Subject Code : CS 2351 **Subject : Artificial Intelligence**

CS2351 ARTIFICIAL INTELLIGENCE

UNIT I PROBLEM SOLVING

Introduction – Agents – Problem formulation – uninformed search strategies – heuristics – informed search strategies – constraint satisfaction

UNIT II LOGICAL REASONING

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

UNIT III PLANNING

Planning with state-space search – partial-order planning – planning graphs – planning and acting in the real world

UNIT IV UNCERTAIN KNOWLEDGE AND REASONING

Uncertainty – review of probability - probabilistic Reasoning – Bayesian networks – inferences in Bayesian networks – Temporal models – Hidden Markov models

UNIT V LEARNING

Learning from observation - Inductive learning – Decision trees – Explanation based learning – Statistical Learning methods - Reinforcement Learning

TEXT BOOK:

1. S. Russel and P. Norvig, “Artificial Intelligence – A Modern Approach”, Second Edition, Pearson Education, 2003.

REFERENCES:

1. David Poole, Alan Mackworth, Randy Goebel, ”Computational Intelligence : a logical approach”, Oxford University Press, 2004.
2. G. Luger, “Artificial Intelligence: Structures and Strategies for complex problem solving”, Fourth Edition, Pearson Education, 2002.
3. J. Nilsson, “Artificial Intelligence: A new Synthesis”, Elsevier Publishers, 1998.

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.

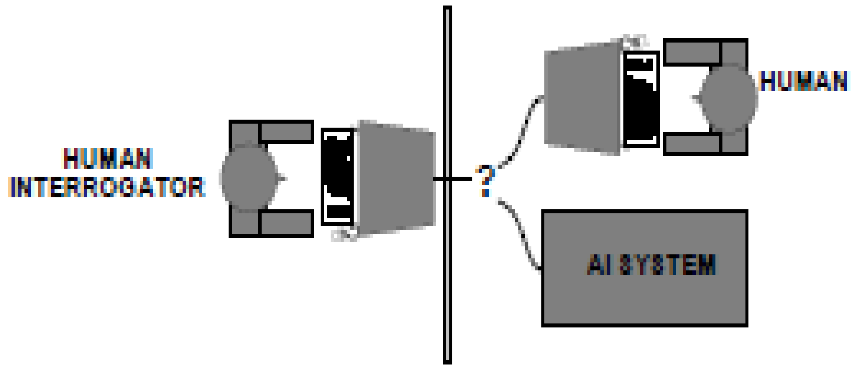
A system is **rational** if it does the "right thing," given what it knows.

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.

<u>Foundation of Artificial Intelligence</u>

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	10^{10} bits RAM	10^{11} neurons
	10^{11} 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	10^9	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

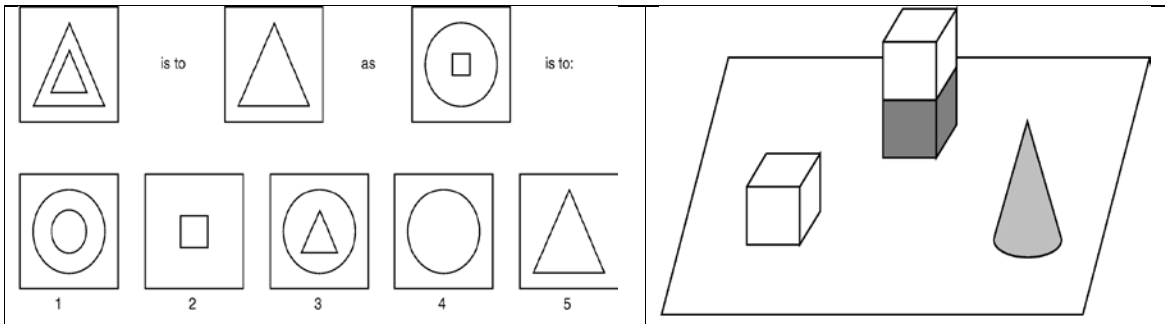


Fig : The Tom Evan's ANALOGY program could solve geometric analogy problems as shown.

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.

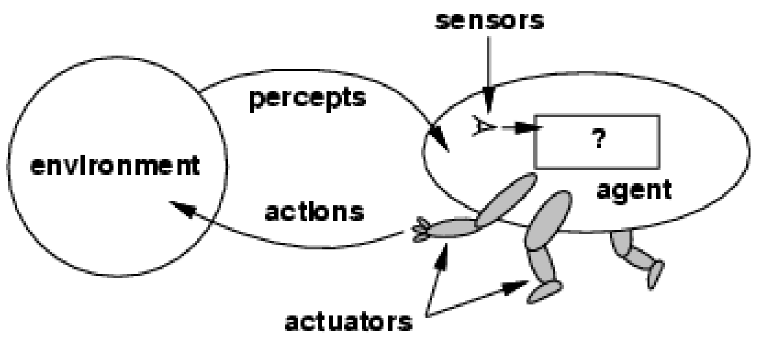


Fig : Agents interact with environments through sensors and effectors (accuators)

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.

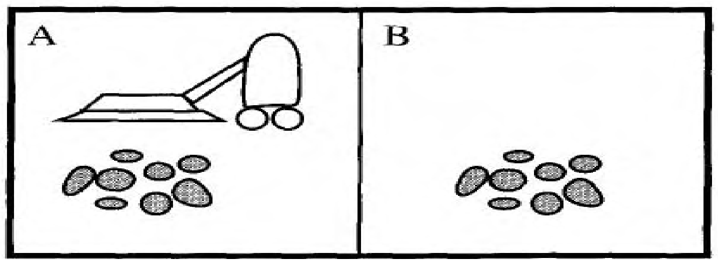


Fig : A vacuum-cleaner world with just two locations

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 t*o* 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 bin	Conveyor belt with parts, bins	Jointed arm and hand	Camera, joint angle sensors
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 categorization 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.

Environment Characteristics

Examples of task environments and their characteristics

Task Environment	**Observable**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Agent**
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Stochastic	Sequential	Semi	Discrete	Multi
Poker	Partially	Stochastic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi dnving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking	Partially	Stochastic	Episodic	Dynamic	Continuous	Single

robot						
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

- The simplest environment is
  - Fully observable, deterministic, episodic, static, discrete and single-agent.
- 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 initiate-braking

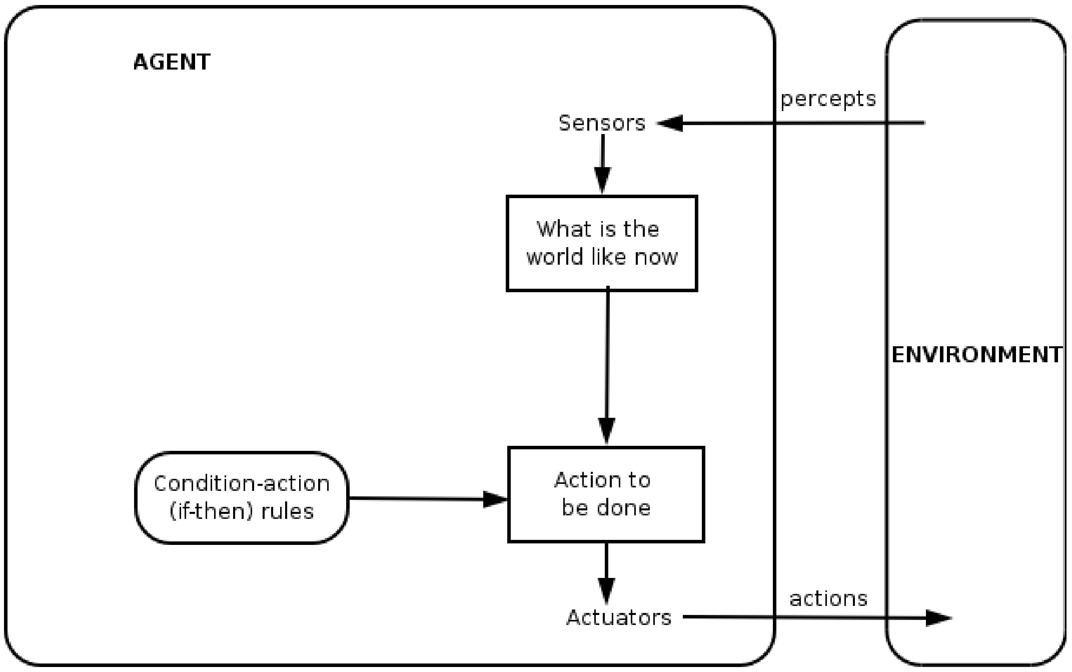


Fig : Schematic diagram of a simple reflex agent.

Rectangles - to denote the current internal state of the agent's decision process
Ovals - to represent the background information used in the process.

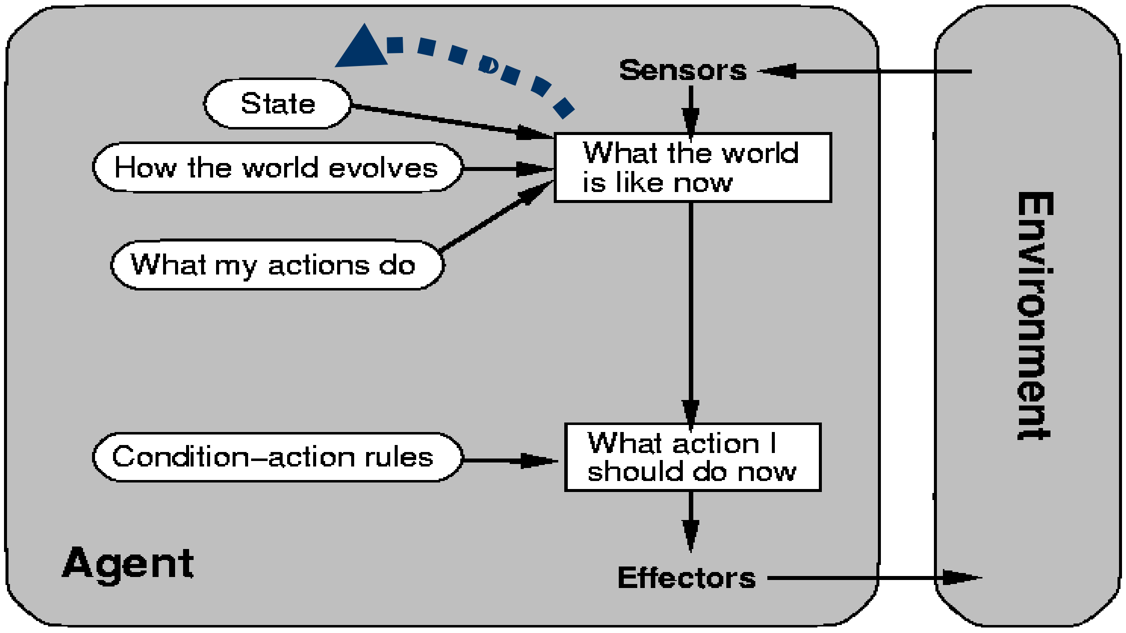
```
function SIMPLE-REFLEX-AGENT(percept) returns action
static : rules, a set of condition-action rules
state < -  I N T E R P R E T - I  N P U T (percept )
rule <-  R U L E - M A T C H (state, rules),
action <-  R U L E - A C T I O N [rule ]
return action
```

- **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 description of the current state.



```
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]
return action
```

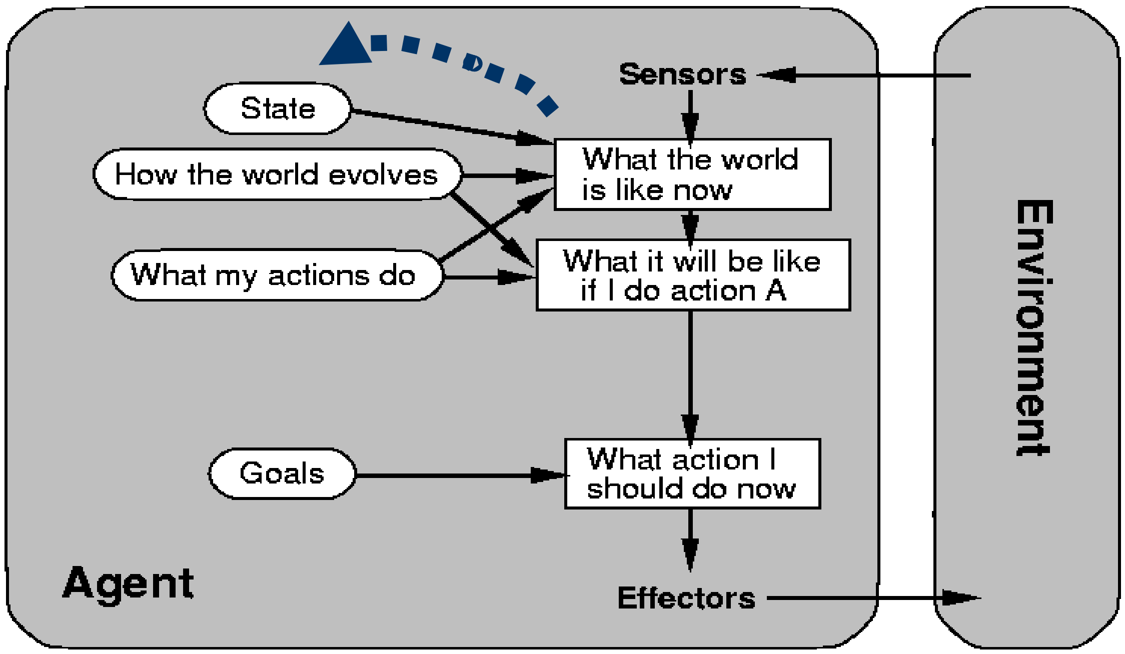
UPDATE-STATE - is responsible for creating the new internal state description

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.


