v >= β then return v
α <- MAx(α, v)
return v
```

```
function MIN - VALUE ( state, α,β)  returns α utility
value
inputs: state, current state in game
α , the value of the best alternative for MAX along the
path to state
β, the value of the best alternative for MIN along the
path to state
if TERMINAL-TEST(state)  then  return) UTILITY( s t a t e
)
v<- +∞
for α, S in S U C C E S S O R S (state) do
v <-  MIN(v,MAX-VALUE(S, α, β ))
if v <= α then return v
β <- MIN(β, v)
return v
```

Effectiveness of Alpha – Beta Pruning

Alpha - Beta pruning algorithm needs to examine only $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ with minimax algorithm, that is effective branching factor is $\sqrt{b}$ instead of b.

Imperfect Real Time Decisions

The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it. However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time-typically a few minutes at most.

Shannon's proposed instead that programs should cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning non terminal nodes into terminal leaves.

- The utility function is replaced by an Evaluation function
- The terminal test is replaced by a Cut-off test

1. Evaluation function

Example: Chess Problem

In chess problem each material (Queen, Pawn, etc) has its own value that is called as material value. From this depends on the move the evaluation function is calculated and it is applied to the search tree.

This suggests that the evaluation function should be specified by the rules of probability.

For example If player A has a 100% chance of winning then its evaluation function is 1.0 and if player A has a 50% chance of winning, 25% of losing and 25% of being a draw then the probability is calculated as; 1x0.50 -lxO.25 + OxO.25 = 0.25.

As per this example is concerned player A is rated higher than player B. The material value, of each piece can be calculated independently with-out considering other pieces in the board is also called as one kind of evaluation function and it is named as weighted linear function. It can be expressed as

Eval(s) = $w_1f_1(s) + w_2f_2(s) + w_3f_3(s)..... + w_nf_n(s)$ w - Weights of the pieces (1 for Pawn, 3 for Bishop etc) f - A numeric value which represents the numbers of each kind of piece on the
board.

2. Cut - off test

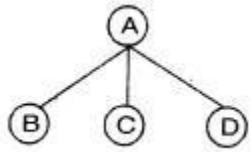
To perform a cut-off test, an evaluation function, should be applied to positions that are quiescent, that is a position that will not swing in bad value for long time in the search tree is known as waiting for quiescence.

Quiescence search - A search which is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

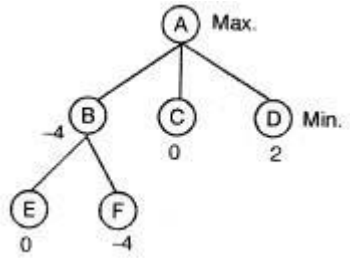
Horizon problem - When the program is facing a move by the opponent that causes serious damage and is ultimately unavoidable

Example:

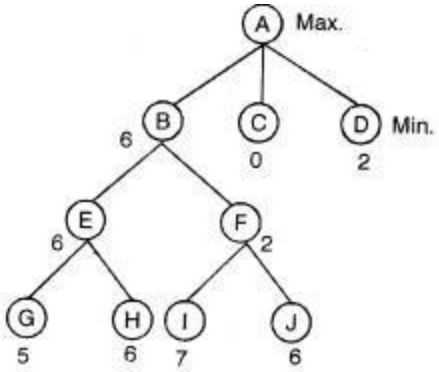
1. Beginning of the search - one ply



2. This diagram shows the situation of horizon problem that is when one level is generated from B, it causes bad value for B



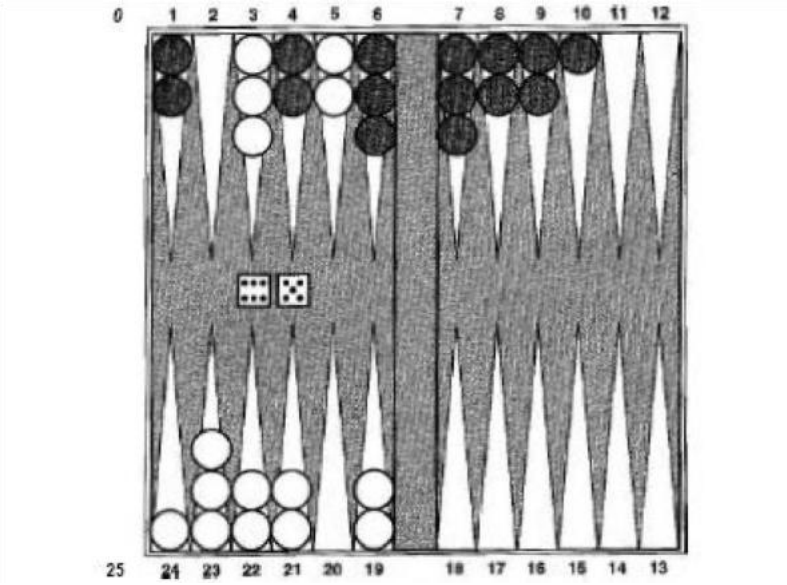
3. When one more successor level is generated from E and F and situation comes down and the value of B is retained as a good move. The time B is waited for this situation is called waiting for quiescence.



Games That Include An Element Of Chance

Backgammon Game

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves.



Goal State

The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counterclockwise toward 0

Successor Function or Operator

Move to any position except where two or more of the opponent pieces. If it moves to a position with one opponent piece it is captured and again it has to start from Beginning

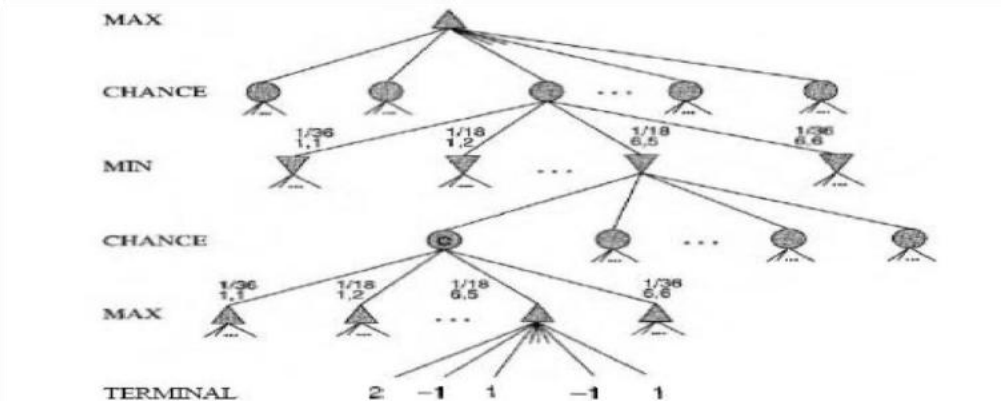
Task : In the position shown, White has rolled 6-5. So Find out the legal moves for the set of the dice thrown as 6 and 5.

Solution :

There are Four legal moves. They are

- (5-11,5-10)
- (5-11, 19-24)
- (10-16,5-10)
- (5-11,11-16)

A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles. The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur. There are 36 ways to roll two dice, each equally likely; but because a 6-5 is the same as a 5-6, there are only 21 distinct rolls. The six doubles (1-1 through 6-6) have a 1/36 chance of coming up, the other 15 distinct rolls a 1/18 chance each.



The resulting positions do not have definite minimax values. Instead, we have to only calculate the **expected value,** where the expectation is taken over all the possible dice rolls that could occur.

Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before; chance nodes are evaluated by taking the weighted average of the values resulting from all possible dice rolls, that is,

EXPECTIMINIMAX(n)=

UTILITY(n) if n is a terminal state

Max $_{s \in successors(n)}$ EXPECTIMINIMAX(S) if n is a MAX node

Min $_{s \in successors(n)}$ EXPECTIMINIMAX(S) if n is a MIN node

$\sum_{s \in successors(n)}$ P(s).EXPECTIMINIMAX(S) if n is a chance node

where the successor function for a chance node n simply augments the state of n with each possible dice roll to produce each successor s and P(s) is the probability that that dice roll occurs.

Card games

Card games are interesting for many reasons besides their connection with gambling.

Imagine two players, MAX and MIN, playing some practice hands of four-card two handed bridge with all the cards showing.

The hands are as follows, with MAX to play first:

MAX : ♢6 , ♣6 , ♣9 , 8

♠MIN: ♡ 2 , ♣ 4 , ♣10 , 5

Suppose that MAX ♣ ♣ leads wiht 9. MIN must now follow suit, playing either with ♣ 10 or ♣ 5 . MIN plays with 10 and wins the trick.

MIN goes next turn leads the with ♠ 2. MAX has no spades (and so cannot win the trick) and therefore must throw away some card. The obvious choice is the ♢6 because the other two remaining cards are winners.

Now, whichever card MIN leads for the next trick, MAX will win both remaining tricks and the game will be tied at two tricks each.

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:-

o In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.

o For example,

At(Home)^ ¬ Have(Milk)^ ¬ Have(Bananas)^ ¬ Have(Drill)^....

o Goals are also described by conjunctions of literals.

o For example,

At(Home)^Have(Milk)^ Have(Bananas)^ Have(Drill)

o Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as

- **Representation for actions:-** o Our STRIPS operators consist of three components:

o 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:-** o Consider a simple problem: o Putting on a pair of shoes o 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:-

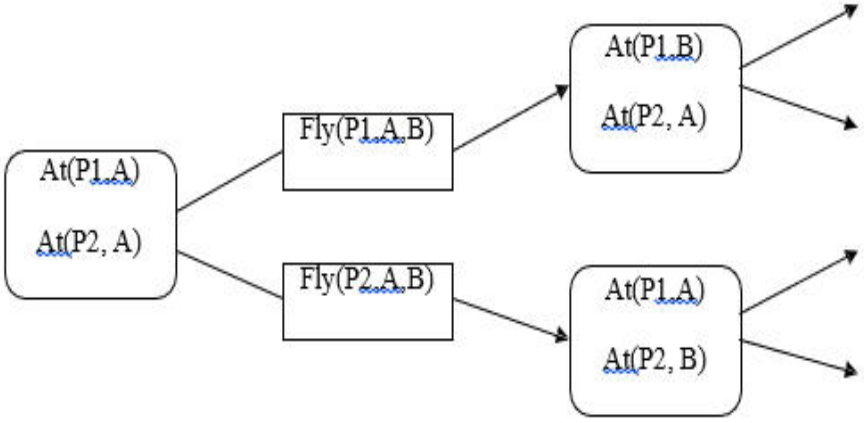
- Systems design
- Budgeting
- Manufacturing product
- Robot programming and control
- Military activities

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.

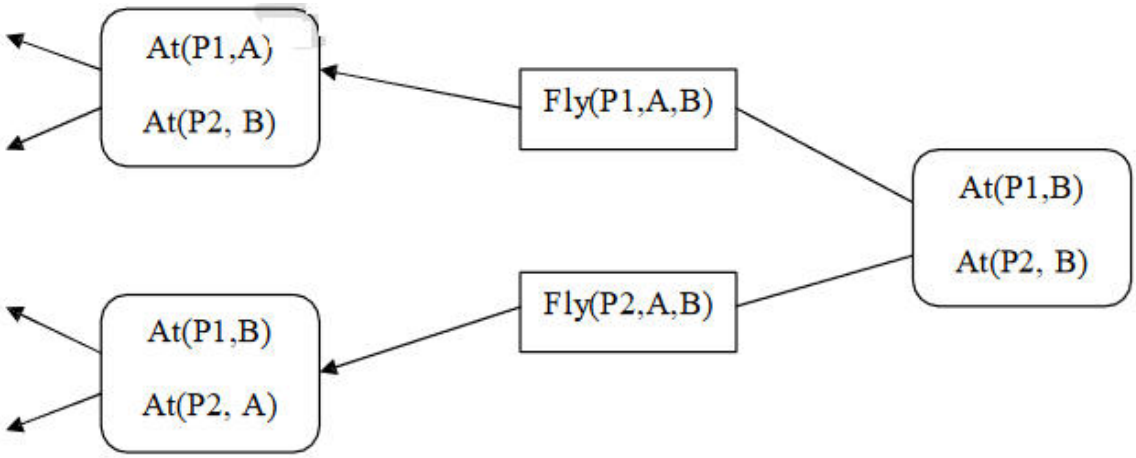
• The formulation of planning problems as state-space search problems is as follows, o The **Initial state** of the search is the initial state from the planning problem. o In general, each state will be a set of positive ground literals; literals not appearing are false. o The **actions** that are applicable 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. o 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:- o 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. o 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:- o The goal in our 10-airport cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

At(C1,B) ∧ At(C2,B) ∧.....∧ At(C20,B)

o 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. o 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) ∧At(C2,B) ∧.....∧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) ∧At(p,B), Moreover the subgoal At(C1,B) should not be true in the predecessor state.

- The predecessor description is

In(C1,p) ∧ At(p,B) ∧ At(C2,B) ∧.....∧ At(C20,B)

In addition to insisting that actions achieve some desired literal, we must insist that the actions not undo any desired literals.

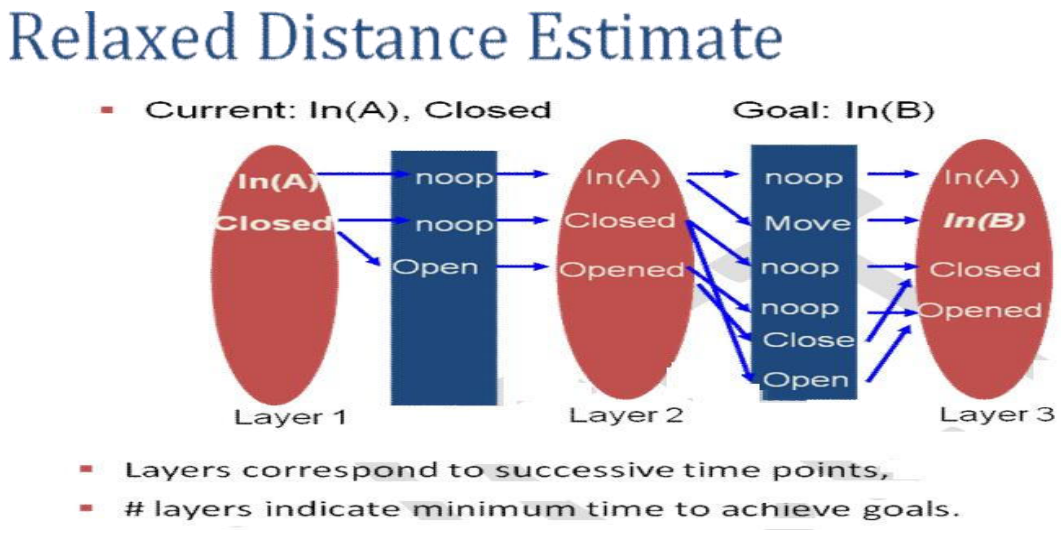
- An action that satisfies this restriction is called consistent.
- From definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search.
- Given a goal description G, let A be an action that is relevant and consistent. The corresponding predecessor is as follows o any positive effects of A that appear in G are deleted o 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}



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.

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, o A description of the agent's goal in some formal language, and o 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:-

o 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:-** o 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:-**

o Consider a simple problem:

o Putting on a pair of shoes o Goal

RightShoeOn ^ LeftShoeOn

o Four operators:

Op(Action:RightShoe,PreCond:RightSockOn,Effect:Right
Op(Action:RightSock , Effect:
Op(Action:LeftShoe, Precond:LeftSockOn,
Op(Action:LeftSock,Effect:Left

- **Least** The general strategy of delaying a choice during search is commit
- **Partial-order** Any planning algorithm that can place two actions without specifying which come first is called a partial order
- **Lineari** - The partial-order solution corresponds to six possible total order of these is called a linearization of the partial
- **Total order** - Planner in which plans consist of a simple lists
- A plan is defined as a data
  - A set of plan
  - A set of step
  - A set of variable binding
  - A set of causal i ç s ″s achieves j″
- Initial plan before any
  Start <

Refine and manipulate until a plan that is a

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

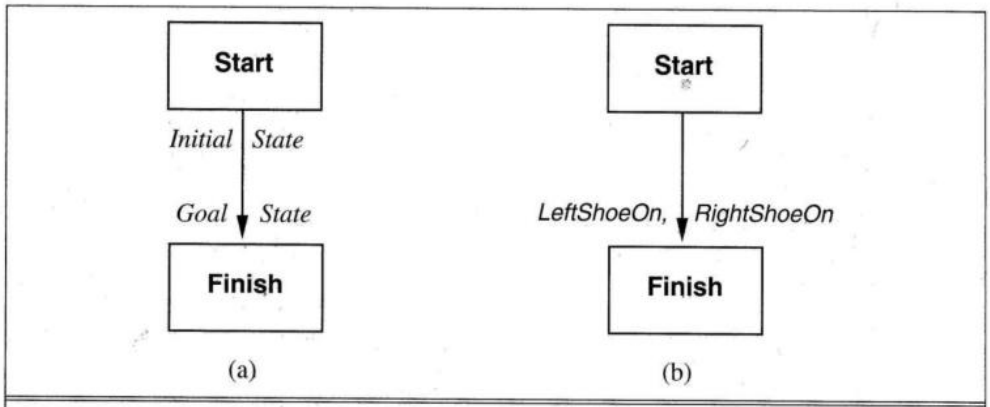
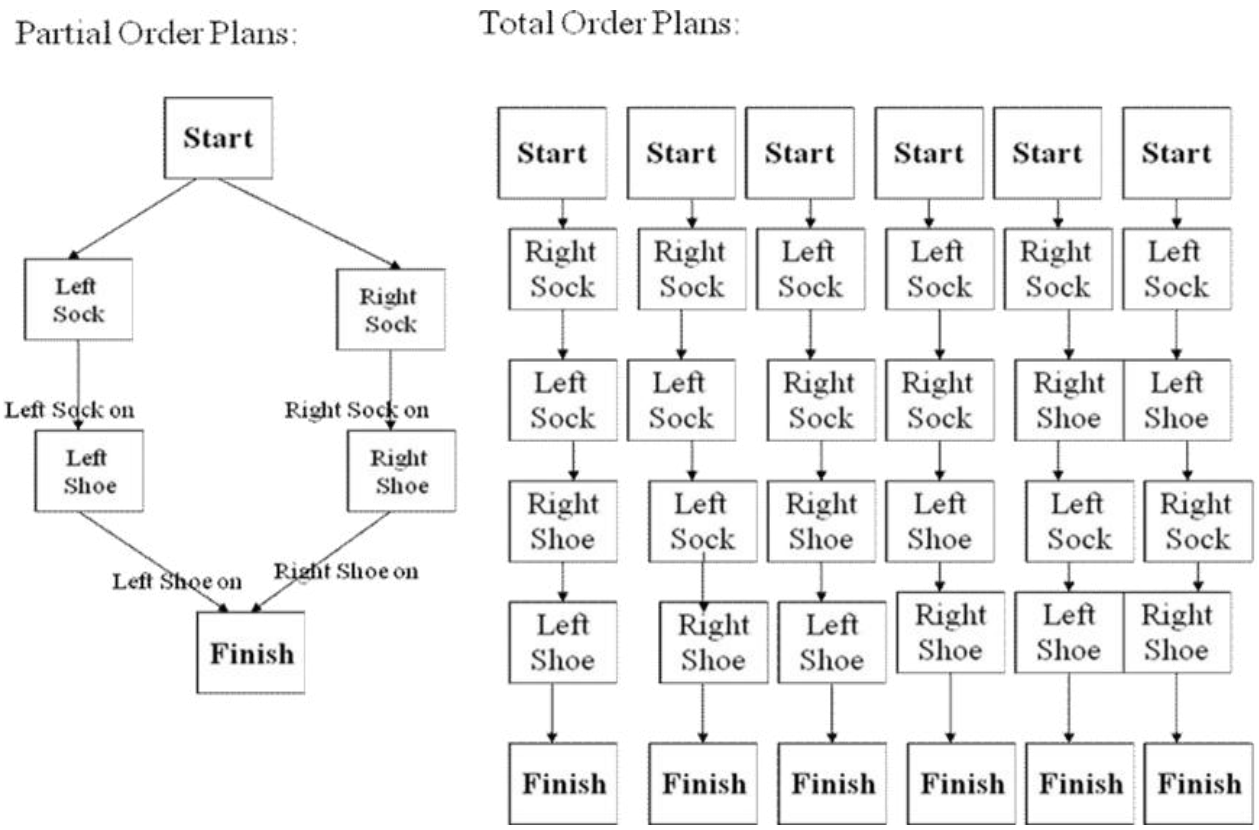


Figure 11.4 (a) Problems are defined by partial plans containing only *Start* and *Finish* steps. The initial state is entered as the effects of the *Start* step, and the goal state is the precondition of the *Finish* step. Ordering constraints are shown as arrows between boxes. (b) The initial plan 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:-** o Shopping problem: "get milk, banana, drill and bring them back home" o 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

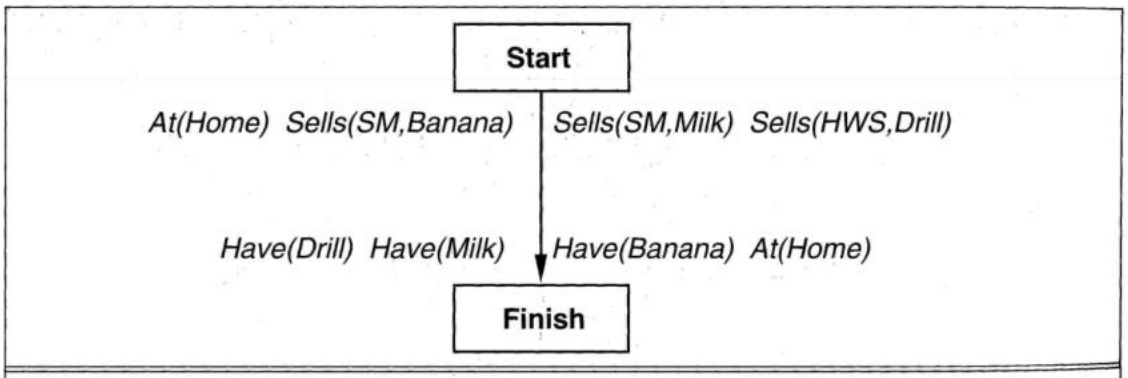
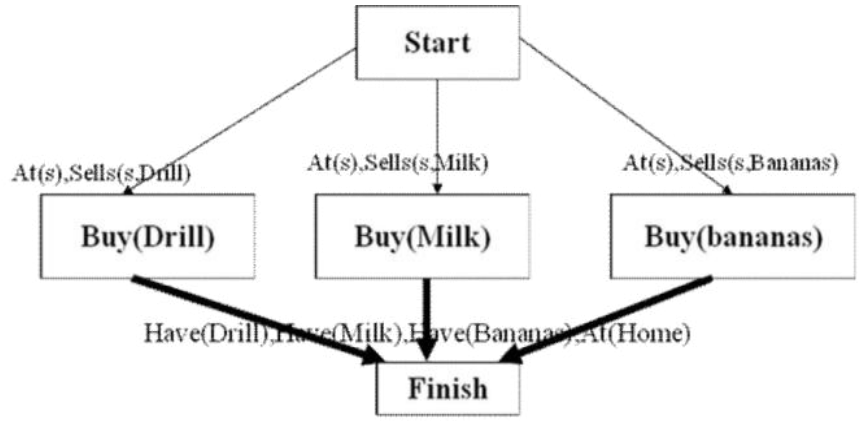
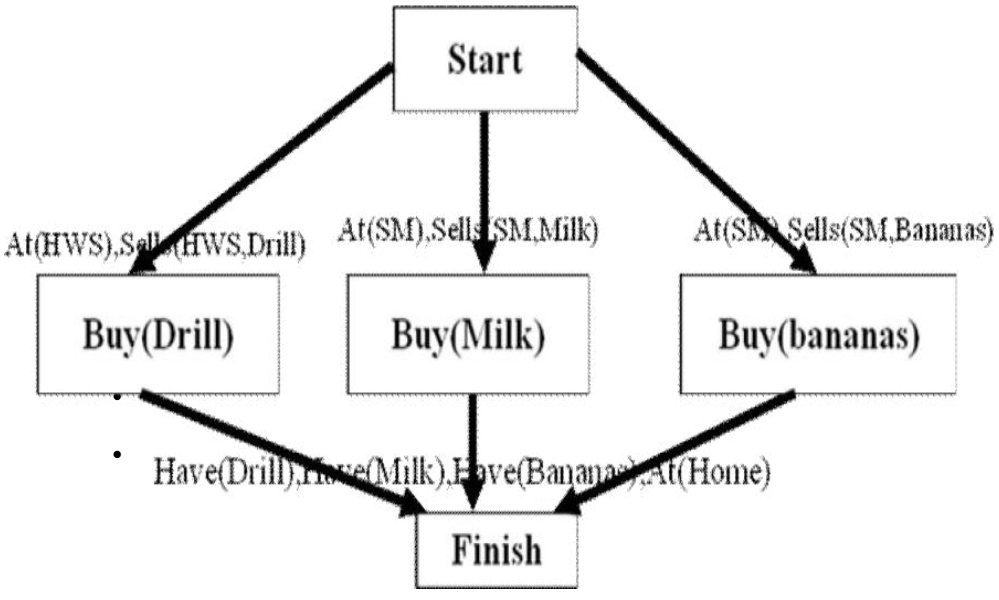


Figure 11.6 The initial plan for the shopping problem.

The following diagram shows the, partial plan that achieves three of four preconditions of finish

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.





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

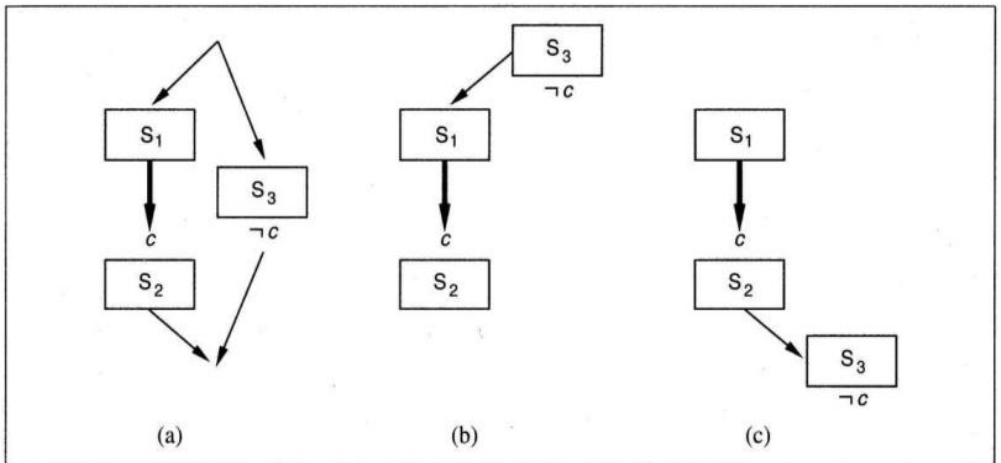
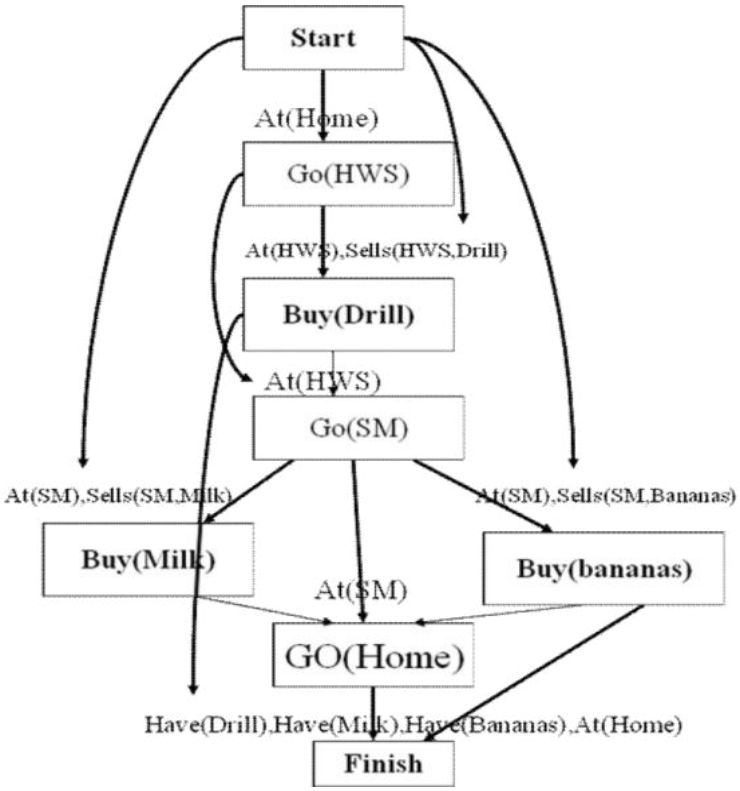


Figure 11.10 Protecting causal links. In (a), the step S_3 threatens a condition c that is established by S_1 and protected by the causal link from S_1 to S_2. In (b), S_3 has been demoted to come before S_1, and in (c) it has been promoted to come after S_2.

- 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 probleminstance

(5) pose problems to the planner and get back plans

- (ex) The blocks world o

(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 o Achieve better heuristic estimates o 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 o Records only a restricted subset of possible negative interactions among actions.
- Each level consists of o *Literals* = all those that *could* be true at that time step, depending upon the actions executed at preceding time steps.

o *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) $\wedge$ Eaten(Cake))

Action(Eat(Cake),

PRECOND: Have(Cake)

EFFECT: ¬Have(Cake) $\wedge$ Eaten(Cake))

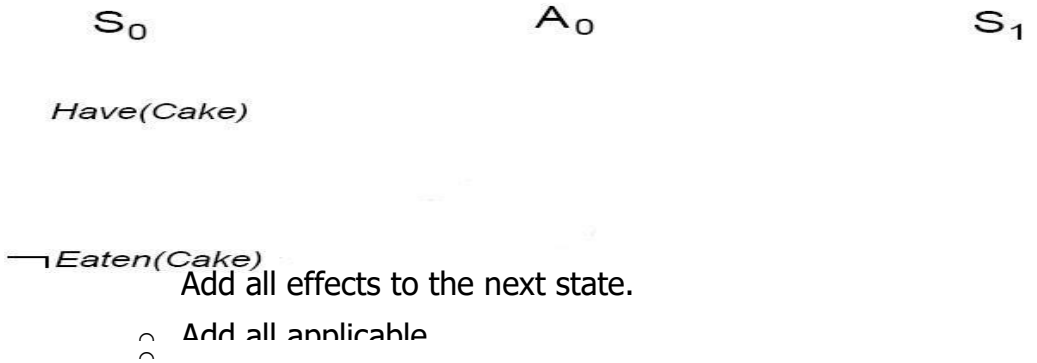
Action(Bake(Cake),

PRECOND: ¬ Have(Cake)

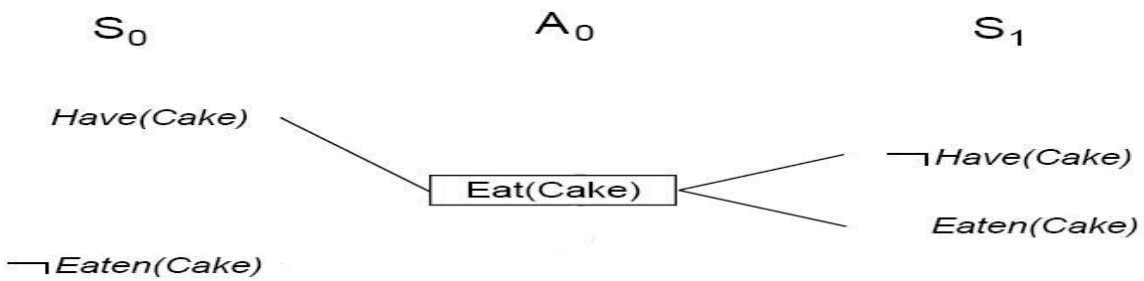
EFFECT: Have(Cake))

- Steps to create planning graph for the example,

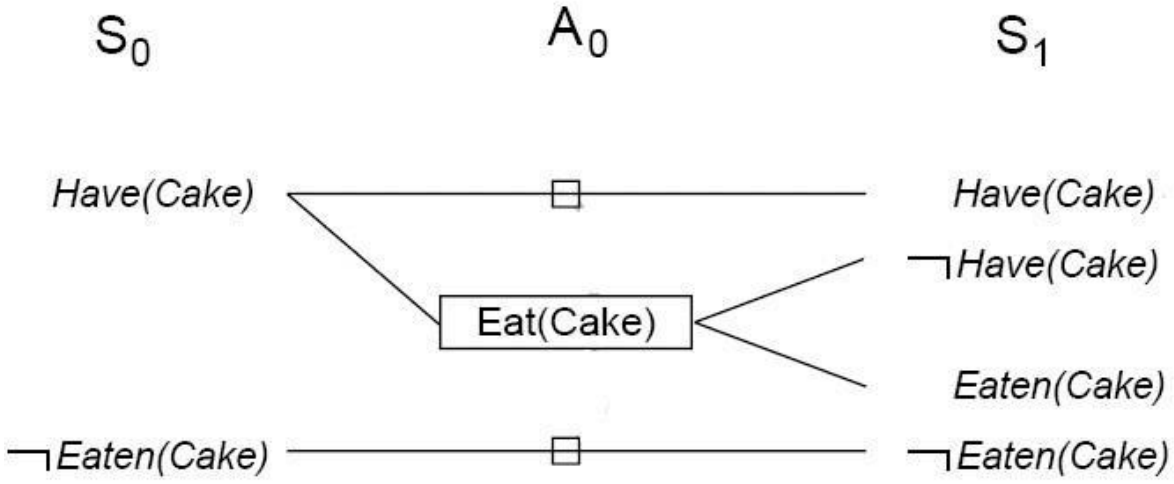
- Create level 0 from initial problem



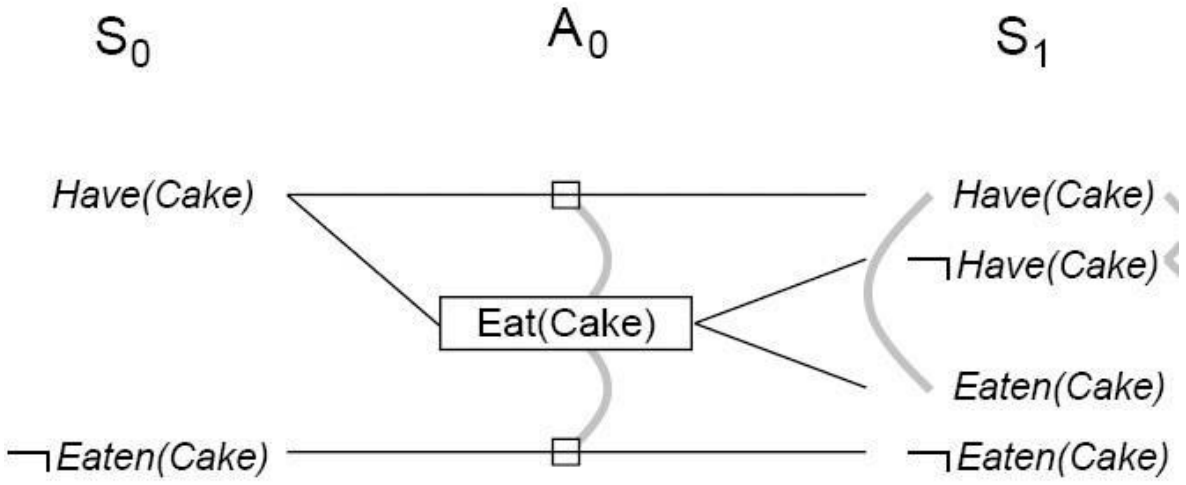
- Add all effects to the next state.
- Add all applicable



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



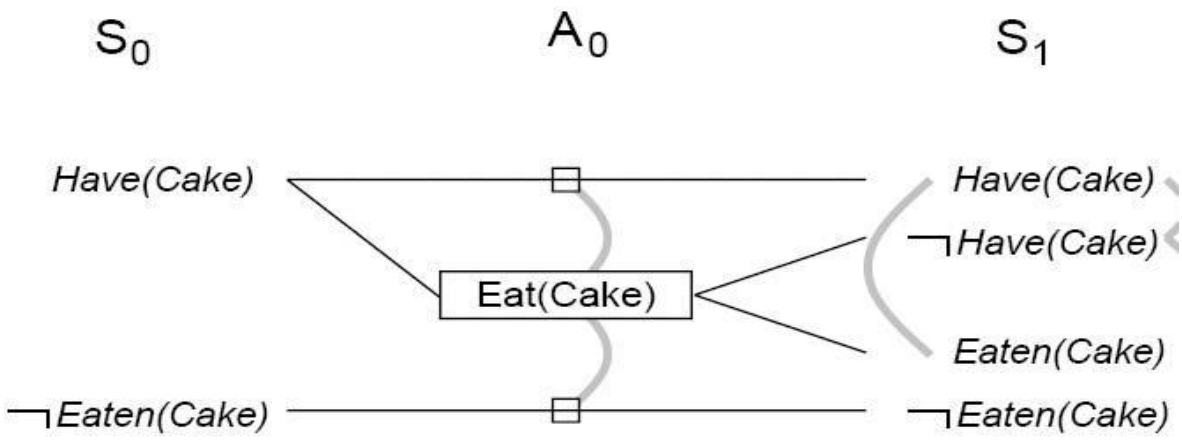
- Ide *mutual* between actions and literals based on potential conflicts.



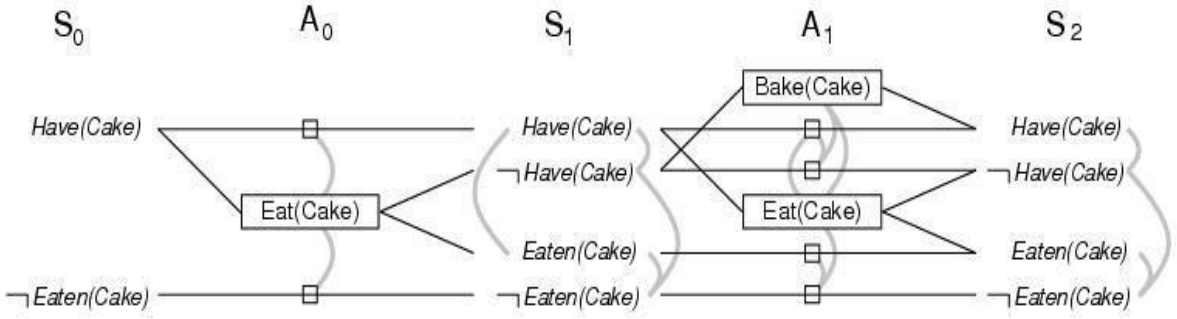
- Mutual
  - A mutex relation holds **two** w
    - *Inconsistent* : one action negates the effect of
    - *Interfer* : one of the effects of one action is the negation of a precondition of the
    - *Competing* : one of the preconditions of one action is mutually exclusive with the precondition of the
  - A mutex relation holds **two** w

- 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 o Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.

o 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
    (explanation follows



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

3 Planning Graphs for Heuristic :

