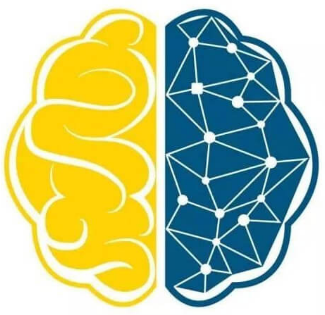
ARTIFICIAL INTELLIGENCE



MACHINE
LEARNING

By
H. Ateeq Ahmed, ***M.Tech, (Ph.D)***
Asst. Professor of CSE,
Kurnool.

NOT FOR SALE

SYLLABUS & CONTENTS

UNIT- I

FOUNDATIONS OF AI

WHAT IS AI?

Artificial Intelligence (AI) is a branch of Science which deals with helping machines finding solutions to complex problems in a more human-like fashion. This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way. A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behaviour appears. AI is generally associated with Computer Science, but it has many important links with other fields such as Maths, Psychology, Cognition, Biology and Philosophy, among many others. Our ability to combine knowledge from all these fields will ultimately benefit our progress in the quest of creating an intelligent artificial being.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavour. In this sense, it is truly a universal field.

HISTORY OF AI

The origin of artificial intelligence lies in the earliest days of machine computations. During the 1940s and 1950s, AI begins to grow with the emergence of the modern computer. Among the first researchers to attempt to build intelligent programs were Newell and Simon. Their first well known program, logic theorist, was a program that proved statements using the accepted rules of logic and a problem-solving program of their own design. By the late fifties, programs existed that could do a passable job of translating technical documents and it was seen as only a matter of extra databases and more computing power to apply the techniques to less formal, more ambiguous texts. Most problem-solving work revolved around the work of Newell, Shaw and Simon, on the general problem solver (GPS). Unfortunately, the GPS did not fulfil its promise and did not because of some simple lack of computing capacity. In the 1970’s the most important concept of AI was developed known as Expert System which exhibits as a set rules the knowledge of an expert. The application area of expert system is very large. The 1980’s saw the development of neural networks as a method learning examples.

Prof. Peter Jackson (University of Edinburgh) classified the history of AI into three periods as:

1. Classical
2. Romantic
3. Modern

1. Classical Period:

It was started from 1950. In 1956, the concept of Artificial Intelligence came into existence. During this period, the main research work carried out includes game plying, theorem proving and concept of state space approach for solving a problem.

2. Romantic Period:

It was started from the mid 1960 and continues until the mid 1970. During this period people were interested in making machine understand, that is usually mean the understanding of natural language. During this period the knowledge representation technique "semantic net" was developed.

3. Modern Period:

It was started from 1970 and continues to the present day. This period was developed to solve more complex problems. This period includes the research on both theories and practical aspects of Artificial Intelligence. This period includes the birth of concepts like Expert system, Artificial Neurons, Pattern Recognition etc. The research of the various advanced concepts of Pattern Recognition and Neural Network are still going on.

Components of AI

There are three types of components in AI

1) Hardware Components of AI

a) Pattern Matching
b) Logic Representation
c) Symbolic Processing
d) Numeric Processing
e) Problem Solving
f) Heuristic Search
g) Natural Language processing
h) Knowledge Representation
i) Expert System
j) Neural Network
k) Learning
l) Planning
m) Semantic Network

2) Software Components

a) Machine Language
b) Assembly language
c) High level Language
d) LISP Language
e) Fourth generation Language
f) Object Oriented Language
g) Distributed Language
h) Natural Language
i) Particular Problem Solving Language

3) Architectural Components

a) Uniprocessor
b) Multiprocessor
c) Special Purpose Processor
d) Array Processor
e) Vector Processor
f) Parallel Processor
g) Distributed Processor

Definition of Artificial intelligence

1. AI is the study of how to make computers do things which at the moment people do better. This is ephemeral as it refers to the current state of computer science and it excludes a major area; problems that cannot be solved well either by computers or by people at the moment.

2. AI is a field of study that encompasses computational techniques for performing tasks that apparently require intelligence when performed by humans.

3. AI is the branch of computer science that is concerned with the automation of intelligent behaviour. A I is based upon the principles of computer science namely data structures used in knowledge representation, the algorithms needed to apply that knowledge and the languages and programming techniques used in their implementation.

4. AI is the field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes.

5. AI is about generating representations and procedures that automatically or autonomously solve problems heretofore solved by humans.

6. A I is the part of computer science concerned with designing intelligent computer systems, that is, computer systems that exhibit the characteristics we associate with intelligence in human behaviour such as understanding language, learning, reasoning and solving problems.

7. A I is the study of mental faculties through the use of computational models.

8. A I is the study of the computations that make it possible to perceive, reason, and act.

9. A I is the exciting new effort to make computers think machines with minds, in the full and literal sense.

10. AI is concerned with developing computer systems that can store knowledge and effectively use the knowledge to help solve problems and accomplish tasks. This brief statement sounds a lot like one of the commonly accepted goals in the education of humans. We want students to learn (gain knowledge) and to learn to use this knowledge to help solve problems and accomplish tasks.

STRONG AND WEAK AI

There are two conceptual thoughts about AI namely the Weak AI and Strong AI. The strong AI is very much promising about the fact that the machine is almost capable of solve a complex problem like an intelligent man. They claim that a computer is much more efficient to solve the problems than some of the human experts. According to strong AI, the computer is not merely a tool in the study of mind, rather the appropriately programmed computer is really a mind. Strong AI is the supposition that some forms of artificial intelligence can truly reason and solve problems. The term strong AI was originally coined by John Searle.

In contrast, the weak AI is not so enthusiastic about the outcomes of AI and it simply says that some thinking like features can be added to computers to make them more useful tools. It says that computers to make them more useful tools. It says that computers cannot be made intelligent equal to human being, unless constructed significantly differently. They claim that computers may be similar to human experts but not equal in any cases. Generally weak AI refers to the use of software to study or accomplish specific problem solving that do not encompass the full range of human cognitive abilities. An example of weak AI would be a chess program. Weak AI programs cannot be called “intelligent” because they cannot really think.

THE STATE OF THE ART

The increasingly advanced technology provides researchers with new tools that are capable of achieving important goals, and these tools are great starting points in and of themselves. Among the achievements of recent years, the following are some specific domains:

- **Machine learning;**
- **Reinforcement learning;**
- **Deep learning;**
- **Natural language processing.**

Machine Learning (ML)

Machine learning is a subcategory of AI that often uses statistical techniques to give machines the ability to absorb data without explicitly receiving instructions to do so. This process is known as ‘training’ a ‘model’ using a learning ‘algorithm’, which progressively improves performance on a specific activity. The successes achieved in this field have encouraged researchers to push harder on the accelerator.

Reinforcement Learning (RL)

Reinforcement learning is an area of ML that is related to software agents that learn ‘goal-oriented’ behavior by trying and making mistakes in environments that provide rewards in response to the agents’ actions (called ‘Policy’) toward achieving the objectives.

This field is perhaps the one that has most captured the attention of researchers in the last decade.

Deep Learning

Also within ML, deep learning takes inspiration from the activity of neurons within the brain to learn how to recognize complex patterns through learned data. This is thanks to the use of algorithms, mainly statistical calculations. The word ‘deep’ refers to the large number of levels of neurons that ML models simultaneously, which helps acquire rich representations of data to obtain performance gains.

Natural Language Processing (NLP)

Natural language processing is the mechanism by which machines acquire the ability to analyze, understand, and manipulate textual data. 2019 was a great year for NPL with Google AI's BERT and Transformer, Allen Institute's ELMo, OpenAI's Transformer, Ruder and Howard's ULMFit, and finally, Microsoft's MT-DNN. All of these have shown that pre-taught language models can substantially improve performance on a wide variety of NLP tasks.

INTELLIGENT AGENTS

AGENTS

An AI system is composed of an agent and its environment. The agents act in their environment. The environment may contain other agents.

An **agent** is anything that can perceive its environment through **sensors** and acts upon that environment through **effectors.**

- A **human agent** has sensory organs such as eyes, ears, nose, tongue and skin parallel to the sensors, and other organs such as hands, legs, mouth, for effectors.
- A **robotic agent** replaces cameras and infrared range finders for the sensors, and various motors and actuators for effectors.
- A **software agent** has encoded bit strings as its programs and actions.

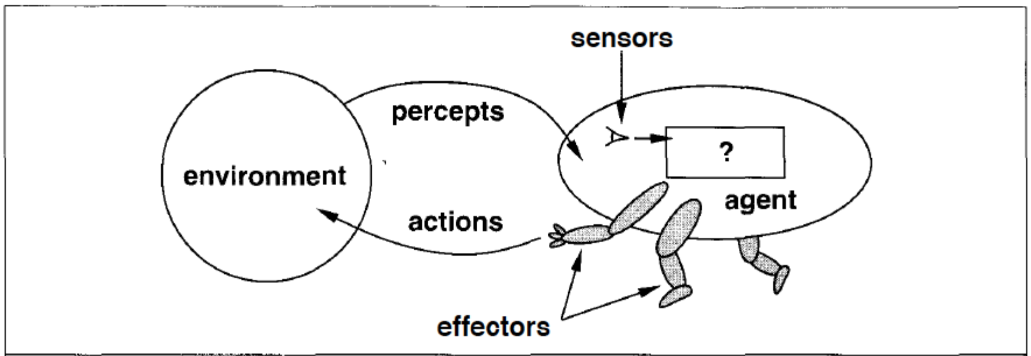


Figure 2.1 Agents interact with environments through sensors and effectors.

Agent Terminology

- **Performance Measure of Agent** − It is the criteria, which determines how successful an agent is.
- **Behavior of Agent** − It is the action that agent performs after any given sequence of percepts.
- **Percept** − It is agent's perceptual inputs at a given instance.
- **Percept Sequence** − It is the history of all that an agent has perceived till date.
- **Agent Function** − It is a map from the precept sequence to an action.

How Agents Should Act

A rational agent is one that does the right thing. Obviously, this is better than doing the wrong thing, but what does it mean? As a first approximation, we will say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding how and when to evaluate the agent's success.

We use the term performance measure for the how—the criteria that determine how successful an agent is. Obviously, there is not one fixed measure suitable for all agents. We could ask the agent for a subjective opinion of how happy it is with its own performance, but some agents would be unable to answer, and others would delude themselves. (Human agents in particular are notorious for "sour grapes"—saying they did not really want something after they are unsuccessful at getting it.) Therefore, we will insist on an objective performance measure imposed by some authority. In other words, we as outside observers establish a standard of what it means to be successful in an environment and use it to measure the performance of agents.

As an example, consider the case of an agent that is supposed to vacuum a dirty floor. A plausible performance measure would be the amount of dirt cleaned up in a single eight-hour shift. A more sophisticated performance measure would factor in the amount of electricity consumed and the amount of noise generated as well. A third performance measure might give highest marks to an agent that not only cleans the floor quietly and efficiently, but also finds time to go windsurfing at the weekend.'

The when of evaluating performance is also important. If we measured how much dirt the agent had cleaned up in the first hour of the day, we would be rewarding those agents that start fast (even if they do little or no work later on), and punishing those that work consistently. Thus, we want to measure performance over the long run, be it an eight-hour shift or a lifetime.

We need to be careful to distinguish between rationality and omniscience. An omniscient agent knows the actual outcome of its actions, and can act accordingly; but omniscience is; impossible in reality.

In summary, what is rational at any given time depends on four things:

- The performance measure that defines degree of success.
- Everything that the agent has perceived so far. We will call this complete perceptual history the percept sequence.
- What the agent knows about the environment.
- The actions that the agent can perform.

This leads to a definition of an ideal rational agent: For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

THE CONCEPT OF RATIONALITY

Rationality is nothing but status of being reasonable, sensible, and having good sense of judgment.

Rationality is concerned with expected actions and results depending upon what the agent has perceived. Performing actions with the aim of obtaining useful information is an important part of rationality.

What is Ideal Rational Agent?

An ideal rational agent is the one, which is capable of doing expected actions to maximize its performance measure, on the basis of −

- Its percept sequence
- Its built-in knowledge base

Rationality of an agent depends on the following −

- The **performance measures**, which determine the degree of success.
- Agent's **Percept Sequence** till now.
- The agent's **prior knowledge about the environment**.
- The **actions** that the agent can carry out.

A rational agent always performs right action, where the right action means the action that causes the agent to be most successful in the given percept sequence. The problem the agent solves is characterized by Performance Measure, Environment, Actuators, and Sensors (PEAS).

ENVIRONMENTS

In this section, we will see how to couple an agent to an environment. In all cases, however, the nature of the connection between them is the same: actions are done by the agent on the environment, which in turn provides percepts to the agent. First, we will describe the different types of environments and how they affect the design of agents. Then we will describe environment programs that can be used as testbeds for agent programs.

Properties of environments

Environments come in several flavors. The principal distinctions to be made are as follows:

Accessible vs. inaccessible

If an agent's sensory apparatus gives it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

Deterministic vs. nondeterministic.

If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is inaccessible, however, then it may appear to be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or nondeterministic/rom the point of view of the agent.

Episodic vs. nonepisodic

In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

Static vs. dynamic

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is semidynamic.

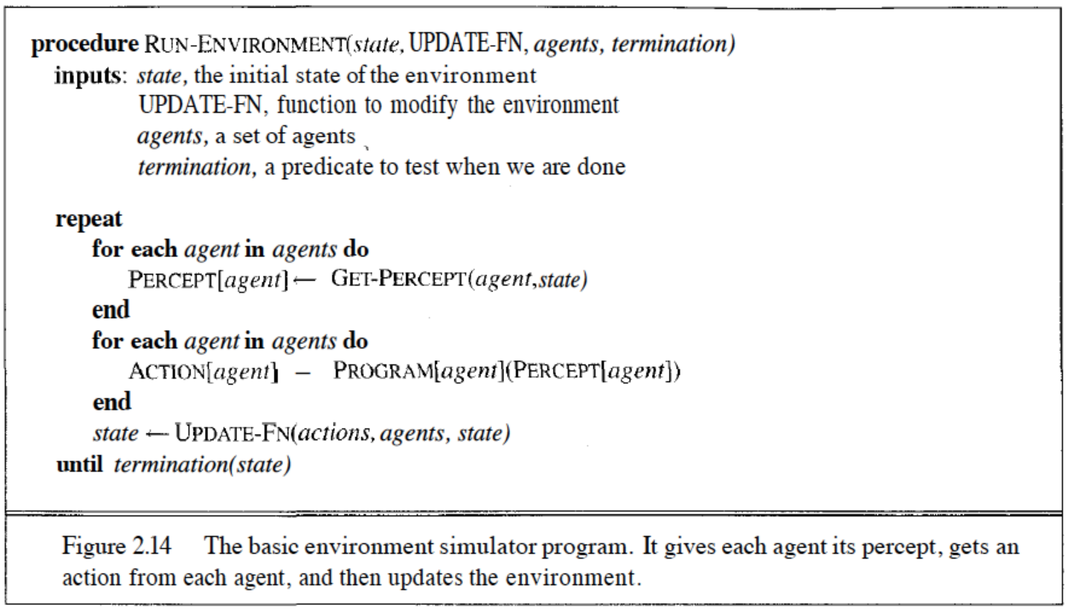
Discrete vs. continuous

If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.

We will see that different environment types require somewhat different agent programs to deal with them effectively. It will turn out, as you might expect, that the hardest case is inaccessible, nonepisodic, dynamic, and continuous.

Environment programs

The generic environment program in Figure 2.14 illustrates the basic relationship between agents and environments. In this book, we will find it convenient for many of the examples and exercises to use an environment simulator that follows this program structure. The simulator takes one or more agents as input and arranges to repeatedly give each agent the right percepts and receive back an action. The simulator then updates the environment based on the actions, and possibly other dynamic processes in the environment that are not considered to be agents (rain, for example). The environment is therefore defined by the initial state and the update function. Of course, an agent that works in a simulator ought also to work in a real environment that provides the same kinds of percepts and accepts the same kinds of actions.



```
procedure RUN-ENVIRONMENT(state, UPDATE-FN, agents, termination)
  inputs: state, the initial state of the environment
          UPDATE-FN, function to modify the environment
          agents, a set of agents
          termination, a predicate to test when we are done

  repeat
     for each agent in agents do
        PERCEPT[agent] ← GET-PERCEPT(agent,state)
     end
     for each agent in agents do
        ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
     end
     state ← UPDATE-FN(actions, agents, state)
  until termination(state)
```

Figure 2.14 The basic environment simulator program. It gives each agent its percept, gets an action from each agent, and then updates the environment.

