Module 1

ARTIFICIAL INTELLIGENCE

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.

Problems of AI:

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	
instance:	finding
tense:	past
agent:	Rani
object:	Thing1

Event2	
instance:	liking
tense:	past
modifier:	much
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.

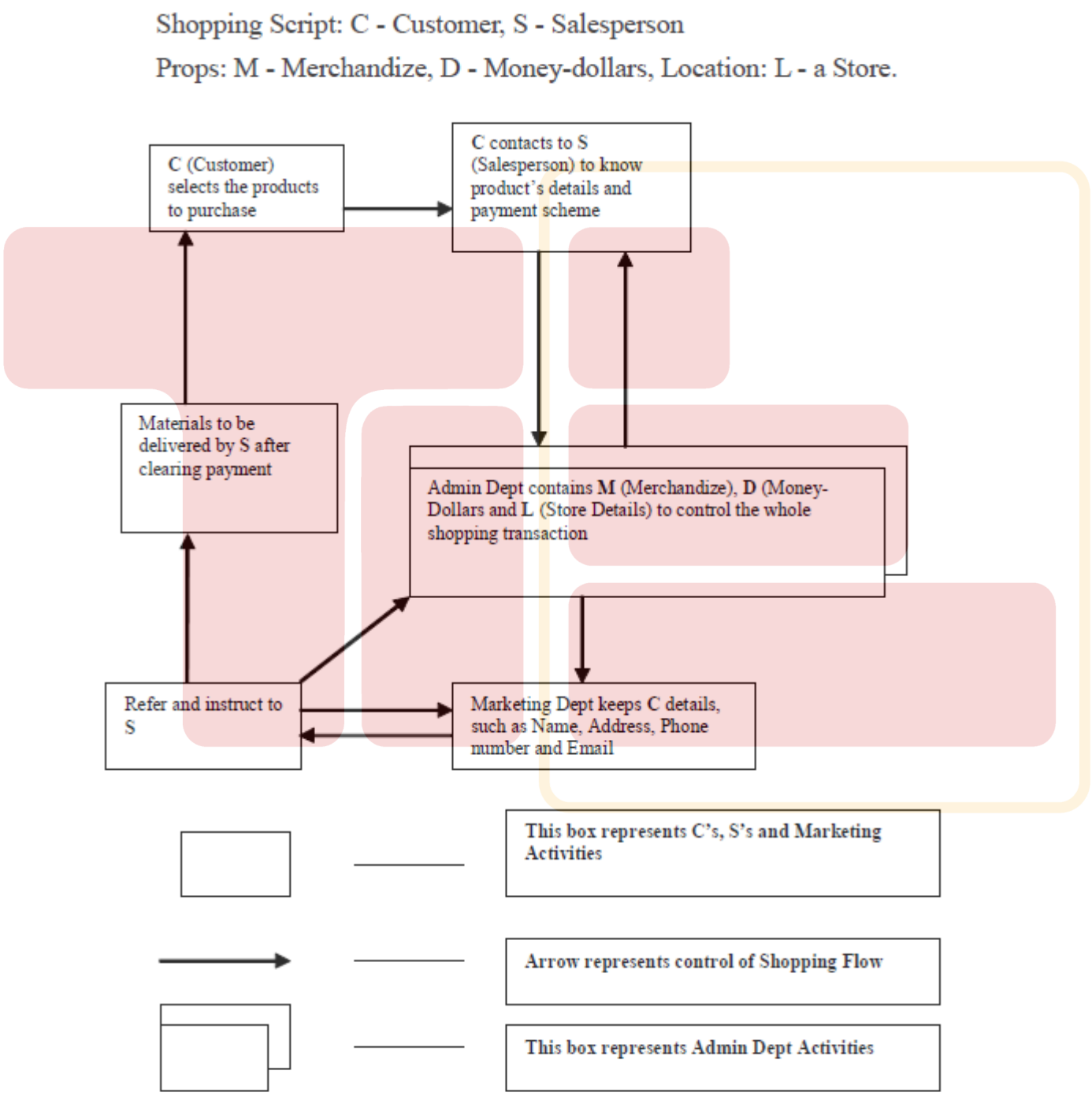


Fig. 1.1 Diagrammatic Representation of Shopping Script

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*.

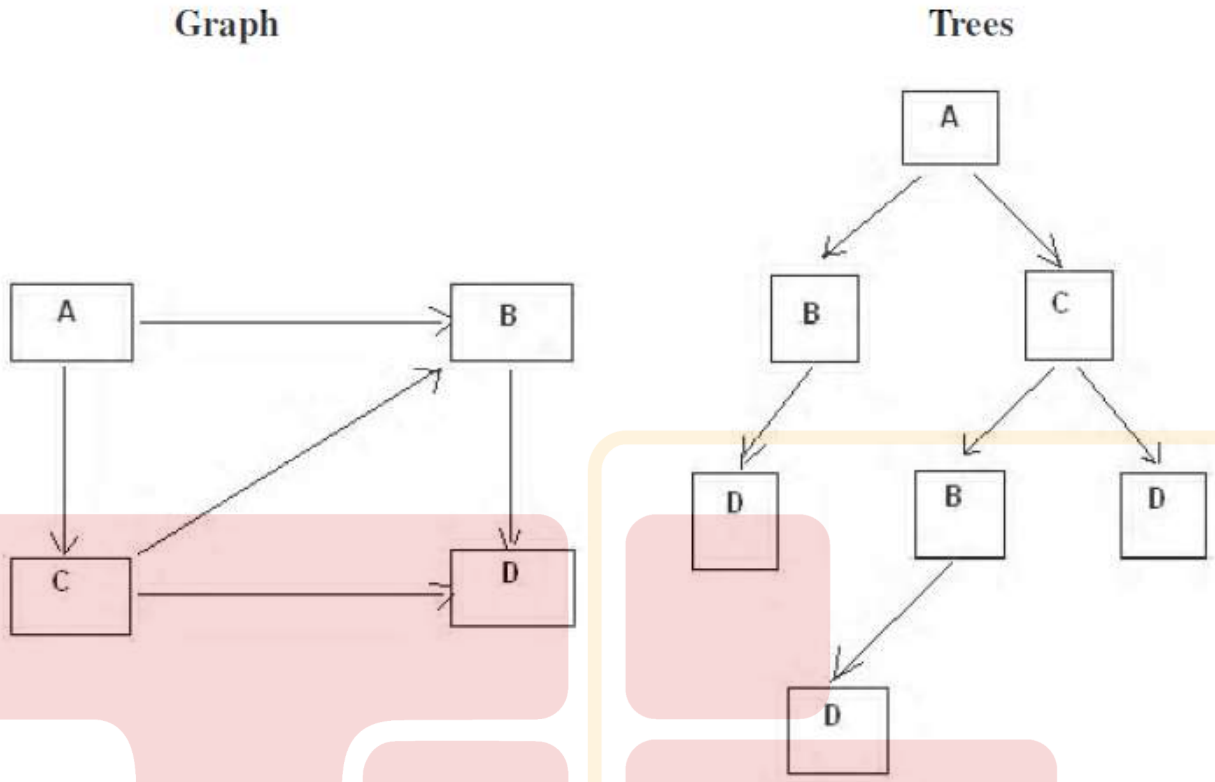


Fig. 2.1 Graph and Tree

• **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 **four** = 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.

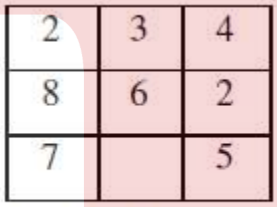
- A set of rules each consisting of a left side that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.
- A database of current facts established during the process of inference.

- 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?

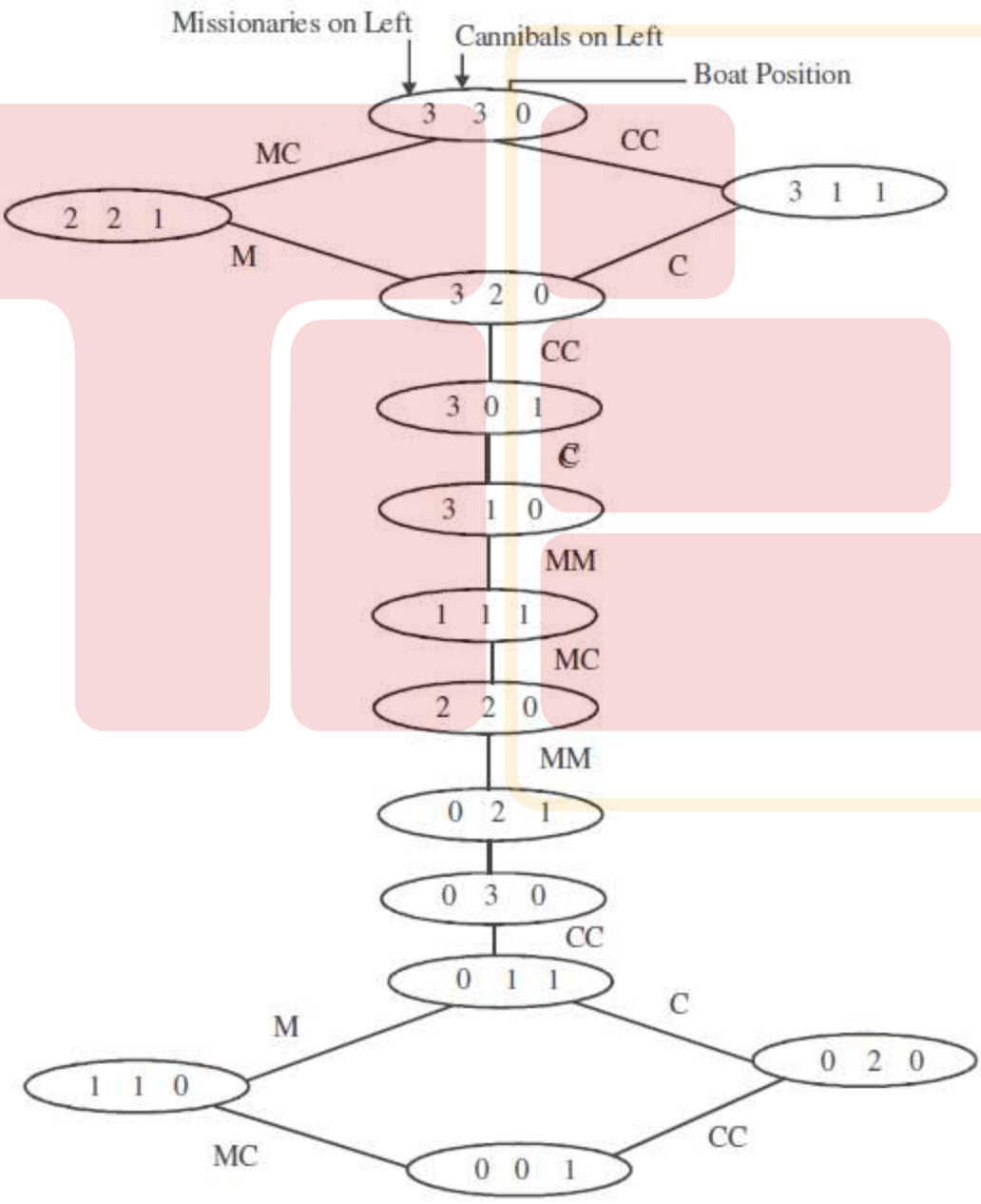
State representation:

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

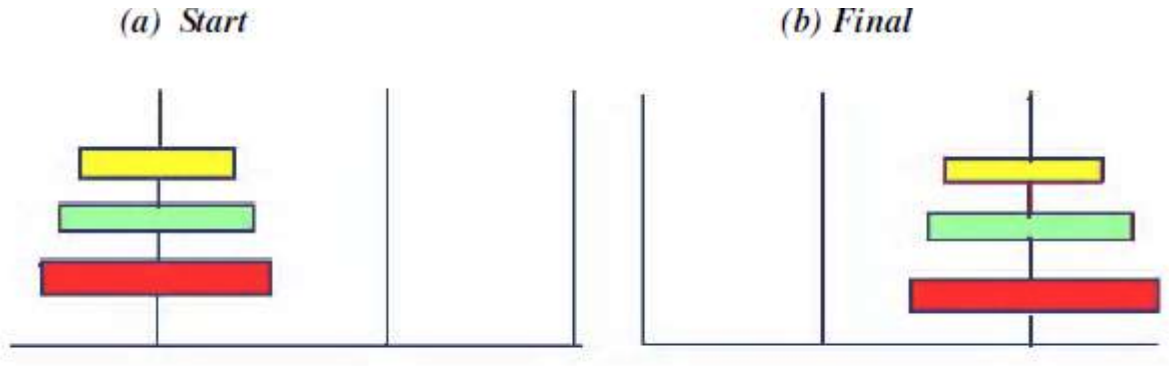


Fig. 2.4 Tower of Hanoi Puzzle

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*.