• PG's provide information about the problem o PG is a relaxed problem. o A literal that does not appear in the final level of the graph cannot be achieved by any plan.

- H(n) = ∞ o 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 o Max-level: maximum first level of any of the goals o Sum-level: sum of first levels of all the goals

o 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

o Inconsistent effects

• E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and **not** At(Spare, Ground)

o Interference

• E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and **not** At(Flat,Axle) as EFFECT

o Competing needs

- E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and **not** At(Flat, Axle)

o 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

o Solution might exist and EXTRACT-SOLUTION will try to find it

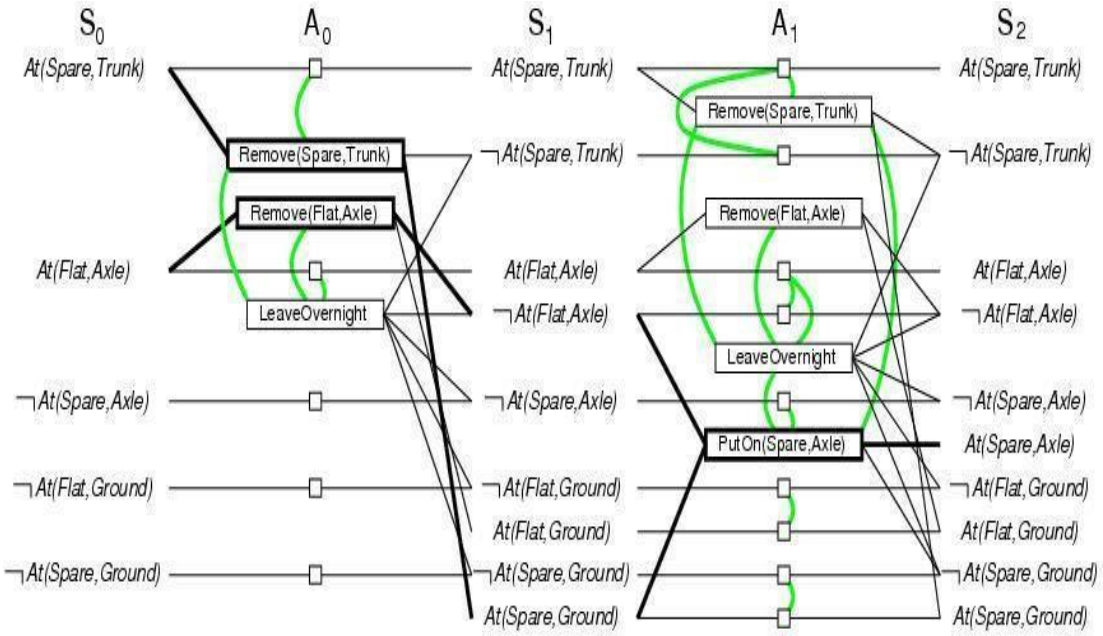
- EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:

o Initial state = last level of PG and goal goals of planning problem

o Actions = select any set of non-conflicting actions that cover the goals in the state

o Goal = reach level S0 such that all goals are satisfied

o 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, o Hubble Space Telescope Observations o Operating factories

o 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.

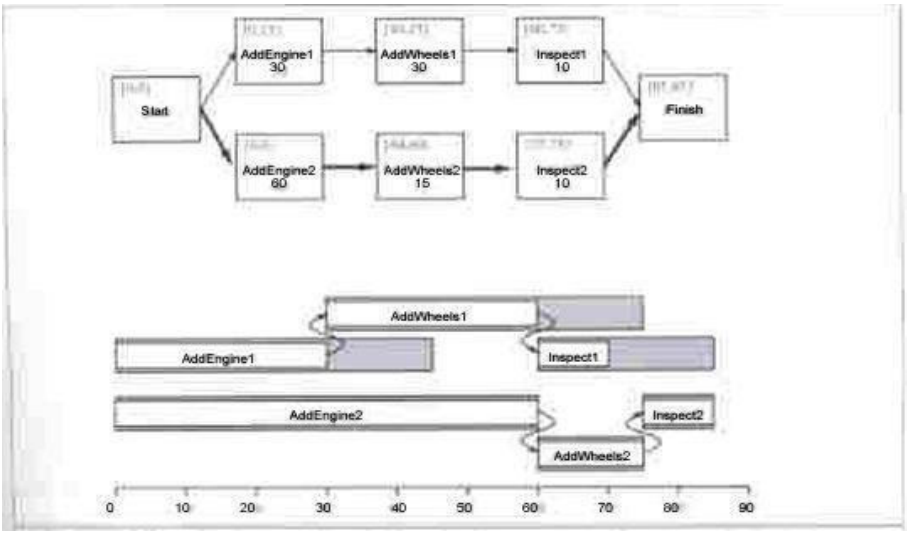
• The problem is to determine a schedule that minimizes the total time required to complete all the jobs, while respecting the resource constraints.

• 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 notat ion 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 shorterthan.



- 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 πBES (A) +Duration(A) LS (Finish) =ES (Finish) LS (A) = min A πBLS (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) $\wedge$chassis(C2)

$\wedge$ Engine (E1,C1,30)

$\wedge$ Engine (E2,C2,60)

$\wedge$ Wheels (W1,C1,30) $\wedge$ Wheels (W2,C2,15) $\wedge$EngineHoists (1) $\wedge$WheelStations (1) $\wedge$Inspectors (2))

Goal (Done(C1) $\wedge$ 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))

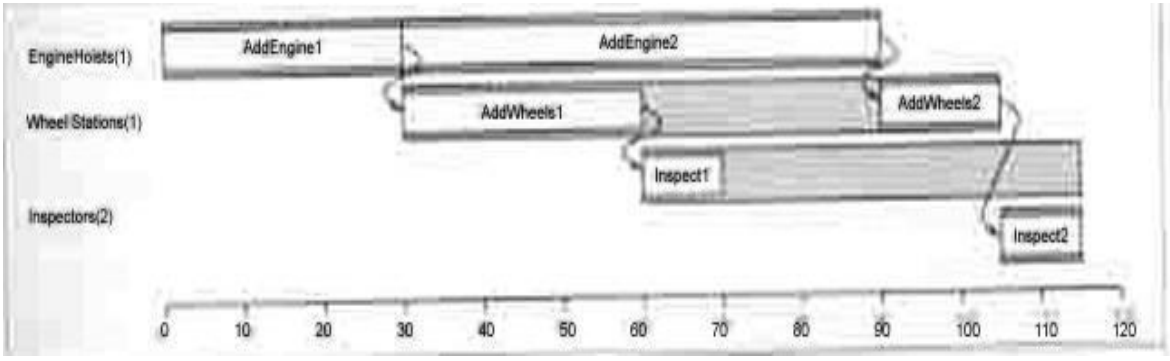
Action (Inspect(c),

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.

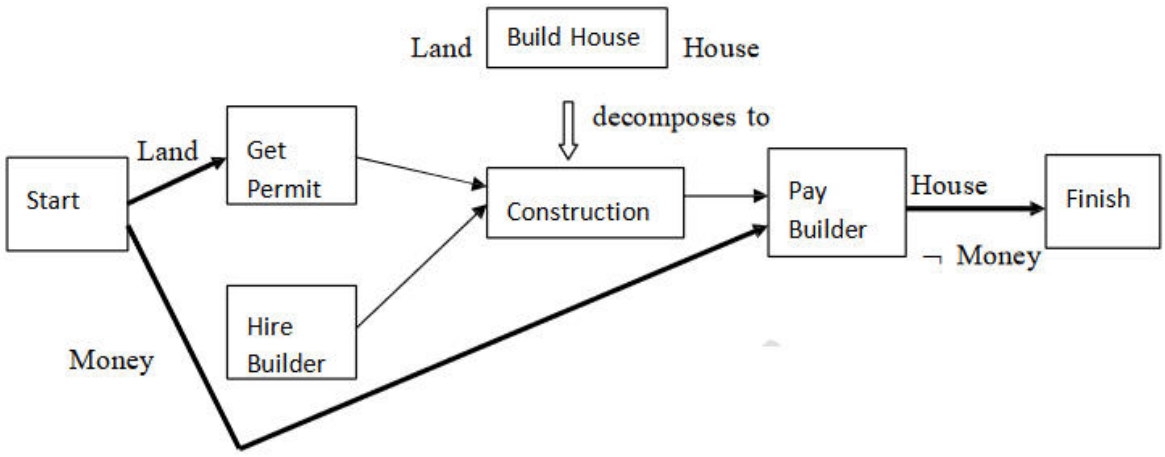
- One simple but popular heuristic is the minimum slack algorithm.
- it schedules actions in a greedy fashion.
- On each iteration, it considers the unscheduled actions that have had all their predecessors scheduled and schedules the one with the least slack for the earliest possible start.
- It then 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 ∧¬ Money)

Action (GetLoan, PRECOND: GoodCredit, EFFECT:Money ∧ Mortgage)

Action (BuildHouse, PRECOND: Land, EFFECT: House)

Action (GetPermit, PRECOND: Land, EFFECT: Permit)

Action (HireBuilder, EFFECT: Contract)

Action (Construction, PRECOND: Permit ∧Contract, EFFECT: HouseBuilt ∧¬ Permit)

Action (PayBuilder, PRECOND: Money ∧ HouseBuilt, EFFECT: ¬ Money ∧ House ∧¬ Contract)

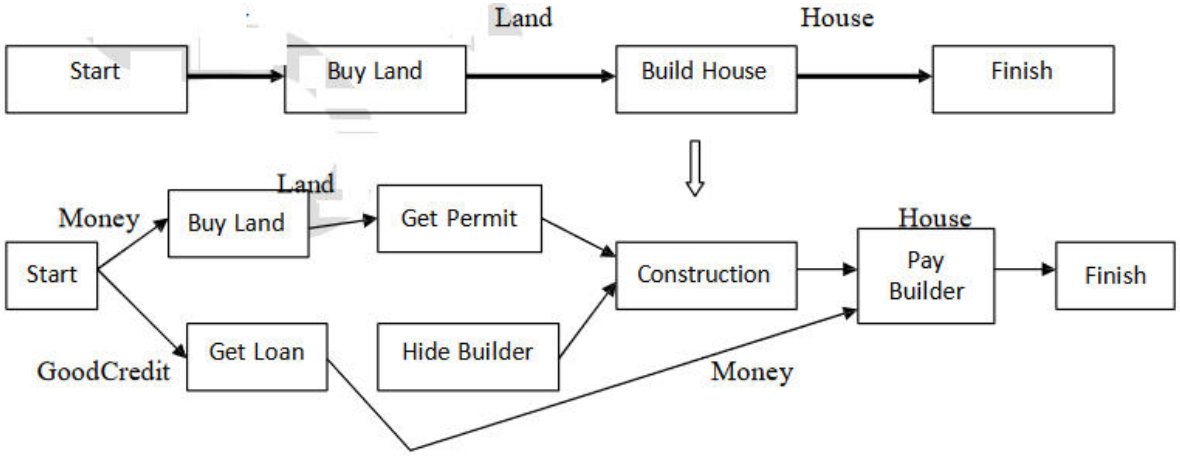
Decompose (BuildHouse, Plan (Steps :

{S1: GetPermit, S2: HireBuilder, S3: Construction, S4: PayBuilder} ORDERINGS: {Start π S1 π S3 π S4 Finish, Start π S2 π S3}, Links: {Start LandS1, Start MoneyS4, S1permitS3, S2 ContractS3, S3 HouseBuilt S4, S4 HouseFinish, S4 ¬MoneyFinish}))

- 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, o First the action a' is removed from P.Then for each step S in the decomposition d' o 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, o **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.

o **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, o

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.

o **Continuous Planning:-**

o It is designed to persist over a lifetime.

o 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) o initial state : the robot in the right square of a clean world;o the environment is fully observable:

AtR ∧CleanL∧CleanR.o

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∧¬ 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)

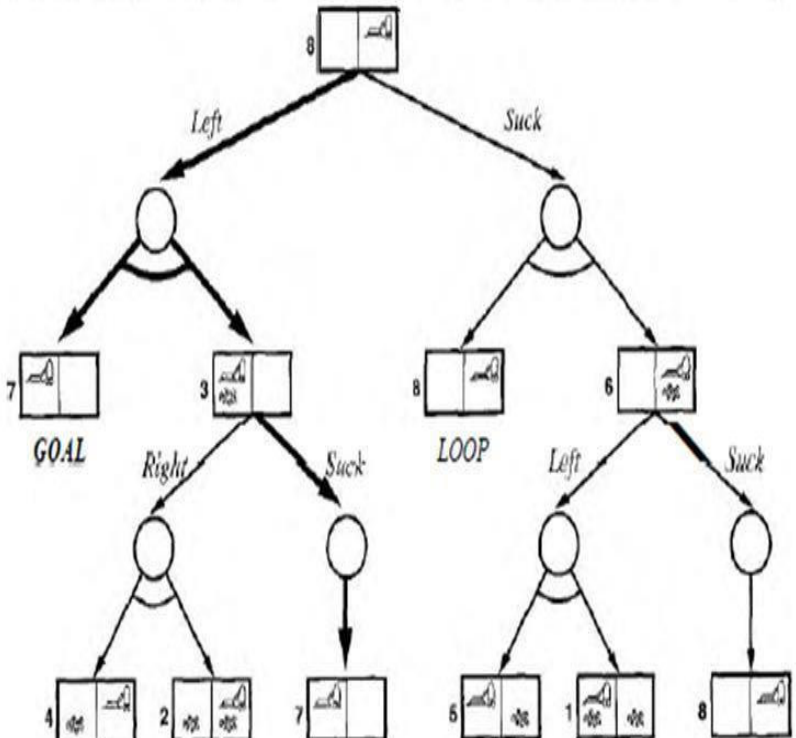
o Conditional steps for creating conditional plans:

if test then planA else planB e.g., if AtL ^ CleanL then Right else Suck

o 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.

o State nodes are OR nodes where some action must be chosen.



- Chance nodes, circles, are wher outcome handled, as indicated by the arc linking the
- The solution is **bo** in

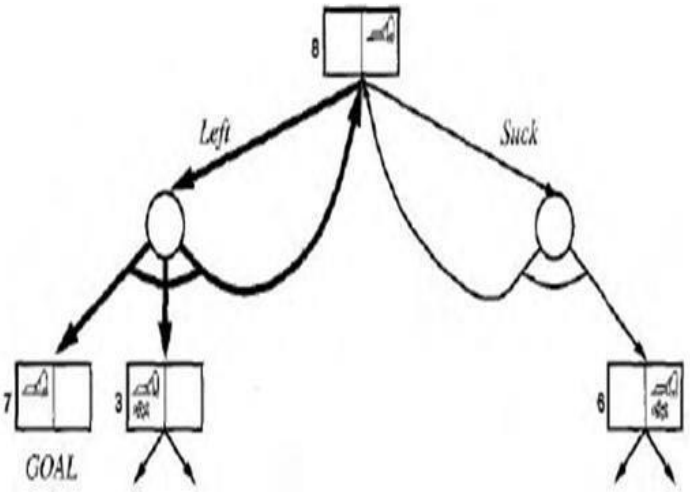
- The following table shows the recursive depth first algorithm for

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(INITIAL-STATE[problem], problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if GOAL-TEST[problem](state) then return the empty plan
  if state is on path then return failure
  for each action, state-set in SUCCESSORS[problem](state) do
      plan ← AND-SEARCH(state_set, problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(state_set, problem, path) returns a conditional plan, or failure
  for each s_i in state-set do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

- The following figure shows the part of
- c t a n l any solutio AND-OR-GRAPH-SEARCH r w failur i how cyclic i kee L u i w



- The first level of the search graph for the triple Murphy vacuum world, cycles
- All solutions for this problem

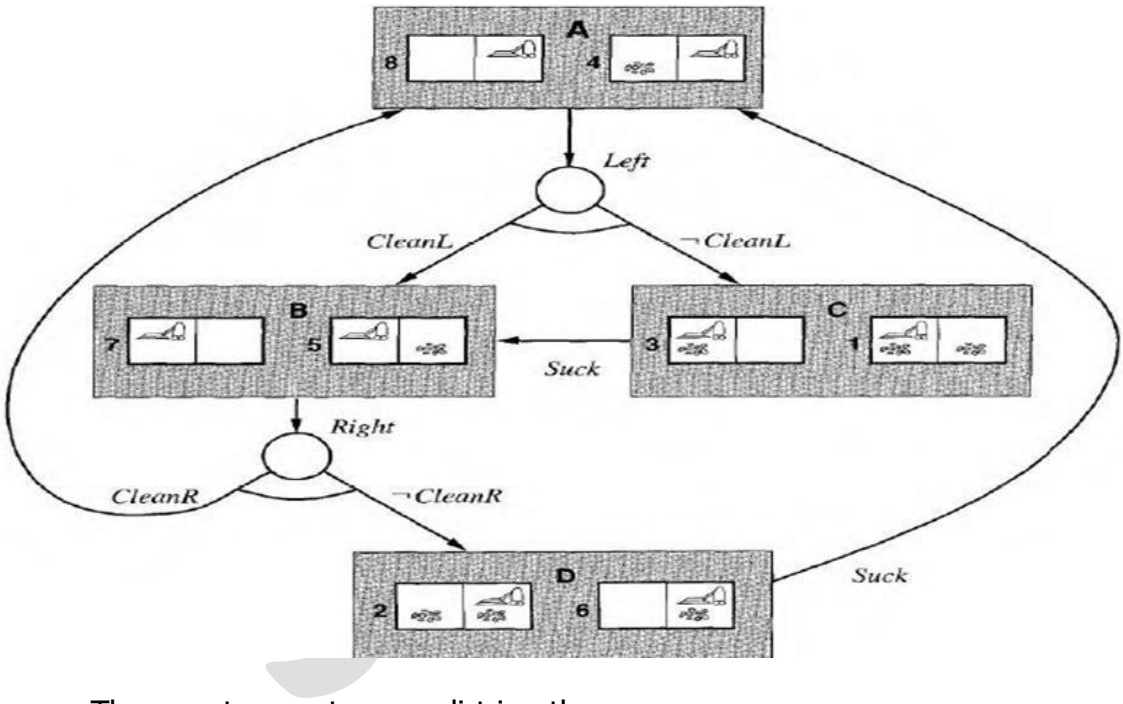
- The cyclic solution is as follows,

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

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
  - The state set is a way of describing the agents initial belief
  - "both squares are " with local dirt
    - the vacuum agent is AtR and knows about R, how about
  - The following shows part of the AND-OR f the alternate Mu vaccum
  - In which Dirt can sometimes be left behind when the agent leaves a clean



  - The agent cannot sense dirt in other

- Sets of full state descriptions o { *(AtR ∧ CleanR ∧ CleanL), (AtR ∧ CleianR ∧ ¬CleanL) }*
- Logical sentences that capture exactly the set of possible worlds in the belief state.

o *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.

There are two choices here,

o **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, o Simple, but weak form called action monitoring – whereby the agent checks the environment to verify that the next action will work. o 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.

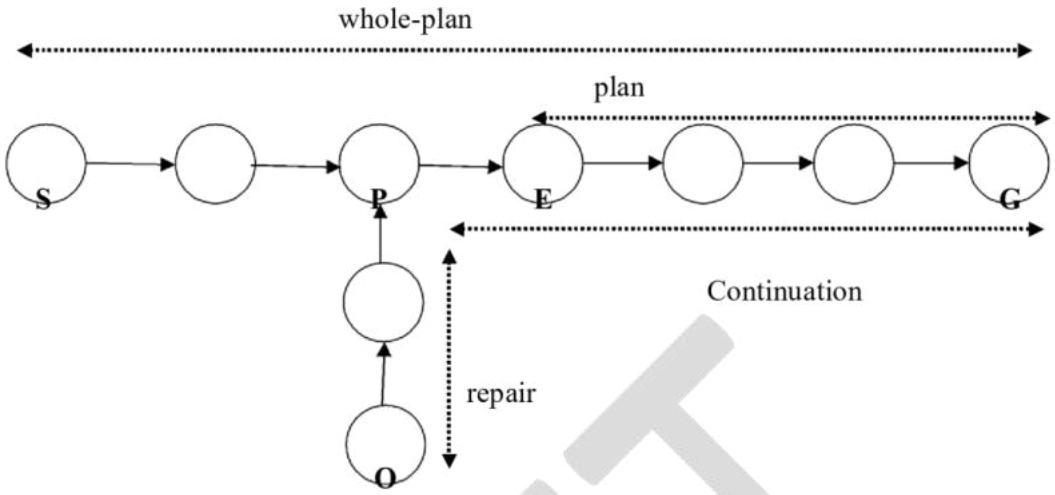
- If they have, the agent will try to get back on track by replanning a should take it back to some point in