THE NATURE OF ENVIRONMENTS

Some programs operate in the entirely **artificial environment** confined to keyboard input, database, computer file systems and character output on a screen.

In contrast, some software agents (software robots or softbots) exist in rich, unlimited softbots domains. The simulator has a **very detailed, complex environment**. The software agent needs to choose from a long array of actions in real time. A softbot designed to scan the online preferences of the customer and show interesting items to the customer works in the **real** as well as an **artificial** environment.

The most famous **artificial environment** is the **Turing Test environment**, in which one real and other artificial agents are tested on equal ground. This is a very challenging environment as it is highly difficult for a software agent to perform as well as a human.

Turing Test

- The success of an intelligent behavior of a system can be measured with Turing Test.
- Two persons and a machine to be evaluated participate in the test. Out of the two persons, one plays the role of the tester. Each of them sits in different rooms. The tester is unaware of who is machine and who is a human. He interrogates the questions by typing and sending them to both intelligences, to which he receives typed responses.
- This test aims at fooling the tester. If the tester fails to determine machine's response from the human response, then the machine is said to be intelligent.

THE STRUCTURE OF AGENTS

Agent's structure can be viewed as −

- Agent = Architecture + Agent Program
- Architecture = the machinery that an agent executes on.
- Agent Program = an implementation of an agent function.

So far, we have talked about agents by describing their behavior—the action that is performed after any given sequence of percepts. Now, we will have to bite the bullet and talk about how the insides work.

The job of AI is to design the agent program: a function that implements the agent mapping from percepts to actions. We assume this program will run on some sort of computing device, which we will call the architecture. Obviously, the program we choose has to be one that the architecture will accept and run. The architecture might be a plain computer, or it might include special-purpose hardware for certain tasks, such as processing camera images or filtering audio input. It might also include software that provides a degree of insulation between the raw computer and the agent program, so that we can program at a higher level. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the effectors as they are generated.

The relationship among agents, architectures, and programs can be summed up as follows:

agent = architecture + program

Before we design an agent program, we must have a pretty good idea of the possible percepts and actions, what goals or performance measure the agent is supposed to achieve, and what sort of environment it will operate in. These come in a wide variety. Figure 2.3 shows the basic elements for a selection of agent types.

It may come as a surprise to some readers that we include in our list of agent types programs that seem to operate in the entirely artificial environment defined by keyboard input and character output on a screen. "Surely," one might say, "this is not a real environment, is it?" In fact, what matters is not the distinction between "real" and "artificial" environments, but the complexity of the relationship among the behavior of the agent, the percept sequence generated by the environment, and the goals that the agent is supposed to achieve. Some "real" environments are actually quite simple. For example, a robot designed to inspect parts as they come by on a conveyer belt can make use of a number of simplifying assumptions: that the lighting is always just so, that the only thing on the conveyer belt will be parts of a certain kind, and that there are only two actions—accept the part or mark it as a reject.

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Figure 2.3 Examples of agent types and their PAGE descriptions.

Agent Programs

Intelligent Agents will all have the same skeleton, namely, accepting percepts from an environment and generating actions. The early versions of agent programs will have a very simple form (Figure 2.4). Each will use some internal data structures that will be updated as new percepts arrive. These data structures are operated on by the agent's decision-making procedures to generate an action choice, which is then passed to the architecture to be executed.

There are two things to note about this skeleton program. First, even though we defined the agent mapping as a function from percept sequences to actions, the agent program receives only a single percept as its input. It is up to the agent to build up the percept sequence in memory, if it so desires. In some environments, it is possible to be quite successful without storing the percept sequence, and in complex domains, it is infeasible to store the complete sequence.

```
function SKELETON-AGENT(percept) returns action
  static: memory, the agent's memory of the world

  memory — UPDATE-MEMORY(memory, percept)
  action ⟵ CHOOSE-BEST-ACTION(memory)
  memory — UPDATE-MEMORY(memory, action)
  return action
```

Figure 2.4 A skeleton agent. On each invocation, the agent's memory is updated to reflect the new percept, the best action is chosen, and the fact that the action was taken is also stored in memory. The memory persists from one invocation to the next.

Second, the goal or performance measure is not part of the skeleton program. This is because the performance measure is applied externally to judge the behavior of the agent, and it is often possible to achieve high performance without explicit knowledge of the performance measure (see, e.g., the square-root agent).

Example

At this point, it will be helpful to consider a particular environment, so that our discussion can become more concrete. Mainly because of its familiarity, and because it involves a broad range of skills, we will look at the job of designing an automated taxi driver.

We must first think about the percepts, actions, goals and environment for the taxi. They are summarized in Figure 2.6 and discussed in turn.

Agent Type	Percepts	Actions	Goals	Environment
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers

Figure 2.6 The taxi driver agent type.

The taxi will need to know where it is, what else is on the road, and how fast it is going. This information can be obtained from the percepts provided by one or more controllable TV cameras, the speedometer, and odometer. To control the vehicle properly, especially on curves, it should have an accelerometer; it will also need to know the mechanical state of the vehicle, so it will need the usual array of engine and electrical system sensors. It might have instruments that are not available to the average human driver: a satellite global positioning system (GPS) to give it accurate position information with respect to an electronic map; or infrared or sonar sensors to detect distances to other cars and obstacles. Finally, it will need a microphone or keyboard for the passengers to tell it their destination.

The actions available to a taxi driver will be more or less the same ones available to a human driver: control over the engine through the gas pedal and control over steering and braking. In addition, it will need output to a screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles.

What performance measure would we like our automated driver to aspire to? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time and/or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits. Obviously, some of these goals conflict, so there will be trade-offs involved.

Finally, were this a real project, we would need to decide what kind of driving environment the taxi will face. Should it operate on local roads, or also on freeways? Will it be in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not? Will it always be driving on the right, or might we want it to be flexible enough to drive on the left in case we want to operate taxis in Britain or Japan? Obviously, the more restricted the environment, the easier the design problem.

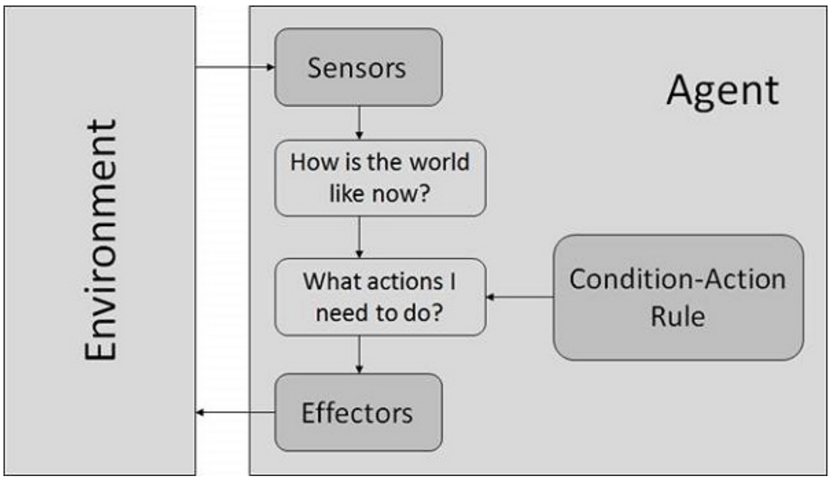
We will consider four types of agent program:

- Simple reflex agents
- Model based reflex agents (Agents that keep track of the world)
- Goal-based agents
- Utility-based agents

1. Simple Reflex Agents

- They choose actions only based on the current percept.
- They are rational only if a correct decision is made only on the basis of current precept.
- Their environment is completely observable.

Condition-Action Rule − It is a rule that maps a state (condition) to an action.



2. Model Based Reflex Agents

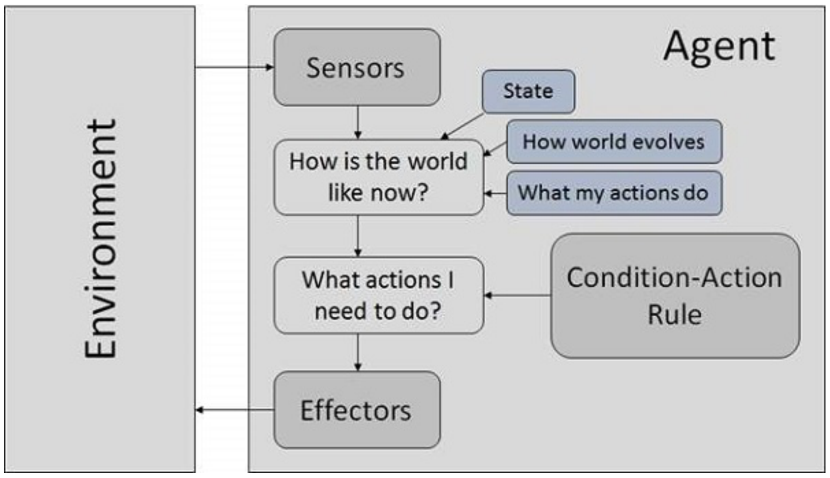
They use a model of the world to choose their actions. They maintain an internal state.

Model − knowledge about "how the things happen in the world".

Internal State − It is a representation of unobserved aspects of current state depending on percept history.

Updating the state requires the information about −

- How the world evolves.
- How the agent's actions affect the world.

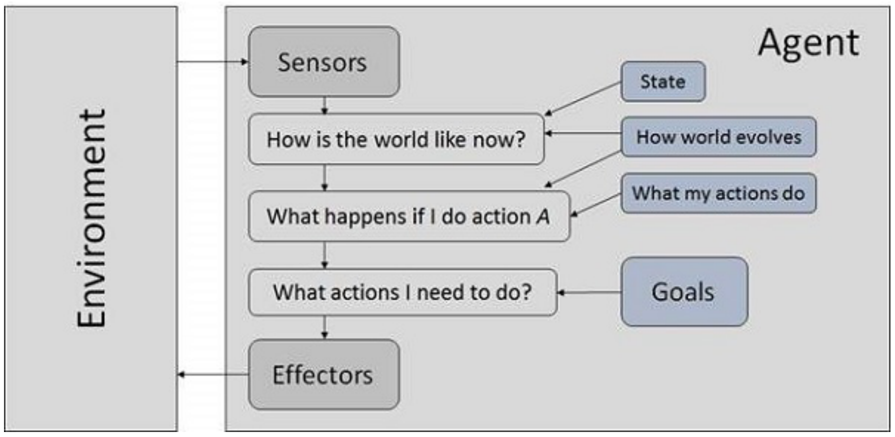


3. Goal Based Agents

They choose their actions in order to achieve goals.

Goal-based approach is more flexible than reflex agent since the knowledge supporting a decision is explicitly modeled, thereby allowing for modifications.

Goal − It is the description of desirable situations.

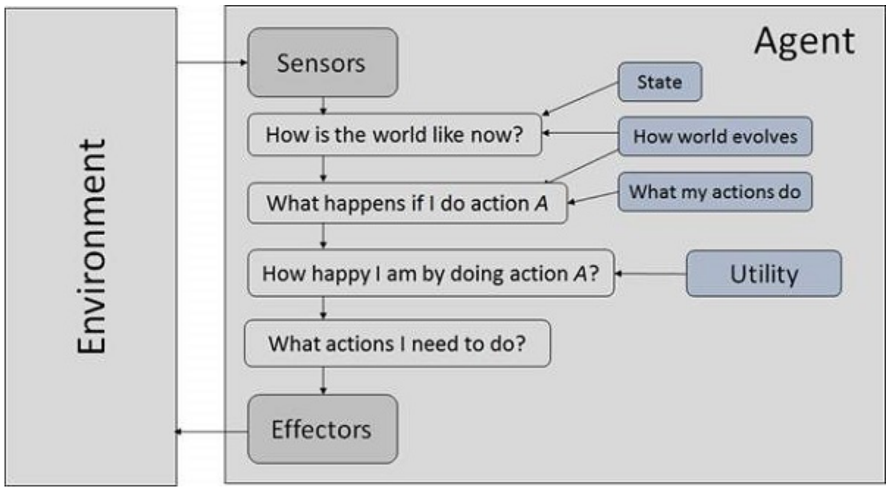

Environment
Agent
Sensors
State
How is the world like now?
How world evolves
What my actions do
What happens if I do action A
What actions I need to do?
Goals
Effectors

4. Utility Based Agents

They choose actions based on a preference (utility) for each state.

Goals are inadequate when −

- There are conflicting goals, out of which only few can be achieved.
- Goals have some uncertainty of being achieved and you need to weigh likelihood of success against the importance of a goal.


Environment
Agent
Sensors
State
How is the world like now?
How world evolves
What my actions do
What happens if I do action A
How happy I am by doing action A?
Utility
What actions I need to do?
Effectors

****

UNIT- II

SOLVING PROBLEMS BY SEARCHING

In previous chapter, we saw that simple reflex agents are unable to plan ahead. They are limited in what they can do because their actions are determined only by the current percept. Furthermore, they have no knowledge of what their actions do nor of what they are trying to achieve.

In this chapter, we describe one kind of goal-based agent called a problem-solving agent. Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states. We discuss informally how the agent can formulate an appropriate view of the problem it faces. The problem type that results from the formulation process will depend on the knowledge available to the agent: principally, whether it knows the current state and the outcomes of actions. We then define more precisely the elements that constitute a "problem" and its "solution," and give several examples to illustrate these definitions. Given precise definitions of problems, it is relatively straightforward to construct a search process for finding solutions.

PROBLEM SOLVING AGENTS

Intelligent agents are supposed to act in such a way that the environment goes through a sequence of states that maximizes the performance measure. In its full generality, this specification is difficult to translate into a successful agent design. As we mentioned in previous chapter, the task is somewhat simplified if the agent can adopt a goal and aim to satisfy it. Let us first look at how and why an agent might do this.

Goal formulation, based on the current situation, is the first step in problem solving. As well as formulating a goal, the agent may wish to decide on some other factors that affect the desirability of different ways of achieving the goal.

Problem formulation is the process of deciding what actions and states to consider, and follows goal formulation. We will discuss this process in more detail. For now, let us assume that the agent will consider actions at the level of driving from one major town to another. The states it will consider therefore correspond to being in a particular town.

In general, then, an agent with several immediate options of unknown value can decide what to do by first examining; different possible sequences of actions that lead to states of known value, and then choosing the best one. This process of looking for such a sequence is called ***search.***

A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the ***execution phase.***

Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do, and then removing that step from the sequence. Once the solution has been executed, the agent will find a new goal.

EXAMPLE PROBLEMS

The range of task environments that can be characterized by well-defined problems is vast. We can distinguish between so-called, toy problems, which are intended to illustrate or exercise various problem-solving methods, and so-called real-world problems, which tend to be more difficult and whose solutions people actually care about. In this section, we will give examples of both. By nature, toy problems can be given a concise, exact description. This means that they can be easily used by different researchers to compare the performance of algorithms. Real-world problems, on the other hand, tend not to have a single agreed-upon description, but we will attempt to give the general flavor of their formulations.

1. Toy Problems

The 8-puzzle

The 8-puzzle, an instance of which is shown in Figure 3.4, consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach the configuration shown on the right of the figure. One important trick is to notice that rather than use operators such as "move the 3 tile into the blank space," it is more sensible to have operators such as "the blank space changes places with the tile to its left." This is because there are fewer of the latter kind of operator.

This leads us to the following formulation:

- States: a state description specifies the location of each of the eight tiles in one of the nine squares. For efficiency, it is useful to include the location of the blank.
- Operators: blank moves left, right, up, or down.
- Goal test: state matches the goal configuration shown in Figure 3.4.
- Path cost: each step costs 1, so the path cost is just the length of the path.

The 8-puzzle belongs to the family of sliding-block puzzles. This general class is known to be NP-complete, so one does not expect to find methods significantly better than the search algorithms described in this chapter and the next. The 8-puzzle and its larger cousin, the 15-puzzle, are the standard test problems for new search algorithms in Al.

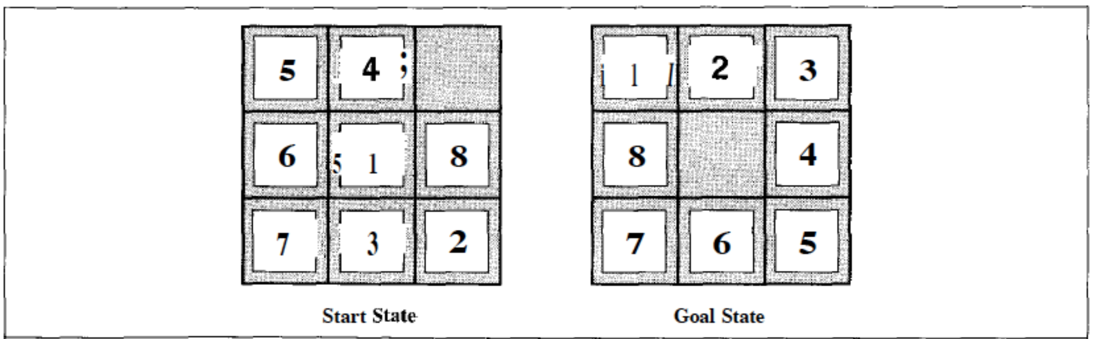


Figure 3.4 A typical instance of the 8-puzzle.

