

Introduction to Artificial Intelligence

Artificial Intelligence

Artificial intelligence is **the simulation of human intelligence processes by machines, especially computer systems**. Specific applications of AI include expert systems, natural language processing, speech recognition and machine vision.

AI will provide human-like interactions with software and offer decision support for specific tasks, but it's not a replacement for humans – and won't be anytime soon.

Artificial Intelligence is composed of two words **Artificial** and **Intelligence**, where Artificial defines "*man-made*," and intelligence defines "*thinking power*", hence AI means "*a man-made thinking power*."

"It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions."

Why Artificial Intelligence?

Before Learning about Artificial Intelligence, we should know that what is the importance of AI and why should we learn it. Following are some main reasons to learn about AI:

- With the help of AI, you can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.
- With the help of AI, you can create your personal virtual Assistant, such as Cortana, Google Assistant, Siri, etc.
- With the help of AI, you can build such Robots which can work in an environment where survival of humans can be at risk.
- AI opens a path for other new technologies, new devices, and new Opportunities.

Need for Artificial Intelligence

1. To create expert systems that exhibit intelligent behavior with the capability to learn, demonstrate, explain, and advise its users.
2. Helping machines find solutions to complex problems like humans do and applying them as algorithms in a computer-friendly manner.

Goals of Artificial Intelligence

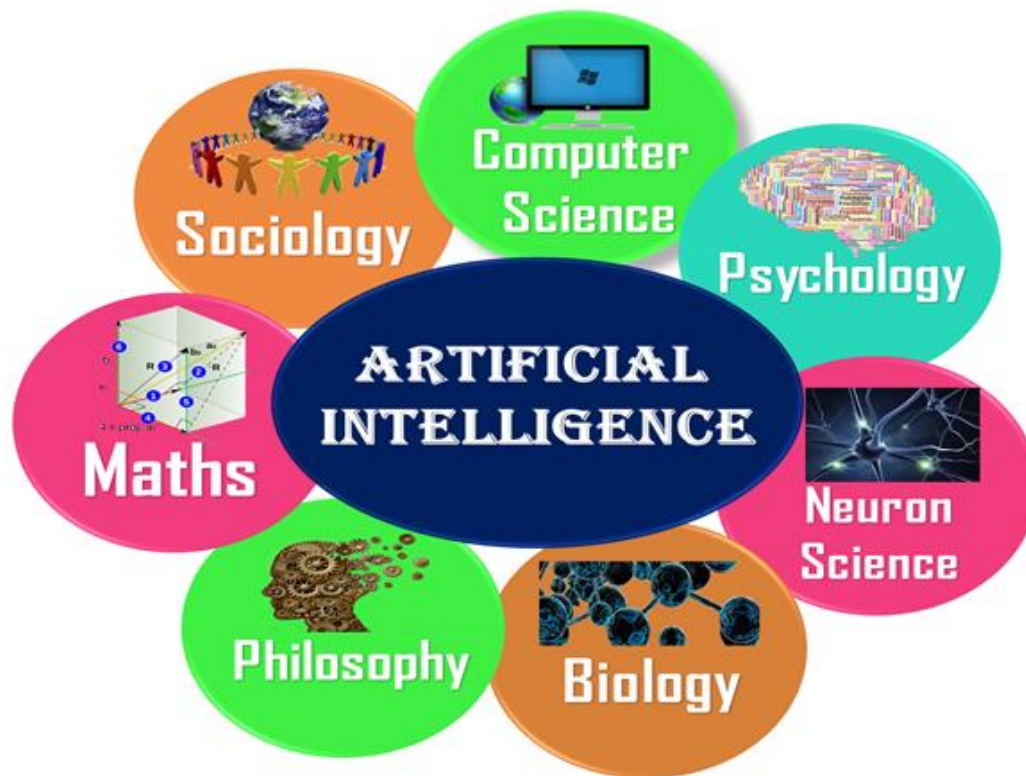
1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action
4. Building a machine which can perform tasks that requires human intelligence such as:
 - Proving a theorem
 - Playing chess
 - Plan some surgical operation
 - Driving a car in traffic
5. Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise to its user.

What Comprises to Artificial Intelligence?

Artificial Intelligence is not just a part of computer science even it's so vast and requires lots of other factors which can contribute to it. To create the AI first we should know that how intelligence is composed, so the Intelligence is an intangible part of our brain which is a combination of **Reasoning, learning, problem-solving perception, language understanding, etc.**

To achieve the above factors for a machine or software Artificial Intelligence requires the following discipline:

- Mathematics
- Biology
- Psychology
- Sociology
- Computer Science
- Neurons Study
- Statistics



Subfields of Artificial Intelligence

Here, are some important subfields of Artificial Intelligence:

[Machine Learning \(Links to an external site.\)](#): Machine learning is the art of studying algorithms that learn from examples and experiences. Machine learning is based on the idea that some patterns in the data were identified and used for future predictions. The difference from hardcoding rules is that the machine learns to find such rules.

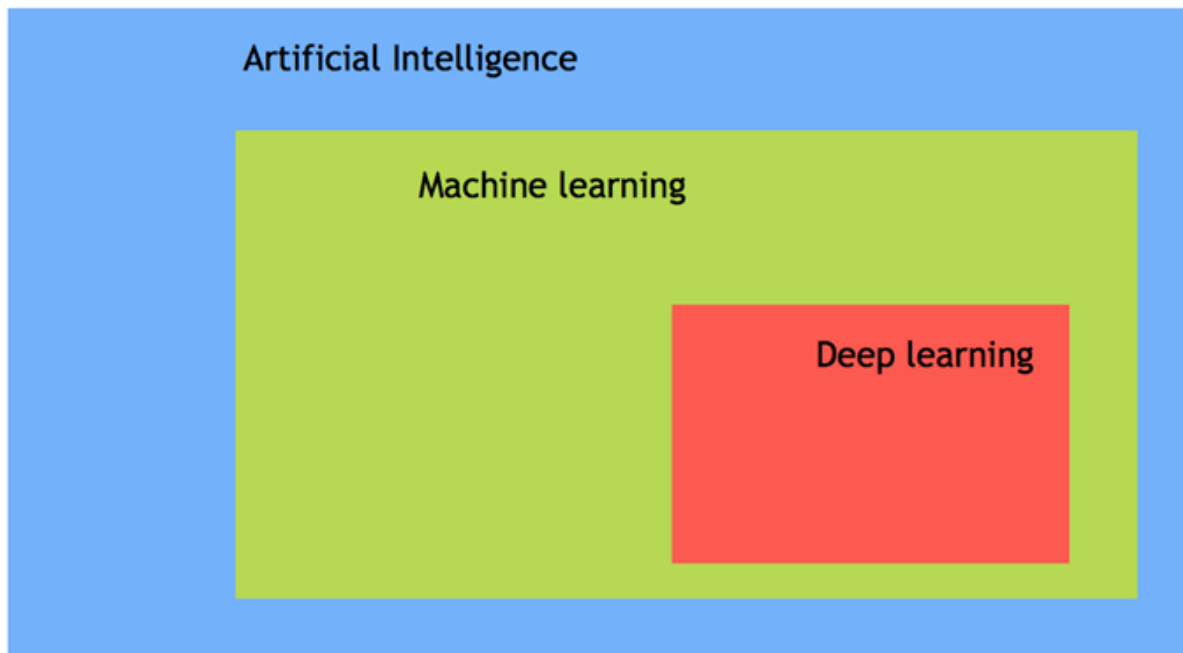
[Deep Learning \(Links to an external site.\)](#): Deep learning is a sub-field of machine learning. Deep learning does not mean the machine learns more in-depth knowledge; it uses different layers to learn from the data. The depth of the model is represented by the number of layers in the model. For instance, the Google LeNet model for image recognition counts 22 layers.

[Natural Language Processing \(Links to an external site.\)](#): A neural network is a group of connected I/O units where each connection has a weight associated with its computer programs. It helps you to build predictive models from large databases. This model builds upon the human nervous system. You can use this model to conduct image understanding, human learning, computer speech, etc.

[Expert Systems \(Links to an external site.\)](#): An expert system is an interactive and reliable computer-based decision-making system that uses facts and heuristics to solve complex decision-making problems. It is also considered at the highest level of

human intelligence. The main goal of an expert system is to solve the most complex issues in a specific domain.

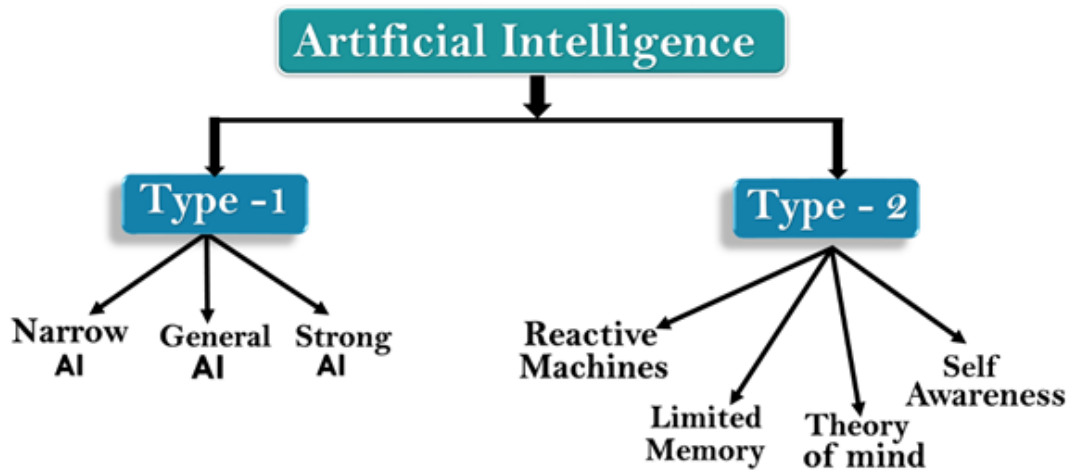
Fuzzy Logic (Links to an external site.): Fuzzy Logic is defined as a many-valued logic form that may have truth values of variables in any real number between 0 and 1. It is the handle concept of partial truth. In real life, we may encounter a situation where we can't decide whether the statement is true or false.



Types, Advantages and disadvantages of AI

Types of Artificial Intelligence:

Artificial Intelligence can be divided in various types, there are mainly two types of main categorization which are based on capabilities and based on functionality of AI. Following is flow diagram which explain the types of AI.



AI type-1: Based on Capabilities

1. Weak AI or Narrow AI:

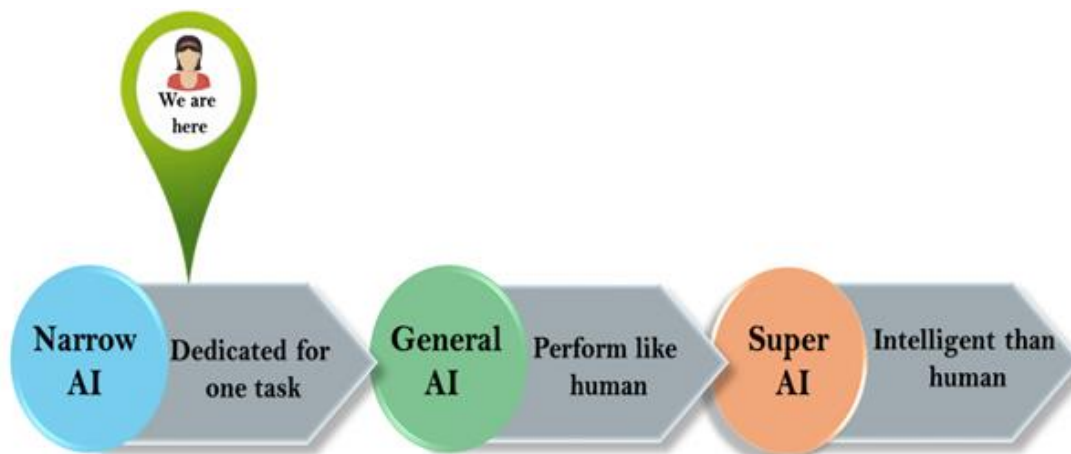
- Narrow AI is a type of AI which is able to perform a dedicated task with intelligence. The most common and currently available AI is Narrow AI in the world of Artificial Intelligence.
- Narrow AI cannot perform beyond its field or limitations, as it is only trained for one specific task. Hence it is also termed as weak AI. Narrow AI can fail in unpredictable ways if it goes beyond its limits.
- Apple Siri is a good example of Narrow AI, but it operates with a limited pre-defined range of functions.
- IBM's Watson supercomputer also comes under Narrow AI, as it uses an Expert system approach combined with Machine learning and natural language processing.
- Some Examples of Narrow AI are playing chess, purchasing suggestions on e-commerce site, self-driving cars, speech recognition, and image recognition.

2. General AI:

- General AI is a type of intelligence which could perform any intellectual task with efficiency like a human.
- The idea behind the general AI is to make such a system which could be smarter and think like a human by its own.
- Currently, there is no such system that exists which could come under general AI and can perform any task as perfect as a human.
- The worldwide researchers are now focused on developing machines with General AI.
- As systems with general AI are still under research, and it will take lots of efforts and time to develop such systems.

3. Super AI:

- Super AI is a level of Intelligence of Systems at which machines could surpass human intelligence, and can perform any task better than human with cognitive properties. It is an outcome of general AI.
- Some key characteristics of strong AI include capability include the ability to think, to reason, solve the puzzle, make judgments, plan, learn, and communicate by its own.
- Super AI is still a hypothetical concept of Artificial Intelligence. Development of such systems in real is still world changing task.



Artificial Intelligence type-2: Based on functionality

1. Reactive Machines

- Purely reactive machines are the most basic types of Artificial Intelligence.
- Such AI systems do not store memories or past experiences for future actions.
- These machines only focus on current scenarios and react on it as per possible best action.
- IBM's Deep Blue system is an example of reactive machines.
- Google's AlphaGo is also an example of reactive machines.

2. Limited Memory

- Limited memory machines can store past experiences or some data for a short period of time.
- These machines can use stored data for a limited time period only.

- Self-driving cars are one of the best examples of Limited Memory systems. These cars can store recent speed of nearby cars, the distance of other cars, speed limit, and other information to navigate the road.

3. Theory of Mind

- Theory of Mind AI should understand the human emotions, people, beliefs, and be able to interact socially like humans.
- This type of AI machines are still not developed, but researchers are making lots of efforts and improvement for developing such AI machines.

4. Self-Awareness

- Self-awareness AI is the future of Artificial Intelligence. These machines will be super intelligent, and will have their own consciousness, sentiments, and self-awareness.
- These machines will be smarter than human mind.
- Self-Awareness AI does not exist in reality still and it is a hypothetical concept.

Advantages of Artificial Intelligence

Following are some main advantages of Artificial Intelligence:

- **High Accuracy with less errors:** AI machines or systems are prone to less errors and high accuracy as it takes decisions as per pre-experience or information.
- **High-Speed:** AI systems can be of very high-speed and fast-decision making, because of that AI systems can beat a chess champion in the Chess game.
- **High reliability:** AI machines are highly reliable and can perform the same action multiple times with high accuracy.
- **Useful for risky areas:** AI machines can be helpful in situations such as defusing a bomb, exploring the ocean floor, where to employ a human can be risky.
- **Digital Assistant:** AI can be very useful to provide digital assistant to the users such as AI technology is currently used by various E-commerce websites to show the products as per customer requirement.
- **Useful as a public utility:** AI can be very useful for public utilities such as a self-driving car which can make our journey safer and hassle-free, facial recognition for security purpose, Natural language processing to communicate with the human in human-language, etc.

Disadvantages of Artificial Intelligence

Every technology has some disadvantages, and the same goes for Artificial intelligence. Being so advantageous technology still, it has some disadvantages

which we need to keep in our mind while creating an AI system. Following are the disadvantages of AI:

- **High Cost:** The hardware and software requirement of AI is very costly as it requires lots of maintenance to meet current world requirements.
- **Can't think out of the box:** Even we are making smarter machines with AI, but still they cannot work out of the box, as the robot will only do that work for which they are trained, or programmed.
- **No feelings and emotions:** AI machines can be an outstanding performer, but still it does not have the feeling so it cannot make any kind of emotional attachment with human, and may sometime be harmful for users if the proper care is not taken.
- **Increase dependency on machines:** With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
- **No Original Creativity:** As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative.

Artificial Intelligence V/S Machine learning

Machine learning

Machine learning is about extracting knowledge from the data. It can be defined as,

Machine learning is a subfield of artificial intelligence, which enables machines to learn from past data or experiences without being explicitly programmed.

Machine learning enables a computer system to make predictions or take some decisions using historical data without being explicitly programmed. Machine learning uses a massive amount of structured and semi-structured data so that a machine learning model can generate accurate result or give predictions based on that data.

Machine learning works on algorithm which learn by its own using historical data. It works only for specific domains such as if we are creating a machine learning model to detect pictures of dogs, it will only give result for dog images, but if we provide a new data like cat image then it will become unresponsive. Machine learning is being

used in various places such as for online recommender system, for Google search algorithms, Email spam filter, Facebook Auto friend tagging suggestion, etc.