- The **has an agent that does action**
- It uses a complete state-space planning algorithm called PLANNER
- If the preconditions of the next action are not met, the agent loops through p in whole-plan,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

Fu REPLANNING- **r** an
S KB, a Knowledge base (includes action
Plan, a plan,
Whole-plan, a plan,
Goal,

TELL(KB,MAKE-PERCEPT-
C STATE-
I pla **t**
whol
I PRECONDITIONS(FIRST(plan)) not cu**t**rently
Can SORT(whole-plan, ordered by
Find state s in
Failure
Contin the tail of whole-plan
Who p APPEND(repair,
R PO

- The following diagram shows the schematic illustration
- The illustration of process is also called as
- The replanner notices that the preconditions of the first action in plan are curre
- It then calls the planner to come up with a new subplan called repair that current situation to some state s on

a



- 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,

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

- 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 o Detect failure by checking the *preconditions for success* of the *entire remaining* plan o Useful when a goal is serendipitously achieved

 While you're painting the chair, someone comes painting the table with the same color

o 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

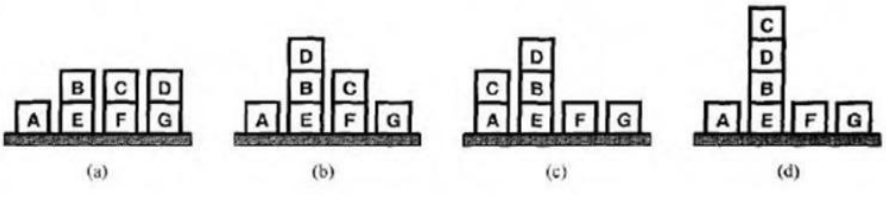
- If one insists on checking every precondition, it might never get around to actually doing anything
- RP - monitors during execution

3.4.4.3 Continuous Planning

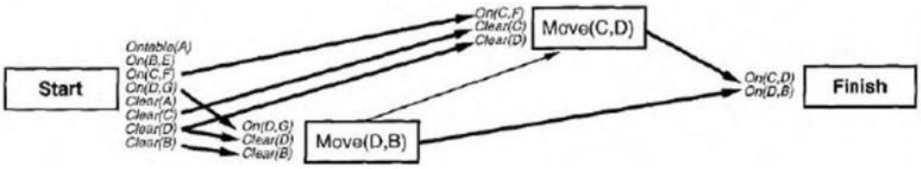
- Continuous planning agent o execute some steps ready to be executed o refine the plan to resolve standard deficiencies o refine the plan with additional information o fix the plan according to unexpected changes
  - recover from execution errors
  - remove steps that have been made redundant
- Goal ->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:- o 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) ∧Clear(y) ∧On(x ,z),EFFECT: On(x, y ∧Clear(z) ∧¬Clear(y) ∧¬On(x, z))

o Goal: On(C, D)∧On(D ,B)

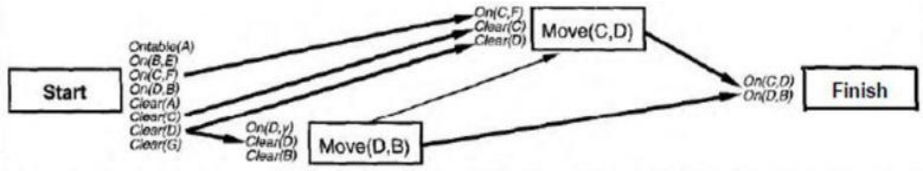
- *Start* is used as the label for the current state
- The following seven diagram shows the continuous planning agent approach towards the goal
- Plan and execution o Steps in execution:
  - Ordering - *Move(D,B),* then *Move(C,D)*
  - Another agent did *Move(D,B) - change the plan*
  - Remove the redundant step



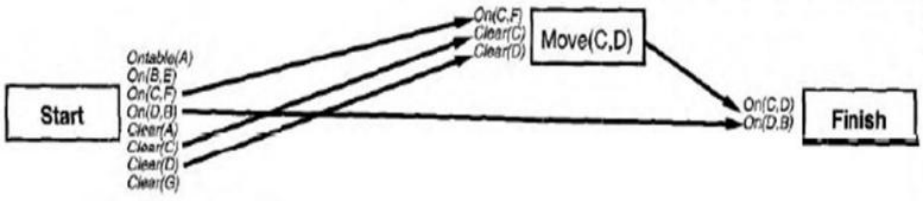
- The sequences of states as the continuous planning agent tries to *D On(D ,B) as*
- The start
- At (b), another agent has interfered,
- At (c), the agent has executed Move(C, D) but has failed,
- It retries Move(C, D), reaching the



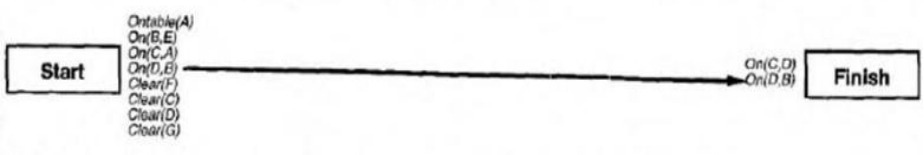
- The initial plan constructed by the continuous
- The plan is indistinguishable, so far, from that produced



- After someone else moves D onto B, the unsupported links supplying are dropped,

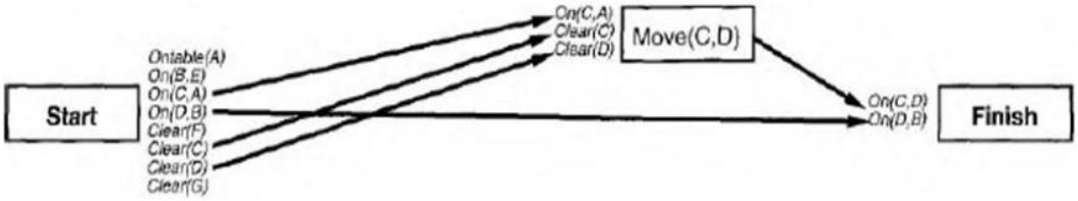


- The link Move(D, B) has been one from Start, and redundant step Move(D, B) has

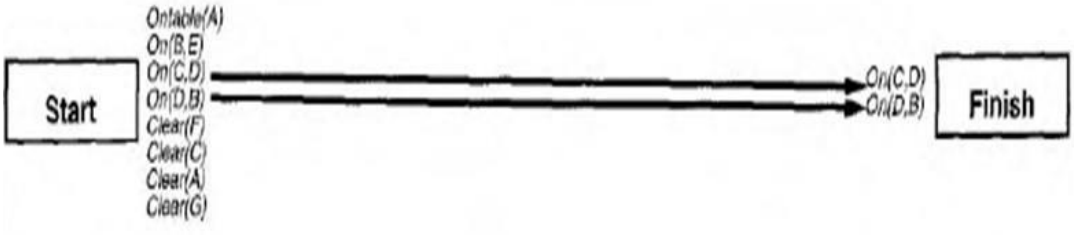


- Make a mistake, so *On(C,A)*
- Still one open condition
- *Planning one more time - Move(C,D)*
- Final state: start -> finish

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



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



- After Move(C, D) is executed dropped the plan, the remaining open On(C, D) is resolved by adding a causal link from the new start
- Now the plan is
- From this example, we can see that continuous planning is quite similar to
- On each iteration, the algorithm finds something about the plan that needs fixing a so- **plan** and
- The POP algorithm can be seen as a flaw-removal algorithm where the two flaws are preconditions and causal
- On the other hand, the continuous planning agent addresses a much broader range of foll
  - Missing
  - Open
  - Causal
  - Unsupporte
  - Redundant
  - Unexecuted
  - Unnecessary
- The following table shows the continuous algo POP-

Fun CONTINUOUS-POP-AGENT **re** an
S plan, a plan, initially with just Start,

Ac NoOp (the
EFFECTS [Start] = UPDATE(EFFECTS [Start],
REMOVE-FLAW (plan) // possibly updating
R a

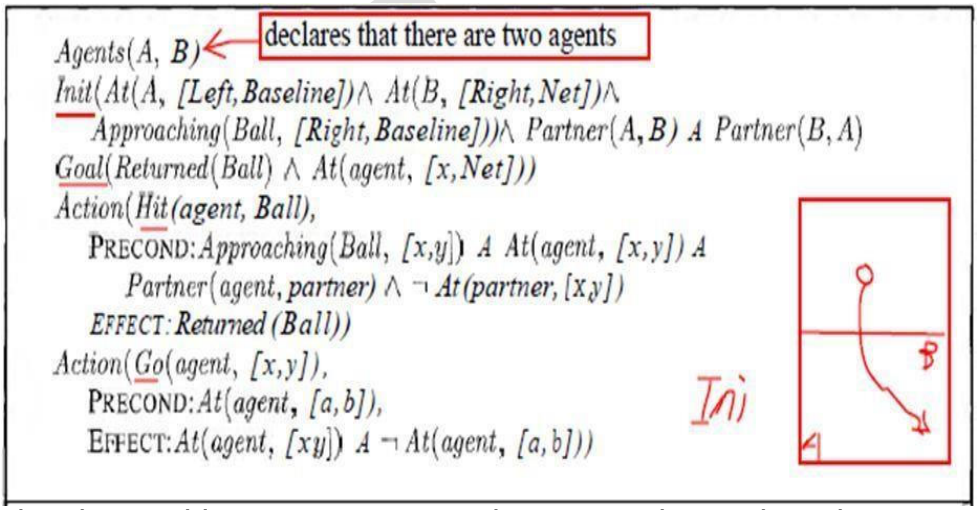
- It has a "perceive, remove "
- It keeps a persistent plan in its KB, and on each turn it removes one ,flaw from
- It then takes an action and repeats

- 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 **single-agent**
- Multiagent **coop** o **com**
- For
  - the problem is team planning in
- Plans can be constructed that specify actions for both
- Our objective is to construct plans
- To do this we need requires **coordin** possibly **commun**
- The following table shows the double



In the above table, Two agents are playing together and can be in one of four

- fo
  - [Left,
  - [Right,]
  - [Le]
  - [Rig]

- 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:

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

o 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:

o 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

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:

o 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:

All probabilities are between 0 and 1. $0 \leq P(A) \leq 1$

I. 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

II. The probability of a disjunction is given by $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$

III. Let $B = \neg A$ in the axiom (III)

IV. P(True) = P(A) + P(¬A) – P(False) (by logical equivalence)

V. 1 = P(A) + P(¬A) (by step 2) VII. P(¬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

SVCET

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

Conditional Probability

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

= = 0.80

Bayes'Rule: $\frac{0.04}{0.04+0.01}$

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:

   $P(Y / X) =$

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

   $P(Y / X,E) =$
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) = = 0.0002

7. Normalization

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

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

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

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

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

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

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

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