The 8-queens problem

The **eight queens puzzle** is the **problem** of placing **eight** chess **queens** on an **8**×**8** chessboard so that no two **queens** threaten each other; thus, a solution requires that no two **queens** share the same row, column, or diagonal.

The 8-queens problem can be defined as follows: Place 8 queens on an (8 by 8) chess board such that none of the queens attacks any of the others. A configuration of 8 queens on the board is shown in figure 1, but this does not represent a solution as the queen in the first column is on the same diagonal as the queen in the last column.

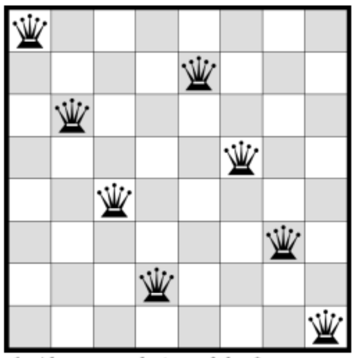


Figure 1: Almost a solution of the 8-queens problem

Although efficient special-purpose algorithms exist for this problem and the whole n queens family, it remains an interesting test problem for search algorithms. There are two main kinds of formulation. The incremental formulation involves placing queens one by one, whereas the complete-state formulation starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts; algorithms are thus compared only on search cost. Thus, we have the following goal test and path cost:

- Goal test: 8 queens on board, none attacked.
- Path cost: zero.

There are also different possible states and operators. Consider the following simple-minded formulation:

- States: any arrangement of 0 to 8 queens on board.
- Operators: add a queen to any square.

In this formulation, we have 648 possible sequences to investigate. A more sensible choice would use the fact that placing a queen where it is already attacked cannot work, because subsequent placings of other queens will not undo the attack. So we might try the following:

- States: arrangements of 0 to 8 queens with none attacked.
- Operators: place a queen in the left-most empty column such that it is not attacked by any other queen.

It is easy to see that the actions given can generate only states with no attacks; but sometimes no actions will be possible. For example, after making the first seven choices (left-to-right) in Figure 1, there is no action available in this formulation. The search process must try another choice. A quick calculation shows that there are only 2057 possible sequences to investigate. The right formulation makes a big difference to the size of the search space. Similar considerations apply for a complete-state formulation. For example, we could set the problem up as follows:

- States: arrangements of 8 queens, one in each column.
- Operators: move any attacked queen to another square in the same column.

This formulation would allow the algorithm to find a solution eventually, but it would be better to move to an unattacked square if possible.

2. Real-world problems

Route finding

We have already seen how route finding is defined in terms of specified locations and transitions! along links between them. Route-finding algorithms are used in a variety of applications, such! as routing in computer networks, automated travel advisory systems, and airline travel planning! systems. The last application is somewhat more complicated, because airline travel has a very complex path cost, in terms of money, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on. Furthermore, the actions in the problem do not have completely known outcomes: flights can be late or overbooked, connections can be missed, and fog or emergency maintenance can cause delays.

Touring and travelling salesperson problems

Consider the problem, "Visit every city in Figure 3.3 at least once, starting and ending in Bucharest." This seems very similar to route finding, because the operators still correspond to trips between adjacent cities. But for this problem, the state space must record more information. In addition to the agent's location, each state must keep track of the set of cities the agent has visited. So the initial state would be "In Bucharest; visited {Bucharest}," a typical intermediate state would be "In Vaslui; visited {Bucharest,Urziceni,Vaslui}," and the goal test would check if the agent is in Bucharest and that all 20 cities have been visited. The travelling salesperson problem (TSP) is a famous touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.The problem is NP-hard (Karp, 1972), but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for travelling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit board drills.

VLSI Layout

The design of silicon chips is one of the most complex engineering design tasks currently undertaken, and we can give only a brief sketch here. A typical VLSI chip can have as many as a million gates, and the positioning and connections of every gate are crucial to the successful operation of the chip. Computer-aided design tools are used in every phase of the process. Two of the most difficult tasks are cell layout and channel routing. These come after the components and connections of the circuit have been fixed; the purpose is to lay out the circuit on the chip so as to minimize area and connection lengths, thereby maximizing speed. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire using the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Robot navigation

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a simple, circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite.

SEARCHING FOR SOLUTIONS

We have seen how to define a problem, and how to recognize a solution. The remaining part— finding a solution—is done by a search through the state space. The idea is to maintain and extend a set of partial solution sequences. In this section, we show how to generate these sequences and how to keep track of them using suitable data structures.

Generating action sequences

To solve the route-finding problem from Arad to Bucharest, for example, we start off with just the initial state, Arad. The first step is to test if this is a goal state. Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad." Because this is ; not a goal state, we need to consider some other states. This is done by applying the operators; to the current state, thereby generating a new set of states. The process is called expanding the state. In this case, we get three new states, "in Sibiu," "in Timisoara," and "in Zerind," because there is a direct one-step route from Arad to these three cities. If there were only one possibility; we would just take it and continue. But whenever there are multiple possibilities, we must make a choice about which one to consider further.

This is the essence of search—choosing one option and putting the others aside for later, in ' case the first choice does not lead to a solution. Suppose we choose Zerind. We check to see if it is a goal state (it is not), and then expand it to get "in Arad" and "in Oradea." We can then choose any of these two, or go back and choose Sibiu or Timisoara. We continue choosing, testing, and expanding until a solution is found, or until there are no more states to be expanded. The choice of which state to expand first is determined by the search strategy.

It is helpful to think of the search process as building up a search tree that is superimposed over the state space. The root of the search tree is a search node corresponding to the initial state. The leaf nodes of the tree correspond to states that do not have successors in the tree, either because they have not been expanded yet, or because they were expanded, but generated the empty set. At each step, the search algorithm chooses one leaf node to expand. Figure 3.8 shows some of the expansions in the search tree for route finding from Arad to Bucharest. The general search algorithm is described informally in Figure 3.9.

It is important to distinguish between the state space and the search tree. For the route finding problem, there are only 20 states in the state space, one for each city. But there are an infinite number of paths in this state space, so the search tree has an infinite number of nodes. For example, in Figure 3.8, the branch Arad-Sibiu-Arad continues Arad-Sibiu-Arad-Sibiu-Arad, and so on, indefinitely. Obviously, a good search algorithm avoids following such paths.

Data structures for search trees

There are many ways to represent nodes, but in this chapter, we will assume a node is a data structure with five components:

- the state in the state space to which the node corresponds;
- the node in the search tree that generated this node (this is called the parent node);
- the operator that was applied to generate the node;
- the number of nodes on the path from the root to this node (the depth of the node);
- the path cost of the path from the initial state to the node.

The node data type is thus:
datatype node
    components: STATE, PARENT-NODE, OPERATOR, DEPTH, PATH-COST

It is important to remember the distinction between nodes and states. A node is a bookkeeping data structure used to represent the search tree for a particular problem instance as generated by a particular algorithm. A state represents a configuration (or set of configurations) of the world. Thus, nodes have depths and parents, whereas states do not. (Furthermore, it is quite possible for two different nodes to contain the same state, if that state is generated via two different sequences of actions.) The EXPAND function is responsible for calculating each of the components of the nodes it generates.

SEARCH STRATEGIES

Search Algorithm 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.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

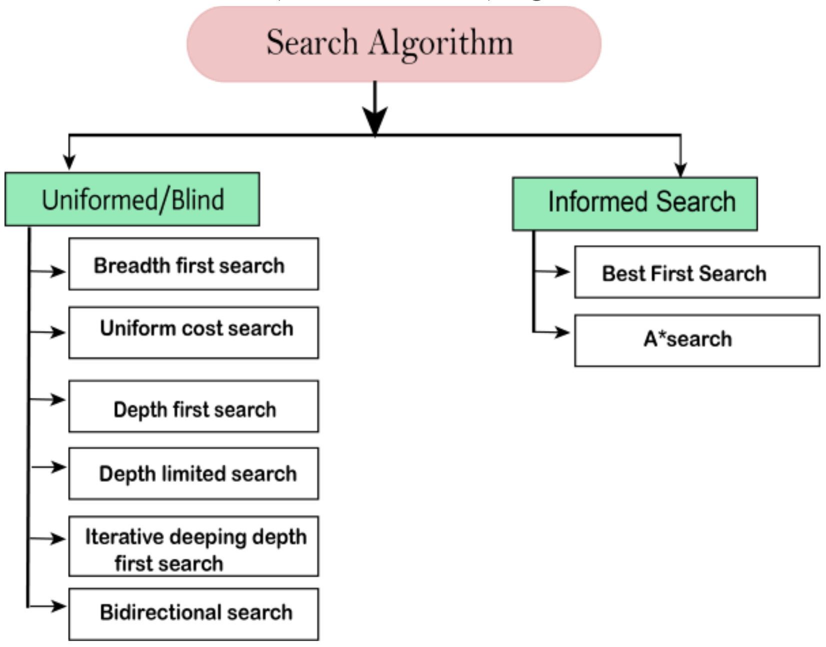
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



1. UNINFORMED SEARCH (Blind Search):

The term means that they have no information about the number of steps or the path cost from the current state to the goal—all they can do is distinguish a goal state from a nongoal state. 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
- Iterative deepening depth-first search
- Bidirectional Search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

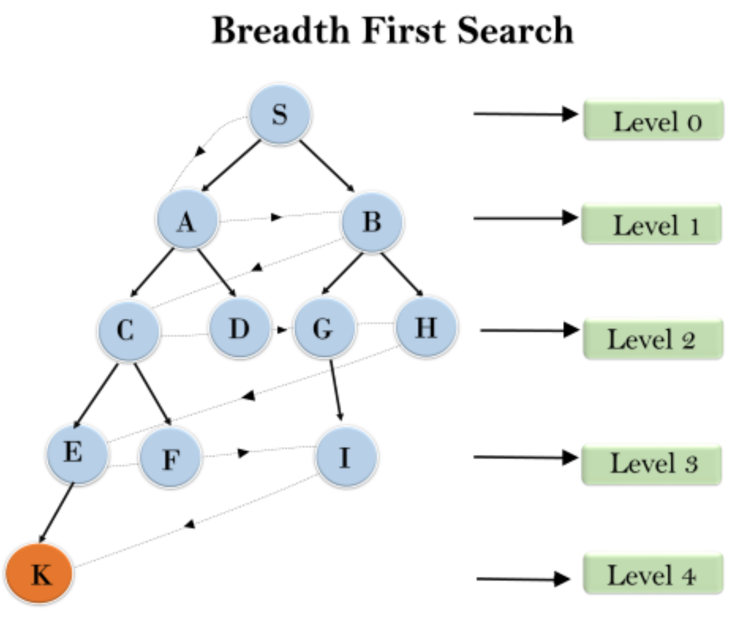
Disadvantages:

- t requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity:

Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.......+ b^d= O(b^d)$

Space Complexity:
Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
Completeness:
BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
Optimality:
BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

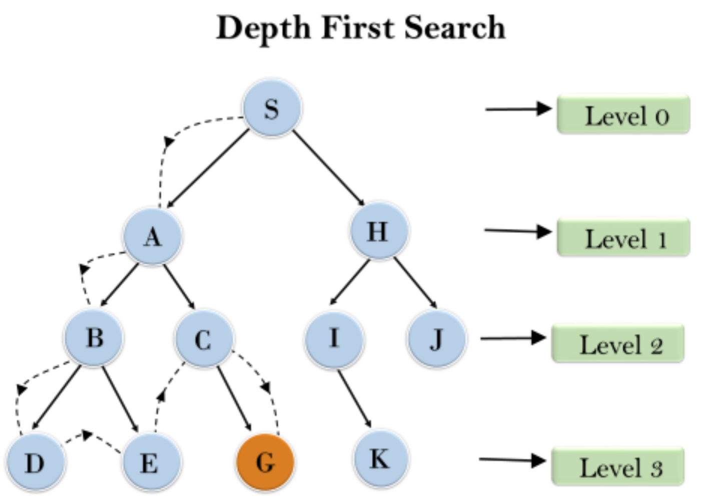
Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example
In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
Root node--->Left node ----> right node.



It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

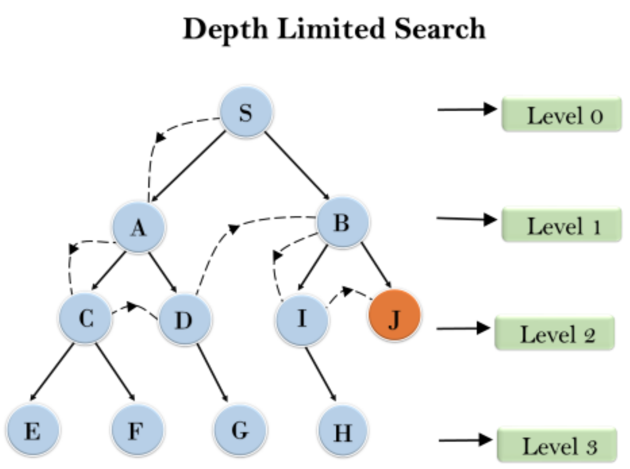
Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
$T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)$
Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)
Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:
A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
Depth-limited search can be terminated with two Conditions of failure:
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

Advantages:
Depth-limited search is Memory efficient.

Disadvantages:
- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.
Time Complexity: Time complexity of DLS algorithm is **O(b$^{\ell}$)**.
Space Complexity: Space complexity of DLS algorithm is O(**b×ℓ)**.
Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

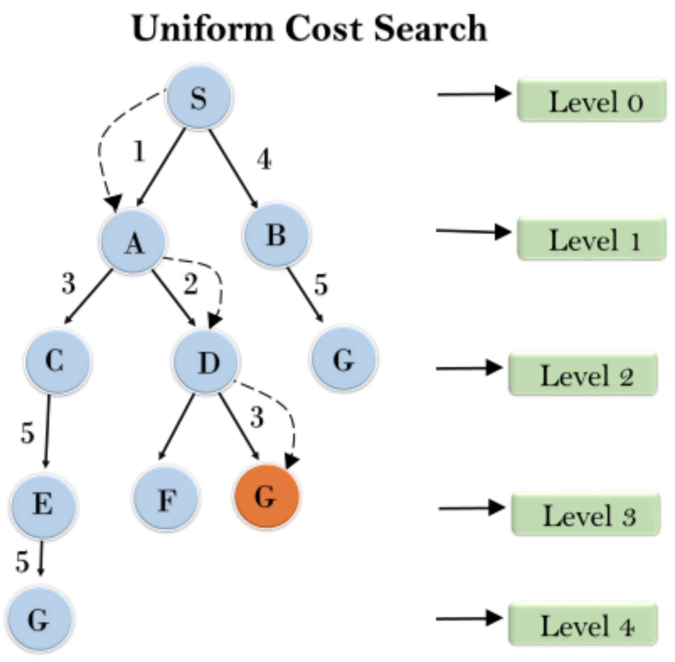
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.
-

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example



Completeness:
Uniform-cost search is complete, such as if there is a solution, UCS will find it.
Time Complexity:
Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
Hence, the worst-case time complexity of Uniform-cost search is**$O(b^{1 + [C^*/\varepsilon]})$**/.
Space Complexity:
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**.
Optimal:
Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

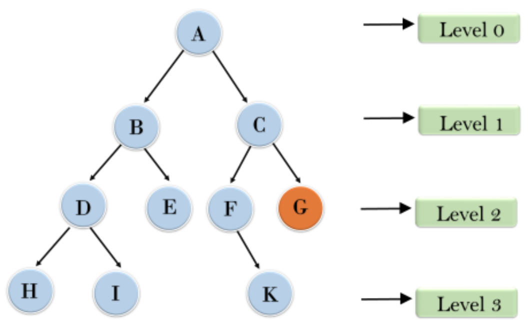
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example
Following tree structure is showing the iterative deepening depth-first search.
IDDFS algorithm performs various iterations until it does not find the goal node.
The iteration performed by the algorithm is given as:

Iterative deepening depth first search



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

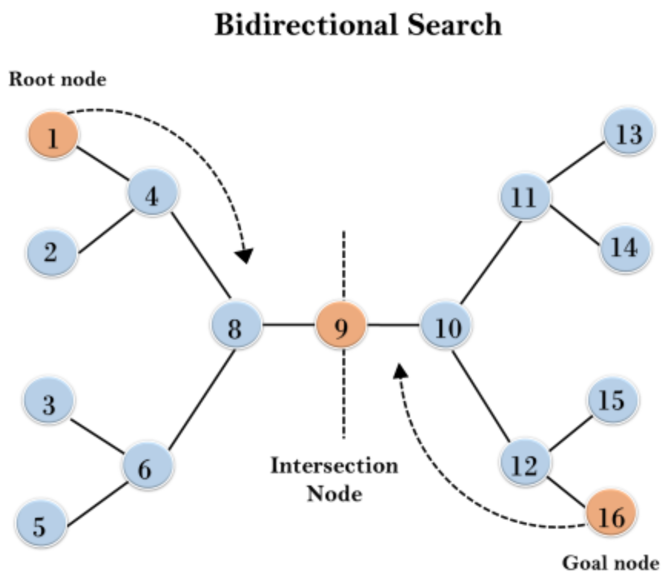
Completeness:
This algorithm is complete is if the branching factor is finite.
Time Complexity:
Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.
Space Complexity:
The space complexity of IDDFS will be **O(bd)**.
Optimal:
IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm
Bidirectional search algorithm runs two simultaneous searches, one form 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.

