ELECTIVE-I: ARTIFICIAL INTELLIGENCE & ROBOTICS
Sub Code : 18MIT25E

UNIT-I: The AI Problems: AI technique – Criteria for success – Define the Problem as a state space search – Production System – Characteristics – Problem Characteristics.

UNIT-II: Heuristic Search Techniques: Generate and Test – Hill climbing –Best First Search – Problem Reduction – Constraints Satisfactions – Means End Analysis.

UNIT-III: Knowledge Representation Issues: Approaches to knowledge Representation – The Frame Problem – Computable Functions & Predicates – Resolution – Procedural versus Declarative Knowledge.

UNIT-IV: Fundamentals of Robotics: Introduction, classification of Robots, History of Robots, Advantages and Disadvantages of Robot, Robot components, Robot degree of freedom, Robot joints and coordinates, Robot workspace, Robot reach, Robot languages.

UNIT-V-:Sensors: Introduction to internal and external sensors of the Robot, Position sensors, Velocity sensors, Acceleration sensors, SONAR and IR sensors, Touch and tactile sensors. Applications of Robots: Applications of robots, selection of robots, economic factors and justification for Robotic application; safety requirements.

TEXT BOOKS

1. Elaine Rich and Kevin Knight, "Artificial Intelligence", Tata McGraw Hill, Second Edition.

2. Craig J J, "Introduction to Robotics, Mechanics and Control", Pearson Education, New Delhi, 2004.

REFERENCE BOOKS

1. Saeed B Niku, "Introduction to Robotics", Pearson Education, New Delhi 2003. 2. George F Luger, "Artificial Intelligence", Pearson Edition Publication, 4th Edition

Study Material

ELECTIVE-I: ARTIFICIAL INTELLIGENCE & ROBOTICS

Sub Code : 18MIT25E

UNIT-I: The AI Problems: AI technique – Criteria for success – Define the Problem as a state space search – Production System – Characteristics – Problem Characteristics.

Prepared by
Dr.P.Radha

What is Artificial Intelligence?

It is a branch of Computer Science that pursues creating the computers or machines as intelligent as human beings.

It is the science and engineering of making intelligent machines, especially intelligent computer programs.

It is related to the similar task of using computers to understand human intelligence, but **AI** does not have to confine itself to methods that are biologically observable

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

According to the father of Artificial Intelligence, John McCarthy, it is *"The science and engineering of making intelligent machines, especially intelligent computer programs".*

Artificial Intelligence is a way of **making a computer, a computer-controlled robot, or a software think intelligently**, in the similar manner the intelligent humans think.

AI is accomplished by studying how human brain thinks and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

It has gained prominence recently due, in part, to big data, or the increase in speed, size and variety of data businesses are now collecting. AI can perform tasks such as identifying patterns in the data more efficiently than humans, enabling businesses to gain more insight out of their data.

From a **business** perspective AI is a set of very powerful tools, and methodologies for using those tools to solve business problems.

From a **programming** perspective, AI includes the study of symbolic programming, problem solving, and search.

AI Vocabulary

Intelligence relates to tasks involving higher mental processes, e.g. creativity, solving problems, pattern recognition, classification, learning, induction, deduction, building analogies, optimization, language processing, knowledge and many more. Intelligence is the computational part of the ability to achieve goals.

Intelligent behaviour is depicted by perceiving one's environment, acting in complex environments, learning and understanding from experience, reasoning to solve problems and discover hidden knowledge, applying knowledge successfully in new situations, thinking abstractly, using analogies, communicating with others and more.

Science based goals of AI pertain to developing concepts, mechanisms and understanding biological intelligent behaviour. The emphasis is on understanding intelligent behaviour.

Engineering based goals of AI relate to developing concepts, theory and practice of building intelligent machines. The emphasis is on system building.

AI Techniques depict how we represent, manipulate and reason with knowledge in order to solve problems. Knowledge is a collection of 'facts'. To manipulate these facts by a program, a suitable representation is required. A good representation facilitates problem solving.

Learning means that programs learn from what facts or behaviour can represent. Learning denotes changes in the systems that are adaptive in other words, it enables the system to do the same task(s) more efficiently next time.

Applications of AI refers to problem solving, search and control strategies, speech recognition, natural language understanding, computer vision, expert systems, etc.

AI Problems:

Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory and computation. Many points on the spectrum of intelligence versus cost are viable, from insects to humans. AI seeks to understand the computations required from intelligent behaviour and to produce computer systems that exhibit intelligence. Aspects of intelligence studied by AI include perception, communicational using human languages, reasoning, planning, learning and memory.

The following questions are to be considered before we can step forward:
1. What are the underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level human intelligence can be modelled?
4. When will it be realized when an intelligent program has been built?

Branches of AI:

A list of branches of AI is given below. However some branches are surely missing, because no one has identified them yet. Some of these may be regarded as concepts or topics rather than full branches.

Logical AI — In general the facts of the specific situation in which it must act, and its goals are all represented by sentences of some mathematical logical language. The program decides what to do by inferring that certain actions are appropriate for achieving its goals.

Search — Artificial Intelligence programs often examine large numbers of possibilities – for example, moves in a chess game and inferences by a theorem proving program. Discoveries are frequently made about how to do this more efficiently in various domains.

Pattern Recognition — When a program makes observations of some kind, it is often planned to compare what it sees with a pattern. For example, a vision program may try to match a pattern of eyes and a nose in a scene in order to find a face. More complex patterns are like a natural language text, a chess position or in the history of some event. These more complex patterns require quite different methods than do the simple patterns that have been studied the most.

Representation — Usually languages of mathematical logic are used to represent the facts about the world.

Inference — Others can be inferred from some facts. Mathematical logical deduction is sufficient for some purposes, but new methods of *non-monotonic* inference have been added to the logic since the 1970s. The simplest kind of non-monotonic reasoning is default reasoning in which a conclusion is to be inferred by default. But the conclusion can be withdrawn if there is evidence to the divergent. For example, when we hear of a bird, we infer that it can fly, but this conclusion can be reversed when we hear that it is a penguin. It is the possibility that a conclusion may have to be withdrawn that constitutes the non-monotonic character of the reasoning. Normal logical reasoning is monotonic, in that the set of conclusions can be drawn from a set of premises, i.e. monotonic increasing function of the premises. Circumscription is another form of non-monotonic reasoning.

Common sense knowledge and Reasoning — This is the area in which AI is farthest from the human level, in spite of the fact that it has been an active research area since the 1950s. While there has been considerable progress in developing systems of *non-monotonic reasoning* and theories of action, yet more new ideas are needed.

Learning from experience — There are some rules expressed in logic for learning. Programs can only learn what facts or behaviour their formalisms can represent, and unfortunately learning systems are almost all based on very limited abilities to represent information.

Planning — Planning starts with general facts about the world (especially facts about the effects of actions), facts about the particular situation and a statement of a goal. From these, planning programs generate a strategy for achieving the goal. In the most common cases, the strategy is just a sequence of actions.

Epistemology — This is a study of the kinds of knowledge that are required for solving problems in the world.

Ontology — Ontology is the study of the kinds of things that exist. In AI the programs and sentences deal with various kinds of objects and we study what these kinds are and what their basic properties are. Ontology assumed importance from the 1990s.

Heuristics — A heuristic is a way of trying to discover something or an idea embedded in a program. The term is used variously in AI. *Heuristic functions* are used in some approaches to search or to measure how far a node in a search tree seems to be from a goal. *Heuristic predicates* that compare two nodes in a search tree to see if one is better than the other, i.e. constitutes an advance toward the goal, and may be more useful.

Genetic programming — Genetic programming is an automated method for creating a working computer program from a high-level problem statement of a problem. Genetic programming starts from a high-level statement of 'what needs to be done' and automatically creates a computer program to solve the problem.