It can be divided into three types:

- **Supervised learning**
- **Reinforcement learning**
- **Unsupervised learning**

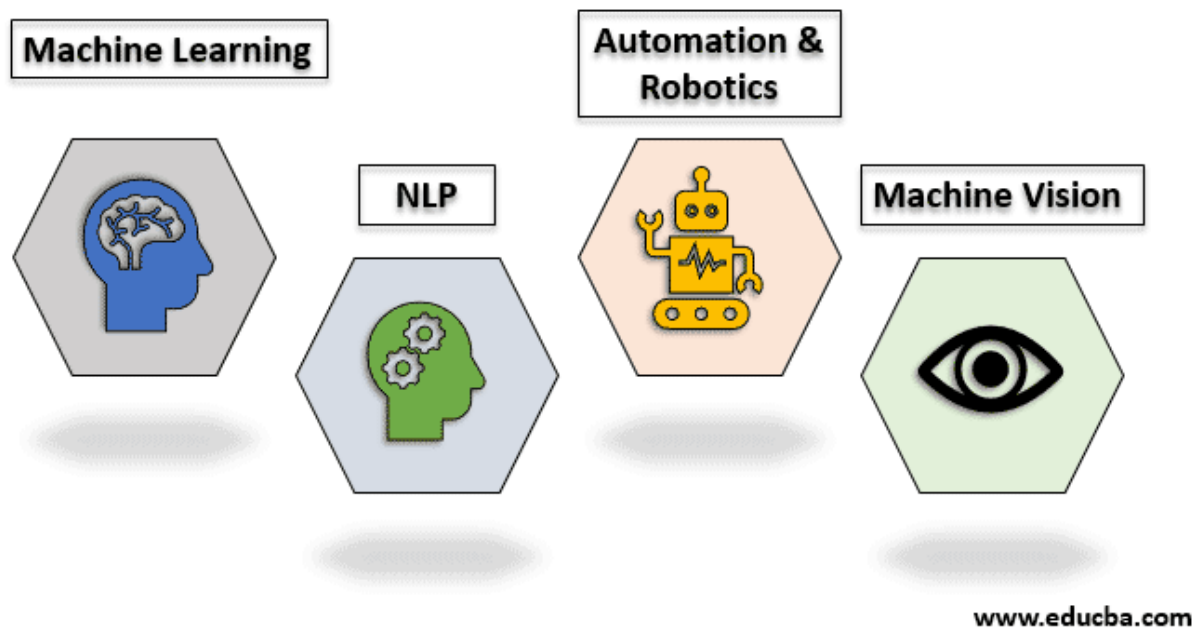
Key differences between Artificial Intelligence (AI) and Machine learning (ML):

| Artificial Intelligence | Machine learning |
|--|--|
| Artificial intelligence is a technology which enables a machine to simulate human behavior. | Machine learning is a subset of AI which allows a machine to automatically learn from past data without programming explicitly. |
| The goal of AI is to make a smart computer system like humans to solve complex problems. | The goal of ML is to allow machines to learn from data so that they can give accurate output. |
| In AI, we make intelligent systems to perform any task like a human. | In ML, we teach machines with data to perform a particular task and give an accurate result. |
| Machine learning and deep learning are the two main subsets of AI. | Deep learning is a main subset of machine learning. |
| AI has a very wide range of scope. | Machine learning has a limited scope. |
| AI is working to create an intelligent system which can perform various complex tasks. | Machine learning is working to create machines that can perform only those specific tasks for which they are trained. |
| AI system is concerned about maximizing the chances of success. | Machine learning is mainly concerned about accuracy and patterns. |
| The main applications of AI are Siri, customer support using chatbots, Expert System, Online game playing, intelligent humanoid robot, etc. | The main applications of machine learning are Online recommender system, Google search algorithms, Facebook auto friend tagging suggestions, etc. |
| On the basis of capabilities, AI can be divided into three types, which are, Weak AI, General AI, and Strong AI. | Machine learning can also be divided into mainly three types that are Supervised learning, Unsupervised learning, and Reinforcement learning. |
| It includes learning, reasoning, and self-correction. | It includes learning and self-correction when introduced with new data. |

| | |
|--|--|
| AI completely deals with Structured, semi-structured, and unstructured data. | Machine learning deals with Structured and semi-structured data. |
|--|--|

AI Techniques

Top 4 Techniques of Artificial Intelligence



1. Machine Learning

It is one of the applications of AI where machines are not explicitly programmed to perform certain tasks; rather, they learn and improve from experience automatically. Deep Learning is a subset of machine learning based on artificial neural networks for predictive analysis. There are various machine learning algorithms, such as Unsupervised Learning, Supervised Learning, and Reinforcement Learning. In Unsupervised Learning, the algorithm does not use classified information to act on it without any guidance. In Supervised Learning, it deduces a function from the training

data, which consists of a set of an input object and the desired output. Reinforcement learning is used by machines to take suitable actions to increase the reward to find the best possibility which should be taken in to account.

2. NLP (Natural Language Processing)

It is the interactions between computers and human language where the computers are programmed to process natural languages. Machine Learning is a reliable technology for Natural Language Processing to obtain meaning from human languages. In NLP, the audio of a human talk is captured by the machine. Then the audio to text conversation occurs, and then the text is processed where the data is converted into audio. Then the machine uses the audio to respond to humans. Applications of [Natural Language Processing \(Links to an external site.\)](#) can be found in IVR (Interactive Voice Response) applications used in call centres, language translation applications like Google Translate and word processors such as Microsoft Word to check the accuracy of grammar in text. However, the nature of human languages makes the Natural Language Processing difficult because of the rules which are involved in the passing of information using natural language, and they are not easy for the computers to understand. So NLP uses algorithms to recognize and abstract the rules of the natural languages where the unstructured data from the human languages can be converted to a format that is understood by the computer.

3. Automation and Robotics

The purpose of Automation is to get the monotonous and repetitive tasks done by machines which also improve productivity and in receiving cost-effective and more efficient results. Many organizations use machine learning, [neural networks \(Links to an external site.\)](#), and graphs in automation. Such automation can prevent fraud issues while financial transactions online by using CAPTCHA technology. Robotic process automation is programmed to perform high volume repetitive tasks which can adapt to the change in different circumstances.

4. Machine Vision

Machines can capture visual information and then analyze it. Here cameras are used to capture the visual information, the analogue to digital conversion is used to convert the image to digital data, and digital signal processing is employed to process the data. Then the resulting data is fed to a computer. In machine vision, two vital aspects are sensitivity, which is the ability of the machine to perceive impulses that are weak and resolution, the range to which the machine can distinguish the objects. The usage of machine vision can be found in signature identification, [pattern recognition \(Links to an external site.\)](#), and medical image analysis, etc.

Knowledge possess following properties:

- It is voluminous.
- It is not well-organised or well-formatted.

- It is constantly changing.
- It differs from data. And it is organised in a way that corresponds to its usage.

AI technique is a method that exploits knowledge that should be represented in such a way that:

- Knowledge captures generalisation. Situations that share common properties are grouped together. Without the property, inordinate amount of memory and modifications will be required.
- It can be easily modified to correct errors and to reflect changes in the world.
- It can be used in many situations even though it may not be totally accurate or complete.
- It can be used to reduce its own volume by narrowing range of possibilities.

LISP programming: Syntax and numeric functions

LISP (list processing)

LISP, an acronym for *list processing*, is a programming language that was designed for easy manipulation of data strings. Developed in 1959 by John McCarthy, it is a commonly used language for artificial intelligence ([AI \(Links to an external site.\)](#)) programming. It is one of the oldest programming languages still in relatively wide use.

John McCarthy invented LISP in 1958, shortly after the development of FORTRAN. It was first implemented by Steve Russell on an IBM 704 computer.

It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information effectively.

Common Lisp originated, during the 1980s and 1990s, in an attempt to unify the work of several implementation groups that were successors to Maclisp, like ZetaLisp and NIL (New Implementation of Lisp) etc.

It serves as a common language, which can be easily extended for specific implementation.

Programs written in Common LISP do not depend on machine-specific characteristics, such as word length etc.

Features of Common LISP

- It is machine-independent
- It uses iterative design methodology, and easy extensibility.
- It allows updating the programs dynamically.
- It provides high level debugging.
- It provides advanced object-oriented programming.
- It provides a convenient macro system.
- It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
- It is expression-based.
- It provides an object-oriented condition system.
- It provides a complete I/O library.
- It provides extensive control structures.

Applications Built in LISP

Large successful applications built in Lisp.

- Emacs
- G2
- AutoCad
- Igor Engraver
- Yahoo Store

Program Structure

LISP expressions are called symbolic expressions or s-expressions. The s-expressions are composed of three valid objects, atoms, lists and strings.

Any s-expression is a valid program.

LISP programs run either on an **interpreter** or as **compiled code**.

A Simple Program

Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt.

```
(+ 7 9 11)
```

LISP returns the result –

```
27
```

If you would like to run the same program as a compiled code, then create a LISP source code file named myprog.lisp and type the following code in it.

```
(write (+ 7 9 11))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is –

```
27
```

LISP Uses Prefix Notation

You might have noted that LISP uses **prefix notation**.

In the above program the + symbol works as the function name for the process of summation of the numbers.

In prefix notation, operators are written before their operands. For example, the expression,

```
a * ( b + c ) / d
```

will be written as –

```
(/ (* a (+ b c) ) d)
```

Evaluation of LISP Programs

Evaluation of LISP programs has two parts –

- Translation of program text into Lisp objects by a reader program
- Implementation of the semantics of the language in terms of these objects by an evaluator program

The evaluation process takes the following steps –

- The reader translates the strings of characters to LISP objects or **s-expressions**.
- The evaluator defines syntax of Lisp **forms** that are built from s-expressions. This second level of evaluation defines a syntax that determines which **s-expressions** are LISP forms.
- The evaluator works as a function that takes a valid LISP form as an argument and returns a value. This is the reason why we put the LISP expression in parenthesis, because we are sending the entire expression/form to the evaluator as arguments.

Basic building blocks in LISP

Basic Building Blocks in LISP

LISP programs are made up of three basic building blocks –

- atom
- list
- string

ATOM

An **atom** is a number or string of contiguous characters. It includes numbers and special characters.

Following are examples of some valid atoms –

```
hello-from-tutorials-point
```

```
name
```

```
123008907
```

```
*hello*
```

```
Block#221
```

```
abc123
```

LIST

A **list** is a sequence of atoms and/or other lists enclosed in parentheses.

Following are examples of some valid lists –

```
( i am a list)

(a ( a b c) d e fgh)

(father tom ( susan bill joe))

(sun mon tue wed thur fri sat)

( )
```

STRING

A **string** is a group of characters enclosed in double quotation marks.

Following are examples of some valid strings –

```
" I am a string"

"a ba c d efg #$$^&!"

"Please enter the following details : "

"Hello from 'Tutorials Point'! "
```

Adding Comments

The semicolon symbol (;) is used for indicating a comment line.

For Example,

```
(write-line "Hello World") ; greet the world

; tell them your whereabouts
```



```
(write-line "I am at 'Tutorials Point'! Learning LISP")
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is –

```
Hello World
```

```
I am at 'Tutorials Point'! Learning LISP
```

Some Notable Points before Moving to Next

Following are some of the important points to note –

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45
- LISP expressions are case-insensitive, cos 45 or COS 45 are same.
- LISP tries to evaluate everything, including the arguments of a function. Only three types of elements are constants and always return their own value
 - Numbers
 - The letter **t**, that stands for logical true.
 - The value **nil**, that stands for logical false, as well as an empty list.

Little More about LISP Forms

In the previous chapter, we mentioned that the evaluation process of LISP code takes the following steps.

- The reader translates the strings of characters to LISP objects or **s-expressions**.
- The evaluator defines syntax of Lisp **forms** that are built from s-expressions. This second level of evaluation defines a syntax that determines which s-expressions are LISP forms.

Now, a LISP forms could be.

- An Atom

- An empty or non-list
- Any list that has a symbol as its first element

The evaluator works as a function that takes a valid LISP form as an argument and returns a value. This is the reason why we put the **LISP expression in parenthesis**, because we are sending the entire expression/form to the evaluator as arguments.

Naming Conventions in LISP

Name or symbols can consist of any number of alphanumeric characters other than whitespace, open and closing parentheses, double and single quotes, backslash, comma, colon, semicolon and vertical bar. To use these characters in a name, you need to use escape character (\).

A name can have digits but not entirely made of digits, because then it would be read as a number. Similarly a name can have periods, but can't be made entirely of periods.

Use of Single Quotation Mark

LISP evaluates everything including the function arguments and list members.

At times, we need to take atoms or lists literally and don't want them evaluated or treated as function calls.

To do this, we need to precede the atom or the list with a single quotation mark.

The following example demonstrates this.

Create a file named main.lisp and type the following code into it.

```
(write-line "single quote used, it inhibits evaluation")

(write '(* 2 3))

(write-line " ")

(write-line "single quote not used, so expression evaluated")

(write (* 2 3))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is –

```
single quote used, it inhibits evaluation
```

```
(* 2 3)
```

single quote not used, so expression evaluated

```
6
```

Basic list manipulation functions

List Manipulation in LISP

List manipulating functions are used to manipulate and return the list of elements

The following table provides some commonly used list manipulating functions.

| Sr.No. | Function & Description |
|--------|---|
| 1 | car It takes a list as argument, and returns its first element. |
| 2 | cdr It takes a list as argument, and returns a list without the first element |
| 3 | cons It takes two arguments, an element and a list and returns a list with the element inserted at the beginning. |
| 4 | list It takes any number of arguments and returns a list with the arguments as member elements. |
| 5 | append It merges two or more lists into one. |
| 6 | last It takes a list and returns a list containing the last element. |

| | |
|---|---|
| 7 | member It takes two arguments of which the second must be a list, if the first argument is a member of the list it returns the remainder of the list beginning with the first argument. |
| 8 | reverse It takes a list and returns a list with the top elements in reverse order. |

1. car() Function:

This function is used to display the first element in the list

Syntax:

```
(car '(element1 element2 ..... element n))
```

Example:

- Lisp

```
;list of integer elements
```

```
(write (car '(1 2 3 4 5)))
```

```
(terpri)
```

```
;list of characters
```

```
(write (car '(a b)))
```

```
(terpri)
```

Output:

```
1
```

```
A
```

2. cdr() Function:

This function is used to display the list of all elements excluding the first element

Syntax:

```
(cdr '(element1 element2 ..... element n))
```

Example:

- Lisp

```
;list of integer elements
```

```
(write (cdr '(1 2 3 4 5)))
```

```
(terpri)
```

```
;list of characters
```

```
(write (cdr '(a b)))
```

```
(terpri)
```

Output:

```
(2 3 4 5)
```

```
(B)
```

3.cons() Function:

This function will take two items. one is a value and another is the list. This will insert the value into first place in the list and returns the new list

Syntax:

```
(cons 'value '(list_of_elements))
```

where,

- **value** is the element to be inserted
- **list_of_elements** is the list of elements

Example:

- Lisp

```
;list of integer elements
```

```
;insert 100
```

```
(write (cons '100 '(1 2 3 4 5)))
```

```
(terpri)
```

```
;list of characters
```

```
;insert a
```

```
(write (cons 'a '(b c d e f)))
```

```
(terpri)
```

Output:

```
(100 1 2 3 4 5)
```

```
(A B C D E F)
```

4. list() Function:

This function will take multiple lists and display it

Syntax:

```
(list 'list1 'list2 ..... 'listn)
```

Example:

- Lisp

```
;two lists and one variable
```

```
(write (list '100 '(1 2 3 4 5) '(g e e k s)))
```

```
(terpri)
```

Output:

```
(100 (1 2 3 4 5) (G E E K S))
```

5. append() Function:

This function will append the two or multiple lists.

Syntax:

```
(append 'list1 'list2 ..... 'listn)
```

Example:

- Lisp

```
;consider two lists and append  
(write (append '(1 2 3 4 5) '(g e e k s)))  
(terpri)  
  
;consider three lists and append  
(write (append '(10 20) '(20 30 45) '(a s d)))  
(terpri)
```

Output:

```
(1 2 3 4 5 G E E K S)  
(10 20 20 30 45 A S D)
```

6. last() Function:

This function returns the list that contains the last element in the list

Syntax:

```
(last 'list)
```

Example:

- Lisp

```
;consider a list and get last element  
(write (last '(1 2 3 4 5) ))  
(terpri)  
  
;consider a list and get last element  
(write (last '(10 20 ( g e e k s))))  
(terpri)
```

Output:

```
(5)  
((G E E K S))
```

7. member() Function:

This function is used to return the list based on members given.

It will take two arguments:

1. The first argument will be the value.
2. The second argument will be the list.

If the value is present in the list, then it will return the list

Otherwise, NIL is returned.

Syntax:

```
(member 'value 'list )
```

Example:

- Lisp

```
;consider a list with 1 as member
```

```
(write (member '1 '(1 2 3 4 5) ))
```

```
(terpri)
```

```
;consider a list with 10 as member
```

```
(write (member '10 '(1 2 3 4 5) ))
```

```
(terpri)
```

Output:

```
(1 2 3 4 5)
```

```
NIL
```

8. reverse() Function:

This function reverses the list and returns the reversed list

Syntax:

```
(reverse 'list)
```

Example:

- Lisp

```
;consider a list and reverse
```

```
(write (reverse '(1 2 3 4 5) ))
```

```
(terpri)
```

```
;consider a list and reverse
```

```
(write (reverse '(g e e k s) ))
```

```
(terpri)
```

Output:

```
(5 4 3 2 1)
```

```
(S K E E G)
```

Input output and local variables

Variables in LISP

LISP supports two types of variables:

1. Local variables
2. Global variables

Local Variables:

They are also called Lexical variables, these variables are defined inside a particular function and only the code within the binding form can refer to them as they can not be accessed outside of that function. Parameters of the functions are also local variables.

Defining a local variable:

New local variables can be created using the special operator **LET**. These variables then can be used within the body of the LET.
The syntax for creating a variable with LET is:

```
(let (variable)
  body-of-expressions)
```

A let expression has two parts

- The first part consists of instructions for creating a variable and assigning values to them
- The second part consists of a list of s-expressions

Now let's create local variables using the above-mentioned syntax

- Lisp