Advantages:
- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:
- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

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.



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.

2. INFORMED SEARCH (Heuristic 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.

Heuristics function:
Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

1. $h(n) <= h^*(n)$

Here h(n) is heuristic cost, and h*(n) is the estimated cost.

Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1. Best-first Search Algorithm (Greedy Search):

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.

f(n)= g(n).

Were, 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:

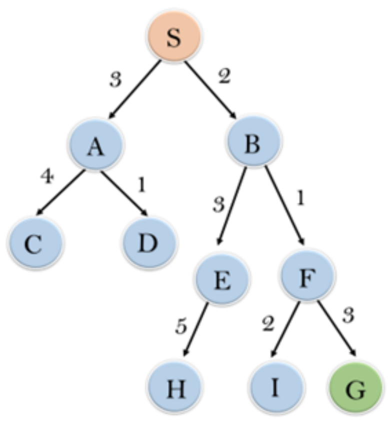
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

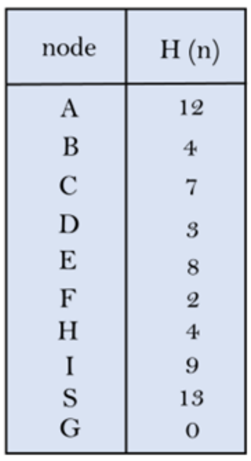
Disadvantages:

- 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

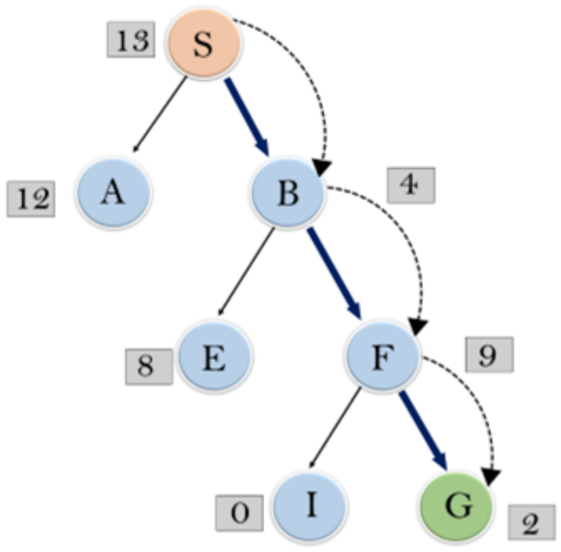
Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

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]

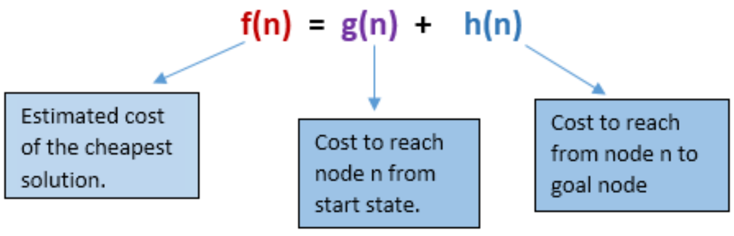
: Open [I, E, A], Closed [S, B, F, G]

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

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

2. A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).
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**.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

Algorithm of A* search:
Step1: Place the starting node in the OPEN list.
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
Step 6: Return to **Step 2**.

Advantages:

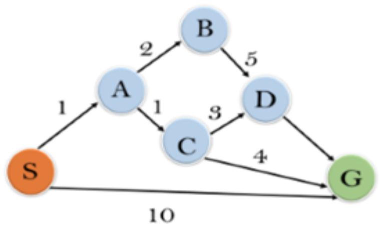
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

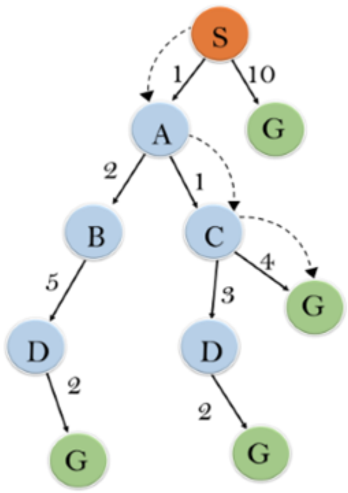
Example

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

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)**

****

UNIT- III

KNOWLEDGE REPRESENTATION

Humans are best at understanding, reasoning, and interpreting knowledge. Human knows things, which is knowledge and as per their knowledge they perform various actions in the real world. **But how machines do all these things comes under knowledge representation and reasoning**. Hence we can describe Knowledge representation as following:

- 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).

Knowledge: Knowledge is awareness or familiarity gained by experiences of facts, data, and situations. Following are the types of knowledge in artificial intelligence:

ONTOLOGICAL ENGINEERING

Ontology engineering is a set of tasks related to the development of **ontologies** for a particular domain. ... One of the approaches for the formal conceptualization of represented knowledge domains is the use of machine-interpretable **ontologies**, which provide structured data in, or based on, RDF, RDFS, and OWL.

The process of building a knowledge base is called ***knowledge engineering.*** A knowledge engineer is someone who investigates a particular domain, determines what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. Often, the knowledge engineer is trained in representation but is not an expert in the domain at hand, be it circuit design, space station mission scheduling, or whatever. The knowledge engineer will usually interview the real experts to become educated about the domain and to elicit the required knowledge, in a process called ***knowledge acquisition.*** This occurs prior to, or interleaved with, the process of creating formal representations. In this chapter, we will use domains that should already be fairly familiar, so that we can concentrate on the representational issues involved.

One does not become a proficient knowledge engineer just by studying the syntax and semantics of a representation language. It takes practice and exposure to lots of examples before one can develop a good style in any language, be it a language for programming, reasoning, or communicating. Sections 8.1 and 8.2 discuss the principles and pitfalls of knowledge engineering. We then show how to represent knowledge in the fairly narrow domain of electronic circuits in Section 8.3. A number of narrow domains can be tackled by similar techniques, but domains such as shopping in a supermarket seem to require much more general representations. In Section 8.4, we discuss ways to represent time, change, objects, substances, events, actions, money, measures, and so on. These are important because they show up in one form or another in every domain. Representing these very general concepts is sometimes called ***ontological engineering.***

CATEGORIES AND OBJECTS

The organization of objects into categories is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, much of reasoning takes place at the level of categories. For example, a shopper might have the goal of buying a cantaloupe, rather than a particular cantaloupe such as Cantaloupe, Categories also serve to make predictions about objects once they are classified. One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green, mottled skin, large size, and ovoid shape, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

There are two main choices for representing categories in first-order logic. The first we have already seen: categories are represented by unary predicates. The predicate symbol Tomato, for example, represents the unary relation that is true only for objects that are tomatoes, and Tomato(x) means that x is a tomato.

The second choice is to reify the category. Reification is the process of turning a predicate or function into an object in the language. We will see several examples of reification in this chapter. In this case, we use Tomatoes as a constant symbol referring to the object that is the set of all tomatoes. We use x G Tomatoes to say that x is a tomato. Reified categories allow us to make assertions about the category itself, rather than about members of the category. For example, we can say Population(Humans)=5,000,000,000, even though there is no individual human with a population of five billion.

Categories perform one more important role: they serve to organize and simplify the; knowledge base through inheritance. If we say that all instances of the category Food are edible,: and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit, then we know '. that every apple is edible. We say that the individual apples inherit the property of edibility, in this case from their membership in the Food category.

Subclass relations organize categories into a taxonomy or taxonomic hierarchy. Taxonomies have been used explicitly for centuries in technical fields. For example, systematic; biology aims to provide a taxonomy of all living and extinct species; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products. Taxonomies are also an important aspect of general common sense knowledge, as we will see in our investigations that follow.

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members:

- An object is a member of a category. For example:
  $\mathit{Tomato}_{12} \mathrm{G}\ \mathit{Tomatoes}$
- A category is a subclass of another category. For example:
  $\mathit{Tomatoes}\ \mathrm{C}\ \mathit{Fruit}$
- All members of a category have some properties. For example:
  $\forall x\ \ x\,\mathrm{G}\ \mathit{Tomatoes} \Rightarrow \mathit{Red}(x)\ \mathrm{A}\ \mathit{Round}(x)$
- Members of a category can be recognized by some properties. For example:
  $\forall x\ \ \mathit{Red}(\mathit{Interior}(x))\mathrm{A}\ \ \mathit{Green}(\mathit{Exterior}(x))\mathrm{A}\ x \in \mathit{Melons} \Rightarrow x \in \mathit{Watermelons}$
- A category as a whole has some properties. For example:
  $\mathit{Tomatoes}\ \mathrm{G}\ \mathit{DomesticatedSpecies}$

Notice that because Tomatoes is a category, and is a member of DomesticatedSpecies, then DomesticatedSpecies must be a category of categories. One can even have categories of categories of categories, but they are not much use.

EVENTS

Situation calculus is perfect for the vacuum world, the wumpus world, or any world in which a single agent takes discrete actions. Unfortunately, situation calculus has two problems that limit its applicability. First, situations are instantaneous points in time, which are not very useful for describing the gradual growth of a kitten into a cat, the flow of electrons along a wire, or any other process where change occurs continuously over time. Second, situation calculus works best when only one action happens at a time. When there are multiple agents in the world, or when the world can change spontaneously, situation calculus begins to break down. It is possible to prop it back up for a while by defining composite actions, as in Exercise 7.12. If there are actions that have different durations, or whose effects depend on duration, then situation calculus in its intended form cannot be used at all.

Because of these limitations, we now turn to a different approach toward representing change, which we call the event calculus, although the name is not standard. Event calculus is rather like a continuous version of the situation-calculus "movie" shown in Figure 7.3. We think of a particular universe as having both a "spatial" and a temporal dimension. The "spatial" dimension ranges over all of the objects in an instantaneous "snapshot" or "cross-section" of the universe. The temporal dimension ranges over time. An event is, informally, just a "chunk" of this universe with both temporal and spatial extent. Figure 8.3 gives the general idea.

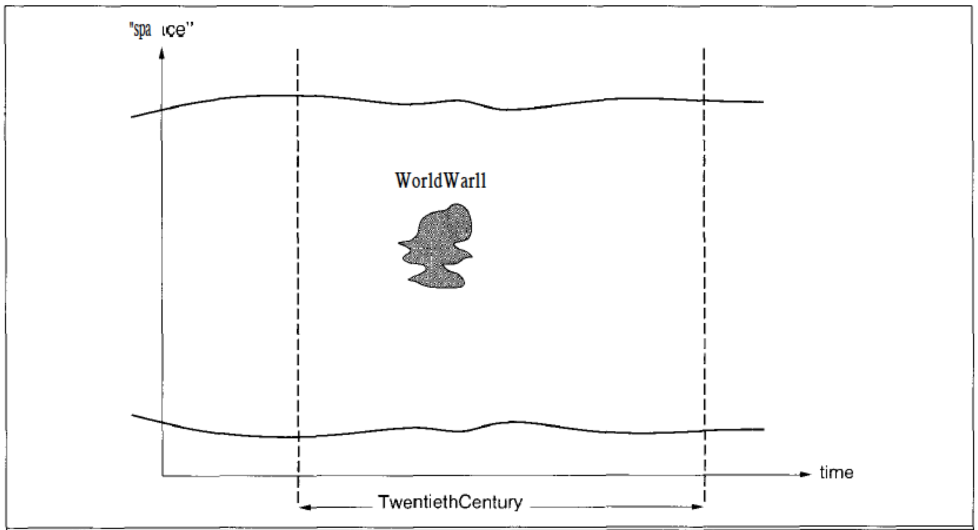


Figure 8.3 The major entities in event calculus. *Intervals* such as the *TwentiethCentury* contain as subevents all of the events occurring within a given time period. Ordinary events such as *WorldWarII* have temporal and "spatial" extent.

Let us look at an example: World War II, referred to by the symbol *WorldWarII*. World War II has parts that we refer to as **subevents**:[9]

SubEvent(BattleOfBritain, WorldWarII)

Similarly, World War II is a subevent of the twentieth century:

SubEvent(WorldWarII, TwentiethCentury)

The twentieth century is a special kind of event called an interval. An interval is an event that includes as subevents all events occurring in a given time period. Intervals are therefore entire temporal sections of the universe, as the figure illustrates. In situation calculus, a given fact is true in a particular situation. In event calculus, a given event occurs during a particular interval. The previous SubEvent sentences are examples of this kind of statement.

MENTAL EVENTS AND MENTAL OBJECTS

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge about beliefs or deduction. For single-agent domains, knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, if one knows that one does not know anything about Romanian geography, then one need not expend enormous computational effort trying to calculate the shortest path from Arad to Bucharest. In 'multiagent domains, it becomes important for an agent to reason about the mental processes of the other agents. Suppose a shopper in a supermarket has the goal of buying some anchovies.

The agent deduces that a good plan is to go where the anchovies are, pick some up, and bring them to the checkout stand. A key step is for the shopper to realize that it cannot execute this plan until it knows where the anchovies are, and that it can come to know where they are by asking someone. The shopper should also deduce that it is better to ask a store employee than another customer, because the employee is more likely to know the answer. To do this kind of deduction, an agent needs to have a model of what other agents know, as well as some knowledge of its own knowledge, lack of knowledge, and inference procedures. In effect, we want to have a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects.

The model should be faithful, but it does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction, nor do we have to predict what neurons will fire when an animal is faced with a particular visual stimulus. But we do want an abstract model that says that if a logical agent believes P V Q and it learns not(symbol) P, then it should come to believe Q.

The first step is to ask how mental objects are represented. That is, if we have a relation Believes(Agent,x), what kind of thing is xl First of all, its clear that x cannot be a logical sentence. If Flies(Superman) is a logical sentence, we can't say Believes(Agent, Flies(Superman)) because only terms (not sentences) can be arguments of relations. But if Flies(Superman) is reified as a fluent, then it is a candidate for being a mental object, and Believes can be a relation that takes an agent and a prepositional fluent that the agent believes in. We could define other relations such as Knows and Wants to express other relationships between agents and propositions. Relations of this kind are called propositional attitudes.

This appears to give us what we want: the ability for an agent to reason about the beliefs of agents. Unfortunately, there is a problem with this approach. If Clark and Superman are one and the same (i.e., Clark = Superman) then Clark flying and Superman flying are one and the same event. Thus, if the object of propositional attitudes are reified events, we must conclude that if Lois believes that Superman can fly, she also believes that Clark can fly, even if she doesn't believe that Clark is Superman. That is,

$$(Superman = Clark) \models$$
$$(Believes(Lois, Flies(Superman)) \Leftrightarrow Believes(Lois, Flies(Clark)))$$

There is a sense in which this is right: Lois does believe of a certain person, who happens to be called Clark sometimes, that that person can fly. But there is another sense in which this is wrong: if you asked Lois "Can Clark fly?" she would certainly say no. Reified objects and events work fine for the first sense of Believes, but for the second sense we need to reify descriptions of those objects and events, so that Clark and Superman can be different descriptions (even though they refer to the same object).

Technically, the property of being able to freely substitute a term for an equal term is called referential transparency. In first-order logic, every relation is referentially transparent. We would like to define Believes (and the other propositional attitudes) as relations whose second argument is referentially opaque—that is, one cannot substitute an equal term for the second argument without changing the meaning.

REASONING SYSTEMS FOR CATEGORIES

Categories are the primary building blocks of any large-scale knowledge representation scheme. Categories are the primary building blocks of any large-scale knowledge representation scheme. This topic describes systems specially designed for organizing and reasoning with category. This topic describes systems specially designed for organizing and reasoning with category. There are two types of Reasoning Systems There are two types of Reasoning Systems Semantic networks and description logic. Semantic networks and description logic.