Applications of AI

AI has applications in all fields of human study, such as finance and economics, environmental engineering, chemistry, computer science, and so on. Some of the applications of AI are listed below:

- Perception
    - Machine vision
    - Speech understanding
    - Touch (*tactile* or *haptic*) sensation
- Robotics
- Natural Language Processing
    - Natural Language Understanding
    - Speech Understanding
    - Language Generation
    - Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing

AI Technique:

Artificial Intelligence research during the last three decades has concluded that *Intelligence requires knowledge*. To compensate overwhelming quality, knowledge possesses less desirable properties.

A. It is huge.
B. It is difficult to characterize correctly.
C. It is constantly varying.
D. It differs from data by being organized in a way that corresponds to its application.
E. It is complicated.

An AI technique is a method that exploits knowledge that is represented so that:

- The knowledge captures generalizations that share properties, are grouped together, rather than being allowed separate representation.
- It can be understood by people who must provide it—even though for many programs bulk of the data comes automatically from readings.
- In many AI domains, how the people understand the same people must supply the knowledge to a program.
- It can be easily modified to correct errors and reflect changes in real conditions.
- It can be widely used even if it is incomplete or inaccurate.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must be usually considered.

In order to characterize an AI technique let us consider initially OXO or tic-tac-toe and use a series of different approaches to play the game.

The programs increase in complexity, their use of generalizations, the clarity of their knowledge and the extensibility of their approach. In this way they move towards being representations of AI techniques.

Example-1: Tic-Tac-Toe

1.1 The first approach (simple)

The Tic-Tac-Toe game consists of a nine element vector called BOARD; it represents the numbers 1 to 9 in three rows.

1	2	3
4	5	6
7	8	9

An element contains the value 0 for blank, 1 for X and 2 for O. A MOVETABLE vector consists of 19,683 elements (3^9) and is needed where each element is a nine element vector. The contents of the vector are especially chosen to help the algorithm.

The algorithm makes moves by pursuing the following:

1. View the vector as a ternary number. Convert it to a decimal number.
2. Use the decimal number as an index in MOVETABLE and access the vector.
3. Set BOARD to this vector indicating how the board looks after the move. This approach is capable in time but it has several disadvantages. It takes more space and requires stunning

effort to calculate the decimal numbers. This method is specific to this game and cannot be completed.

1.2 The second approach

The structure of the data is as before but we use 2 for a blank, 3 for an X and 5 for an O. A variable called TURN indicates 1 for the first move and 9 for the last. The algorithm consists of three actions:

MAKE2 which returns 5 if the centre square is blank; otherwise it returns any blank non-corner square, i.e. 2, 4, 6 or 8. POSSWIN (p) returns 0 if player p cannot win on the next move and otherwise returns the number of the square that gives a winning move.

It checks each line using products 3*3*2 = 18 gives a win for X, 5*5*2=50 gives a win for O, and the winning move is the holder of the blank. GO (n) makes a move to square n setting BOARD[n] to 3 or 5.

This algorithm is more involved and takes longer but it is more efficient in storage which compensates for its longer time. It depends on the programmer's skill.

1.3 The final approach

The structure of the data consists of BOARD which contains a nine element vector, a list of board positions that could result from the next move and a number representing an estimation of how the board position leads to an ultimate win for the player to move.

This algorithm looks ahead to make a decision on the next move by deciding which the most promising move or the most suitable move at any stage would be and selects the same.

Consider all possible moves and replies that the program can make. Continue this process for as long as time permits until a winner emerges, and then choose the move that leads to the computer program winning, if possible in the shortest time.

Actually this is most difficult to program by a good limit but it is as far that the technique can be extended to in any game. This method makes relatively fewer loads on the programmer in terms of the game technique but the overall game strategy must be known to the adviser.

Example-2: Question Answering

Let us consider Question Answering systems that accept input in English and provide answers also in English. This problem is harder than the previous one as it is more difficult to specify the problem properly. Another area of difficulty concerns deciding whether the answer obtained is correct, or not, and further what is meant by 'correct'. For example, consider the following situation:

2.1 Text

Rani went shopping for a new Coat. She found a red one she really liked.
When she got home, she found that it went perfectly with her favourite dress.

2.2 Question

1. What did Rani go shopping for?

2. What did Rani find that she liked?
3. Did Rani buy anything?

Method 1

2.3 Data Structures

A set of templates that match common questions and produce patterns used to match against inputs. Templates and patterns are used so that a template that matches a given question is associated with the corresponding pattern to find the answer in the input text. For example, the template who did **x y** generates **x y z** if a match occurs and **z** is the answer to the question. The given text and the question are both stored as strings.

2.4 Algorithm

Answering a question requires the following four steps to be followed:

- Compare the template against the questions and store all successful matches to produce a set of text patterns.
- Pass these text patterns through a substitution process to change the person or voice and produce an expanded set of text patterns.
- Apply each of these patterns to the text; collect all the answers and then print the answers.

2.5 Example

In **question 1** we use the template WHAT DID X Y which generates Rani go shopping for **z** and after substitution we get Rani goes shopping for **z** and Rani went shopping for **z** giving **z** [equivalence] a new coat

In **question 2** we need a very large number of templates and also a scheme to allow the insertion of ‘find’ before ‘that she liked’; the insertion of ‘really’ in the text; and the substitution of ‘she’ for ‘Rani’ gives the answer ‘a red one’.

Question 3 cannot be answered.

2.6 Comments

This is a very primitive approach basically not matching the criteria we set for intelligence and worse than that, used in the game. Surprisingly this type of technique was actually used in ELIZA which will be considered later in the course.

Method 2

2.7 Data Structures

A structure called English consists of a dictionary, grammar and some semantics about the vocabulary we are likely to come across. This data structure provides the knowledge to convert English text into a storable internal form and also to convert the response back into English. The structured representation of the text is a processed form and defines the context of the input text by making explicit all references such as pronouns. There are three types of such *knowledge representation* systems: production rules of the form 'if x then y', slot and filler systems and statements in mathematical logic. The system used here will be the slot and filler system.

Take, for example sentence:

'She found a red one she really liked'.

Event2		**Event2**	
instance:	finding	instance:	liking
tense:	past	tense:	past
agent:	Rani	modifier:	much
object:	Thing1	object:	Thing1

Thing1	
instance:	coat
colour:	Red

The question is stored in two forms: as input and in the above form.

2.8 Algorithm

- Convert the question to a structured form using English know how, then use a marker to indicate the substring (like 'who' or 'what') of the structure, that should be returned as an answer. If a slot and filler system is used a special marker can be placed in more than one slot.
- The answer appears by matching this structured form against the structured text.
- The structured form is matched against the text and the requested segments of the question are returned.

2.9 Examples

Both questions 1 and 2 generate answers via a new coat and a red coat respectively. Question 3 cannot be answered, because there is no direct response.

2.10 Comments

This approach is more meaningful than the previous one and so is more effective. The extra power given must be paid for by additional search time in the knowledge bases. A warning

must be given here: that is – to generate unambiguous English knowledge base is a complex task and must be left until later in the course. The problems of handling pronouns are difficult.

For example:

Rani walked up to the salesperson: she asked where the toy department was.
Rani walked up to the salesperson: she asked her if she needed any help.

Whereas in the original text the linkage of 'she' to 'Rani' is easy, linkage of 'she' in each of the above sentences to Rani and to the salesperson requires additional knowledge about the context via the people in a shop.

Method 3

2.11 Data Structures

World model contains knowledge about objects, actions and situations that are described in the input text. This structure is used to create integrated text from input text. The diagram shows how the system's knowledge of shopping might be represented and stored. This information is known as a script and in this case is a shopping script. (**See figure 1.1 next page)**

1.8.2.12 Algorithm

Convert the question to a structured form using both the knowledge contained in Method 2 and the World model, generating even more possible structures, since even more knowledge is being used. Sometimes filters are introduced to prune the possible answers.

To answer a question, the scheme followed is: Convert the question to a structured form as before but use the world model to resolve any ambiguities that may occur. The structured form is matched against the text and the requested segments of the question are returned.

2.13 Example

Both questions 1 and 2 generate answers, as in the previous program. Question 3 can now be answered. The shopping script is instantiated and from the last sentence the path through step 14 is the one used to form the representation. 'M' is bound to the red coat-got home. '**Rani buys a red coat'** comes from step 10 and the integrated text generates that she bought a red coat.