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Effectors
Environment

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

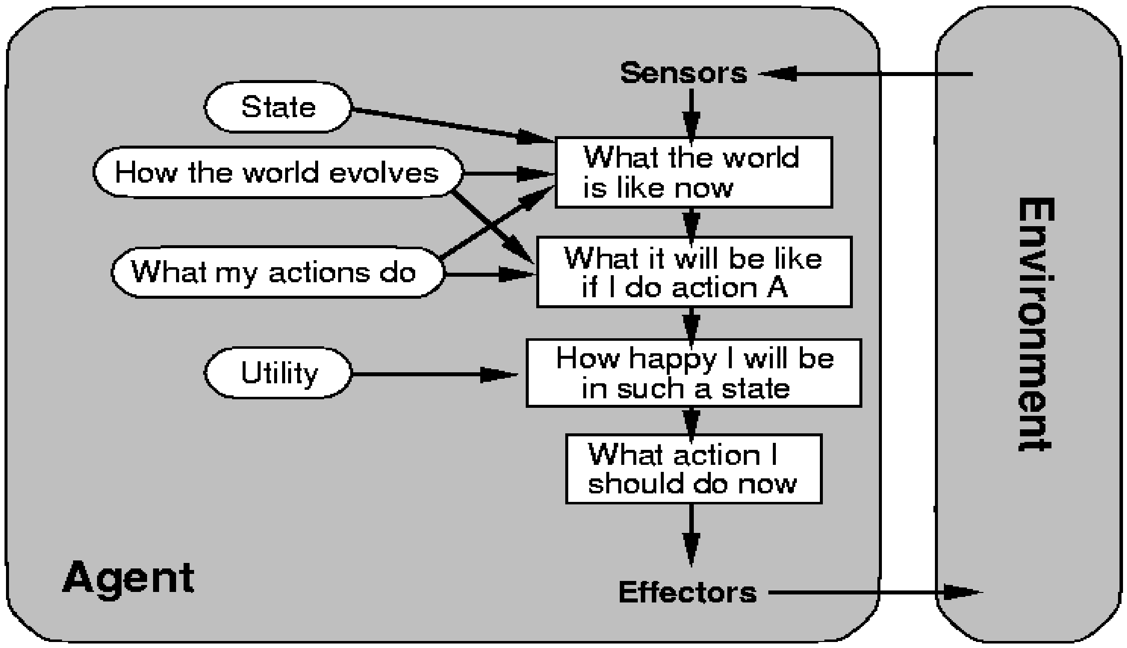
An agent which generates a goal state with high – quality behavior (i.e) if more than one sequence exists to reach the goal state then the sequence with more reliable, safer, quicker and cheaper than others to be selected.

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


Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Effectors
Environment

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

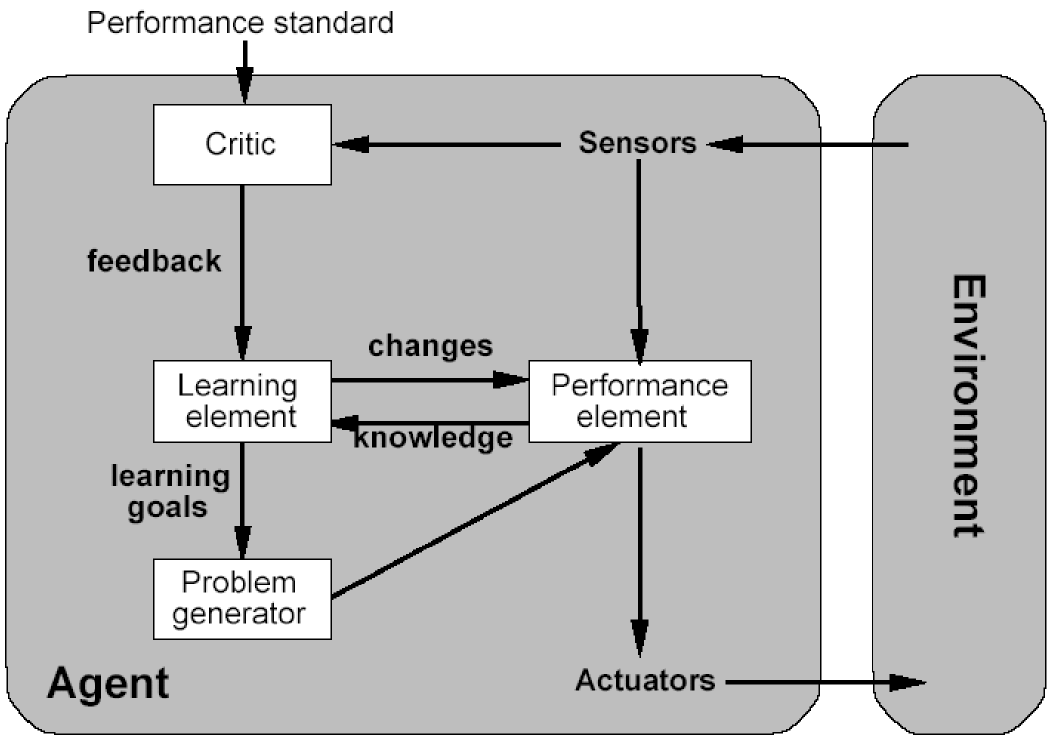
All agents can improve their performance through Learning

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
```

<u>**Note :**</u>

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 problem-solving 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.

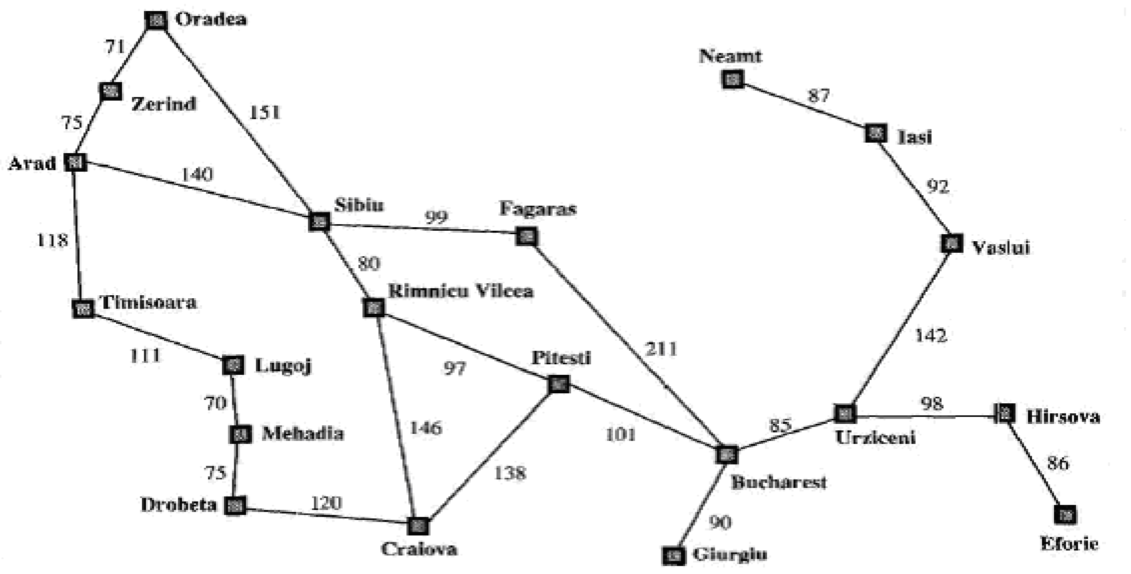
Formulating problems

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.

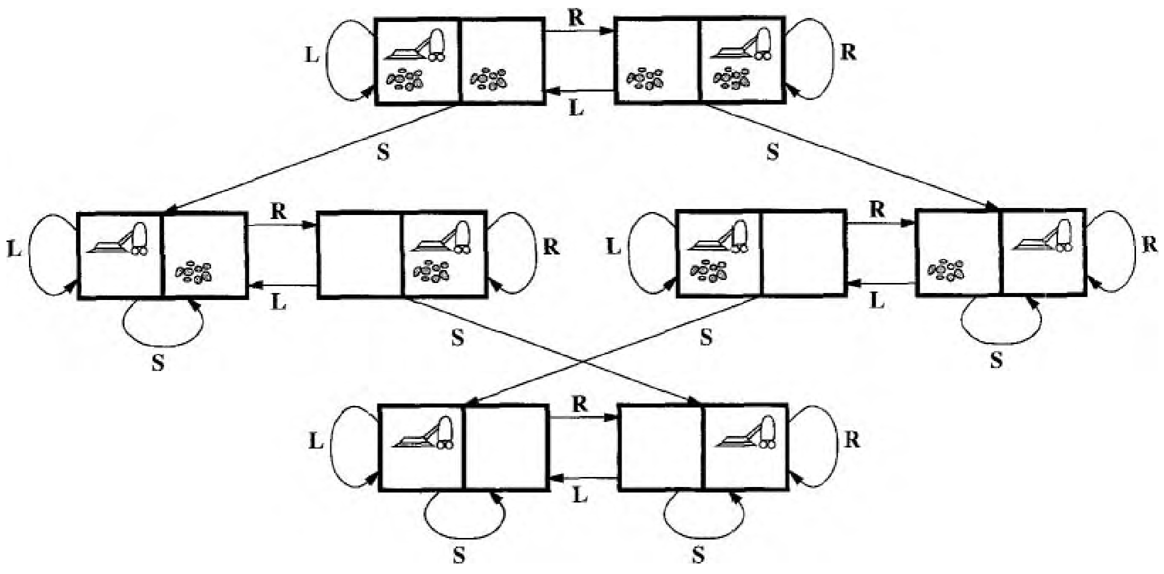
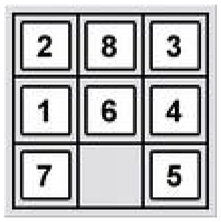


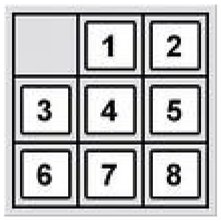
Fig : The complete state space for Vacuum World

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.



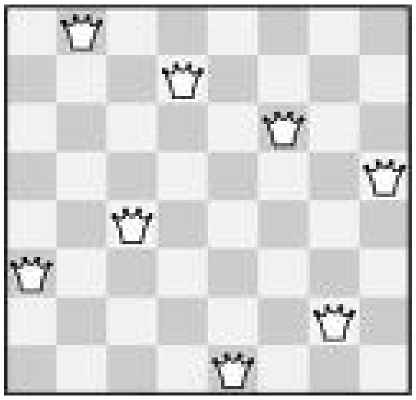


Initial State **Goal State**

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:

Bank1		Boat		Bank2	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)	->	(2,0)	->	(3,1)	(ii)
(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, frequent-flyer 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.

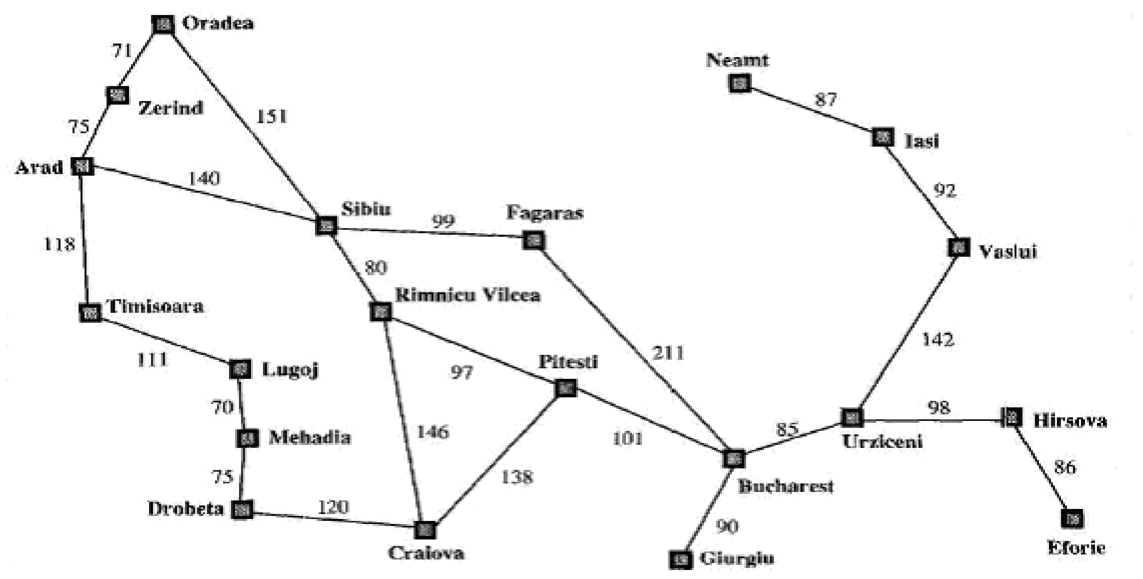
Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially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

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

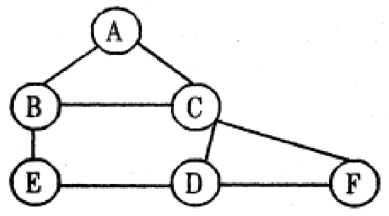
In this case, we get three new states: *In(Sibiu),In(Timisoara),* and *In(Zerind).* Now we must choose which of these three possibilities to consider further.

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

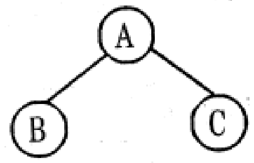


1. Start the sequence with the initial state and check whether it is a goal state or not.

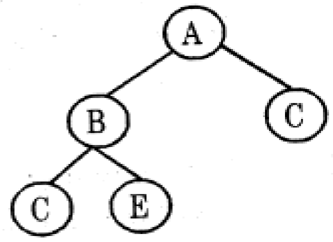
a, If it is a goal state return success.
b. Otherwise perform the following sequence of steps

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.

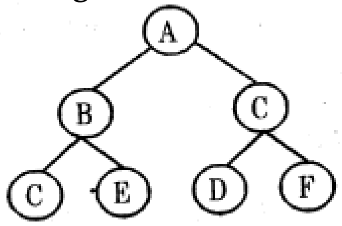
Expanding A



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:

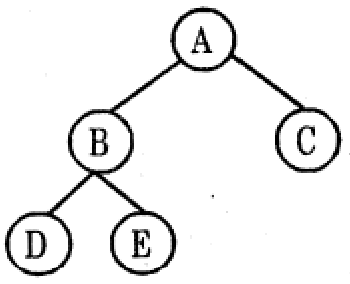
MAKE-QUEUE(elements) - creates a queue with the given elements
EMPTY(queue)-returns true only if there are no more elements in the queue.
REM0VE-FIRST(queue) - removes the element at the front of the queue and returns it
INSERT ALL (elements, queue) - inserts set of elements into the queue and returns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

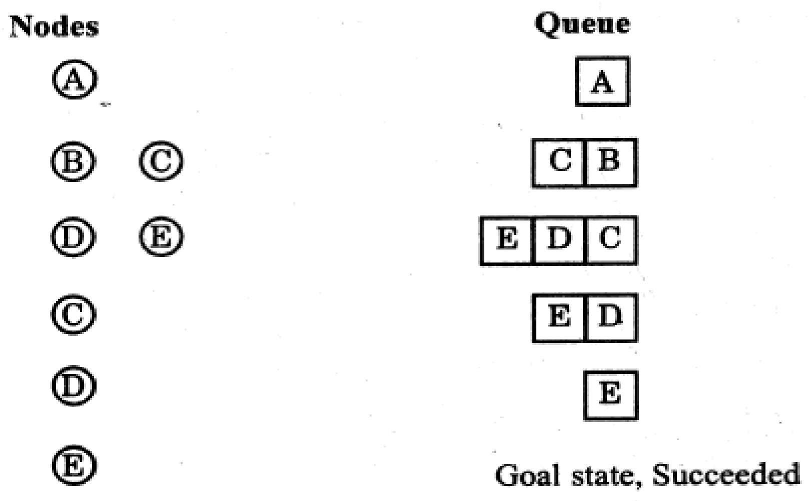
```
function TREE-SEARCH(problem,fringe) returns a
solution, or failure
fringe <- INSERT(MAKE-NODE(INITIAL-STATE[problem]),
fringe)
loop do
if EMPTY?(fringe) then return failure
node <- REMOVE-FIRST(fringe)
ifGOAL-TEST[problenl]applied to STATE[node] succeeds
then return SOLUTION(node)
fringe <- INSERT-ALL(EXPAND(node,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