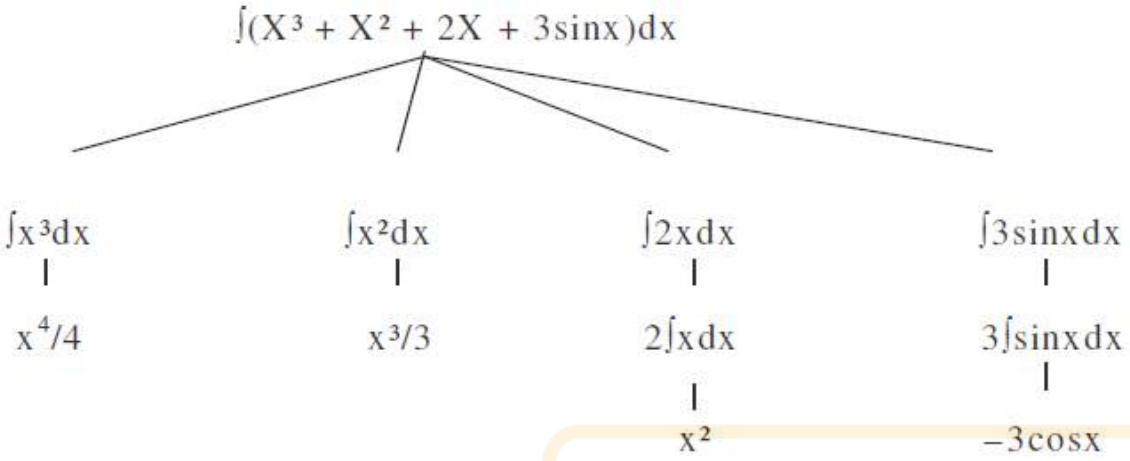
To use the heuristic search for problem solving, we suggest analysis of the problem for the following considerations:

- 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:

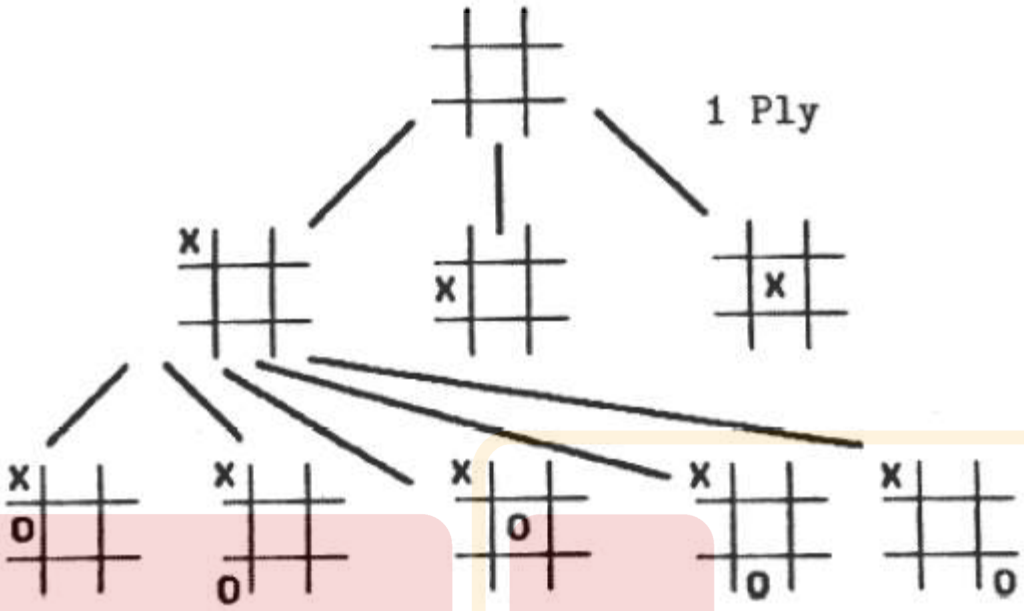
X		0
	0	
X		X

- **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?

HEURISTIC SEARCH TECHNIQUES:

Search Algorithms
Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solutions within some practical time and space limits. Consequently, many special techniques are developed, using ***heuristic functions.***
The algorithms that use *heuristic functions* are called ***heuristic algorithms***.

• Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.
• Heuristic algorithms are more efficient because they take advantage of feedback from the data to direct the search path.
• ***Uninformed search algorithms*** or *Brute-force algorithms*, search through the search space all possible candidates for the solution checking whether each candidate satisfies the problem's statement.
• ***Informed search algorithms*** use heuristic functions that are specific to the problem, apply them to guide the search through the search space to try to reduce the amount of time spent in searching.

A good heuristic will make an informed search dramatically outperform any uninformed search: for example, the Traveling Salesman Problem (TSP), where the goal is to find is a good solution instead of finding the best solution.

In such problems, the search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution. Such techniques help in finding a solution within reasonable time and space (memory). Some prominent intelligent search algorithms are stated below:

1. ***Generate and Test Search***
2. ***Best-first Search***
3. ***Greedy Search***
4. ***A* Search***
5. ***Constraint Search***
6. ***Means-ends analysis***

There are some more algorithms. They are either improvements or combinations of these.
• **Hierarchical Representation of Search Algorithms:** A Hierarchical representation of most search algorithms is illustrated below. The representation begins with two types of search:
• **Uninformed Search:** Also called blind, exhaustive or brute-force search, it uses no information about the problem to guide the search and therefore may not be very efficient.
• **Informed Search:** Also called heuristic or intelligent search, this uses information about the problem to guide the search—usually guesses the distance to a goal state and is therefore efficient, but the search may not be always possible.

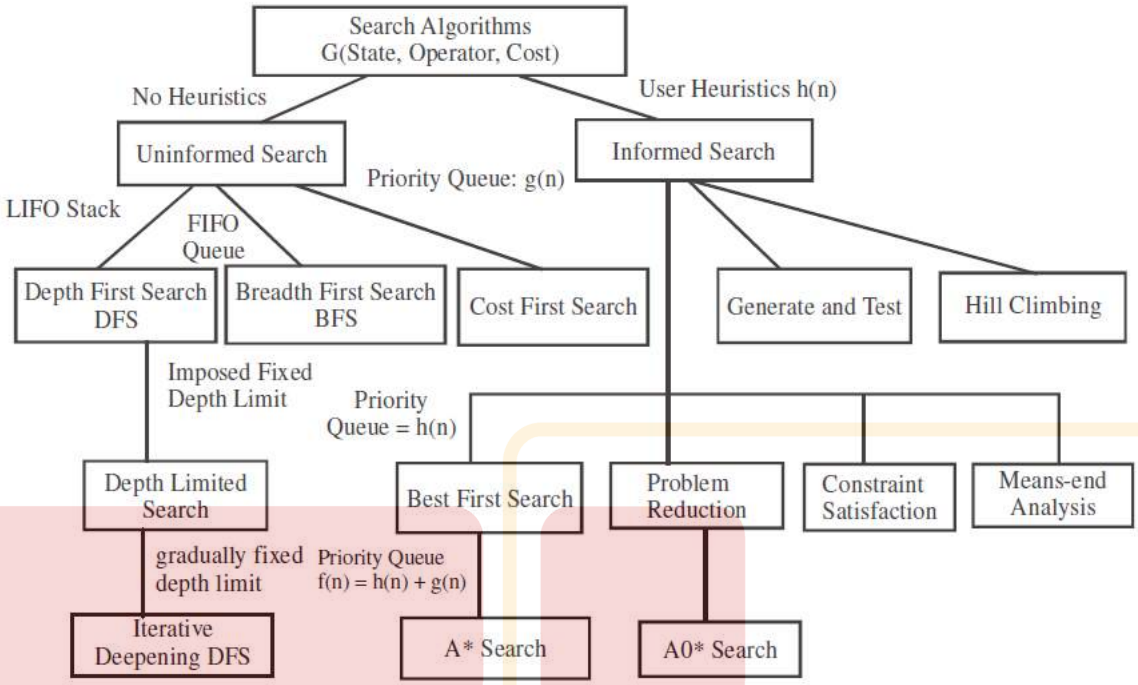
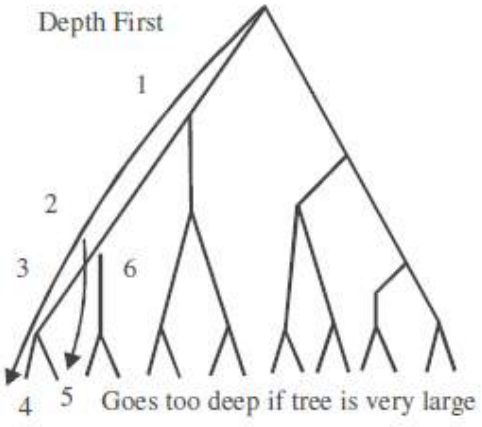
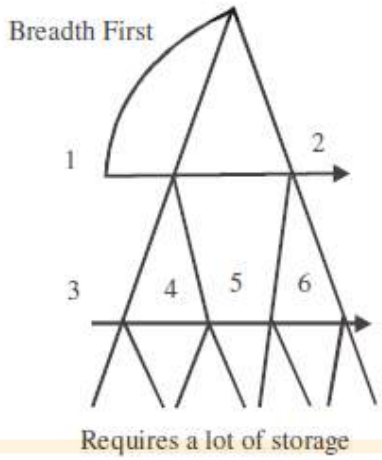


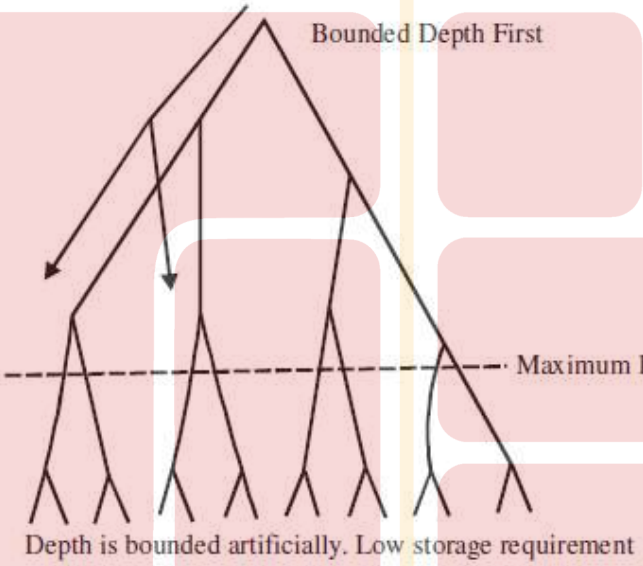
Fig. *Different Search Algorithms*

The first requirement is that it causes motion, in a game playing program, it moves on the board and in the water jug problem, filling water is used to fill jugs. It means the control strategies without the motion will never lead to the solution.

The second requirement is that it is systematic, that is, it corresponds to the need for global motion as well as for local motion. This is a clear condition that neither would it be rational to fill a jug and empty it repeatedly, nor it would be worthwhile to move a piece round and round on the board in a cyclic way in a game. We shall initially consider two systematic approaches for searching. Searches can be classified by the order in which operators are tried: depth-first, breadth-first, bounded depth-first.







Breadth-first search

A Search strategy, in which the highest layer of a decision tree is searched completely before proceeding to the next layer is called *Breadth-first search (BFS).*

• In this strategy, no viable solutions are omitted and therefore it is guaranteed that an optimal solution is found.

• This strategy is often not feasible when the search space is large.

Algorithm

1. Create a variable called LIST and set it to be the starting state.
2. Loop until a goal state is found or LIST is empty, Do

a. Remove the first element from the LIST and call it E. If the LIST is empty, quit.

b. For every path each rule can match the state E, Do

(i) Apply the rule to generate a new state.

(ii) If the new state is a goal state, quit and return this state.

(iii) Otherwise, add the new state to the end of LIST.

Advantages

1. Guaranteed to find an optimal solution (in terms of shortest number of steps to reach the goal).
2. Can always find a goal node if one exists (complete).

Disadvantages

1. High storage requirement: *exponential* with tree depth.

Depth-first search

A search strategy that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path is called *Depth-first search (DFS).*

• This strategy does not guarantee that the optimal solution has been found.

• In this strategy, search reaches a satisfactory solution more rapidly than breadth first, an advantage when the search space is large.

Algorithm

Depth-first search applies operators to each newly generated state, trying to drive directly toward the goal.

1. If the starting state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signalled:

a. Generate a successor E to the starting state. If there are no more successors, then signal failure.
b. Call Depth-first Search with E as the starting state.
c. If success is returned signal success; otherwise, continue in the loop.

Advantages

1. Low storage requirement: *linear* with tree depth.
2. Easily programmed: function call stack does most of the work of maintaining state of the search.

Disadvantages

1. May find a sub-optimal solution (one that is deeper or more costly than the best solution).
2. Incomplete: without a depth bound, may not find a solution even if one exists.

2.4.2.3 Bounded depth-first search

Depth-first search can spend much time (perhaps infinite time) exploring a very deep path that does not contain a solution, when a shallow solution exists. An easy way to solve this problem is to put a maximum depth bound on the search. Beyond the depth bound, a failure is generated automatically without exploring any deeper.

Problems:

1. It's hard to guess how deep the solution lies.
2. If the estimated depth is too deep (even by 1) the computer time used is dramatically increased, by a factor of *bextra*.
3. If the estimated depth is too shallow, the search fails to find a solution; all that computer time is wasted.

Heuristics

A heuristic is a method that improves the efficiency of the search process. These are like tour guides. There are good to the level that they may neglect the points in general interesting directions; they are bad to the level that they may neglect points of interest to particular individuals. Some heuristics help in the search process without sacrificing any claims to entirety that the process might previously had. Others may occasionally cause an excellent path to be overlooked. By sacrificing entirety it increases efficiency. Heuristics may not find the best

solution every time but guarantee that they find a good solution in a reasonable time. These are particularly useful in solving tough and complex problems, solutions of which would require infinite time, i.e. far longer than a lifetime for the problems which are not solved in any other way.

Heuristic search

To find a solution in proper time rather than a complete solution in unlimited time we use heuristics. 'A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers'. Heuristic search methods use knowledge about the problem domain and choose promising operators first. These heuristic search methods use heuristic functions to evaluate the next state towards the goal state. For finding a solution, by using the heuristic technique, one should carry out the following steps:

1. Add domain—specific information to select what is the best path to continue searching along.
2. Define a heuristic function h(n) that estimates the 'goodness' of a node n.

Specifically, h(n) = estimated cost(or distance) of minimal cost path from n to a goal state.

3. The term, heuristic means 'serving to aid discovery' and is an estimate, based on domain specific information that is computable from the current state description of how close we are to a goal.

Finding a route from one city to another city is an example of a search problem in which different search orders and the use of heuristic knowledge are easily understood.

1. State: The current city in which the traveller is located.
2. Operators: Roads linking the current city to other cities.
3. Cost Metric: The cost of taking a given road between cities.
4. Heuristic information: The search could be guided by the direction of the goal city from the current city, or we could use airline distance as an estimate of the distance to the goal.

Heuristic search techniques

For complex problems, the traditional algorithms, presented above, are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed, using ***heuristic functions.***

• Blind search is not always possible, because it requires too much time or Space (memory).

Heuristics are ***rules of thumb***; they do not guarantee a solution to a problem.

• Heuristic Search is a weak technique but can be effective if applied correctly; it requires domain specific information.

Characteristics of heuristic search

• Heuristics are knowledge about domain, which help search and reasoning in its domain.
• Heuristic search incorporates domain knowledge to improve efficiency over blind search.
• Heuristic is a function that, when applied to a state, returns value as estimated merit of state, with respect to goal.

- ✓ Heuristics might (for reasons) *underestimate* or *overestimate* the merit of a state with respect to goal.
- ✓ Heuristics that underestimate are desirable and called admissible.

• Heuristic evaluation function estimates likelihood of given state leading to goal state.
• Heuristic search function estimates cost from current state to goal, presuming function is efficient.