2.14 Comments

This program is more powerful than both the previous programs because it has more knowledge. Thus, like the last game program it is exploiting AI techniques. However, we are not yet in a position to handle any English question. The major omission is that of a general reasoning mechanism known as inference to be used when the required answer is not explicitly given in the input text. But this approach can handle, with some modifications, questions of the following form with the answer—Saturday morning Rani went shopping. Her brother tried to call her but she did not answer.

Question: Why couldn't Rani's brother reach her?

Answer: Because she was not in.

This answer is derived because we have supplied an additional fact that a person cannot be in two places at once. This patch is not sufficiently general so as to work in all cases and does not provide the type of solution we are really looking for.

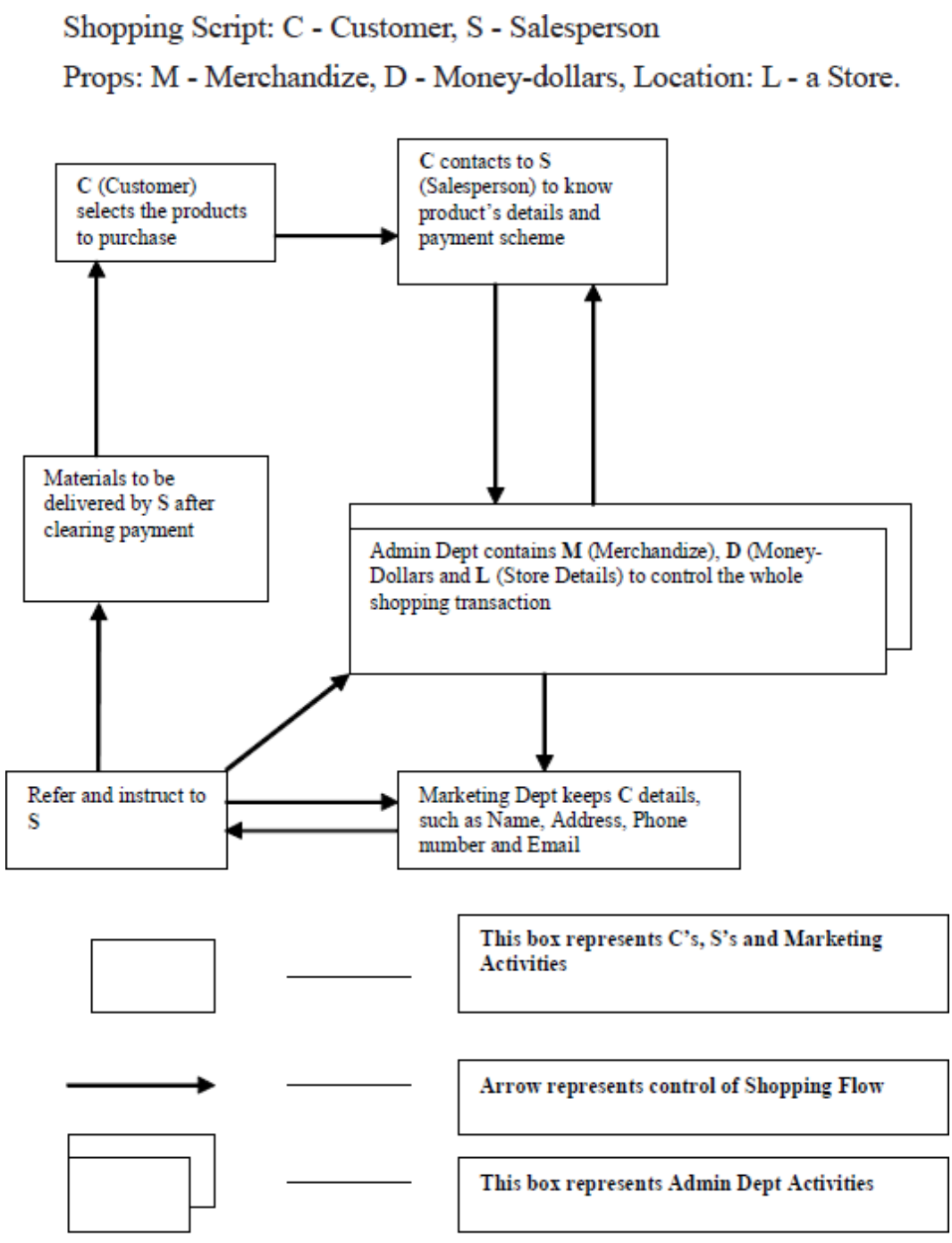


Fig. 1.1 Diagrammatic Representation of Shopping Script

LEVEL OF THE AI MODEL

'What is our goal in trying to produce programs that do the intelligent things that people do?'

Are we trying to produce programs that do the tasks the same way that people do?

OR

Are we trying to produce programs that simply do the tasks the easiest way that is possible?

Programs in the first class attempt to solve problems that a computer can easily solve and do not usually use AI techniques. AI techniques usually include a search, as no direct method is available, the use of knowledge about the objects involved in the problem area and abstraction on which allows an element of pruning to occur, and to enable a solution to be found in real time; otherwise, the data could explode in size. Examples of these trivial problems in the first class, which are now of interest only to psychologists are EPAM (Elementary Perceiver and Memorizer) which memorized garbage syllables.

The second class of problems attempts to solve problems that are non-trivial for a computer and use AI techniques. We wish to model human performance on these:

1. To test psychological theories of human performance. Ex. PARRY [Colby, 1975] – a program to simulate the conversational behavior of a paranoid person.
2. To enable computers to understand human reasoning – for example, programs that answer questions based upon newspaper articles indicating human behavior.
3. To enable people to understand computer reasoning. Some people are reluctant to accept computer results unless they understand the mechanisms involved in arriving at the results.
4. To exploit the knowledge gained by people who are best at gathering information. This persuaded the earlier workers to simulate human behavior in the SB part of AISB simulated behavior. Examples of this type of approach led to GPS (General Problem Solver).

Questions for Practice:

1. What is *intelligence*? How do we measure it? Are these measurements useful?
2. When the temperature falls and the thermostat turns the heater on, does it act because it *believes* the room to be too cold? Does it *feel* cold? What sorts of things can have beliefs or feelings? Is this related to the idea of consciousness?
3. Some people believe that the relationship between your mind (a non-physical thing) and your brain (the physical thing inside your skull) is exactly like the relationship between a computational process (a non-physical thing) and a physical computer. Do you agree?
4. How good are machines at playing chess? If a machine can consistently beat all the best human chess players, does this prove that the machine is *intelligent*?
5. What is AI Technique? Explain Tic-Tac-Toe Problem using AI Technique.

PROBLEMS, PROBLEM SPACES AND SEARCH

To solve the problem of building a system you should take the following steps:

1. Define the problem accurately including detailed specifications and what constitutes a suitable solution.
2. Scrutinize the problem carefully, for some features may have a central affect on the chosen method of solution.
3. Segregate and represent the background knowledge needed in the solution of the problem.
4. Choose the best solving techniques for the problem to solve a solution.

***Problem solving* is a process** of generating solutions from observed data.

- a *'problem'* is characterized by a set of *goals*,
- a set of *objects*, and
- a set of *operations*.

These could be ill-defined and may evolve during problem solving.

- A **'*problem space*'** is an abstract space.
  - ✓ A problem space encompasses all *valid states* that can be generated by the application of any combination of *operators* on any combination of *objects*.
  - ✓ The problem space may contain one or more *solutions*. A solution is a combination of *operations* and *objects* that achieve the *goals*.
- A '***search***' refers to the search for a solution in a problem space.
  - ✓ Search proceeds with different types of '*search control strategies*'.
  - ✓ The *depth-first search and breadth-first search* are the two common *search strategies.*

2.1 AI - General Problem Solving

Problem solving has been the key area of concern for Artificial Intelligence.