```
(let ((x 10) (y 20) z)
  (format t "Local Variables :~%")
  (format t "x = ~a, y = ~a & z=~a~%" x y z))
```

Here we have assigned **x** and **y** with initial values and for **z** no value is provided, hence **NIL** will be set as z's default value.

Output :

```
Local Variables :
x = 10, y = 20 & z=NIL
```

Global Variables:

These are the variables that have a global scope i.e. you can call access them from anywhere in the program. In LISP global variables are also called Dynamic variables.

Defining a global variable:

New global variables can be defined using **DEFVAR** and **DEFPARAMETER** construct. The difference between both of them is that DEFPARAMETER always assigns initial value and DEFVAR form can be used to create a global variable without giving it a value.

The syntax for creating global variables is:

```
(defvar *name* initial-value)
```

```
"Documentation string")

(defparameter *name* initial-value

"Documentation string")
```

Example:

Let's create a global variable that holds the rate of the pencil as ***pencil-rate***, and with help of one function calculate the total bill by multiplying the total number of pencils with our global variable ***pencil-rate***.

- Lisp

```
(defparameter *pencil-rate* 10

"Pencil rate is set to 10")

(defun calc-bill (n)

"Calculates the total bill by multiplying the number of pencils with its global
rate"

(* *pencil-rate* n))

(write (calc-bill 20))
```

Output:

```
200
```

Input Functions

The following table provides the most commonly used input functions of LISP –

| Sr.No. | Function & Description |
|--------|--|
| 1 | read & optional <i>input-stream eof-error-p eof-value recursive-p</i> It reads in the printed representation of a Lisp object from input-stream, builds a correspond |
| 2 | read-preserving-whitespace & optional <i>in-stream eof-error-p eof-value recursive-p</i> |

| | |
|----|---|
| | It is used in some specialized situations where it is desirable to determine precisely what c |
| 3 | read-line & optional <i>input-stream eof-error-p eof-value recursive-p</i> It reads in a line of text terminated by a newline. |
| 4 | read-char & optional <i>input-stream eof-error-p eof-value recursive-p</i> It takes one character from input-stream and returns it as a character object. |
| 5 | unread-char <i>character & optional input-stream</i> It puts the character most recently read from the input-stream, onto the front of input-str |
| 6 | peek-char & optional <i>peek-type input-stream eof-error-p eof-value recursive-p</i> It returns the next character to be read from input-stream, without actually removing it from |
| 7 | listen & optional <i>input-stream</i> The predicate listen is true if there is a character immediately available from input-stream |
| 8 | read-char-no-hang & optional <i>input-stream eof-error-p eof-value recursive-p</i> It is similar to read-char , but if it does not get a character, it does not wait for a character, |
| 9 | clear-input & optional <i>input-stream</i> It clears any buffered input associated with <i>input-stream</i> . |
| 10 | read-from-string <i>string</i> & optional <i>eof-error-p eof-value & key :start :end :preserve-whites</i> It takes the characters of the string successively and builds a LISP object and returns the o character in the string not read, or the length of the string (or, length +1), as the case may |
| 11 | parse-integer <i>string & key :start :end :radix :junk-allowed</i> It examines the substring of string delimited by :start and :end (default to the beginning a whitespace characters and then attempts to parse an integer. |
| 12 | read-byte <i>binary-input-stream</i> & optional <i>eof-error-p eof-value</i> It reads one byte from the binary-input-stream and returns it in the form of an integer. |

Reading Input from Keyboard

The **read** function is used for taking input from the keyboard. It may not take any argument.

For example, consider the code snippet –

```
(write ( + 15.0 (read)))
```

Example

Create a new source code file named main.lisp and type the following code in it –

```
; the function AreaOfCircle

; calculates area of a circle

; when the radius is input from keyboard


(defun AreaOfCircle()

(terpri)

(princ "Enter Radius: ")

(setq radius (read))

(setq area (* 3.1416 radius radius))

(princ "Area: ")

(write area))

(AreaOfCircle)
```

When you execute the code, it returns the following result –

```
Enter Radius: 5 (STDIN Input)
Area: 78.53999
```

The Output Functions

All output functions in LISP take an optional argument called *output-stream*, where the output is sent. If not mentioned or *nil*, output-stream defaults to the value of the variable **standard-output**.

The following table provides the most commonly used output functions of LISP –

| Sr.No. | Function and Description |
|--------|---|
| 1 | <p>write <i>object</i> & key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :arbitrary-precision :lines :pprint-dispatch</p> <p>Both write the object to the output stream specified by :stream, which defaults to the value of <i>*standard-output*</i>. The other arguments are used to control the printing of the object. The default to the corresponding global variables set for printing.</p> |
| 2 | <p>prin1 <i>object</i> & optional <i>output-stream</i></p> <p>print <i>object</i> & optional <i>output-stream</i></p> <p>pprint <i>object</i> & optional <i>output-stream</i></p> <p>princ <i>object</i> & optional <i>output-stream</i></p> <p>All these functions outputs the printed representation of object to <i>output-stream</i>. However</p> <ul style="list-style-type: none"> • prin1 returns the object as its value. • print prints the object with a preceding newline and followed by a space. It returns the object. • pprint is just like print except that the trailing space is omitted. • princ is just like prin1 except that the output has no escape character |
| 3 | <p>write-to-string <i>object</i> & key :escape :radix :base :circle :pretty :level :length :case :gensym :arbitrary-precision :lines :pprint-dispatch</p> <p>prin1-to-string <i>object</i></p> <p>princ-to-string <i>object</i></p> <p>The object is effectively printed and the output characters are made into a string, which is returned.</p> |
| 4 | <p>write-char <i>character</i> & optional <i>output-stream</i></p> <p>It outputs the character to <i>output-stream</i>, and returns character.</p> |

| | |
|----|--|
| 5 | <p>write-string <i>string</i> & optional <i>output-stream</i> & key :start :end</p> <p>It writes the characters of the specified substring of <i>string</i> to the <i>output-stream</i>.</p> |
| 6 | <p>write-line <i>string</i> & optional <i>output-stream</i> & key :start :end</p> <p>It works the same way as write-string, but outputs a newline afterwards.</p> |
| 7 | <p>terpri & optional <i>output-stream</i></p> <p>It outputs a newline to <i>output-stream</i>.</p> |
| 8 | <p>fresh-line & optional <i>output-stream</i></p> <p>it outputs a newline only if the stream is not already at the start of a line.</p> |
| 9 | <p>finish-output & optional <i>output-stream</i></p> <p>force-output & optional <i>output-stream</i></p> <p>clear-output & optional <i>output-stream</i></p> <ul style="list-style-type: none"> • The function finish-output attempts to ensure that all output sent to output-stream only then returns nil. • The function force-output initiates the emptying of any internal buffers but returns without acknowledgment. • The function clear-output attempts to abort any outstanding output operation and flushes output as possible to continue to the destination. |
| 10 | <p>write-byte <i>integer</i> <i>binary-output-stream</i></p> <p>It writes one byte, the value of the <i>integer</i>.</p> |

Example

Create a new source code file named main.lisp and type the following code in it.

```
; this program inputs a numbers and doubles it

(defun DoubleNumber()

  (terpri)

  (princ "Enter Number : ")
```

```
(setq n1 (read))

(setq doubled (* 2.0 n1))

(princ "The Number: ")

(write n1)

(terpri)

(princ "The Number Doubled: ")

(write doubled)

)

(DoubleNumber)
```

When you execute the code, it returns the following result –

```
Enter Number : 3456.78 (STDIN Input)
The Number: 3456.78
The Number Doubled: 6913.56
```

property lists and arrays

Property Lists in LISP

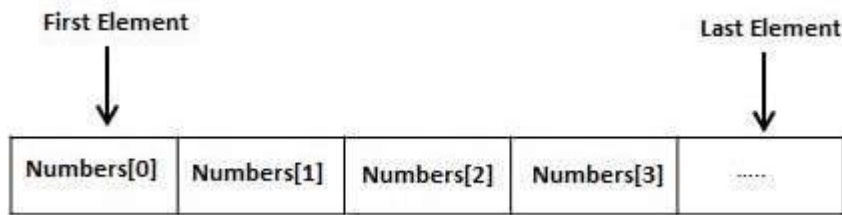
In Lisp, every symbol has a property list (*plist*). When a symbol is created initially its property list is empty. A property list consists of entries where every entry consists of a key called an indicator and a value called a property. There are no duplicates among the indicators. In contrast to association lists the operations for adding and removing *plist* entries alter the *plist* rather than creating a new one.

| Function | Syntax | Usage |
|----------------------|--|---|
| get function | get symbol indicator &optional default | get searched the <i>plist</i> for an indicator equivalent to indicator. If the found value is returned or else the default is returned. If the default is not specified nil is returned. |
| setf function | setf((get function) property/value) | <i>setf</i> is used with getting to create a new property value pair. |
| symbol-plist | (symbol-plist symbol) | <i>symbol-plist</i> allows you to see all the properties of a symbol |
| remprop | remprop symbol indicator | <i>remprop</i> function is used to remove the property equivalent to the indicator. |
| getf | <i>getf</i> place indicator &optional default | <i>getf</i> searches the property list stored in place for an indicator equivalent to an indicator. If one is found, then the corresponding value is returned; otherwise, the default is returned. If the default is not specified, then nil is used for default. |
| remf | <i>remf</i> place indicator | <i>remf</i> removes the property equivalent to the given indicator from the given place. |

LISP - Arrays

LISP allows you to define single or multiple-dimension arrays using the **make-array** function. An array can store any LISP object as its elements.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



The number of dimensions of an array is called its rank.

In LISP, an array element is specified by a sequence of non-negative integer indices. The length of the sequence must equal the rank of the array. Indexing starts from zero.

For example, to create an array with 10- cells, named my-array, we can write –

```
(setf my-array (make-array '(10)))
```

The aref function allows accessing the contents of the cells. It takes two arguments, the name of the array and the index value.

For example, to access the content of the tenth cell, we write –

```
(aref my-array 9)
```

Example 1

Create a new source code file named main.lisp and type the following code in it.

[Live Demo \(Links to an external site.\)](#)

```
(write (setf my-array (make-array '(10))))  
  
(terpri)  
  
(setf (aref my-array 0) 25)  
  
(setf (aref my-array 1) 23)  
  
(setf (aref my-array 2) 45)  
  
(setf (aref my-array 3) 10)  
  
(setf (aref my-array 4) 20)  
  
(setf (aref my-array 5) 17)  
  
(setf (aref my-array 6) 25)  
  
(setf (aref my-array 7) 19)
```

```
(setf (aref my-array 8) 67)

(setf (aref my-array 9) 30)

(write my-array)
```

When you execute the code, it returns the following result –

```
 #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
 #(25 23 45 10 20 17 25 19 67 30)
```

Unit 2

Problem Solving, Search and Control Strategies

Problem definitions:

A problem is defined by its elements and their relations.

To provide a formal description of a problem, we need to do 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.
- d. 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.

Problem definitions:

This process is known as search.

- a.) Search is fundamental to the problem-solving process.
- b.) Search is a general mechanism that can be used when more direct method is not known.
- c.) 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 directed graph, where nodes represent search state and paths represent the operators applied to change the state.
- To simplify a 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.

Problem Space

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.

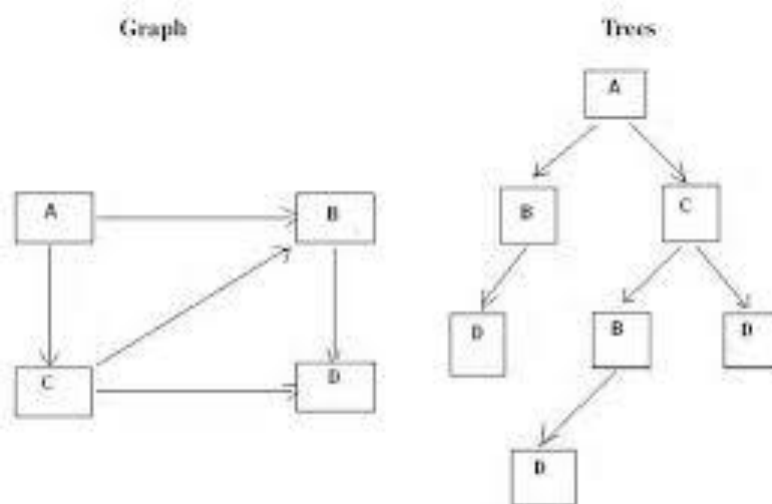


Fig. 2.1 Graph and Tree

States

A state is a representation of elements at 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 tell - how to select elements ?

Among all possible states, there are two special states called :

- a.) Initial state is the start point
- b.) Final state is the goal state

State Change: Successor Function

- a.) A Successor Function is needed for state change.
- b.) The successor function moves one state to another state.

Successor Function :

- ◇ Is a description of possible actions; a set of operators.
- ◇ Is a transformation function on a state representation, which converts that state into another state.
- ◇ Defines a relation of accessibility among states.
- ◇ 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. Definitions of terms :

- ◇ A state space forms a graph (or map) in which the nodes are states and the arcs between nodes are actions.
- ◇ In 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

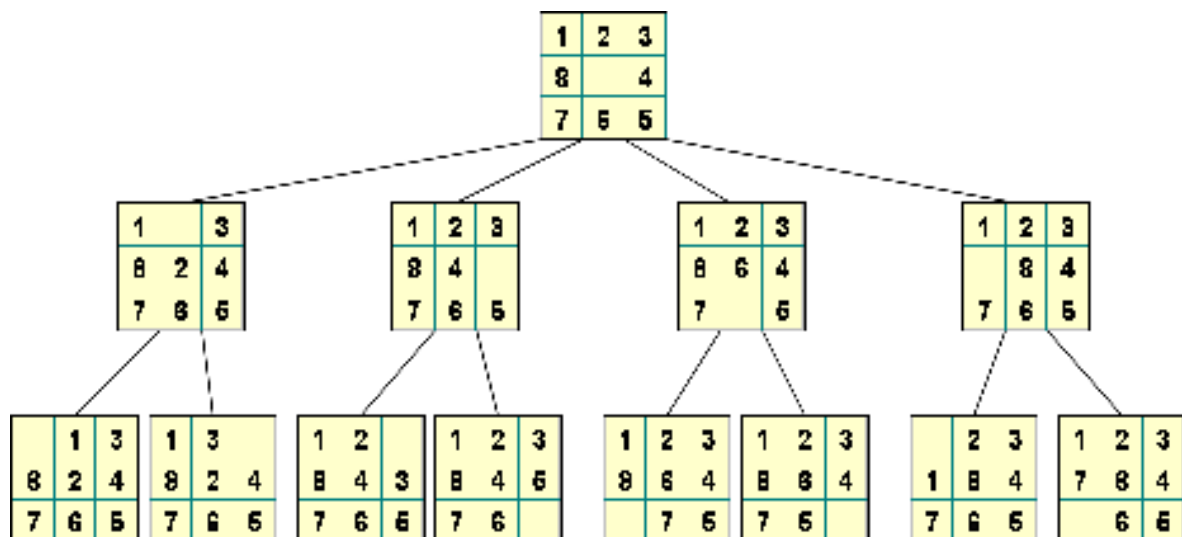
The Structures of state space are trees and graphs.

- Tree is a hierarchical structure in a graphical form; and
 - Graph is a non-hierarchical structure.
 - ◇ 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.
 - ◇ 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.
 - ◇ operators are directed arcs between nodes.
- 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 sometime 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 solution may be any or optimal or all.
- ◇ The importance of cost depends on the problem and the type of solution asked.

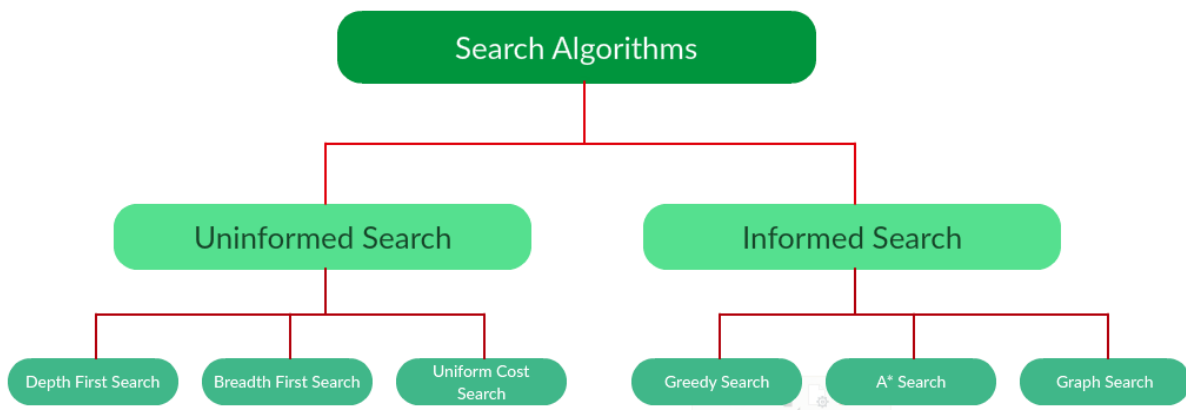


Search Algorithm, Informed V/s uninformed search

Search Algorithm and their terminologies

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
 1. **Search Space:** Search space represents a set of possible solutions, which a system may have.
 2. **Start State:** It is a state from where agent begins **the search**.
 3. **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Types of search algorithms:



Uninformed/Blind Search:

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search

Informed Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Informed search can solve much complex problem which could not be solved in another way.

An example of informed search algorithms is a traveling salesman problem.

1. Greedy Search
2. A* Search

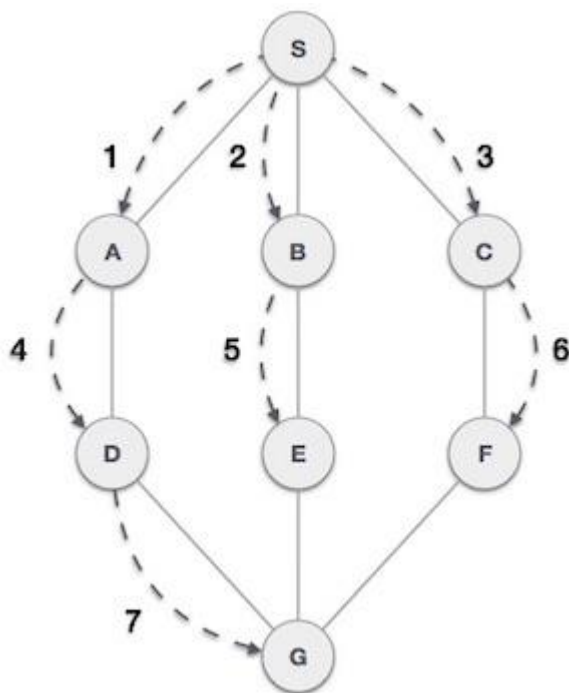
Solutions Informed Search vs. Uninformed Search is depicted pictorially as follows:

| Parameters | Informed Search | Uninformed Search |
|----------------------------|---|--|
| Known as | It is also known as Heuristic Search. | It is also known as Blind Search. |
| Using Knowledge | It uses knowledge for the searching process. | It doesn't use knowledge for the searching process. |
| Performance | It finds a solution more quickly. | It finds solution slow as compared to an informed search. |
| Completion | It may or may not be complete. | It is always complete. |
| Cost Factor | Cost is low. | Cost is high. |
| Time | It consumes less time because of quick searching. | It consumes moderate time because of slow searching. |
| Direction | There is a direction given about the solution. | No suggestion is given regarding the solution in it. |
| Implementation | It is less lengthy while implemented. | It is more lengthy while implemented. |
| Efficiency | It is more efficient as efficiency takes into account cost and performance. The incurred cost is less and speed of finding solutions is quick. | It is comparatively less efficient as incurred cost is more and the speed of finding the Breadth-First solution is slow. |
| Computational requirements | Computational requirements are lessened. | Comparatively higher computational requirements. |
| Size of search problems | Having a wide scope in terms of handling large search problems. | Solving a massive search task is challenging. |
| Examples of Algorithms | <ul style="list-style-type: none"> • Greedy Search • A* Search • AO* Search • Hill Climbing Algorithm | <ul style="list-style-type: none"> • Depth First Search (DFS) • Breadth First Search (BFS) • Branch and Bound |

Depth first search and Breadth first search

BFS

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



Input:

```
A
 / \
B   C
 /   / \
D   E   F
```

Output:

A, B, C, D, E, F

Applications of BFS

Here, are Applications of BFS:

Un-weighted Graphs:

BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.

P2P Networks:

BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.

Web Crawlers:

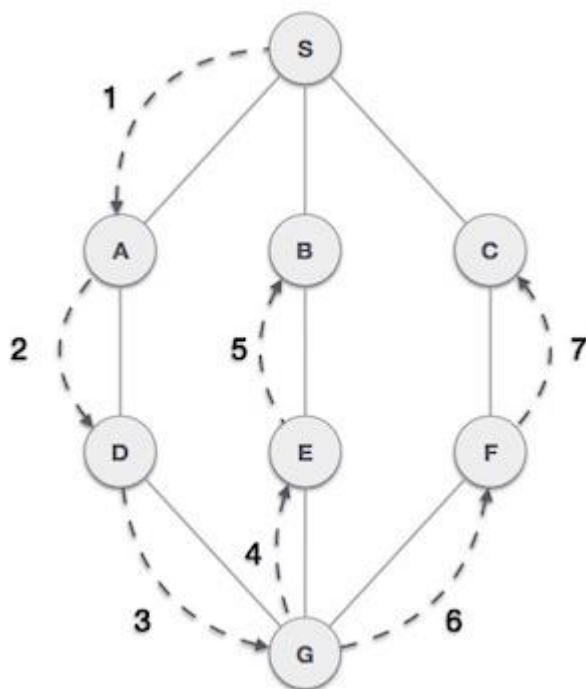
Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.

Network Broadcasting:

A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

DFS(Depth First Search)

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.



Input :



Output:

A, B, C, D, E, F

Applications of DFS

Here are Important applications of DFS:

Weighted Graph:

In a weighted graph, DFS graph traversal generates the shortest path tree and minimum spanning tree.

Detecting a Cycle in a Graph:

A graph has a cycle if we found a back edge during DFS. Therefore, we should run DFS for the graph and verify for back edges.

Path Finding:

We can specialize in the DFS algorithm to search a path between two vertices.

Topological Sorting:

It is primarily used for scheduling jobs from the given dependencies among the group of jobs. In computer science, it is used in instruction scheduling, data serialization, logic synthesis, determining the order of compilation tasks.

Searching Strongly Connected Components of a Graph:

It is used in DFS graph when there is a path from each and every vertex in the graph to other remaining vertices.

Solving Puzzles with Only One Solution:

DFS algorithm can be easily adapted to search all solutions to a maze by including nodes on the existing path in the visited set.

Following are the important differences between BFS and DFS.

| Sr. No. | Key | BFS | DFS |
|---------|-------------------------------|--|--|
| 1 | Definition | BFS, stands for Breadth First Search. | DFS, stands for Depth First Search. |
| 2 | Data structure | BFS uses Queue to find the shortest path. | DFS uses Stack to find the shortest path. |
| 3 | Source | BFS is better when target is closer to Source. | DFS is better when target is far from source. |
| 4 | Suitability for decision tree | As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games. | DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won. |
| 5 | Speed | BFS is slower than DFS. | DFS is faster than BFS. |
| 6 | Time Complexity | Time Complexity of BFS = $O(V+E)$ where V is vertices and E is edges. | Time Complexity of DFS is also $O(V+E)$ where V is vertices and E is edges. |

Hill climbing algorithm

HILL CLIMBING ALGORITHM

A hill-climbing algorithm is a local search algorithm that moves continuously upward (increasing) until the best solution is attained. This algorithm comes to an end when the peak is reached.

The aim of the algorithm is to reach an optimal state which is better than its current state. The starting point which is the non-optimal state is referred to as the base of the hill and it tries to constantly iterate (climb) until it reaches the peak value, that is why it is called Hill Climbing Algorithm.

Key Features of Hill Climbing in Artificial Intelligence

Following are few of the key features of Hill Climbing Algorithm

- **Greedy Approach:** The algorithm moves in the direction of optimizing the cost i.e. finding Local Maxima/Minima
- **No Backtracking:** It cannot remember the previous state of the system so backtracking to the previous state is not possible
- **Feedback Mechanism:** The feedback from the previous computation helps in deciding the next course of action i.e. whether to move up or down the slope

State Space Diagram – Hill Climbing in Artificial Intelligence

- **Local Maxima/Minima:** Local Minima is a state which is better than its neighbouring state, however, it is not the best possible state as there exists a state where objective function value is higher
- **Global Maxima/Minima:** It is the best possible state in the state diagram. Here the value of the objective function is highest
- **Current State:** Current State is the state where the agent is present currently
- **Flat Local Maximum:** This region is depicted by a straight line where all neighbouring states have the same value so every node is local maximum over the region

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state.** It only checks its one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
 1. If it is goal state, then return success and quit.
 2. Else if it is better than the current state then assign new state as a current state.

- 3. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
 1. Let SUCC be a state such that any successor of the current state will be better than it.
 2. For each operator that applies to the current state:
 - I. Apply the new operator and generate a new state.
 - II. Evaluate the new state.
 - III. If it is goal state, then return it and quit, else compare it to the SUCC.
 - IV. If it is better than SUCC, then set new state as SUCC.
 - V. If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

3. Stochastic hill climbing:

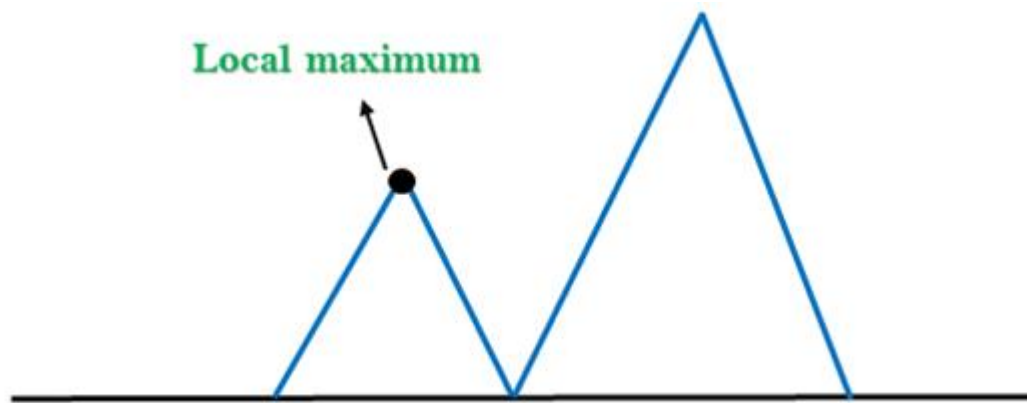
Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm

Here we discuss the problems in the hill-climbing algorithm:

1. Local Maximum

The algorithm terminates when the current node is local maximum as it is better than its neighbours. However, there exists a global maximum where objective function value is higher



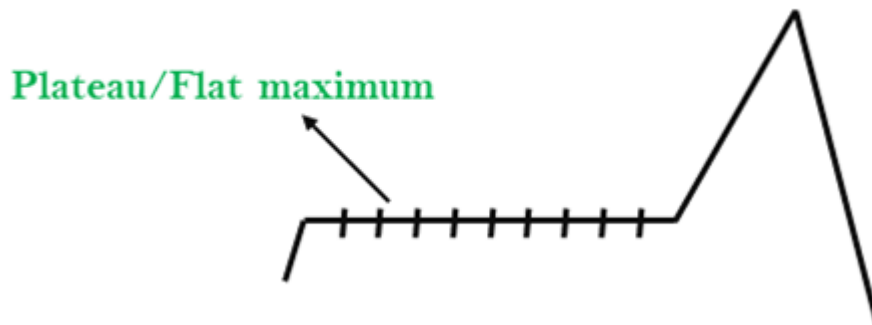
Solution: Back Propagation can mitigate the problem of Local maximum as it starts exploring alternate paths when it encounters Local Maximum

2. Ridge

Ridge occurs when there are multiple peaks and all have the same value or in other words, there are multiple local maxima which are same as global maxima

2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



Bidirectional algorithm

Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Why bidirectional approach?

Because in many cases it is faster, it dramatically reduce the amount of required exploration.

Suppose if branching factor of tree is **b** and distance of goal vertex from source is **d**, then the normal BFS/DFS searching complexity would be $O(b^d)$. On the other hand, if we execute two search operation then the complexity would be $O(b^{d/2})$ for each search and total complexity would be $O(b^{d/2} + b^{d/2})$ which is far less than $O(b^d)$.

When to use bidirectional approach?

We can consider bidirectional approach when-

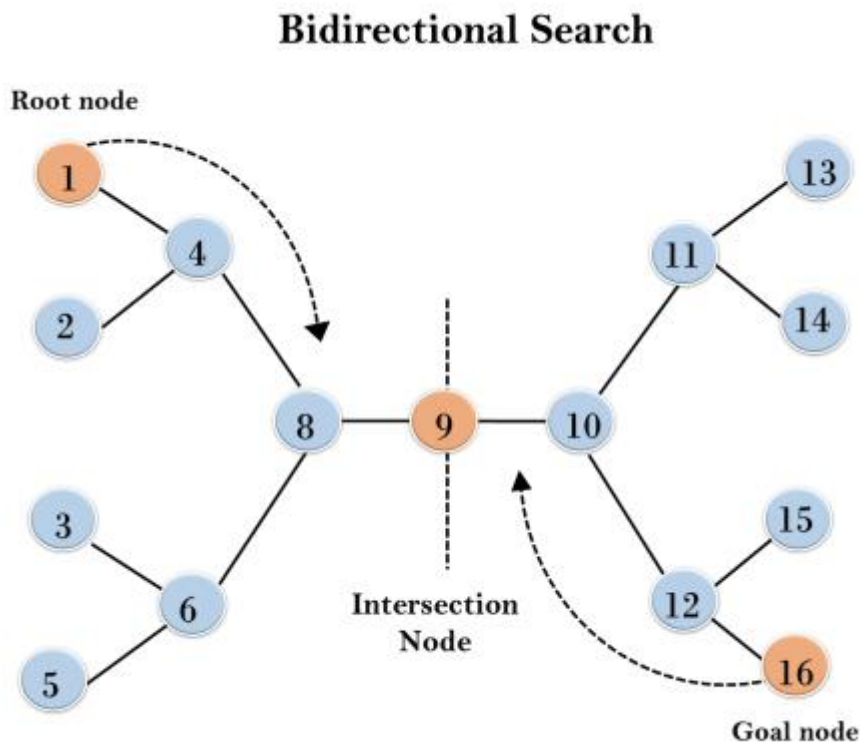
1. Both initial and goal states are unique and completely defined.

2. The branching factor is exactly the same in both directions.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory
- One of the main advantages of bidirectional searches is the speed at which we get the desired results.
- It drastically reduces the time taken by the search by having simultaneous searches.
- It also saves resources for users as it requires less memory capacity to store all the searches.

Disadvantages:

- Implementation of the bidirectional search tree is difficult.

- In bidirectional search, one should know the goal state in advance.
- The fundamental issue with bidirectional search is that the user should be aware of the goal state to use bidirectional search and thereby to decrease its use cases drastically.
- The implementation is another challenge as additional code and instructions are needed to implement this algorithm, and also care has to be taken as each node and step to implement such searches.
- The algorithm must be robust enough to understand the intersection when the search should come to an end or else there's a possibility of an infinite loop.
- It is also not possible to search backwards through all states.

Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is $O(b^d)$.

Space Complexity: Space complexity of bidirectional search is $O(b^d)$.

Optimal: Bidirectional search is Optimal.

Best first search algorithm

Best First Search

Best First Search is a searching algorithm which works on a set of defined rules. It makes use of the concept of priority queues and heuristic search. The objective of this algorithm is to reach the goal state or final state from an initial state by the shortest route possible.

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

[\(Links to an external site.\)](#) [\(Links to an external site.\)](#) [\(Links to an external site.\)](#)

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

Where, $h(n)$ = estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n , from the OPEN list which has the lowest value of $h(n)$, and places it in the CLOSED list.
- **Step 4:** Expand the node n , and generate the successors of node n .
- **Step 5:** Check each successor of node n , and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function $f(n)$, and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

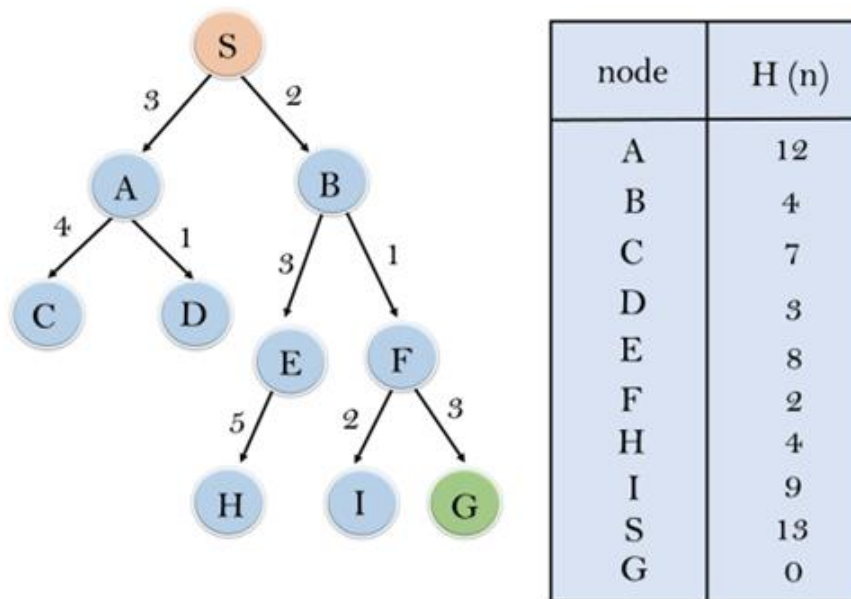
Advantages:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

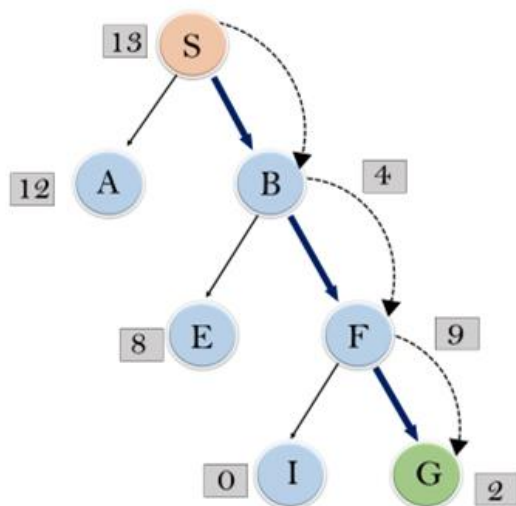
Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

Example:



In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S-----> B----->F-----> G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

pace Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

Beam Search

Beam Search Algorithm

Beam search is an algorithm used in many NLP and speech recognition models as a final decision making layer to choose the best output given target variables like maximum probability or next output character.

Beam search is a heuristic search algorithm that explores a graph by expanding the most optimistic node in a limited set. Beam search is an optimization of **best-first search** that reduces its memory requirements.

Best-first search is a graph search that orders all partial solutions according to some heuristic. But in beam search, only a predetermined number of best partial solutions are kept as candidates. Therefore, it is a **greedy** algorithm.

Beam search uses **breadth-first search** to build its search tree. At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost.

The beam width can either be **fixed** or **variable**. One approach that uses a variable beam width starts with the width at a minimum. If no solution is found, the beam is widened, and the procedure is repeated.

Components of Beam Search

A beam search takes three components as its input:

1. A problem to be solved,
2. A set of heuristic rules for pruning,
3. And a memory with a limited available capacity.

Beam Search Algorithm

beamSearch(problemSet, ruleSet, memorySize)

openMemory = new memory of size memorySize

nodeList = problemSet.listOfNodes

node = root or initial search node

Add node to openMemory;

while (node is not a goal node)

 Delete node from openMemory;

 Expand node and obtain its children, evaluate those children;

 If a child node is pruned according to a rule in ruleSet, delete it;

 Place remaining, non-pruned children into openMemory;

 If memory is full and has no room for new nodes, remove the worst

 node, determined by ruleSet, in openMemory;

 node = the least costly node in openMemory;

Uses of Beam Search

A beam search is most often used to maintain tractability in large systems with insufficient memory to store the entire search tree. For example,

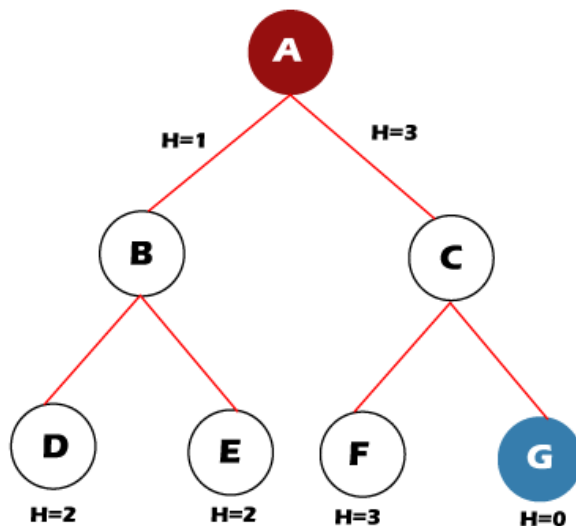
- It has been used in many machine translation systems.
- Each part is processed to select the best translation, and many different ways of translating the words appear.
- According to their sentence structures, the top best translations are kept, and the rest are discarded. The translator then evaluates the translations according to a given criterion, choosing the translation which best keeps the goals.
- The first use of a beam search was in the ***Harpy Speech Recognition System***, CMU 1976.

Drawbacks of Beam Search

Here is a drawback of the Beam Search with an example:

- In general, the Beam Search Algorithm is not **complete**. Despite these disadvantages, beam search has found success in the practical areas of **speech recognition, vision, planning, and machine learning**.
- The main disadvantages of a beam search are that the search may not result in an optimal goal and may not even reach a goal at all after given unlimited time and memory when there is a path from the start node to the goal node.
- The beam search algorithm terminates for two cases: a required goal node is reached, or a goal node is not reached, and there are no nodes left to be explored.

For example, let's take the value of $B = 2$ for the tree shown below. So, follow the following steps to find the goal node.



Step 1: OPEN = {A}

Step 2: OPEN = {B, C}

Step 3: OPEN = {D, E}

Step 4: OPEN = {E}

Step 5: OPEN = { }

The open set becomes empty without finding the goal node.

Time Complexity of Beam Search

The time complexity of the Beam Search algorithm depends on the following things, such as:

- The accuracy of the heuristic function.
- In the worst case, the heuristic function leads Beam Search to the deepest level in the search tree.
- The worst-case time = $O(B*m)$

B is the beam width, and m is the maximum depth of any path in the search tree.

Space Complexity of Beam Search

The space complexity of the Beam Search algorithm depends on the following things, such as:

- Beam Search's memory consumption is its most desirable trait.
- Since the algorithm only stores B nodes at each level in the search tree.
- The worst-case space complexity = $O(B*m)$

B is the beam width, and m is the maximum depth of any path in the search tree.

A* Algorithm

What is an A* Algorithm?

It is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

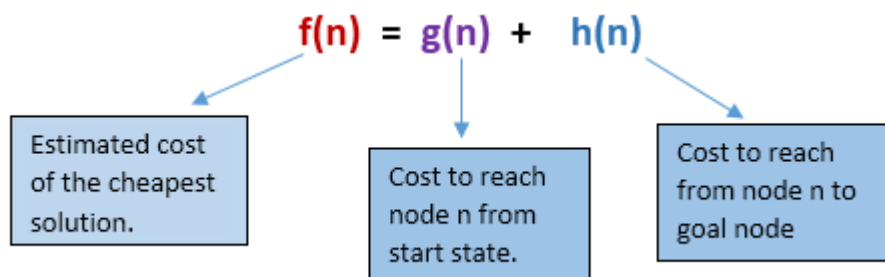
- A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