Heuristic search compared with other search
The Heuristic search is compared with Brute force or Blind search techniques below:

Comparison of Algorithms

Brute force / Blind search	**Heuristic search**
Can only search what it has knowledge about already	Estimates 'distance' to goal state through explored nodes
No knowledge about how far a node node from goal state	Guides search process toward goal
	Prefers states (nodes) that lead close to and not away from goal state

Example: Travelling salesman
A salesman has to visit a list of cities and he must visit each city only once. There are different routes between the cities. The problem is to find the shortest route between the cities so that the salesman visits all the cities at once.

Suppose there are N cities, then a solution would be to take N! possible combinations to find the shortest distance to decide the required route. This is not efficient as with N=10 there are 36,28,800 possible routes. This is an example of *combinatorial explosion*.

There are better methods for the solution of such problems: one is called *branch* and *bound*. First, generate all the complete paths and find the distance of the first complete path. If the next path is shorter, then save it and proceed this way avoiding the path when its length exceeds the saved shortest path length, although it is better than the previous method.

Generate and Test Strategy

Generate-And-Test Algorithm
Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.
Algorithm: Generate-And-Test
1.Generate a possible solution.
2.Test to see if this is the expected solution.
3.If the solution has been found quit else go to step 1.
Potential solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state.

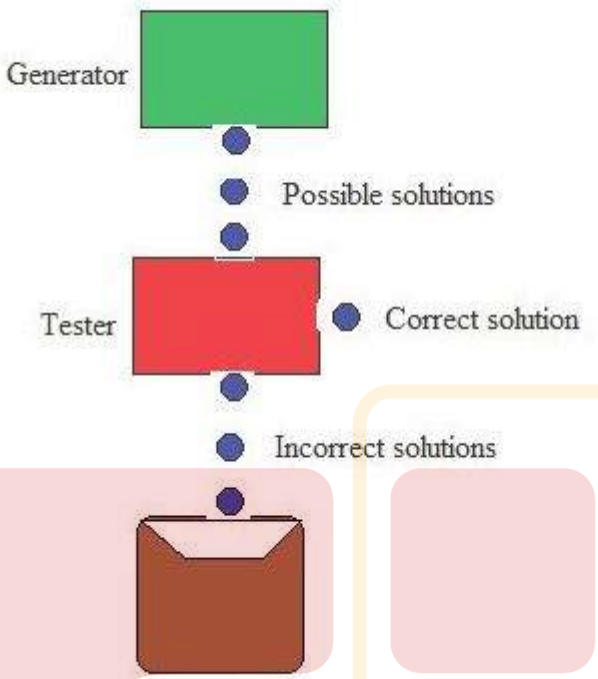


Figure: Generate And Test

Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed. This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

Systematic Generate-And-Test

While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.

Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

Generate-And-Test And Planning

Exhaustive generate-and-test is very useful for simple problems. But for complex problems even heuristic generate-and-test is not very effective technique. But this may be made effective by combining with other techniques in such a way that the space in which to search is restricted. An AI program DENDRAL, for example, uses plan-Generate-and-test technique. First, the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated substructures. Then the generate-and-test procedure uses the lists generated and required to explore only a limited set of structures. Constrained in this way, generate-and-test proved highly effective. A major weakness of planning is that it often produces inaccurate solutions as there is no feedback from the world. But if it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.

Hill Climbing

Hill Climbing is heuristic search used for mathematical optimization problems in the field of Artificial Intelligence .

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

- In the above definition, mathematical optimization problems implies that hill climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in reasonable time.
- A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

Features of Hill Climbing

1. Variant of generate and test algorithm : It is a variant of generate and test algorithm. The generate and test algorithm is as follows :

1. Generate a possible solutions.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.

Hence we call Hill climbing as a variant of generate and test algorithm as it takes the feedback from test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.

2. Uses the Greedy approach : At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.

Types of Hill Climbing

1. Simple Hill climbing : It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node.
   Algorithm for Simple Hill climbing :

Step 1 : Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.
Step 2 : Loop until the solution state is found or there are no new operators present which can be applied to current state.
a) Select a state that has not been yet applied to the current state and apply it to produce a new state.
b) Perform these to evaluate new state
i. If the current state is a goal state, then stop and return success.
ii. If it is better than the current state, then make it current state and proceed further.
iii. If it is not better than the current state, then continue in the loop until a solution is found.

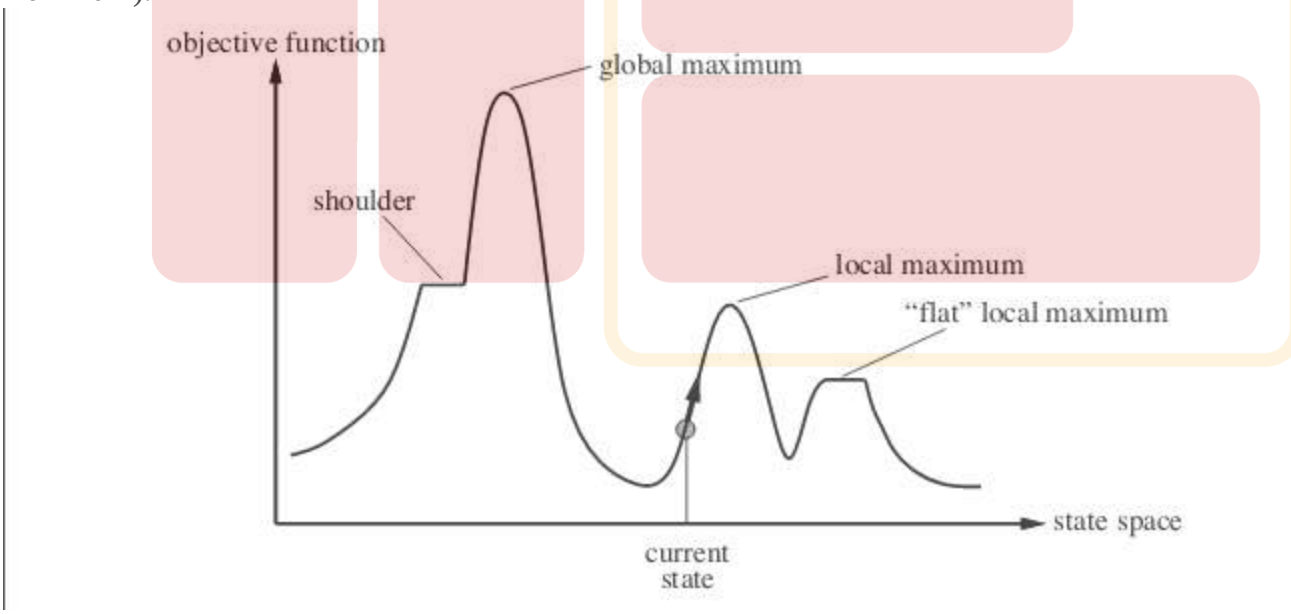
Step 3 : Exit.

2. Steepest-Ascent Hill climbing : It first examines all the neighboring nodes and then selects the node closest to the solution state as next node.

Step 1 : Evaluate the initial state. If it is goal state then exit else make the current state as initial state
Step 2 : Repeat these steps until a solution is found or current state does not change
i. Let 'target' be a state such that any successor of the current state will be better than it;
ii. for each operator that applies to the current state
a. apply the new operator and create a new state
b. evaluate the new state
c. if this state is goal state then quit else compare with 'target'
d. if this state is better than 'target', set this state as 'target'
e. if target is better than current state set current state to Target
Step 3 : Exit

3. Stochastic hill climbing : It does not examine all the neighboring nodes before deciding which node to select .It just selects a neighboring node at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

State Space diagram for Hill Climbing
State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).
X-axis : denotes the state space ie states or configuration our algorithm may reach.
Y-axis : denotes the values of objective function corresponding to to a particular state.
The best solution will be that state space where objective function has maximum value(global maximum).



Different regions in the State Space Diagram

1. Local maximum : It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here value of objective function is higher than its neighbors.

2. Global maximum : It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
3. Plateua/flat local maximum : It is a flat region of state space where neighboring states have the same value.
4. Ridge : It is region which is higher than its neighbours but itself has a slope. It is a special kind of local maximum.
5. Current state : The region of state space diagram where we are currently present during the search.
6. Shoulder : It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

1. Local maximum : At a local maximum all neighboring states have a values which is worse than than the current state. Since hill climbing uses greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
   To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
2. Plateau : On plateau all neighbors have same value . Hence, it is not possible to select the best direction.

To overcome plateaus : Make a big jump. Randomly select a state far away from current state. Chances are that we will land at a non-plateau region

3. Ridge : Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
   To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Best First Search (Informed Search)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

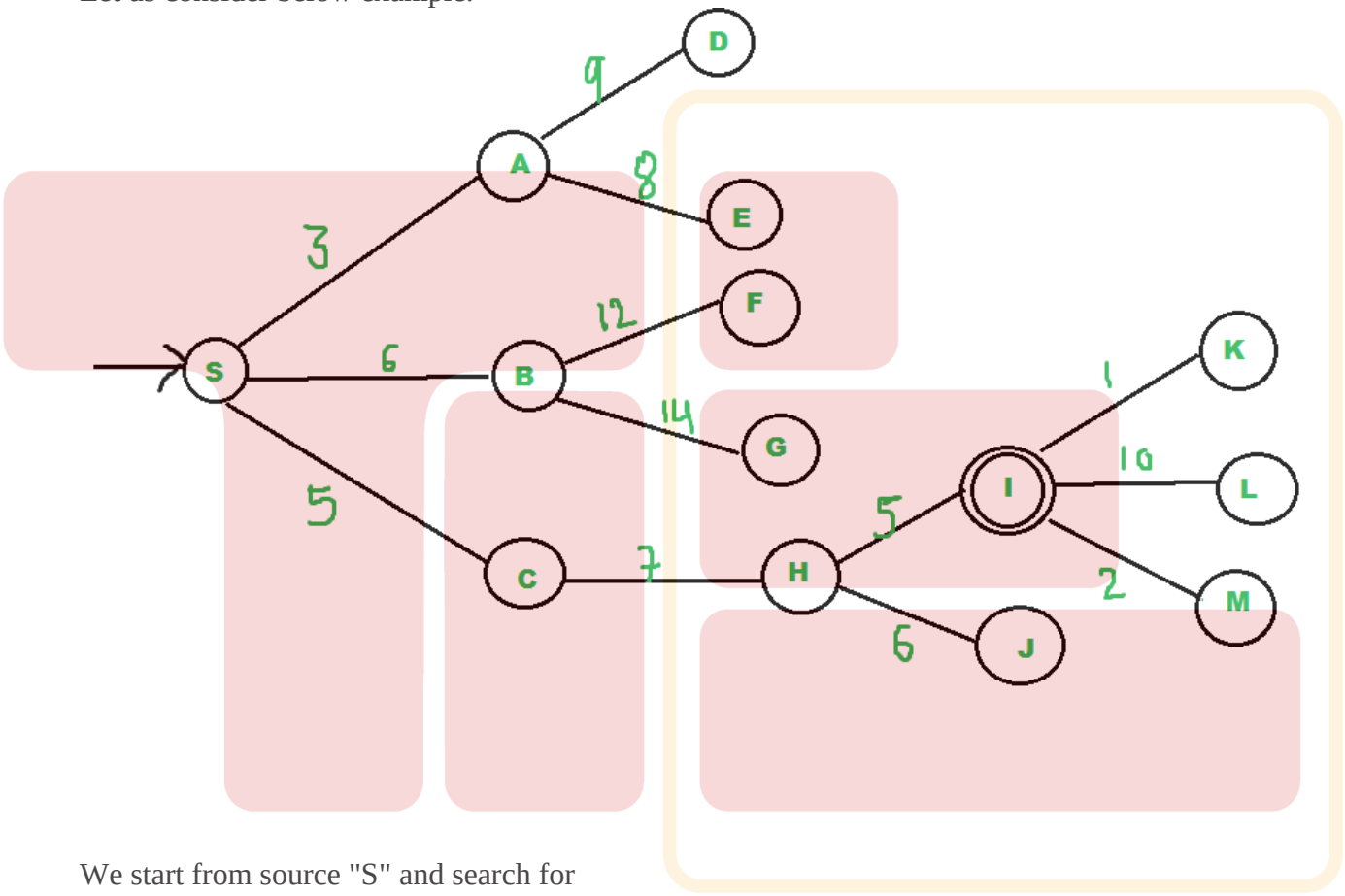
Algorithm:

Best-First-Search(Grah g, Node start)

1) Create an empty PriorityQueue
   PriorityQueue pq;
2) Insert "start" in pq.
   pq.insert(start)
3) Until PriorityQueue is empty
   u = PriorityQueue.DeleteMin

```
        If u is the goal
          Exit
        Else
          Foreach neighbor v of u
            If v "Unvisited"
              Mark v "Visited"
              pq.insert(v)
          Mark v "Examined"
End procedure
```

Let us consider below example.



We start from source "S" and search for goal "I" using given costs and Best First search.

pq initially contains S
We remove s from and process unvisited neighbors of S to pq.
pq now contains {A, C, B} (C is put before B because C has lesser cost)

We remove A from pq and process unvisited neighbors of A to pq.
pq now contains {C, B, E, D}

We remove C from pq and process unvisited
neighbors of C to pq.
pq now contains {B, H, E, D}

We remove B from pq and process unvisited
neighbors of B to pq.
pq now contains {H, E, D, F, G}

We remove H from pq. Since our goal
"I" is a neighbor of H, we return.

Analysis :

- The worst case time complexity for Best First Search is O(n * Log n) where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take O(log n) time.
- Performance of the algorithm depends on how well the cost or evaluation function is designed.

A* Search Algorithm

A* is a type of search algorithm. Some problems can be solved by representing the world in the initial state, and then for each action we can perform on the world we generate states for what the world would be like if we did so. If you do this until the world is in the state that we specified as a solution, then the route from the start to this goal state is the solution to your problem.

In this tutorial I will look at the use of state space search to find the shortest path between two points (pathfinding), and also to solve a simple sliding tile puzzle (the 8-puzzle). Let's look at some of the terms used in Artificial Intelligence when describing this state space search.