Types of Reasoning Systems Semantic networks. Semantic networks provides graphical aids for visualizing a knowledge base in patterns of interconnected nodes and arcs. Semantic networks provides graphical aids for visualizing a knowledge base in patterns of interconnected nodes and arcs. It has an efficient algorithms for inferring of and object on the basis of its category membership. It has an efficient algorithms for inferring of and object on the basis of its category membership. In 1909 Charles Peirce proposed a graphical notation of nodes and arcs called existential graphs that is a visual notation for logical expressions In 1909 Charles Peirce proposed a graphical notation of nodes and arcs called existential graphs that is a visual notation for logical expressions

Semantic networks. A typical graphical notation displays objects or categories names in oval or boxes and connects them with labeled arcs. A typical graphical notation displays objects or categories names in oval or boxes and connects them with labeled arcs. For example a MemberOf link between Mary and female persons corresponds to the logical assertion Mary E (element) of female person. For example a MemberOf link between Mary and female persons corresponds to the logical assertion Mary E (element) of female person. We can connect categories using SubsetOf of links We can connect categories using SubsetOf of links A single box used to assert properties of every member of a category. A single box used to assert properties of every member of a category.

Semantic networks Semantic network notation makes it very convenient to perform inheritance reasoning. Semantic network notation makes it very convenient to perform inheritance reasoning. For example by virtue of being a person marry inherits the property of having two legs. For example by virtue of being a person marry inherits the property of having two legs. The simplicity and efficiency of this inference mechanism, compared with logical theorem has been one of the main attractions of semantic networks. The simplicity and efficiency of this inference mechanism, compared with logical theorem has been one of the main attractions of semantic networks. Inheritance becomes complicated when an object belong to more than one category Inheritance becomes complicated when an object belong to more than one category And also when a category can be a subset of more than one other category which is called multiple inheritance. And also when a category can be a subset of more than one other category which is called multiple inheritance.

Semantic networks multiple inheritance can cause the algorithm to find two or more conflicting values answering a query. multiple inheritance can cause the algorithm to find two or more conflicting values answering a query. For this reason multiple inheritance is banned in some object oriented programming like Java. For this reason multiple inheritance is banned in some object oriented programming like Java. Another form of inference is the use of inverse links. Another form of inference is the use of inverse links. For example HasSister is the inverse of SisterOf. For example HasSister is the inverse of SisterOf. Its ok to have inverse links as long as they are made into objects in their own right(Its ok to have inverse links as long as they are made into objects in their own right(

Semantic networks Semantic network provide direct indexing only for objects, categories and the links emanating from them Semantic network provide direct indexing only for objects, categories and the links emanating from them The drawback is that links between bubbles represent only one binary relation. The drawback is that links between bubbles represent only one binary relation.

Description logics They are logics notations that are designed to make it easier to describe definition and properties of categories. They are logics notations that are designed to make it easier to describe definition and properties of categories. It evolved from semantic network It evolved from semantic network The principal inference task for description logic are checking if one category is a subset of another by comparing their definition. The principal inference task for description logic are checking if one category is a subset of another by comparing their definition. By checking whether the membership criteria are logically certifiable.

REASONING WITH DEFAULT INFORMATION

Commonsense reasoning is often said to involve "jumping to conclusions." For example, when one sees a car parked on the street, one would normally be willing to accept that it has four wheels even though only three are visible. (If you feel that the existence of the fourth wheel is dubious, consider also the question as to whether the three visible wheels are real or merely cardboard facsimiles.) Probability theory can certainly provide a conclusion that the fourth wheel exists with high probability. On the other hand, introspection suggests that the possibility of the car not having four wheels does not even arise unless some new evidence presents itself. Thus, it seems that the four-wheel conclusion is reached by default, in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit nonmonotonicity, because the set of beliefs does not grow monotonically over time as new evidence arrives. First-order logic, on the other hand, exhibits strict monotonicity.

Reasoning schemes such as default logic (Reiter, 1980), nonmonotonic logic (McDermott and Doyle, 1980) and circumscription (McCarthy, 1980) are designed to handle reasoning with default rules and retraction of beliefs. Although the technical details of these systems are quite different, they share a number of problematic issues that arise with default reasoning:

• What is the semantic status of default rules? If "Cars have four wheels" is false, what does it mean to have it in one's knowledge base? What is a good set of default rules to have? Without a good answer to these questions, default reasoning systems will be nonmodular, and it will be hard to develop a good knowledge engineering methodology.

• What happens when the evidence matches the premises of two default rules with conflicting conclusions? We saw examples of this in the discussion of multiple inheritance in Section 10.6. In some schemes, one can express priorities between rules so that one rule takes precedence. Specificity preference is a commonly used form of priority in which a special-case rule takes precedence over the general case. For example, "Three-wheeled cars have three wheels" takes precedence over "Cars have four wheels."

• Sometimes a system may draw a number of conclusions on the basis of a belief that is later retracted. How can a system keep track of which conclusions need to be retracted as a result? Conclusions that have multiple justifications, only some of which have been abandoned, should be retained; whereas those with no remaining justifications should be dropped. These problems have been addressed by truth maintenance systems, which are discussed in Section 10.8.

• How can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning. Decisions often involve trade-offs, and one therefore needs to compare the strength of belief in the outcomes of different actions. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as "threshold probability" statements. For example, the default rule "My brakes are always OK" really means "The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them." When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence to suggest that the brakes are faulty.

To date, no default reasoning system has successfully addressed all of these issues. Furthermore, most systems are formally undecidable, and very slow in practice. There have been several attempts to subsume default reasoning in a probabilistic system, using the idea that a default rule is basically a conditional probability of 1 — f. For reasons already mentioned, such an approach is likely to require a full integration of decision making before it fully captures the desirable features of default reasoning.

THE INTERNET SHOPPING WORLD

Artificial Intelligence (AI) is changing the way you shop online. Specifically, it’s driving a change in two areas – search and customer service.
Advancements in AI like deep learning and machine leaning combined with the rise in mobile commerce are now being put to use by retailers and e-commerce sites such as Amazon to improve the lives of online consumers. Both are software that learn to perform complex tasks without active supervision by humans.

How search is changing

Traditional search engines such as Google have been using algorithms that pick on a string of keywords to throw up results that they think are closest to what searchers are looking for. While this approach worked for those looking for news items and some such, it was not really ideal for online shoppers. With advancements in AI, online search has got that much more “cleverer”, with even “visual search” gaining grounds with retailers.

Retail companies are using a combination of analytics, machine learning, deep learning, computer vision and facial recognition to compete in online business. The latter, for example, is the same technology that drives self-driving cars. Amazon is already using these technologies in its new Amazon Go grocery stores. For now, it’s deployed this in one store in Seattle, but expansion will follow suit.

One of the biggest examples of visual search is ‘Pinterest Lens’. Launched late 2016, this service allows consumers to point their mobile phone cameras at any commodity, and then conduct an online visual search. Lens, thus, goes beyond words to include images from the real world.
How does it work? While image-detection programs identify an object, visual search digs up images similar to that object.

The other advancement in online search uses Neuro-Linguistic Programming (NLP). This is a science that looks at the way in which human beings think (Neuro), the language patterns they use (Linguistic) and behaviours (Programming), and how these interact to effect human beings as individuals, both positive or negative.

Some retailers have started applying the NLP approach, which is based on AI enabling computer programs to understand human speech, to search, allowing shoppers to search for items in a manner which is similar to asking a store help in person for the same.

There are some retail companies that have claimed an amount of success in sales after deployment of advances in search. Well-known American luxury department store Neiman Marcus, for example, has increased app usage and customer engagement after implementing visual search. Other retail brands that are successfully using visual search include Shoes.com, Nordstrom, and Urban Outfitters.

You are being serviced by a chatbot

A recent report by Juniper Research found that chatbots would "redefine" the customer service industry, and forecast that the technology would save over US $8 billion in operational costs by 2022. Another study by IBM showed that 65% of millennials preferred interacting with bots instead of talking to real human beings. Needless to say, the bots have truly arrived.

Bots have become popular with retailers because of the increasing reliance on mobile messaging apps and the need for 24/7 customer service as business goes global. Brands are using chatbots to offer recommendations and services, and even to automate the purchase process.

The most popular chatbot platforms are Facebook Messenger, WhatsApp, WeChat, and Slack. Top brands like Uber and Burberry have successfully deployed chatbots to acquire new customers, while retaining the old ones.

****

UNIT- IV

LEARNING FROM EXAMPLES

Learning involves generalization from experience. Computer system is said to have learning if it is able to not only do the "repetition of same task" more effectively, but also the similar tasks of the related domain. Learning is possible due to some factors like the skill refinement and knowledge acquisition. Skill refinement refers to the situation of improving the skill by performing the same task again and again. If machines are able to improve their skills with the handling of task, they can be said having skill of learning.

What is learning?

- According to **Herbert Simon**, learning denotes changes in a system that enable a system to do the same task more efficiently the next time.

Why do we require machine learning?

- Machine learning plays an important role in improving and understanding the efficiency of human learning.
- Machine learning is used to discover a new things not known to many human beings.

FORMS OF LEARNING

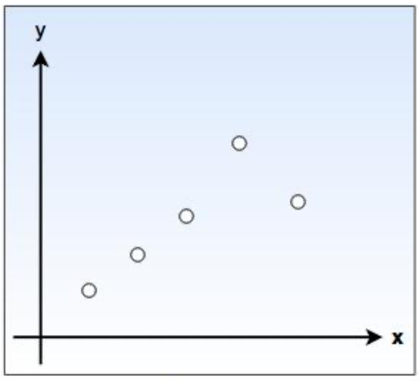
1. Rote learning

- Rote learning is possible on the basis of memorization.
- This technique mainly focuses on memorization by avoiding the inner complexities. So, it becomes possible for the learner to recall the stored knowledge.
  For example: When a learner learns a poem or song by reciting or repeating it, without knowing the actual meaning of the poem or song.

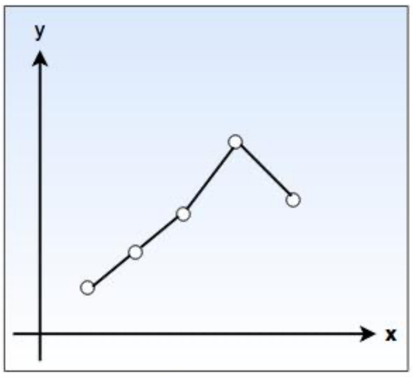
2. Induction learning (Learning by example).

- Induction learning is carried out on the basis of supervised learning.
- In this learning process, a general rule is induced by the system from a set of observed instance.
- However, class definitions can be constructed with the help of a classification method.
  For Example:
  Consider that **'*f*'** is the target function and example is a pair (x f(x)), where 'x' is input and f(x) is the output function applied to 'x'.
  Given problem: Find hypothesis h such as $h \approx f$

- So, in the following fig-a, points (x,y) are given in plane so that $y = f(x)$, and the task is to find a function h(x) that fits the point well.

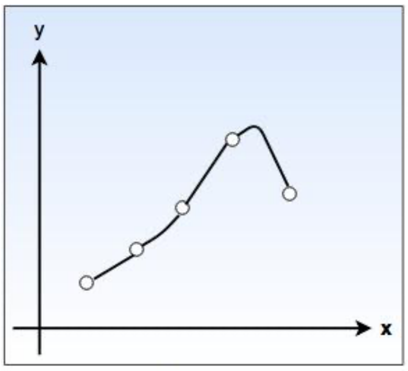


Fig(a)



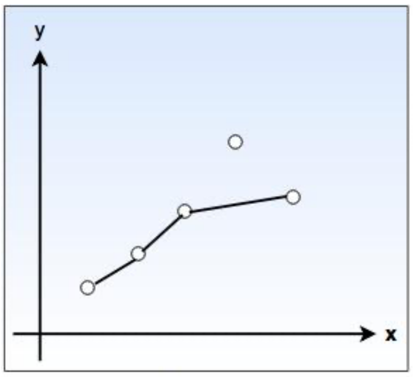
Fig(b)

- In fig-b, a piecewise-linear 'h' function is given, while the fig-c shows more complicated 'h' function.



Fig(c)

- Both the functions agree with the example points, but differ with the values of 'y' assigned to other x inputs.



Fig(d)

- As shown in fig.(d), we have a function that apparently ignores one of the example points, but fits others with a simple function. The true/ is unknown, so there are many choices for h, but without further knowledge, we have no way to prefer (b), (c), or (d).

3. Learning by taking advice

- This type is the easiest and simple way of learning.
- In this type of learning, a programmer writes a program to give some instructions to perform a task to the computer. Once it is learned (i.e. programmed), the system will be able to do new things.
- Also, there can be several sources for taking advice such as humans(experts), internet etc.
- However, this type of learning has a more necessity of inference than rote learning.
- As the stored knowledge in knowledge base gets transformed into an operational form, the reliability of the knowledge source is always taken into consideration.

4. Explanation based learning

- Explanation-based learning (EBL) deals with an idea of single-example learning.
- This type of learning usually requires a substantial number of training instances but there are two difficulties in this:
  I. it is difficult to have such a number of training instances
  ii. Sometimes, it may help us to learn certain things effectively, specially when we have enough knowledge.
  Hence, it is clear that instance-based learning is more data-intensive, data-driven while EBL is more knowledge-intensive, knowledge-driven.

5. Learning in Problem Solving

- Humans have a tendency to learn by solving various real world problems.
- The forms or representation, or the exact entity, problem solving principle is based on reinforcement learning.
- Therefore, repeating certain action results in desirable outcome while the action is avoided if it results into undesirable outcomes.
- As the outcomes have to be evaluated, this type of learning also involves the definition of a utility function. This function shows how much is a particular outcome worth?
- There are several research issues which include the identification of the learning rate, time and algorithm complexity, convergence, representation (frame and qualification problems), handling of uncertainty (ramification problem), adaptivity and "unlearning" etc.
- In reinforcement learning, the system (and thus the developer) know the desirable outcomes but does not know which actions result into desirable outcomes.

SUPERVISED LEARNING

Supervised learning is the types of machine learning in which machines are trained using well "labelled" training data, and on basis of that data, machines predict the output. The labelled data means some input data is already tagged with the correct output.

In supervised learning, the training data provided to the machines work as the supervisor that teaches the machines to predict the output correctly. It applies the same concept as a student learns in the supervision of the teacher.

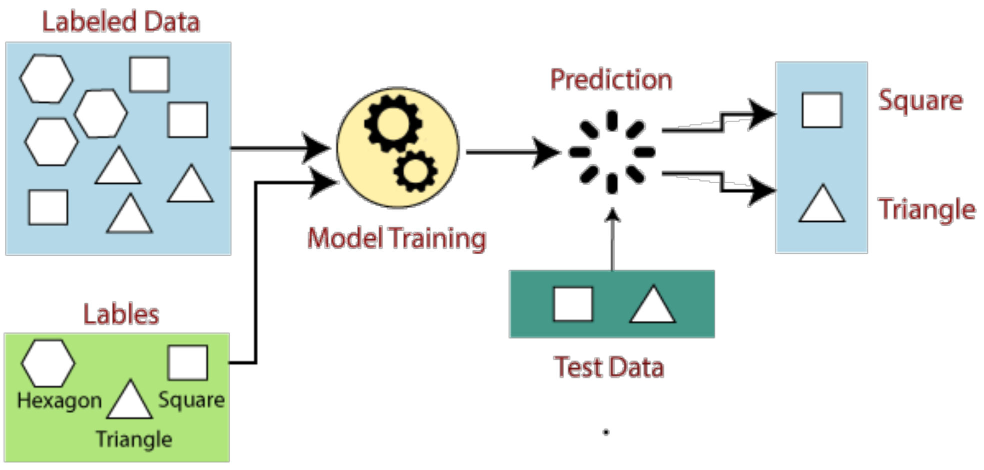
Supervised learning is a process of providing input data as well as correct output data to the machine learning model. The aim of a supervised learning algorithm is to **find a mapping function to map the input variable(x) with the output variable(y)**.

In the real-world, supervised learning can be used for **Risk Assessment, Image classification, Fraud Detection, spam filtering**, etc.

How Supervised Learning Works?

In supervised learning, models are trained using labelled dataset, where the model learns about each type of data. Once the training process is completed, the model is tested on the basis of test data (a subset of the training set), and then it predicts the output.

The working of Supervised learning can be easily understood by the below example and diagram:



Suppose we have a dataset of different types of shapes which includes square, rectangle, triangle, and Polygon. Now the first step is that we need to train the model for each shape.

