Artificial Intelligence

By
Dr Zafar. M. Alvi

Table of Contents:

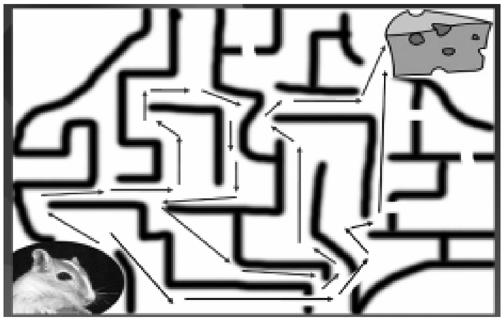
Artificial Intelligence

1 Introduction

This booklet is organized as chapters that elaborate on various concepts of Artificial Intelligence. The field itself is an emerging area of computer sciences and a lot of work is underway in order to mature the concepts of this field.
In this booklet we will however try to envelop some important aspects and basic concepts which will help the reader to get an insight into the type of topics that Artificial Intelligence deals with.
We have used the name of the field i.e. Artificial Intelligence (commonly referred as AI) without any explanation of the name itself. Let us now look into a simple but comprehensive way to define the field.
To define AI, let us first try to understand that what is **Intelligence**?

1.1 What is Intelligence?

If you were asked a simple question; **how can we define Intelligence**? many of you would exactly know what it is but most of you won't exactly be able to define it. **Is it something tangible?** We all know that it does exist but what actually it is. Some of us will attribute intelligence to living beings and would be of the view that all living species are intelligent. But how about these plants and tress, they are living species but are they also intelligent? So can we say that Intelligence is a trait of **some** living species? Let us try to understand the phenomena of intelligence by using a few examples.
Consider the following image where a mouse is trying to search a maze in order to find its way from the bottom left to the piece of cheese in the top right corner of the image.



This problem can be considered as a common real life problem which we deal with many times in our life, i.e. finding a path, may be to a university, to a friends house, to a market, or in this case to the piece of cheese. The mouse tries various paths as shown by arrows and can reach the cheese by more than one path. In other words the mouse can find more than one solutions to this problem. The mouse was **intelligent** enough to find a solution to the problem at hand. Hence the **ability of problem solving** demonstrates intelligence.

Let us consider another problem. Consider the sequence of numbers below:

1, 3, 7, 13, 21, ___

If you were asked to find the next number in the sequence what would be your answer? Just to help you out in the answer let us solve it for you “adding the next even number to the” i.e. if we add 2 to 1 we get 3, then we add 4 to 3 we get 7, then we get 6 to 7 we get 13, then we add 8 to 13 we get 21 and finally if we’ll add 10 to 21 we’ll get 31 as the answer. Again answering the question requires a little bit intelligence. The characteristic of intelligence comes in when we try to solve something, we check various ways to solve it, we check different combinations, and many other things to solve different problems. All this thinking, this memory manipulation capability, this numerical processing ability and a lot of other things add to ones intelligence.

All of you have experienced your college life. It was very easy for us to look at the timetable and go to the respective classes to attend them. Not even caring that how that time table was actually developed. In simple cases developing such a timetable is simple. But in cases where we have 100s of students studying in different classes, where we have only a few rooms and limited time to schedule all those classes. This gets tougher and tougher. The person who makes the timetable has to look into all the time schedule, availability of the teachers, availability of the rooms, and many other things to fit all the items correctly within a fixed span of time. He has to look into many expressions and thoughts like “If room A is free AND teacher B is ready to take the class AND the students of the class are not studying any other course at that time” THEN “the class can be scheduled”. This is a fairly simple one, things get complex as we add more and more parameters e.g. if we were to consider that teacher B might teach more than one course and he might just prefer to teach in room C and many other things like that. The problem gets more and more complex. We are pretty much sure than none of us had ever realized the complexity through which our teachers go through while developing these schedules for our classes. However, like we know such time tables can be developed. All this information has to reside in the developer’s brain. His intelligence helps him to create such a schedule. Hence **the ability to think, plan and schedule** demonstrate intelligence.

Consider a doctor, he checks many patients daily, diagnoses their disease, gives them medicine and prescribes them behaviors that can help them to get cured. Let us think a little and try to understand that what actually he does. Though checking a patient and diagnosing the disease is much more complex but we’ll try to keep our discussion very simple and will intentionally miss out stuff from this discussion.

A person goes to doctor, tells him that he is not feeling well. The doctor asks him a few questions to clarify the patient’s situation. The doctor takes a few measurements to check the physical status of the person. These measurements might just include the temperature (T), Blood Pressure (BP), Pulse Rate (PR) and things like that. For simplicity let us consider that some doctor only checks these measurements and tries to come up with a diagnosis for the disease. He takes these measurements and based on his previous knowledge he tries to diagnose the disease. His previous knowledge is based on rules like: “If the patient has a high BP and normal T and normal PR then he is not well”. “If only the BP is normal then what ever the other measurements may be the person should be healthy”, and many such other rules.

The key thing to notice is that by using such rules the doctor might classify a person to be healthy or ill and might as well prescribe different medicines to him using the information observed from the measurements according to his previous knowledge. Diagnosing a disease has many other complex information and observations involved, we have just mentioned a very simple case here. However, the doctor is actually faced with solving a problem of diagnosis having looked at some specific measurements. It is important to consider that a doctor who would have a better memory to store all this precious knowledge, better ability of retrieving the correct portion of the knowledge for the correct patient will be better able to classify a patient. Hence, telling us that **memory and correct and efficient memory and information manipulation** also counts towards ones intelligence.