- It is essentially a best first search algorithm.

- It maintains a tree of paths originating at the start node.
- It extends those paths one edge at a time.
- It continues until its termination criterion is satisfied.

Why A* Search Algorithm?

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



Algorithm-

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

The algorithm is as follows-

Step-01:

- Define a list OPEN.
- Initially, OPEN consists solely of a single node, the start node S.

Step-02:

If the list is empty, return failure and exit.

Step-03:

- Remove node n with the smallest value of $f(n)$ from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step-04:

Expand node n .

Step-05:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S .
- Otherwise, go to Step-06.

Step-06:

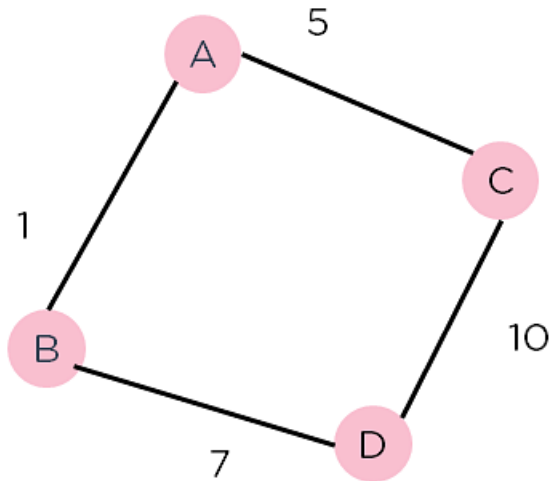
For each successor node,

- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

Step-07:

Go back to Step-02.

How Does the A* Algorithm Work?



Consider the weighted graph depicted above, which contains nodes and the distance between them. Let's say you start from A and have to go to D.

Now, since the start is at the source A, which will have some initial heuristic value. Hence, the results are

$$f(A) = g(A) + h(A)$$

$$f(A) = 0 + 6 = 6$$

Next, take the path to other neighbouring vertices :

$$f(A-B) = 1 + 4$$

$$f(A-C) = 5 + 2$$

Now take the path to the destination from these nodes, and calculate the weights :

$$f(A-B-D) = (1 + 7) + 0$$

$$f(A-C-D) = (5 + 10) + 0$$

It is clear that node B gives you the best path, so that is the node you need to take to reach the destination.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition $f(n) < c^*$, where c^* is the cost of optimal solution path. $f(n) > c^*$

Complete: A* algorithm is complete as long as:

- Branching factor is finite.

- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that $h(n)$ should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d . So the time complexity is $O(b^d)$, where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **$O(b^d)$**

AO* Algorithm

Introduction:

AO* Algorithm basically based on problem decomposition (Breakdown problem into small pieces)

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.

AND-OR Graph

The figure shows an AND-OR graph

1. To pass any exam, we have two options, either cheating or hard work.
2. In this graph we are given two choices, first do cheating **or (The red line)** work hard and **(The arc)** pass.
3. When we have more than one choice and we have to pick one, we apply **OR condition** to choose one.(That's what we did here).
4. Basically the **ARC** here denote **AND condition**.
5. Here we have replicated the arc between the work hard and the pass because by doing the hard work possibility of passing an exam is more than cheating.

A* Vs AO*

1. Both are part of informed search technique and use heuristic values to solve the problem.
2. The solution is guaranteed in both algorithm.
3. A* **always** gives an **optimal solution** (shortest path with low cost) But It is not guaranteed to that **AO*** always provide **an optimal solutions**.
4. **Reason:** Because AO* does not explore all the solution path once it got solution.

How AO* works

OPEN: It contains the nodes that have been traversed but yet not been marked solvable or unsolvable.

CLOSE: It contains the nodes that have already been processed.

h(n): The distance from the current node to the goal node.

Working of AO algorithm:

The AO* algorithm works on the formula given below :

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

where,

- $g(n)$: The actual cost of traversal from initial state to the current state.
- $h(n)$: The estimated cost of traversal from the current state to the goal state.
- $f(n)$: The actual cost of traversal from the initial state to the goal state.

AO* Search Algorithm

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T_0 .

Step 3: Select a node n that is both on OPEN and a member of T_0 . Remove it from OPEN and place it in CLOSE

Step 4: If n is the terminal goal node then levelled n as solved and levelled 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.

Let's try to understand it with the following diagram

The algorithm always moves towards a **lower cost value**.

Basically, We will calculate the **cost function** here ($F(n) = G(n) + H(n)$)

H: heuristic/ estimated value of the nodes. and **G:** actual cost or edge value (here unit value).

Here we have taken the **edges value 1** , meaning we have to focus solely on the **heuristic value**.

1. The **Purple color** values are **edge values (here all are same that is one)**.
2. The **Red color** values are **Heuristic values for nodes**.
3. **The Green color** values are **New Heuristic values for nodes**.

Branch and Bound technique

Branch and Bound technique

Branch and Bound (B&B) is a problem-solving technique which is widely used for various problems encountered in operations research and combinatorial mathematics. Various heuristic search procedures used in artificial intelligence (AI) are considered to be related to B&B procedures.

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

Branch and Bound is an algorithmic technique which finds the optimal solution by keeping the best solution found so far. If partial solution can't improve on the best it is abandoned, by this method the number of nodes which are explored can also be reduced. It also deals with the optimization problems over a search that can be presented as the leaves of the search tree. The usual technique for eliminating the sub trees from the search tree is called pruning. For Branch and Bound algorithm we will use stack data structure.

Concept:

Step 1: Traverse the root node.

Step 2: Traverse any neighbour of the root node that is maintaining least distance from the root node.

Step 3: Traverse any neighbour of the neighbour of the root node that is maintaining least distance from the root node.

Step 4: This process will continue until we are getting the goal node.

Algorithm:

Step 1: PUSH the root node into the stack.

Step 2: If stack is empty, then stop and return failure.

Step 3: If the top node of the stack is a goal node, then stop and return success.

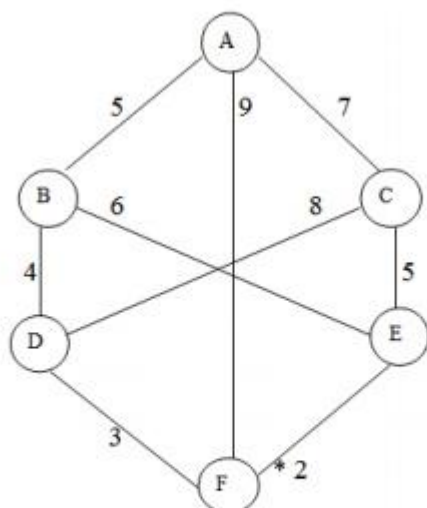
Step 4: Else POP the node from the stack. Process it and find all its successors. Find out the path containing all its successors as well as predecessors and then PUSH the successors which are belonging to the minimum or shortest path.

Step 5: Go to step 5.

Step 6: Exit.

Implementation:

Let us take the following example for implementing the Branch and Bound algorithm.



Figure

Step 1:

Consider the node A as our root node. Find its successors i.e. B, C, F. Calculate the distance from the root and PUSH them according to least distance.

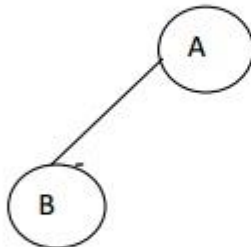
| | |
|---|--|
| A | |
|---|--|

B: $0+5 = 5$ (The cost of A is 0 as it is the starting node)

F: $0+9 = 9$

C: $0+7 = 7$

Here B (5) is the least distance.

**Step 2:**

Now the stack will be

| | | | |
|---|---|---|--|
| C | F | B | |
|---|---|---|--|

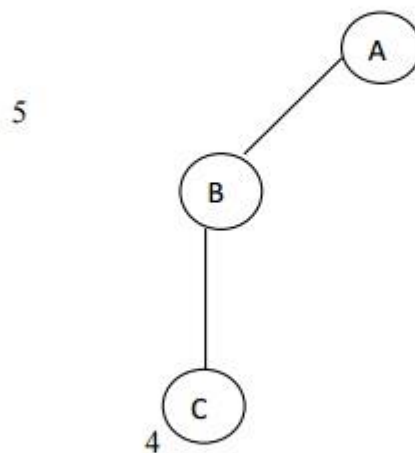
A

As B is on the top of the stack so calculate the neighbours of B.

$$D: 0+5+4 = 9$$

$$E: 0+5+6 = 11$$

The least distance is D from B. So it will be on the top of the stack.



Step 3:

As the top of the stack is D. So calculate neighbours of D.



$$C: 0+5+4+8 = 17$$

$$F: 0+5+4+3 = 12$$

The least distance is F from D and it is our goal node. So stop and return success.

Step 4:



Hence the searching path will be A-B -D-F

Advantages:

As it finds the minimum path instead of finding the minimum successor so there should not be any repetition.

The time complexity is less compared to other algorithms.

Disadvantages:

The load balancing aspects for Branch and Bound algorithm make it parallelization difficult.

The Branch and Bound algorithm is limited to small size network. In the problem of large networks, where the solution search space grows exponentially with the scale of the network, the approach becomes relatively prohibitive.

Constraint satisfaction problem

Constraint Satisfaction Problems in Artificial Intelligence

Constraint satisfaction is a technique where a problem is solved when its values satisfy certain constraints or rules of the problem. Such type of technique leads to a deeper understanding of the problem structure as well as its complexity.

Constraint satisfaction depends on three components, namely:

- **X:** It is a set of variables.
- **D:** It is a set of domains where the variables reside. There is a specific domain for each variable.
- **C:** It is a set of constraints which are followed by the set of variables.

In constraint satisfaction, domains are the spaces where the variables reside, following the problem specific constraints. These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of **{scope, rel}**. The **scope** is a tuple of variables which participate in the constraint and **rel** is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.

Popular Problems with CSP

The following problems are some of the popular problems that can be solved using CSP:

1. CryptArithmetic (Coding alphabets to numbers.)
2. n-Queen (In an n-queen problem, n queens should be placed in an $n \times n$ matrix such that no queen shares the same row, column or diagonal.)
3. Map Coloring (coloring different regions of map, ensuring no adjacent regions have the same color)
4. Crossword (everyday puzzles appearing in newspapers)
5. Sudoku (a number grid)
6. Latin Square Problem

Solving Constraint Satisfaction Problems

The requirements to solve a constraint satisfaction problem (CSP) is:

- A state-space
- The notion of the solution.

A state in state-space is defined by assigning values to some or all variables such as $\{X_1=v_1, X_2=v_2, \text{ and so on...}\}$.

An assignment of values to a variable can be done in three ways:

- **Consistent or Legal Assignment:** An assignment which does not violate any constraint or rule is called Consistent or legal assignment.
- **Complete Assignment:** An assignment where every variable is assigned with a value, and the solution to the CSP remains consistent. Such assignment is known as Complete assignment.
- **Partial Assignment:** An assignment which assigns values to some of the variables only. Such type of assignments are called Partial assignments.

Types of Domains in CSP

There are following two types of domains which are used by the variables :

- **Discrete Domain:** It is an infinite domain which can have one state for multiple variables. **For example,** a start state can be allocated infinite times for each variable.
- **Finite Domain:** It is a finite domain which can have continuous states describing one domain for one specific variable. It is also called a continuous domain.

Constraint Types in CSP

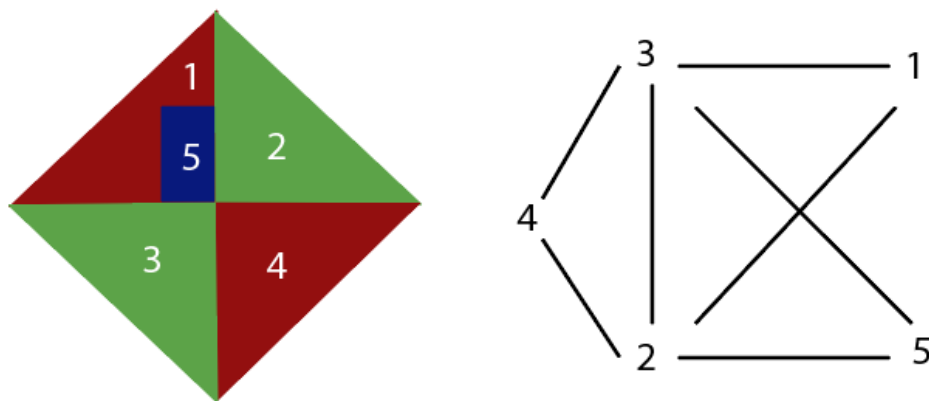
With respect to the variables, basically there are following types of constraints:

- **Unary Constraints:** It is the simplest type of constraints that restricts the value of a single variable.
- **Binary Constraints:** It is the constraint type which relates two variables. A value x_2 will contain a value which lies between x_1 and x_3 .
- **Global Constraints:** It is the constraint type which involves an arbitrary number of variables.

CSP Problems

Constraint satisfaction includes those problems which contains some constraints while solving the problem. CSP includes the following problems:

- **Graph Coloring:** The problem where the constraint is that no adjacent sides can have the same color.



Graph Coloring

2. SUDOKU -

SUDOKU

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | | | | | | | 5 | 9 |
| 2 | 6 | | 5 | | | | 3 | |
| | | | | 9 | 2 | | | |
| | | 2 | | 6 | | | 1 | |
| | | 3 | 8 | 1 | 9 | 7 | | |
| | 7 | | | 3 | | 5 | | |
| | | | 3 | 4 | | | | |
| | 3 | | | | 6 | | 2 | 7 |
| 5 | 9 | | | | | | | 6 |

Puzzle

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 7 | 6 | 8 | 3 | 2 | 5 | 9 |
| 2 | 6 | 9 | 5 | 7 | 1 | 8 | 3 | 4 |
| 3 | 8 | 5 | 4 | 9 | 2 | 6 | 7 | 1 |
| 8 | 4 | 2 | 7 | 6 | 5 | 9 | 1 | 3 |
| 6 | 5 | 3 | 8 | 1 | 9 | 7 | 4 | 2 |
| 9 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 8 |
| 7 | 2 | 6 | 3 | 4 | 8 | 1 | 9 | 5 |
| 1 | 3 | 8 | 9 | 5 | 6 | 4 | 2 | 7 |
| 5 | 9 | 4 | 1 | 2 | 7 | 3 | 8 | 6 |

Solution

- **n-queen problem:** In n-queen problem, the constraint is that no queen should be placed either diagonally, in the same row or column.

Note: The n-queen problem is already discussed in Problem-solving in AI section.

- **Crossword:** In crossword problem, the constraint is that there should be the correct formation of the words, and it should be meaningful.

| | | | | | | | | | | | | | | |
|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | B | | | | | | | | | |
| | | | | | A | | | | | B | A | B | Y | |
| | C | | | | B | | | | | | | O | | |
| | R | | | | Y | | D | | | J | | T | | C |
| | I | | | | S | | I | | | D | O | C | T | O |
| | B | I | R | T | H | | A | | | H | | L | | Y |
| | | | | | O | | P | | | N | | E | | |
| | | | | | W | | E | | | S | | | | |
| | | | | | E | | R | | | O | | | | |
| | | | U | L | T | R | A | S | O | U | N | D | | |
| | | | | | | | | | | | | | | |
| | | | | | | T | W | I | N | S | | | | |


Cryptarithmic Problem

Cryptarithmic Problem is a type of [constraint satisfaction problem \(Links to an external site.\)](#) where the game is about digits and its unique replacement either with alphabets or other symbols. In **cryptarithmic problem**, the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