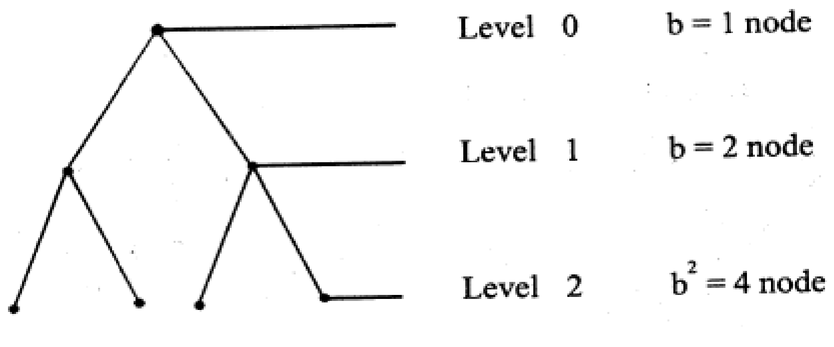


Measuring problem solving performance

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

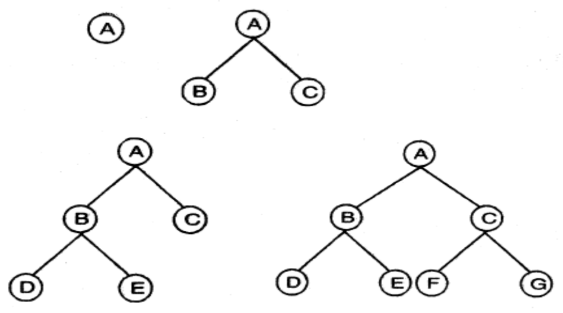
Breadth-first search

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.

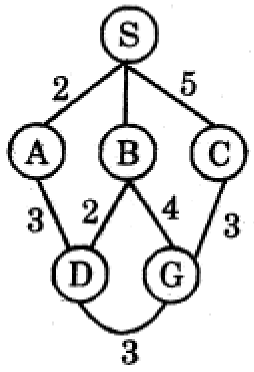
Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first.

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

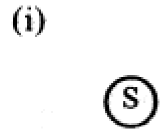
Breadth first search trees after node expansions

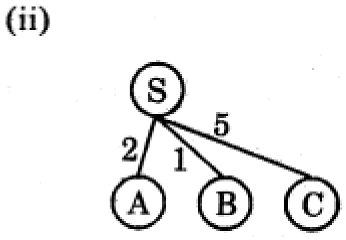


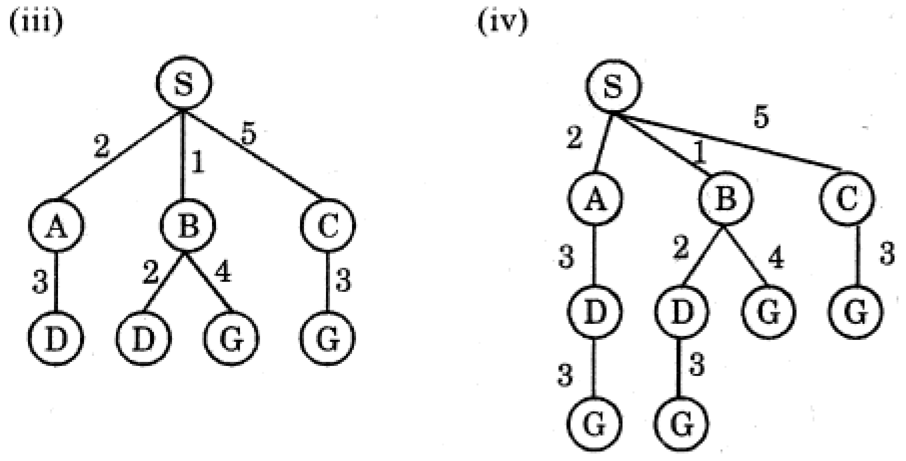
Example: Route finding problem



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 BFS{problem) returns a solution or failure
return TREE-SEARCH (problem, FIFO-QUEUE( ))
```

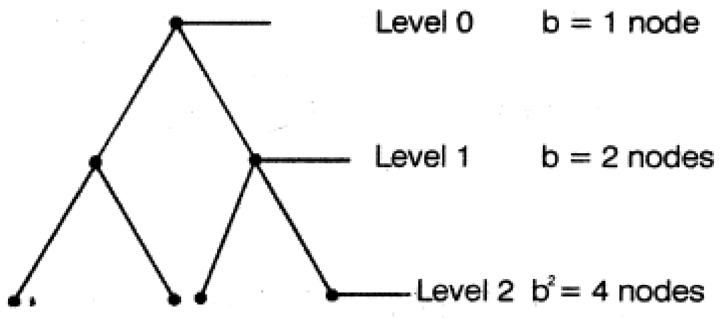
Time and space complexity:

Example:

Time complexity

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

$= O(b^d)$



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

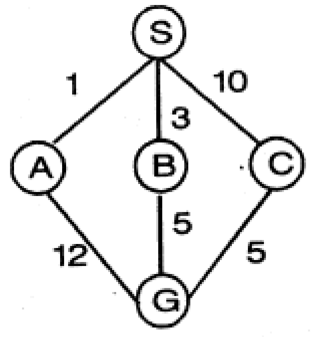
Disadvantage: Suitable for only smallest instances problem (i.e.) (number of levels to be minimum (or) branching factor to be minimum)

Uniform-cost search

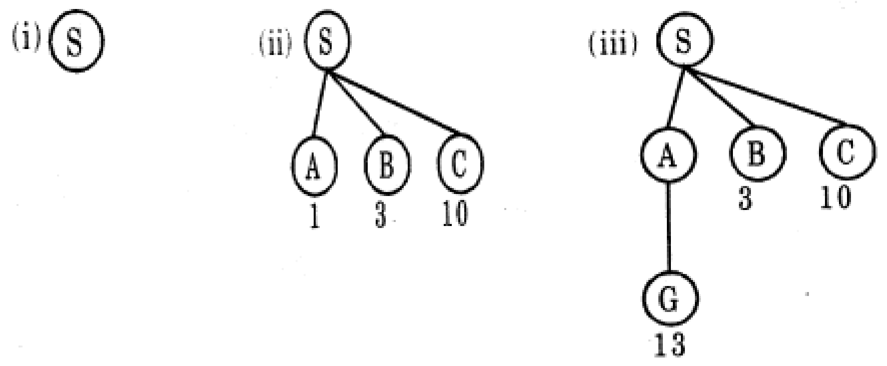
Breadth-first search is optimal when all step costs are equal, because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n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

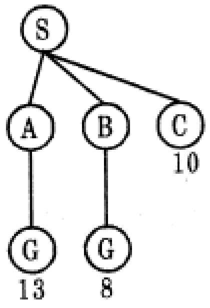


Task **: Find a minimum path cost from S to G**



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.

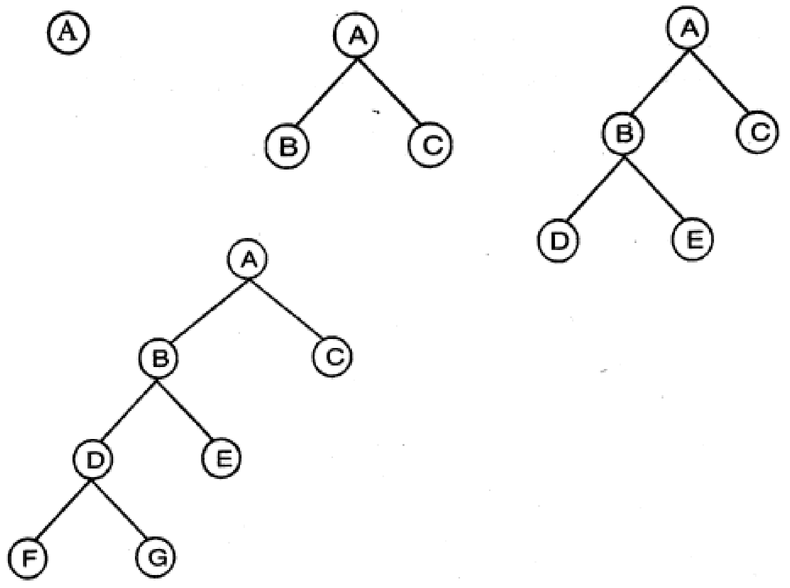
Disadvantage: Suitable for only smallest instances problem.

<u>Depth-first search</u>

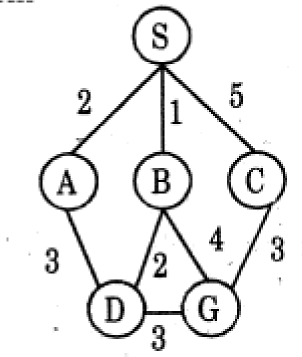
Depth-first search always expands the *deepest* node in the current fringe of the search tree

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

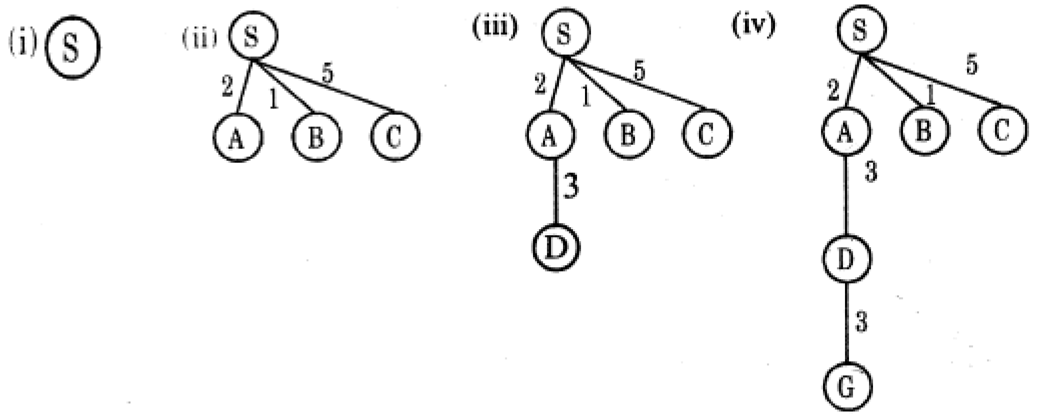
Depth first search tree with 3 level expansion



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())
```

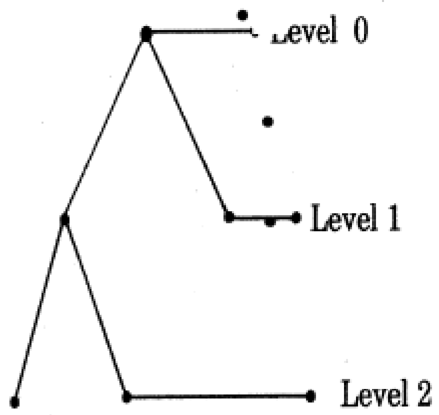
Time and space complexity:

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

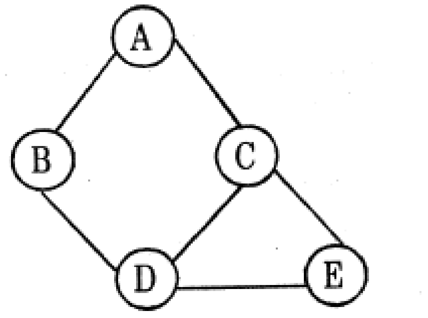
Advantage**:** If more than one solution exists (or) number of levels is high then DFS is best because exploration is done only in a small portion of the whole space.