Some terminology

A *node* is a state that the problem's world can be in. In pathfinding a node would be just a 2d coordinate of where we are at the present time. In the 8-puzzle it is the positions of all the tiles. Next all the nodes are arranged in a *graph* where links between nodes represent valid steps in solving the problem. These links are known as *edges*. In the 8-puzzle diagram the edges are shown as blue lines. See figure 1 below.
State space search, then, is solving a problem by beginning with the start state, and then for each node we expand all the nodes beneath it in the graph by applying all the possible moves that can be made at each point.

Heuristics and Algorithms

At this point we introduce an important concept, the *heuristic*. This is like an algorithm, but with a key difference. An algorithm is a set of steps which you can follow to solve a problem, which always works for valid input. For example you could probably write an algorithm yourself for

multiplying two numbers together on paper. A heuristic is not guaranteed to work but is useful in that it may solve a problem for which there is no algorithm.
We need a heuristic to help us cut down on this huge search problem. What we need is to use our heuristic at each node to make an estimate of how far we are from the goal. In pathfinding we know exactly how far we are, because we know how far we can move each step, and we can calculate the exact distance to the goal.
But the 8-puzzle is more difficult. There is no known algorithm for calculating from a given position how many moves it will take to get to the goal state. So various heuristics have been devised. The best one that I know of is known as the Nilsson score which leads fairly directly to the goal most of the time, as we shall see.

Cost

When looking at each node in the graph, we now have an idea of a heuristic, which can estimate how close the state is to the goal. Another important consideration is the cost of getting to where we are. In the case of pathfinding we often assign a movement cost to each square. The cost is the same then the cost of each square is one. If we wanted to differentiate between terrain types we may give higher costs to grass and mud than to newly made road. When looking at a node we want to add up the cost of what it took to get here, and this is simply the sum of the cost of this node and all those that are above it in the graph.

8 Puzzle
Let's look at the 8 puzzle in more detail. This is a simple sliding tile puzzle on a 3*3 grid where one tile is missing and you can move the other tiles into the gap until you get the puzzle into the goal position. See figure 1.

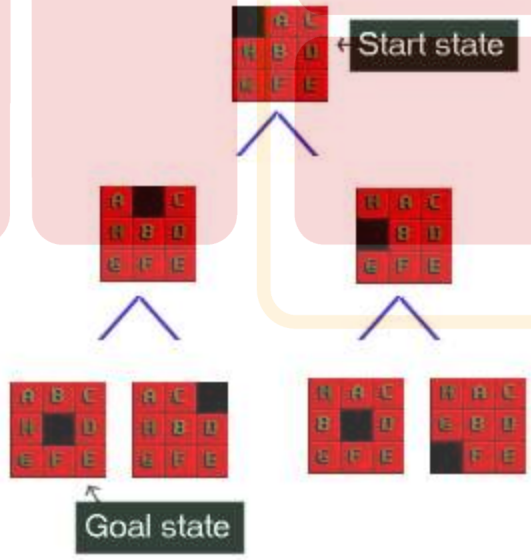


Figure 1 : The 8-Puzzle state space for a very simple example

There are 362,880 different states that the puzzle can be in, and to find a solution the search has to find a route through them. From most positions of the search the number of edges (that's the

blue lines) is two. That means that the number of nodes you have in each level of the search is 2^d where d is the depth. If the number of steps to solve a particular state is 18, then that's 262,144 nodes just at that level.

The 8 puzzle game state is as simple as representing a list of the 9 squares and what's in them. Here are two states for example; the last one is the GOAL state, at which point we've found the solution. The first is a jumbled up example that you may start from.

Start state SPACE, A, C, H, B, D, G, F, E
Goal state A, B, C, H, SPACE, D, G, F, E
The rules that you can apply to the puzzle are also simple. If there is a blank tile above, below, to the left or to the right of a given tile, then you can move that tile into the space. To solve the puzzle you need to find the path from the start state, through the graph down to the goal state.

There is example code to to solve the 8-puzzle on the github site.

Pathfinding

In a video game, or some other pathfinding scenario, you want to search a state space and find out how to get from somewhere you are to somewhere you want to be, without bumping into walls or going too far. For reasons we will see later, the A* algorithm will not only find a path, if there is one, but it will find the shortest path. A state in pathfinding is simply a position in the world. In the example of a maze game like Pacman you can represent where everything is using a simple 2d grid. The start state for a ghost say, would be the 2d coordinate of where the ghost is at the start of the search. The goal state would be where pacman is so we can go and eat him. There is also example code to do pathfinding on the github site.

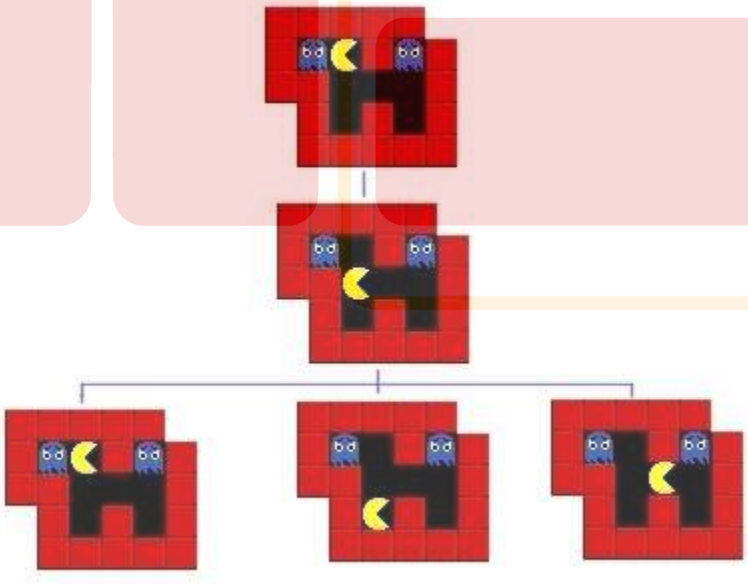


Figure 2 : The first three steps of a pathfinding state space

Implementing A*

We are now ready to look at the operation of the A* algorithm. What we need to do is start with the goal state and then generate the graph downwards from there. Let's take the 8-puzzle in figure 1. We ask how many moves can we make from the start state? The answer is 2, there are two directions we can move the blank tile, and so our graph expands.

If we were just to continue blindly generating successors to each node, we could potentially fill the computer's memory before we found the goal node. Obviously we need to remember the best nodes and search those first. We also need to remember the nodes that we have expanded already, so that we don't expand the same state repeatedly.

Let's start with the OPEN list. This is where we will remember which nodes we haven't yet expanded. When the algorithm begins the start state is placed on the open list, it is the only state we know about and we have not expanded it. So we will expand the nodes from the start and put those on the OPEN list too. Now we are done with the start node and we will put that on the CLOSED list. The CLOSED list is a list of nodes that we have expanded.

$f = g + h$

Using the OPEN and CLOSED list lets us be more selective about what we look at next in the search. We want to look at the best nodes first. We will give each node a score on how good we think it is. This score should be thought of as the cost of getting from the node to the goal plus the cost of getting to where we are. Traditionally this has been represented by the letters f, g and h. 'g' is the sum of all the costs it took to get here, 'h' is our heuristic function, the estimate of what it will take to get to the goal. 'f' is the sum of these two. We will store each of these in our nodes.

Using the f, g and h values the A* algorithm will be directed, subject to conditions we will look at further on, towards the goal and will find it in the shortest route possible.

So far we have looked at the components of the A*, let's see how they all fit together to make the algorithm :

Pseudocode

Hopefully the ideas we looked at in the preceding paragraphs will now click into place as we look at the A* algorithm pseudocode. You may find it helpful to print this out or leave the window open while we discuss it.

To help make the operation of the algorithm clear we will look again at the 8-puzzle problem in figure 1 above. Figure 3 below shows the f,g and h scores for each of the tiles.

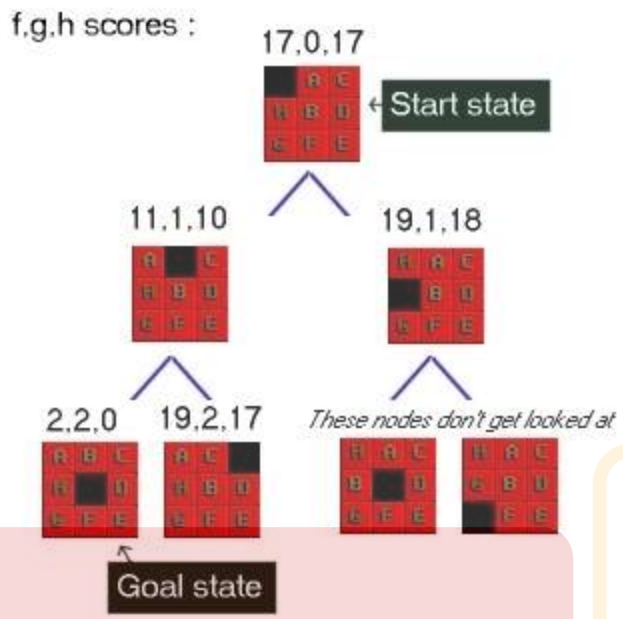


Figure 3 : 8-Puzzle state space showing f,g,h scores

First of all look at the g score for each node. This is the cost of what it took to get from the start to that node. So in the picture the center number is g. As you can see it increases by one at each level. In some problems the cost may vary for different state changes. For example in pathfinding there is sometimes a type of terrain that costs more than other types.
Next look at the last number in each triple. This is h, the heuristic score. As I mentioned above I am using a heuristic known as Nilsson's Sequence, which converges quickly to a correct solution in many cases. Here is how you calculate this score for a given 8-puzzle state :

Advantages:

It is complete and optimal.

It is the best one from other techniques. It is used to solve very complex problems.

It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

Disadvantages:

This algorithm is complete if the branching factor is finite and every action has fixed cost.

The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n).

AO* Search: (And-Or) Graph

The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

OPEN:

It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.

CLOSE:

It contains the nodes that have already been processed.

6 7: The distance from current node to goal node.

Algorithm:

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T0.

Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in

CLOSE

Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.

Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.

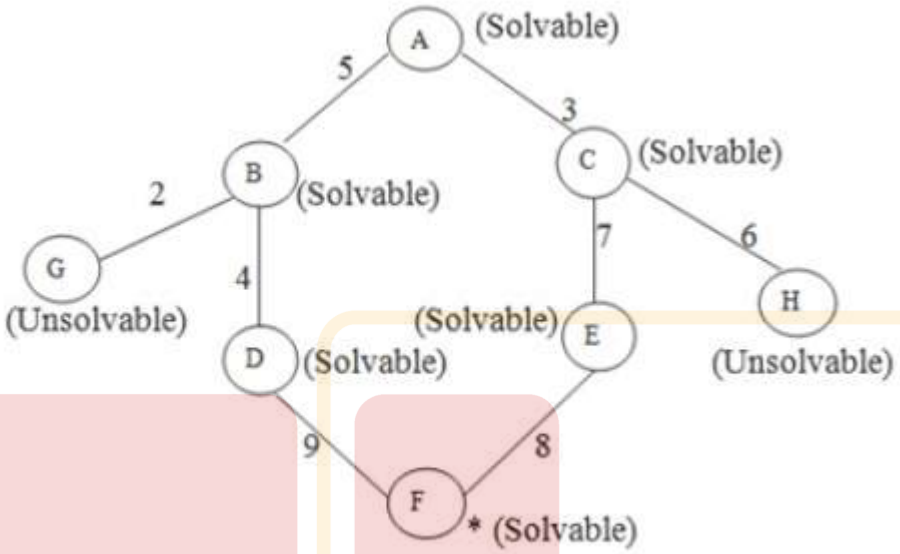
Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

Step 7: Return to Step 2.

Step 8: Exit.

Implementation:

Let us take the following example to implement the AO* algorithm.



Figure

Step 1:

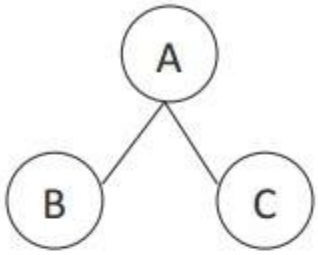
In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H. Take A as the starting node. So place A into OPEN.

i.e. OPEN = A CLOSE = (NULL)

Step 2:

The children of A are B and C which are solvable. So place them into OPEN and place A into the CLOSE.

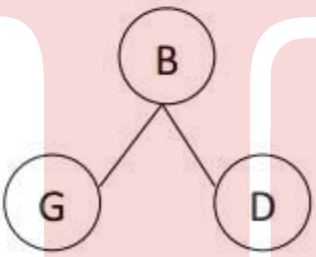
i.e. OPEN = | B | C | CLOSE = | A |



Step 3:

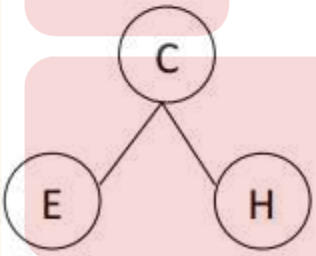
Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.

So OPEN = | G | D | E | |



(O)

C | A | B | C |

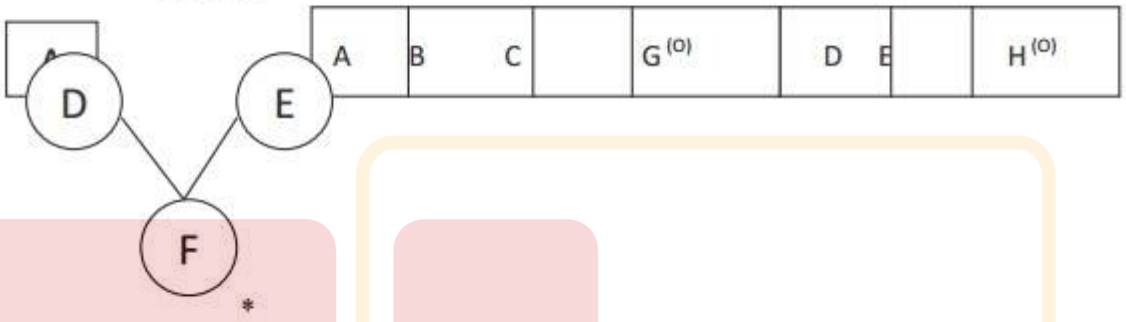


'O' indicated that the nodes G and H are unsolvable.