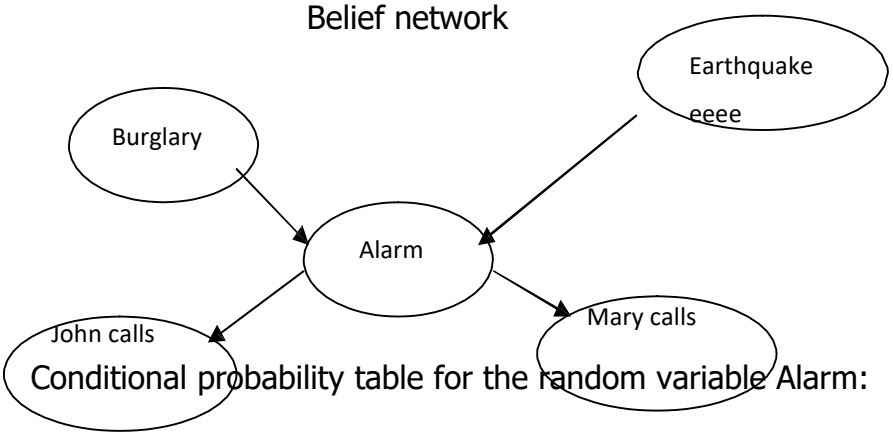
Uncertainty:

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.

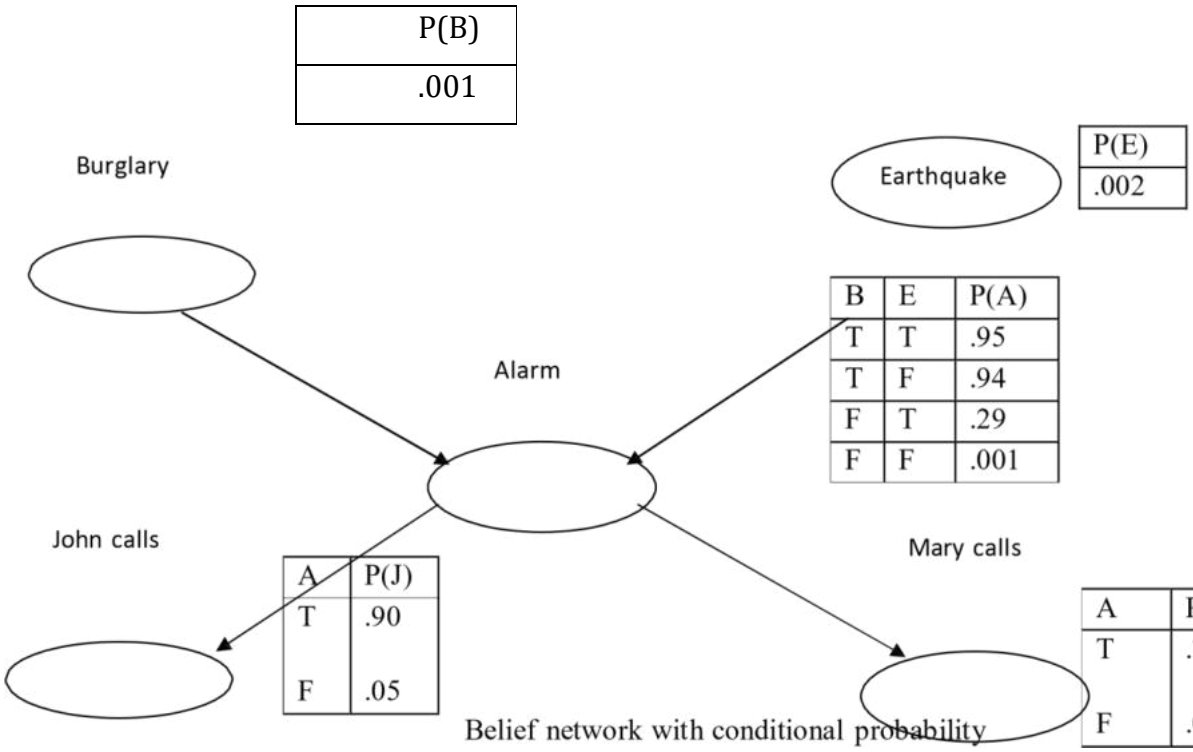
Belief network



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 probabilit y 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 $P(x_1.....x_n)$as an abbreviation for this.The value of this entry is given by the following formula:

  $P(x_1.....x_n) = \prod_{i=1}^{n} P(x_i|Parents(X_i))$

- 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.

  $P(J \wedge M \wedge A \wedge \neg B \wedge \neg E)$

  $= P(J/A)\ P(M/A)\ P(A|\neg B \wedge \neg E)\ P(\neg B)\ P(\neg E)$

  $= 0.90 * 0.70 * 0.001 * 0.999 * 0.998$

  $= 0.00062$

  Noisy OR: It is the logical relationship of uncertaint y.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:

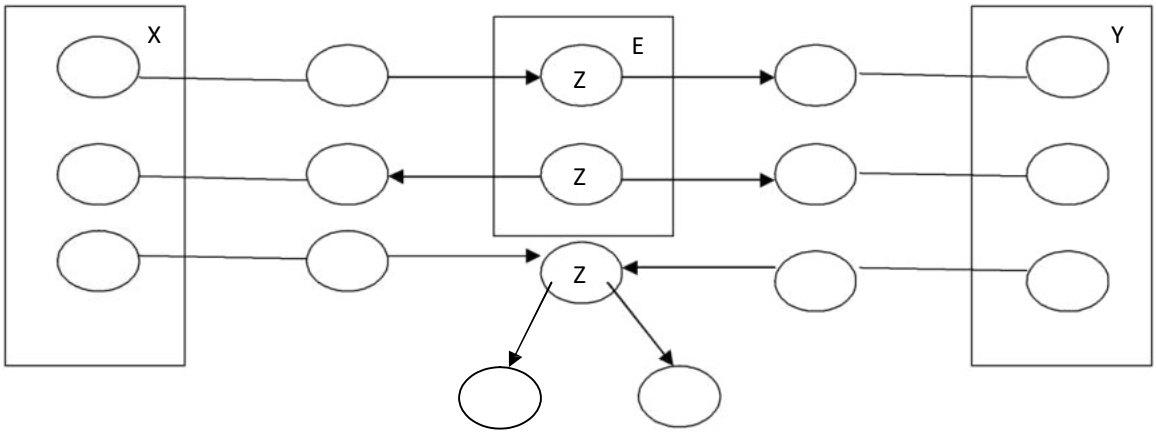
I. If no parent node is true then the output is false with 100% certainity.

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.



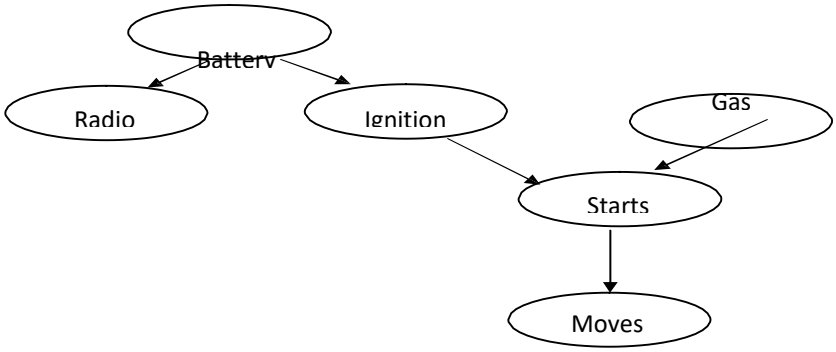
A path from X to Y can be blocked given evidence E

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

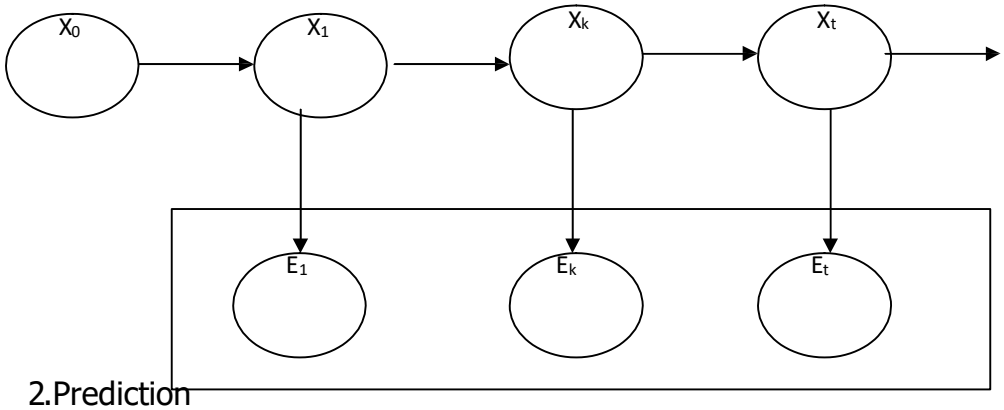
The generic temporal model has the following set of inference tasks:

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,

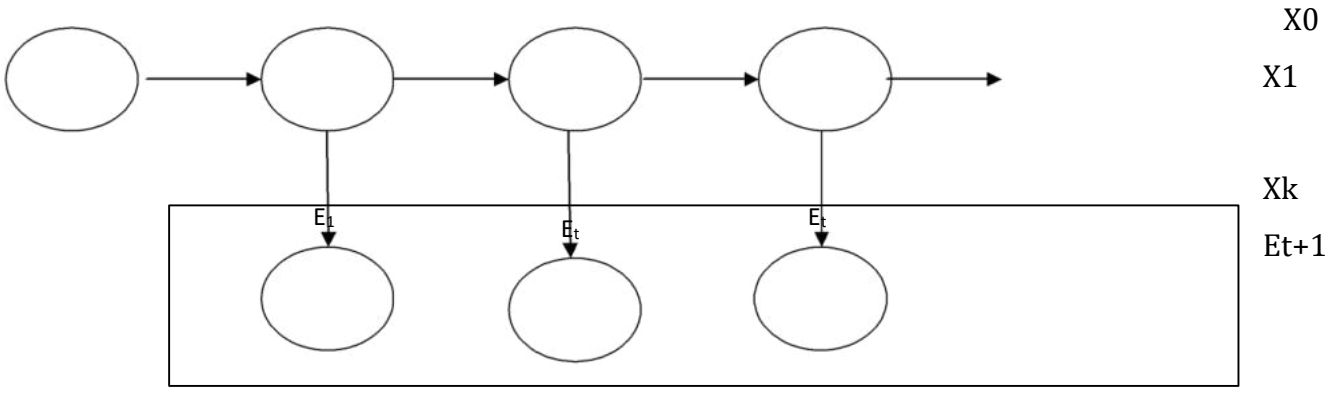
  given all the observation of the umbrella so far, including today



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

$$P(Xt+1|e1:t+1) = P(Xt+1|e1:t,et+1) = P(Xt+1|et+1,e1:t)$$

- 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.

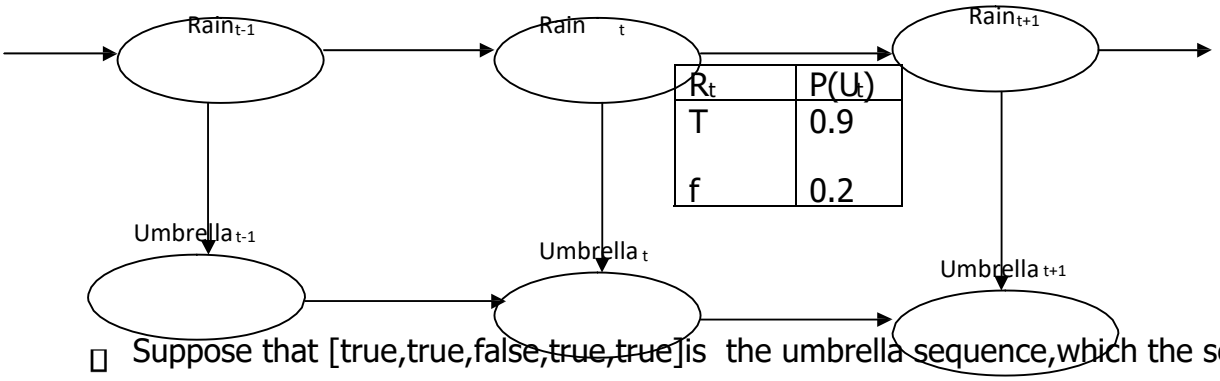
- Stationary distribution of the Markov process dooms to failure any attempt to predict the actual state for a number of steps ahead that is more than a small fraction of the mixing time.

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.

Rt-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 guar

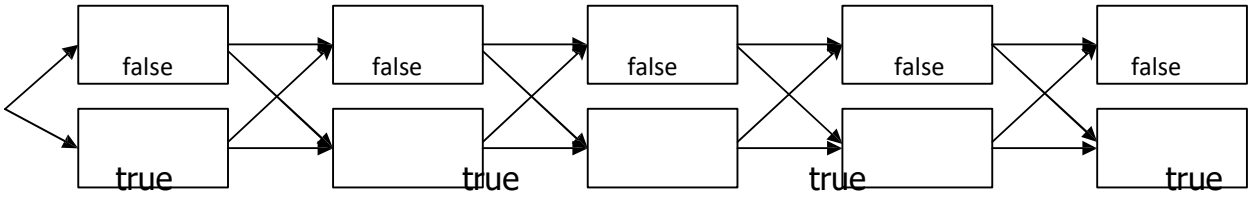
observes first five days on the job.

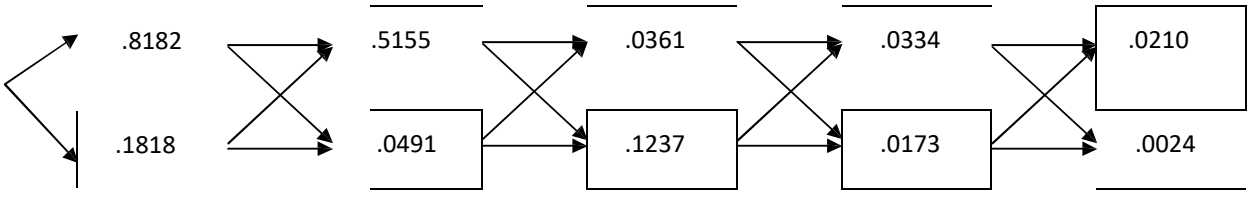
- What is the weather sequence most likely to explain this out of 2^5=32 possible sequences,i.e.

  argmax X1:t P(X1:t|e1: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.

Rain1 Rain2 Rain3 Rain4 Rain5

true true true true true





m1:1 m1:2 m1:3 m1:4 m1:5

Viterbi algorithm:

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

$$M1:t = \max_{X1,\ldots,Xt-1} P(X1,\ldots\ldots,Xt-1,Xt|e1:t)$$

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

$$m_{1:t+1} = \max_{X1,\ldots,Xt} P(X1,\ldots\ldots,Xt,Xt+1|e1:t+1)$$

$$= \max_{Xt}(P(X_{t+1}|X_t)\max_{X1,\ldots,Xt-1} P(X1,\ldots,Xt-1,Xt|e1:t))$$

$$=\alpha P(et+1|Xt+1)$$

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

  Tij = P(Xt = j|Xt-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.

$$O1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}$$

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

  The forward equation becomes

  $F1{:}t+1 = \alpha\, Ot+1\, T^T\, f1{:}t$ ……(1) and the backward equation becomes

  $bk+1{:}t = TOk+1\, bk+2{:}t$ ……(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.

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

- 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

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

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

  Bt-d+2:t+1 = Ot-d+1-1 T-1 Bt-d+1:t TOt+1. ……..(5)

- This equation provides an incremental update for the B matrix,which in turn(eqn (4))

  allows us to compute the backward message $b_{t-d+2:t+1}$.

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, o Which components of performance element are to be learned. o What feedback is available to make these components learn o What representation is used for the component.
- The components of these agents includes the following, o A direct mapping from conditions on current state to actions
  - A means to infer relevant properties of the world from the percept sequence o 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 o 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 o For Example: - An agent is training to become a taxi driver. o 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 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.
  o Supervised learning o Unsupervised learning o Reinforcement learning
- **Supervised Learning:-** o 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:-** o It is learning pattern, in which
- Correct answers are not given for the input.
- It is mainly used in probabilistic learning system.
- **Reinforcement Learning:-** o Here learning pattern is rather than being told by a teacher. o It learns from reinforcement (i.e.) by occasional rewards

o 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

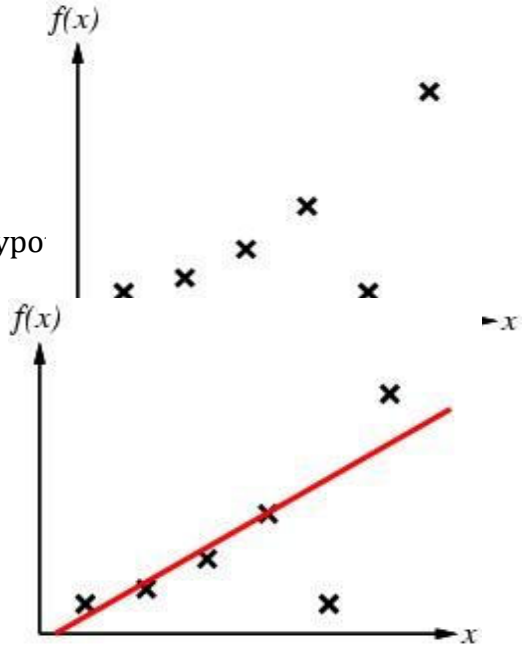
- Learn a function from example,
- For example:- *f* is target function
  An example is a pair (x, *f*(x)) where x = input and *f*(x) = output of the function is applied to x
- The pure inductive inference or induction is "given a training set of example of *f,* return a function *h* that approximates *f.*
- Where the function *h* is called hypothesis

- This is a simplified model of real learning, because it o Ignores prior knowledge o Assumes a deterministic, observable “environment”.
- A good hypothesis will generalize well, i.e., able to predict based on unseen examples

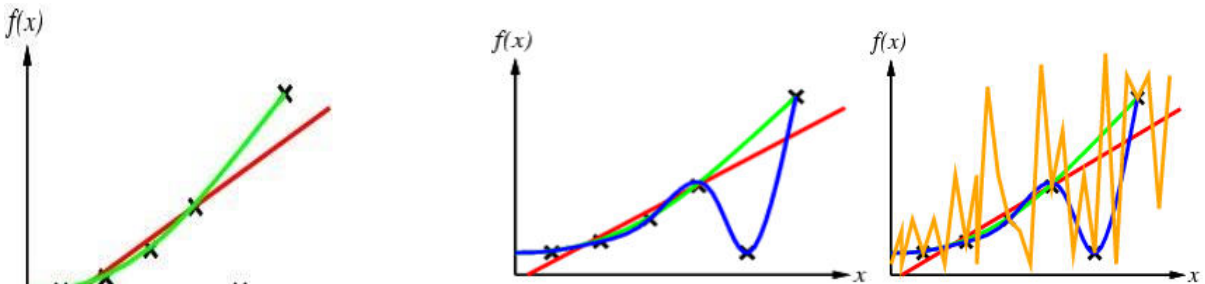
5.2.1 Inductive learning method:-

- 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 fitti ng example
- Given

- Linear hypo



- 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:- o A decision tree for deciding whether to wait for a table at a restaurant. o The aim here is to learn a definition for the **goal predicate.**

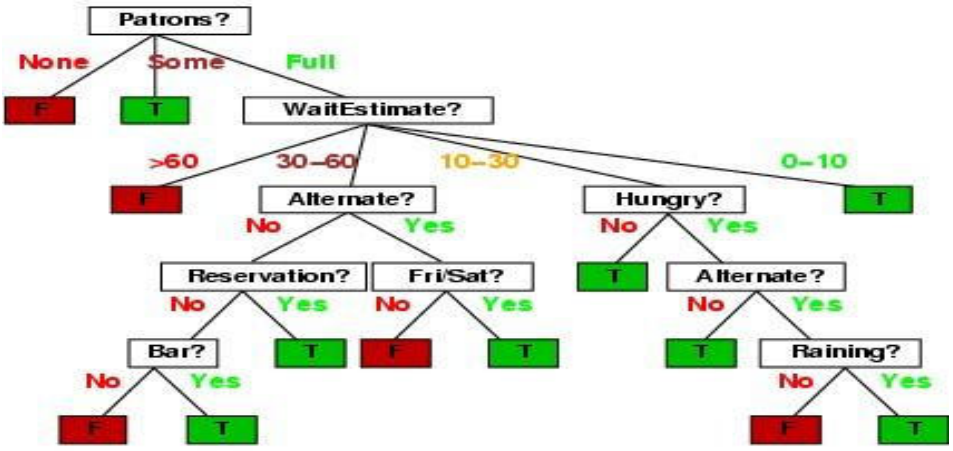
  o we will see how to automate the task the following attributes are decided.

  - 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



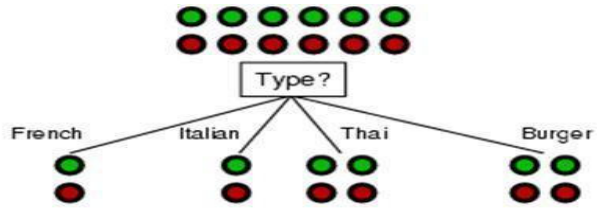
The above decision tree does not use price and type as irrelevant.

♣ 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

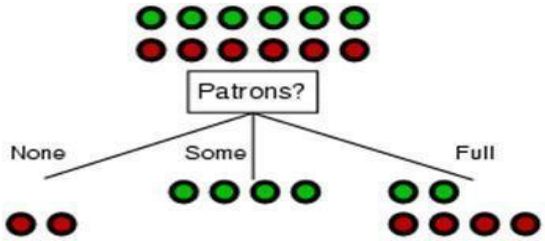
♣ Classification of example is positive (T) or negative (F) shown in both table and in

decision tree.

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 and negative
- The below diagram Splitting on Patrons does a good job of separating positive and exa



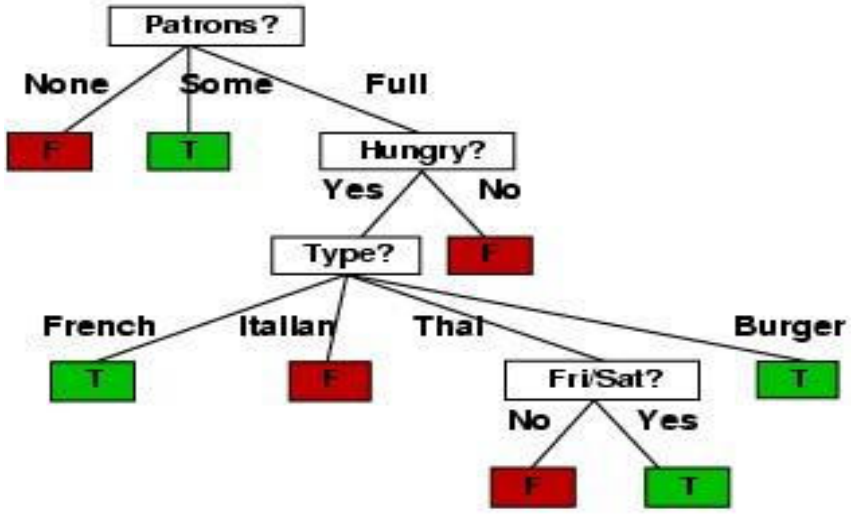
- The following table shows the Decision Tree Learning