Disadvantage: Not guaranteed to find a solution

Depth - limited search

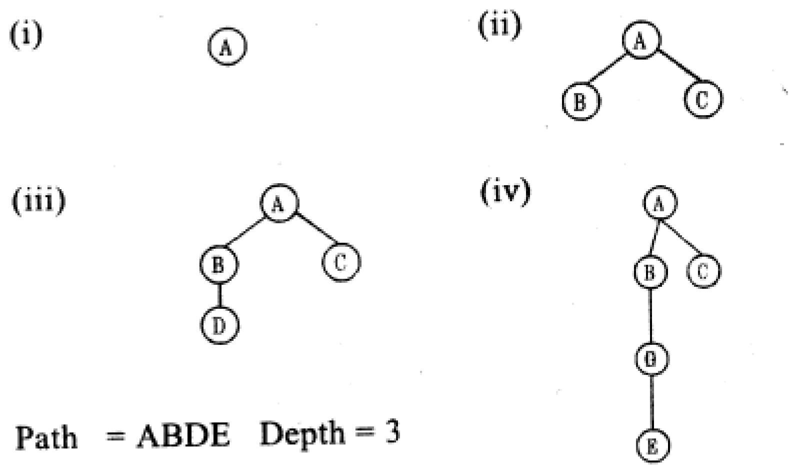
1. *Definition:* A cut off (maximum level of the depth) is introduced in this search technique to overcome the disadvantage of depth first search. The cutoff value depends on the number of states.

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.

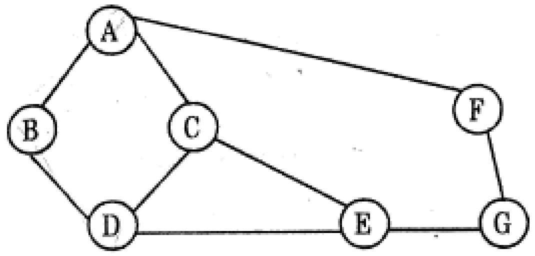
Advantage: Cut off level is introduced in the DFS technique

Disadvantage **:** Not guaranteed to find the optimal solution.
Iterative deepening search

<u>Iterative deepening search</u>

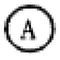
Definition: Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits.

Example: **Route finding problem**

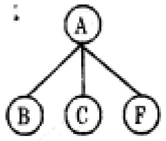


Task: **Find a path from A to G**

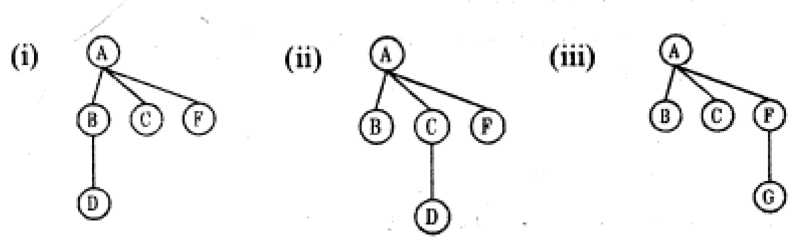
Limit = 0



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.

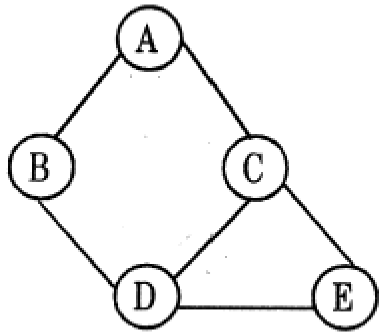
Advantage: This method is preferred for large state space and the depth of the search is not known.

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

<u>Bidirectional search</u>

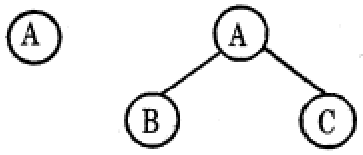
Definition **:** Bidirectional search is a strategy that simultaneously search both the directions (i.e.) forward from the initial state and backward from the goal, and stops when the two searches meet in the middle.

Example: **Route finding problem**

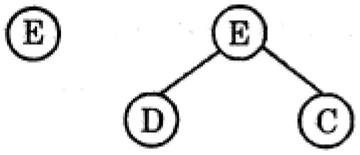


Task **: Find a path from A to E.**

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

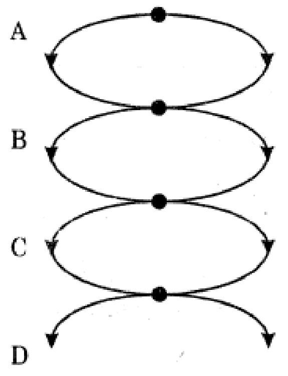
Comparing uninformed search strategies

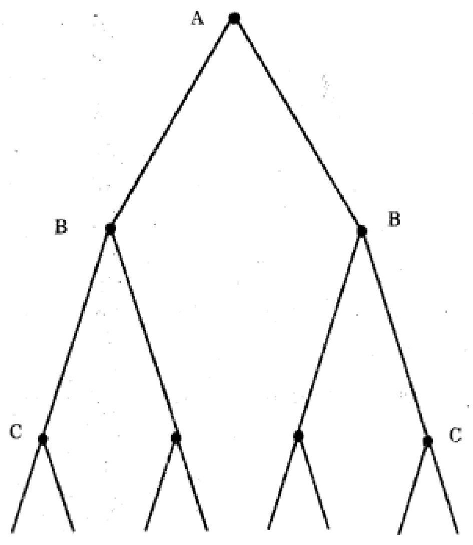
Criterion	**Breadth First**	**Uniform Cost**	**Depth First**	**Depth Limited**	**Iterative Deepening**	**Bi direction**
Complete	**Yes**	**Yes**	**No**	**No**	**Yes**	**Yes**
Time	**$O(b^d)$**	**$O(b^d)$**	**O(bm)**	**$O(b^l)$**	**$O(b^d)$**	**$O(b^{d/2})$**
Space	**$O(b^d)$**	**$O(b^d)$**	**O(bm)**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

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.

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)

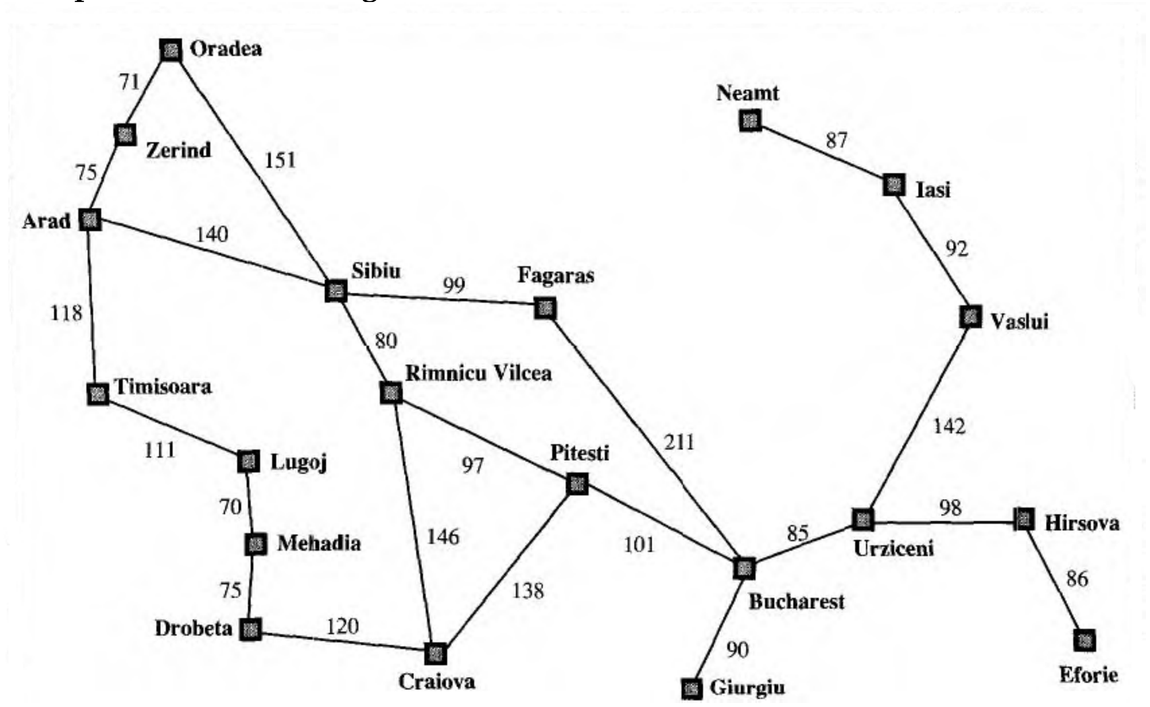
Definition **:** A best first search that uses h(n) to select next node to expand is called greedy search
Evaluation function **:** The estimated cost to reach the goal state, denoted by the letter h*(n)*

```
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 $h_{SLD}(n)$.

```
hSLD(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

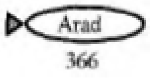
<u>Values of h_{SLD}-straight-line distances to Bucharest</u>

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

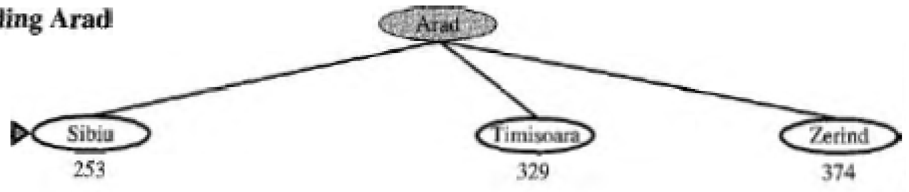
Solution :

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

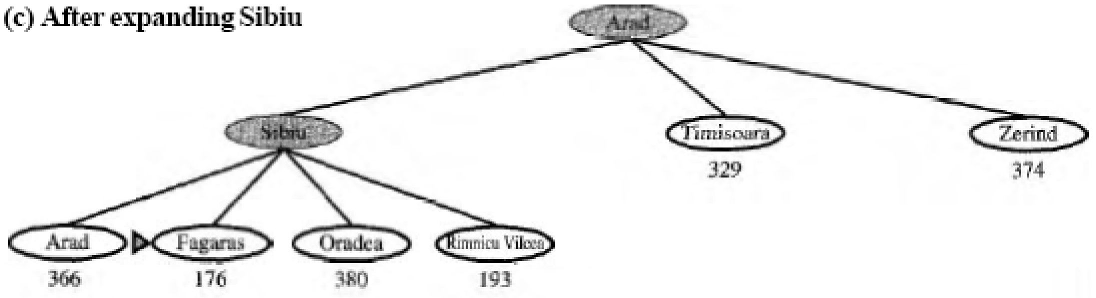
(a) The initial state



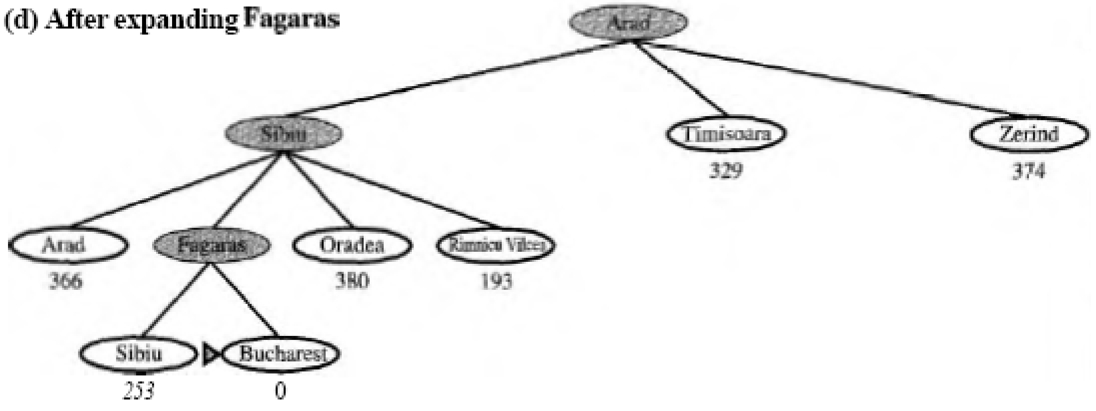
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



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.