Things are not all that simple. People don't think about problems in the same manner. Let us give you an extremely simple problem. Just tell us about your height. Are you short, medium or tall? An extremely easy question! Well you might just think that you are tall but your friend who is taller than you might say that NO! You are not. The point being that some people might have such a distribution in their mind that people having height around 4ft are short, around 5ft are medium and around 6ft are tall. Others might have this distribution that people having height around 4.5ft are short, around 5.5ft are medium and around 6.5ft are tall. Even having the same measurements different people can get to completely different results as they approach the problem in different fashion. Things can be even more complex when the same person, having observed same measurements solves the same problem in two different ways and reaches different solutions. But we all know that we answer such fuzzy questions very efficiently in our daily lives. Our intelligence actually helps us do this. Hence the **ability to tackle ambiguous and fuzzy problems** demonstrates intelligence.

Can you recognize a person just by looking at his/her fingerprint? Though we all know that every human has a distinct pattern of his/her fingerprint but just by looking at a fingerprint image a human generally can't just tell that this print must be of person XYZ. On the other hand having distinct fingerprint is really important information as it serves as a unique ID for all the humans in this world.

Let us just consider 5 different people and ask a sixth one to have a look at different images of their fingerprints. We ask him to somehow learn the patterns, which make the five prints distinct in some manner. After having seen the images a several times, that sixth person might get to find something that is making the prints distinct. Things like one of them has fever lines in the print, the other one has sharply curved lines, some might have larger distance between the lines in the print and some might have smaller displacement between the lines and many such features. The point being that after some time, which may be in hours or days or may be even months, that sixth person will be able to look at a new fingerprint of one of those five persons and he might with some degree of accuracy recognize that which one amongst the five does it belong. Only with 5 people the problem was hard to solve. His intelligence helped him to learn the features that distinguish one finger print from the other. Hence the **ability to learn and recognize** demonstrates intelligence.

Let us give one last thought and then will get to why we have discussed all this. A lot of us regularly watch television. Consider that you switch off the volume of your TV set. If you are watching a VU lecture you will somehow perceive that the person standing in front of you is not singing a song, or anchoring a musical show or playing some sport. So just by observing the sequence of images of the person you are able to perceive meaningful information out of the video. Your intelligence helped you to perceive and understand what was happening on the TV. Hence the **ability to understand and perceive** demonstrates intelligence.

1.2 Intelligent Machines

The discussion in the above section has a lot of consequences when we see it with a different perspective. Let us show you something really interesting now and hence informally define the field of Artificial Intelligence at the same time.
What if?

- A machine searches through a mesh and finds a path?
- A machine solves problems like the next number in the sequence?
- A machine develops plans?
- A machine diagnoses and prescribes?
- A machine answers ambiguous questions?
- A machine recognizes fingerprints?
- A machine understands?
- A machine perceives?
- A machine does MANY MORE SUCH THINGS!
- A machine behaves as HUMANS do? **HUMANOID!!!**

We will have to call such a machine **Intelligent**. Is this real or natural intelligence? **NO!** This is **Artificial Intelligence**.

1.3 Formal Definitions for Artificial Intelligence

In their book "Artificial Intelligence: A Modern Approach" Stuart Russell and Peter Norvig comment on artificial intelligence in a very comprehensive manner. They present the definitions of artificial intelligence according to eight recent textbooks. These definitions can be broadly categorized under two themes. The ones in the left column of the table below are concerned with **thought process** and **reasoning**, where as the ones in the right column address **behavior**.

Systems that think like humans	**Systems that act like humans**
"The exciting new effort to make computers think … machines with minds, in the full and literal sense" (Haugeland, 1985)	"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil 1990)
"[The automation of] activities that we associate with human thinking, activities such	"The study of how to make computers do things at which, at the moment, people are

as decision making, problem solving, learning …” (Bellman, 1978)	better” (Rich and Knight, 1991)
“The study of mental faculties through the use of computational models” (Charniak and McDermott)	“A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes” (Schalkoff, 1990)
“The study of computation that make it possible to perceive reason and act” (Winston 1992)	“The branch of computer science that is concerned with the automation of intelligent behavior” (Luger and Stubblefield, 1993)

To make computers think like humans we first need to devise ways to determine that how humans think. This is not that easy. For this we need to get inside the actual functioning of the human brain. There are two ways to do this:

- Introspection: that is trying to catch out own thoughts as they go by.
- Psychological Experiments: that concern with the study of science of mental life.

Once we accomplish in developing some sort of comprehensive theory that how humans think, only then can we come up with computer programs that follow the same rules. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the working of human mind.

The issue of acting like humans comes up when AI programs have to interact with people or when they have to do something physically which human usually do in real life. For instance when a natural language processing system makes a dialog with a person, or when some intelligent software gives out a medical diagnosis, or when a robotic arm sorts out manufactured goods over a conveyer belt and many other such scenarios.

Keeping in view all the above motivations let us give a fairly comprehensive comment that Artificial Intelligence is an effort to create systems that can learn, think, perceive, analyze and act in the same manner as real humans.

People have also looked into understanding the phenomena of Artificial Intelligence from a different view point. They call this strong and weal AI.

Strong AI means that machines act intelligently and they have real conscious minds. Weak AI says that machines **can be made** to act as if they are intelligent. That is Weak AI treats the brain as a black box and just emulates its functionality. While strong AI actually tries to recreate the functions of the inside of the brain as opposed to simply emulating behavior.

The concept can be explained by an example. Consider you have a very intelligent machine that does a lot of tasks with a lot of intelligence. On the other hand you have very trivial specie e.g. a cat. If you throw both of them into a pool of water, the cat will try to save her life and would swim out of the pool. The "intelligent" machine would die out in the water without any effort to save itself. The mouse had strong Intelligence, the machine didn't. If the machine has strong artificial intelligence, it would have used its knowledge to counter for this totally new situation in its environment. But the machine only knew what we taught it or in other wards only knew what was programmed into it. It never had the inherent capability of intelligence which would have helped it to deal with this new situation.

Most of the researchers are of the view that strong AI can't actually ever be created and what ever we study and understand while dealing with the field of AI is related to weak AI. A few are also of the view that we can get to the essence of strong AI as well. However it is a standing debate but the purpose was to introduce you with another aspect of thinking about the field.