Step 4:

As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN = CLOSE =



Step 5:

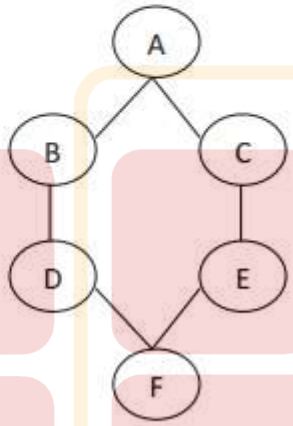
Now we have been reached at our goal state. So place F into CLOSE.

i.e. CLOSE =

Step 6:

Success and Exit

AO* Graph:



Figure

Advantages:

It is an optimal algorithm.

If traverse according to the ordering of nodes. It can be used for both OR and AND graph.

Disadvantages:

Sometimes for unsolvable nodes, it can't find the optimal path. Its complexity is than other algorithms.

PROBLEM REDUCTION

Problem Reduction with AO* Algorithm.

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All

these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

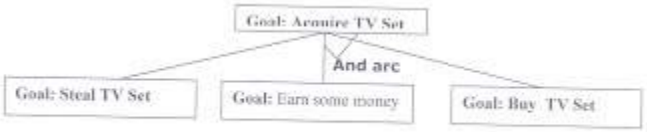


Figure shows AND - Or graph - an example.

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

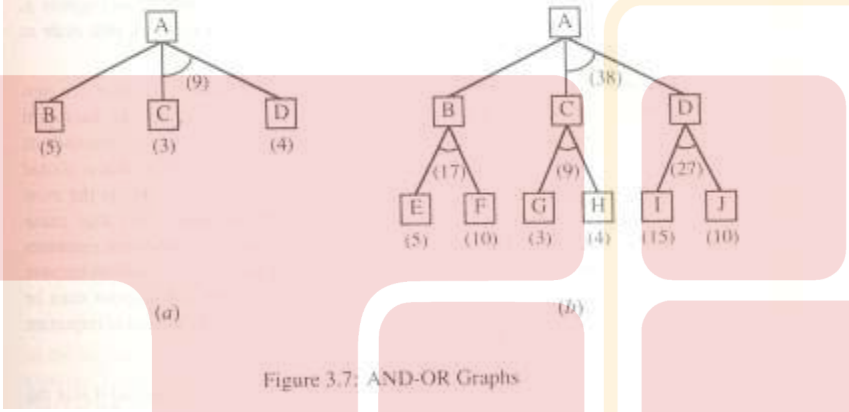


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

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

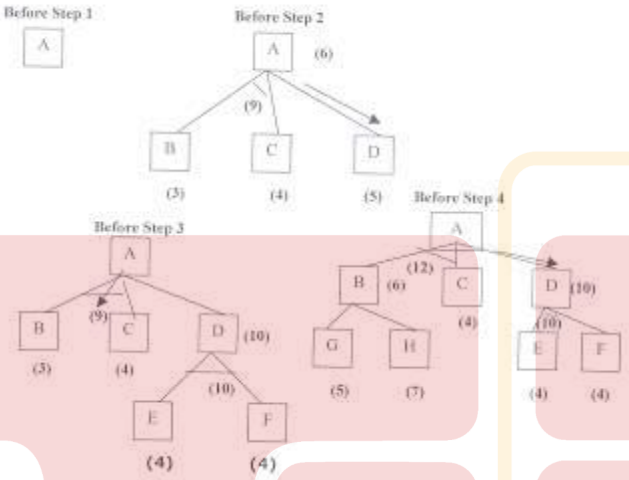
Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current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

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

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

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

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



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

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

AO* ALGORITHM:
1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).

2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

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

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

(a) add SUCCESSOR to graph G

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

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

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

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

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

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

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

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

(Refered From Artificial Intelligence TMH)

AO* Search Procedure.

1. Place the start node on open.

2. Using the search tree, compute the most promising solution tree TP .

3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.

4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.

5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open ,with unsolvable ancestors.

6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.

7. Go back to step(2)

Note: AO* will always find minimum cost solution.

CONSTRAINT SATISFACTION:-

Many problems in AI can be considered as problems of constraint satisfaction, in which the goal state satisfies a given set of constraint. constraint satisfaction problems can be solved by using any of the search strategies. The general form of the constraint satisfaction procedure is as follows:

Until a complete solution is found or until all paths have led to lead ends, do

1. select an unexpanded node of the search graph.

2. Apply the constraint inference rules to the selected node to generate all possible new constraints.

3. If the set of constraints contains a contradiction, then report that this path is a dead end.

4. If the set of constraints describes a complete solution then report success.

5. If neither a constraint nor a complete solution has been found then apply the rules to generate new partial solutions. Insert these partial solutions into the search graph.

Example: consider the crypt arithmetic problems.

```
   SEND
+ MORE
----------
MONEY
----------
```

Assign decimal digit to each of the letters in such a way that the answer to the problem is correct to the same letter occurs more than once , it must be assign the same digit each time . no two different letters may be assigned the same digit. Consider the crypt arithmetic problem.

```
  SEND
+ MORE
----------
MONEY
----------
```

CONSTRAINTS:-

1. no two digit can be assigned to same letter.

2. only single digit number can be assign to a letter.

1. no two letters can be assigned same digit.

2. Assumption can be made at various levels such that they do not contradict each other.

3. The problem can be decomposed into secured constraints. A constraint satisfaction approach may be used.

4. Any of search techniques may be used.

5. Backtracking may be performed as applicable us applied search techniques.

6. Rule of arithmetic may be followed.

Initial state of problem.
D=?
E=?
Y=?
N=?
R=?
O=?
S=?
M=?
C1=?
C2=?
C1 ,C 2, C3 stands for the carry variables respectively.

Goal State: the digits to the letters must be assigned in such a manner so that the sum is satisfied.

Solution Process:

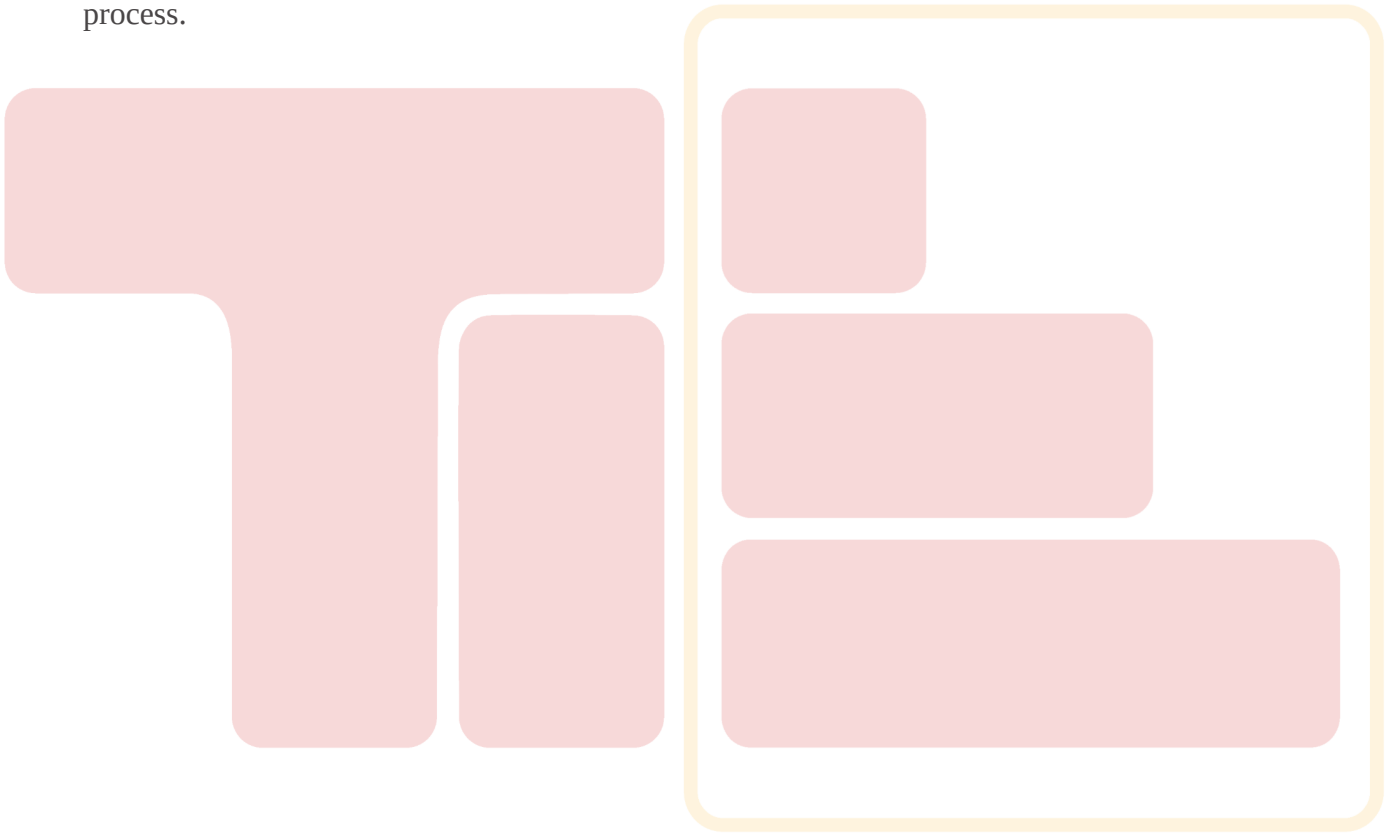
We are following the depth-first method to solve the problem.

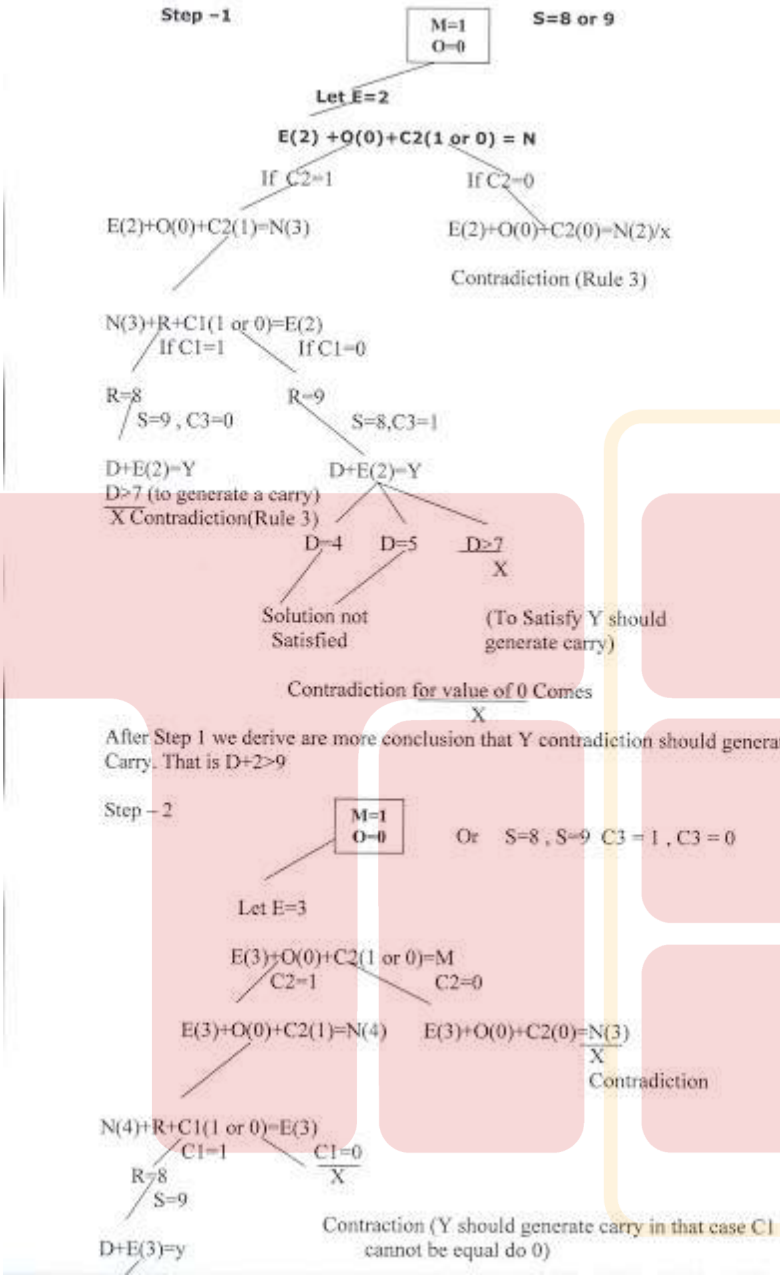
1. initial guess m=1 because the sum of two single digits can generate at most a carry '1'.

2. When n=1 o=0 or 1 because the largest single digit number added to m=1 can generate the sum of either 0 or 1 depend on the carry received from the carry sum. By this we conclude that o=0 because m is already 1 hence we cannot assign same digit another letter(rule no.)

3. We have m=1 and o=0 to get o=0 we have s=8 or 9, again depending on the carry received from the earlier sum.

The same process can be repeated further. The problem has to be composed into various constraints. And each constraints is to be satisfied by guessing the possible digits that the letters can be assumed that the initial guess has been already made . rest of the process is being shown in the form of a tree, using depth-first search for the clear understandability of the solution process.

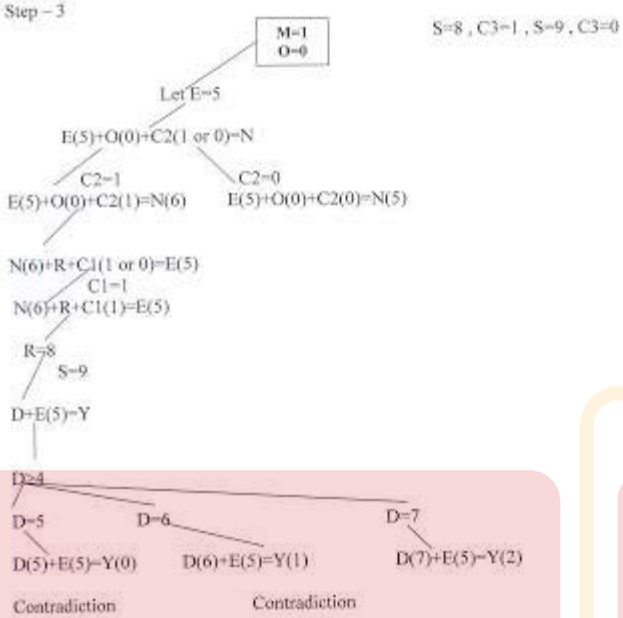




D>6(Controduction)

After Step 2 , we found that C1 cannot be Zero, Since Y has to generate a carry to satisfy goal state. From this step onwards, no need to branch for C1=0.