Problem solving is a process of generating solutions from observed or given data. It is however not always possible to use direct methods (i.e. go directly from data to solution). Instead, problem solving often needs to use indirect or modelbased methods.

General Problem Solver (GPS) was a computer program created in 1957 by Simon and Newell to build a universal problem solver machine. *GPS* was based on Simon and Newell's theoretical work on logic machines. *GPS* in principle can solve any formalized symbolic problem, such as theorems proof and geometric problems and chess playing. *GPS* solved many simple problems, such as the Towers of Hanoi, that could be sufficiently formalized, but ***GPS could not solve any real-world problems***.

To build a system to solve a particular problem, we need to:

- Define the problem precisely – find input situations as well as final situations for an acceptable solution to the problem

- Analyze the problem – find few important features that may have impact on the appropriateness of various possible techniques for solving the problem
- Isolate and represent task knowledge necessary to solve the problem
- Choose the best problem-solving technique(s) and apply to the particular problem

Problem definitions
A problem is defined by its '*elements*' and their '*relations*'. To provide a formal description of a problem, we need to do the following:

a. Define a *state space* that contains all the possible configurations of the relevant objects, including some impossible ones.
b. Specify one or more states that describe possible situations, from which the problem-solving process may start. These states are called *initial states*.
c. Specify one or more states that would be acceptable solution to the problem.

These states are called *goal states*.

Specify a set of *rules* that describe the actions (*operators*) available.

The problem can then be solved by using the *rules*, in combination with an appropriate *control strategy*, to move through the *problem space* until a *path* from an *initial state* to a *goal state* is found. This process is known as ***'search'***. Thus:

- *Search* is fundamental to the problem-solving process.
- *Search* is a general mechanism that can be used when a more direct method is not known.
- *Search* provides the framework into which more direct methods for solving subparts of a problem can be embedded. A very large number of AI problems are formulated as search problems.
- Problem space

A *problem space* is represented by a directed graph, where *nodes* represent search state and *paths* represent the operators applied to change the *state*.

To simplify search algorithms, it is often convenient to logically and programmatically represent a problem space as a **tree**. A *tree* usually decreases the complexity of a search at a cost. Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph, e.g. node ***B*** and node ***D.***

A *tree is a graph* in which any two vertices are connected by exactly one path. Alternatively, any connected *graph with no cycles is a tree*.

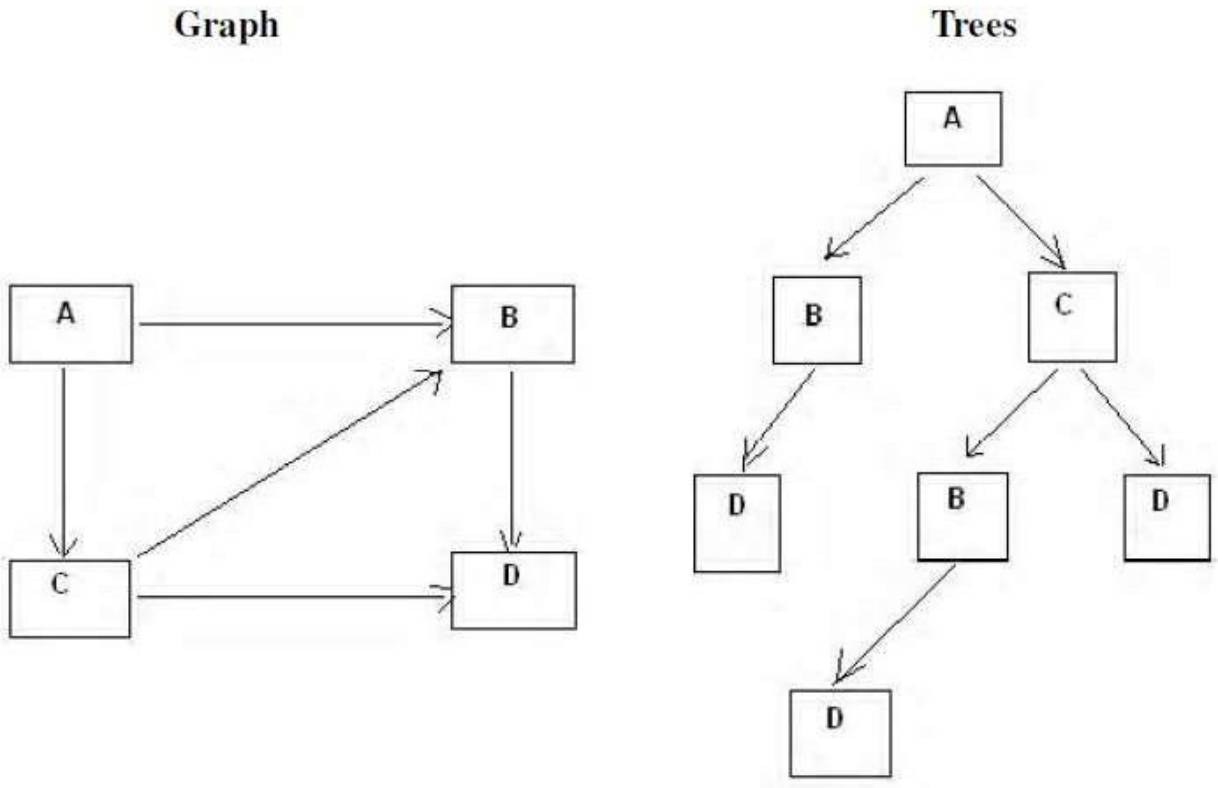


Fig. 2.1 Graph and Tree

• **Problem solving:** The term, Problem Solving relates to analysis in AI. Problem solving may be characterized as a systematic search through a range of possible actions to reach some predefined goal or solution. Problem-solving methods are categorized as *special purpose* and *general purpose*.

• A *special-purpose method* is tailor-made for a particular problem, often exploits very specific features of the situation in which the problem is embedded.

• A *general-purpose method* is applicable to a wide variety of problems. One General-purpose technique used in AI is *'means-end analysis':* a step-bystep, or incremental, reduction of the difference between current state and final goal.

2.3 DEFINING PROBLEM AS A STATE SPACE SEARCH

To solve the problem of playing a game, we require the rules of the game and targets for winning as well as representing positions in the game. The opening position can be defined as the initial state and a winning position as a goal state. Moves from initial state to other states leading to the goal state follow legally. However, the rules are far too abundant in most games— especially in chess, where they exceed the number of particles in the universe. Thus, the rules cannot be supplied accurately and computer programs cannot handle easily. The storage also presents another problem but searching can be achieved by hashing.

The number of rules that are used must be minimized and the set can be created by expressing each rule in a form as possible. The representation of games leads to a state space representation and it is common for well-organized games with some structure. This representation allows for the formal definition of a problem that needs the movement from a set of initial positions to one of a set of target positions. It means that the solution involves using known techniques and a systematic search. This is quite a common method in Artificial Intelligence.

2.3.1 State Space Search

A *state space* represents a problem in terms of *states* and *operators* that change states.
A state space consists of:

- A representation of the *states* the system can be in. For example, in a board game, the board represents the current state of the game.
- A set of *operators* that can change one state into another state. In a board game, the operators are the legal moves from any given state. Often the operators are represented as programs that change a state representation to represent the new state.
- An *initial state.*
- A set of *final states*; some of these may be desirable, others undesirable. This set is often represented implicitly by a program that detects terminal states.

2.3.2 The Water Jug Problem

In this problem, we use two jugs called **four** and **three;** four holds a maximum of four gallons of water and **three** a maximum of three gallons of water. How can we get two gallons of water in the **four** jug?

The state space is a set of prearranged pairs giving the number of gallons of water in the pair of jugs at any time, i.e., (**four, three**) where **fou**r = 0, 1, 2, 3 or 4 and **three** = 0, 1, 2 or 3.