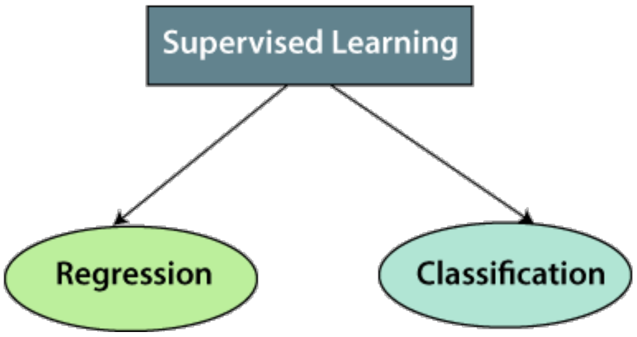
- If the given shape has four sides, and all the sides are equal, then it will be labelled as a **Square**.
- If the given shape has three sides, then it will be labelled as a **triangle**.
- If the given shape has six equal sides then it will be labelled as **hexagon**.

Now, after training, we test our model using the test set, and the task of the model is to identify the shape.

The machine is already trained on all types of shapes, and when it finds a new shape, it classifies the shape on the bases of a number of sides, and predicts the output.

Types of supervised Machine learning Algorithms:

Supervised learning can be further divided into two types of problems:



1. Regression

Regression algorithms are used if there is a relationship between the input variable and the output variable. It is used for the prediction of continuous variables, such as Weather forecasting, Market Trends, etc. Below are some popular Regression algorithms which come under supervised learning:

- Linear Regression
- Regression Trees
- Non-Linear Regression
- Bayesian Linear Regression
- Polynomial Regression

2. Classification

Classification algorithms are used when the output variable is categorical, which means there are two classes such as Yes-No, Male-Female, True-false, etc.

Spam Filtering,

- Random Forest
- Decision Trees
- Logistic Regression
- Support vector Machines

Advantages of Supervised learning:

- With the help of supervised learning, the model can predict the output on the basis of prior experiences.
- In supervised learning, we can have an exact idea about the classes of objects.
- Supervised learning model helps us to solve various real-world problems such as **fraud detection, spam filtering**, etc.

Disadvantages of supervised learning:

- Supervised learning models are not suitable for handling the complex tasks.
- Supervised learning cannot predict the correct output if the test data is different from the training dataset.
- Training required lots of computation times.
- In supervised learning, we need enough knowledge about the classes of object.

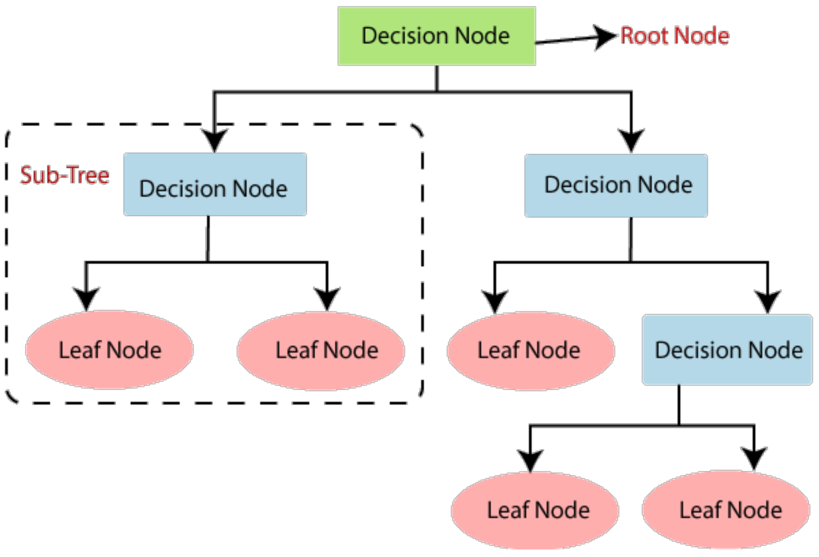
LEARNING DECISION TREES

Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems.

It is a tree-structured classifier, where **internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**

In a Decision tree, there are two nodes, which are the **Decision Node** and **Leaf Node.** Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.

- ***It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.***
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the **CART algorithm,** which stands for **Classification and Regression Tree algorithm.**
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
- Below diagram explains the general structure of a decision tree:



○

Why use Decision Trees?

- Decision Trees usually mimic human thinking ability while making a decision, so it is easy to understand.
- The logic behind the decision tree can be easily understood because it shows a tree-like structure.
-

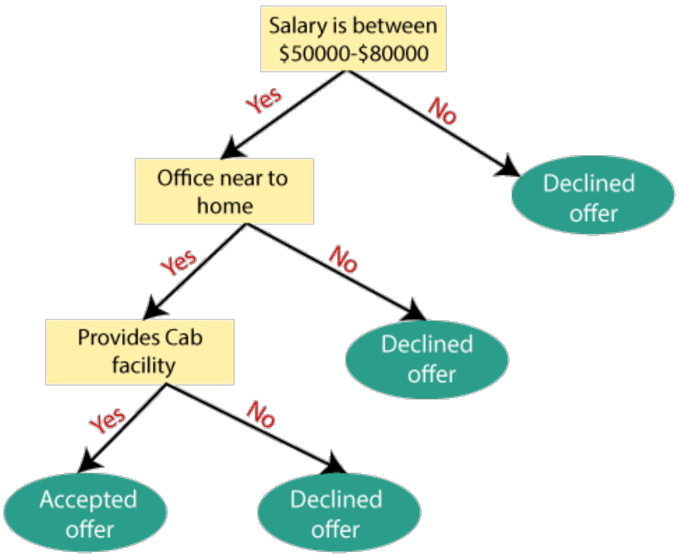
Decision Tree Terminologies

 Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.

 Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.

- **Splitting:** Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.
- **Branch/Sub Tree:** A tree formed by splitting the tree.
- **Pruning:** Pruning is the process of removing the unwanted branches from the tree.
- **Parent/Child node:** The root node of the tree is called the parent node, and other nodes are called the child nodes.

Example

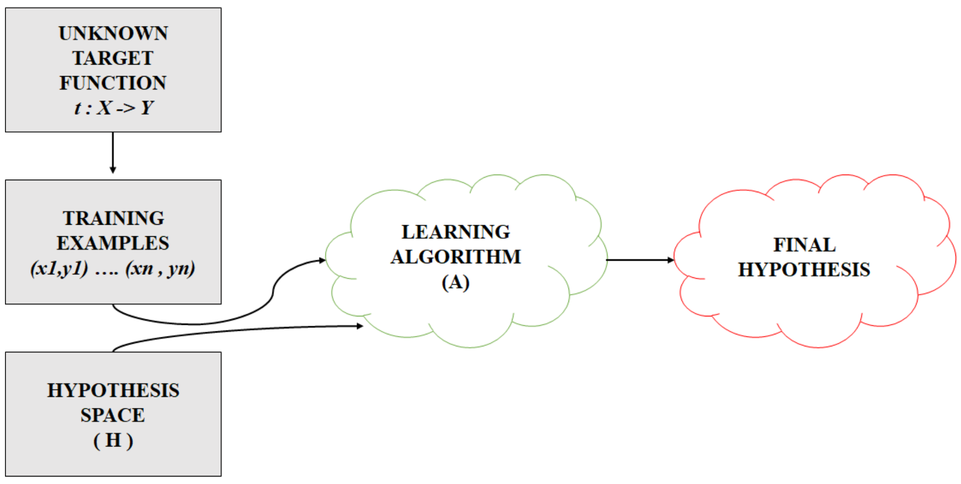


EVALUATING AND CHOOSING THE BEST HYPOTHESIS

In most supervised machine learning algorithm, our main goal is to find out a possible hypothesis from the hypothesis space that could possibly map out the inputs to the proper outputs.

The following figure shows the common method to find out the possible hypothesis from the Hypothesis space:

PRODUCING FINAL HYPOTHESIS

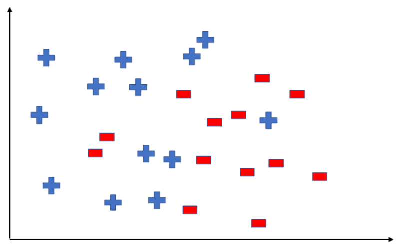


Hypothesis Space (H):

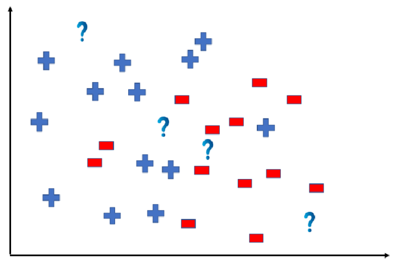
Hypothesis space is the set of all the possible legal hypothesis. This is the set from which the machine learning algorithm would determine the best possible (only one) which would best describe the target function or the outputs.

Hypothesis (h):

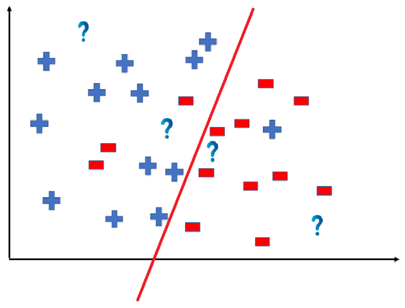
A hypothesis is a function that best describes the target in supervised machine learning. The hypothesis that an algorithm would come up depends upon the data and also depends upon the restrictions and bias that we have imposed on the data. To better understand the Hypothesis Space and Hypothesis consider the following coordinate that shows the distribution of some data:



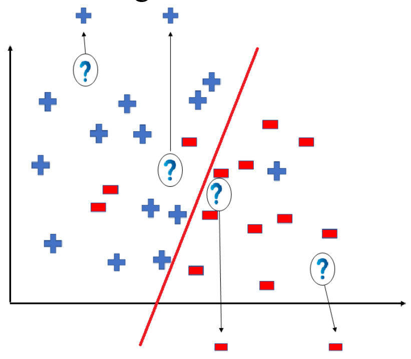
Say suppose we have test data for which we have to determine the outputs or results. The test data is as shown below:



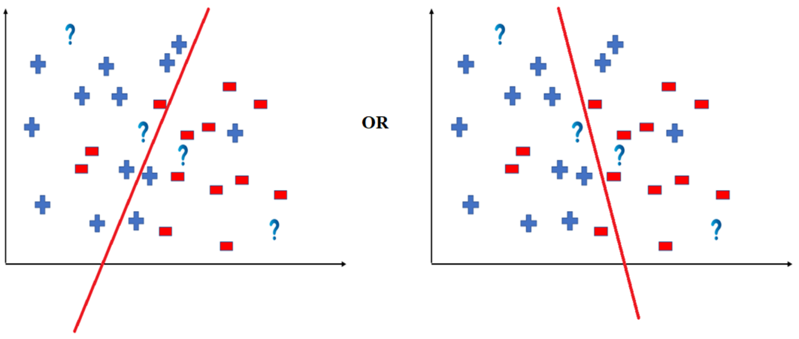
We can predict the outcomes by dividing the coordinate as shown below:



So the test data would yield the following result:



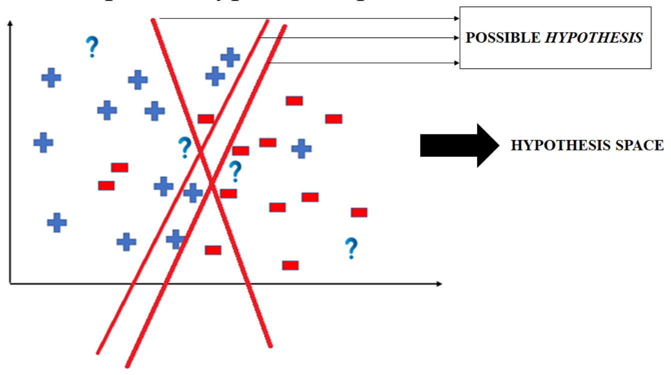
But note here that we could have divided the coordinate plane as:



The way in which the coordinate would be divided depends on the data, algorithm and constraints.

All these legal possible ways in which we can divide the coordinate plane to predict the outcome of the test data composes of the Hypothesis Space.

- Each individual possible way is known as the hypothesis.
- Hence, in this example the hypothesis space would be like:



THE THEORY OF LEARNING

Machine Learning Theory, also known as Computational Learning Theory, aims to understand the fundamental principles of learning as a computational process and combines tools from Computer Science and Statistics.

What is Machine Learning?

The area of Machine Learning deals with the design of programs that can learn rules from data, adapt to changes, and improve performance with experience. In addition to being one of the initial dreams of Computer Science, Machine Learning has become crucial as computers are expected to solve increasingly complex problems and become more integrated into our daily lives.

Writing a computer program is a bit like writing down instructions for an extremely literal child who just happens to be millions of times faster than you. Yet many of the problems we now want computers to solve are no longer tasks we know how to explicitly tell a computer how to do. These include identifying faces in images, autonomous driving in the desert, finding relevant documents in a database (or throwing out irrelevant ones, such as spam email), finding patterns in large volumes of scientific data, and adjusting internal parameters of systems to optimize performance. That is, we may ourselves be good at identifying people in photographs, but we do not know how to directly tell a computer how to do it. Instead, methods that take labeled training data (images labeled by who is in them, or email messages labeled by whether or not they are spam) and then learn appropriate rules from the data, seem to be the best approaches to solving these problems. Furthermore, we need systems that can adapt to changing conditions, that can be user-friendly by adapting to needs of their individual users, and that can improve performance over time.

What is Machine Learning Theory?

Machine Learning Theory, also known as Computational Learning Theory, aims to under- stand the fundamental principles of learning as a computational process. This field seeks to understand at a precise mathematical level what capabilities and information are fundamentally needed to learn different kinds of tasks successfully, and to understand the basic algorithmic principles involved in getting computers to learn from data and to improve performance with feedback. The goals of this theory are both to aid in the design of better automated learning methods and to understand fundamental issues in the learning process itself.

Machine Learning Theory draws elements from both the Theory of Computation and Statistics and involves tasks such as:

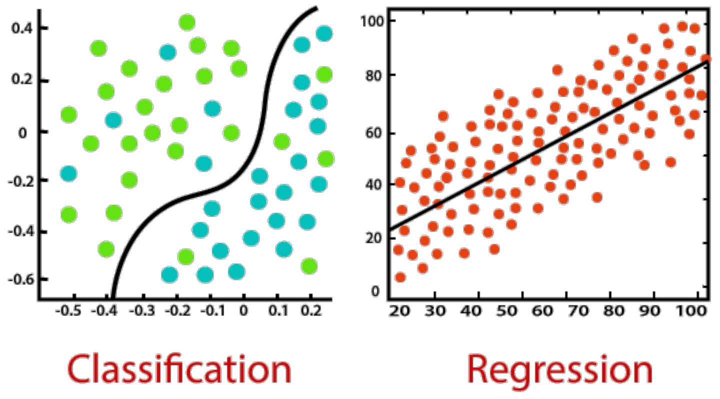
- Creating mathematical models that capture key aspects of machine learning, in which one can analyze the inherent ease or difficulty of different types of learning problems.
- Proving guarantees for algorithms (under what conditions will they succeed, how much data and computation time is needed) and developing machine learning algorithms that provably meet desired criteria.
- Mathematically analyzing general issues, such as: “Why is Occam’s Razor a good idea?”, “When can one be confident about predictions made from limited data?”, “How much power does active participation add over passive observation for learning?”, and “What kinds of methods can learn even in the presence of large quantities of distracting information?”.

REGRESSION AND CLASSIFICATION WITH LEARNER MODELS

Regression and Classification algorithms are Supervised Learning algorithms. Both the algorithms are used for prediction in Machine learning and work with the labeled datasets. But the difference between both is how they are used for different machine learning problems.

The main difference between Regression and Classification algorithms that Regression algorithms are used to **predict the continuous** values such as price, salary, age, etc. and Classification algorithms are used to **predict/Classify the discrete values** such as Male or Female, True or False, Spam or Not Spam, etc.

Consider the below diagram:



Classification

Classification is a process of finding a function which helps in dividing the dataset into classes based on different parameters. In Classification, a computer program is trained on the training dataset and based on that training, it categorizes the data into different classes.

The task of the classification algorithm is to find the mapping function to map the input(x) to the discrete output(y).

Example:

The best example to understand the Classification problem is Email Spam Detection. The model is trained on the basis of millions of emails on different parameters, and whenever it receives a new email, it identifies whether the email is spam or not. If the email is spam, then it is moved to the Spam folder.