Step – 3



At Step (4) we have assigned a single digit to every letter in accordance with the constraints & production rules.

Now by backtracking , we find the different digits assigned to different letters and hence reach the solution state.

Solution State:-

Y = 2
D = 7
S = 9
R = 8
N = 6
E = 5
O = 0
M = 1
C1 = 1
C2 = 0
C3 = 0

	C3(0)	C2(1)	C1(1)	
	S(9)	E(5)	N(6)	D(7)
+	M(1)	O(0)	R(8)	E(5)
M(1)	O(0)	N(6)	E(5)	Y(2)

MEANS - ENDS ANALYSIS:-

Most of the search strategies either reason forward of backward however, often a mixture o the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems the arise when combining them together. Such a technique is called "Means - Ends Analysis".

The means -ends analysis process centers around finding the difference between current state and goal state. The problem space of means - ends analysis has an initial state and one or more goal state, a set of operate with a set of preconditions their application and difference functions that computes the difference between two state a(i) and s(j). A problem is solved using means - ends analysis by

1. Computing the current state s1 to a goal state s2 and computing their difference D12.

2. Satisfy the preconditions for some recommended operator op is selected, then to reduce the difference D12.

3. The operator OP is applied if possible. If not the current state is solved a goal is created and means- ends analysis is applied recursively to reduce the sub goal.

4. If the sub goal is solved state is restored and work resumed on the original problem.

(the first AI program to use means - ends analysis was the GPS General problem solver)

means- ends analysis I useful for many human planning activities. Consider the example of planing for an office worker. Suppose we have a different table of three rules:

1. If in out current state we are hungry , and in our goal state we are not hungry , then either the "visit hotel" or "visit Canteen " operator is recommended.

2. If our current state we do not have money , and if in your goal state we have money, then the "Visit our bank" operator or the "Visit secretary" operator is recommended.

3. If our current state we do not know where something is , need in our goal state we do know, then either the "visit office enquiry" , "visit secretary" or "visit co worker " operator is recommended.

KNOWLEDGE REPRESENTATION

KNOWLEDGE REPRESENTATION:-

For the purpose of solving complex problems c\encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. In all variety of knowledge representations , we deal with two kinds of entities.

A. Facts: Truths in some relevant world. These are the things we want to represent.

B. Representations of facts in some chosen formalism . these are things we will

actually be able to manipulate.

One way to think of structuring these entities is at two levels : (a) the knowledge level, at which facts are described, and (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

The facts and representations are linked with two-way mappings. This link is called representation mappings. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

One common representation is natural language (particularly English) sentences. Regardless of the representation for facts we use in a program , we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. We need mapping functions from English sentences to the representation we actually use and from it back to sentences.

Representations and Mappings

- In order to solve complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions.
- Knowledge and Representation are two distinct entities. They play central but distinguishable roles in the intelligent system.
- Knowledge is a description of the world. It determines a system’s competence by what it knows.
- Moreover, Representation is the way knowledge is encoded. It defines a system’s performance in doing something.
- Different types of knowledge require different kinds of representation.

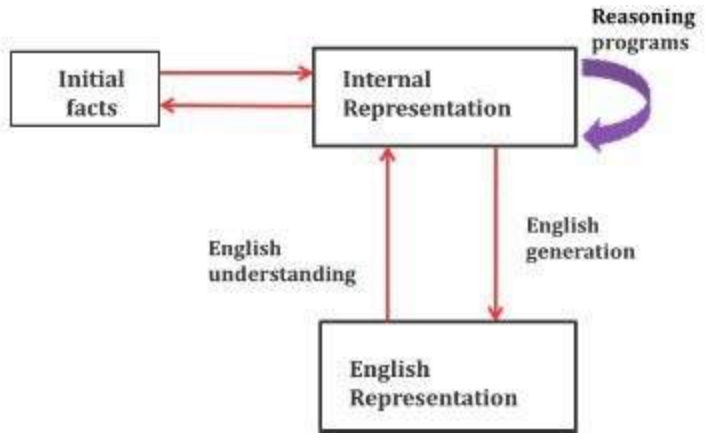


Fig: Mapping between Facts and Representations

The Knowledge Representation models/mechanisms are often based on:

- Logic
- Rules
- Frames
- Semantic Net

Knowledge is categorized into two major types:

1. Tacit corresponds to “informal” or “implicit“
    - Exists within a human being;
    - It is embodied.
    - Difficult to articulate formally.
    - Difficult to communicate or share.
    - Moreover, Hard to steal or copy.
    - Drawn from experience, action, subjective insight
2. Explicit formal type of knowledge, Explicit
    - Explicit knowledge
    - Exists outside a human being;
    - It is embedded.
    - Can be articulated formally.
    - Also, Can be shared, copied, processed and stored.
    - So, Easy to steal or copy
    - Drawn from the artifact of some type as a principle, procedure, process, concepts.

A variety of ways of representing knowledge have been exploited in AI programs.

There are two different kinds of entities, we are dealing with.

1. Facts: Truth in some relevant world. Things we want to represent.
2. Also, Representation of facts in some chosen formalism. Things we will actually be able to manipulate.

These entities structured at two levels:

1. The knowledge level, at which facts described.
2. Moreover, The symbol level, at which representation of objects defined in terms of symbols that can manipulate by programs

Framework of Knowledge Representation

- The computer requires a well-defined problem description to process and provide a well-defined acceptable solution.

- Moreover, To collect fragments of knowledge we need first to formulate a description in our spoken language and then represent it in formal language so that computer can understand.
- Also, The computer can then use an algorithm to compute an answer.

So, This process illustrated as,

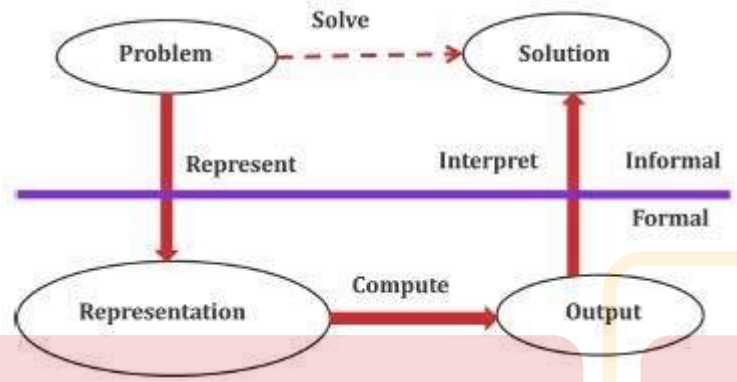


Fig: Knowledge Representation Framework

The steps are:

- The informal formalism of the problem takes place first.
- It then represented formally and the computer produces an output.
- This output can then represented in an informally described solution that user understands or checks for consistency.

The Problem solving requires,

- Formal knowledge representation, and
- Moreover, Conversion of informal knowledge to a formal knowledge that is the conversion of implicit knowledge to explicit knowledge.

Mapping between Facts and Representation

- Knowledge is a collection of facts from some domain.
- Also, We need a representation of "facts" that can manipulate by a program.
- Moreover, Normal English is insufficient, too hard currently for a computer program to draw inferences in natural languages.
- Thus some symbolic representation is necessary.

A good knowledge representation enables fast and accurate access to knowledge and understanding of the content.

A knowledge representation system should have following properties.

1. Representational Adequacy
    - The ability to represent all kinds of knowledge that are needed in that domain.
2. Inferential Adequacy
    - Also, The ability to manipulate the representational structures to derive new structures corresponding to new knowledge inferred from old.
3. Inferential Efficiency
    - The ability to incorporate additional information into the knowledge structure that can be used to focus the attention of the inference mechanisms in the most promising direction.
4. Acquisitional Efficiency
    - Moreover, The ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Knowledge Representation Schemes

Relational Knowledge

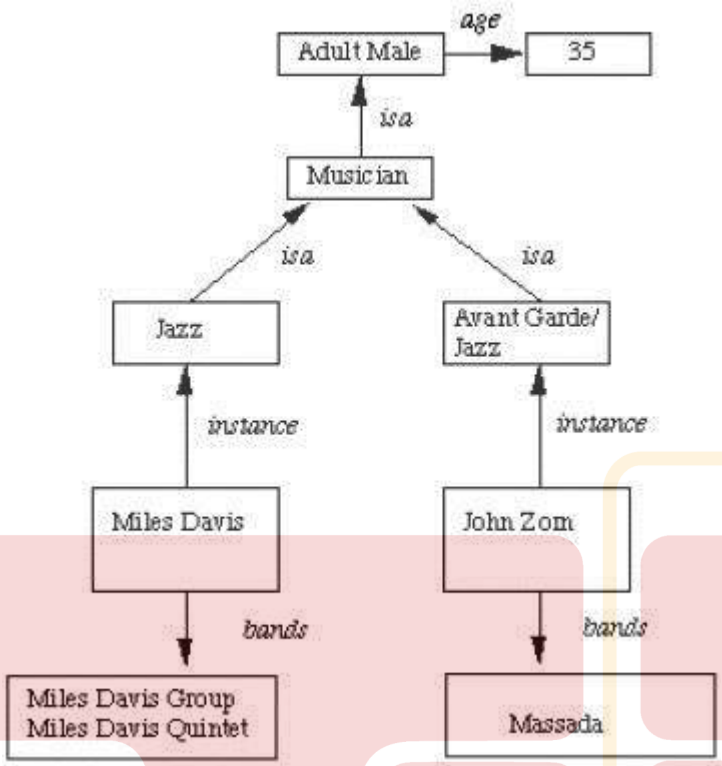
- The simplest way to represent declarative facts is a set of relations of the same sort used in the database system.
- Provides a framework to compare two objects based on equivalent attributes. o Any instance in which two different objects are compared is a relational type of knowledge.
- The table below shows a simple way to store facts.
    - Also, The facts about a set of objects are put systematically in columns.
    - This representation provides little opportunity for inference.

Player	Height	Weight	Bats - Throws
Aaron	6-0	180	Right - Right
Mays	5-10	170	Right - Right
Ruth	6-2	215	Left - Left
Williams	6-3	205	Left - Right

- Given the facts, it is not possible to answer a simple question such as: "Who is the heaviest player?"
- Also, But if a procedure for finding the heaviest player is provided, then these facts will enable that procedure to compute an answer.
- Moreover, We can ask things like who "bats – left" and "throws – right".

Inheritable Knowledge

- Here the knowledge elements inherit attributes from their parents.
- The knowledge embodied in the design hierarchies found in the functional, physical and process domains.
- Within the hierarchy, elements inherit attributes from their parents, but in many cases, not all attributes of the parent elements prescribed to the child elements.
- Also, The inheritance is a powerful form of inference, but not adequate.
- Moreover, The basic KR (Knowledge Representation) needs to augment with inference mechanism.
- Property inheritance: The objects or elements of specific classes inherit attributes and values from more general classes.
- So, The classes organized in a generalized hierarchy.



- Boxed nodes — objects and values of attributes of objects.
- Arrows — the point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

The steps to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of an instance, if none fail
4. Also, Go to that node and find a value for the attribute and then report it
5. Otherwise, search through using is until a value is found for the attribute.

Inferential Knowledge

- This knowledge generates new information from the given information.
- This new information does not require further data gathering form source but does require analysis of the given information to generate new knowledge.
- Example: given a set of relations and values, one may infer other values or relations. A predicate logic (a mathematical deduction) used to infer from a set of attributes. Moreover, Inference through predicate logic uses a set of logical operations to relate individual data.
- Represent knowledge as formal logic: All dogs have tails $\forall x: dog(x) \rightarrow hastail(x)$
- Advantages:
    - A set of strict rules.
    - Can use to derive more facts.
    - Also, Truths of new statements can be verified.
    - Guaranteed correctness.
- So, Many inference procedures available to implement standard rules of logic popular in AI systems. e.g Automated theorem proving.

Procedural Knowledge

- A representation in which the control information, to use the knowledge, embedded in the knowledge itself. For example, computer programs, directions, and recipes; these indicate specific use or implementation;
- Moreover, Knowledge encoded in some procedures, small programs that know how to do specific things, how to proceed.
- Advantages:
    - Heuristic or domain-specific knowledge can represent.
    - Moreover, Extended logical inferences, such as default reasoning facilitated.
    - Also, Side effects of actions may model. Some rules may become false in time. Keeping track of this in large systems may be tricky.
- Disadvantages:
    - Completeness — not all cases may represent.
    - Consistency — not all deductions may be correct. e.g If we know that Fred is a bird we might deduce that Fred can fly. Later we might discover that Fred is an emu.
    - Modularity sacrificed. Changes in knowledge base might have far-reaching effects.
    - Cumbersome control information.

USING PREDICATE LOGIC

Representation of Simple Facts in Logic

Propositional logic is useful because it is simple to deal with and a decision procedure for it exists.

Also, In order to draw conclusions, facts are represented in a more convenient way as,

1. Marcus is a man.
    - man(Marcus)
2. Plato is a man.
    - man(Plato)
3. All men are mortal.
    - mortal(men)

But propositional logic fails to capture the relationship between an individual being a man and that individual being a mortal.

- How can these sentences be represented so that we can infer the third sentence from the first two?
- Also, Propositional logic commits only to the existence of facts that may or may not be the case in the world being represented.
- Moreover, It has a simple syntax and simple semantics. It suffices to illustrate the process of inference.
- Propositional logic quickly becomes impractical, even for very small worlds.

Predicate logic

First-order Predicate logic (FOPL) models the world in terms of

- Objects, which are things with individual identities
- Properties of objects that distinguish them from other objects
- Relations that hold among sets of objects

- Functions, which are a subset of relations where there is only one "value" for any given "input"

First-order Predicate logic (FOPL) provides

- Constants: a, b, dog33. Name a specific object.
- Variables: X, Y. Refer to an object without naming it.
- Functions: Mapping from objects to objects.
- Terms: Refer to objects
- Atomic Sentences: in(dad-of(X), food6) Can be true or false, Correspond to propositional symbols P, Q.

A well-formed formula (*wff*) is a sentence containing no "free" variables. So, That is, all variables are "bound" by universal or existential quantifiers.

(∀x)P(x, y) has x bound as a universally quantified variable, but y is free.