The start state is (0, 0) and the goal state is (2, n) where n may be any but it is limited to **three** holding from 0 to 3 gallons of water or empty. Three and four shows the name and numerical number shows the amount of water in jugs for solving the water jug problem. The major production rules for solving this problem are shown below:

Initial condition	**Goal comment**
1. (four, three) if four < 4	(4, three) fill four from tap
2. (four, three) if three< 3	(four, 3) fill three from tap
3. (four, three) If four > 0	(0, three) empty four into drain
4. (four, three) if three > 0	(four, 0) empty three into drain
5. (four, three) if four + three<4	(four + three, 0) empty three into four
6. (four, three) if four + three<3	(0, four + three) empty four into three
7. (0, three) If three > 0	(three, 0) empty three into four
8. (four, 0) if four > 0	(0, four) empty four into three
9. (0, 2)	(2, 0) empty three into four
10. (2, 0)	(0, 2) empty four into three
11. (four, three) if four < 4	(4, three-diff) pour diff, 4-four, into four from three
12. (three, four) if three < 3	(four-diff, 3) pour diff, 3-three, into three from four and a solution is given below four three rule

***(Fig. 2.2** Production Rules for the Water Jug Problem)*

Gallons in Four Jug	**Gallons in Three Jug**	**Rules Applied**
0	0	-
0	3	2
3	0	7
3	3	2
4	2	11
0	2	3
2	0	10

***(Fig. 2.3** One Solution to the Water Jug Problem)*

The problem solved by using the production rules in combination with an appropriate control strategy, moving through the problem space until a path from an initial state to a goal state is found. In this problem solving process, search is the fundamental concept. For simple problems it is easier to achieve this goal by hand but there will be cases where this is far too difficult.

2.4 PRODUCTION SYSTEMS

Production systems provide appropriate structures for performing and describing search processes. A production system has four basic components as enumerated below.

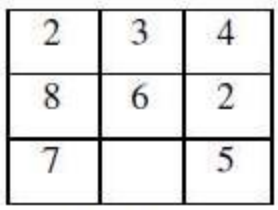
- A set of rules each consisting of a left side that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.
- A database of current facts established during the process of inference.

- A control strategy that specifies the order in which the rules will be compared with facts in the database and also specifies how to resolve conflicts in selection of several rules or selection of more facts.
- A rule firing module.

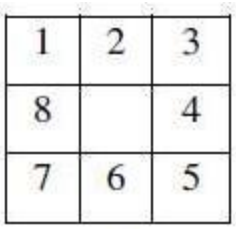
The production rules operate on the knowledge database. Each rule has a precondition—that is, either satisfied or not by the knowledge database. If the precondition is satisfied, the rule can be applied. Application of the rule changes the knowledge database. The control system chooses which applicable rule should be applied and ceases computation when a termination condition on the knowledge database is satisfied.

Example: Eight puzzle (8-Puzzle)

The 8-puzzle is a 3 × 3 array containing eight square pieces, numbered 1 through 8, and one empty space. A piece can be moved horizontally or vertically into the empty space, in effect exchanging the positions of the piece and the empty space. There are four possible moves, UP (move the blank space up), DOWN, LEFT and RIGHT. The aim of the game is to make a sequence of moves that will convert the board from the start state into the goal state:



Initial State



Goal State

This example can be solved by the operator sequence UP, RIGHT, UP, LEFT, DOWN.

Example: Missionaries and Cannibals

The Missionaries and Cannibals problem illustrates the use of state space search for planning under constraints:

Three missionaries and three cannibals wish to cross a river using a two person boat. If at any time the cannibals outnumber the missionaries on either side of the river, they will eat the missionaries. How can a sequence of boat trips be performed that will get everyone to the other side of the river without any missionaries being eaten?

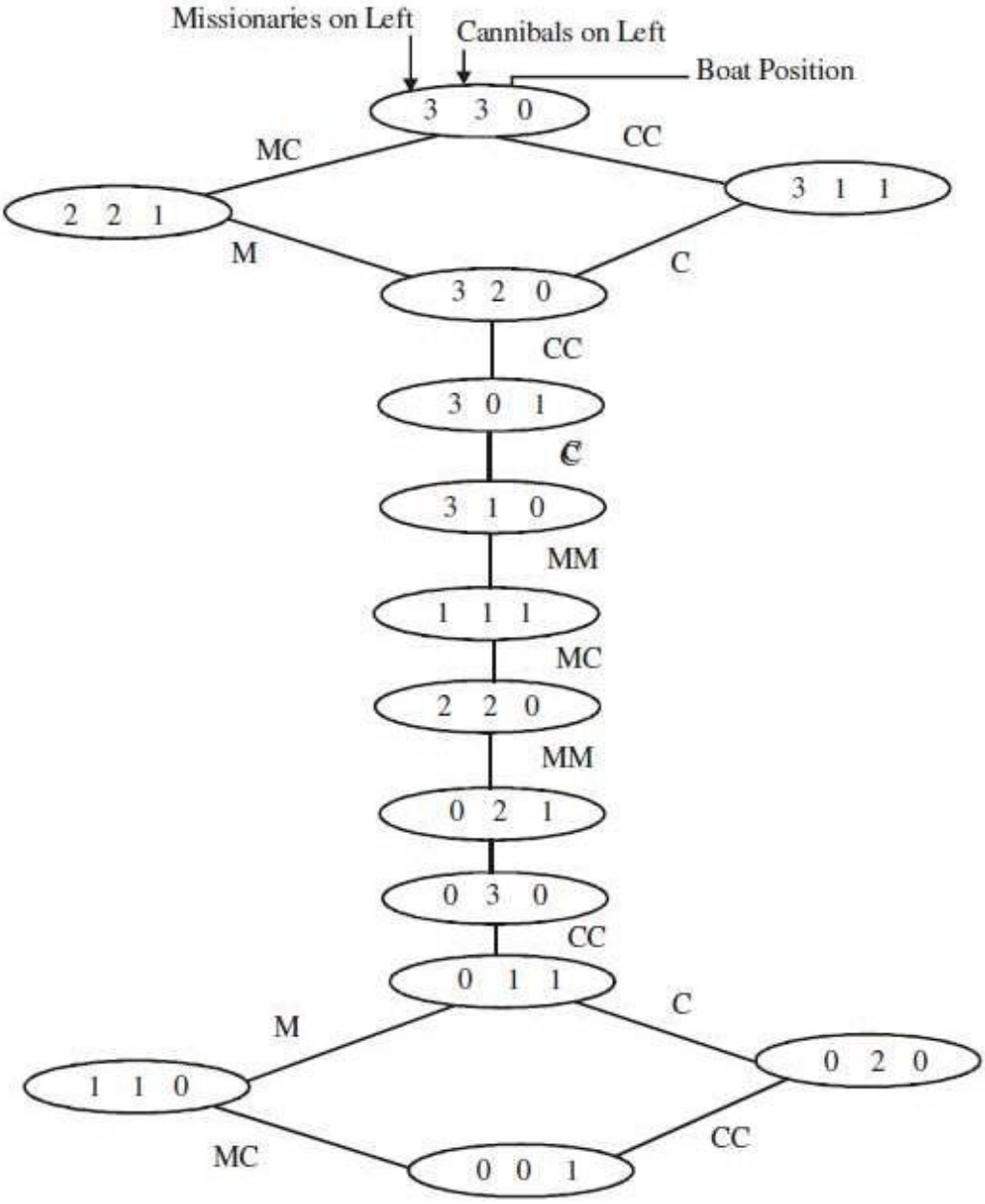
State representation:

1. BOAT position: original (T) or final (NIL) side of the river.
2. Number of Missionaries and Cannibals on the original side of the river.
3. Start is (T 3 3); Goal is (NIL 0 0).

Operators:

(MM 2 0)	Two Missionaries cross the river.
(MC 1 1)	One Missionary and one Cannibal.
(CC 0 2)	Two Cannibals.
(M 1 0)	One Missionary.
(C 0 1)	One Cannibal.

Missionaries/Cannibals Search Graph



2.4.1 Control Strategies

The word '***search***' refers to the search for a solution in a *problem space*.

- Search proceeds with different types of *'**search control strategies**'*.
- A strategy is defined by picking the order in which the nodes expand.