1.4 History and Evolution of Artificial Intelligence

AI is a young field. It has inherited its ideas, concepts and techniques from many disciplines like philosophy, mathematics, psychology, linguistics, biology etc. From over a long period of traditions in philosophy theories of reasoning and learning have emerged. From over 400 years of mathematics we have formal theories of logic, probability, decision-making and computation. From psychology we have the tools and techniques to investigate the human mind and ways to represent the resulting theories. Linguistics provides us with the theories of structure and meaning of language. From biology we have information about the network structure of a human brain and all the theories on functionalities of different human organs. Finally from computer science we have tools and concepts to make AI a reality.

1.4.1 First recognized work on AI

The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943). Their work based on three sources:

- The basic physiology and function of neurons in the human brain
- The prepositional logic
- The Turing's theory of computation

The proposed an artificial model of the human neuron. Their model proposed a human neuron to be a bi-state element i.e. on or off and that the state of the neuron depending on response to stimulation by a sufficient number of neighboring neurons. They showed, for example, that some network of connected neurons could compute any computable function, and that all the logical connectives can be implemented by simple net structures. They also suggested that suitably connected networks can also learn but they didn't pursue this idea much at that time. Donald Hebb (1949) demonstrated a simple updating rile for the modifying the connection strengths between neurons, such that learning could take place.

1.4.2 The name of the field as "Artificial Intelligence"

In 1956 some of the U.S researchers got together and organized a two-month workshop at Dartmouth. There were altogether only 10 attendees. Allen Newell and Herbert Simon actually dominated the workshop. Although all the researchers had some excellent ideas and a few even had some demo programs like checkers, but Newell and Herbert already had a reasoning program, the Logic Theorist. The program came up with proofs for logic theorems. The Dartmouth workshop didn't lead to any new breakthroughs, but it did all the major people who were working in the field to each other. Over the next twenty years these people, their students and colleagues at MIT, CMU, Stanford and IBM, dominated the field of artificial intelligence. The most lasting and memorable thing that came out of that workshop was an agreement to adopt the new name for the field: **Artificial Intelligence**. So this was when the term was actually coined.

1.4.3 First program that thought humanly

In the early years AI met drastic success. The researchers were highly motivated to try out AI techniques to solve problems that were not yet been solved. Many of them met great successes. Newell and Simon's early success was followed up with the General Problem Solver. Unlike Logic Theorist, this program was developed in the manner that it attacked a problem imitating the steps that human take when solving a problem. Though it catered for a limited class of problems but it was found out that it addressed those problems in a way very similar to that as humans. It was probably the first program that imitated human thinking approach.

1.4.4 Development of Lisp

In 1958 In MIT AI Lab, McCarthy defined the high-level language **Lisp** that became the dominant AI programming language in the proceeding years. Though McCarthy had the required tools with him to implement programs in this language but access to scarce and expensive computing resources were also a serious problem. Thus he and other researchers at MIT invented time sharing. Also in 1958 he published a paper titled **Programs with Common Sense**, in which he mentioned **Advice Taker** a hypothetical that can be seen as the first complete AI system. Unlike the other systems at that time, it was to cater for the general knowledge of the world. For example he showed that how some simple rules could help a program generate a plan to drive to an airport and catch the plane.

1.4.5 Microworlds

Marvin Minsky (1963), a researcher at MIT supervised a number of students who chose limited problems that appeared to require intelligence to solve. These limited domains became known as **Microworlds.** Some of them developed programs that solved calculus problems; some developed programs, which were able to accept input statements in a very restricted subset of English language, and generated answers to these statements. An example statement and an answer can be:

Statement:

If Ali is 2 years younger than Umar and Umar is 23 years old. How old is Ali?

Answer:

Ali is 21 years old.

In the same era a few researchers also met significant successes in building neural networks but neural networks will be discusses in detail in the section titled "**Learning**" in this book.

1.4.6 Researchers started to realize problems

In the beginning the AI researchers very confidently predicted their up coming successes. Herbert Simon in 1957 said:

It is not my aim to surprise of shock you -- but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until -- in a visible future – the range of problems they can handle will be coextensive with the range to which human mind has been applied

In 1958 he predicted that computers would be chess champions, and an important new mathematical theorem would be proved by machine. But over the years it was revealed that such statements and claims were really optimistic. A major problem that AI researchers started to realize was that though their techniques worked fairly well on one or two simple examples but most of them turned out to fail when tried out on wider selection of problems and on more difficult tasks.

One of the problems was that early programs often didn't have much knowledge of their subject matter, and succeeded by means of simple syntactic manipulations e.g. Weizenbaum's **ELIZA** program (1965), which could apparently engage in serious conversation on any topic, actually just borrowed and manipulated the sentences typed into it by a human. Many of the language translation programs tried to translate sentences by just a replacement of words without having catered for the context in which they were used, hence totally failing to maintain the subject matter in the actual sentence, which was to be translated. The famous retranslation of "**the spirit is willing but the flesh is weak**" as "**the vodka is good but the meat is rotten**" illustrates the difficulties encountered.

Second kind of difficulty was that many problems that AI was trying to solve were intractable. Most of the AI programs in the early years tried to attack a problem by finding different combinations in which a problem can be solved and then combined different combinations and steps until the right solution was found. **This didn't work always**. There were many intractable problems in which this approach failed.

A third problem arose because of the fundamental limitations on the basic structures being used to generate intelligent behavior. For example in 1969, Minsky and Papert's book **Perceptrons** proved that although perceptrons could be shown to learn anything they were capable of representing, they could represent very little.

However, in brief different happenings made the researchers realize that as they tried harder and more complex problem the pace of their success decreased so they now refrained from making highly optimistic statements.