The rules or constraints on a cryptarithmic problem are as follows:

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., $2+2=4$, nothing else.
- Digits should be from **0-9** only.
- There should be only one carry forward, while performing the addition operation on a problem.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S)**, or **righthand side (R.H.S)**

1.

| | | |
|--------|---|---|
| BASE | |  |
| + BALL | | |
| <hr/> | | |
| GAMES | | |
| <hr/> | | |
| | | |
| | | |
| B | 7 | |
| A | 4 | |
| S | 8 | |
| E | 3 | |
| L | 5 | |
| G | 1 | |
| M | 9 | |

Unite 3

Knowledge Representation in AI

Knowledge Representation

- Knowledge representation and reasoning (KR, KRR) is the part of Artificial intelligence which concerned with AI agents thinking and how thinking contributes to intelligent behavior of agents.
- It is responsible for representing information about the real world so that a computer can understand and can utilize this knowledge to solve the complex real world problems such as diagnosis a medical condition or communicating with humans in natural language.
- It is also a way which describes how we can represent knowledge in artificial intelligence. Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

What to Represent:

Following are the kind of knowledge which needs to be represented in AI systems:

- **Object:** All the facts about objects in our world domain. E.g., Guitars contains strings, trumpets are brass instruments.
- **Events:** Events are the actions which occur in our world.
- **Performance:** It describe behavior which involves knowledge about how to do things.
- **Meta-knowledge:** It is knowledge about what we know.
- **Facts:** Facts are the truths about the real world and what we represent.
- **Knowledge-Base:** The central component of the knowledge-based agents is the knowledge base. It is represented as KB. The Knowledgebase is a group of the Sentences (Here, sentences are used as a technical term and not identical with the English language).

Different Types of Knowledge

There are 5 types of Knowledge such as:



1. Declarative Knowledge:

- Declarative knowledge is to know about something.
- It includes concepts, facts, and objects.
- It is also called descriptive knowledge and expressed in declarative sentences.
- It is simpler than procedural language.

2. Procedural Knowledge

- It is also known as imperative knowledge.
- Procedural knowledge is a type of knowledge which is responsible for knowing how to do something.
- It can be directly applied to any task.
- It includes rules, strategies, procedures, agendas, etc.
- Procedural knowledge depends on the task on which it can be applied.

3. Meta-knowledge:

- Knowledge about the other types of knowledge is called Meta-knowledge.

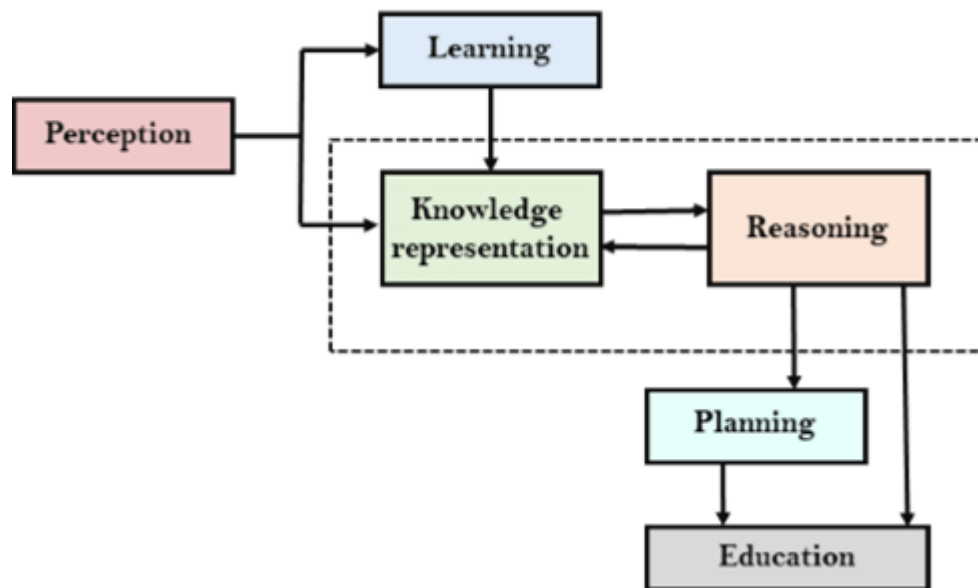
4. Heuristic knowledge:

- Heuristic knowledge is representing knowledge of some experts in a field or subject.
- Heuristic knowledge is rules of thumb based on previous experiences, awareness of approaches, and which are good to work but not guaranteed.

5. Structural knowledge:

- Structural knowledge is basic knowledge to problem-solving.
- It describes relationships between various concepts such as kind of, part of, and grouping of something.
- It describes the relationship that exists between concepts or objects.

Cycle of Knowledge Representation in AI



Techniques of Knowledge Representation in AI



Logical Representation

Logical representation is a language with some definite rules which deal with propositions and has no ambiguity in representation.

Each sentence can be translated into logics using syntax and semantics.

Syntax

- It decides how we can construct legal sentences in logic.
- It determines which symbol we can use in knowledge representation.
- Also, how to write those symbols.

Semantics

- Semantics are the rules by which we can interpret the sentence in the logic.
- It assigns a meaning to each sentence.

Semantic Network Representation

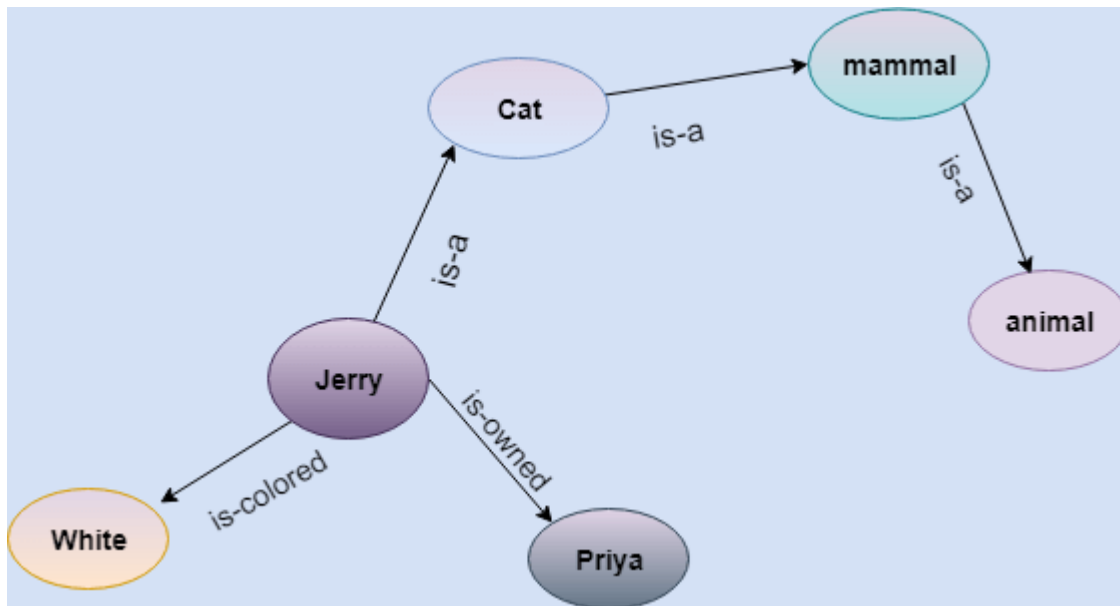
Semantic networks work as an **alternative** of **predicate logic** for knowledge representation. In Semantic networks, you can represent your knowledge in the form of graphical networks.

This representation consist of two types of relations:

- **IS-A relation (Inheritance)**
- **Kind-of-relation**

Statements:

1. Jerry is a cat.
2. Jerry is a mammal
3. Jerry is owned by Priya.
4. Jerry is brown colored.
5. All Mammals are animal.



Frame Representation

A frame is a **record** like structure that consists of a **collection of attributes** and values to describe an entity in the world. These are the AI data structure that divides knowledge into substructures by representing stereotypes situations

Example: 1

Let's take an example of a frame for a book

| Slots | Filters |
|----------------|-------------------------|
| Title | Artificial Intelligence |
| Genre | Computer Science |
| Author | Peter Norvig |
| Edition | Third Edition |
| Year | 1996 |
| Page | 1152 |

Production Rules

In production rules, agent checks for the **condition** and if the condition exists then production rule fires and corresponding action is carried out. The condition part of the rule determines which rule may be applied to a problem. Whereas, the action part carries out the associated problem-solving steps. This complete process is called a recognize-act cycle.

The production rules system consists of three main parts:

- **The set of production rules**
- **Working Memory**
- **The recognize-act-cycle**

Example:

- IF (at bus stop AND bus arrives) THEN action (get into the bus)
- IF (on the bus AND paid AND empty seat) THEN action (sit down).
- IF (on bus AND unpaid) THEN action (pay charges).
- IF (bus arrives at destination) THEN action (get down from the bus).

Proposition logic

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example

- a) Sun is hot (True proposition)
- b) The Sun rises from West (False proposition)
- c) $3+3=7$ (False proposition)
- d) 5 is a prime number (True proposition)

Following are some basic facts about propositional logic:

- Because it operates with 0 and 1, propositional logic is also known as Boolean logic.
- In propositional logic, symbolic variables are used to express the logic, and any symbol can be used to represent a proposition, such as A, B, C, P, Q, R, and so on.
- Propositions can be true or untrue, but not both at the same time.

- An object, relations or functions, and **logical connectives** make up propositional logic.
- Logical operators are another name for these connectives.
- The essential parts of propositional logic are propositions and connectives.
- Connectives are logical operators that link two sentences together.
- **Tautology**, commonly known as a legitimate sentence, is a proposition formula that is always true.
- **Contradiction** is a proposition formula that is always false.
- Statements that are inquiries, demands, or opinions are not propositions, such as "**Where is Rohini**", "**How are you**", and "**What is your name**" are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

1. **Atomic Propositions**
2. **Compound propositions**

- **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

a) $2+2$ is 4, it is an atomic proposition as it is a true fact.

b) "The Sun is cold" is also a proposition as it is a false fact.

- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:-

a) "It is raining today, and street is wet."

b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

| Sl. | Type | Symbol | Description |
|-----|----------|----------|--|
| 1 | Negation | $\neg P$ | It represents a Negative condition. P is a positive statement, and $\neg P$ indicates NOT condition. Example: Today is Monday (P), Today is not a Monday ($\neg P$) |

| | | | |
|---|----------------|-----------------------|---|
| 2 | Conjunction | $P \wedge Q$ | It joins two statements P, Q with AND clause. Example: Ram is a cricket player (P). Ram is a Hockey player (Q). Ram plays both cricket and Hockey is represented by $(P \wedge Q)$ |
| 3 | Disjunction | $P \vee Q$ | It joins two statements P, Q with OR Clause. Example: Ram leaves for Mumbai (P) and Ram leaves for Chennai (Q). Ram leaves for Chennai or Mumbai is represented by $(P \vee Q)$. In this complex statement, at any given point of time if P is True Q is not true and vice versa. |
| 4 | Implication | $P \rightarrow Q$ | Sentence (Q) is dependent on sentence (P), and it is called implication. It follows the rule of If then clause. If sentence P is true, then sentence Q is true. The condition is unidirectional. Example: If it is Sunday (P) then I will go to Movie (Q), and it is represented as $P \rightarrow Q$ |
| 5 | Bi-conditional | $P \Leftrightarrow Q$ | Sentence (Q) is dependent on sentence (P), and vice versa and conditions are bi-directional in this connective. If a conditional statement and its converse are true, then it is called as bi-conditional connective (Implication condition in both the directions $P \rightarrow Q$ and $Q \rightarrow P$). If and only if all conditions are true, then the end statement is true. Example: If I have 1000 Rupees then only I will go to Bar. The converse condition that I will go to Bar if and only if I have Rs 1000. The first statement covers necessity and the second one covers sufficiency. |

Truth Table

The following table depicts the truth values of various combinations of Boolean conditions for Statements P and Q for all the logical connectives.

| P | Q | Negation | | Conjunction | Disjunction | Implication | Bi-conditional |
|-------|-------|----------|----------|--------------|-------------|-------------------|-----------------------|
| | | $\neg P$ | $\neg Q$ | $P \wedge Q$ | $P \vee Q$ | $P \rightarrow Q$ | $P \Leftrightarrow Q$ |
| True | True | False | False | True | True | True | True |
| True | False | False | True | False | True | False | False |
| False | True | True | False | False | True | True | False |
| False | False | True | True | False | False | True | True |

Precedence of connectives:

Precedence

First Precedence

Operators

Parenthesis

Second Precedence

Negation

Third Precedence

Conjunction(AND)

Fourth Precedence

Disjunction(OR)

Fifth Precedence

Implication

Six Precedence

Biconditional

Examples of PL sentences

- P means "It is hot."
- Q means "It is humid."
- R means "It is raining."
- $(P \wedge Q) \rightarrow R$

"If it is hot and humid, then it is raining"

$Q \rightarrow P$

"If it is humid, then it is hot"

• A better way:

Hot = "It is hot"

Humid = "It is humid"

Raining = "It is raining"

First Order Predicate Logic

What is first-order logic (FOL)?

1. **FOL** is a mode of representation in Artificial Intelligence. It is an extension of PL.
2. FOL represents natural language statements in a concise way.
3. FOL is also called *predicate logic*. It is a powerful language used to develop information about an object and express the relationship between objects.
4. FOL not only assumes that does the world contains facts (like PL does), but it also assumes the following:
 - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pit, etc.

- **Relations:** It is unary relation such as red, round, sister of, brother of, etc.
- **Function:** father of, best friend, third inning of, end of, etc.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

| | |
|--------------------|--|
| Constant | 1, 2, A, John, Mumbai, cat,.... |
| Variables | x, y, z, a, b,.... |
| Predicates | Brother, Father, >,.... |
| Function | sqrt, LeftLegOf, |
| Connectives | \wedge , \vee , \neg , \Rightarrow , \Leftrightarrow |
| Equality | == |
| Quantifier | \forall , \exists |

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2,, term n)**.

Example: Ravi and Ajay are brothers: \Rightarrow Brothers(Ravi, Ajay).
Chinky is a cat: \Rightarrow cat (Chinky).

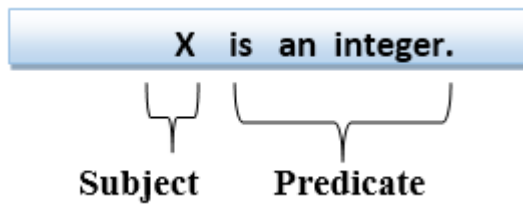
Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
 1. **Universal Quantifier, (for all, everyone, everything)**
 2. **Existential quantifier, (for some, at least one).**

Universal Quantifier:

The Universal quantifier is represented by a symbol \forall , which resembles an inverted A.

Note: In universal quantifier we use implication " \rightarrow ".

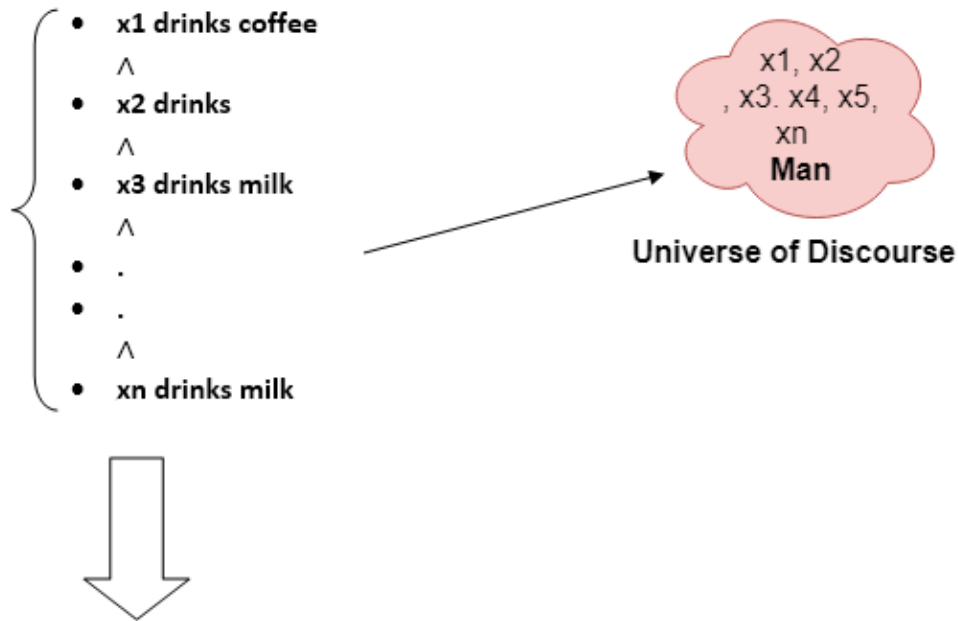
If x is a variable, then $\forall x$ is read as:

- **For all x**
- **For each x**
- **For every x .**

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



So in shorthand notation, we can write it as :

$\forall x \text{ man}(x) \rightarrow \text{drink}(x, \text{coffee}).$

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

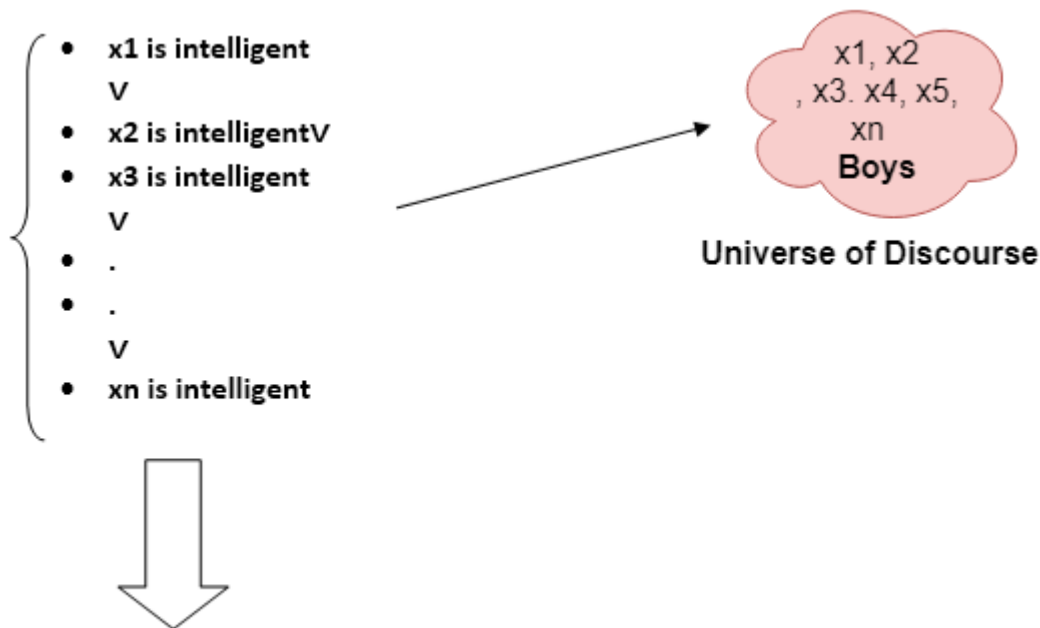
It is denoted by the logical operator \exists , which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

Note: In Existential quantifier we always use AND or Conjunction symbol (\wedge).

- There exists a ' x .'
- For some ' x .'
- For at least one ' x .'

Example:

Some boys are intelligent.



So in short-hand notation, we can write it as:

Some Examples of FOL using quantifier:

1. All birds fly.

In this question the predicate is "**fly(bird).**"

And since there are all birds who fly so it will be represented as follows.

$$\forall x \text{ bird}(x) \rightarrow \text{fly}(x).$$

2. Every man respects his parent.

In this question, the predicate is "**respect(x, y),**" where $x = \text{man}$, and $y = \text{parent}$.

Since there is every man so will use \forall , and it will be represented as follows:

$$\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent}).$$

3. Some boys play cricket.

In this question, the predicate is "**play(x, y),**" where $x = \text{boys}$, and $y = \text{game}$. Since there are some boys so we will use \exists , and it will be represented as:

$$\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket}).$$

4. Not all students like both Mathematics and Science.

In this question, the predicate is "**like(x, y),**" where $x = \text{student}$, and $y = \text{subject}$.

Since there are not all students, so we will use \forall with negation, so following representation for this:

$$\neg \forall (x) [\text{student}(x) \rightarrow \text{like}(x, \text{Mathematics}) \wedge \text{like}(x, \text{Science})].$$

5. Only one student failed in Mathematics.

In this question, the predicate is "**failed(x, y),**" where $x = \text{student}$, and $y = \text{subject}$.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$$\exists (x) [\text{student}(x) \rightarrow \text{failed}(x, \text{Mathematics}) \wedge \forall (y) [\neg (x=y) \wedge \text{student}(y) \rightarrow \neg \text{failed}(x, \text{Mathematics})].$$

Skolemization

Every first-order formula may be converted into Skolem normal form while not changing its satisfiability via a process called **Skolemization** (sometimes spelled **Skolemization**). The resulting formula is not necessarily equivalent to the original one, but is equisatisfiable with it: it is satisfiable if and only if the original one is satisfiable

Reduction to Skolem normal form is a method for removing existential quantifiers from formal logic statements. It is the super standard normal form. Skolemization eliminates existential quantifiers by replacing each existentially quantified variable with a Skolem constant or Skolem function.

What is skolem function in AI?

The basic idea is to have universally quantifiers appropriately flagged and to have existentially quantified variables replaced by what are called **Skolem functions**. ... This means to rename some of the dummy variables if necessary to insure that each quantifier has its own unique dummy variables.

There are some rules that are followed when we convert first order predicate logic into skolem standard form:-

1. Convert FOPL into PNF(Prenex Normal Form), if it is not in PNF
 2. Convert PNF into CNF(Conjunctive normal form)
 3. Apply skolemization
- Remove all existential variable with new skolemization constant (A Skolem constant is a function of no variables.) if there is no universal quantifier before related existential quantifier. Remove existential quantifier.
 - Remove all existential variable with new skolemization function if there are universal quantifier before related to existential quantifier. In another word an existential variable is replaced by a Skolem function of all the universal variables to its left. Remove existential quantifier.

Example:-

$$\begin{aligned} & (\exists x) (\forall y) (\forall v) (\exists z) (\forall u) (\exists w) P(x, y, z, u, v, w) \\ & (\forall y) (\forall v) (\forall w) P(c, y, f(y, v), g(y, v, w), v, w) \end{aligned}$$

4. Skolemize

- substitute new name for each existential var

$$\exists x. P(x) \Rightarrow P(\text{Fred})$$

$$\exists x, y. R(x, y) \Rightarrow R(\text{Thing1}, \text{Thing2})$$

$$\exists x. P(x) \wedge Q(x) \Rightarrow P(\text{Fleep}) \wedge Q(\text{Fleep})$$

$$\exists x. P(x) \wedge \exists x. Q(x) \Rightarrow P(\text{Frog}) \wedge Q(\text{Grog})$$

$$\exists y. \forall x. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Englebert})$$

- substitute new function of all universal vars in outer scopes

$$\forall x. \exists y. \text{Loves}(x, y) \Rightarrow \forall x. \text{Loves}(x, \text{Beloved}(x))$$

$$\forall x. \exists y. \forall z. \exists w. P(x, y, z) \wedge R(y, z, w) \Rightarrow$$

$$P(x, F(x), z) \wedge R(F(x), z, G(x, z))$$

Resolution Principle

Resolution

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

Clause: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

1. John likes all kind of food.
 2. Apple and vegetable are food
 3. Anything anyone eats and not killed is food.
 4. Anil eats peanuts and still alive
 5. Harry eats everything that Anil eats.
- Prove by resolution that:**
6. John likes peanuts.

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.

- a. $\forall x: \text{food}(x) \rightarrow \text{likes}(\text{John}, x)$
 - b. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 - c. $\forall x \forall y: \text{eats}(x, y) \wedge \neg \text{killed}(x) \rightarrow \text{food}(y)$
 - d. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$.
 - e. $\forall x : \text{eats}(\text{Anil}, x) \rightarrow \text{eats}(\text{Harry}, x)$
 - f. $\forall x: \neg \text{killed}(x) \rightarrow \text{alive}(x)$
 - g. $\forall x: \text{alive}(x) \rightarrow \neg \text{killed}(x)$
 - h. $\text{likes}(\text{John}, \text{Peanuts})$
- } **added predicates.**

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- **Eliminate all implication (\rightarrow) and rewrite**
 1. $\forall x \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 2. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 3. $\forall x \forall y \neg [\text{eats}(x, y) \wedge \neg \text{killed}(x)] \vee \text{food}(y)$
 4. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$
 5. $\forall x \neg \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{Harry}, x)$
 6. $\forall x \neg [\neg \text{killed}(x)] \vee \text{alive}(x)$
 7. $\forall x \neg \text{alive}(x) \vee \neg \text{killed}(x)$
 8. $\text{likes}(\text{John}, \text{Peanuts})$.
- **Move negation (\neg)inwards and rewrite**
 1. $\forall x \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 2. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 3. $\forall x \forall y \neg \text{eats}(x, y) \vee \text{killed}(x) \vee \text{food}(y)$
 4. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$
 5. $\forall x \neg \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{Harry}, x)$

6. $\forall x \neg \text{killed}(x) \supset \vee \text{alive}(x)$
 7. $\forall x \neg \text{alive}(x) \vee \neg \text{killed}(x)$
 8. $\text{likes}(\text{John}, \text{Peanuts})$.
- **Rename variables or standardize variables**
 1. $\forall x \neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 2. $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$
 3. $\forall y \forall z \neg \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$
 4. $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$
 5. $\forall w \neg \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Harry}, w)$
 6. $\forall g \neg \text{killed}(g) \supset \vee \text{alive}(g)$
 7. $\forall k \neg \text{alive}(k) \vee \neg \text{killed}(k)$
 8. $\text{likes}(\text{John}, \text{Peanuts})$.
 - **Eliminate existential instantiation quantifier by elimination.**
 In this step, we will eliminate existential quantifier \exists , and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
 - **Drop Universal quantifiers.**
 In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.
 1. $\neg \text{food}(x) \vee \text{likes}(\text{John}, x)$
 2. $\text{food}(\text{Apple})$
 3. $\text{food}(\text{vegetables})$
 4. $\neg \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$
 5. $\text{eats}(\text{Anil}, \text{Peanuts})$
 6. $\text{alive}(\text{Anil})$
 7. $\neg \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Harry}, w)$
 8. $\text{killed}(g) \vee \text{alive}(g)$
 9. $\neg \text{alive}(k) \vee \neg \text{killed}(k)$
 10. $\text{likes}(\text{John}, \text{Peanuts})$.

Note: Statements " $\text{food}(\text{Apple}) \wedge \text{food}(\text{vegetables})$ " and " $\text{eats}(\text{Anil}, \text{Peanuts}) \wedge \text{alive}(\text{Anil})$ " can be written in two separate statements.

- **Distribute conjunction \wedge over disjunction \vee .**
 This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as $\neg \text{likes}(\text{John}, \text{Peanuts})$

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



- # Unification Principle

What is Unification?

- Unification is the process of finding a substitute that makes two separate logical atomic expressions identical. The substitution process is necessary for unification.
- It accepts two literals as input and uses substitution to make them identical.
- Let Ψ_1 and Ψ_2 be two atomic sentences, and be a unifier such that $\Psi_{1\sigma} = \Psi_{2\sigma}$, then **UNIFY(Ψ_1, Ψ_2)** can be written.
- **Example: Find the MGU for Unify{King(x), King(John)}**
Let $\Psi_1 = \text{King}(x)$, $\Psi_2 = \text{King}(\text{John})$,

Substitution $\theta = \{\text{John}/x\}$ is a unifier for these atoms, and both equations will be equivalent if this substitution is used.

- For unification, the UNIFY algorithm is employed, which takes two atomic statements and returns a unifier for each of them (If any exist).
- All first-order inference techniques rely heavily on unification.
- If the expressions do not match, the result is failure.
- The replacement variables are referred to as MGU (Most General Unifier).

○

E.g. Let's say there are two different expressions, **P(x, y)**, and **P(a, f(z))**.

In this case, we must make both of the preceding assertions identical. For this, we'll make a substitute.

P(x, y)..... (i)

P(a, f(z))..... (ii)

Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.

The first expression will be identical to the second expression with both replacements, and the substitution set will be: **[a/x , f(z)/y]**.

Conditions for Unification

The following are some fundamental requirements for unification:

- Atoms or expressions with various predicate symbols can never be united.
- Both phrases must have the same number of arguments.
- If two comparable variables appear in the same expression, unification will fail.

Implementation of the Algorithm

Step 1: Begin by making the substitute set empty.

Step 2: Unify atomic sentences in a recursive manner:

a. Check for expressions that are identical.

b. If one expression is a variable $v\psi_i$, and the other is a term t_i which does not contain variable v_i , then:

a. Substitute t_i / v_i in the existing substitutions

b. Add t_i / v_i to the substitution setlist.

c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

Find the most general unifier for each pair of the following atomic statements (If exist).

1. Find **the MGU of $\{p(f(a), g(Y)) \text{ and } p(X, X)\}$**

Sol: $S_0 \Rightarrow$ Here, $\psi_1 = p(f(a), g(Y))$, and $\psi_2 = p(X, X)$

SUBST $\theta = \{f(a) / X\}$

$S_1 \Rightarrow \psi_1 = p(f(a), g(Y))$, and $\psi_2 = p(f(a), f(a))$

SUBST $\theta = \{f(a) / g(y)\}$, **Unification failed.**

Unification is not possible for these expressions.

2. Find **the MGU of $\{p(b, X, f(g(Z))) \text{ and } p(Z, f(Y), f(Y))\}$**

$S_0 \Rightarrow \{p(b, X, f(g(Z))); p(Z, f(Y), f(Y))\}$

SUBST $\theta = \{b/Z\}$

$S_1 \Rightarrow \{p(b, X, f(g(b))); p(b, f(Y), f(Y))\}$

SUBST $\theta = \{f(Y) / X\}$

$S_2 \Rightarrow \{p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))\}$

SUBST $\theta = \{g(b) / Y\}$

$S_2 \Rightarrow \{p(b, f(g(b)), f(g(b))); p(b, f(g(b)), f(g(b)))\}$ **Unified Successfully.**

And Unifier = $\{b/Z, f(Y) / X, g(b) / Y\}$.

3. Find **the MGU of $\{p(X, X), \text{ and } p(Z, f(Z))\}$**

Here, $\psi_1 = p(X, X)$, and $\psi_2 = p(Z, f(Z))$

$S_0 \Rightarrow \{p(X, X), p(Z, f(Z))\}$

SUBST $\theta = \{X/Z\}$

$S_1 \Rightarrow \{p(Z, Z), p(Z, f(Z))\}$

SUBST $\theta = \{f(Z) / Z\}$, **Unification Failed.**

Therefore, unification is not possible for these expressions.

5. Find **the MGU of $Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)$**

Here, $\Psi_1 = Q(a, g(x, a), f(y))$, and $\Psi_2 = Q(a, g(f(b), a), x)$

$S_0 = \{Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)\}$

SUBST $\theta = \{f(b)/x\}$

$S_1 = \{Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))\}$

SUBST $\theta = \{b/y\}$

SUBST $\theta = \{f(Y)/X\}$

$S_2 = \{p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))\}$

$S_1 = \{Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))\}$, **Successfully Unified.**

Unifier: $[a/a, f(b)/x, b/y]$.

6. UNIFY(**knows(Richard, x), knows(Richard, John)**)

Here, $\Psi_1 = \text{knows(Richard, x)}$, and $\Psi_2 = \text{knows(Richard, John)}$

$S_0 = \{\text{knows(Richard, x); knows(Richard, John)}\}$

S SUBST $\theta = \{\text{John}/x\}$

$S_1 = \{\text{knows(Richard, John); knows(Richard, John)}\}$, **Successfully Unified.**

Unifier: $\{\text{John}/x\}$.

Horn's Clauses

Horn clause is clause (a disjunction of literals) with at most one positive, i.e. unnegated, literal. A clause with at most one positive (unnegated) literal is called a Horn Clause.

Deductive Database:

A type of database which can make conclusions based on sets of well-defined rules, stored in database. RDBMS and logic programming are combined using this.

Deductive database designing is done with help of pure declarative programming language known as Datalog.

Types of Horn Clauses :

- **Definite clause / Strict Horn clause –**
It has exactly one positive literal.
- **Unit clause –**
Definite clause with no negative literals.

- **Goal clause –**
Horn clause without a positive literal.

In Datalog, rules are expressed as a restricted form of clauses called Horn clauses, in which a clause can contain at most one positive literal.

A formula can have following quantifiers:

- **Universal quantifier –**
It can be understood as – “For all x , $P(x)$ holds”, meaning $P(x)$ is true for every object x in universe. For example, all trucks has wheels.
- **Existential quantifier –**
It can be understood as – “There exists an x such that $P(x)$ ”, meaning $P(x)$ is true for at least one object x of universe. For example, someone cares for you.

Forms of a Horn Clause :

1. $\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \dots \text{ OR } \text{NOT}(P_n) \text{ OR } Q$
2. $\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \dots \text{ OR } \text{NOT}(P_n)$

Transformation of Horn Clause :