The Search strategies are evaluated along the following dimensions: Completeness, Time complexity, Space complexity, Optimality (the search- related terms are first explained, and then the search algorithms and control strategies are illustrated next).

Search-related terms

• **Algorithm's performance and complexity**

Ideally we want a common measure so that we can compare approaches in order to select the most appropriate algorithm for a given situation.

- ✓ *Performance* of an algorithm depends on internal and external factors.

  Internal factors/ External factors
  - ***Time*** required to run
  - ***Size*** of input to the algorithm
  - ***Space*** (memory) required to run
  - ***Speed*** of the computer
  - ***Quality*** of the compiler
- ✓ *Complexity* is a measure of the performance of an algorithm. *Complexity* measures the internal factors, usually in time than space.

• **Computational complexity**

It is the measure of resources in terms of ***Time*** and ***Space***.

- ✓ If ***A*** is an algorithm that solves a decision problem ***f***, then run-time of ***A*** is the number of steps taken on the input of length ***n.***
- ✓ *Time Complexity* ***T(n)*** of a decision problem ***f*** is the run-time of the 'best' algorithm ***A*** for ***f.***
- ✓ *Space Complexity* ***S(n)*** of a decision problem ***f*** is the amount of memory used by the 'best' algorithm ***A*** for ***f.***

• '**Big - O**' **notation**

The ***Big-O***, theoretical *measure of the execution of an* algorithm, usually indicates the ***time*** or the ***memory*** needed, given the problem size ***n***, which is usually the number of items.

• ***Big-O*** notation

The ***Big-O*** notation is used to give an approximation to the *run-time- efficiency of an algorithm;* the letter '**O**' is for order of magnitude of operations or space at run-time.

• The ***Big-O*** of an Algorithm ***A***

- ✓ If an algorithm ***A*** requires time proportional to ***f(n)***, then algorithm ***A*** is said to be of order ***f(n)***, and it is denoted as ***O(f(n))***.
- ✓ If algorithm ***A*** requires time proportional to ***n2***, then the order of the algorithm is said to be ***O(n2).***
- ✓ If algorithm ***A*** requires time proportional to ***n***, then the order of the algorithm is said to be ***O(n).***

The function ***f(n)*** is called the algorithm's *growth-rate function*. In other words, if an algorithm has performance complexity ***O(n)***, this means that the run-time ***t*** should be directly proportional to ***n***, ie ***t • n or t = k n*** where ***k*** is constant of proportionality.
Similarly, for algorithms having performance complexity ***O(log2(n)), O(log N), O(N log N), O(2N)*** and so on.

Example 1:
Determine the ***Big-O*** of an algorithm:

Calculate the sum of the ***n*** elements in an integer array ***a[0..n-1]***.

Line no.	**Instructions**	**No of execution steps**
line 1	Sum	1
line 2	for (i = 0; i < n; i++)	n + 1
line 3	sum += a[i]	n
line 4	print sum	1
	Total	**2n + 3**

Thus, the polynomial ***(2n + 3)*** is dominated by the 1st term as ***n*** while the number of elements in the array becomes very large.

• In determining the ***Big-O***, ignore constants such as ***2*** and ***3***. So the algorithm is of order ***n***.
• So the ***Big-O*** of the algorithm is ***O(n)***.
• In other words the run-time of this algorithm increases roughly as the size of the input data ***n***, e.g., an array of size ***n***.

Tree structure
Tree is a way of organizing objects, related in a hierarchical fashion.
- Tree is a type of data structure in which each *element* is attached to one or more elements directly beneath it.
- The connections between elements are called *branches*.
- Tree is often called *inverted trees* because it is drawn with the *root* at the top.
- The elements that have no elements below them are called *leaves*.
- A *binary tree* is a special type: each element has only two branches below it.

Properties
- Tree is a special case of a *graph*.
- The topmost node in a tree is called the *root node*.
- At root node all operations on the tree begin.
- A node has at most one parent.
- The topmost node (root node) has no parents.
- Each node has zero or more *child nodes*, which are below it .
- The nodes at the bottommost level of the tree are called *leaf nodes*.
  Since *leaf nodes* are at the bottom most level, they do not have children.
- A node that has a child is called the child's *parent node*.
- The *depth of a node* ***n*** is the length of the path from the root to the node.
- The root node is at depth zero.

• **Stacks and Queues**

The *Stacks* and *Queues* are data structures that maintain the order of *last-in, first-out* and *first-in, first-out* respectively. Both *stacks* and *queues* are often implemented as linked lists, but that is not the only possible implementation.

Stack - Last In First Out (LIFO) lists

- An ordered list; a sequence of items, piled one on top of the other.
- The ***insertions*** and ***deletions*** are made at one end only, called ***Top***.
- If Stack ***S = (a[1], a[2],..... a[n])*** then ***a[1]*** is bottom most element
- Any intermediate element ***(a[i])*** is on top of element ***a[i-1], 1 < i <= n.***
- In Stack all operation take place on ***Top***.

The ***Pop*** operation removes item from top of the stack.
The ***Push*** operation adds an item on top of the stack.

Queue - First In First Out (FIFO) lists

• An ordered list; a sequence of items; there are restrictions about how items can be added to and removed from the list. A queue has two ends.
• All ***insertions*** (enqueue) take place at one end, called ***Rear*** or ***Back***
• All ***deletions*** (dequeue) take place at other end, called ***Front***.
• If Queue has ***a[n]*** as rear element then ***a[i+1]*** is behind ***a[i] , 1 < i <= n.***
• All operation takes place at one end of queue or the other.

The ***Dequeue*** operation removes the item at ***Front*** of the queue.
The ***Enqueue*** operation adds an item to the ***Rear*** of the queue.

Search

Search is the systematic examination of *states* to find path from the *start / root state* to the *goal state*.

• Search usually results from a lack of knowledge.
• Search explores knowledge alternatives to arrive at the best answer.
• Search algorithm output is a solution, that is, a path from the initial state to a state that satisfies the goal test.

For general-purpose problem-solving – '***Search' is an approach***.

• Search deals with finding *nodes* having certain properties in a *graph* that represents search space.
• Search methods explore the search space 'intelligently', evaluating possibilities without investigating every single possibility.

Examples:

• For a Robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT—until the goal is reached.
• Puzzles and Games have explicit rules: e.g., the '***Tower of Hanoi***' puzzle

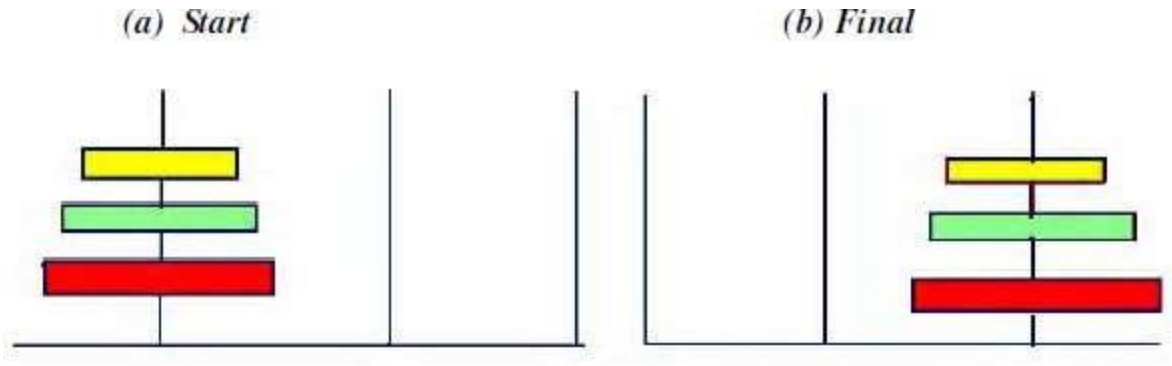


Fig. 2.4 Tower of Hanoi Puzzle

This puzzle involves a set of rings of different sizes that can be placed on three different pegs.
• The puzzle starts with the rings arranged as shown in Figure 2.4(a)
• The goal of this puzzle is to move them all as to Figure 2.4(b)
• Condition: Only the top ring on a peg can be moved, and it may only be placed on a smaller ring, or on an empty peg.

