UNIT – 01

Meaning and definition of artificial intelligence

Unit-01/Lecture-01

Introduction to AI

What is artificial intelligence?

Artificial Intelligence is the branch of computer science concerned with making computers behave like humans.

Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success. John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."

The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

Systems that think like humans

—The exciting new effort to make computers think … machines with minds,in the full and literal sense.||(Haugeland,1985)

Systems that think rationally

—The study of mental faculties through the use of computer models.||

(Charniak and McDermont,1985)

Systems that act like humans

The art of creating machines that perform functions that require intelligence when performed by people.||(Kurzweil,1990)

Systems that act rationally

“Computational intelligence is the study of the design of intelligent agents.||(Poole et al.,1998)

The four approaches in more detail are as follows :
(a)Acting humanly : The Turing Test approach
o Test proposed by Alan Turing in 1950
o The computer is asked questions by a human interrogator.
The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass ,the computer need to possess the following capabilities :
 Natural language processing to enable it to communicate successfully in English.
 Knowledge representation to store what it knows or hears
 Automated reasoning to use the stored information to answer questions and to draw new conclusions.
Machine learning to adapt to new circumstances and to detect and extrapolate patterns
To pass the complete Turing Test, the computer will need
Computer vision to perceive the objects, and Robotics to manipulate objects and move about.
(b)Thinking humanly : The cognitive modeling approach We need to get inside actual working of the human mind :
(a) through introspection – trying to capture our own thoughts as they go by;
(b) through psychological experiments
Allen Newell and Herbert Simon, who developed GPS, the - General Problem Solver‖ tried to trace the reasoning steps to traces of human subjects solving the same problems. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind
(c)Thinking rationally : The “laws of thought approach”
The Greek philosopher Aristotle was one of the first to attempt to codify —right thinkin, that is irrefutable reasoning processes. His syllogism provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example, Socrates is a man; all men are mortal; therefore Socrates is mortal.‖.
These laws of thought were supposed to govern the operation of the mind; their study initiated a field called logic.
(d)Acting rationally : The rational agent approach
An agent is something that acts. Computer agents are not mere programs ,but they are expected to have the following attributes also :
(a) operating under autonomous control,
(b) perceiving their environment,
(c) persisting over a prolonged time period,
(e) adapting to change.
A rational agent is one that acts so as to achieve the best outcome.

The 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.

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.

A1 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.

A1 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.

The state of art:
What can AI do today?

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

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

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

Diagnosis: Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.

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

Robotics: Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia

et al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of hip replacement prosthesis.

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

INTELLIGENT AGENTS

Agents and environments

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through **actuators.** This simple idea is illustrated in Figure 1.

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

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

o 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.

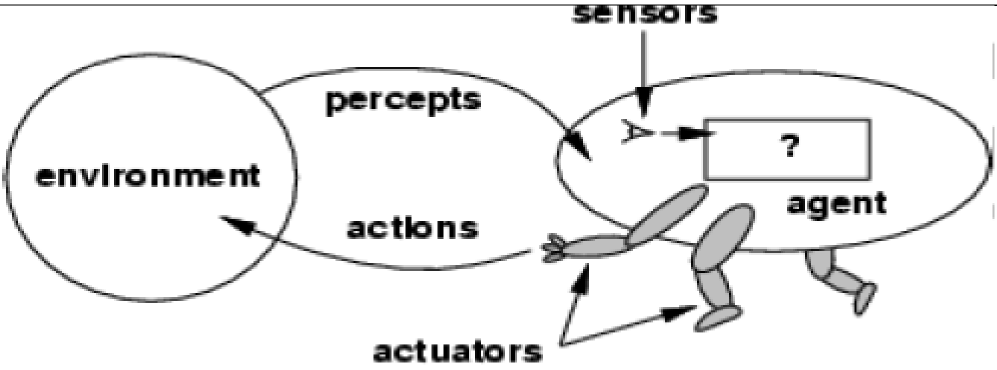


Figure 1 Agents interact with environments through sensors and actuators

Percept

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

Percept Sequence

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

Agent function

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

Agent program

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

Rational Agent

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

Performance measures

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

What is rational at any given time depends on four things:
o The performance measure that defines the criterion of success. o The agent's prior knowledge of the environment.

o The actions that the agent can perform.
o The agent's percept sequence to date.
This leads to a definition of a rational agent:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Omniscience, learning, and autonomy

An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality. Doing actions in order to modify future precepts-sometimes called information gathering-is an important part of rationality.

Our definition requires a rational agent not only to gather information, but also to learn as much as possible from what it perceives.

To the extent that an agent relies on the prior knowledge of its designer rather than on its own precepts, we say that the agent lacks autonomy. A rational agent should be autonomous-it should learn what it can to compensate for partial or incorrect prior knowledge.

Task environments

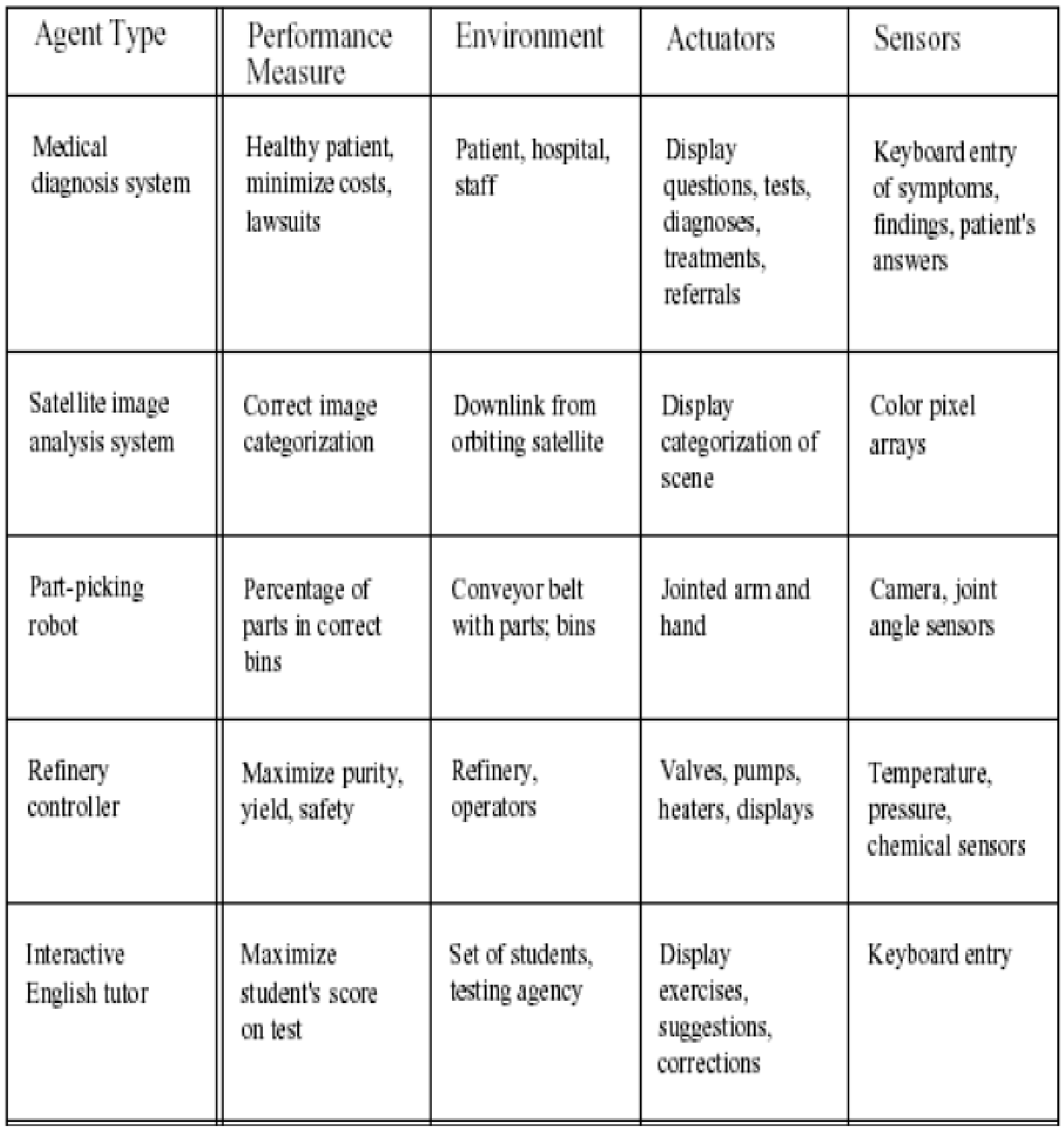
We must think about task environments, which are essentially the "problems" to which rational agents are the "solutions."

Specifying the task environment
The rationality of the simple vacuum-cleaner agent, needs specification of o the performance measure
o the environment
o the agent's actuators and sensors.

PEAS

All these are grouped together under the heading of the task environment. We call this the PEAS (Performance, Environment, Actuators, Sensors) description.

In designing an agent, the first step must always be to specify the task environment as fully as possible.



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

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Explain Artificial Intelligence. Also discuss the areas where it can be used.	June 2014	7
Q.2	What are the characteristics of AI problems? Explain the areas where AI can be applied	June,2010	6
Q.3	Define artificial intelligence. Also tell the areas where it can be used.	June,2005	10

Unit-01/Lecture-02

Various types of production systems

A Knowledge representation formalism consists of collections of condition-action rules (Production Rules or Operators), a database which is modified in accordance with the rules, and a Production System Interpreter which controls the operation of the rules i.e The 'control mechanism' of a Production System, determining the order in which Production Rules are fired.