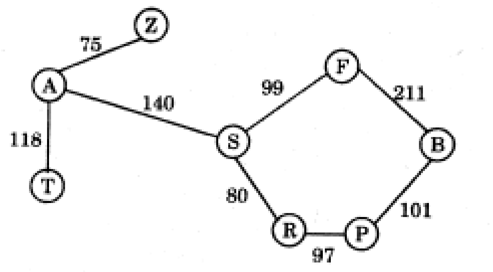
Minimizing *h(n)* is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, because it is closest to Fagaras, but it is a dead end. The solution is to go first to Vaslui-a step that is actually farther from the goal according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



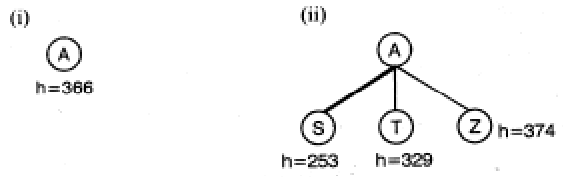
Straight - line distance to B from A:

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

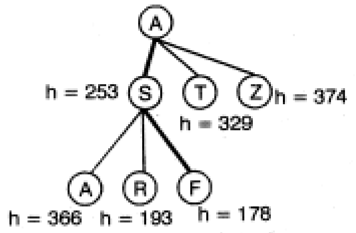
Solution :

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

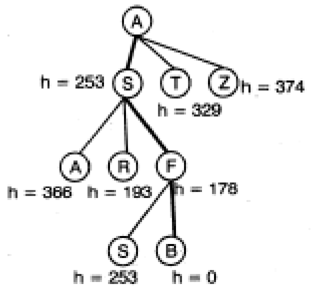
Apply the evaluation function *h(n)* to find a path from A to B



(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)

<u>A* search (Minimizing the total estimated solution cost)</u>

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

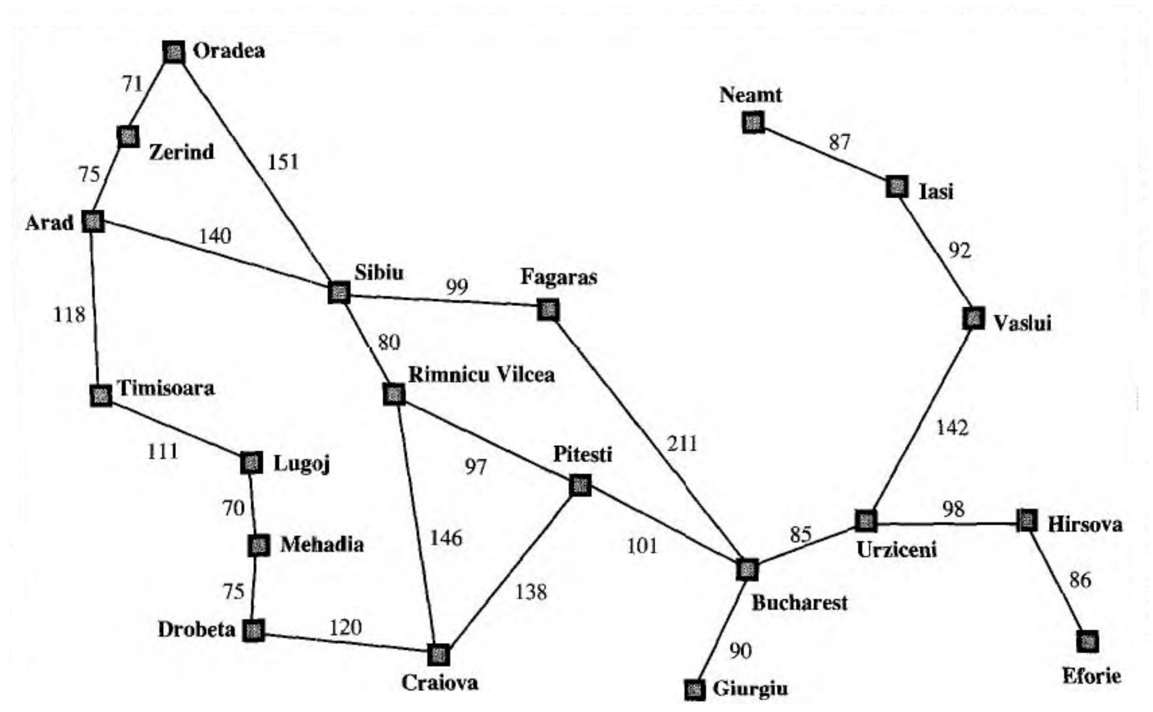
```
f(n) =g(n) + h(n)
```

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

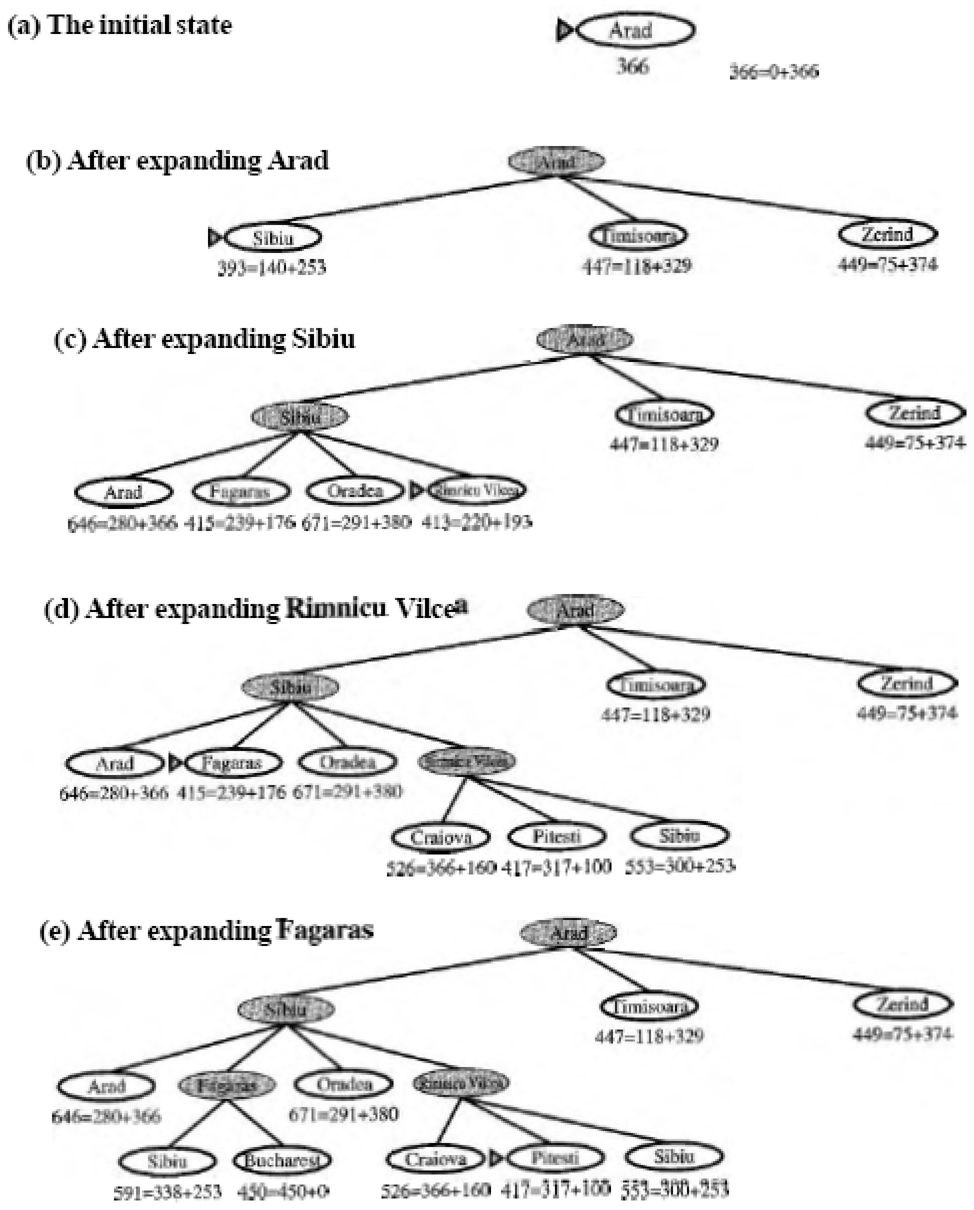
Heuristic function : A good heuristic function for route-finding problems is Straight-Line Distance to the goal and it is denoted as $h_{SLD}(n)$.

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

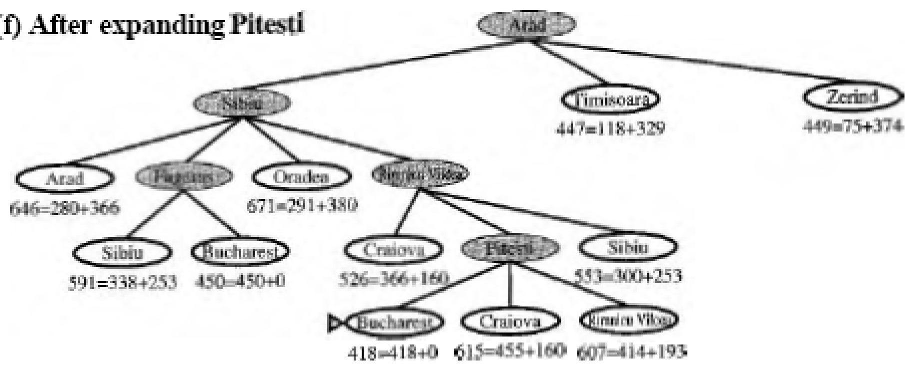
Values of h_{SLD}-straight-line distances to Bucharest

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

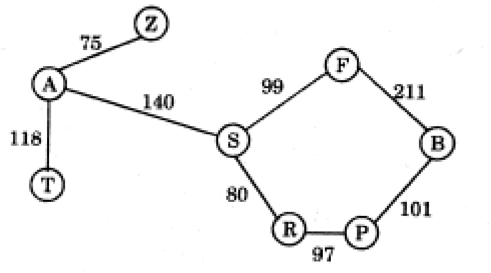
<u>Stages in an A* search for Bucharest. Nodes are labeled with f (n) = g (n) + h(n)</u>



(f) After expanding Pitesti



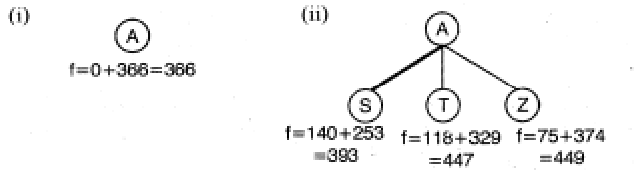
Example 2 : Finding the path from one node to another node



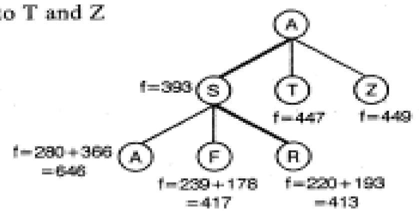
Straight - line distance to B from A:

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

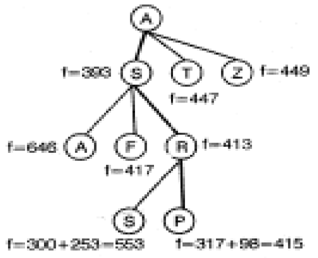
Solution:
From the given graph and estimated cost, the goal state is identified as B from A
Apply the evaluation *function f(n)* = *g(n)* +*h(n)* to find a path from A to B



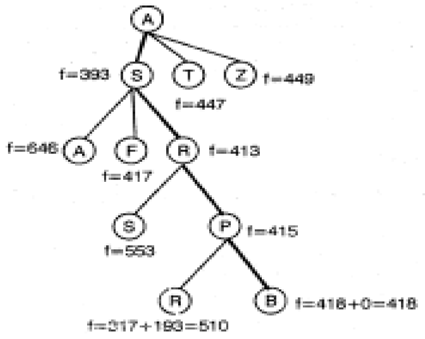
(iii) S is selected for next level of expansion, since f(S) is minimum from S, when comparing to T and Z



(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*

<u>**Memory bounded heuristic search**</u>

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 iterative-deepening 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 a*n 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:

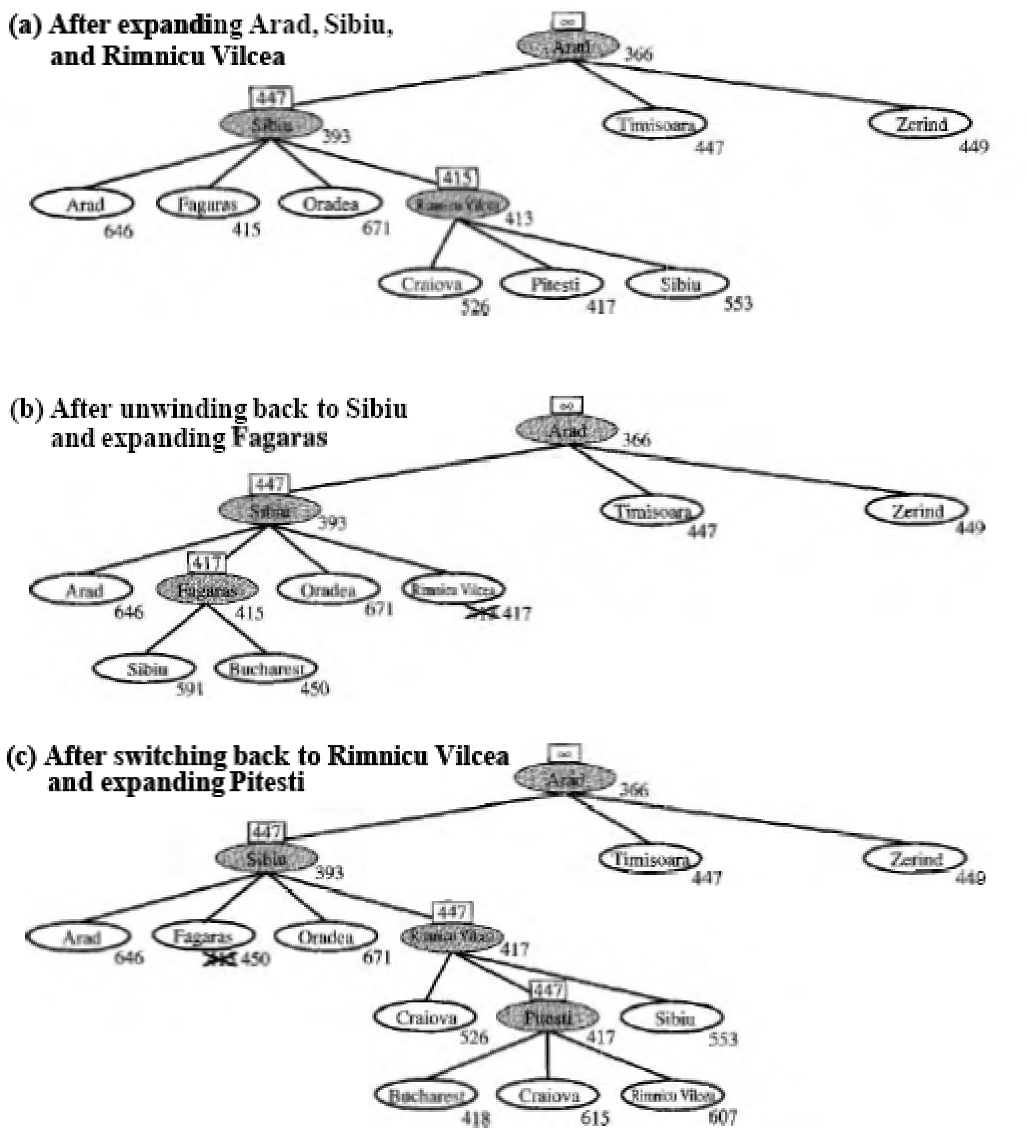
- Recursive Best First Search (RBfS)
- Memory bounded A* search (MA*)

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

Stages in an RBFS search for the shortest route to Bucharest.

