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 problemsolving 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*.

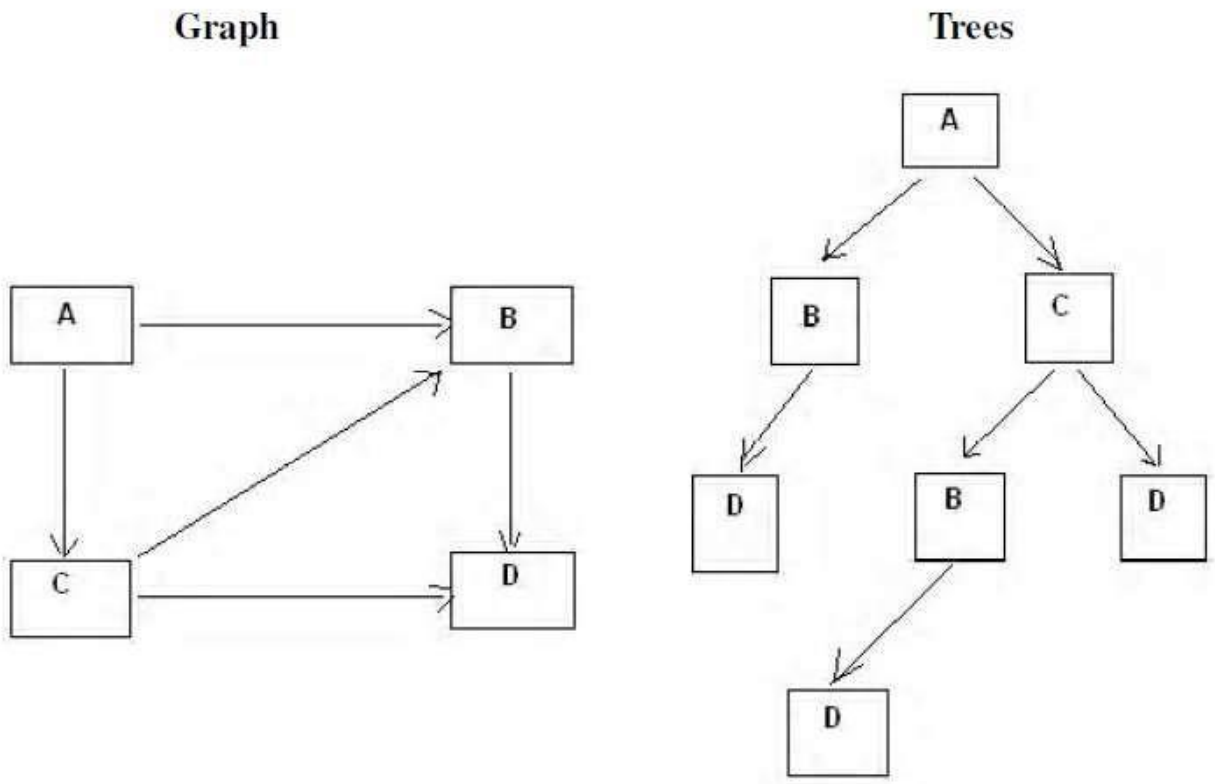


Fig. 2.1 Graph and Tree

- **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.

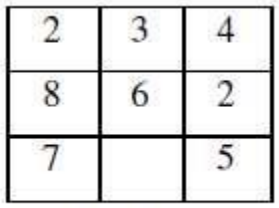
- A set of rules each consisting of a left side that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.
- A database of current facts established during the process of inference.
- A control strategy that specifies the order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

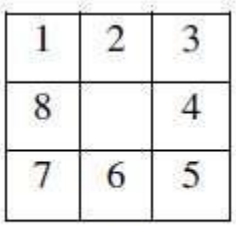
Example: Eight puzzle (8-Puzzle)

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?