A system that uses this form of knowledge representation is called a production system.

A production system consists of rules and factors. Knowledge is encoded in a declarative from which comprises of a set of rules of the form

Situation ------------ Action

SITUATION that implies ACTION.

Example:-

IF the initial state is a goal state THEN quit.

The major components of an AI production system are

i. A global database

ii. A set of production rules and

iii. A control system

The goal database is the central data structure used by an AI production system. The production system. The production rules operate on the global database. Each rule has a precondition that is either satisfied or not by the database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the database. The control system chooses which applicable rule should be applied and ceases computation when a termination condition on the database is satisfied. If several rules are to fire at the same time, the control system resolves the conflicts.

Four classes of production systems:-

1. A monotonic production system

2. A non monotonic production system

3. A partially commutative production system

4. A commutative production system.

Advantages of production systems:-

1. Production systems provide an excellent tool for structuring AI programs.

2. Production Systems are highly modular because the individual rules can be added, removed or modified independently.

3. The production rules are expressed in a natural form, so the statements contained in the knowledge base should the a recording of an expert thinking out loud.

Disadvantages of Production Systems:-

One important disadvantage is the fact that it may be very difficult analyse the flow of control within a production system because the individual rules don’t call each other.

Production systems describe the operations that can be performed in a search for a solution to the problem. They can be classified as follows.

Monotonic production system :- A system in which the application of a rule never prevents the later application of another rule, that could have also been applied at the time the first rule was selected.

Partially commutative production system:-

A production system in which the application of a particular sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state x into state Y.

Theorem proving falls under monotonic partially communicative system. Blocks world and 8 puzzle problems like chemical analysis and synthesis come under monotonic, not partially commutative systems. Playing the game of bridge comes under non monotonic , not partially commutative system.

For any problem, several production systems exist. Some will be efficient than others. Though it may seem that there is no relationship between kinds of problems and kinds of production systems, in practice there is a definite relationship.

Partially commutative, monotonic production systems are useful for solving ignorable problems. These systems are important for man implementation standpoint because they can be implemented without the ability to backtrack to previous states, when it is discovered that an incorrect path was followed. Such systems increase the efficiency since it is not necessary to keep track of the changes made in the search process.
Monotonic partially commutative systems are useful for problems in which changes occur but can be reversed and in which the order of operation is not critical (ex: 8 puzzle problem).
Production systems that are not partially commutative are useful for many problems in which irreversible changes occur, such as chemical analysis. When dealing with such systems, the order in which operations are performed is very important and hence correct decisions have to be made at the first time itself.

Characteristics of production systems

- Separation of Knowledge (the Rules) and Control (Recognize-Act Cycle)
- Natural Mapping onto State Space Search (Data or Goal Driven)
- Modularity of Production Rules (Rules represent chunks of knowledge)
- Pattern-Directed Control (More flexible than algorithmic control)
- Opportunities for Heuristic Control can be built into the rules
- Tracing and Explanation (Simple control, informative rules)
- Language Independence
- Model for Human Problem Solving (SOAR, ACT*)

8 Puzzle Problem

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.

The program is to change the initial configuration into the goal configuration. A solution to the problem is an appropriate sequence of moves, such as “move tiles 5 to the right, move tile 7 to the left, move tile 6 to the down, etc”.

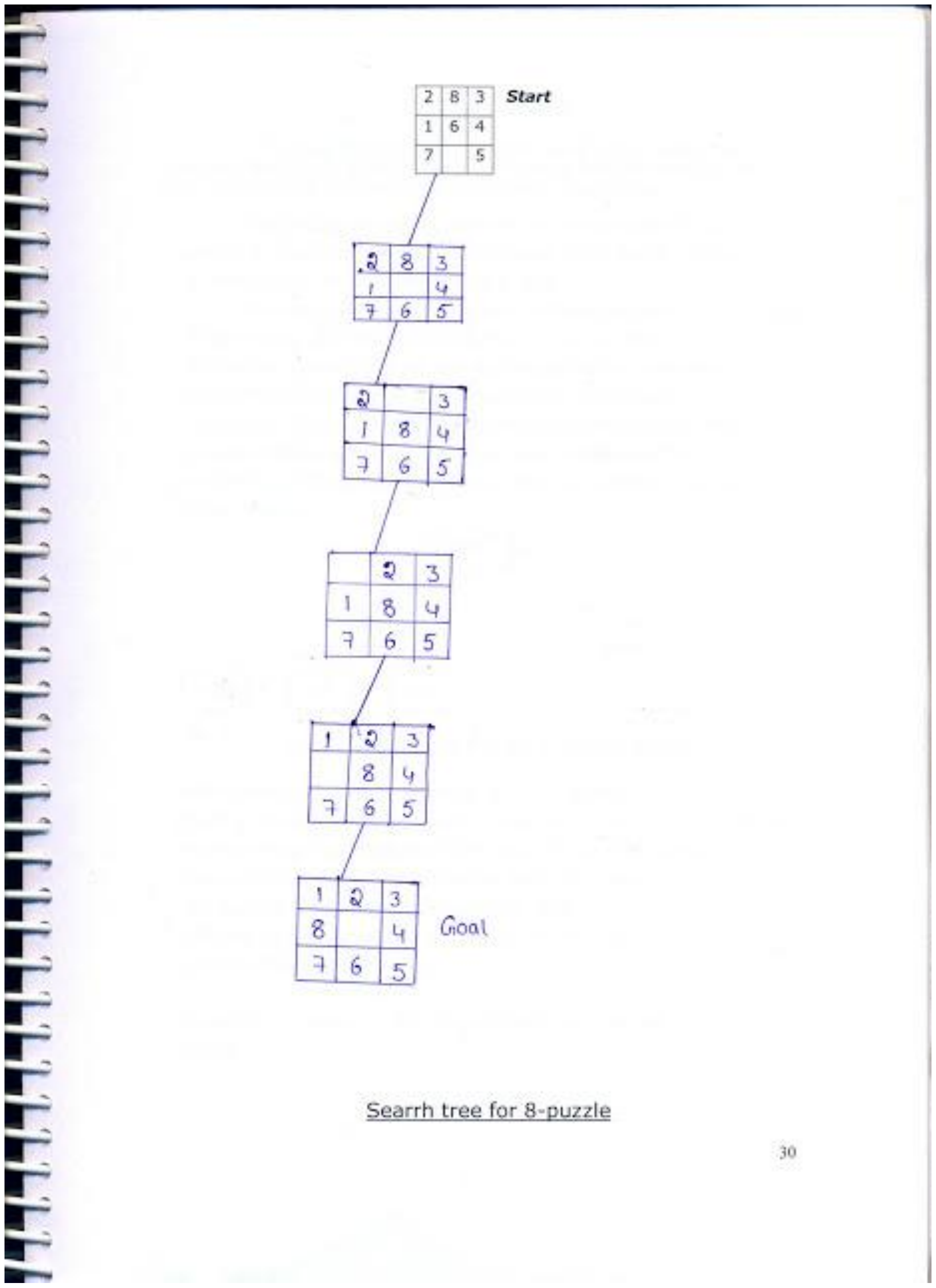
To solve a problem using a production system, we must specify the global database the rules, and the control strategy. For the 8 puzzle problem that correspond to these three components. These elements are the problem states, moves and goal. In this problem each tile configuration is a state. The set of all configuration in the space of problem states or the problem space, there are only 3,62,880 different configurations o the 8 tiles and blank space. Once the problem states have been conceptually identified, we must construct a computer representation, or description of them. This description is then used as the database of a production system. For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. The initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.

A move transforms one problem state into another state. The 8-puzzle is convenjently interpreted as having the following for moves. Move empty space (blank) to the left, move blank up, move blank to the right and move blank down,. These moves are modelled by production rules that operate on the state descriptions in the appropriate manner.

The rules each have preconditions that must be satisfied by a state description in order for them to be applicable to that state description. Thus the precondition for the rule associated with “move blank up” is derived from the requirement that the blank space must not already be in the top row.