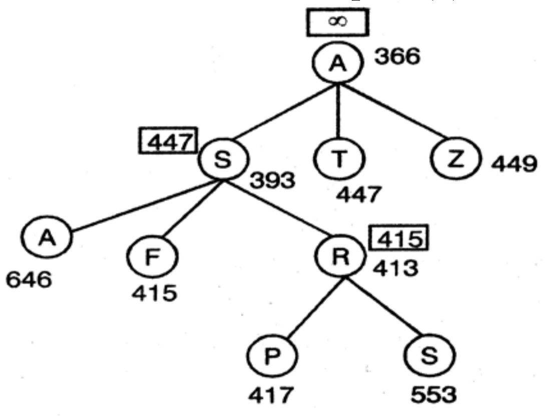

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea

(b) After unwinding back to Sibiu and expanding Fagaras

(c) After switching back to Rimnicu Vilcea and expanding Pitesti

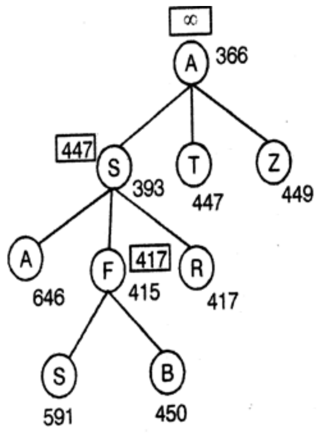
Example:

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



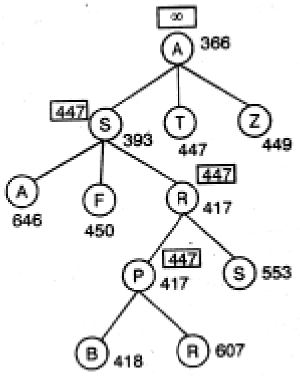
f-limit value of each recursive call is shown on top of each current node. After expanding R, the condition f[best] >f-limit (417 > 415) is true and returns f[best] to that node.

b) 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.

c) 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.

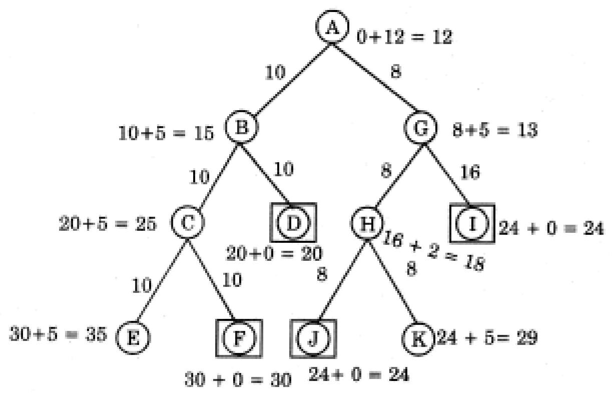
Space and Time complexity: depends on the available number of node.

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

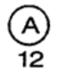
From the Figure we identified that the goal states are D,F,J,I because the *h* value of these nodes are zero (marked as a square)

Available memory - 3 nodes of storage space.

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

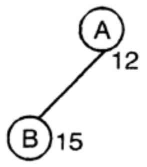
Solution:

(i)



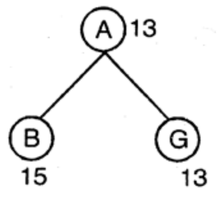
Initial state

(ii)



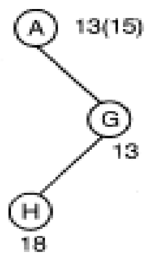
One successor to be added at each stage, therefore B is added to A.

(iii)



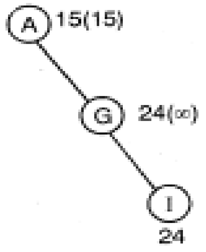
Now, all the children of A is expanded, update the f – cost of root node to the minimum of its children, that is f(A) is 13, the memory is now full.

(iv)



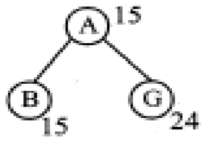
Now, expand the node G to its next successor, but already the memory is full. B is discarded from the queue and it is added as a forgotten descendent of A, which is shown in the parenthesis. The node H is added to G, with $f(H)=18$ but H is not a goal state and it uses all the available memory. Hence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

(v)



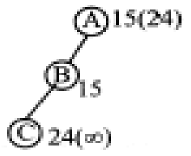
G is expanded again. We drop H and add I (i.e) $f(I) = 24$. The value of G is updated with minimum of H and I and the forgotten value in the parenthesis (i.e) $f(G) = 24$ (∞) Again the value of A is also updated with minimum of $f(B) = 15$ and $f(G) = 24$ and the forgotten value in the parenthesis (i.e) $f(A) = 15(15)$. 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)



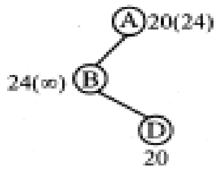
The path from the root (A) leads to B because f(A) = 15 in the previous step. Therefore B is generated for the second time.

(vii)



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

(viii)



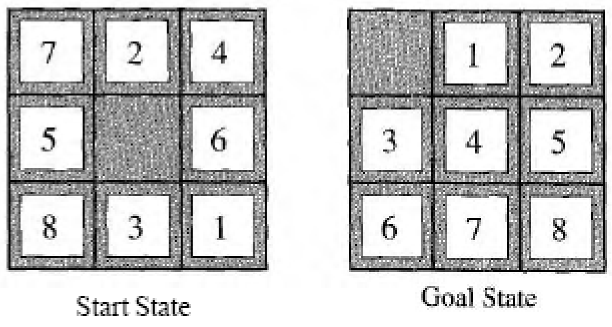
Drop C, add the another successor (D) to B (i.e) $f(D) = 20$, and this value is inherited by B and A.

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.

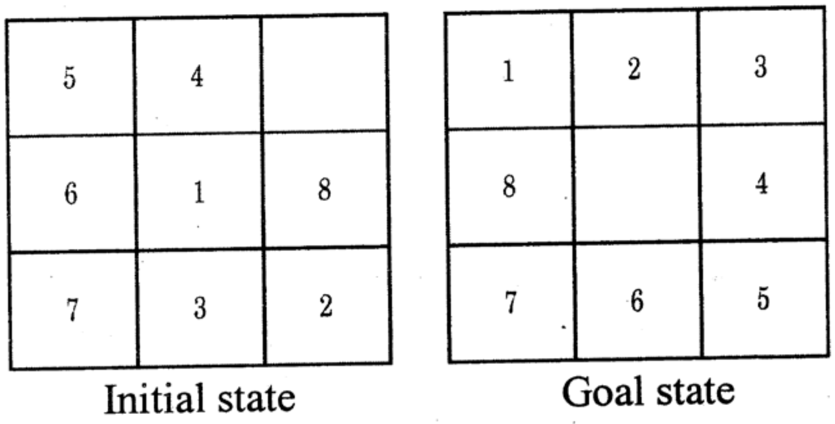
All of the eight tiles are out of position in the above figure, so the start state would have h_l = 8. h_l is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

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 :



Initial state Goal state

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.

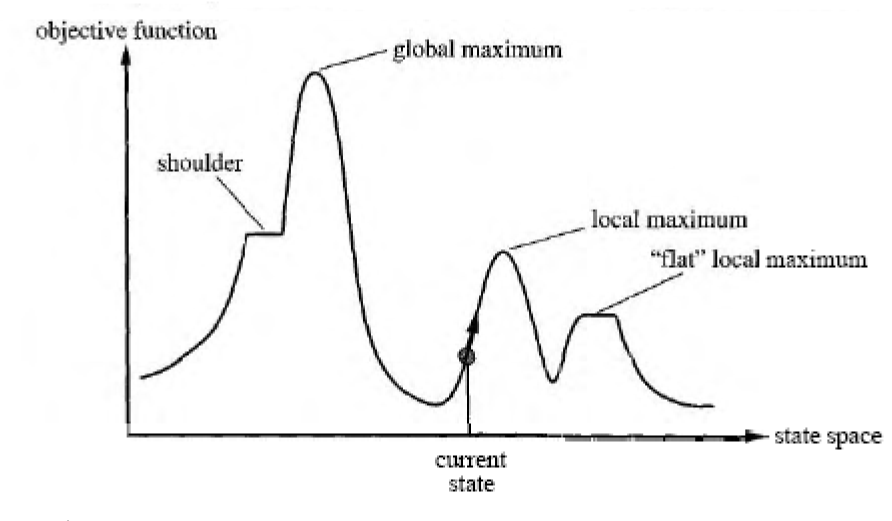
The local search problem is explained with the state space land scape. A landscape has:

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.

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



<u>Applications</u>

Integrated - circuit design
Factory - floor layout
Job-shop scheduling
Automatic programming
Vehicle routing
Telecommunications network Optimization
<u>Advantages</u>

- ❖ 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

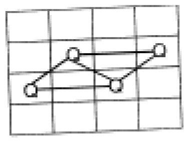
```
function HILL-CLIMBING(problem) returns a state that is a
local maximum
inputs: problem, a problem
local variables:current, a node and neighbor, a node
current <- MAKE-NODE(INITIAL-STATE[problem])
loop do
neighbor <- a highest-valued successor of current
if VALUE[neighbor] <= VALUE[current] then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

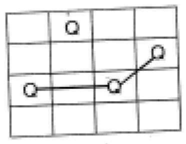
To illustrate hill-climbing, we will use the 8-queens, where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors).

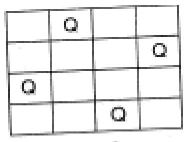
Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

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

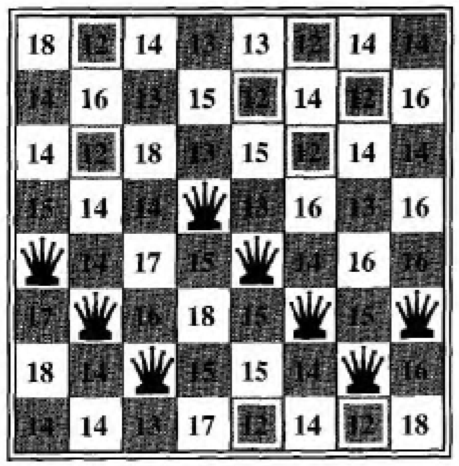
The global minimum of this function is zero, which occurs only at perfect solutions.



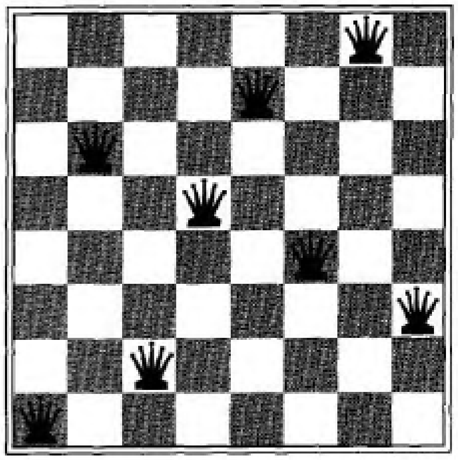




An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked.



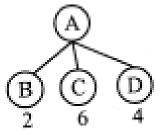
A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.



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

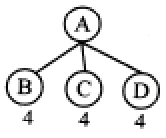
Example :



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.

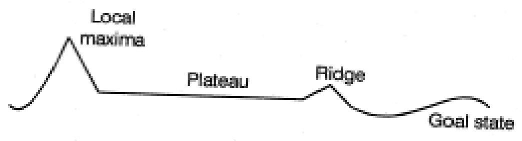
Example :



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

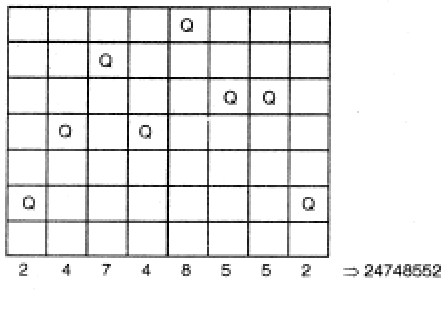
GA begins with a set of k randomly generated states, called the **population.** Each state, or **individual,** is represented as a string over a finite alphabet.

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 :

24748552

32752411

24415124

32543213

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

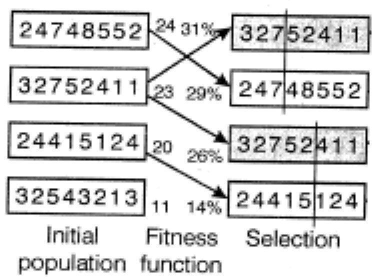
Maximum fitness value is : 8*7/2 = 28 for a solution

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

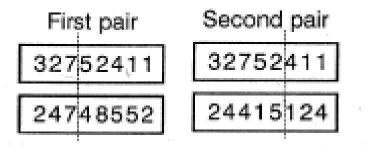
The probability of being chosen for reproduction is directly proportional to the fitness score, which is denoted as percentage.

24 / (24+23+20+11) = 31%
23 / (24+23+20+11) = 29%
20 / (24+23+20+11) = 26%
11 / (24+23+20+11) = 14%

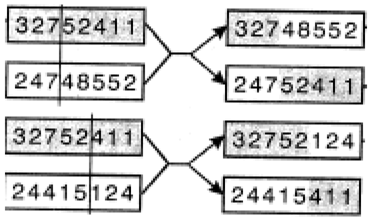
Selection : A random choice of two pairs is selected for reproduction, by considering the probability of fitness function of each state. In the example one state is chosen twice probability of 29%) and the another one state is not chosen (Probability of 14%)



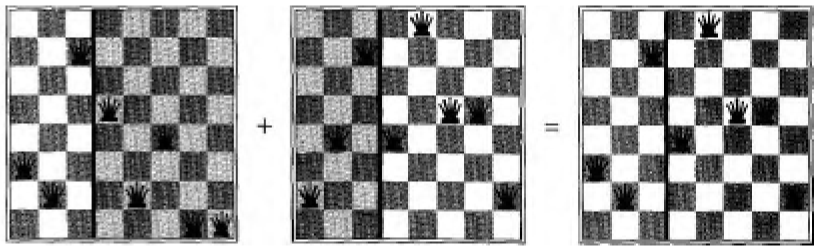
Cross over: Each pair to be mated, a crossover point is randomly chosen. For the first pair the crossover point is chosen after 3 digits and after 5 digits for the second pair.



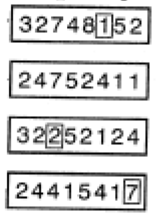
offspring **:** Offspring is created by crossing over the parent strings in the crossover point. That is, the first child of the first pair gets the first 3 digits from the first parent and the remaining digits from the second parent. Similarly the second child of the first pair gets the first 3 digits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



The 8-queens states corresponding to the first two parents

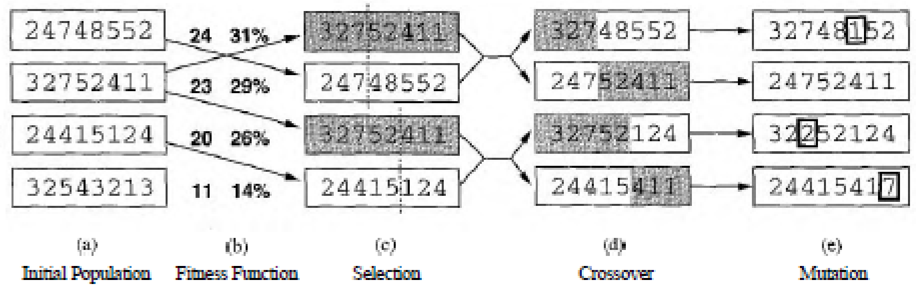


Mutation : Each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring



Mutation

<u>**Production of Next Generation of States**</u>



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) do
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.

<u>Online Search Agents and Unknown Environments</u>

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).

Online search agents

An online algorithm, expand only a node that it physically occupies. After each action, an online agent receives a percept to know the state it has reached and it can augment its map of the environment, to decide where to go next.

An online depth-first search agent: This agent is implemented with the following requirements:

- result *[a, s]* - records the state resulting from executing action in state *s.*
- An action from the current state has not been explored, then it is explored and returns action *a.*
- If there is no action exists from the current and possible to backtrack then backtracking is done with action *b* returns action *a.*
- If the agent has run out of states to which it can backtrack then its search is complete.

An online search agent that uses depth-first exploration