```
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 ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value v_i of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← 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

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] :
- $I(P(v_1), \ldots, P(v_n)) = \sum_{i=1}^{n} -P(v_i) \log_2 P(v_i)$
- 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 $E1, \ldots, Ev$ according to their values for A, where A has v distinct values

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

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

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

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

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

- 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]

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

$$IG(Type) = 1 - \left[\frac{2}{12} I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{2}{12} I\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{4}{12} I\left(\frac{2}{4}, \frac{2}{4}\right) + \frac{4}{12} I\left(\frac{2}{4}, \frac{2}{4}\right)\right] = 0 \text{ bits}$$

- Patrons has the highest IG of all attributes and so is chosen as the root

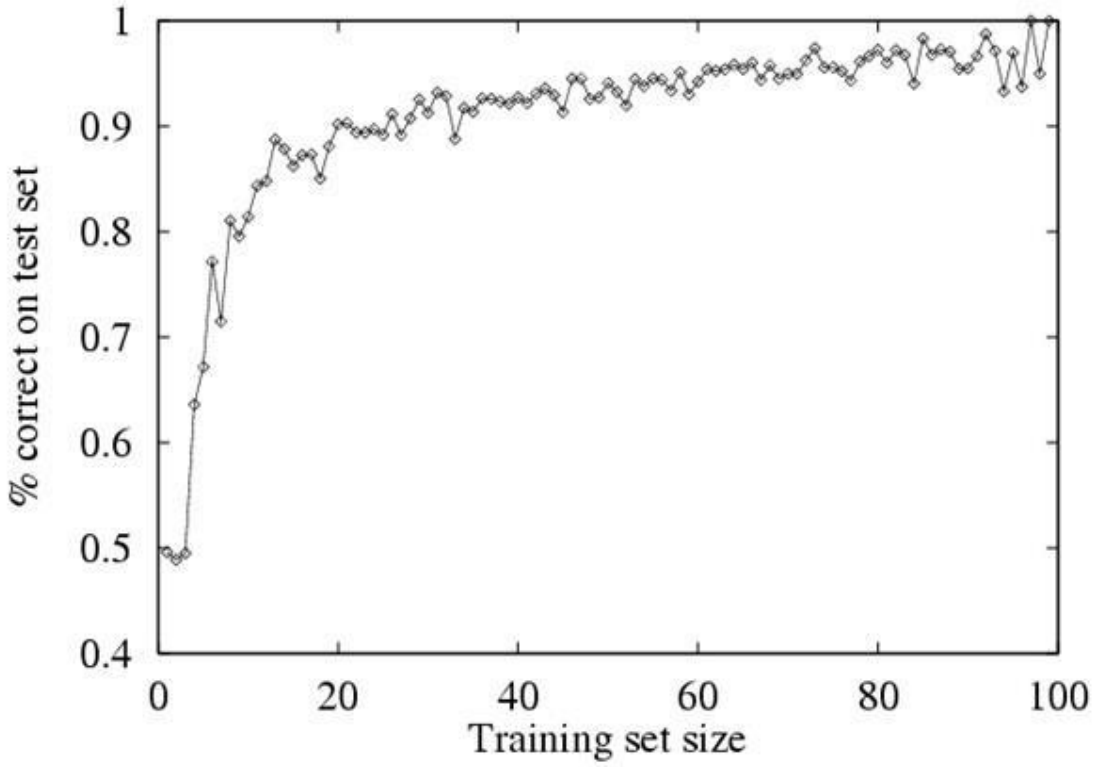
5.3.5 Assessing the performance of the learning Algorithm:

- 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, o The training example o A Goal concept o An Operationality Criteria o A Domain theory
- An **EBL** accepts four kinds of input as follows, o **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 o **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:-** o 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 |= Classification
- 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
        - $Rewrite(u, v) \wedge Simplify(v, w) \Rightarrow Simplify(u, w)$
        - $Primitive(u) \Rightarrow Simplify(u, u)$
        - $ArithmeticUnknown(u) \Rightarrow Primitive(u)$
        - $Number(u) \Rightarrow Primitive(u)$
        - $Rewrite(1\ x\ u, u)$
        - $Rewrite(0\ x\ u, u)$

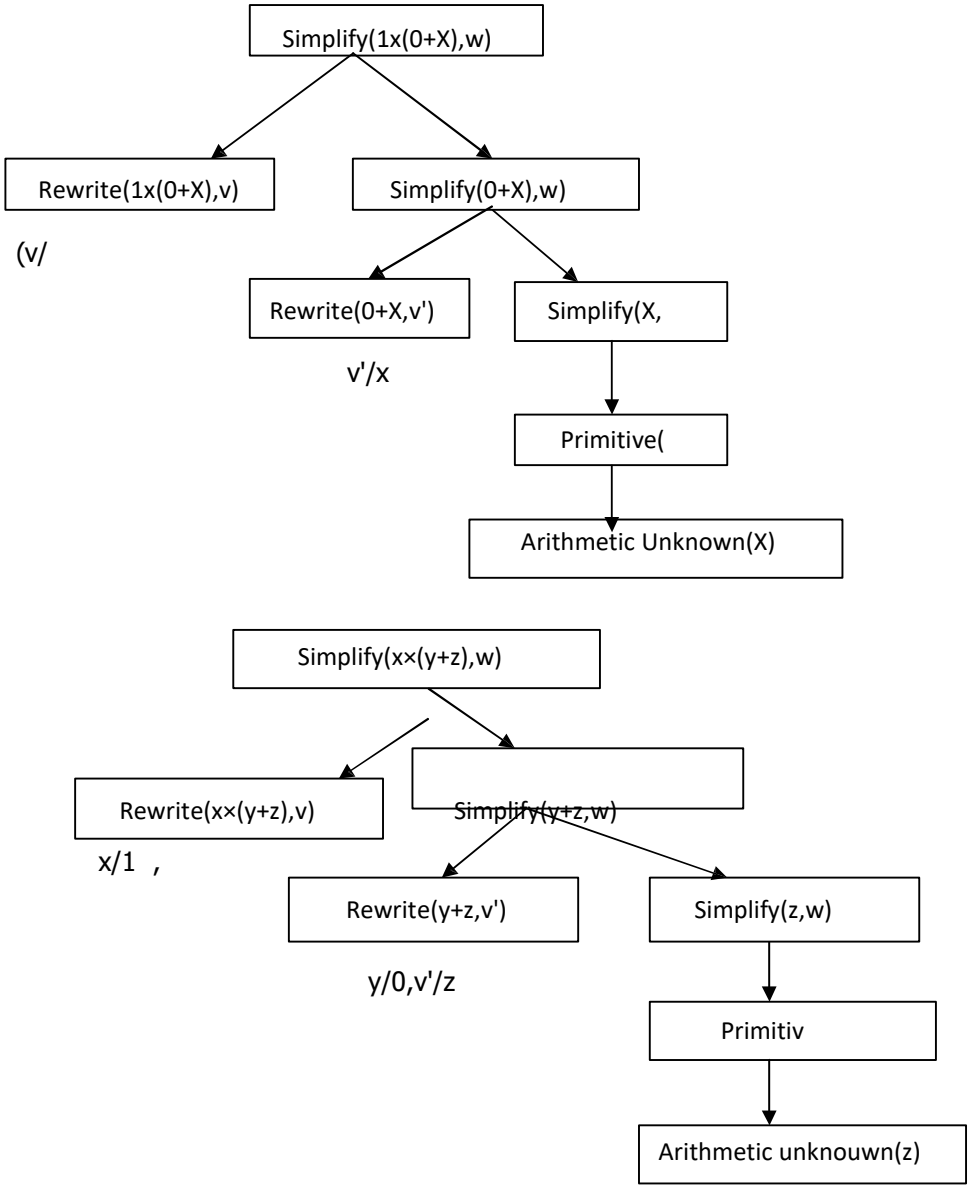
EBL Process Working

- 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 o *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, o *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, o *Simplify (y + z, w),* yielding the rule o *Simplify (y + z, w)* *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, o 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.