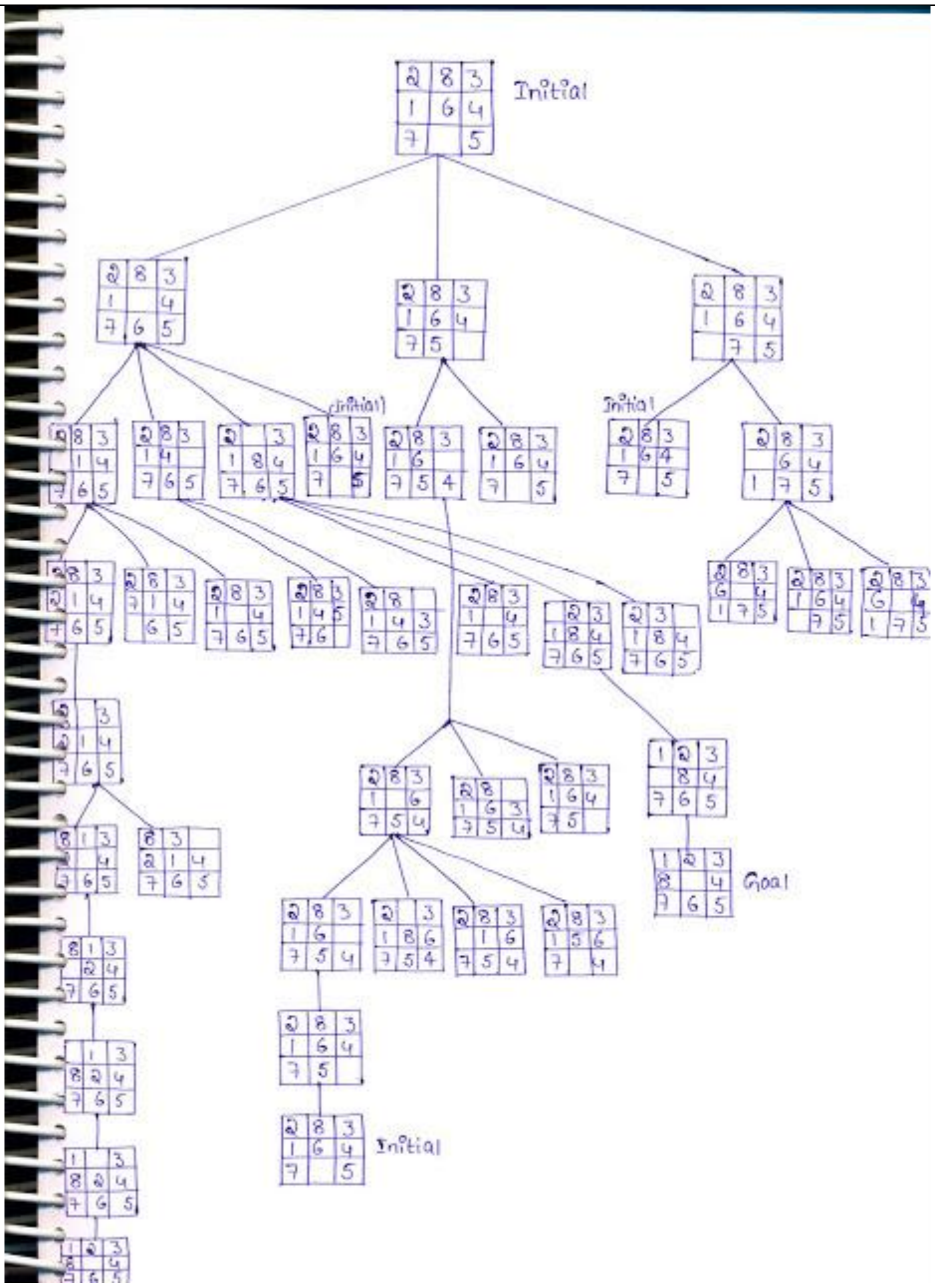
The problem goal condition forms the basis for the termination condition of the production system. The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced. It also keeps track of rules that have been applied so that it can compose them into sequence representing the problem solution. A solution to the 8-puzzle problem is given in the following figure.

8 – puzzle problem is shown in following diagrams.



Searrh tree for 8-puzzle

30


Initial
(Initial)
Initial
Goal
Initial

Water-Jug Problem

Statement: - We are given 2 jugs, a 4 litter one and a 3- litter one. Neither have any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can we get exactly 2 liters of water in to the 4-liter jugs?

Solution:-

The state space for this problem can be defined as

{ (i ,j) i = 0,1,2,3,4 j = 0,1,2,3}

'i' represents the number of liters of water in the 4-liter jug and 'j' represents the number of liters of water in the 3-liter jug. The initial state is (0, 0) that is no water on each jug. The goal state is to get (2, n) for any value of 'n'.

To solve this we have to make some assumptions not mentioned in the problem. They are

1. We can fill a jug from the pump.

2. We can pour water out of a jug to the ground.

3. We can pour water from one jug to another.

4. There is no measuring device available.

The various operators (Production Rules) that are available to solve this problem may be stated as given in the following figure.

Rule No	Production Rule	Action
1	(i, j) ——— (4,j) if i<4.	Fill the 4-liter jug, if 4-liter jug is not full.
2	(i, j) ——— (i,3) if j<3.	Fill the 3-liter jug, if 3-liter jug is not full.
3	(i, j) ———(i-s,j) if i>0.	Pour some water our of the 4-liter jug, if 4-liter jug is not empty.
4	(i, j) ———(i,j-s) if j>0.	Pour some water out of the 3-liter jug, if 3-liter jug is not empty.
5	(i, j) ——— (0,j) if i>0.	Empty the 4-liter jug on the ground, if 4-liter jug is not empty.
6	(i, j) ——— (i,0) if j>0.	Empty the 3-liter jug on the ground, if 3-liter jug is not empty.
7	(i, j) ———(4,j-(4-i)) if (i+j)>=4 & j>0.	Pour water from the 30-liter jug into the 4-liter jug until the 4 liter jug is full, if the combined content is >= 4 and 3-liter jug is not empty.
8	(i, j) ———(i,(3-j),3) if (i+j) >= 3 & i>0.	Pour water from the 4-liter the into the 3 liter jug until the 3-liter jug is full if the combined content is >=3 and 4-liter jug is not empty.
9	(i, j) ——— (i+j,0) if (i+j)<=4 and j>0.	Pour all the water from the 3-liter jug into the 4-liter jug if the jug, combined content is <=4 and 3-liter jug is not empty.
10	(i, j) _____(0,i+j) if i+j)<=3 and i>0	Pour all the water from the 4-liter jug into the3-liter jug, if the combined content is <=3 and 4-liter jug is not empty

For the water jug problem , there are several sequence of operators that will solve the problem . let us see of them.

SOLUTION 1;-

Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
4	0	1
1	3	8
1	0	6
0	1	10
4	1	1
2	3	8

Solution 2:-

Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9

Solution 3:-

Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
4	0	1
1	3	8
0	3	5
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9

Figures gives comparative study of the above 3 different solutions.

Fig :- comparison of 3 solutions.

We see that, when there is no limit for water prevails then solution is the most efficient. When water is limited then Solution2 is the best suited. In no way, solution 3 is good, Because it it requires 8 steps to solution and wastes 5 liters of water.

The Missionaries and Cannibals Problem Statement

Three missionaries and three cannibals find themselves on one side of a river. They have would like to get to the other side. But the missionaries are not sure what else the cannibals agreed to. So the missionaries managed the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side. The only boar available holds only two at a time. How can everyone get across the river without the missionaries risking being eaten?

Solution:-
The state for this problem can be defined as
{(i, j)/ i=0, 1, 2, 3, : j=0, 1, 2, 3} where i represents the number missionaries in one side of a river . j represents the number of cannibals in the same side of river. The initial state is (3,3), that is three missionaries and three cannibals on one side of a river , (Bank 1) and (0,0) on another side of the river (bank 2) . The goal state is to get (3,3) at bank 2 and (0,0) at bank 1.

To sole this problem we will make the following assumptions:

1. Number of cannibals should lesser than the missionaries on either side.

2. Only one boat is available to travel.

3. Only one or maximum of two people can go in the boat at a time.

4. All the six have to cross the river from bank.

5. There is no restriction on the number of trips that can be made to reach of the goal.

6. Both the missionaries and cannibals can row the boat.

Production Rules for Missionaries and Cannibals Problem.

Rule No	Production Rule and Action
1	(i,j) : Two missionaries can go only when i-2 >=j or i-2=0 in one bank and i+2>=j in the other bank.
2	(i,j) : Two cannibals can cross the river only when j-2, <=i or i=0 in one bank and j+2 <=i or I+0 or i=0 in the other.
3	(i,j) : One missionary and one cannibal can go in a goat only when i-1>= j-1 or i=0 in one bank and i+1 >= j+1 or i=0 in the other.
4	(i,j) : one missionary can cross the river only when i-1>=j or i=0 in one bank and i+1 >=j in the other bank.
5	(i,j) : One cannibal can cross the river only when j-1 < i or i=0 in one bank and j+1<=i or j=0 in the other bank of the river.

Fig:- Production rules for the missionaries and cannibals problem.

For this problem, there are several sequences of operators that will solve the problem . The next figure is one of the solutions.

PRODUCTION SYSTEMS

A production system consists of rules and factors. Knowledge is encoded in a declarative from which comprises of a set of rules of the form

Situation ------------ Action

Bank 1	Boat	Bank 2	Production rule applied
(3,3)	(0,2)	(0,0)	
(3,1)	(0,1)	(0,2)	------------- 2
(3,2)	(0,2)	(0,1)	------------- 5
(3,0)	(0,1)	(0,3)	------------- 2
(3,1)	(2,0)	(0,2)	------------- 5
(1,1)	(1,1)	(2,2)	------------- 1
(2,2)	(2,0)	(1,1)	------------- 3
(0,2)	(0,1)	(3,1)	--------------1
(0,3)	(0,2)	(3,0)	------------- 5
(0,1)	(0,1)	(3,2)	--------------2
(0,2)	(0,2)	(3,1)	------------- 5
(0,0)		(3,3)	------------- 2

Fig:- One solution to missionaries and cannibals problem

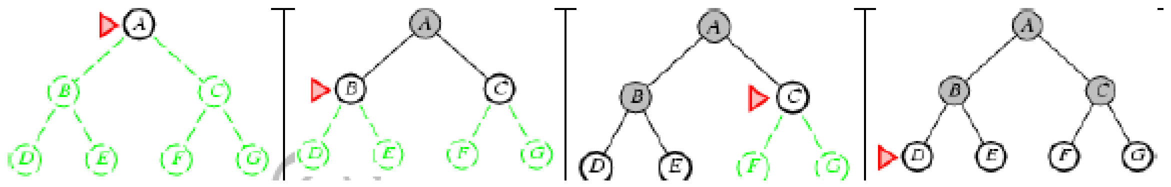
	RGPV QUESTIONS	Year	Marks
Q.1	Discuss the characteristics of production systems	June.2014	7
Q.2	Define the various types of production system & give its characteristics	June.2013	10
Q.3	You are given 2 jugs, a 4 gallon jug and a 3 gallon one. Neither have any measuring markers on it. There is a pump that can be used to fill the jug with water. How can you get exactly 2 gallon of water in the 4 gallon jug ? write the production rule for the above problem.	June.2012	10
Q.4	Explain the purpose of production system with examples. Also explain the various types of production system.	June.2012	10
Q.5	Explain various types of production systems with their characteristics	June.2011	10
Q.6	Find the state space representation of 8-Puzzle problem	June.2011	10
Q.7	Explain the purpose of production system. What are its characteristics? Mention different types of production system with brief explanation. https://www.rgpvonline.com	June.2006	10

Unit-01/Lecture-03

Breadth First Search (BFS)

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

Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling TREE-SEARCH (problem, FIFO-QUEUE ()) results in breadth-first-search. The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.