Types of ML Classification Algorithms:

Classification Algorithms can be further divided into the following types:

- Logistic Regression
- K-Nearest Neighbours
- Support Vector Machines
- Kernel SVM
- Naïve Bayes
- Decision Tree Classification
- Random Forest Classification

Regression

Regression is a process of finding the correlations between dependent and independent variables. It helps in predicting the continuous variables such as prediction of **Market Trends**, prediction of House prices, etc.

The task of the Regression algorithm is to find the mapping function to map the input variable(x) to the continuous output variable(y).

Example:

Suppose we want to do weather forecasting, so for this, we will use the Regression algorithm. In weather prediction, the model is trained on the past data, and once the training is completed, it can easily predict the weather for future days.

Types of Regression Algorithm:

- Simple Linear Regression
- Multiple Linear Regression
- Polynomial Regression
- Support Vector Regression
- Decision Tree Regression
- Random Forest Regression

Difference between Regression and Classification

Regression Algorithm	Classification Algorithm
In Regression, the output variable must be of continuous nature or real value.	In Classification, the output variable must be a discrete value.
The task of the regression algorithm is to map the input value (x) with the continuous output variable(y).	The task of the classification algorithm is to map the input value(x) with the discrete output variable(y).
Regression Algorithms are used with continuous data.	Classification Algorithms are used with discrete data.
In Regression, we try to find the best fit line, which can predict the output more accurately.	In Classification, we try to find the decision boundary, which can divide the dataset into different classes.
Regression algorithms can be used to solve the regression problems such as Weather Prediction, House price prediction, etc.	Classification Algorithms can be used to solve classification problems such as Identification of spam emails, Speech Recognition, Identification of cancer cells, etc.
The regression Algorithm can be further divided into Linear and Non-linear Regression.	The Classification algorithms can be divided into Binary Classifier and Multi-class Classifier.

NONPARAMETRIC MODELS

Machine learning algorithms are classified into two distinct groups: parametric and nonparametric models.

What is the parametric model?

A learning model that summarizes data with a set of fixed-size parameters (independent on the number of instances of training).

Parametric machine learning algorithms are which optimizes the function to a known form. n a parametric model, you know exactly which model you are going to fit in with the data, for example, linear regression line.

b0 + b1*x1 + b2*x2 = 0

where,

b0, b1, b2 → the coefficients of the line that control the intercept and slope

x1, x2 → input variables

The assumed functional form is always a linear combination of input variables and as such parametric machine learning algorithms are also frequently referred to as '***linear machine learning algorithms***.'

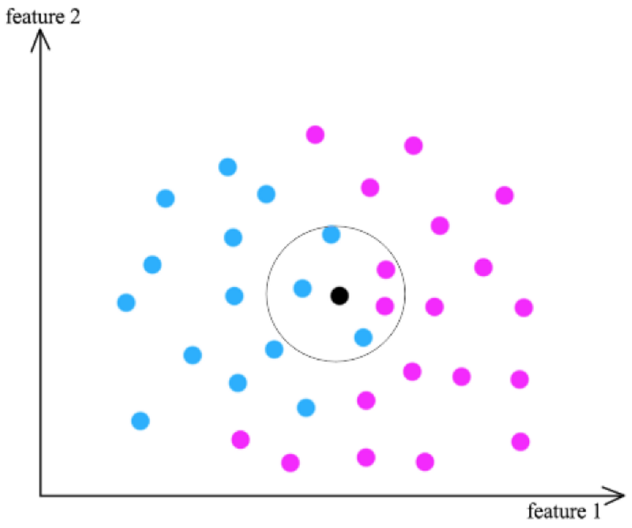
Some more examples of parametric machine learning algorithms include:

- Logistic Regression
- Linear Discriminant Analysis
- Perceptron
- Naive Bayes
- Simple Neural Networks

What is the nonparametric model?

Nonparametric machine learning algorithms are those which do not make specific assumptions about the type of the mapping function. They are prepared to choose any functional form from the training data, by not making assumptions.

The word nonparametric does not mean that the value lacks parameters existing in it, but rather that the parameters are adjustable and can change. When dealing with ranked data one may turn to nonparametric modeling, in which the sequence in that they are ordered is some of the significance of the parameters.



A simple to understand the nonparametric model is the k-nearest neighbors' algorithm, making predictions for a new data instance based on the most similar training patterns k. The only assumption it makes about the data set is that the training patterns that are the most similar are most likely to have a similar result.

Some more examples of popular nonparametric machine learning algorithms are:

- k-Nearest Neighbors
- Decision Trees like CART and C4.5
- Support Vector Machines

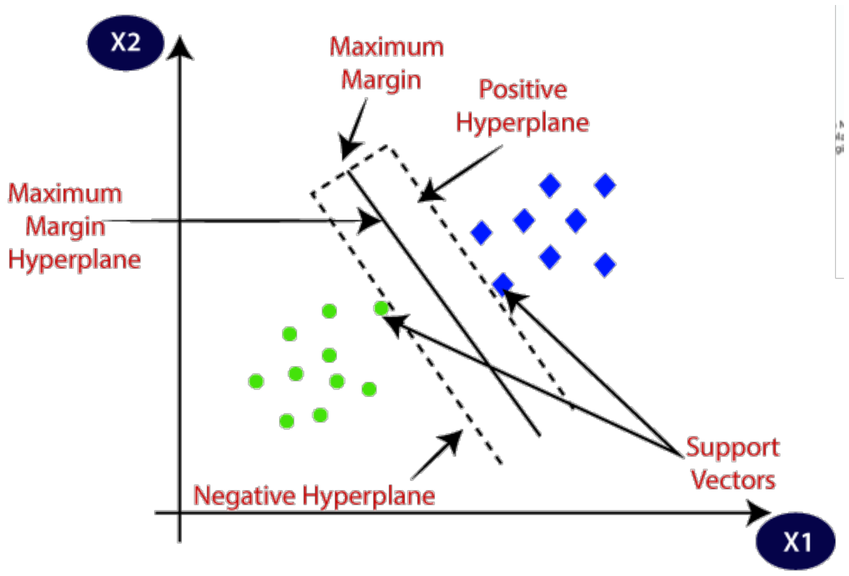
SUPPORT VECTOR MACHINES

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

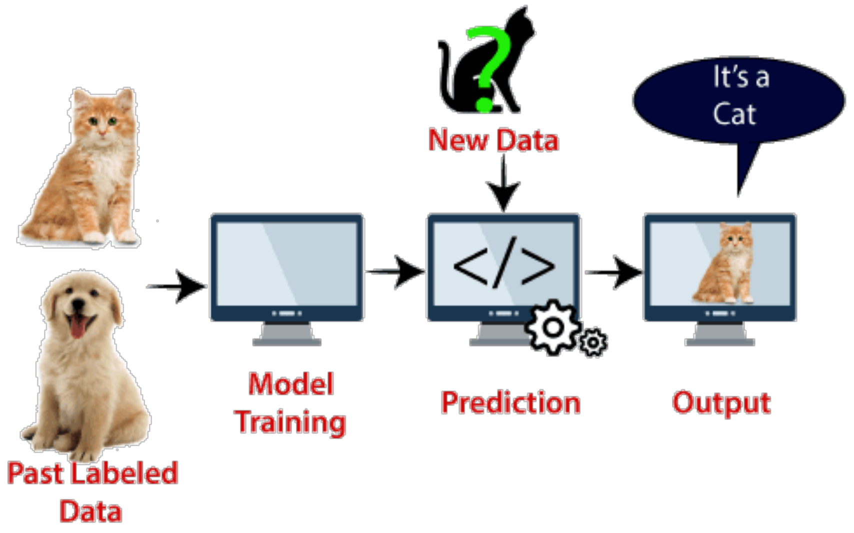
SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine.

Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



Example

SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. On the basis of the support vectors, it will classify it as a cat. Consider the below diagram:



SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.

Types of SVM

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane:

There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.

We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

Support Vectors:

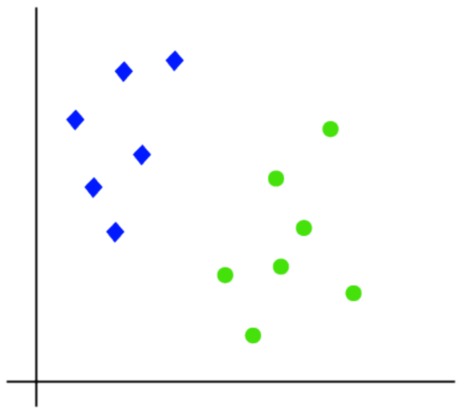
The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

How does SVM works?

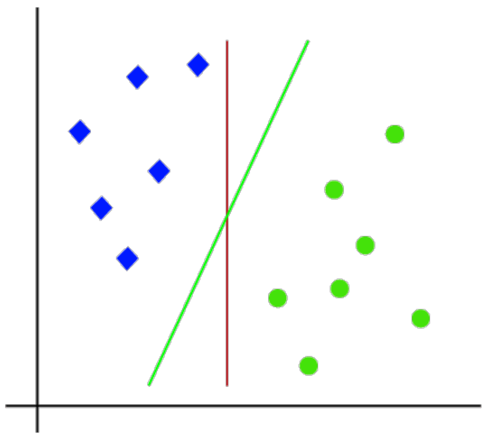
Linear SVM:

The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue.

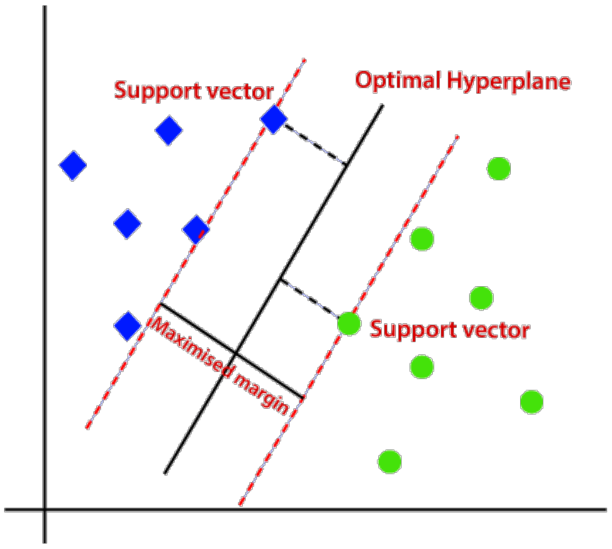
Consider the below image:



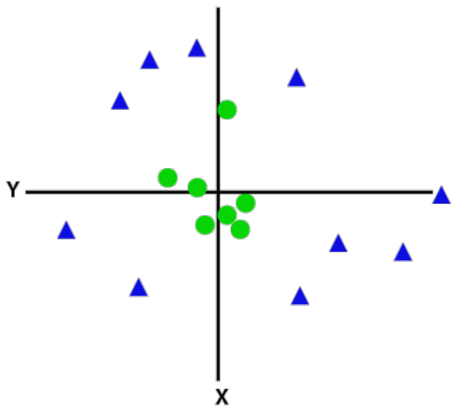
So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:



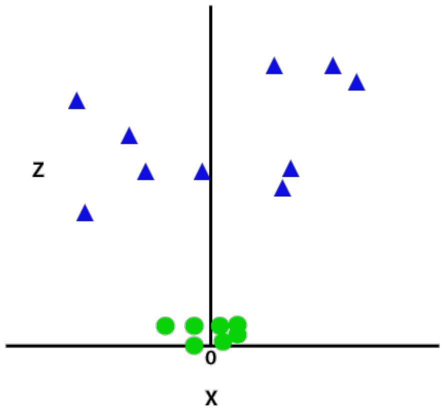
Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a hyperplane. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as margin. And the goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the optimal hyperplane.



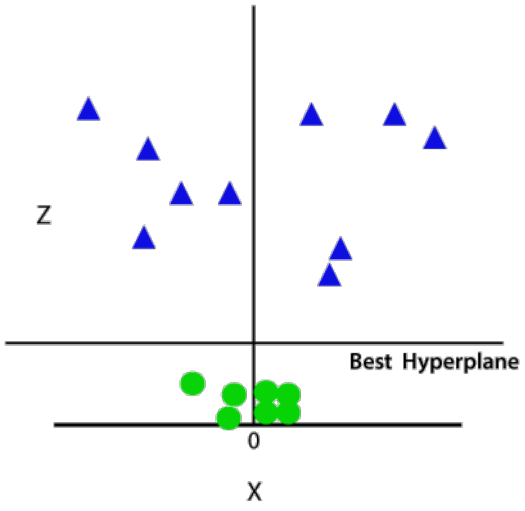
Non-Linear SVM:
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line.
Consider the below image:



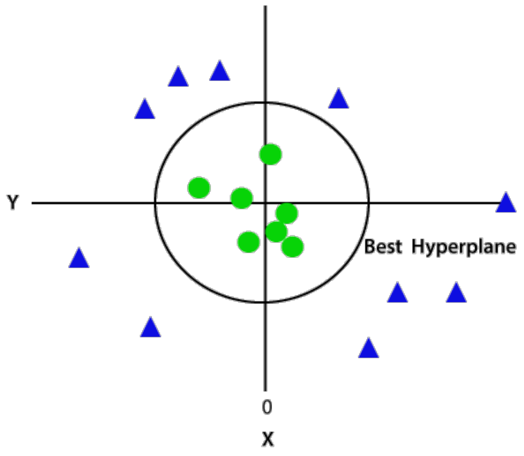
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:
z=x2 +y2
By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way.
Consider the below image:



Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:



Hence we get a circumference of radius 1 in case of non-linear data.

ENSEMBLE LEARNING

Ensemble learning is the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular computational intelligence problem. Ensemble learning is primarily used to improve the (classification, prediction, function approximation, etc.) performance of a model, or reduce the likelihood of an unfortunate selection of a poor one.

An ensemble-based system is obtained by combining diverse models (henceforth classifiers). Therefore, such systems are also known as multiple classifier systems, or just ensemble systems. There are several scenarios where using an ensemble based system makes statistical sense, which are discussed below in detail. However, in order to fully and practically appreciate the importance of using multiple classifier systems, it is perhaps instructive to look at a psychological backdrop to this otherwise statistically sound argument: we use such an approach routinely in our daily lives by asking the opinions of several experts before making a decision. For example, we typically ask the opinions of several doctors before agreeing to a medical procedure, we read user reviews before purchasing an item (particularly big ticket items), we evaluate future employees by checking their references, etc. In fact, even this article is reviewed by several experts before being accepted for publication. In each case, a final decision is made by combining the individual decisions of several experts. In doing so, the primary goal is to minimize the unfortunate selection of an unnecessary medical procedure, a poor product, an unqualified employee or even a poorly written and misguiding article.

Model Selection

This is perhaps the primary reason why ensemble based systems are used in practice: what is the most appropriate classifier for a given classification problem? This question can be interpreted in two different ways: i) what type of classifier should be chosen among many competing models, such as multilayer perceptron (MLP), support vector machines (SVM), decision trees, naive Bayes classifier, etc; ii) given a particular classification algorithm, which realization of this algorithm should be chosen - for example, different initializations of MLPs can give rise to different decision boundaries, even if all other parameters are kept constant.

Too much or too little data

Ensemble based systems can be - perhaps surprisingly - useful when dealing with large volumes of data or lack of adequate data. When the amount of training data is too large to make a single classifier training difficult, the data can be strategically partitioned into smaller subsets. Each partition can then be used to train a separate classifier which can then be combined using an appropriate combination rule (see below for different combination rules). If, on the other hand, there is too little data, then bootstrapping can be used to train different classifiers using different **bootstrap samples** of the data

Data Fusion

In many applications that call for automated decision making, it is not unusual to receive data obtained from different sources that may provide complementary information. A suitable combination of such information is known as **data or information fusion,** and can lead to improved accuracy of the classification decision compared to a decision based on any of the individual data sources alone.

Confidence Estimation

The very structure of an ensemble based system naturally allows assigning a confidence to the decision made by such a system. Consider having an ensemble of classifiers trained on a classification problem. If a vast majority of the classifiers agree with their decisions, such an outcome can be interpreted as the ensemble having high confidence in its decision. If, however, half the classifiers make one decision and the other half make a different decision, this can be interpreted as the ensemble having low confidence in its decision. It should be noted that an ensemble having high confidence in its decision does not mean that decision is correct, and conversely, a decision made with low confidence need not be incorrect.

PRACTICAL MACHINE LEARNING

We can read authoritative definitions of machine learning, but really, machine learning is defined by the problem being solved. Therefore the best way to understand machine learning is to look at some example problems.

In this section we will first look at some well known and understood examples of machine learning problems in the real world. We will then look at a taxonomy (naming system) for standard machine learning problems and learn how to identify a problem as one of these standard cases. This is valuable, because knowing the type of problem we are facing allows us to think about the data we need and the types of algorithms to try.