1.4.7 AI becomes part of Commercial Market

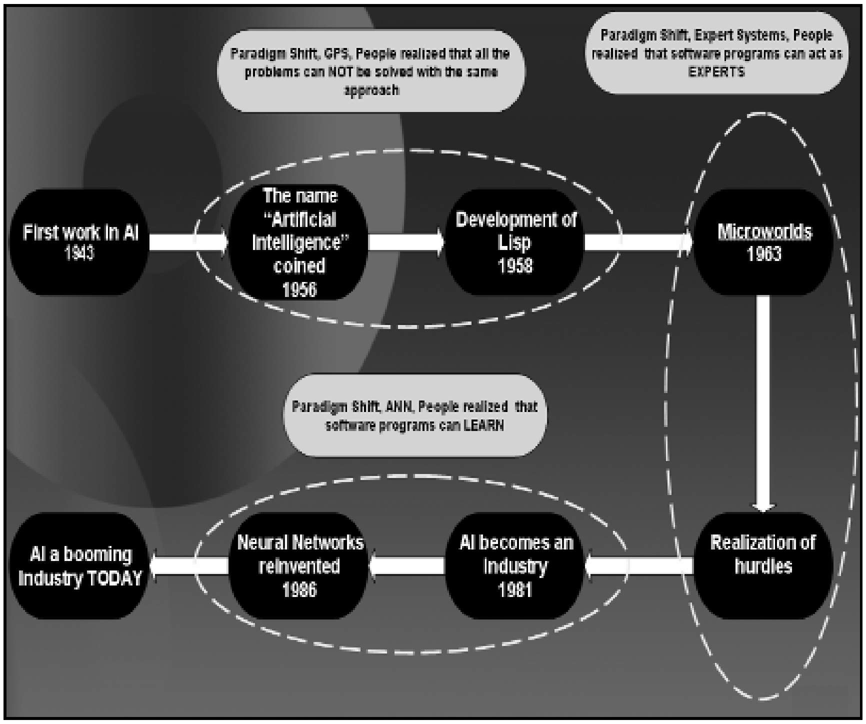
Even after realizing the basic hurdles and problems in the way of achieving success in this field, the researchers went on exploring grounds and techniques. The first successful commercial **expert system, R1**, began operation at Digital Equipment Corporation (McDermott, 1982). The program basically helped to configure the orders for new computer systems. Detailed study of what expert systems are will be dealt later in this book. For now consider expert systems as a programs that somehow solves a certain problem by using previously stored information about some rules and fact of the domain to which that problem belongs.

In 1981, the Japanese announced the “Fifth Generation” project, a 10-year plan to build intelligent computers running Prolog in much the same way that ordinary computers run the machine code. The project proposed to achieve full-scale natural language understanding along with many other ambitious goals. However, by this time people began to invest in this field and many AI projects got commercially funded and accepted.

1.4.8 Neural networks reinvented

Although computer science had rejected this concept of neural networks after Minsky and Papert’s **Perceptrons** book, but in 1980s at least four different groups reinvented the back propagation learning algorithm which was first found in 1969 by Bryson and Ho. The algorithm was applied to many learning problem in computer science and the wide spread dissemination of the results in the collection **Parallel Distributed Processing** (Rumelhart and McClelland, 1986) caused great excitement.

People tried out the back propagation neural networks as a solution to many learning problems and met great success.


Paradigm Shift, GPS, People realized that all the problems can NOT be solved with the same approach
Paradigm Shift, Expert Systems, People realized that software programs can act as EXPERTS
First work in AI 1943
The name "Artificial Intelligence" coined 1956
Development of Lisp 1958
Microworlds 1963
Paradigm Shift, ANN, People realized that software programs can LEARN
AI a booming Industry TODAY
Neural Networks reinvented 1986
AI becomes an Industry 1981
Realization of hurdles

The diagram above summarizes the history and evolution of AI in a comprehensive shape.

1.5 Applications

Artificial finds its application in a lot of areas not only related to computer sciences but many other fields as well. We will briefly mention a few of the application areas and throughout the content of this booklet you will find various applications of the field in detail later.

Many information retrieval systems like Google search engine uses artificially intelligent crawlers and content based searching techniques to efficiency and accuracy of the information retrieval.

A lot of computer based games like chess, 3D combat games even many arcade games use intelligent software to make the user feel as if the machine on which that game is running is intelligent.

Computer Vision is a new area where people are trying to develop the sense of visionary perception into a machine. Computer vision applications help to establish tasks which previously required human vision capabilities e.g. recognizing human faces, understanding images and to interpret them, analyzing medical scan and innumerable amount of other tasks.

Natural language processing is another area which tries to make machines speak and interact with humans just like humans themselves. This requires a lot from the field of Artificial Intelligence.

Expert systems form probably the largest industrial applications of AI. Software like MYCIN and XCON/R1 has been successfully employed in medical and manufacturing industries respectively.

Robotics again forms a branch linked with the applications of AI where people are trying to develop robots which can be rather called as humanoids. Organizations have developed robots that act as pets, visitor guides etc.

In short there are vast applications of the field and a lot of research work is going on around the globe in the sub-branches of the field. Like mentioned previously, during the course of the booklet you will find details of many application of AI.

1.6 Summary

- Intelligence can be understood as a trait of some living species
- Many factors and behaviors contribute to intelligence
- Intelligent machines can be created
- To create intelligent machines we first need to understand how the real brain functions
- Artificial intelligence deals with making machines think and act like humans
- It is difficult to give one precise definition of AI
- History of AI is marked by many interesting happenings through which the field gradually evolved
- In the early years people made optimistic claims about AI but soon they realized that it’s not all that smooth
- AI is employed in various different fields like gaming, business, law, medicine, engineering, robotics, computer vision and many other fields
- This book will guide you through basic concepts and some core algorithms that form the fundamentals of Artificial Intelligence
- AI has enormous room for research and posses a diverse future

Lecture No. 4 -10

2 Problem Solving

In chapter one, we discussed a few factors that demonstrate intelligence. Problem solving was one of them when we referred to it using the examples of a mouse searching a maze and the next number in the sequence problem.
Historically people viewed the phenomena of intelligence as strongly related to problem solving. They used to think that the person who is able to solve more and more problems is more intelligent than others.
In order to understand how exactly problem solving contributes to intelligence, we need to find out how intelligent species solve problems.