Quantifiers

Universal quantification

- (∀x)P(x) means that P holds for all values of x in the domain associated with that variable
- E.g., (∀x) dolphin(x) → mammal(x)

Existential quantification

- (∃ x)P(x) means that P holds for some value of x in the domain associated with that variable
- E.g., (∃ x) mammal(x) ∧ lays-eggs(x)

Also, Consider the following example that shows the use of predicate logic as a way of representing knowledge.

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. Also, All Pompeians were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of well-formed formulas (*wffs*) as follows:

1. Marcus was a man.
    - man(Marcus)
2. Marcus was a Pompeian.
    - Pompeian(Marcus)
3. All Pompeians were Romans.
    - ∀x: Pompeian(x) → Roman(x)
4. Caesar was a ruler.
    - ruler(Caesar)
5. All Pompeians were either loyal to Caesar or hated him.
    - inclusive-or
    - ∀x: Roman(x) → loyalto(x, Caesar) ∨ hate(x, Caesar)
    - exclusive-or
    - ∀x: Roman(x) → (loyalto(x, Caesar) ∧¬ hate(x, Caesar)) ∨
    - (¬loyalto(x, Caesar) ∧ hate(x, Caesar))

6. Everyone is loyal to someone.
    - ∀x: ∃y: loyalto(x, y)
7. People only try to assassinate rulers they are not loyal to.
    - ∀x: ∀y: person(x) ∧ ruler(y) ∧ tryassassinate(x, y)
    - →¬loyalto(x, y)
8. Marcus tried to assassinate Caesar.
    - tryassassinate(Marcus, Caesar)

Now suppose if we want to use these statements to answer the question: ***Was Marcus loyal to Caesar?***

Also, Now let's try to produce a formal proof, reasoning backward from the desired goal: ¬ Ioyalto(Marcus, Caesar)

In order to prove the goal, we need to use the rules of inference to transform it into another goal (or possibly a set of goals) that can, in turn, transformed, and so on, until there are no unsatisfied goals remaining.

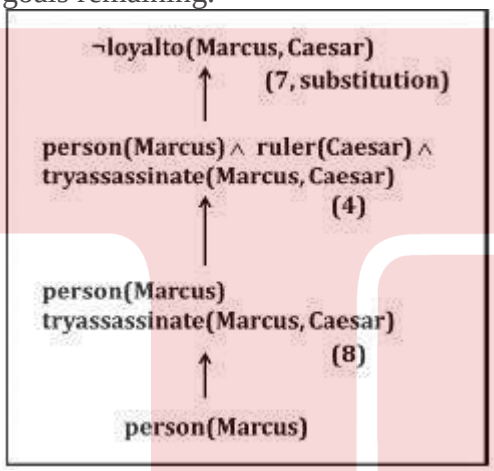


Figure: An attempt to prove ¬loyalto(Marcus, Caesar).

- The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. Also, We need to add the representation of another fact to our system, namely***: ∀man(x) → person(x)***
- Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar.
- Moreover, From this simple example, we see that three important issues must be addressed in the process of converting English sentences into logical statements and then using those statements to deduce new ones:
    1. Many English sentences are ambiguous (for example, 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
    2. Also, There is often a choice of how to represent the knowledge. Simple representations are desirable, but they may exclude certain kinds of reasoning.
    3. Similalry, Even in very simple situations, a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand. In order to be able to use a set of statements effectively. Moreover, It is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

Representing Instance and ISA Relationships

- Specific attributes **instance** and **isa** play an important role particularly in a useful form of reasoning called property inheritance.
- The predicates instance and isa explicitly captured the relationships they used to express, namely class membership and class inclusion.
- 4.2 shows the first five sentences of the last section represented in logic in three different ways.
- The first part of the figure contains the representations we have already discussed. In these representations, class membership represented with unary predicates (such as Roman), each of which corresponds to a class.
- Asserting that P(x) is true is equivalent to asserting that x is an instance (or element) of P.
- The second part of the figure contains representations that use the ***instance*** predicate explicitly.

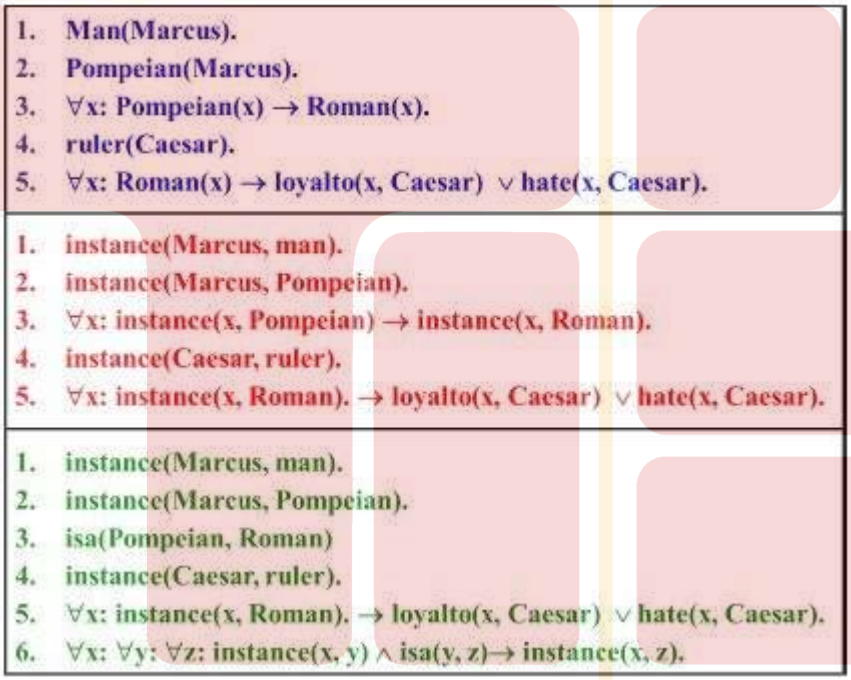


Figure: Three ways of representing class membership: ISA Relationships

- The predicate ***instance*** is a binary one, whose first argument is an object and whose second argument is a class to which the object belongs.
- But these representations do not use an explicit ***isa*** predicate.
- Instead, subclass relationships, such as that between Pompeians and Romans, described as shown in sentence 3.
- The implication rule states that if an object is an instance of the subclass Pompeian then it is an instance of the superclass Roman.
- Note that this rule is equivalent to the standard set-theoretic definition of the subclass-superclass relationship.
- The third part contains representations that use both the ***instance*** and ***isa*** predicates explicitly.
- The use of the ***isa*** predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

Computable Functions and Predicates

- To express simple facts, such as the following greater-than and less-than relationships: gt(1,O) It(0,1) gt(2,1) It(1,2) gt(3,2) It(2,3)
- It is often also useful to have computable functions as well as computable predicates. Thus we might want to be able to evaluate the truth of gt(2 + 3,1)
- To do so requires that we first compute the value of the plus function given the arguments 2 and 3, and then send the arguments 5 and 1 to gt.

Consider the following set of facts, again involving Marcus:

1) Marcus was a man.
   man(Marcus)
2) Marcus was a Pompeian.
   Pompeian(Marcus)
3) Marcus was born in 40 A.D.
   born(Marcus, 40)
4) All men are mortal.
   x: man(x) → mortal(x)
5) All Pompeians died when the volcano erupted in 79 A.D.
   erupted(volcano, 79) ∧ ∀ x : [Pompeian(x) → died(x, 79)]
6) No mortal lives longer than 150 years.
   x: t1: At2: *mortal(x) born(x, t1) gt(t2 – t1,150) → died(x, t2)*
7) It is now 1991.
   now = 1991

So, Above example shows how these ideas of computable functions and predicates can be useful. It also makes use of the notion of equality and allows equal objects to be substituted for each other whenever it appears helpful to do so during a proof.

- So, Now suppose we want to answer the question "Is Marcus alive?"
- The statements suggested here, there may be two ways of deducing an answer.
- Either we can show that Marcus is dead because he was killed by the volcano or we can show that he must be dead because he would otherwise be more than 150 years old, which we know is not possible.
- Also, As soon as we attempt to follow either of those paths rigorously, however, we discover, just as we did in the last example, that we need some additional knowledge. For example, our statements talk about dying, but they say nothing that relates to being alive, which is what the question is asking.

So we add the following facts:

8) Alive means not dead.
   x: t: [alive(x, t) → ¬ dead(x, t)] [¬ dead(x, t) → alive(x, t)]
9) If someone dies, then he is dead at all later times.
   x: t1: At2*: died(x, t1) gt(t2, t1) → dead(x, t2)*

So, Now let's attempt to answer the question "Is Marcus alive?" by proving: ¬ ***alive(Marcus, now)***

Resolution

Propositional Resolution

1. Convert all the propositions of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and ¬ *L* such that one of the parent clauses contains *L* and the other contains ¬*L*, then select one such pair and eliminate both *L* and ¬ *L* from the resolvent.
    3. If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of classes available to the procedure.

The Unification Algorithm

- In propositional logic, it is easy to determine that two literals cannot both be true at the same time.
- Simply look for L and ¬L in predicate logic, this matching process is more complicated since the arguments of the predicates must be considered.
- For example, man(John) and ¬man(John) is a contradiction, while the man(John) and ¬man(Spot) is not.
- Thus, in order to determine contradictions, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical.
- There is a straightforward recursive procedure, called the unification algorithm, that does it.

Algorithm: Unify(L1, L2)

1. If L1 or L2 are both variables or constants, then:
    1. If L1 and L2 are identical, then return NIL.
    2. Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
    3. Also, Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL}, else return (L1/L2). d. Else return {FAIL}.
2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.
3. If LI and L2 have a different number of arguments, then return {FAIL}.
4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)
5. For I ← 1 to the number of arguments in L1 :
    1. Call Unify with the i^{th} argument of L1 and the i^{th} argument of L2, putting the result in S.
    2. If S contains FAIL then return {FAIL}.
    3. If S is not equal to NIL then:
        2. Apply S to the remainder of both L1 and L2.
        3. SUBST: = APPEND(S, SUBST).
6. Return SUBST.

Resolution in Predicate Logic

We can now state the resolution algorithm for predicate logic as follows, assuming a set of given statements F and a statement to be proved P:

Algorithm: Resolution

1. Convert all the statements of F to clause form.
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until a contradiction found, no progress can make, or a predetermined amount of effort has expanded.
    1. Select two clauses. Call these the parent clauses.
    2. Resolve them together. The resolvent will the disjunction of all the literals of both parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and ¬T2 such that one of the parent clauses contains T2 and the other contains T1 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 Complementary literals. Use the substitution produced by the unification to create the resolvent. If there is more than one pair of complementary literals, only one pair should omit from the resolvent.
    3. If the resolvent is an empty clause, then a contradiction has found. Moreover, If it is not, then add it to the set of classes available to the procedure.

Resolution Procedure

- Resolution is a procedure, which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form.
- Resolution produces proofs by refutation.
- In other words, ***to prove a statement (i.e., to show that it is valid), resolution attempts to show that the negation of the statement produces a contradiction with the known statements (i.e., that it is unsatisfiable).***
- The resolution procedure is a simple iterative process: at each step, two clauses, called the parent clauses, are compared (resolved), resulting in a new clause that has inferred from them. The new clause represents ways that the two parent clauses interact with each other. Suppose that there are two clauses in the system:

winter **V** ***summer***

¬ ***winter*** **V** ***cold***

- Now we observe that precisely one of winter and ¬ winter will be true at any point.
- If winter is true, then cold must be true to guarantee the truth of the second clause. If ¬ winter is true, then summer must be true to guarantee the truth of the first clause.
- Thus we see that from these two clauses we can deduce ***summer V cold***
- This is the deduction that the resolution procedure will make.
- Resolution operates by taking two clauses that each contains the same literal, in this example, ***winter***.
- Moreover, The literal must occur in the positive form in one clause and in negative form in the other. The resolvent obtained by combining all of the literals of the two parent clauses except the ones that cancel.
- If the clause that produced is the empty clause, then a contradiction has found.

For example, the two clauses

winter

¬ winter

will produce the empty clause.

Natural Deduction Using Rules

Testing whether a proposition is a tautology by testing every possible truth assignment is expensive—there are exponentially many. We need a **deductive system**, which will allow us to construct proofs of tautologies in a step-by-step fashion.

The system we will use is known as **natural deduction**. The system consists of a set of **rules of inference** for deriving consequences from premises. One builds a proof tree whose root is the proposition to be proved and whose leaves are the initial assumptions or axioms (for proof trees, we usually draw the root at the bottom and the leaves at the top).

For example, one rule of our system is known as **modus ponens**. Intuitively, this says that if we know P is true, and we know that P implies Q, then we can conclude Q.

$$\frac{P \qquad P \Rightarrow Q}{Q} \text{ (modus ponens)}$$

The propositions above the line are called **premises**; the proposition below the line is the **conclusion**. Both the premises and the conclusion may contain metavariables (in this case, P and Q) representing arbitrary propositions. When an inference rule is used as part of a proof, the metavariables are replaced in a consistent way with the appropriate kind of object (in this case, propositions).

Most rules come in one of two flavors: **introduction** or **elimination** rules. Introduction rules introduce the use of a logical operator, and elimination rules eliminate it. Modus ponens is an elimination rule for ⇒. On the right-hand side of a rule, we often write the name of the rule. This is helpful when reading proofs. In this case, we have written (modus ponens). We could also have written (⇒-elim) to indicate that this is the elimination rule for ⇒.

Rules for Conjunction

Conjunction (∧) has an introduction rule and two elimination rules:

$$\frac{P \qquad Q}{P \wedge Q} \text{ (∧-intro)} \qquad \frac{P \wedge Q}{P} \text{ (∧-elim-left)} \qquad \frac{P \wedge Q}{Q} \text{ (∧-elim-right)}$$

Rule for T

The simplest introduction rule is the one for T. It is called "unit". Because it has no premises, this rule is an **axiom**: something that can start a proof.

$$\frac{}{T} \text{ (unit)}$$

Rules for Implication

In natural deduction, to prove an implication of the form P ⇒ Q, we assume P, then reason under that assumption to try to derive Q. If we are successful, then we can conclude that P ⇒ Q.