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

Properties of breadth-first-search

Complete?? Yes (if b is finite)

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

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

Optimal?? No, unless step costs are constant

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

Time complexity for BFS

Assume every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b2 at the second level. Each of these generates b more nodes, yielding b3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, we would expand all but the last node at level d, generating bd+1 - b nodes at level d+1.

Then the total number of nodes generated is b + b2 + b3 + …+ bd + (bd+1 + b) = O(bd+1).

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space complexity is, therefore ,the same as the time complexity

Depth first search

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure 1.31. 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.

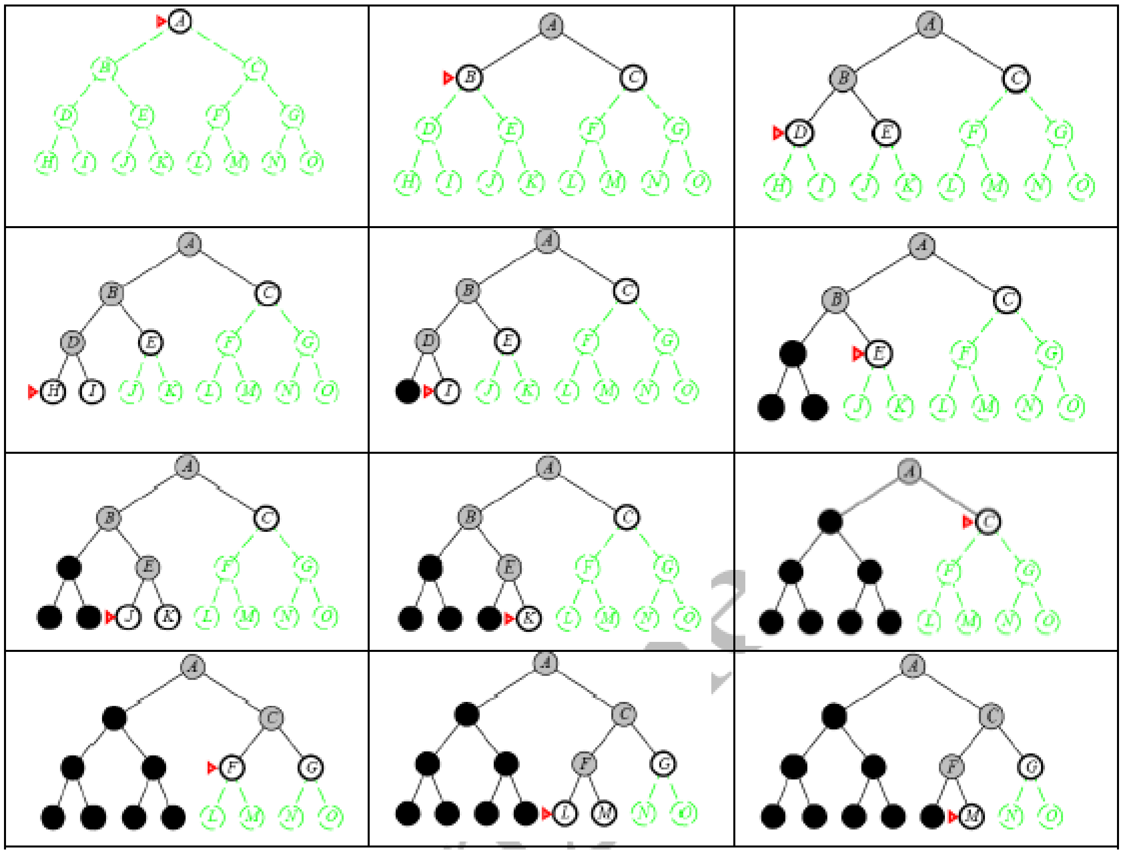


Fig. Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

Depth-first-search has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored

For a state space with a branching factor b and maximum depth m,depth-first-search requires storage of only bm + 1 nodes.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

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

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Compare breadth first search and depth first search techniques.	June.2014	7
Q.2	Explain best first, depth first and breadth first search. When would best first search be worse than simple breadth first search?	June.2011, Dec.2009	10
Q.3	Differentiate between Breadth First Search and Depth First Search.	June.2007	7

Unit-01/Lecture-04

INFORMED SEARCH AND EXPLORATION

Informed(Heuristic) Search Strategies

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

Best-first search Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function f(n). The node with lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal.

This can be implemented using a priority-queue, a data structure that will maintain the fringe in ascending order of f-values.

Heuristic functions

A Heuristic technique helps in solving problems, even though there is no guarantee that it will never lead in the wrong direction. There are heuristics of every general applicability as well as domain specific. The strategies are general purpose heuristics. In order to use them in a specific domain they are coupler with some domain specific heuristics. There are two major ways in which domain - specific, heuristic information can be incorporated into rule-based search procedure.

- In the rules themselves
- As a heuristic function that evaluates individual problem states and determines how desired they are.

A heuristic function is a function that maps from problem state description to measures desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution. Well designed heuristic functions can provide a fairly good estimate of whether a path is good or not. (“The sum of the distances travelled so far" is a simple heuristic function in the travelling salesman problem). The purpose of a heuristic function is to guide the search process in the most profitable directions, by suggesting which path to follow first when more than one path is available. However in many problems, the cost of computing the value of a heuristic function would be more than the effort saved in the search process. Hence generally there is a trade-off between the cost of evaluating a heuristic function and the savings in search that the function provides.

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

The key component of Best-first search algorithm is a heuristic function, denoted by h(n):

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

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via a straight-line distance from Arad to Bucharest.

Heuristic functions are the most common form in which additional knowledge is imparted to the search algorithm.

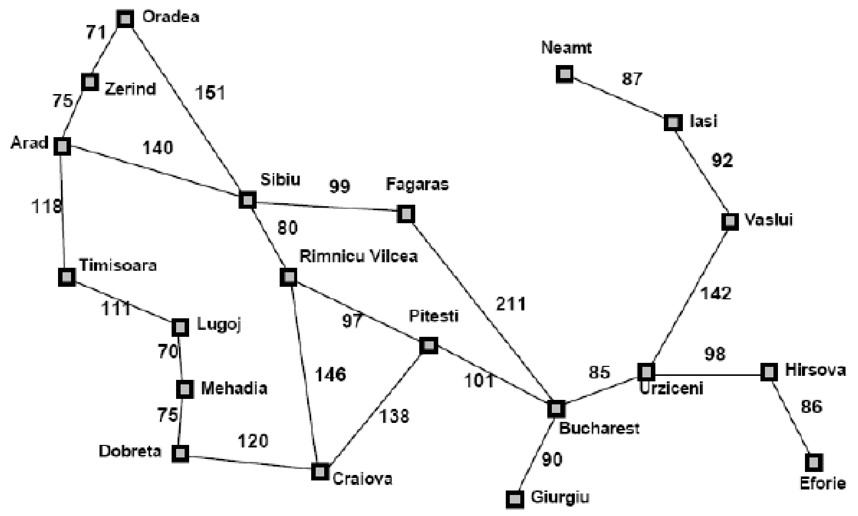
Greedy Best-first search

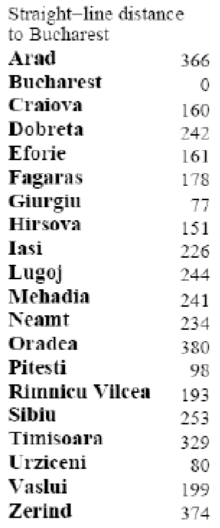
Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to a solution quickly.

It evaluates the nodes by using the heuristic function f(n) = h(n).

Taking the example of Route-finding problems in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities For example, the initial state is In(Arad) ,and the straight line distance heuristic hSLD(In(Arad)) is found to be 366.

Using the straight-line distance heuristic hSLD ,the goal state can be reached faster





Straight–line distance to Bucharest

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

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

Fig. stages in greedy best-first search for Bucharest using straight-line distance heuristic hSLD. Nodes are labelled with their h-values.

Below Figure shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal.

Properties of greedy search

o Complete?? No-can get stuck in loops, e.g., Iasi ! Neamt ! Iasi ! Neamt !