```
function ONLINE-DFS-AGENT(s') returns an action
inputs: s', a percept that identifies the current state
static: result, a table, indexed by action and state,
initially empty
unexplored, a table that lists, for each visited state,
the actions not yet tried
unbacktracked, a table that lists, for each visited
state, the backtracks not yet tried
s, a, the previous state and action, initially null
if GOAL-TEST(s') then return stop
if s' is a new state then unexp1ored[s']<- ACTIONS(s')
if s is not null then do
result[a, s] <- s'
add s to the front of unbacktracked[s']
if unexplored[s'] is empty then
if unbacktracked[s'] is empty then return stop
else a <- an action b such that result[b, s'] =
POP(unbacktracked[s'])
else a <- POP(unexplored[s'])
s <- s'
return a
```

Online local search

Hill climbing search technique can be used to perform online local search because it keeps just one current state in memory. To avoid the drawback of local maxima, random walk is Chosen to explore the environment instead of random restart method. The concept of hill climbing with memory stores a "current best estimate". H(s) of the cost to reach the goal from each state that has been visited is implemented as **Learning Real-Time A* (LRTA*)** algorithm,

Sequence of steps for LRTA* algorithm

◊ It builds a map of the environment using the result table.
◊ H(s) is initially empty, when tile process starts h(s) is initialized for new states and it is updated when the agent gains experience In the state space.
◊ Each state is updated with minimum H(s) out of all possible actions and the same action is returned

LRTA* algorithm

```
function LRTA*-AGENT(s') returns an action
inputs: s' , a percept that identifies the current state
static: result, a table, indexed by action and state,
initially empty
H, a table of cost estimates indexed by state, initially
empty
s, a, the previous state and action, initially null
if GOAL-TEST(s') then return stop
if s' is a new state (not in H) then H[s'] <- h(s')
unless s is null
result[a, s] <- s'
H[s] <- min LRTA*-COST(s, b , result[b,s ],H )
        b ∈ ACTIONS(S)
a <- an action b in ACTIONS(s') that minimizes LRTA*-
COST(s', b,result[b, s'],H )
s <- s'
return a
function LRTA*-COST(s , a, s', H ) returns a cost
estimate
if s' is undefined then return h(s)
else return c(s, a, s') + H[s']
```

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.

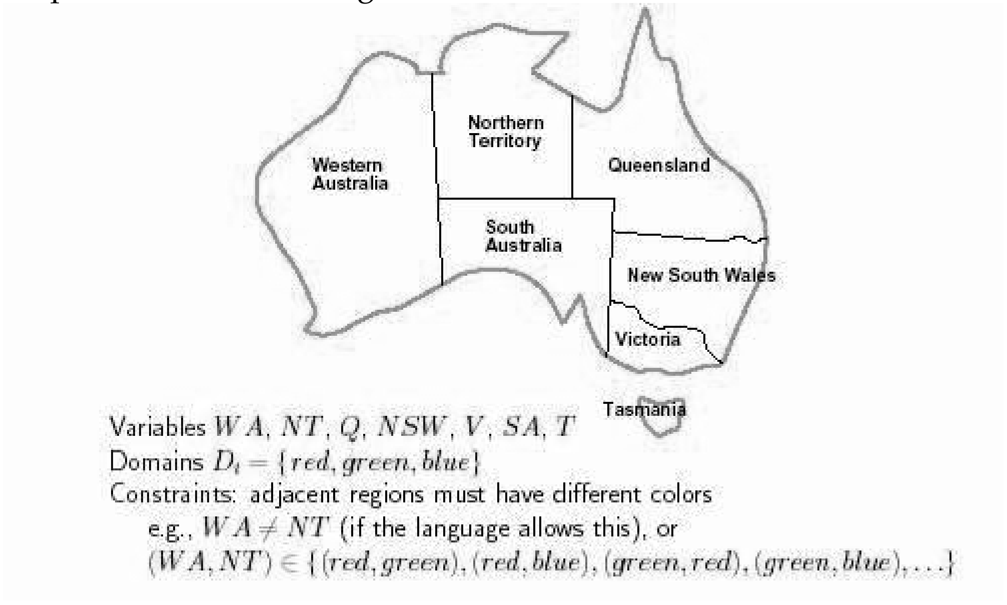
A **State** of the problem is defined by an **assignment** of values to some or all of the variables,{$X_i = v_i$, $X_j = v_j$,…}. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.**

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



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

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 $\neq$ NT)

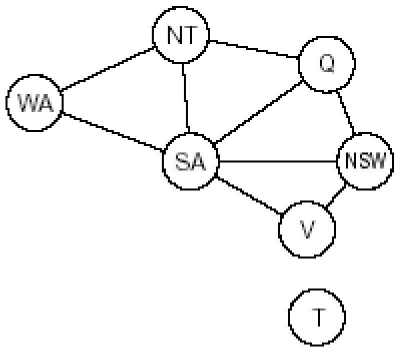
There are many possible solutions, such as

{ 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.

Constraint graph: nodes are variables, arcs show constraints



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 $\neq$ green

Binary constraints - relates pairs of variables.
Example : SA $\neq$ WA

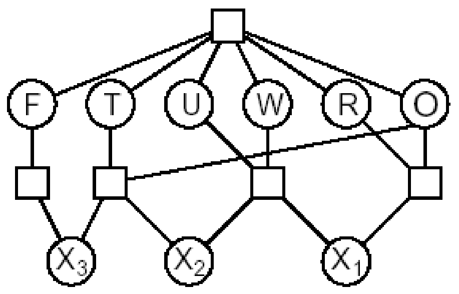
Higher order constraints involve 3 or more variables.

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