2.1 Classical Approach

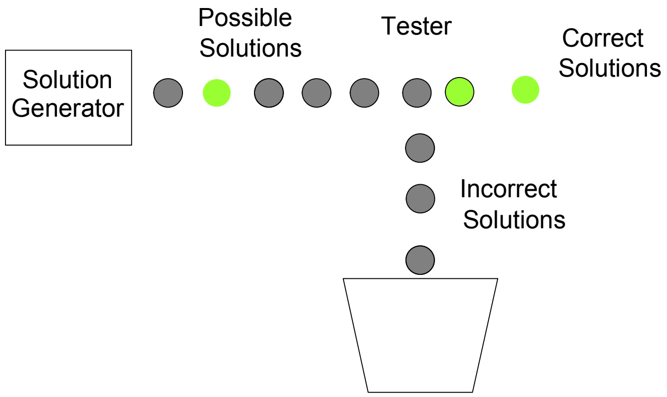
The classical approach to solving a problem is pretty simple. Given a problem at hand use **hit and trial method** to check for various solutions to that problem. This hit and trial approach usually works well for trivial problems and is referred to as the classical approach to problem solving.

Consider the maze searching problem. The mouse travels though one path and finds that the path leads to a dead end, it then back tracks somewhat and goes along some other path and again finds that there is no way to proceed. It goes on performing such search, trying different solutions to solve the problem until a sequence of turns in the maze takes it to the cheese. Hence, of all the solutions the mouse tries, the one that reached the cheese was the one that solved the problem.
Consider that a toddler is to switch on the light in a dark room. He sees the switchboard having a number of buttons on it. He presses one, nothing happens, he presses the second one, the fan gets on, he goes on trying different buttons till at last the room gets lighted and his problem gets solved.
Consider another situation when we have to open a combinational lock of a briefcase. It is a lock which probably most of you would have seen where we have different numbers and we adjust the individual dials/digits to obtain a combination that opens the lock. However, if we don't know the correct combination of digits that open the lock, we usually try 0-0-0, 7-7-7, 7-8-6 or any such combination for opening the lock. We are solving this problem in the same manner as the toddler did in the light switch example.

All this discussion has one thing in common. That different intelligent species use a similar approach to solve the problem at hand. This approach is essentially the classical way in which intelligent species solve problems. Technically we call this hit and trial approach the "Generate and Test" approach.

2.2 Generate and Test

This is a technical name given to the classical way of solving problems where we generate different combinations to solve our problem, and the one which solves the problem is taken as the correct solution. The rest of the combinations that we try are considered as incorrect solutions and hence are destroyed.

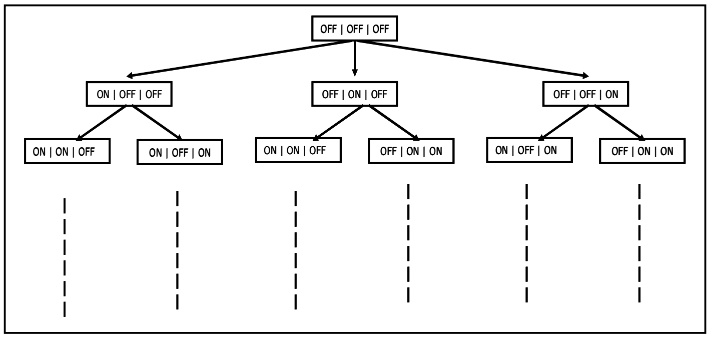


The diagram above shows a simple arrangement of a Generate and Test procedure. The box on the left labeled **"Solution Generator"** generates different solutions to a problem at hand, e.g. in the case of maze searching problem, the solution generator can be thought of as a machine that generates different paths inside a maze. The **"Tester"** actually checks that either a possible solution from the solution generates solves out problem or not. Again in case of maze searching the tester can be thought of as a device that checks that a path is a valid path for the mouse to reach the cheese. In case the tester verifies the solution to be a valid path, the solution is taken to be the "**Correct Solution**". On the other hand if the solution was incorrect, it is discarded as being an "**Incorrect Solution**".

2.3 Problem Representation

All the problems that we have seen till now were trivial in nature. When the magnitude of the problem increases and more parameters are added, e.g. the problem of developing a time table, then we have to come up with procedures better than simple Generate and Test approach.

Before even thinking of developing techniques to systematically solve the problem, we need to know one more thing that is true about problem solving namely problem representation. The key to problem solving is actually good representation of a problem. Natural representation of problems is usually done using graphics and diagrams to develop a clear picture of the problem in your mind. As an example to our comment consider the diagram below.



It shows the problem of switching on the light by a toddler in a graphical form. Each rectangle represents the state of the switch board. OFF | OFF| OFF means that all the three switches are OFF. Similarly OFF| ON | OFF means that the first and the last switch is OFF and the middle one is ON. Starting from the state when all the switches are OFF the child can proceed in any of the three ways by switching either one of the switch ON. This brings the toddler to the next level in the tree. Now from here he can explore the other options, till he gets to a state where the switch corresponding to the light is ON. Hence our problem was reduced to finding a node in the tree which ON is the place corresponding to the light switch. Observe how representing a problem in a nice manner clarifies the approach to be taken in order to solve it.

2.4 Components of Problem Solving

Let us now be a bit more formal in dealing with problem solving and take a look at the topic with reference to some components that constitute problem solving. They are namely: Problem Statement, Goal State, Solution Space and Operators. We will discuss each one of them in detail.