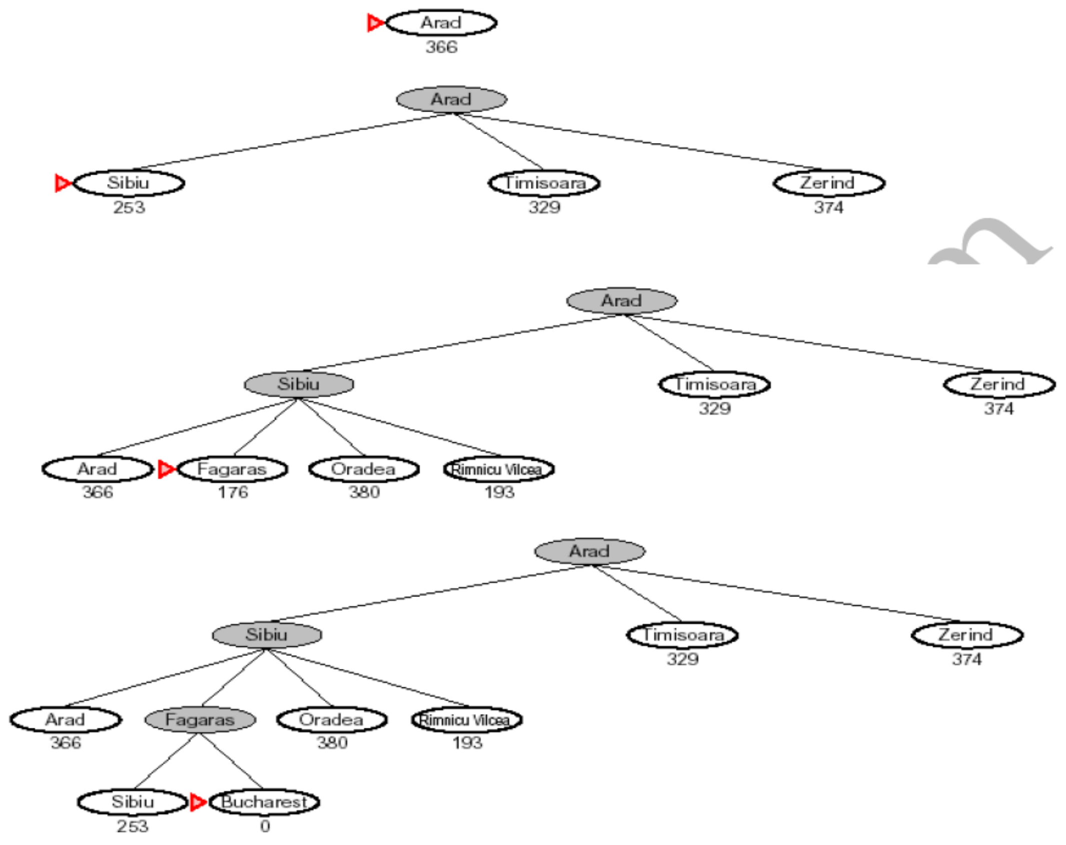
Complete in finite space with repeated-state checking

o Time?? O(bm), but a good heuristic can give dramatic improvement o Space?? O(bm)– keeps all nodes in memory

o Optimal?? No

Greedy best-first search is not optimal, and it is incomplete.

The worst-case time and space complexity is O(bm),where m is the maximum depth of the search space.



A* Search

A* Search is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining

(1) g(n) = the cost to reach the node, and

(2) h(n) = the cost to get from the node to the goal : f(n) = g(n) + h(n).

A* Search is both optimal and complete. A* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance hSLD. It cannot be an overestimate.

A Search is optimal if h(n) is an admissible heuristic – that is, provided that h(n) never overestimates the cost to reach the goal.

An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest. The values of _g _ are computed from the step costs shown in the Romania map Also the values of hSLD are given in Figure.

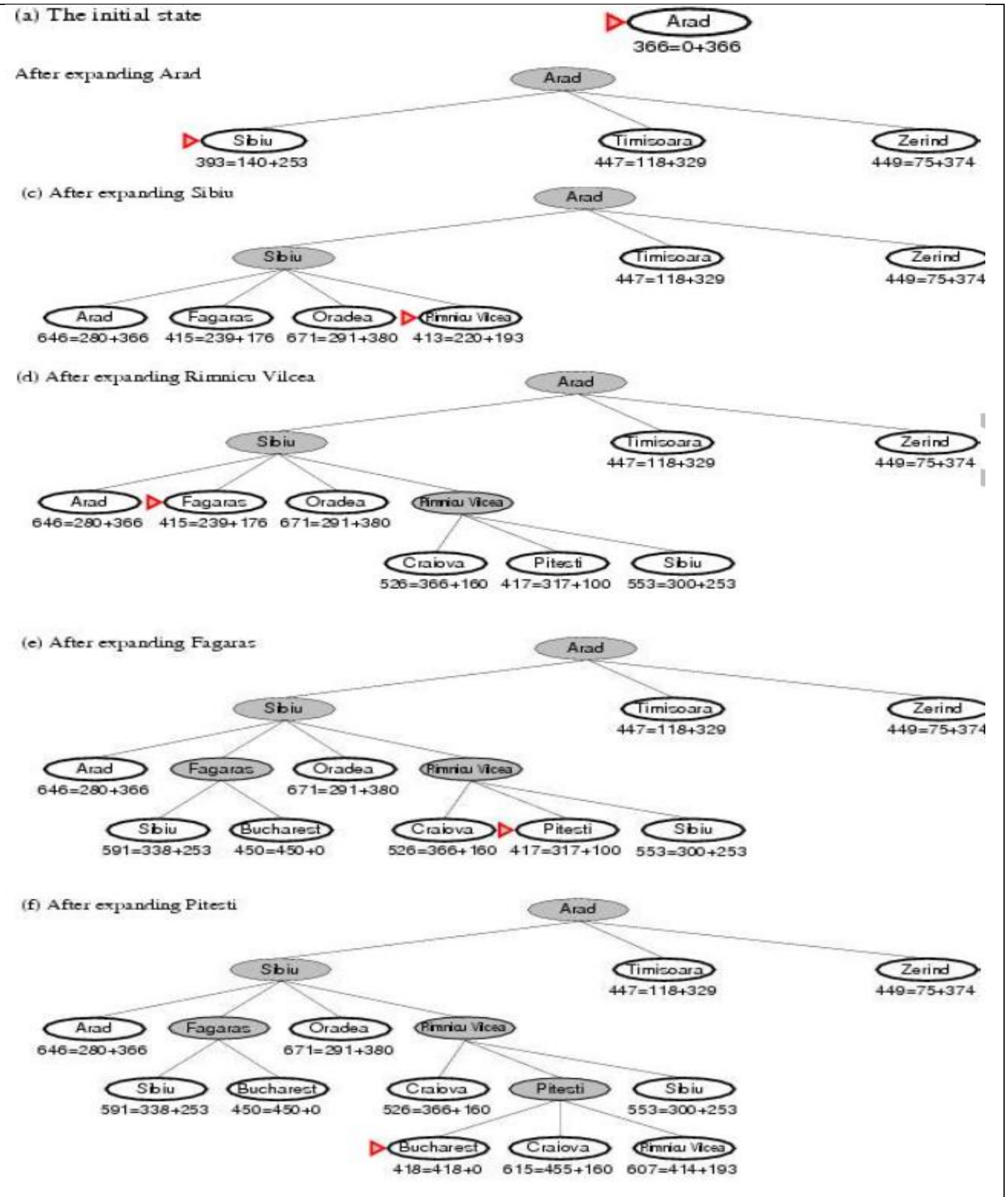


Fig. Stages in A* Search for Bucharest. Nodes are labelled with f = g + h . The h-values are the straight-line distances to Bucharest taken from previous figure

S.NO	RGPV QUESTIONS	Year	Marks

Q.1	What do you understand by heuristic? Explain hill climbing method compare it with generate and test method	June.2014	7
Q.2	Why heuristic search techniques are considered to be more powerful than the traditional search techniques?	June.2011	10
Q.3	What do you understand by heuristic? Explain hill climbing method compare it with generate and test method	June.2006	10

Unit-01/Lecture-05

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

o In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

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

o In such cases, we can use local search algorithms. They operate using a single current state(rather than multiple paths) and generally move only to neighbours of that state.

o The important applications of these class of problems are (a) integrated-circuit design,(b)Factory-floor layout,(c) job-shop scheduling,(d)automatic programming,(e)telecommunications network optimization,(f)Vehicle routing, and (g) portfolio management.

Key advantages of Local Search Algorithms

(1) They use very little memory – usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS

In addition to finding goals, local search algorithms are useful for solving pure optimization problems,in which the aim is to find the best state according to an objective function.

State Space Landscape

To understand local search,it is better explained using state space landscape as shown in figure.

A landscape has both ―location‖ (defined by the state) and ―elevation‖(defined by the value of the heuristic cost function or objective function).

If elevation corresponds to cost,then the aim is to find the lowest valley – a global minimum; if elevation corresponds to an objective function,then the aim is to find the highest peak – a global maximum.