In this *Tower of Hanoi* puzzle:
• Situations encountered while solving the problem are described as *states*.
• Set of all possible configurations of rings on the pegs is called *'problem space*'.
• **States**
A *State* is a representation of elements in a given moment.
A problem is defined by its *elements* and their *relations*.
At each instant of a problem, the elements have specific descriptors and relations; the *descriptors* indicate how to select elements?
Among all possible states, there are two special states called:

- ✓ *Initial state* – the start point
- ✓ *Final state* – the goal state

• **State Change:** Successor Function
A '*successor function*' is needed for state change. The Successor Function moves one state to another state.
Successor Function:

- ✓ It is a description of possible actions; a set of operators.
- ✓ It is a transformation function on a state representation, which converts that state into another state.
- ✓ It defines a relation of accessibility among states.
- ✓ It represents the conditions of applicability of a state and corresponding transformation function.

• **State space**
A *state space* is the set of all *states* reachable from the *initial state*.

- ✓ A *state space* forms a *graph* (or map) in which the *nodes* are states and the *arcs* between nodes are actions.
- ✓ In a *state space*, a *path* is a sequence of states connected by a sequence of actions.
- ✓ The *solution* of a problem is part of the map formed by the *state space*.

• **Structure of a state space**

The *structures* of a *state space* are *trees* and *graphs***.**

- ✓ A *tree* is a hierarchical structure in a graphical form.
- ✓ A *graph* is a non-hierarchical structure.

• A *tree* has only one path to a given node;

i.e., a *tree* has one and only one path from any point to any other point.

• A *graph* consists of a set of nodes (vertices) and a set of edges (arcs). Arcs establish relationships (connections) between the nodes; i.e., a graph has several paths to a given node.

• The *Operators* are directed *arcs* between nodes.

A *search* process explores the *state space*. In the worst case, the search explores all possible *paths* between the *initial state* and the *goal state*.

• **Problem solution**

In the *state space*, a *solution* is a path from the *initial state* to a *goal state* or, sometimes, just a *goal state*.

- ✓ A *solution cost function* assigns a numeric cost to each *path*; it also gives the cost of applying the *operators* to the *states*.
- ✓ A *solution quality* is measured by the *path cost function*; and an optimal solution has the lowest path cost among all solutions.
- ✓ The *solutions* can be *any or optimal or all*.
- ✓ The importance of cost depends on the problem and the type of solution asked

• **Problem description**

A problem consists of the description of:

- ✓ The current state of the world,
- ✓ The actions that can transform one state of the world into another,
- ✓ The desired state of the world.

The following action one taken to describe the problem:

- ✓ *State space* is defined explicitly or implicitly

A *state space* should describe everything that is needed to solve a problem and nothing that is not needed to solve the problem.

- ✓ *Initial state* is start state
- ✓ *Goal state* is the conditions it has to fulfill.

The description by a desired state may be complete or partial.

- ✓ *Operators* are to change state
- ✓ Operators do actions that can transform one state into another;
- ✓ Operators consist of: Preconditions and Instructions;

Preconditions provide partial description of the state of the world that must be true in order to perform the action, and

Instructions tell the user how to create the next state.

- Operators should be as general as possible, so as to reduce their number.
- *Elements of the domain* has relevance to the problem
  - ✓ Knowledge of the starting point.
- *Problem solving* is finding a solution
  - ✓ Find an ordered sequence of operators that transform the current (start) state into a goal state.

- *Restrictions* are solution quality any, optimal, or all
    - ✓ Finding the shortest sequence, or
    - ✓ finding the least expensive sequence defining cost, or
    - ✓ finding any sequence as quickly as possible.

This can also be explained with the help of algebraic function as given below.

PROBLEM CHARACTERISTICS

Heuristics cannot be generalized, as they are domain specific. Production systems provide ideal techniques for representing such heuristics in the form of IF-THEN rules. Most problems requiring simulation of intelligence use heuristic search extensively. Some heuristics are used to define the control structure that guides the search process, as seen in the example described above. But heuristics can also be encoded in the rules to represent the domain knowledge. Since most AI problems make use of knowledge and guided search through the knowledge, AI can be described as *the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about problem domain*.

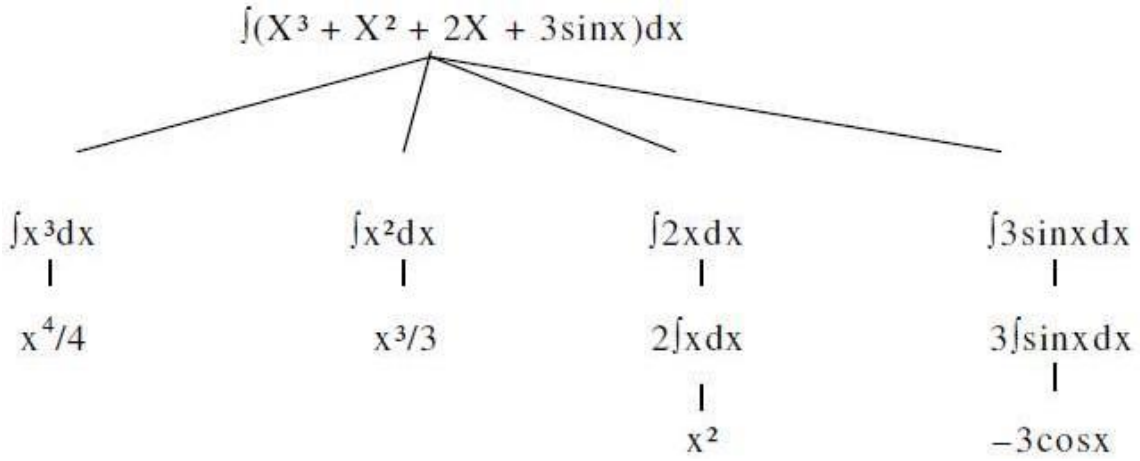
To use the heuristic search for problem solving, we suggest analysis of the problem for the following considerations:

- Decomposability of the problem into a set of independent smaller subproblems
- Possibility of undoing solution steps, if they are found to be unwise
- Predictability of the problem universe
- Possibility of obtaining an obvious solution to a problem without comparison of all other possible solutions
- Type of the solution: whether it is a state or a path to the goal state
- Role of knowledge in problem solving
- Nature of solution process: with or without interacting with the user

The general classes of engineering problems such as planning, classification, diagnosis, monitoring and design are generally knowledge intensive and use a large amount of heuristics. Depending on the type of problem, the knowledge representation schemes and control strategies for search are to be adopted. Combining heuristics with the two basic search strategies have been discussed above. There are a number of other general-purpose search techniques which are essentially heuristics based. Their efficiency primarily depends on how they exploit the domain-specific knowledge to abolish undesirable paths. Such search methods are called ‘weak methods’, since the progress of the search depends heavily on the way the domain knowledge is exploited. A few of such search techniques which form the centre of many AI systems are briefly presented in the following sections.

Problem Decomposition

Suppose to solve the expression is: $+ \int(X^3 + X^2 + 2X + 3\sin x)\, dx$



This problem can be solved by breaking it into smaller problems, each of which we can solve by using a small collection of specific rules. Using this technique of problem decomposition, we can solve very large problems very easily. This can be considered as an intelligent behaviour.

Can Solution Steps be Ignored?

Suppose we are trying to prove a mathematical theorem: first we proceed considering that proving a lemma will be useful. Later we realize that it is not at all useful. We start with another one to prove the theorem. Here we simply ignore the first method.
Consider the 8-puzzle problem to solve: we make a wrong move and realize that mistake. But here, the control strategy must keep track of all the moves, so that we can backtrack to the initial state and start with some new move.
Consider the problem of playing chess. Here, once we make a move we never recover from that step. These problems are illustrated in the three important classes of problems mentioned below:

1. Ignorable, in which solution steps can be ignored. Eg: Theorem Proving
2. Recoverable, in which solution steps can be undone. Eg: 8-Puzzle
3. Irrecoverable, in which solution steps cannot be undone. Eg: Chess

Is the Problem Universe Predictable?