2.4.1 Problem Statement

This is the very essential component where by we get to know what exactly the problem at hand is. The two major things that we get to know about the problem is the Information about what is to be done and constraints to which our solution should comply. For example we might just say that given infinite amount of time, one will be able to solve any problem he wishes to solve. But the constraint "**infinite amount of time**" is not a practical one. Hence whenever addressing a problem we have to see that how much time shall out solution take at max. Time is not the only constraint. Availability of resources, and all the other parameters laid down in the problem statement actually tells us about all the rules that have to be followed while solving a problem. For example, taking the same example of the mouse, are problem statement will tell us things like, the mouse has to reach the cheese as soon as possible and in case it is unable to find a path within an hour, it might die of hunger. The statement might as well tell us that the mouse is located in the lower left corner of the maze and the cheese in the top left corner, the mouse can turn left, right and might or might not be allowed to move backward and things like that. Thus it is the problem statement that gives us a feel of what exactly to do and helps us start thinking of how exactly things will work in the solution.

2.4.2 Problem Solution

While solving a problem, this should be known that what will be out ultimate aim. That is what should be the output of our procedure in order to solve the problem. For example in the case of mouse, the ultimate aim is to reach the cheese. The state of world when mouse will be beside the cheese and probably eating it defines the aim. This state of world is also referred to as the **Goal State** or the state that represents the solution of the problem.

2.4.3 Solution space

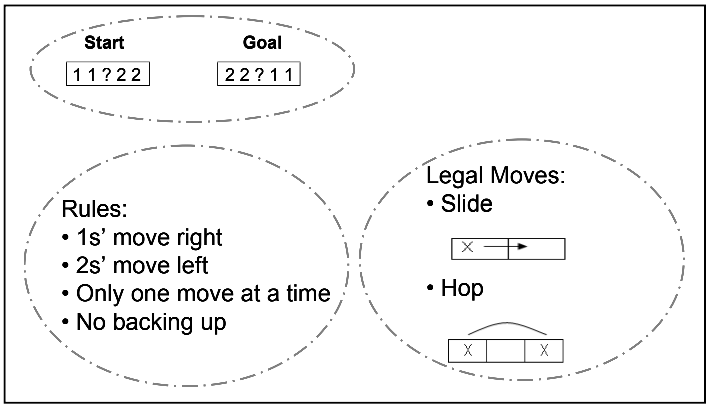
In order to reach the solution we need to check various strategies. We might or might not follow a systematic strategy in all the cases. Whatever we follow, we have to go though a certain amount of states of nature to reach the solution. For example when the mouse was in the lower left corner of the maze, represents a state i.e. the start state. When it was stuck in some corner of the maze represents a state. When it was stuck somewhere else represents another state. When it was traveling on a path represents some other state and finally when it reaches the cheese represents a state called the goal state. The set of the start state, the goal state and all the intermediate states constitutes something which is called a **solution space**.

2.4.4 Traveling in the solution space

We have to travel inside this solution space in order to find a solution to our problem. The traveling inside a solution space requires something called as "**operators**". In case of the mouse example, **turn left**, **turn right**, **go straight** are the operators which help us travel inside the solution space. In short the action that takes us from one state to the other is referred to as an **operator**. So while solving a problem we should clearly know that what are the operators that we can use in order to reach the goal state from the starting state. The **sequence of these operators** is actually the solution to our problem.

2.5 The Two-One Problem

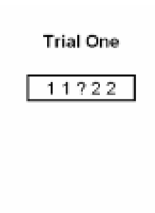
In order to explain the four components of problem solving in a better way we have chosen a simple but interesting problem to help you grasp the concepts. The diagram below shows the setting of our problem called the Two-One Problem.



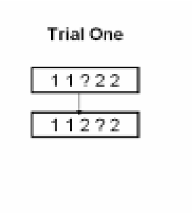
A simple problem statement to the problem at hand is as under.
You are given a rectangular container that has 5 slots in it. Each slot can hold only one coin at a time. Place Rs.1 coins in the two left slots; keep the center slot empty and put Rs.2 coins in the two right slots. A simple representation can be seen in the diagram above where the top left container represents the **Start State** in which the coined are placed as just described. Our aim is to reach a state of the container where the left two slots should contain Rs.2 coins, the center slot should be empty and the right two slots should contain Rs.1 coin as shown in the **Goal State**. There are certain simple rules to play this game. The rules are mentioned clearly in the diagram under the heading of "Rules". The rules actually define the constraints under which the problem has to be solved. The legal moves are the **Operators** that we can use to get from one state to the other. For example we can slide a coin to its left or right if the left or right slot is empty, or we can hop the coin over a single slot. The rules say that Rs.1 coins can slide or hop only towards right. Similarly the Rs.2 coins can slide or hop only towards the left. You can only move one coin at a time.

Now let us try to solve the problem in a trivial manner just by using a hit and trial method without addressing the problem in a systematic manner.