Local search algorithms explore this landscape. A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

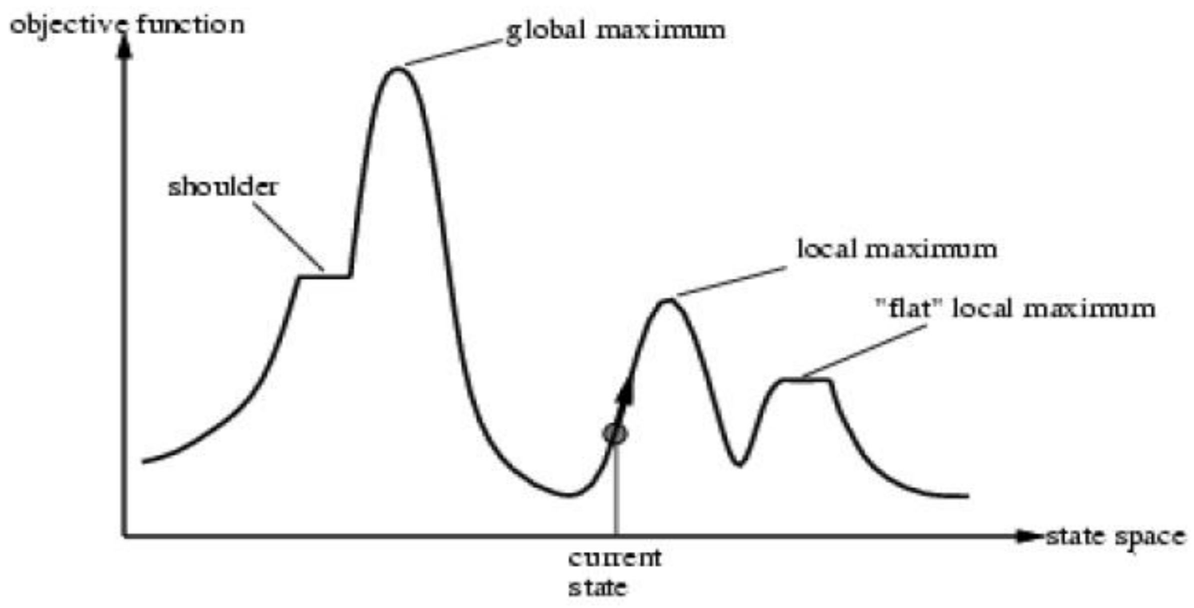


Fig. A one dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it ,as shown by the arrow. The various topographic features are defined in the text

Hill-climbing search

The hill-climbing search algorithm as shown in figure, is simply a loop that continually moves in the direction of increasing value – that is, uphill. It terminates when it reaches a —peak|| where no neighbour has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local maximum
    input: problem, a problem
    local variables: current, a node.
                     neighbor, a node.

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest valued successor of current
        if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Fig. The hill-climbing search algorithm (steepest ascent version), which is the most basic local search technique. At each step the current node is replaced by the best neighbour; the neighbour with the highest VALUE. If the heuristic cost estimate h is used, we could find the neighbour with the lowest h.

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well. Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons :

o Local maxima : a local maximum is a peak that is higher than each of its neighbouring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go

o Ridges : A ridge is shown in Figure 2.10. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

o Plateaux : A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.

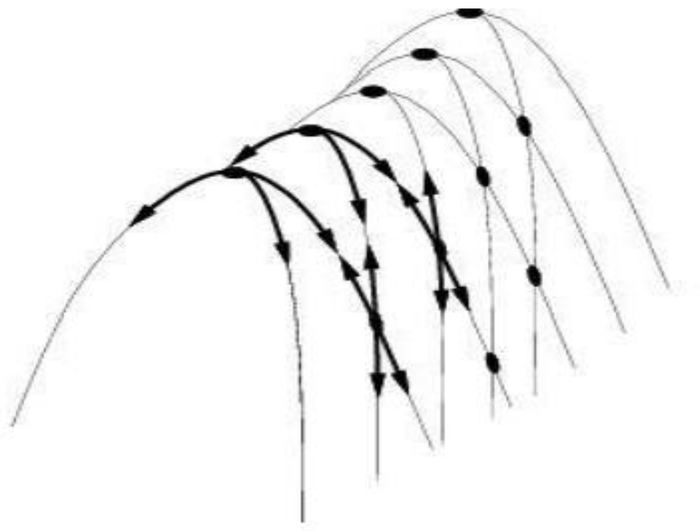


Fig. Illustration of why ridges cause difficulties for hill-climbing.

The grid of states(dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available options point downhill.

Hill-climbing variations

☐Stochastic hill-climbing

o Random selection among the uphill moves.

o The selection probability can vary with the steepness of the uphill move.

☐First-choice hill-climbing

o cfr. stochastic hill climbing by generating successors randomly until a better one is found.

☐Random-restart hill-climbing

o Tries to avoid getting stuck in local maxima. Simulated annealing search

A hill-climbing algorithm that never makes ―downhill‖ moves towards states with lower value(or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient.

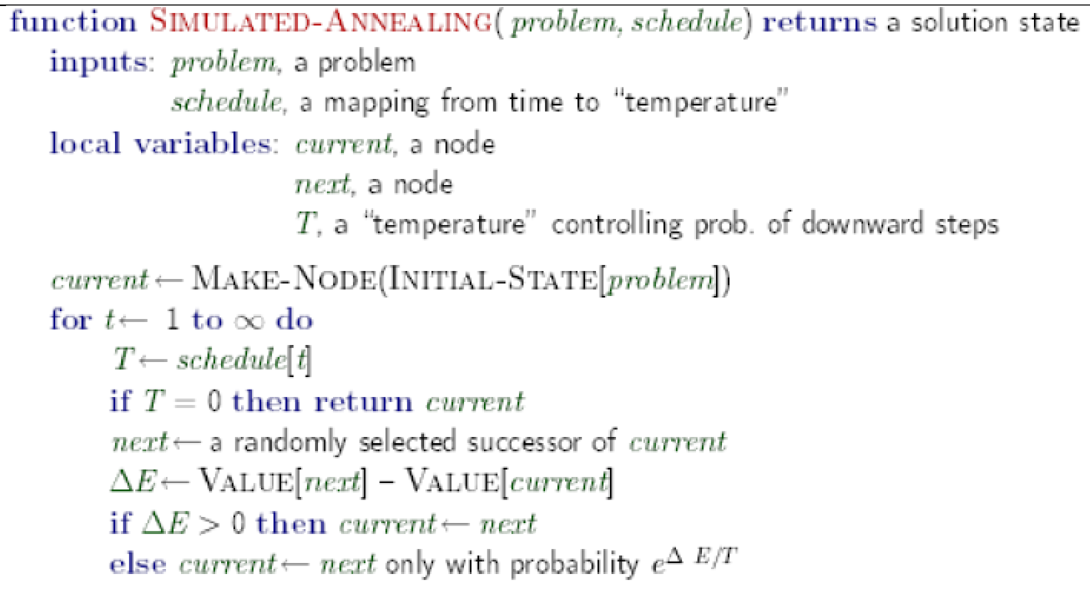
Simulated annealing is an algorithm that combines hill-climbing with a random walk in someway that yields both efficiency and completeness.

Figure shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the ―badness‖ of the move – the amount E by which the evaluation is worsened.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

The SIMULATED ANNEALING algorithm

```
1 Evaluate the initial state
2 If it is goal state Then quit
        otherwise make the current state this initial state and proceed;
3 Make BESTSOFAR to current state
4 Set T from the annealing schedule
5 Repeat
        select an operator that can be applied to the current state
        apply the new operator and create a new state
        evaluate this new state
        find [[Delta]]E = difference between the values of current and new states
        If this new state is goal state Then quit
        Otherwise compare with the current state
        If better set BESTSOFAR to the value of this state and make the current the new
        state
        If it is not better then make it the current state with probability p'. This involves
        generating a random number in the range 0 to 1 and comparing it with a half, if it is
        less than a half do nothing and if it is greater than a half accept this state as the next
        current be a half.
        Revise T in the annealing schedule dependent on number of nodes in tree
        Until a solution is found or no more new operators
6 Return BESTSOFAR as the answer
```



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

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

Fig. The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What do you understand by heuristic? Explain hill climbing method compare it with generate and test method	June.2006	10

UNIT-01/LECTURE-06

HILL CLIMBING PROCEDURE

This is a variety of depth-first (generate - and - test) search. A feedback is used here to decide on the direction of motion in the search space. In the depth-first search, the test function will merely accept or reject a solution. But in hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state. The hill climbing test procedure is as follows :

1. General he first proposed solution as done in depth-first procedure. See if it is a solution. If so quit , else continue.

2. From this solution generate new set of solutions use , some application rules

3. For each element of this set

(i) Apply test function. It is a solution quit.

(ii) Else see whether it is closer to the goal state than the solution already generated. If yes, remember it else discard it.

4. Take the best element so far generated and use it as the next proposed solution.

This step corresponds to move through the problem space in the direction Towards the goal state.

5. Go back to step 2.

Sometimes this procedure may lead to a position, which is not a solution, but from which there is no move that improves things. This will happen if we have reached one of the following three states.

(a) A "local maximum” which is a state better than all its neighbours, but is not better than some other states farther away. Local maxim sometimes occurs within sight of a solution. In such cases they are called “Foothills".

(b) A "plateau'' which is a flat area of the search space, in which neighbouring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

(c) A "ridge" which is an area in the search that is higher than the surrounding areas, but cannot be searched in a simple move.

To overcome these problems we can

(a) Back track to some earlier nodes and try a different direction. This is a good way of dealing with local maxim.

(b) Make a big jump an some direction to a new area in the search. This can be done by applying two more rules of the same rule several times, before testing. This is a good strategy is dealing with plate and ridges.

Hill climbing becomes inefficient in large problem spaces, and when combinatorial explosion occurs. But it is a useful when combined with other methods.

Steepest Ascent Hill Climbing

This differs from the basic Hill climbing algorithm by choosing the best successor rather than the first successor that is better. This indicates that it has elements of the breadth first algorithm.

The algorithm proceeds

Steepest ascent Hill climbing algorithm

```
1 Evaluate the initial state
2 If it is goal state Then quit
otherwise make the current state this initial state and proceed;
3 Repeat
set Target to be the state that any successor of the current state can better;
for each operator that can be applied to the current state
apply the new operator and create a new state
evaluate this state
If this state is goal state Then quit
Otherwise compare with Target
If better set Target to this value
If Target is better than current state set current state to Target
Until a solution is found or current state does not change
```

Both the basic and this method of hill climbing may fail to find a solution by reaching a state from which no subsequent improvement can be made and this state is not the solution.