Consider the 8-Puzzle problem. Every time we make a move, we know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident what the resulting state will be. We can backtrack to earlier moves if they prove unwise.

Suppose we want to play Bridge. We need to plan before the first play, but we cannot play with certainty. So, the outcome of this game is very uncertain. In case of 8-Puzzle, the outcome is very certain. To solve uncertain outcome problems, we follow the process of plan revision as the plan is carried out and the necessary feedback is provided. The disadvantage is that the planning in this case is often very expensive.

Is Good Solution Absolute or Relative?
Consider the problem of answering questions based on a database of simple facts such as the following:

1. Siva was a man.
2. Siva was a worker in a company.
3. Siva was born in 1905.
4. All men are mortal.
5. All workers in a factory died when there was an accident in 1952.
6. No mortal lives longer than 100 years.

Suppose we ask a question: 'Is Siva alive?'
By representing these facts in a formal language, such as predicate logic, and then using formal inference methods we can derive an answer to this question easily.
There are two ways to answer the question shown below:

Method I:

1. Siva was a man.
2. Siva was born in 1905.
3. All men are mortal.
4. Now it is 2008, so Siva's age is 103 years.
5. No mortal lives longer than 100 years.

Method II:

1. Siva is a worker in the company.
2. All workers in the company died in 1952.

Answer: So Siva is not alive. It is the answer from the above methods.

We are interested to answer the question; it does not matter which path we follow. If we follow one path successfully to the correct answer, then there is no reason to go back and check another path to lead the solution.

CHARACTERISTICS OF PRODUCTION SYSTEMS

Production systems provide us with good ways of describing the operations that can be performed in a search for a solution to a problem.
At this time, two questions may arise:

1. Can production systems be described by a set of characteristics? And how can they be easily implemented?
2. What relationships are there between the problem types and the types of production systems well suited for solving the problems?

To answer these questions, first consider the following definitions of classes of production systems:

1. A monotonic production system is a production system in which the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected.
2. A non-monotonic production system is one in which this is not true.
3. A partially communicative production system is a production system with the property that if the application of a particular sequence of rules transforms state P into state Q, then any combination of those rules that is allowable also transforms state P into state Q.
4. A commutative production system is a production system that is both monotonic and partially commutative.

Table 2.1 Four Categories of Production Systems

Production System	Monotonic	Non-monotonic
Partially Commutative	Theorem Proving	Robot Navigation
Non-partially Commutative	Chemical Synthesis	Bridge

Is there any relationship between classes of production systems and classes of problems? For any solvable problems, there exist an infinite number of production systems that show how to find solutions. Any problem that can be solved by any production system can be solved by a commutative one, but the commutative one is practically useless. It may use individual states to represent entire sequences of applications of rules of a simpler, non-commutative system. In the formal sense, there is no relationship between kinds of problems and kinds of production systems Since all problems can be solved by all kinds of systems. But in the practical sense, there is definitely such a relationship between the kinds of problems and the kinds of systems that lend themselves to describing those problems.

Partially commutative, monotonic productions systems are useful for solving ignorable problems. These are important from an implementation point of view without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed. Both types of partially commutative production systems are significant from an implementation point; they tend to lead to many duplications of individual states during the search process. Production systems that are not partially commutative are useful for many problems in which permanent changes occur.

Issues in the Design of Search Programs

Each search process can be considered to be a tree traversal. The object of the search is to find a path from the initial state to a goal state using a tree. The number of nodes generated might be huge; and in practice many of the nodes would not be needed. The secret of a good search routine is to generate only those nodes that are likely to be useful, rather than having a precise tree. The rules are used to represent the tree implicitly and only to create nodes explicitly if they are actually to be of use.

The following issues arise when searching:
• The tree can be searched forward from the initial node to the goal state or backwards from the goal state to the initial state.
• To select applicable rules, it is critical to have an efficient procedure for matching rules against states.
• How to represent each node of the search process? This is the knowledge representation problem or the frame problem. In games, an array suffices; in other problems, more complex data structures are needed.

Finally in terms of data structures, considering the water jug as a typical problem do we use a graph or tree? The breadth-first structure does take note of all nodes generated but the depth-first one can be modified.

Check duplicate nodes

1. Observe all nodes that are already generated, if a new node is present.
2. If it exists add it to the graph.
3. If it already exists, then

    a. Set the node that is being expanded to the point to the already existing node corresponding to its successor rather than to the new one. The new one can be thrown away.

    b. If the best or shortest path is being determined, check to see if this path is better or worse than the old one. If worse, do nothing.

Better save the new path and work the change in length through the chain of successor nodes if necessary.

Example: Tic-Tac-Toe

State spaces are good representations for board games such as Tic-Tac-Toe. The position of a game can be explained by the contents of the board and the player whose turn is next. The board can be represented as an array of 9 cells, each of which may contain an X or O or be empty.

- **State:**
    - ✓ Player to move next: X or O.
    - ✓ Board configuration:

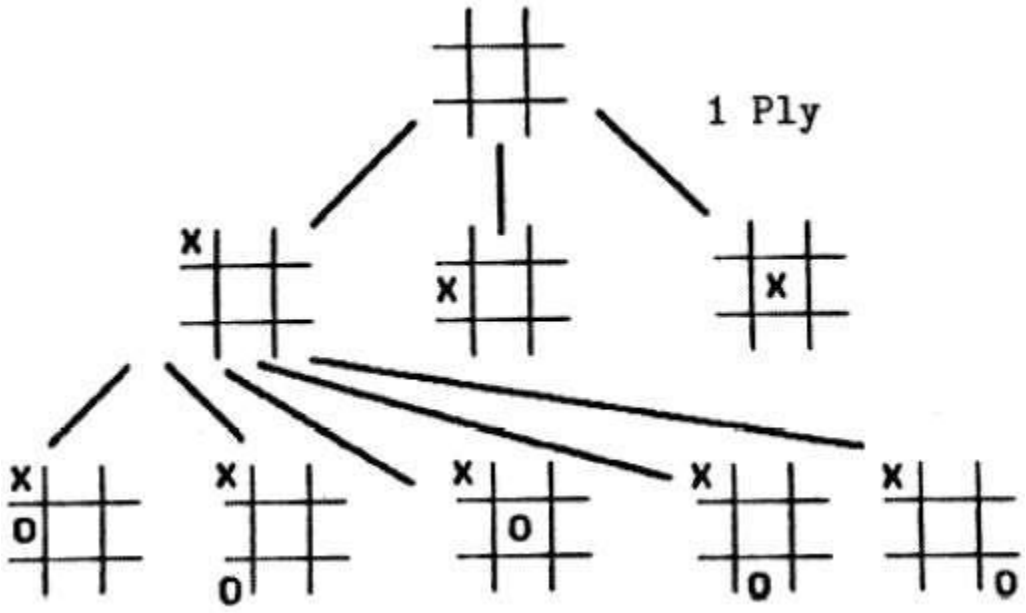
X		0
	0	
X		X

- **Operators:** Change an empty cell to X or O.
- **Start State:** Board empty; X's turn.
- **Terminal States:** Three X's in a row; Three O's in a row; All cells full.

Search Tree

The sequence of states formed by possible moves is called a *search tree*. Each level of the tree is called a *ply*.

Since the same state may be reachable by different sequences of moves, the state space may in general be a graph. It may be treated as a tree for simplicity, at the cost of duplicating states.



Solving problems using search

• Given an informal description of the problem, construct a formal description as a state space:

- ✓ Define a data structure to represent the *state*.
- ✓ Make a representation for the *initial state* from the given data.
- ✓ Write programs to represent *operators* that change a given state representation to a new state representation.
- ✓ Write a program to detect *terminal states*.

• Choose an appropriate search technique:

- ✓ How large is the search space?
- ✓ How well structured is the domain?
- ✓ What knowledge about the domain can be used to guide the search?