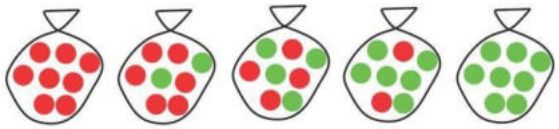
5.5.1 Statistical Learning

- 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:-** o 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 |d) = a P(d | h)P(h)

  i *i* *i*
- For prediction about quantity X :

  P(X|d)= ∑ P(X|d,h)P(h |d)= ∑ P(X|h)P(h |d)

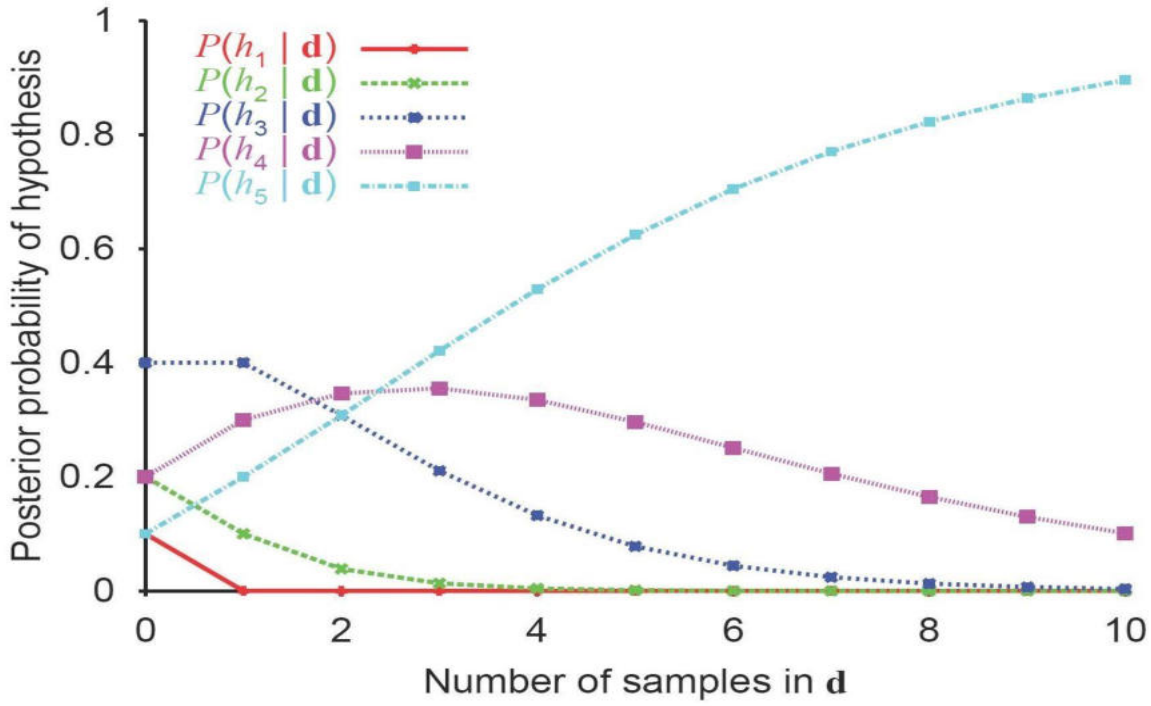
  i *i* *i* *i*
- Where it is assumed that each hypothesis determines a probability distribution over **X.**
- This equation shows that predictions were weighted averages over the predictions of the individual hypothesis
- The key quantities in the Bayesian approach are the o Hypothesis ***Prior,*** P(h_i)

  o **Likelihood** of the data under each hypothesis, P(d | h)

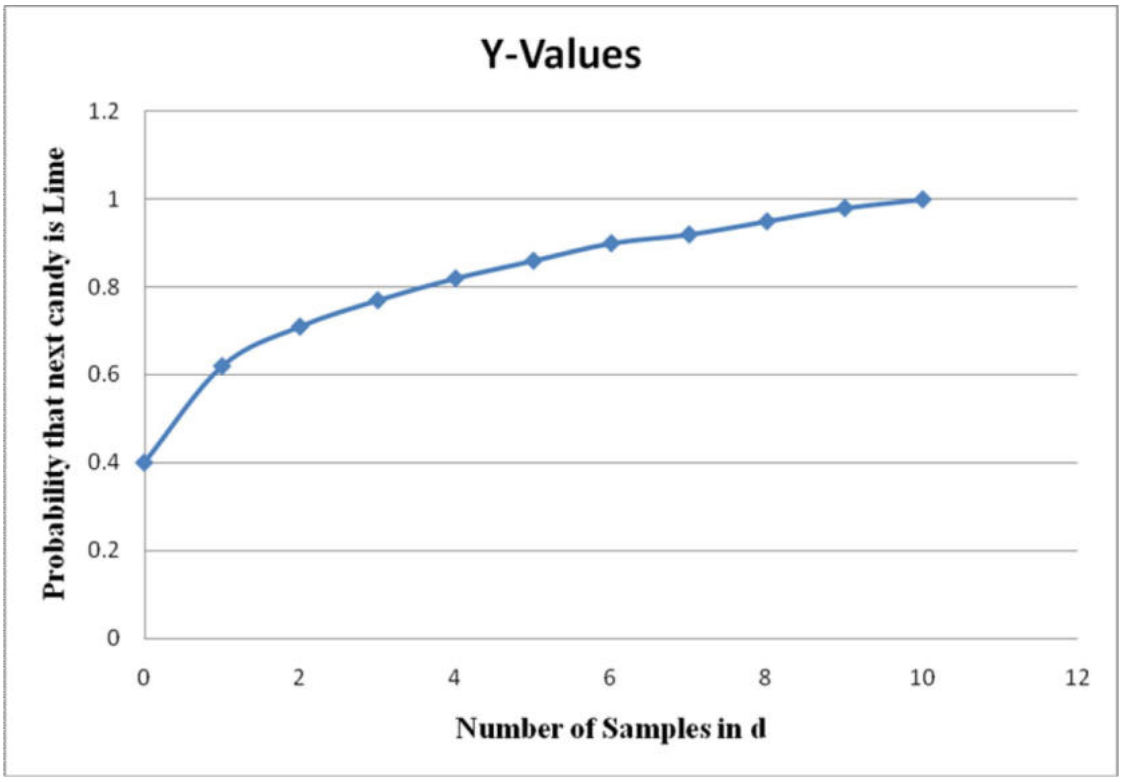
  i
- For candy example, assume the time being that the prior distribution over h1,….h5 is given by (0.1,0.2,0.4,0.2,0.1), as advertised by the manufacturer.
- The likelihood of the data is calculated under the assumption that the observations are i..i..d, that is i= independently, i= identically and d= distributed So that

$$P(d \mid hi) = \prod_j P(dj \mid hi)$$

- 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



5.1.2 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) 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 probabilit y model.
- The structure of the model is fixed.

- Data are complete when each data point contains values for every variable in the probabilit y 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 , which is the proportion of cherry candies, and the hypothesis is *h* .
- The proportion of lime is (1-).
- 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 `.
- 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 / hθ) =∏P(dj \ hθ) =θc .(1–θ)LJ =1
- The maximum-likelihood hypothesis is given by the value of e` that maximizes the expression.
- It can be obtained by maximizing the log likelihood.
  NL1(d | hθ) =P(d | hθ) =∑log(P(dj \ hθ) =clogθ+L log(1–θ)J =1
- To find the ML value of θdifferentiate wrt θand then equate resulting to zero

$$\frac{JL(d\backslash h\theta)}{d\theta} = \frac{c}{\theta} - \frac{L}{1-\theta} \quad , \quad \frac{c}{\theta} - \frac{L}{1-\theta} = 0, \ \theta - \frac{c}{c+l} \quad , \quad \theta - \frac{c}{N} \quad \text{where } c+l = N$$

The standard method for maximum likelihood parameter learning is given by

o 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

$$\frac{\partial L}{\partial \mu} = \frac{1}{\sigma^2} \sum_{j=1}^{N} (x - \mu) = 0 \qquad \Rightarrow \mu = \frac{\sum_j x_j}{N}$$

$$\frac{\partial L}{\partial \sigma} = \frac{N}{\sigma} + \frac{1}{\sigma^3} \sum_{j=1}^{N} (x_j - \mu)^2 = 0 \qquad \Rightarrow \sigma = \sqrt{\frac{\sum_j (x_j - \mu)^2}{N}}$$

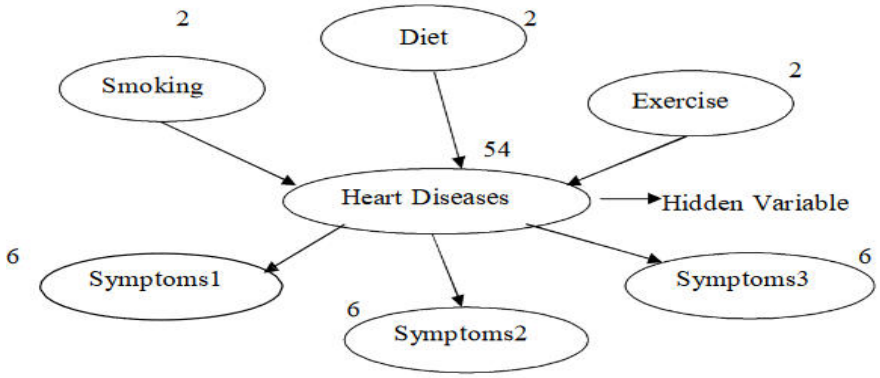
- Maximum likelihood value of the mean is the simple average.
- Maximum likelihood value of the standard deviation is the square root of the simple variance.

5.5.3 Learning with Hidden Variables:

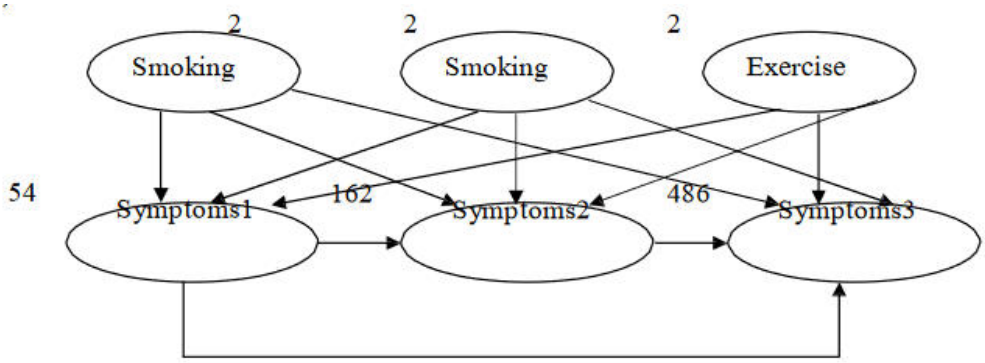
1. Many real world problems have hidden variables (or) latent variables which are not observable in the data that are available for learning.

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.

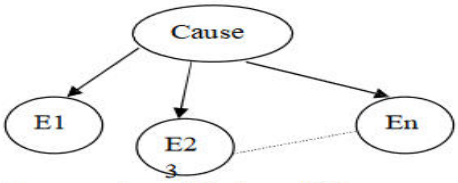
Assumed the diagnostic model for heart disease. There are three observable predisposing factors and 3 observable symptoms. Each variable has 3 possible values (none, moderateand severe)



If hidden is removed the total number of parameters increases from 78 (54 + 2 + 2 + 2 + 6 + 6 + 6) to 708



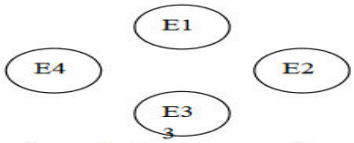
3. Hidden variables can dramatically reduce the number of parameters required to specify the Bayesian network, there by reduce the amount of data needed to learn the parameters.
4. It also includes estimating probabilities when some of the data are missing.
5. The reason we learn Bayesian network with 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.
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.



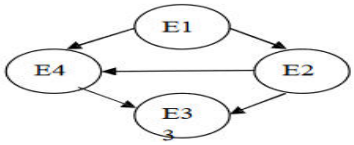
This will have O(n) parameters, because the evident variables are conditionally independent given the causes.

8. Missing data
   - Imagine that we have 2 binary variables A and B that are not independent. We try to estimate the Joint distribution.

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.



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 wayto 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

	$\underline{0}$ $\bar{A}$	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,

	$\underline{0}$ $\bar{A}$	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

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.

	$\underline{0}$ $\bar{A}$	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 6th 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.

$\Pr(B / \neg A, \theta 0) = \Pr(\neg A, B / \theta 0) / \Pr(\neg A / \theta 0)$

= 0.25/0.50.5

H = 0 probabili ty is 0.5

H = 1 probabili ty is 0.

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

	0 — A	1 A
0 B	3.5/ 8	1/ 8
1 B	1.5/ 8	2/ 8

	0 — A	1 A
0 B	0.437 5	0.12 5
1 B	0.187 5	0.25

New estimate is

Pr(H / D, $Q1$) Pr(, B / $Q1$) / Pr(A / $Q1$)

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 $\theta 2$ is

$\Pr(H / D,\theta 2) = \Pr(\neg Ar, B / \theta 2) / \Pr(\neg A / \theta 2) = 0.1625 = 0.260.625 \log$

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 ty 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 ty 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 P fixed and optimizing
   - ✓ and repeat the procedure over and again d. g has some local and global optima as PR(D/ θ)

e. **Example:-**

i. Pick initial 0

ii. Probability of hidden variables given the observed data and the current model.
   Loop until it converges

   P%t + 1(H) = Pr(H / D,θt)arg max

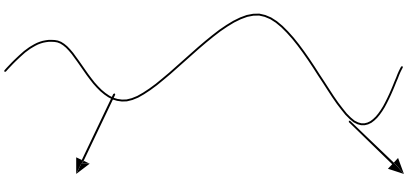
   P%t + 1 =θ ErP%t +1 log Pr(D, H / θ)

   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**

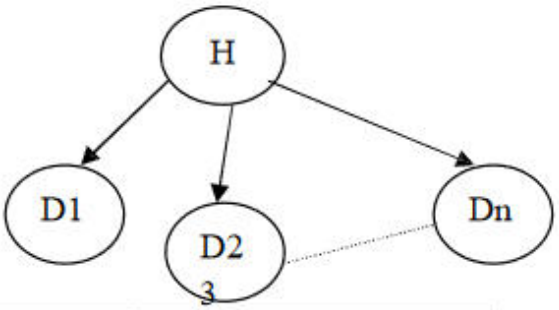
1. Fill in the data set with distribution over values for hidden variables
2. Estimate Conditional probability using expected counts.

   We will compute the probability distribution over H given D and theta ( ), we have 'm' different hidden variables, one for the value of node H in each of the m data cases.

   $P\%t+1(H)=Pr(H/D,\theta t)=\prod Pr(H/D,\theta t)\ m$
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



0_1	0_2	.....	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.

$$E(H) = \sum_m \Pr(H^m / D^m, \theta_t)$$
$$= 1.9$$
$$E(H \wedge D2) = \sum_m \Pr(H^m / D^m, \theta_t) I(D_2^m)$$
$$= 0.9 + 0.2 + 0.2 + 0.5$$
$$= 1.8$$
$$\Pr(D2 / H) = \frac{1.8}{1.9} \text{ Probability of D2 given H}$$
$$= 0.9473$$

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:-** o In the first experiment (**IB1**), to learn a concept simply required the program to store every example. o 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:-** o 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. o 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:-** o 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. o 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.

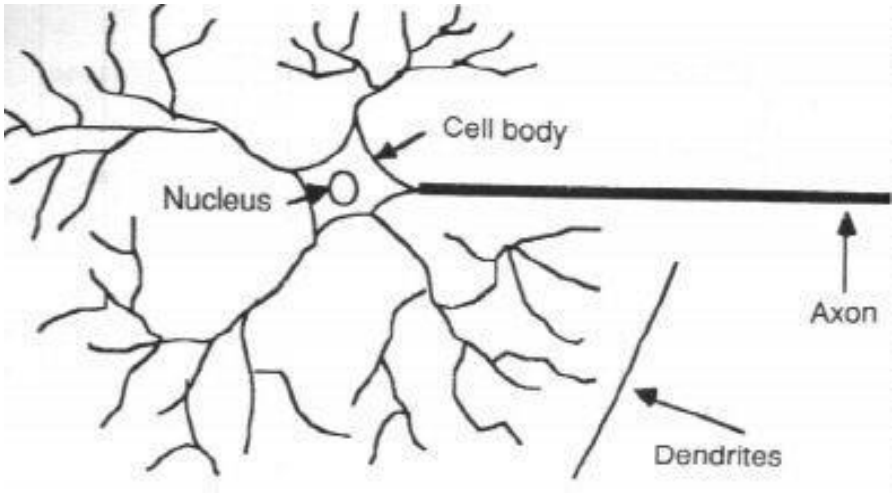
5.5.5 Neural Network:-

- 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.