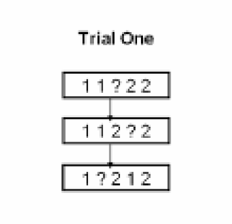
Trial 1



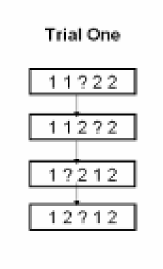
Start State



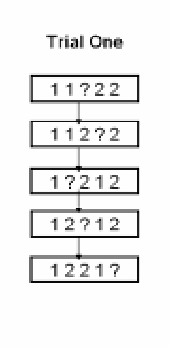
Move 1



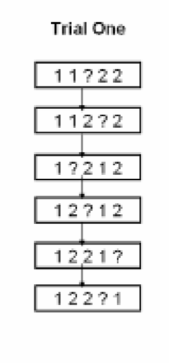
Move 2



Move 3



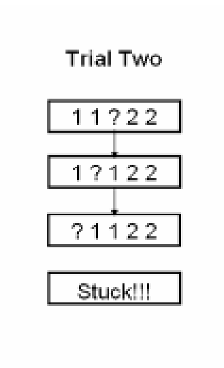
Move 4



Move 5

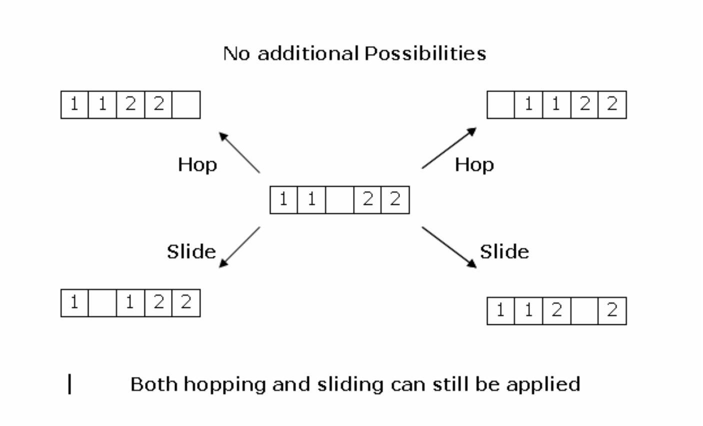
In Move 1 we slide a 2 to the left, then we hop a 1 to the right, then we slide the 2 to the left again and then we hop the 2 to the left, then slide the one to the right hence at least one 2 and one 1 are at the desired positions as required in the goal state but then we are stuck. There is no other valid move which takes us out of this state. Let us consider another trial.

Trial 2

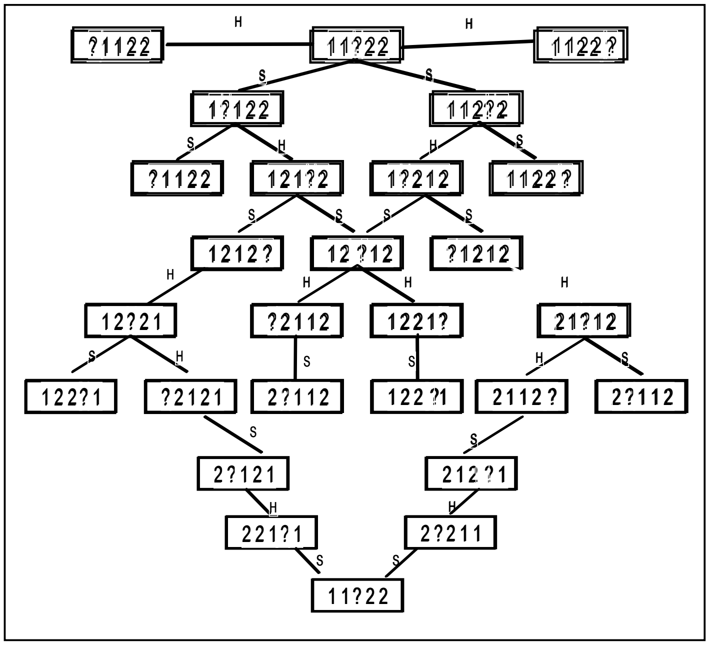


Starting from the start state we first hop a 1 to the right, then we slide the other 1 to the right and then suddenly we get **STUCK**!! Hence solving the problem through a hit and trial might not give us the solution.

Let us now try to address the problem in a systematic manner. Consider the diagram below.


No additional Possibilities
1 1 2 2
1 1 2 2
Hop
Hop
1 1 2 2
Slide
Slide
1 1 2 2
1 1 2 2
Both hopping and sliding can still be applied

Starting from the goal state if we hop, we get stuck. If we slide we can further carry on. Keeping this observation in mind let us now try to develop all the possible combinations that can happen after we slide.



The diagram above shows a tree sort of structure enumerating all the possible states and moves. Looking at this diagram we can easily figure out the solution to our problem. This tree like structure actually represents the "**Solution Space**" of this problem. The labels on the links are H and S representing hop and slide operators respectively. Hence H and S are the operators that help us travel through this solution space in order to reach the goal state from the start state.

We hope that this example actually clarifies the terms problem statement, start state, goal state, solution space and operators in your mind. It will be a nice exercise to design your own simple problems and try to identify these components in them in order to develop a better understanding.

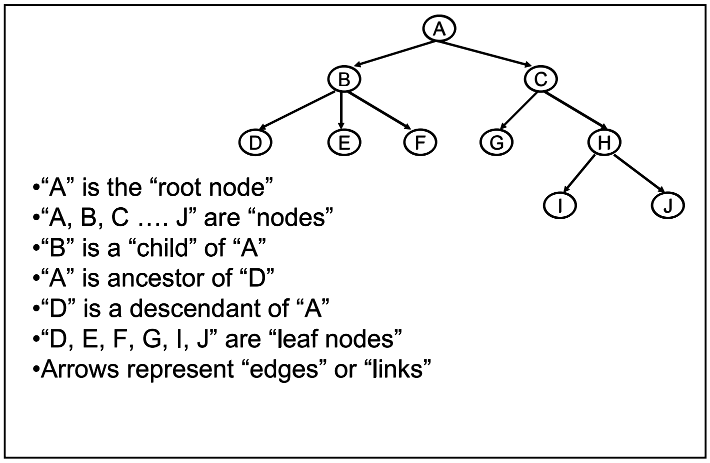
2.6 Searching

All the problems that we have looked at can be converted to a form where we have to start from a start state and **search** for a goal state by traveling through a solution space. Searching is a formal mechanism to explore alternatives.

Most of the solution spaces for problems can be represented in a graph where nodes represent different states and edges represent the operator which takes us from one state to the other. If we can get our grips on algorithms that deal with searching techniques in graphs and trees, we'll be all set to perform problem solving in an efficient manner.

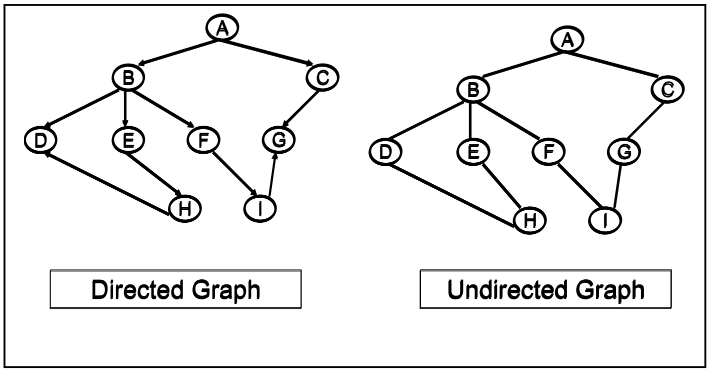
2.7 Tree and Graphs Terminology

Before studying the searching techniques defined on trees and graphs let us briefly review some underlying terminology.