In a proof, we are always allowed to introduce a new assumption P, then reason under that assumption. We must give the assumption a name; we have used the name x in the example below. Each distinct assumption must have a different name.

$$\frac{}{[x : P]} \text{ (assum)}$$

Because it has no premises, this rule can also start a proof. It can be used as if the proposition P were proved. The name of the assumption is also indicated here.
However, you do not get to make assumptions for free! To get a complete proof, all assumptions must be eventually *discharged*. This is done in the implication introduction rule. This rule introduces an implication $P \Rightarrow Q$ by discharging a prior assumption [x : P]. Intuitively, if Q can be proved under the assumption P, then the implication $P \Rightarrow Q$ holds without any assumptions. We write x in the rule name to show which assumption is discharged. This rule and modus ponens are the introduction and elimination rules for implications.

$$\frac{\begin{array}{c}[x : P] \\ \vdots \\ Q\end{array}}{P \Rightarrow Q}\ (\Rightarrow\text{-intro/x}) \qquad \frac{P \qquad P \Rightarrow Q}{Q}\ (\Rightarrow\text{-elim, modus ponens})$$

A proof is valid only if every assumption is eventually discharged. This must happen in the proof tree below the assumption. The same assumption can be used more than once.

Rules for Disjunction

$$\frac{P}{P \vee Q}\ (\vee\text{-intro-left}) \qquad \frac{Q}{P \vee Q}\ (\vee\text{-intro-right}) \qquad \frac{P \vee Q \qquad P \Rightarrow R \qquad Q \Rightarrow R}{R}\ (\vee\text{-elim})$$

Rules for Negation

A negation $\neg P$ can be considered an abbreviation for $P \Rightarrow \bot$:

$$\frac{P \Rightarrow \bot}{\neg P}\ (\neg\text{-intro}) \qquad \frac{\neg P}{P \Rightarrow \bot}\ (\neg\text{-elim})$$

Rules for Falsity

$$\frac{\begin{array}{c}[x : \neg P] \\ \vdots \\ \bot\end{array}}{P}\ (\text{reductio ad absurdum, RAA/x}) \qquad \frac{\bot}{P}\ (\text{ex falso quodlibet, EFQ})$$

Reductio ad absurdum (RAA) is an interesting rule. It embodies proofs by contradiction. It says that if by assuming that P is false we can derive a contradiction, then P must be true. The assumption x is discharged in the application of this rule. This rule is present in classical logic but not in **intuitionistic** (constructive) logic. In intuitionistic logic, a proposition is not considered true simply because its negation is false.

Excluded Middle

Another classical tautology that is not intuitionistically valid is the **the law of the excluded middle**, $P \vee \neg P$. We will take it as an axiom in our system. The Latin name for this rule is *tertium non datur*, but we will call it *magic*.

$$\frac{}{P \vee \neg P}\ (\text{magic})$$

Proofs

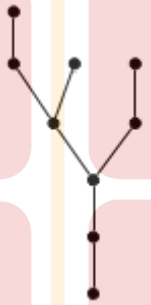
A proof of proposition P in natural deduction starts from axioms and assumptions and derives P with all assumptions discharged. Every step in the proof is an instance of an inference rule with metavariables substituted consistently with expressions of the appropriate syntactic class.

Example

For example, here is a proof of the proposition (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[y : A \wedge B]}\,(A)}{A}\,(\wedge E) \quad \dfrac{}{[x : A \Rightarrow B \Rightarrow C]}\,(A)}{B \Rightarrow C}\,(\Rightarrow E) \quad \dfrac{}{[y : A \wedge B]}\,(A)}{B}\,(\Rightarrow E)}{C}}{A \wedge B \Rightarrow C}\,(\Rightarrow I,y)}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)}\,(\Rightarrow I,x)$$

The final step in the proof is to derive (A ⇒ B ⇒ C) ⇒ (A ∧ B ⇒ C) from (A ∧ B ⇒ C), which is done using the rule (⇒-intro), discharging the assumption [x : A ⇒ B ⇒ C]. To see how this rule generates the proof step, substitute for the metavariables P, Q, x in the rule as follows: P = (A ⇒ B ⇒ C), Q = (A ∧ B ⇒ C), and x = x. The immediately previous step uses the same rule, but with a different substitution: P = A ∧ B, Q = C, x = y.
The proof tree for this example has the following form, with the proved proposition at the root and axioms and assumptions at the leaves.



A proposition that has a complete proof in a deductive system is called a **theorem** of that system.
Soundness and Completeness
A measure of a deductive system's power is whether it is powerful enough to prove all true statements. A deductive system is said to be **complete** if all true statements are theorems (have proofs in the system). For propositional logic and natural deduction, this means that all tautologies must have natural deduction proofs. Conversely, a deductive system is called **sound** if all theorems are true. The proof rules we have given above are in fact sound and complete for propositional logic: every theorem is a tautology, and every tautology is a theorem. Finding a proof for a given tautology can be difficult. But once the proof is found, checking that it is indeed a proof is completely mechanical, requiring no intelligence or insight whatsoever. It is therefore a very strong argument that the thing proved is in fact true.
We can also make writing proofs less tedious by adding more rules that provide reasoning shortcuts. These rules are sound if there is a way to convert a proof using them into a proof using the original rules. Such added rules are called **admissible**.

Procedural versus Declarative Knowledge

We have discussed various search techniques in previous units. Now we would consider a set of rules that represent,

1. Knowledge about relationships in the world and
2. Knowledge about how to solve the problem using the content of the rules.

Procedural vs Declarative Knowledge
Procedural Knowledge

- A representation in which the control information that is necessary to use the knowledge is embedded in the knowledge itself for e.g. computer programs, directions, and recipes; these indicate specific use or implementation;
- The real difference between declarative and procedural views of knowledge lies in where control information reside.

For example, consider the following
Man (Marcus)
Man (Caesar)
Person (Cleopatra)
$\forall x$: *Man(x)* $\rightarrow$ *Person(x)*
Now, try to answer the question. ?Person(y)
The knowledge base justifies any of the following answers.
Y=Marcus
Y=Caesar
Y=Cleopatra

- We get more than one value that satisfies the predicate.
- If only one value needed, then the answer to the question will depend on the order in which the assertions examined during the search for a response.
- If the assertions declarative then they do not themselves say anything about how they will be examined. In case of procedural representation, they say how they will examine.

Declarative Knowledge

- A statement in which knowledge specified, but the use to which that knowledge is to be put is not given.
- For example, laws, people's name; these are the facts which can stand alone, not dependent on other knowledge;
- So to use declarative representation, we must have a program that explains what is to do with the knowledge and how.
- For example, a set of logical assertions can combine with a resolution theorem prover to give a complete program for solving problems but in some cases, the logical assertions can view as a program rather than data to a program.
- Hence the implication statements define the legitimate reasoning paths and automatic assertions provide the starting points of those paths.
- These paths define the execution paths which is similar to the 'if then else "in traditional programming.
- So logical assertions can view as a procedural representation of knowledge.

Logic Programming – Representing Knowledge Using Rules

- Logic programming is a programming paradigm in which logical assertions viewed as programs.
- These are several logic programming systems, PROLOG is one of them.
- ***A PROLOG program consists of several logical assertions where each is a horn clause i.e. a clause with at most one positive literal.***
- Ex : P, P V Q, P $\rightarrow$ Q
- The facts are represented on Horn Clause for two reasons.
    1. Because of a uniform representation, a simple and efficient interpreter can write.
    2. The logic of Horn Clause decidable.

- Also, The first two differences are the fact that PROLOG programs are actually sets of Horn clause that have been transformed as follows:-
    1. If the Horn Clause contains no negative literal then leave it as it is.
    2. Also, Otherwise rewrite the Horn clauses as an implication, combining all of the negative literals into the antecedent of the implications and the single positive literal into the consequent.
- Moreover, This procedure causes a clause which originally consisted of a disjunction of literals (one of them was positive) to be transformed into a single implication whose antecedent is a conjunction universally quantified.
- But when we apply this transformation, any variables that occurred in negative literals and so now occur in the antecedent become existentially quantified, while the variables in the consequent are still universally quantified.

For example the PROLOG clause P(x): – Q(x, y) is equal to logical expression ∀x: ∃y: Q (x, y) → P(x).

- The difference between the logic and PROLOG representation is that the PROLOG interpretation has a fixed control strategy. And so, the assertions in the PROLOG program define a particular search path to answer any question.
- But, the logical assertions define only the set of answers but not about how to choose among those answers if there is more than one.

Consider the following example:

1. Logical representation

$\forall x : pet(x) \square small(x) \rightarrow apartmentpet(x)$
$\forall x : cat(x) \square dog(x) \rightarrow pet(x)$
$\forall x : poodle(x) \rightarrow dog(x) \square small(x)$
$poodle(fluffy)$

2. Prolog representation

apartmentpet (x) : pet(x), small (x)
pet (x): cat (x)
pet (x): dog(x)
dog(x): poodle (x)
small (x): poodle(x)
poodle (fluffy)

Forward versus Backward Reasoning

Forward versus Backward Reasoning
A search procedure must find a path between initial and goal states.
There are two directions in which a search process could proceed.
The two types of search are:

1. Forward search which starts from the start state
2. Backward search that starts from the goal state

The production system views the forward and backward as symmetric processes.
Consider a game of playing 8 puzzles. The rules defined are
Square 1 empty and square 2 contains tile n. →

- *Also, Square 2 empty and square 1 contains the tile n.*

Square 1 empty Square 4 contains tile n. →

- *Also, Square 4 empty and Square 1 contains tile n.*

We can solve the problem in 2 ways:

1. Reason forward from the initial state
   - Step 1. Begin building a tree of move sequences by starting with the initial configuration at the root of the tree.
   - Step 2. Generate the next level of the tree by finding all rules ***whose left-hand side matches*** against the root node. The right-hand side is used to create new configurations.
   - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose left-hand side match.
2. Reasoning backward from the goal states:
   - Step 1. Begin building a tree of move sequences by starting with the goal node configuration at the root of the tree.
   - Step 2. Generate the next level of the tree by finding all rules ***whose right-hand side matches*** against the root node. The left-hand side used to create new configurations.
   - Step 3. Generate the next level by considering the nodes in the previous level and applying it to all rules whose right-hand side match.
   - So, The same rules can use in both cases.
   - Also, In forwarding reasoning, the left-hand sides of the rules matched against the current state and right sides used to generate the new state.
   - Moreover, In backward reasoning, the right-hand sides of the rules matched against the current state and left sides are used to generate the new state.

There are four factors influencing the type of reasoning. They are,

1. Are there more possible start or goal state? We move from smaller set of sets to the length.
2. In what direction is the branching factor greater? We proceed in the direction with the lower branching factor.
3. Will the program be asked to justify its reasoning process to a user? If, so then it is selected since it is very close to the way in which the user thinks.
4. What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new factor, the forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Example 1 of Forward versus Backward Reasoning

- It is easier to drive from an unfamiliar place from home, rather than from home to an unfamiliar place. Also, If you consider a home as starting place an unfamiliar place as a goal then we have to backtrack from unfamiliar place to home.

Example 2 of Forward versus Backward Reasoning

- Consider a problem of symbolic integration. Moreover, The problem space is a set of formulas, which contains integral expressions. Here START is equal to the given formula with some integrals. GOAL is equivalent to the expression of the formula without any integral. Here we start from the formula with some integrals and proceed to an integral free expression rather than starting from an integral free expression.

Example 3 of Forward versus Backward Reasoning

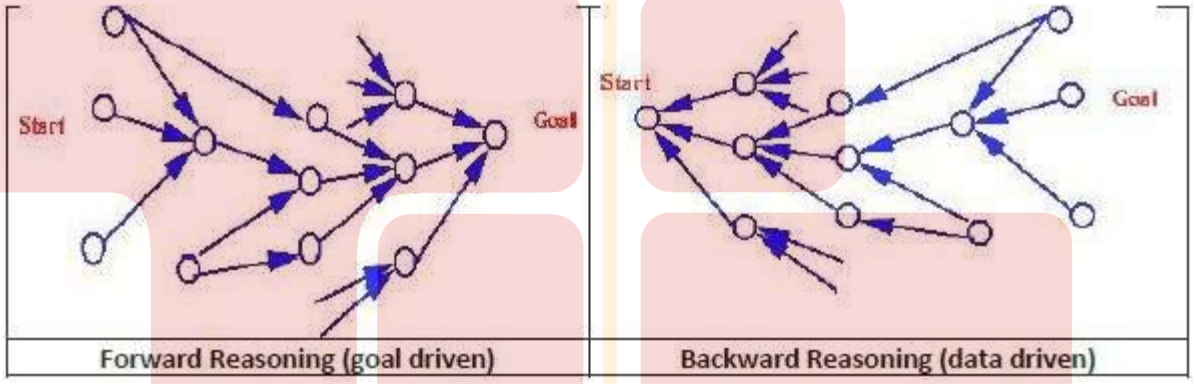
- The third factor is nothing but deciding whether the reasoning process can justify its reasoning. If it justifies then it can apply. For example, doctors are usually unwilling to accept any advice from diagnostics process because it cannot explain its reasoning.

Example 4 of Forward versus Backward Reasoning

- Prolog is an example of backward chaining rule system. In Prolog rules restricted to Horn clauses. This allows for rapid indexing because all the rules for deducing a given fact share the same rule head. Rules matched with unification procedure. Unification tries to find a set of bindings for variables to equate a sub-goal with the head of some rule. Rules in the Prolog program matched in the order in which they appear.

Combining Forward and Backward Reasoning

- Instead of searching either forward or backward, you can search both simultaneously.
- Also, That is, start forward from a starting state and backward from a goal state simultaneously until the paths meet.
- This strategy called Bi-directional search. The following figure shows the reason for a Bidirectional search to be ineffective.



Forward versus Backward Reasoning

- Also, The two searches may pass each other resulting in more work.
- Based on the form of the rules one can decide whether the same rules can apply to both forward and backward reasoning.
- Moreover, If left-hand side and right of the rule contain pure assertions then the rule can reverse.
- And so the same rule can apply to both types of reasoning.
- If the right side of the rule contains an arbitrary procedure then the rule cannot reverse.
- So, In this case, while writing the rule the commitment to a direction of reasoning must make.

Symbolic Reasoning Under Uncertainty

Symbolic Reasoning

- The reasoning is the act of deriving a conclusion from certain properties using a given methodology.
- The reasoning is a process of thinking; reasoning is logically arguing; reasoning is drawing the inference.
- *When a system is required to do something, that it has not been explicitly told how to do, it must reason. It must figure out what it needs to know from what it already knows.*