Local maximum state is a state which is better than its neighbours but is not better than states faraway. These are often known as foothills. Plateau states are states which have approximately the same value and it is not clear in which direction to move in order to reach the solution. Ridge states are special types of local maximum states. The surrounding area is basically unfriendly and makes it difficult to escape from, in single steps, and so the path peters out when surrounded by ridges. Escape relies on: backtracking to a previous good state and proceed in a completely different direction--- involves keeping records of the current path from the outset; making a gigantic leap forward to a different part of the search space perhaps by applying a sensible small step repeatedly, good for plateaux; applying more than one rule at a time before testing, good for ridges.
None of these escape strategies can guarantee success.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1		Jun-2012	08
		June.2006 June.2007 June.2009 Dec.2008 Dec.2009	10
		Dec.2013, June-2006,07	8
		Dec.2005 Dec.2006	5

UNIT-01/LECTURE-07

Best first Search

Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.

Judea Pearl described best-first search as estimating the promise of node n by a "heuristic evaluation function f(n) which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."

Some authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first. This specific type of search is called greedy best-first search.

Efficient selection of the current best candidate for extension is typically implemented using a priority queue.

The A* search algorithm is an example of best-first search, as is B*. Best-first algorithms are often used for path finding in combinatorial search. (Note that neither A* nor B* is a greedy best-first search as they incorporate the distance from start in addition to estimated distances to the goal.)
An algorithm implementing best-first search follows.[3]

```
OPEN = [initial state]
while OPEN is not empty or until a goal is found
do
 1. Remove the best node from OPEN, call it n.
 2. If n is the goal state, backtrace path to n (through recorded parents) and return path.
 3. Create n's successors.
 4. Evaluate each successor, add it to OPEN, and record its parent.
done
```

Note that this version of the algorithm is not complete, i.e. it does not always find a possible path between two nodes, even if there is one. For example, it gets stuck in a loop if it arrives at a dead end, that is a node with the only successor being its parent. It would then go back to its parent, add the dead-end successor to the OPEN list again, and so on.

The following version extends the algorithm to use an additional CLOSED list, containing all nodes that have been evaluated and will not be looked at again. As this will avoid any node being evaluated twice, it is not subject to infinite loops.

```
OPEN = [initial state]
CLOSED = []
while OPEN is not empty
do
 1. Remove the best node from OPEN, call it n, add it to CLOSED.
 2. If n is the goal state, backtrace path to n (through recorded parents) and return path.
 3. Create n's successors.
 4. For each successor do:
      a. If it is not in CLOSED and it is not in OPEN: evaluate it, add it to OPEN, and record its parent.
      b. Otherwise, if this new path is better than previous one, change its recorded parent.
         i.  If it is not in OPEN add it to OPEN.
         ii. Otherwise, adjust its priority in OPEN using this new evaluation.
done
```

Also note that the given pseudo code of both versions just terminates when no path is found. An actual implementation would of course require special handling of this case.

Best-first search in its most basic form consists of the following algorithm (adapted from Pearl, 1984):

The first step is to define the OPEN list with a single node, the starting node. The second step is to check whether or not OPEN is empty. If it is empty, then the algorithm returns failure and exits. The third step is to remove the node with the best score, n, from OPEN and place it in CLOSED. The fourth step “expands” the node n, where expansion is the identification of successor nodes of n. The fifth step then checks each of the successor nodes to see whether or not one of them is the goal node. If any successor is the goal node, the algorithm returns success and the solution, which consists of a path traced backwards from the goal to the start node. Otherwise, the algorithm proceeds to the sixth step. For every successor node, the algorithm applies the evaluation function, f, to it, then checks to see if the node has been in either OPEN or CLOSED. If the node has not been in either, it gets added to OPEN. Finally, the seventh step establishes a looping structure by sending the algorithm back to the second step. This loop will only be broken if the algorithm returns success in step five or failure in step two.

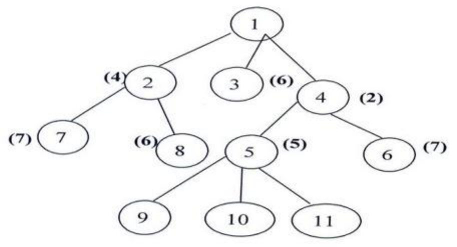
The algorithm is represented here in pseudo-code:

```
1. Define a list, OPEN, consisting solely of a single node, the start node, s.
2. IF the list is empty, return failure.
3. Remove from the list the node n with the best score (the node where f is the minimum), and move it to a list, CLOSED.
4. Expand node n.
5. IF any successor to n is the goal node, return success and the solution (by tracing the path from the goal node to s).
6. FOR each successor node:
    a) apply the evaluation function, f, to the node.
    b) IF the node has not been in either list, add it to OPEN.
7.looping structure by sending the algorithm back to the second step.
```

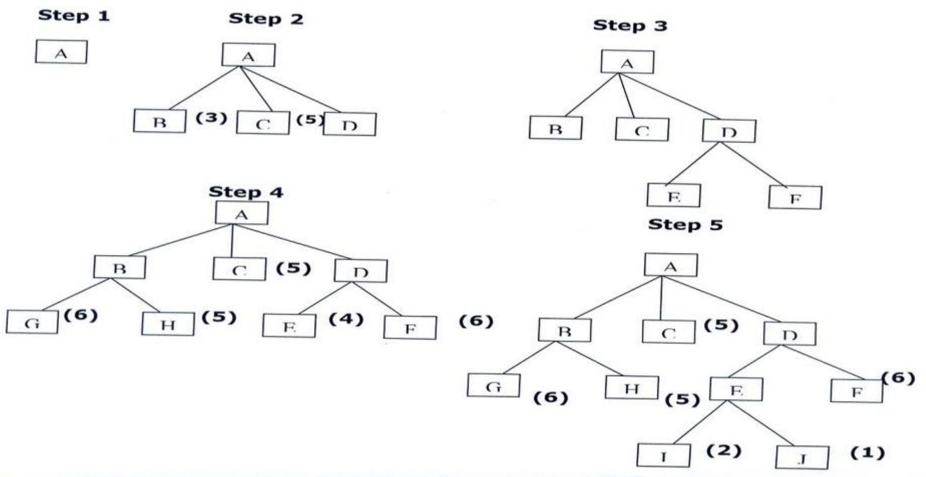
Pearl adds a third step to the FOR loop that is designed to prevent re-expansion of nodes that have already been visited. This step has been omitted above because it is not common to all best-first search algorithms.

BEST - FIRST SEARCH : OR GRAPHS

Best first search combines the advantages of both depth-first and breadth first search methods . it selects the most promising node at each step by applying an approximate heuristic function. The successors of this node are generated using applicable rules. If one of them is a solution , we quit if not all these new nodes are added to the nodes so far generated and the process continues. Figure shows the beginning of a best - first search procedure.



Best –First Search Tree

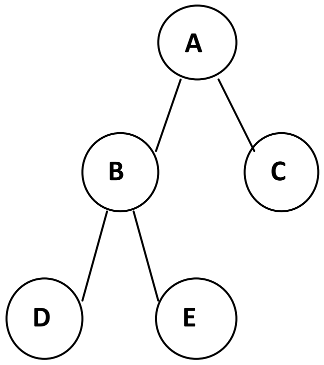


A is an initial node, which is expand to B, C and D. A heuristic function, say cost of reaching the goal, is applied to each of these nodes, since D is most promising; it is expanded next, producing two successor nodes E and F. Heuristic function is applied to them. Now out of the four remaining (B, C and F) B looks more promising and hence it is expand generating nodes G and H. Again when evaluated E appears to be the next stop J has to be expanded giving rise to nodes I and J. In the next step J has to be expanded, since it is more promising. This process continues until a solution is found.

Above figure shows the best - first search tree. Since a search tree may generate duplicate nodes, usually a search graph is preferred.

The best - first search is implemented by an algorithm known as A* algorithm. The algorithm searches a directed graph in which each node represents a point in the problem space. Each node will contain a description of the problem state it represents and it will have links to its parent nodes and successor nodes. In addition it will also indicate how best it is for the search process. A* algorithm uses have been generated, heuristic functions applied to them, but successors not generated. The list CLOSED contains nodes which have been examined, i.e., their successors generated.

A heuristic function f estimates the merits of each generated node. This function f has two components g and h. the function g gives the cost of getting from the initial state to the current node. The function h is an estimate of the addition cost of getting from current node to a goal state. The function f (=g+h) gives the cost of getting from the initial state to a goal state via the current node.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Find the OPEN and CLOSED list nodes for the diagram given below. 	June.2013	10
Q.2	Explain best first, depth first and breadth first search. When would best first search be worse than simple breadth first search?	June.2011	10

UNIT-01/LECTURE-08

A* algorithm

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

It uses a knowledge-plus-heuristic cost function of node x (usually denoted f(x)) to determine the order in which the search visits nodes in the tree. The cost function is a sum of two functions:

the past path-cost function, which is the known distance from the starting node to the current node x (usually denoted g(x))

a future path-cost function, which is an admissible "heuristic estimate" of the distance from x to the goal (usually denoted h(x)).

The h(x) part of the f(x) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

If the heuristic h satisfies the additional condition $h(x) <= d(x,y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once (see closed set below)—and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) := d(x, y) + h(y) - h(x)$.