```
    T W O
+   T W O
  F O U R
```



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 $\neq$ T, F$\neq$ U, and so on.
The addition constraints on the four columns of the puzzle also involve several variables and can be written as

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

$X_1 + T + T = O + 10 . X_3$
$X_3 = F$

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

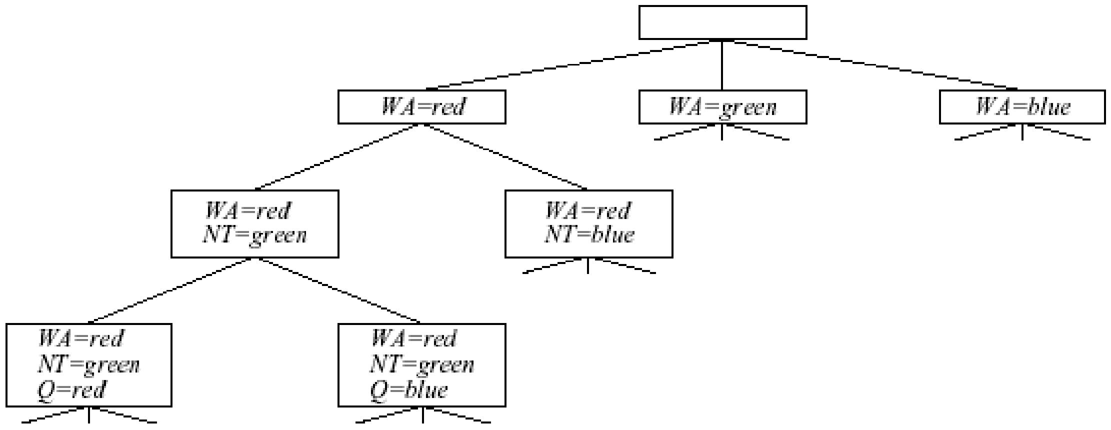
<u>Backtracking Search for CSPs</u>

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.

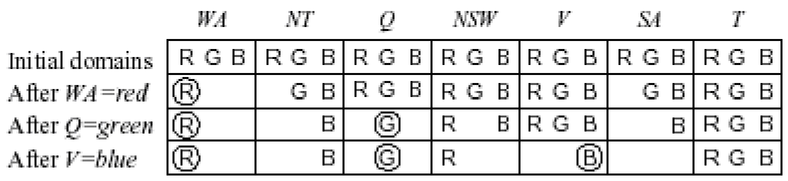
Propagating information through constraints

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*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		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 *NS*W. 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.

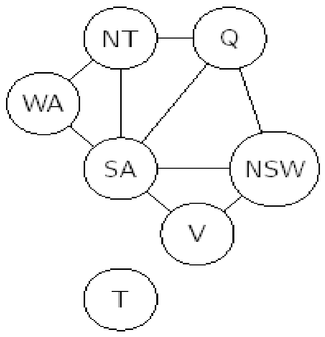


Figure: Australian Territories

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 D*j 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 {X1, X2, . . . ,
Xn}
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(xi,xj) then
for each Xk in NEIGHBORS[Xj) do
add (Xk , Xi)to queue
```

```
function REMOVE-INCONSISTENT-VALUEXS(xi,xj) returns true
iff we remove a value
removed <-false
for each x in DOMAIN[xi] do
if no value y in DOMAIN[xj] allows (x, y) to satisfy the
constraint between Xi and Xj
then delete x from DOMAIN[Xi]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**.

<u>**Handling special constraints**</u>

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

Example: Crypt arithmetic problem

The inconsistency arises in All diff 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.

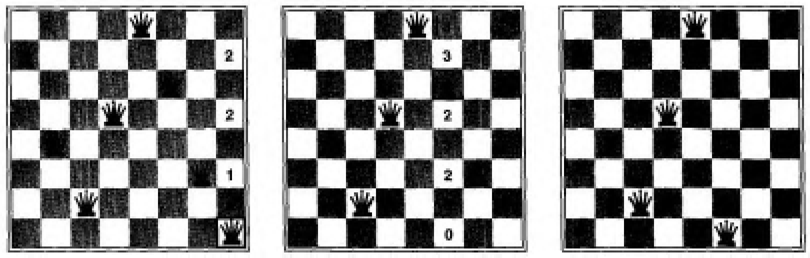
- Initial state - assigns a value to every variable.
- Successor function - works by changing the value of each variable

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.

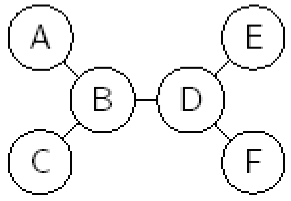


The structure of problems

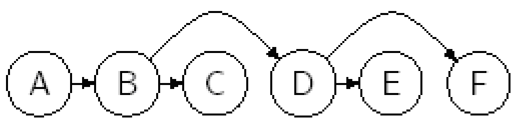
The complexity of solving a CSP-is strongly related to the structure of its constraint graph. If CSP can be divided into independent sub problems, then each sub problem is solved independently then the solutions are combined. When the n variables are divided as *n/c* subproblems, each will take d^c work 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_1 .... 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:

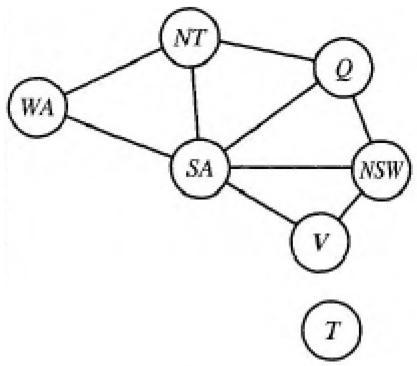
(a) Removing nodes - Cutset conditioning
(b) Collapsing nodes together - Tree decomposition.

(a) Removing nodes - Cutset conditioning

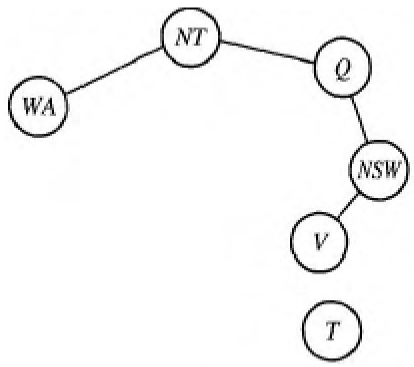
- Assign values to some variables so that the remaining variables form a tree.
- Delete the value assigned variable from the list and from the domains of the other variables any values that are inconsistent with the value chosen for the variable.
- This works for binary CSP's and not suitable for higher 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

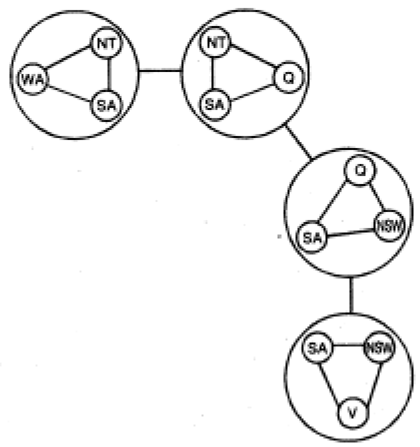
- Construction of tree decomposition the constraint graph is divided into a set of subproblems, solved independently and the resulting solutions are combined.

- 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.

<u>**Adversarial Search**</u>

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.**

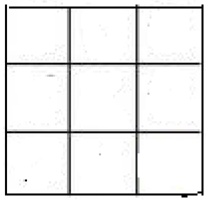
Utility function (also called an objective function or payoff function), - which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1, -1, or 0. Some games have a wider , variety of possible outcomes; the payoffs in backgammon range from +192 to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

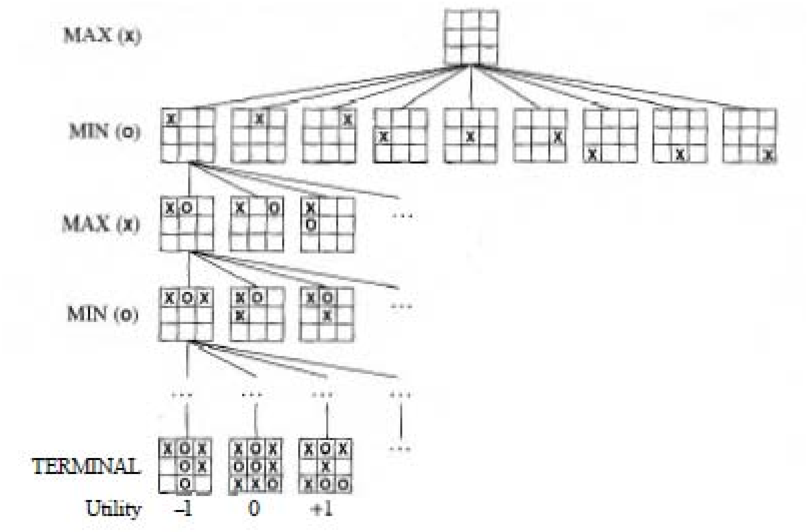
Draw between two Players

X	O	X
O	O	X
X	O	O

Utility function

Win = 1 Draw =0 Loss = -1

A (partial) search tree for the game of tic-tac-toe



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).

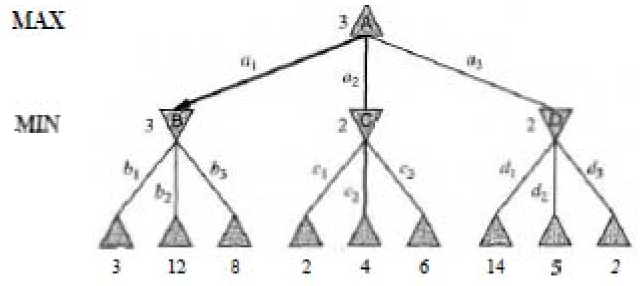
MINIMAX- VALUE(n) =

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 $_{a1}$, 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



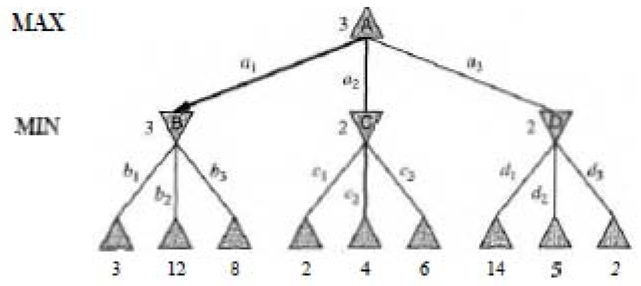
Δ	-	Moves by Max Player
∇	-	Moves by Min Player

- 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_l, because it leads to the successor with the highest minimax value
- MIN'S best reply is b_l, 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
```

```
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

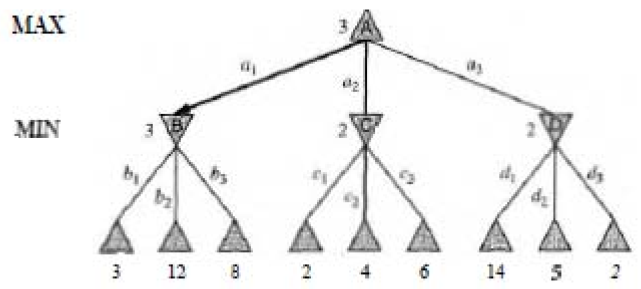
Pruning - The process of eliminating a branch of the search tree from consideration without examining is called *pruning.*

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



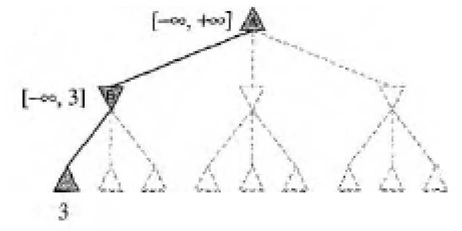
Let the two unevaluated successors of node C have values *x* and *y* and let *z* be the minimum of x and *y*. The value of the root node is given by

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.**

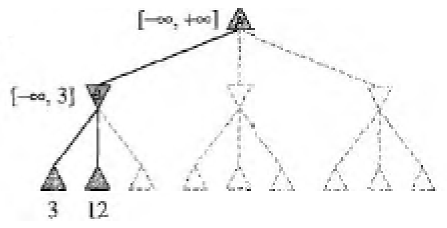
In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves *x* and *y.*

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

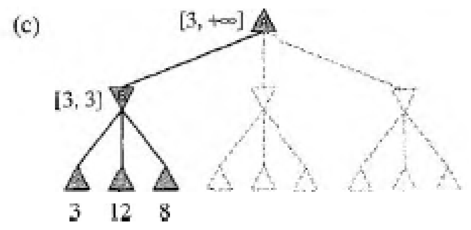
(a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3



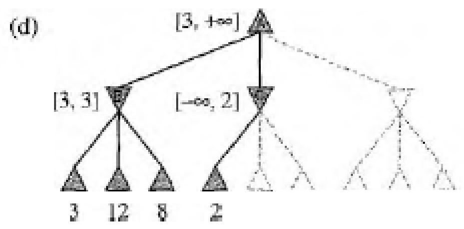
(b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3



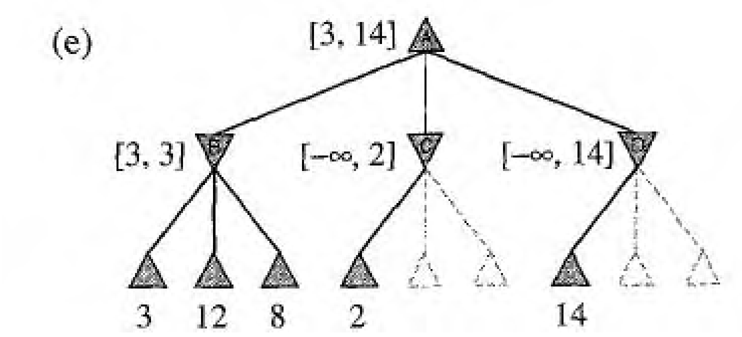
c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of *B* is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root.



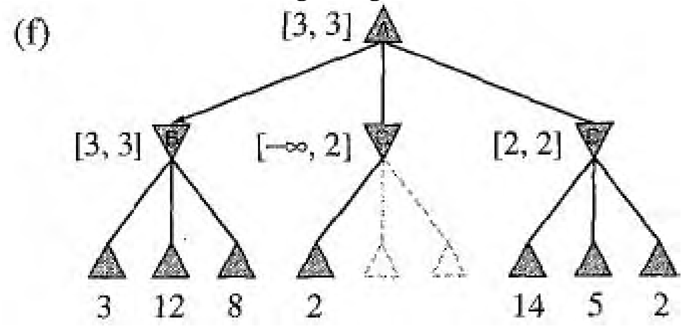
(d) The first leaf below C has the value 2. Hence, *C,* which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha-beta pruning.



(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 MIN - VALUE ( state, $\alpha$,$\beta$)  returns $\alpha$ utility
value
inputs: state, current state in game
$\alpha$  , the value of the best alternative for MAX along the
path to state
$\beta$, 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<- +$\infty$
for $\alpha$, S in S U C C E S S O R S (state) do
v <-  MIN(v,MAX-VALUE(S, $\alpha$, $\beta$ ))
if v <= $\alpha$ then return v
$\beta$ <- MIN($\beta$, 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 $\sqrt{\beta}$ 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

$\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + w_3 f_3(s) \ldots\ldots + w_n f_n(s)$
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.

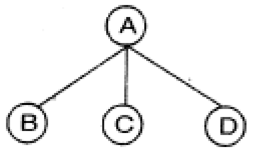
2. Cut - off test

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.*

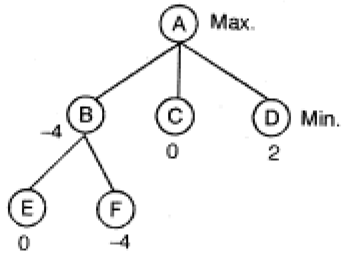
Quiescence search - A search which is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

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

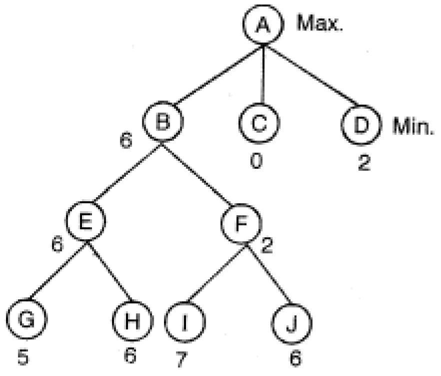
Example:
1. Beginning of the search - one ply



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



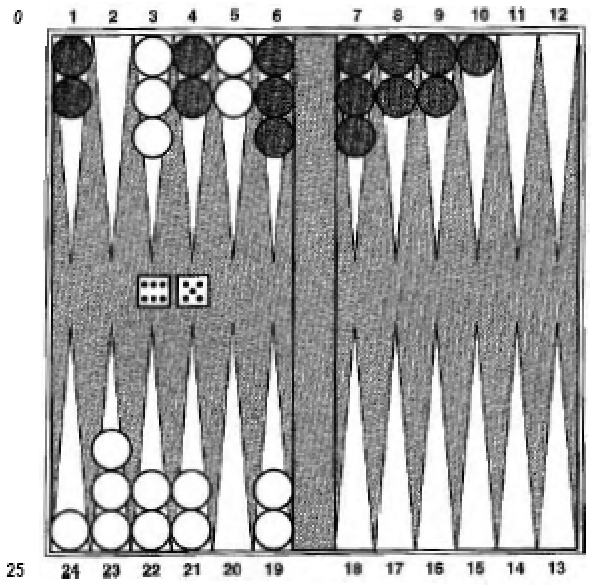
3. When one more successor level is generated from E and F and situation comes down and the value of B is retained as a good move. The time B is waited for this situation is called waiting for quiescence.



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

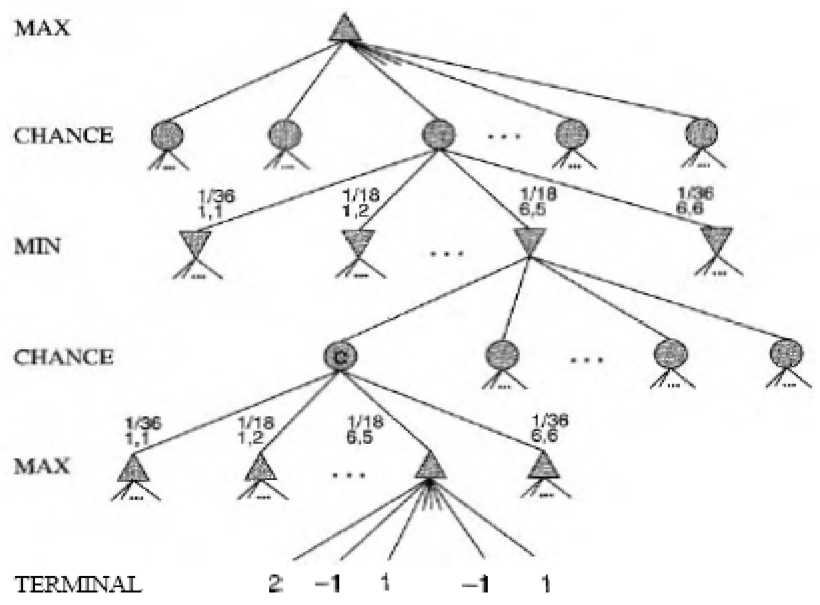
Task : In the position shown, White has rolled 6-5. So Find out the legal moves for the set of the dice thrown as 6 and 5.

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

State Of The Art Game Programs

Designing game playing programs has a dual purpose

1. To better understand how to choose actions in complex domain with uncertain outcomes
2. To develop high performance system for the particular game

Some of the Game playing in AI

1. Chess
2. Checkers
3. Othello
4. Backgammon
5. Go
6. Bridge.