The Horn clause in (1) can be transformed into clause –

$$P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n \rightarrow Q$$

which is written in Datalog as following rules

$$Q \text{ :- } P_1, P_2, \dots, P_n$$

The above Datalog Rule, is hence a Horn clause.

If predicates $P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n$, are all true for a particular binding to their variable arguments, then Q is also true and can hence be inferred.

The Horn clause in (2) can be transformed into –

$$P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n \rightarrow$$

which is written in Datalog as follows:

$$P_1, P_2, \dots, P_n$$

The above *Datalog* expression can be considered as an integrity constraint, where all predicates must be true to satisfy query.

A query in Datalog consists of two components:

- A Datalog program, which is a finite set of rules.
- A literal $P(X_1, X_2, X_3, \dots, X_n)$, where each X is a variable or a constant.

Semantic Networks

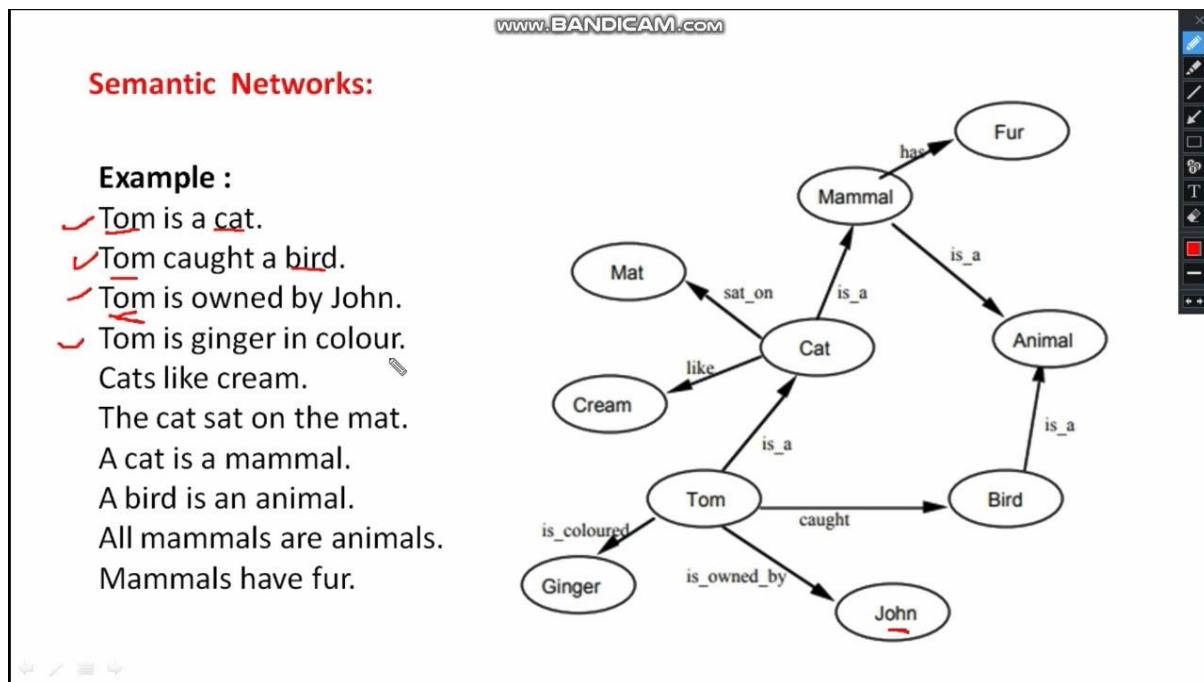
WHAT ARE SEMANTIC NETS IN AI?

Semantic Networks or **Semantic Net** is a knowledge representation technique used for propositional information. Though its versions were long being used in philosophy, **cognitive science** (in the form of semantic memory), and linguistics, Semantic Network's implementation in **computer science** was first developed for artificial intelligence and machine learning. It is a **knowledge base** that **represents concepts** in a network and the systematic relations between them.

SEMANTIC NETWORK EXAMPLES:

Some of the examples of Semantic Networks are:

- **WordNet:** A lexical database of English that groups English words into sets of synonyms (synsets), provides definitions, and records semantic relations between them.
- **Gellish Model:** It is a formal language that is defined as a network of relations between concepts and names of concepts.
- **Logical Descriptions:** Semantic Networks can also be used to represent logical descriptions like Charles Sanders Peirce's existential graphs or John F. Sowa's **conceptual graphs**.



COMPONENTS OF SEMANTIC NETWORKS:

Semantic Networks can further be defined by specifying its fundamental components, which are:

- **Lexical Components** - Consists of:
 - **Nodes represent** the object or concept.
 - **Links:** Denoting relation between nodes.
 - **Labels:** Denoting particular objects & relations.
- **Structural Component** - Here the links and nodes form a directed graph wherein the labels are placed on the link and nodes.
- **Semantic Component** - The meanings here are related to the links and labels of nodes, whereas the facts are dependent on the approved areas.
- **Procedural Part** - The creation of new links and nodes is permitted by constructors, whereas the destructors are responsible for the removal of links and nodes.

SEMANTIC NETWORK ARCHITECTURE:

As stated earlier, the semantic network is a simple knowledge representation technique. It uses **graphic notation** to represent knowledge or data, wherein a graph of labeled nodes and labels are used, with directed arcs to encode knowledge. It follows a simple and comprehensible architecture, which helps add and change information efficiently.

TYPES OF SEMANTIC NETWORKS:

Semantic networks were developed initially for computers in 1956 by Richard H. R. of the Cambridge Language Research Unit (CLRU), for machine translation of **natural languages**. However, now it is used for a variety of functions, like knowledge representation. There are currently six types of semantic networks that enable declarative graphic representation, which is further used to represent knowledge and support automated systems for reasoning about the knowledge. These six types of semantic networks are:

- **Definitional Networks:** Emphasize the subtype or is-a relationship between a concept type and a newly defined subtype.
- **Assertional Networks:** Designed to assert propositions.
- **Implicational Networks:** Uses implications as the primary relationship for connecting nodes.
- **Executable Networks:** Contain mechanisms that can cause some change to the network itself.
- **Learning Networks:** It builds or extends the representation by acquiring knowledge from examples.
- **Hybrid Networks:** These combine two or more of the previous techniques, either in a single network or in separate, but closely interacting networks.

APPLICATION OF SEMANTIC NETWORKS:

With the growing need for intelligent machines, the application of semantic networks is also increasing. Therefore, listed here are some of the areas where Semantic Networks are applied or used:

- In **natural language processing** applications like semantic parsing, word sense disambiguation, etc.
- Specialized retrieval tasks, like plagiarism detection.
- Knowledge Graph proposed by Google in 2012 uses semantic networks in the search engines.

1. ADVANTAGES:

- It is simple and comprehensible.
- Efficient in space requirement.
- Easily clusters related knowledge.
- It is flexible and easy to visualize.
- It is a natural representation of knowledge.
- Conveys meaning in a transparent manner.

2. DISADVANTAGES:

Though the importance of Semantic Networks is immense in Knowledge Representation, we must consider the drawbacks it offers, such as:

- Inheritance cause problems.
- Links on objects represent only binary options.
- Interactable for large domains.
- Don't represent performances or meta-knowledge effectively.
- It's difficult to express some properties using Semantic Networks, like negation, disjunction, etc.

Scripts

Scripts

A script is a structured representation describing a stereotyped sequence of events in a particular context.

Scripts are used in natural language understanding systems to organize a knowledge base in terms of the situations that the system should understand. Scripts use a frame-like structure to represent the commonly occurring experience like going to the movies eating in a restaurant, shopping in a supermarket, or visiting an ophthalmologist.

Thus, a script is a structure that prescribes a set of circumstances that could be expected to follow on from one another.

Scripts are beneficial because:

- Events tend to occur in known runs or patterns.
- A casual relationship between events exist.
- An entry condition exists which allows an event to take place.
- Prerequisites exist upon events taking place.

Components of a script

The components of a script include:

- **Entry condition:** These are basic condition which must be fulfilled before events in the script can occur.
- **Results:** Condition that will be true after events in script occurred.
- **Props:** Slots representing objects involved in events
- **Roles:** These are the actions that the individual participants perform.
- **Track:** Variations on the script. Different tracks may share components of the same scripts.
- **Scenes:** The sequence of events that occur.

Describing a script, special symbols of actions are used. These are:

| Symbol | Meaning | Example |
|---------------|---|----------------|
| ATRANS | transfer a relationship | give |
| PTRANS | transfer physical location of an object | go |
| PROPEL | apply physical force to an object | push |
| MOVE | move body part by owner | kick |
| GRASP | grab an object by an actor | hold |

| | | |
|--------|--------------------------------------|--------|
| INGEST | taking an object by an animal eat | drink |
| EXPEL | expel from animal's body | cry |
| MTRANS | transfer mental information | tell |
| MBUILD | mentally make new information | decide |
| CONC | conceptualize or think about an idea | think |
| SPEAK | produce sound | say |
| ATTEND | focus sense organ | listen |

Example:-Script for going to the bank to withdraw money.

SCRIPT : Withdraw money

TRACK : Bank

PROPS : Money

Counter

Form

Token

Roles : P= Customer

E= Employee

C= Cashier

Entry conditions: P has no or less money.

The bank is open.

Results : P has more money.

Scene 1: Entering

P PTRANS P into the Bank

P ATTEND eyes to E

P MOVE P to E

Scene 2: Filling form

P MTRANS signal to E

E ATRANS form to P

P PROPEL form for writing

P ATRANS form to P

E ATRANS form to P

Scene 3: Withdrawing money

P ATTEND eyes to counter

P PTRANS P to queue at the counter

P PTRANS token to C

C ATRANS money to P

Scene 4: Exiting the bank

P PTRANS P to out of bank

Advantages of Scripts

- Ability to predict events.
- A single coherent interpretation maybe builds up from a collection of observations.

Disadvantages of Scripts

- Less general than frames.
- May not be suitable to represent all kinds of knowledge

Frame system and value inheritance

Frames

Frame based representation is a development of semantic nets and allow us to express the idea of inheritance.

A Frame System consists of a set of frames (or nodes), which are connected together by relations. Each frame describes either an instance or a class. Each frame has one or more slots, which are assigned slot values. This is the way in which the frame system is built up. Rather than simply having links between

frames, each relationship is expressed by a value being placed in a slot.
Example:

Example: 1

Let's take an example of a frame for a book

| Slots | Filters |
|----------------|-------------------------|
| Title | Artificial Intelligence |
| Genre | Computer Science |
| Author | Peter Norvig |
| Edition | Third Edition |
| Year | 1996 |
| Page | 1152 |

Example 2:

Let's suppose we are taking an entity, Peter. Peter is an engineer as a profession, and his age is 25, he lives in city London, and the country is England. So following is the frame representation for this:

| Slots | Filter |
|-----------------------|--------|
| Name | Peter |
| Profession | Doctor |
| Age | 25 |
| Marital status | Single |
| Weight | 78 |

Advantages of frame representation:

1. The frame knowledge representation makes the programming easier by grouping the related data.
2. The frame representation is comparably flexible and used by many applications in AI.
3. It is very easy to add slots for new attribute and relations.
4. It is easy to include default data and to search for missing values.
5. Frame representation is easy to understand and visualize.

Disadvantages of frame representation:

1. In frame system inference mechanism is not be easily processed.
2. Inference mechanism cannot be smoothly proceeded by frame representation.
3. Frame representation has a much generalized approach.

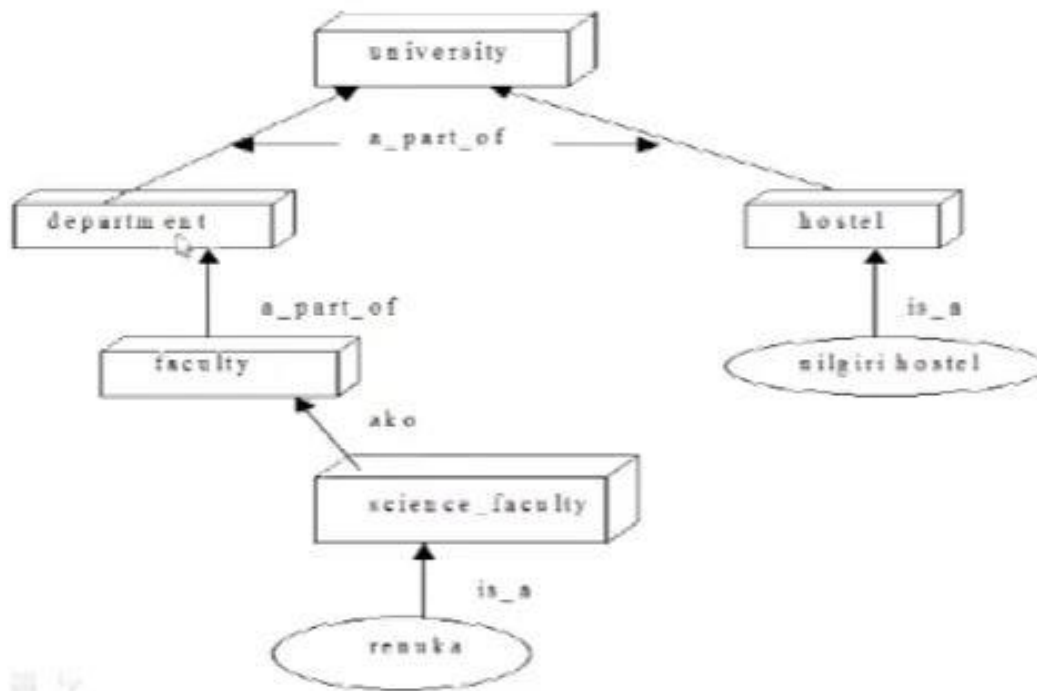
Inheritance in Frames

- Suppose we want to know nationality or phone of an instance-frame13.
- These information are not given in this frame.
- Search will start from frame13 in upward direction till we get our answer or have reached root frame.
- The frame can be easily represented in prolog by choosing predicate name as frame with two arguments.
- First argument is the name of the frame and second argument is a list of slot - facet pair.

Features of Frame Representations

- Frames can support values more naturally than semantic nets (e.g. the value 25)
- Frames can be easily implemented using object-oriented programming techniques.
- Demons allow for arbitrary functions to be embedded in a representation.
- But a price is paid in terms of efficiency, generality, and modularity !
- Inheritance can be easily controlled.

Frames Examples :-



Conceptual Dependency

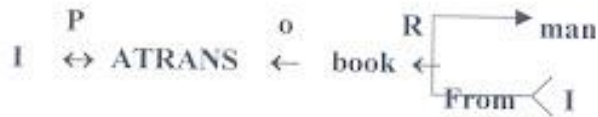
Conceptual Dependency Theory

- The CD theory defines a semantic base for knowledge representation.
- The objective was to understand natural language stories.
- The CD theory is designed for everyday actions.
- More specific domains would require a specific set of primitives.

Conceptual Dependency (CD)

This representation is used in natural language processing in order to represent the meaning of the sentences in such a way that inference can be made from the sentences. It is independent of the language in which the sentences were originally stated. CD representations of a sentence are built out of primitives, which are not words belonging to the language but are conceptual; these primitives are combined to form the meanings of the words. As an example consider the event represented by the sentence.

I gave the man a book.
 In CD from the above sentence can be represented with primitives as



Conceptual dependency theory (cont'd)

Primitive acts:

- ATRANS transfer a relationship (give)
- PTRANS transfer of physical location of an object (go)
- PROPEL apply physical force to an object (push)
- MOVE move body part by owner (kick)
- GRASP grab an object by an actor (grasp)
- INGEST ingest an object by an animal (eat)
- EXPEL expel from an animal's body (cry)
- MTRANS transfer mental information (tell)
- MBUILD mentally make new information (decide)
- CONC conceptualize or think about an idea (think)
- SPEAK produce sound (say)
- ATTEND focus sense organ (listen)

Examples with the basic conceptual dependencies

| | | |
|--|---|-------------------------------|
| 1. $PP \longleftrightarrow ACT$ | John \longleftrightarrow PTRANS | John ran. |
| 2. $PP \longleftrightarrow PA$ | John \longleftrightarrow height (>average) | John is tall. |
| 3. $PP \longleftrightarrow PP$ | John \longleftrightarrow doctor | John is a doctor. |
| 4. $PP \uparrow PA$ | boy \uparrow nice | A nice boy |
| 5. $PP \uparrow \uparrow PP$ | dog $\uparrow \uparrow$ POSS-BY John | John's dog |
| 6. $ACT \xleftarrow{o} PP$ | John \xrightarrow{p} PROPEL \xleftarrow{o} cart | John pushed the cart. |
| 7. $ACT \xleftarrow{R} \begin{matrix} PP \\ PP \end{matrix}$ | John \xrightarrow{p} ATRANS \uparrow_o book $\xleftarrow{R} \begin{matrix} John \\ Mary \end{matrix}$ | John took the book from Mary. |