Machine Learning problems are abound. They make up core or difficult parts of the software you use on the web or on your desktop everyday. Think of the "do you want to follow" suggestions on twitter and the speech understanding in Apple's Siri.

Below are 10 examples of machine learning that really ground what machine learning is all about.

- **Spam Detection**: Given email in an inbox, identify those email messages that are spam and those that are not. Having a model of this problem would allow a program to leave non-spam emails in the inbox and move spam emails to a spam folder. We should all be familiar with this example.
- **Credit Card Fraud Detection**: Given credit card transactions for a customer in a month, identify those transactions that were made by the customer and those that were not. A program with a model of this decision could refund those transactions that were fraudulent.
- **Digit Recognition**: Given a zip codes hand written on envelops, identify the digit for each hand written character. A model of this problem would allow a computer program to read and understand handwritten zip codes and sort envelops by geographic region.
- **Speech Understanding**: Given an utterance from a user, identify the specific request made by the user. A model of this problem would allow a program to understand and make an attempt to fulfil that request. The iPhone with Siri has this capability.
- **Face Detection**: Given a digital photo album of many hundreds of digital photographs, identify those photos that include a given person. A model of this

decision process would allow a program to organize photos by person. Some cameras and software like iPhoto has this capability.

- **Product Recommendation**: Given a purchase history for a customer and a large inventory of products, identify those products in which that customer will be interested and likely to purchase. A model of this decision process would allow a program to make recommendations to a customer and motivate product purchases. Amazon has this capability. Also think of Facebook, GooglePlus and LinkedIn that recommend users to connect with you after you sign-up.
- **Medical Diagnosis**: Given the symptoms exhibited in a patient and a database of anonymized patient records, predict whether the patient is likely to have an illness. A model of this decision problem could be used by a program to provide decision support to medical professionals.
- **Stock Trading**: Given the current and past price movements for a stock, determine whether the stock should be bought, held or sold. A model of this decision problem could provide decision support to financial analysts.
- **Customer Segmentation**: Given the pattern of behaviour by a user during a trial period and the past behaviours of all users, identify those users that will convert to the paid version of the product and those that will not. A model of this decision problem would allow a program to trigger customer interventions to persuade the customer to covert early or better engage in the trial.
- **Shape Detection**: Given a user hand drawing a shape on a touch screen and a database of known shapes, determine which shape the user was trying to draw. A model of this decision would allow a program to show the platonic version of that shape the user drew to make crisp diagrams. The Instaviz iPhone app does this.

Types of Machine Learning Problems

There are common classes of problem in Machine Learning. The problem classes below are archetypes for most of the problems we refer to when we are *doing* Machine Learning.

- **Classification**: Data is labelled meaning it is assigned a class, for example spam/non-spam or fraud/non-fraud. The decision being modelled is to assign labels to new unlabelled pieces of data. This can be thought of as a discrimination problem, modelling the differences or similarities between groups.
- **Regression**: Data is labelled with a real value (think floating point) rather then a label. Examples that are easy to understand are time series data like the price of a stock over time, The decision being modelled is what value to predict for new unpredicted data.
- **Clustering**: Data is not labelled, but can be divided into groups based on similarity and other measures of natural structure in the data. An example from the above list would be organising pictures by faces without names, where the human user has to assign names to groups, like iPhoto on the Mac.
- **Rule Extraction**: Data is used as the basis for the extraction of propositional rules (antecedent/consequent aka *if-then*). Such rules may, but are typically not directed, meaning that the methods discover statistically supportable relationships between attributes in the data, not necessarily involving something that is being predicted.

****

UNIT- V

LEARNING PROBABILISTIC MODELS

Most real-world data is stored in relational form. In contrast, most statistical learning methods work with "flat" data representations, forcing us to convert our data into a form that loses much of the relational structure. The recently introduced framework of *probabilistic relational models* (PRMs) allows us to represent probabilistic models over multiple entities that utilize the relations between them.

STATISTICAL LEARNING

Statistical Learning is a set of tools for understanding data. These tools broadly come under two classes: supervised learning & unsupervised learning. Generally, supervised learning refers to predicting or estimating an output based on one or more inputs. Unsupervised learning, on the other hand, provides a relationship or finds a pattern within the given data without a supervised output.

What is Statistical Learning?

Let, suppose that we observe a response Y and p different predictors $X = (X_1, X_2, \ldots, Xp)$. In general, we can say:

$Y = f(X) + \varepsilon$

Here **f** is an unknown function, and **ε** is the *random error term*.

In essence, statistical learning refers to a set of approaches for estimating f.

In cases where we have set of X readily available, but the output Y, not so much, the error averages to zero, and we can say:

$¥ = f(X)$

where ***f*** represents our estimate of **f** and **¥** represents the resulting prediction.

Hence for a set of predictors X, we can say:

$E(Y — ¥)^2 = E[f(X) + \varepsilon — f(X)]^2 \Rightarrow E(Y — ¥)^2 = [f(X) - f(X)]^2 + Var(\varepsilon)$

where,

- $E(Y — ¥)^2$ represents *the expected value* of the squared difference between actual and expected result.
- $[f(X) — f(X)]^2$ represents the ***reducible error***. It is reducible because we can potentially improve the accuracy of *f* by better modeling.
- *Var(ε)* represents the **irreducible error**. It is irreducible because no matter how well we estimate *f*, we cannot reduce the error introduced by *variance* in ε.

LEARNING WITH COMPLETE DATA

Our development of statistical learning methods begins with the simplest task: parameter learning with complete data. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. Data are complete when each data point contains values for every variable in the probability model being learned. Complete data greatly simplify the problem of learning the parameters of a complex model. We will also look briefly at the problem of learning structure.

Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown—that is, the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The parameter in this case, which we call , is the proportion of cherry candies, and the hypothesis is h. (The proportion of limes is just 1 -.) If we assume that all proportions are equally likely a priori, then a maximum likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, Flavor (the flavor of a randomly chosen candy from the bag). It has values cherry and lime, where the probability of cherry is (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and `=N - c are limes. According to Equation (3), the likelihood of this particular data set is

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell .$$

The maximum-likelihood hypothesis is given by the value of that maximizes this expression. The same value is obtained by maximizing the log likelihood,

$$L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^{N} \log P(d_j|h_\theta) = c\log\theta + \ell\log(1-\theta)$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of , we differentiate L with respect to and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell} = \frac{c}{N}$$

In English, then, the maximum-likelihood hypothesis ML asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far! It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.

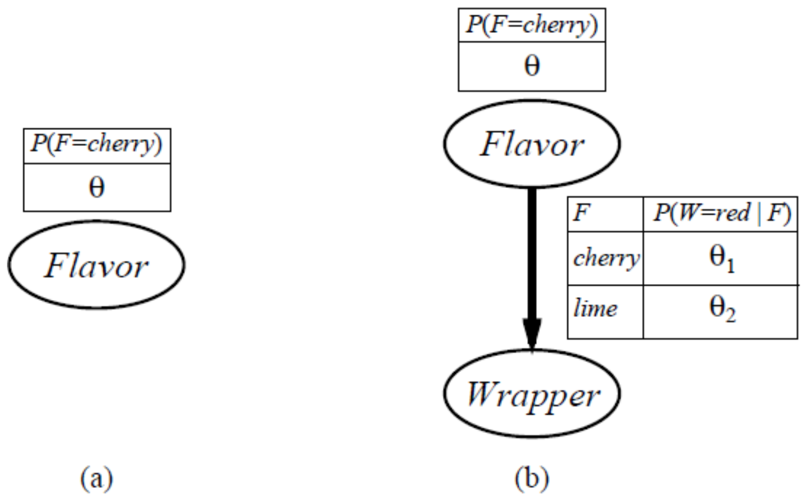


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter 4. The example also illustrates a significant problem with maximum-likelihood learning in general: when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum likelihood hypothesis assigns zero probability to those events. Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of zero.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that, with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter. The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

LEARNING WITH HIDDEN VARIABLES

The next simplest case is where the model is given, but not all variables are observed. A **hidden variable** or a **latent variable** is a variable in a belief network whose value is not observed for any of the examples. That is, there is no column in the data corresponding to that variable.

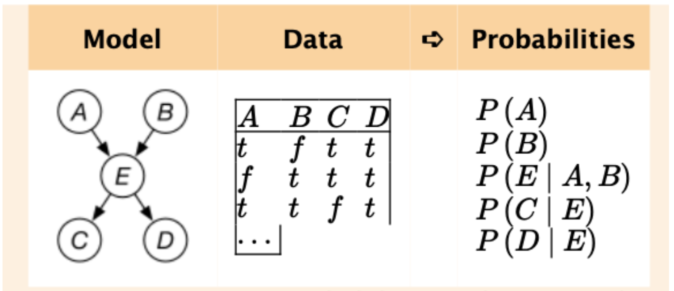


Figure 10.8: Deriving probabilities with missing data

Example

Figure 10.8 shows a typical case. Assume that all the variables are binary. The model contains a hidden variable E that is in the model but not the data set. The aim is to learn the parameters of the model that includes the hidden variable E. There are 10 parameters to learn.

Note that, if E was not part of the model, the algorithm would have to learn P(A), P(B), P(C |AB), P(D |ABC), which has 14 parameters. The reason for introducing hidden variables is, paradoxically, to make the model simpler and, therefore, less prone to overfitting.

THE EM ALGORITHM

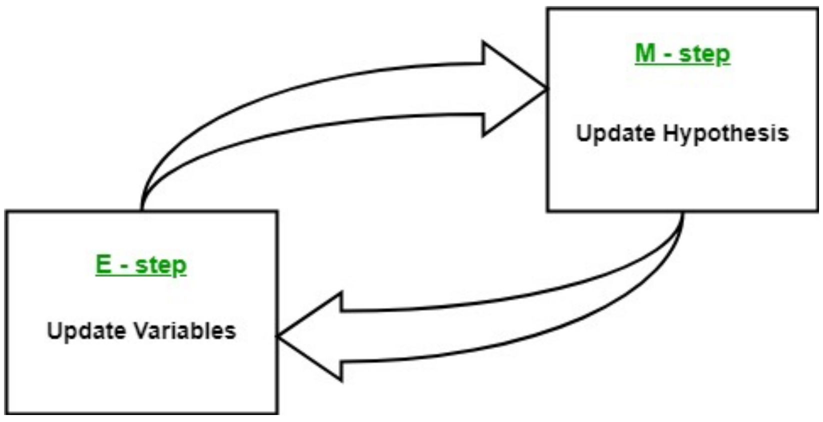
In the real-world applications of machine learning, it is very common that there are many relevant features available for learning but only a small subset of them are observable. So, for the variables which are sometimes observable and sometimes not, then we can use the instances when that variable is visible is observed for the purpose of learning and then predict its value in the instances when it is not observable.

On the other hand, Expectation-Maximization algorithm can be used for the latent variables (variables that are not directly observable and are actually inferred from the values of the other observed variables) too in order to predict their values with the condition that the general form of probability distribution governing those latent variables is known to us. This algorithm is actually at the base of many unsupervised clustering algorithms in the field of machine learning.

It was explained, proposed and given its name in a paper published in 1977 by Arthur Dempster, Nan Laird, and Donald Rubin. It is used to find the local maximum likelihood parameters of a statistical model in the cases where latent variables are involved and the data is missing or incomplete.

Algorithm:

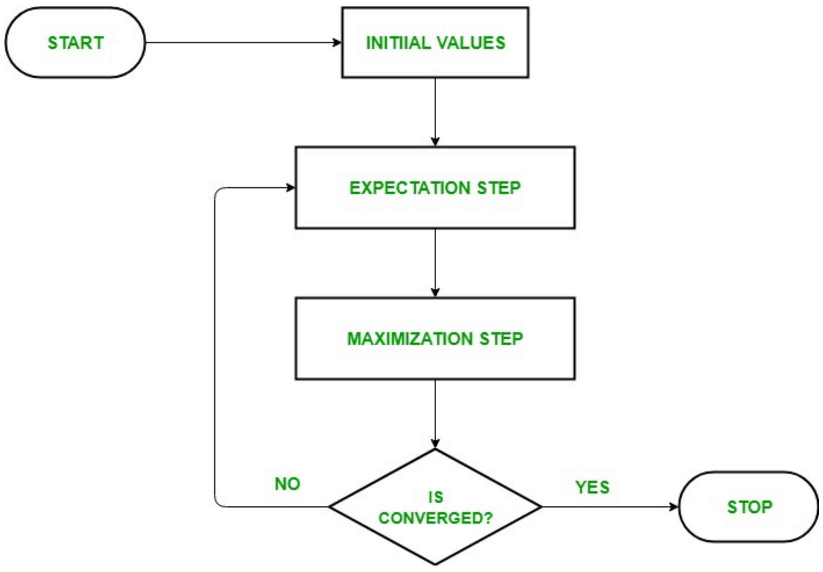
1. Given a set of incomplete data, consider a set of starting parameters.
2. **Expectation step (E – step):** Using the observed available data of the dataset, estimate (guess) the values of the missing data.
3. **Maximization step (M – step):** Complete data generated after the expectation (E) step is used in order to update the parameters.
4. Repeat step 2 and step 3 until convergence.



The essence of Expectation-Maximization algorithm is to use the available observed data of the dataset to estimate the missing data and then using that data to update the values of the parameters. Let us understand the EM algorithm in detail.

- Initially, a set of initial values of the parameters are considered. A set of incomplete observed data is given to the system with the assumption that the observed data comes from a specific model.
- The next step is known as "Expectation" – step or *E-step*. In this step, we use the observed data in order to estimate or guess the values of the missing or incomplete data. It is basically used to update the variables.
- The next step is known as "Maximization"-step or *M-step*. In this step, we use the complete data generated in the preceding "Expectation" – step in order to update the values of the parameters. It is basically used to update the hypothesis.
- Now, in the fourth step, it is checked whether the values are converging or not, if yes, then stop otherwise repeat *step-2* and *step-3* i.e. "Expectation" – step and "Maximization" – step until the convergence occurs.

Flow chart for EM Algorithm



Usage of EM algorithm –

- It can be used to fill the missing data in a sample.
- It can be used as the basis of unsupervised learning of clusters.
- It can be used for the purpose of estimating the parameters of Hidden Markov Model (HMM).
- It can be used for discovering the values of latent variables.

Advantages of EM algorithm –

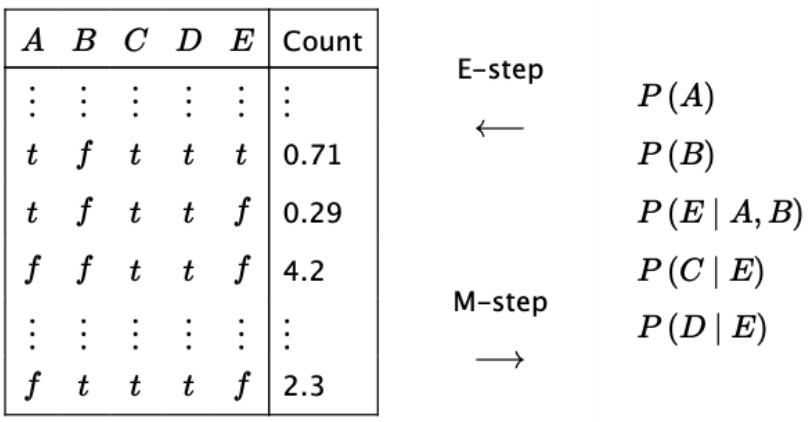
- It is always guaranteed that likelihood will increase with each iteration.
- The E-step and M-step are often pretty easy for many problems in terms of implementation.
- Solutions to the M-steps often exist in the closed form.

Disadvantages of EM algorithm –

- It has slow convergence.
- It makes convergence to the local optima only.
- It requires both the probabilities, forward and backward (numerical optimization requires only forward probability).

Example

The **expectation maximization** or **EM** algorithm for learning belief networks with hidden variables is essentially the same as the EM algorithm for clustering. The E step, shown in Figure 10.9, involves probabilistic inference for each example to infer the probability distribution of the hidden variable(s) given the observed variables for that example. The M step of inferring the probabilities of the model from the augmented data is the same as in the fully observable case discussed in the previous section, but, in the augmented data, the counts are not necessarily integers.



A	B	C	D	E	Count
⋮	⋮	⋮	⋮	⋮	⋮
t	f	t	t	t	0.71
t	f	t	t	f	0.29
f	f	t	t	f	4.2
⋮	⋮	⋮	⋮	⋮	⋮
f	t	t	t	f	2.3

Figure 10.9: EM algorithm for belief networks with hidden variables; E is a hidden variable. The E-step computes P(E | A,B,C,D) for each example, and the M-step learns probabilities from complete data.

****