Process

Like all informed search algorithms, it first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account; the g(x) part of the heuristic is the cost from the starting point, not simply the local cost from the previously expanded node.

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the open set or fringe. The lower f(x) for a given node x, the higher its priority. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

Additionally, if the heuristic is monotonic (or consistent, see below), a closed set of nodes already traversed may be used to make the search more efficient.

Algorithm:

1. Start with OPEN containing the initial node. Its g=0 and f ' = h '

Set CLOSED to empty list.

2. Repeat

If OPEN is empty , stop and return failure

Else pick the BESTNODE on OPEN with lowest f ' value and place it on CLOSED

If BESTNODE is goal state return success and stop

Else

Generate the successors of BESTNODE.

For each SUCCESSOR do the following:

1. Set SUCCESSOR to point back to BESTNODE. (back links will help to recover the path)
2. compute g(SUCCESSOR) = g(BESTNODE) cost of getting from BESTNODE to SUCCESSOR.
3. If SUCCESSOR is the same as any node on OPEN, call that node OLS and add OLD to BESTNODE 's successors. Check g(OLD) and g(SUCCESSOR). It g(SUCCESSOR) is cheaper then reset OLD 's parent link to point to BESTNODE. Update g(OLD) and f '(OLD).
4. If SUCCESSOR was not on OPEN , see if it is on CLOSED . if so call the node CLOSED OLD , and better as earlier and set the parent link and g and f ' values appropriately.
5. If SUCCESSOR was not already on earlier OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE 's successors.

Compute f ' (SUCCESSOR) = g(SUCCESSOR) + h ' (SUCCESSOR)

Best first searches will always find good paths to a goal after exploring the entire state space. All that is required is that a good measure of goal distance be used.

The A* Algorithm

Best first search is a simplified A*.

Start with OPEN holding the initial nodes.

Pick the BEST node on OPEN such that f = g + h' is minimal.

If BEST is goal node quit and return the path from initial to BEST Otherwise

Remove BEST from OPEN and all of BEST's children, labelling each with its path from initial node.

Graceful decay of admissibility

If h' rarely overestimates h by more than d then the A* algorithm will rarely find a solution whose cost is d greater than the optimal solution.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1		Dec.2012 Dec.2009 Jun.2004 Jun.2008	10
Q.2		June.2004 June.2008 Dec.2009	10
Q.3		Dec-2013	7

UNIT-01/LECTURE-09

AO* algorithms

AO* algorithm (pronounced ``A-O-star'') an algorithm similar to A* for heuristic search of an AO* Algorithm

1. Initialise the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value FUTILITY otherwise calculate only f' for each of the successors.
4. If f' is 0 then mark the node as SOLVED
5. Change the value of f' for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as SOLVED then mark the parent node as SOLVED.
7. If starting node is SOLVED or value greater than FUTILITY, stop, else repeat from 2.

AO* Algorithm2.
The AO* ALGORITHM
The problem reduction algorithm we just described is a simplification of an algorithm described in Martelli and Montanari, Martelli and Montanari and Nilson. Nilsson calls it the AO* algorithm , the name we assume.
1. Place the start node s on open.
2. Using the search tree constructed thus far, compute the most promising solution tree T
3. Select a node n that is both on open and a part of T. Remove n from open and place it on closed.
4. If n is a terminal goal node, label n as solved. If the solution of n results in any of n's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where T is the solution tree. Remove from open all nodes with a solved ancestor.
5. If n is not a solvable node (operators cannot be applied), label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of n's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one sub problem, generate their successors to give individual sub problems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h* for each newly generated node and place all such nodes that do not yet have descendents on open. Next, recomputed the values of h* at n and each ancestor of n.
7. Return to step 2.

Problem Reduction with AO* Algorithm.

PROBLEM REDUCTION (AND - OR graphs - AO * Algorithm)

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

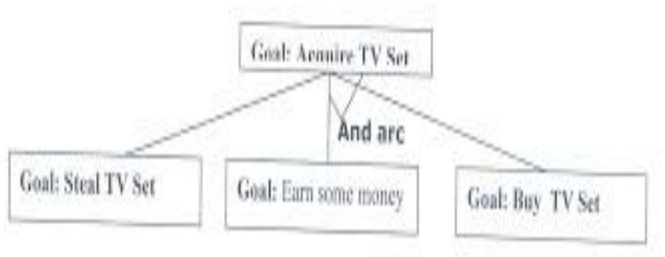


Figure shows AND - Or graph - an example.

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

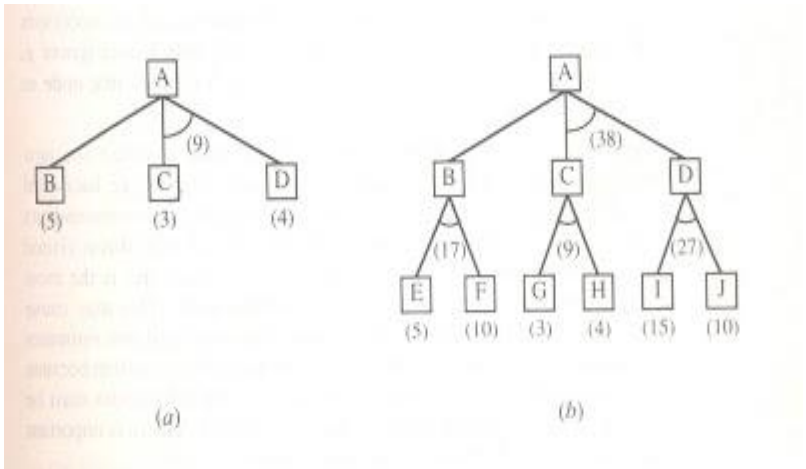


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

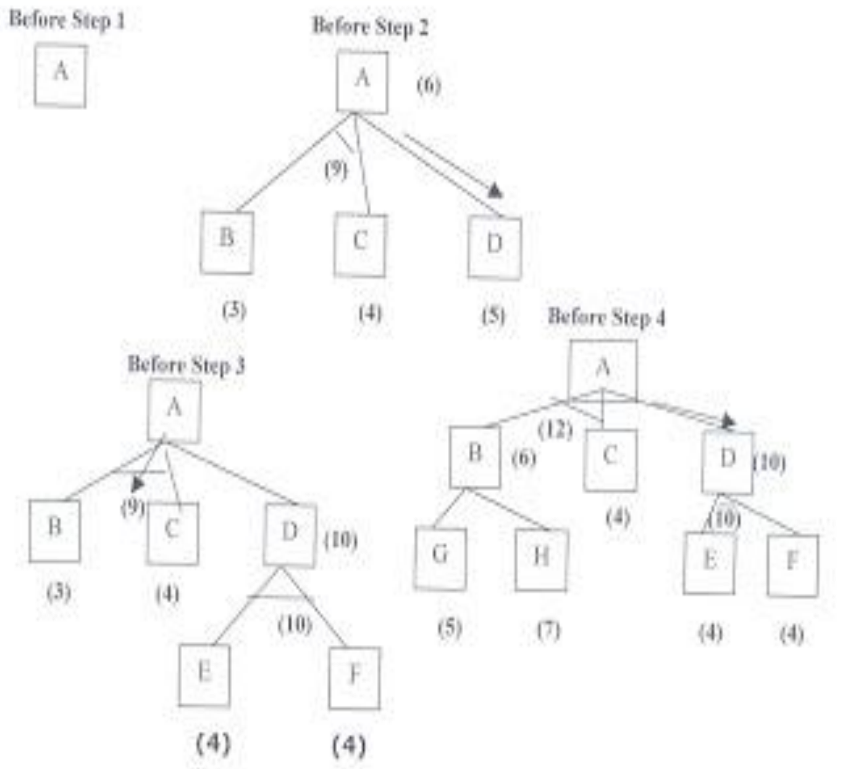
Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current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

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.

2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.

3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

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



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

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.

For representing above graphs AO* algorithm is as follows

AO* ALGORITHM:

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

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

(II) Generate the successors of NODE. if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancester of NODE do the following

(a) add SUCCESSOR to graph G

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

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

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

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

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

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

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

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

AO* Search Procedure.

1. Place the start node on open.
2. Using the search tree, compute the most promising solution tree TP .
3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.
4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.
5. If n is not solvable node, label n as unsolvable. If the start node is labelled as unsolvable, exit with failure. Remove all nodes from open, with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.
7. Go back to step(2)

Note: AO* will always find minimum cost solution.

Various types of control strategies:

We will now consider the problem of deciding which rule to apply next during the process of searching for a solution for a solution. This question arises when more than one rule will have its left side match the current state.

The first requirement of a control strategy is that it must cause motion. The second requirement of a control strategy is that issue must be systematic. We will explain these two with respect to water jug problem. If we have implemented choosing the first operator and then the one which matches the first one, then we would not have solved the problem. If we follow any strategy which can cause some motion then will lead to a solution. But if it is not followed systematically, and then got the solution. One day to follow a systematic control strategy is to construct a tree with the initial state as its root. By applying all possible combinations from the first level leaf nodes. Continue the process until some rule produces a goal state. For the water jug problem a tree can be constructed as given in following diagram.

The control strategy for the search process is called breadth first search. Other systematically control strategies are also available. For example, we can select one single branch of a tree until it yields a solution or until some pre specified depth has been reached. If not we go back and explore to other branches. This is called depth – first – search. The water jug problems will lead to an answer by adoption any control strategy because the problem is simple. This is not always the case.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain AO* algorithm.	June.2014	10
Q.2		June.2004	10