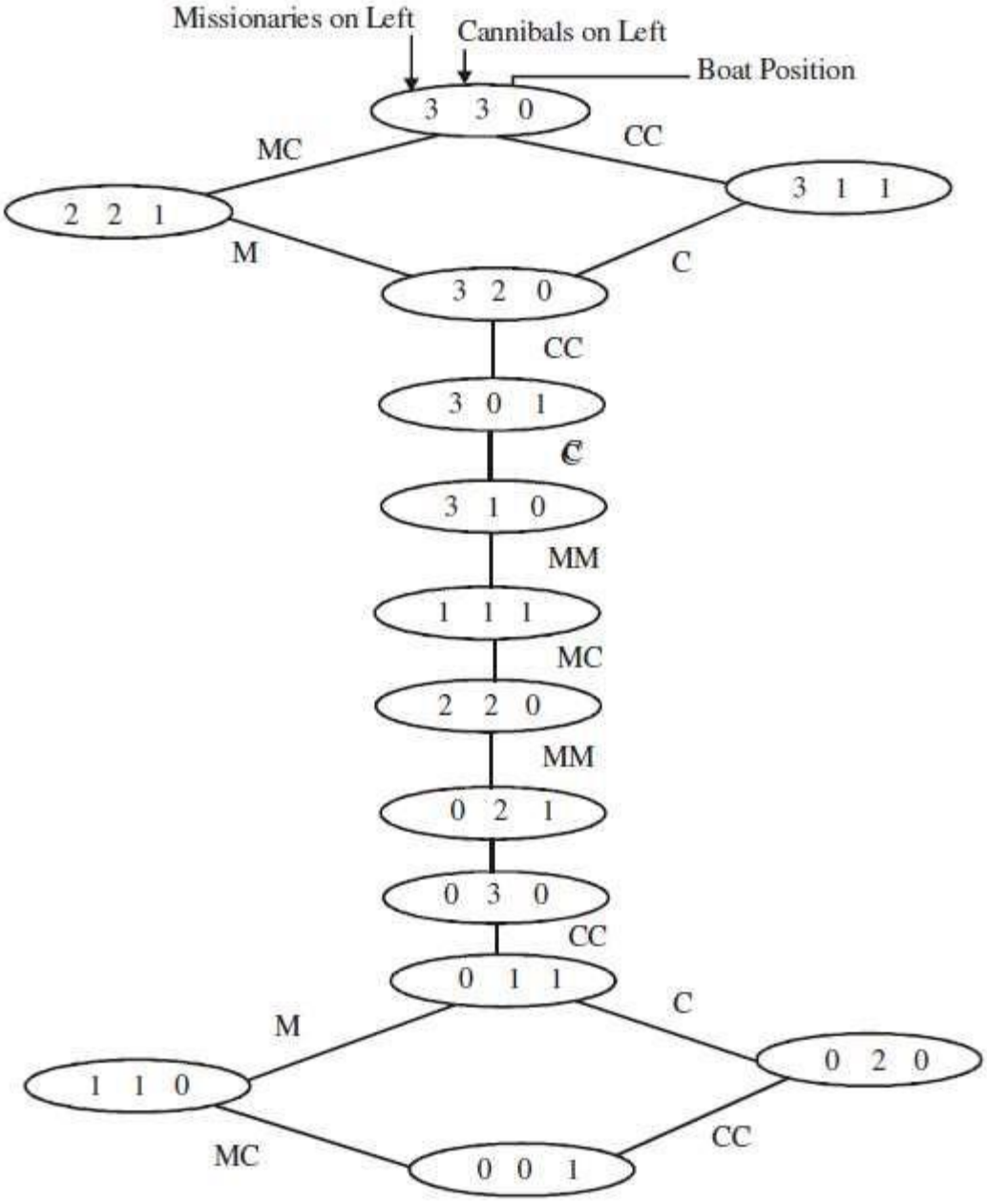
State representation:

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

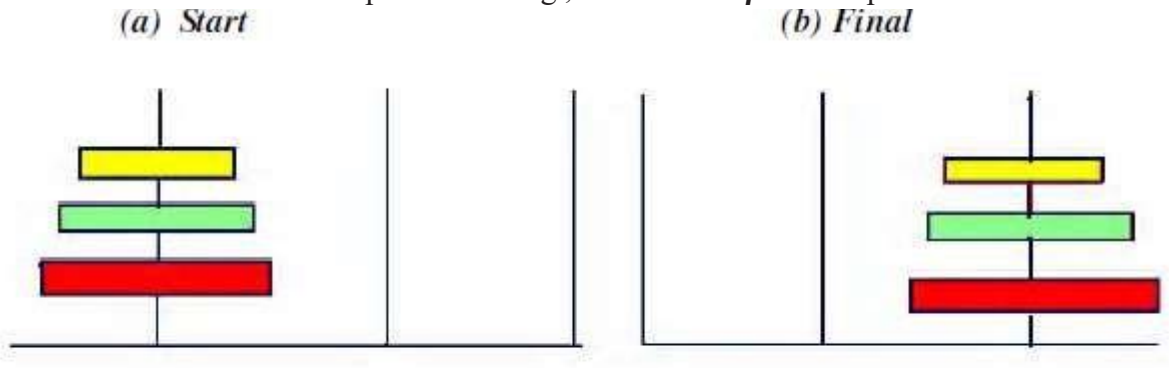


Fig. 2.4 Tower of Hanoi Puzzle

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:

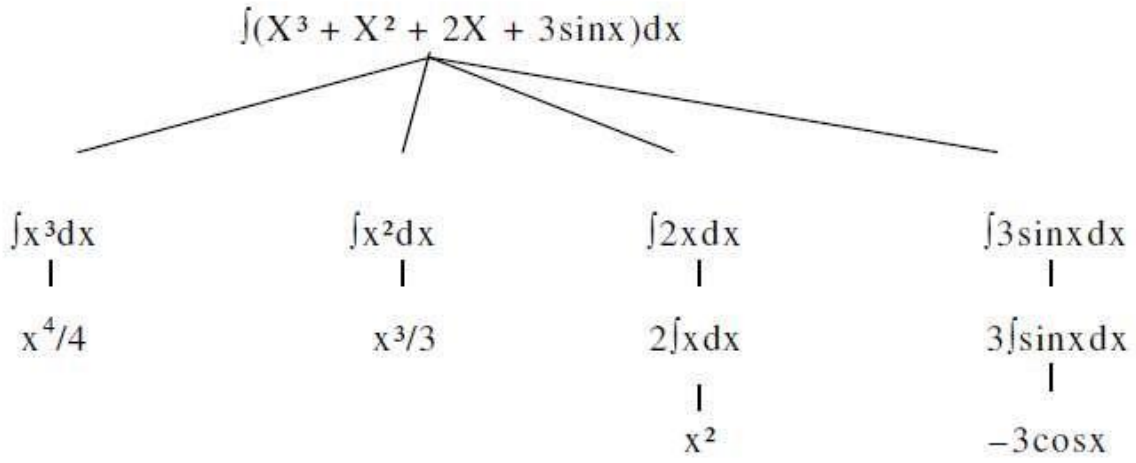
- Decomposability of the problem into a set of independent smaller subproblems
- Possibility of undoing solution steps, if they are found to be unwise
- Predictability of the problem universe
- Possibility of obtaining an obvious solution to a problem without comparison of all other possible solutions

- 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 domainspecific 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: + □(X³ + X² + 2X + 3sinx) 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.