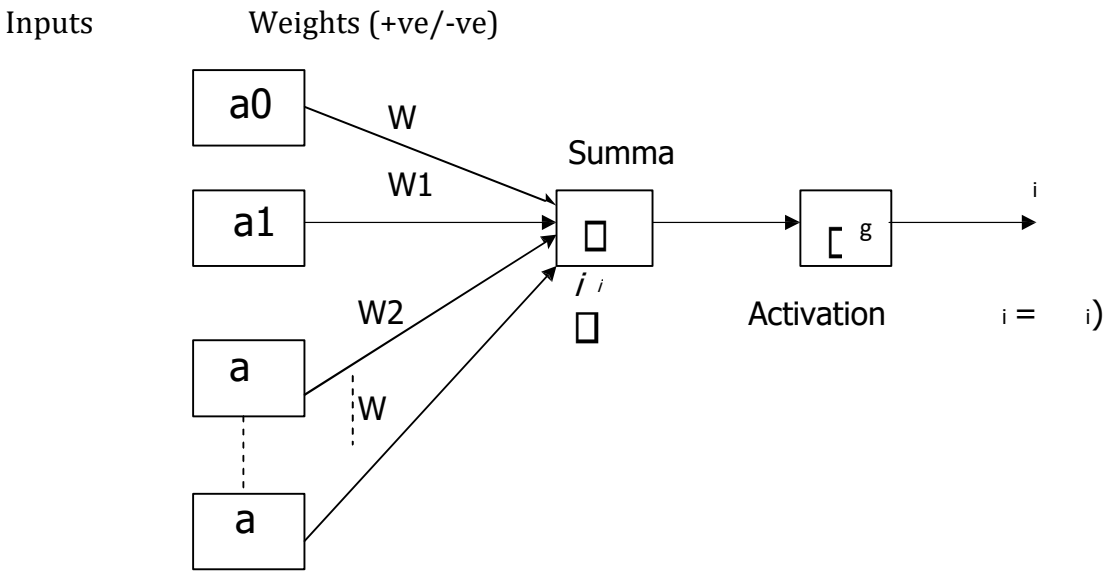


- Each neuron is a cell that uses biochemical reactions to receive, process and transmit information.
- A neurons dendrites tree is connected to a thousand neighboring neurons.
- When one of those neurons fire, a positive or negative charge is received by one of the dendrites.
- The strengths of all 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

$$in_i = \sum W_{j,i} a_j = 0$$

- Then it applies an activation function g to this sum to derive the output.

n

ai =g (ini) =g (∑Wj, iaj)

- 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	105	1014

- 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
nulation 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.

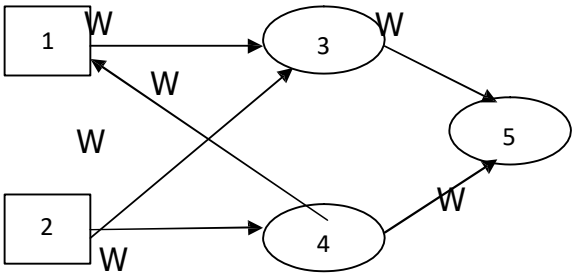
  o Acyclic (or) Feed-forward network
    - Single layer feed-forward network
    - Multilayer feed-forward network o Cyclic (or) Recurrent networks
- The following table shows the difference between Feed-forward network and Recurrent network,

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:-

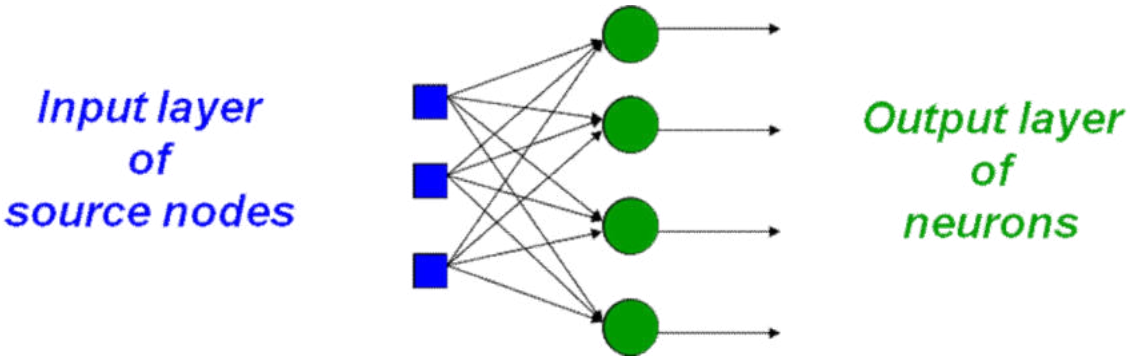
- A feed-forward network represents a function of its current input; thereby it has no internal state other than the weights themselves.
- Consider the following network, which has two hidden input units and an output unit.



- Given an input vector x = (x1, x2), the activations of the input units are set to (a1,a2) = (x1,x2) and the network computes
  $a5 = g(W3,5a3 + W4,5a4) = g(W3,5\, g(W1,3a1 + W2,3a2) + W4,5\, g(W1,4\, a1 + W2,4a2))$

Single Layer feed-forward network:-

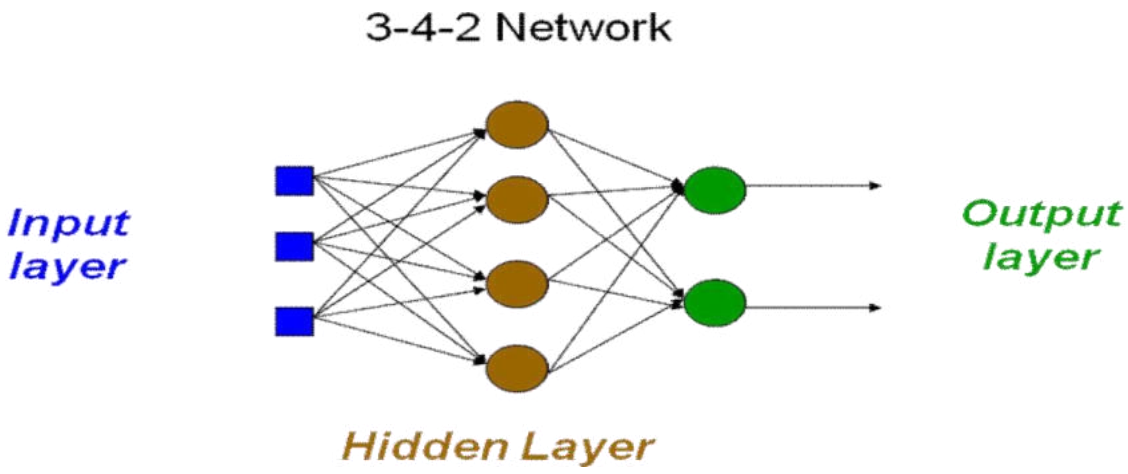
- A single layer network has one layer of connection weights.
- The following figure shows the single layer feed forward network.



- 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.

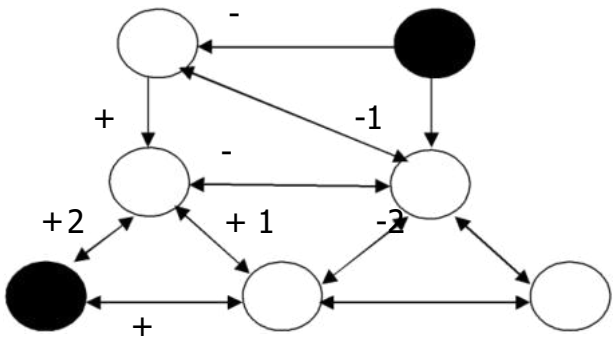
Multi Layer feed-forward network:-

- 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:-** o 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:-** o 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:-** o 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:-** o It is an algorithm, which is proposed for growing a larger network from a smaller one. o it resembles decision-list learning.

  o 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:- o In chess game, the reinforcement is received only at the end of the game.

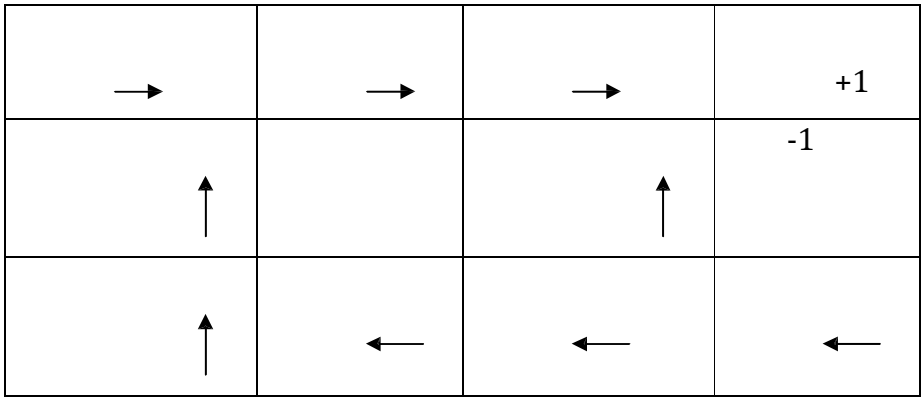
o In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement.

- 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 o Utility-based agent o Q-Learning agent o Reflex agent
- The following are the **Types of Reinforcement Learning,** o **Passive Reinforcement Learning** o **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 is fixed (i.e.) in state s, it always executes the action
  (s).
- Its goal is simply to learn the utility function U (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
  o Neither the transition model T(s, a,s'), which specifies the probabilit y of reaching state's from state s after doing action a;
  o 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 □ .
- 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 → (1, 2)-0.4 → (1,3)-0.4 → (1,2)-0.4 → (1,3)-0.4 → (2,3)-0.4 → (3,3)-0.4 → (4,3)+1
(1 ,1)-0.4 → (1, 2)-0.4 → (1,3)-0.4 → (2,3)-0.4 → (3,3)-0.4 → (3,2)-0.4 → (3,3)-0.4 → (4,3)+1
(1 ,1)-0.4 → (2, 1)-0.4 → (3,1)-0.4 → (3,2)-0.4 → (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 □ (s) associated with each nonterminal state s.
- The utility is defined to be the expected sum of (discounted) rewards obtained if policy is □ followed, the utility function is written as

$$U \prod(s) = E\left| \sum \gamma t R(st) \right| \prod, s0 = s$$

For the 4 x 3 world set $\gamma = 1$

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:-** o 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

$$U\pi(s) = R(s) + \lambda \sum T(s, \pi(s), s`) U\, \pi(s`) s`$$

o It misses opportunities for learning

Reason:- It ignores the connections between states o 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$N_{sas'}[s,a,s']$

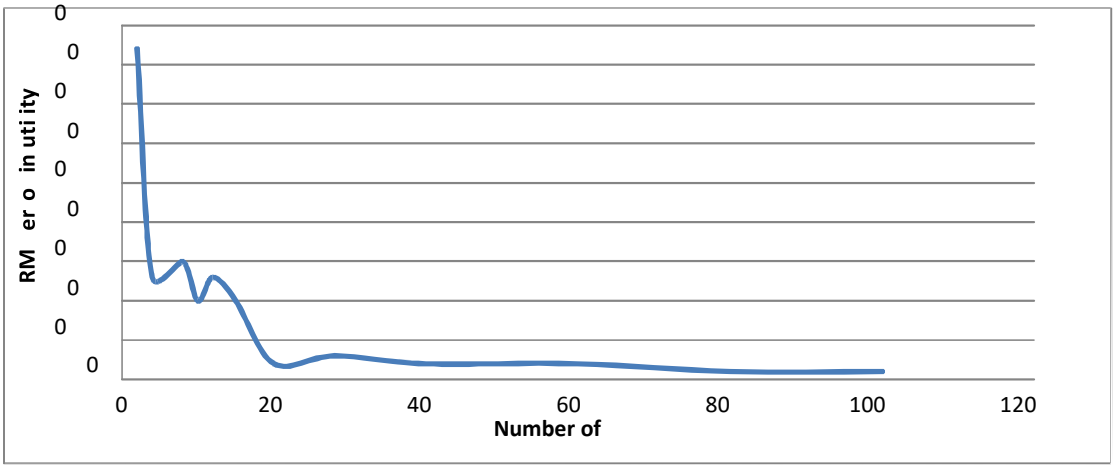
For each t such that $N_{sas'}[s,a,t]$is nonzero **do**

$T[s,a,t] \leftarrow N_{sas'}[s,a,t]/N_{sa}[s,a]$

$U \leftarrow$VALUE-DETERMINATION(π,U,mdb)

If TERMINALS?[s′]**then** s,a←null else s,a←s′,π[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:-** o 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:-** o It is intractable for large state spaces

5.6.2.3 Temporal Difference Learning:-

- In order to approximate the constraint equation Uπ(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 Uπ(S)

U π(S) ←U π(S) +α(R(S) +αU π(S1) –U π(S))

- Where = 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:-** o It is much simpler
  - It requires much less computation per observation
- **Disadvantages:-** o It does not learn quite as fast as the ADP agent o 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, o Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. o 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.

o 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. o 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:-** o 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.

o 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.*

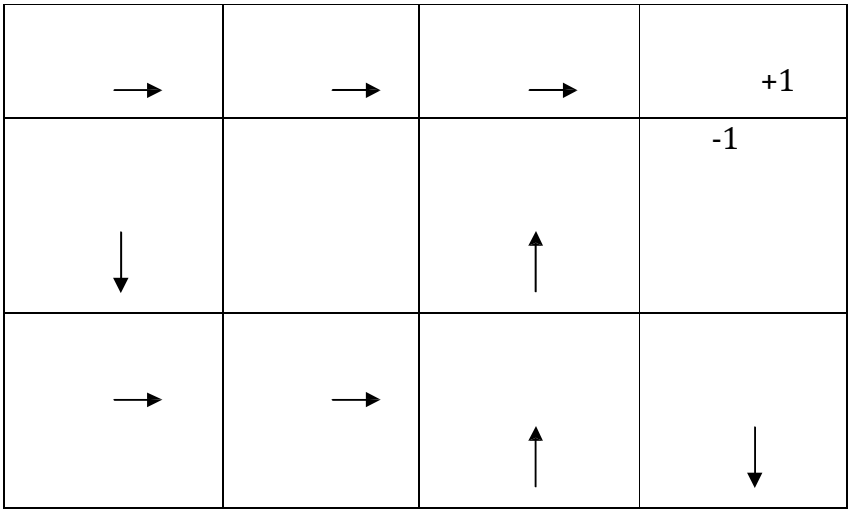
$$U(s) = R(s) + \gamma \max \sum T(s,a, s`)U(s`)s`$$

o These equations can be solved to obtain the utility function U using he value iteration or policy iteration algorithms. o 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;

o 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, o 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 o 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 utilit y 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,

- $U^+(s) \leftarrow R(s) + \gamma \max f\left[\sum T(s, a, s`) U^+(s`), N(a, s)\right]$
- 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:-** o 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. o 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. o 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 Q(a, s*a*

- Q Learning is called a model free method.
- **Reason:-** o It has a very important property: a TD that learns a Q-function does not need a model for either learning or action selection.

o As with utilities, a constraint equation can be written that must hold at equilibrium when the Q-Values are correct,

$$Q(a, s) = R(s) + \gamma \sum T(s, a, s`) \max Q(a`, s`)$$

o 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.

o This does, however, require that a model also be learned because the equation uses T(s, a, Sf).

o The temporal difference approach, on the other hand, requires no model.

o The update equation for TD Q-Learning is

$$Q(a, s) \leftarrow Q(a, s) + \alpha[R(s) + \gamma \max Q(a`, s`) - Q(a, s)$$

o Which is calculated whenever action a is executed in state s leading to state Sf.

o 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 Nsa[s,a]

Q[a,s]←q[a,s] + α(Nsa[s,a])(r + γ maxa' Q[a',s'] – Q[a,s])

If TERMINAL?[s']then s,a,r←null

Else s,a,r←s',argmaxa' f(Q[a',s'],Nsa[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**
- Uθ(S) =θ1 f1 (S) +θ2 f2 (S) + ... +θnfn (S
- The reinforcement learning can learn value for the parameters θ=θ1.....θn.
- Such that the evaluation function $U_{[2]}$ 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 Uθ(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.

$$E\,j\,(S\,) = (U\,(S\,) - U\,(S\,))2\,/\,2$$

- The rate of change of error with respect to each parameter θi is 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:-** o It requires less space. o Function approximation can also be very helpful for learning a model of the environment.
  o It allows for inductive generalization over input states.
- **Disadvantages:-** o The convergence is likely to be displayed. o It could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well.

o Consider the simplest case, which is direct utility estimation. With function approximation, this is an instance of supervised learning.