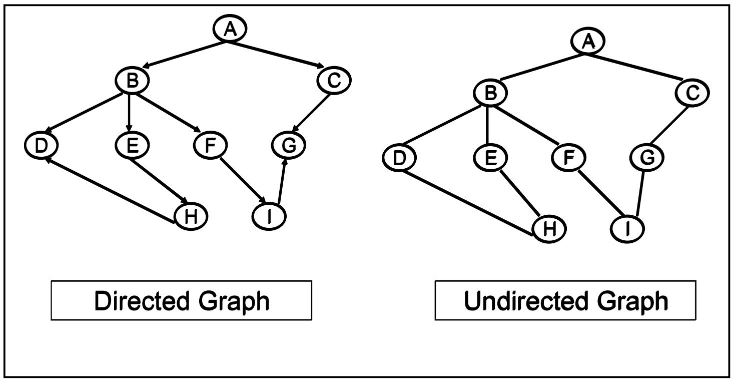


- "A" is the "root node"
- "A, B, C …. J" are "nodes"
- "B" is a "child" of "A"
- "A" is ancestor of "D"
- "D" is a descendant of "A"
- "D, E, F, G, I, J" are "leaf nodes"
- Arrows represent "edges" or "links"

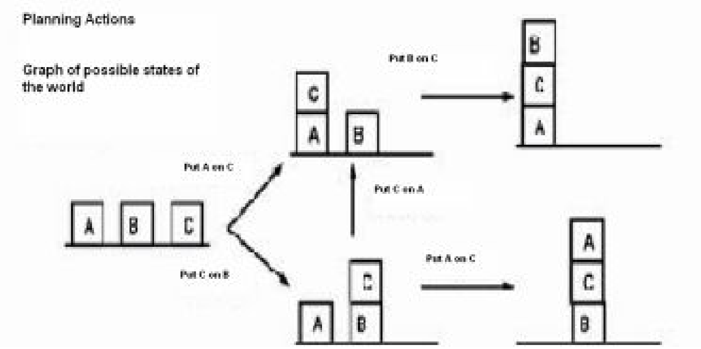
The diagram above is just to refresh your memories on the terminology of a tree. As for graphs, there are undirected and directed graphs which can be seen in the diagram below.



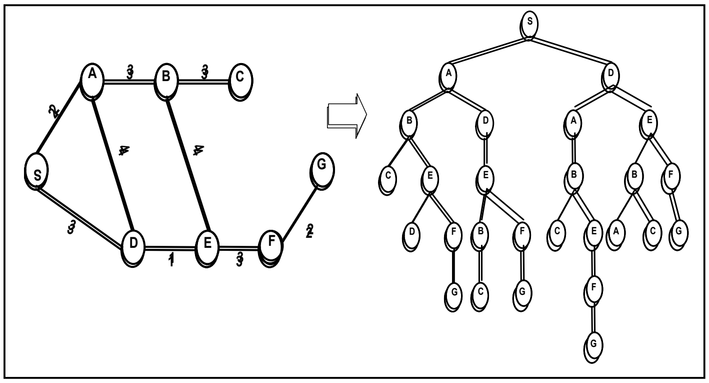
Let us first consider a couple of examples to learn how graphs can represent important information by the help of nodes and edges.
Graphs can be used to represent city routes.



Graphs can be used to plan actions.



We will use graphs to represent problems and their solution spaces. One thing to be noted is that every graph can be converted into a tree, by replicating the nodes. Consider the following example.



The graph in the figure represents a city map with cities labeled as S, A, B, C, D, E, F and G. Just by following a simple procedure we can convert this graph to a tree.

Start from node S and make it the root of your tree, check how many nodes are adjacent to it. In this case A and D are adjacent to it. Hence in the tree make A and D, children of S. Now go on proceeding in this manner and you'll get a tree with a few nodes replicated. In this manner depending on a starting node you can get a different tree too. But just recall that when solving a problem; we usually know the start state and the end state. So we will be able to transform our problem graphs in problem trees. Now if we can develop understanding of algorithms that are defined for tree searching and tree traversals then we will be in a better shape to solve problems efficiently.

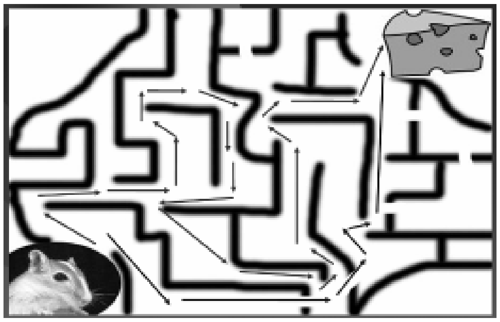
We know that problems can be represented in graphs, and are well familiar with the components of problem solving, let us now address problem solving in a more formal manner and study the searching techniques in detail so that we can systematically approach the solution to a given problem.

2.8 Search Strategies

Search strategies and algorithms that we will study are primarily of four types, blind/uninformed, informed/heuristic, any path/non-optimal and optimal path search algorithms. We will discuss each of these using the same mouse example.

Suppose the mouse does not know where and how far is the cheese and is totally blind to the configuration of the maze. The mouse would blindly search the maze without any hints that will help it turning left or right at any junction. The mouse will purely use a hit and trial approach and will check all combinations till one takes it to the cheese. Such searching is called blind or uninformed searching.

Consider now that the cheese is fresh and the smell of cheese is spread through the maze. The mouse will now use this smell as a guide, or heuristic (we will comment on this word in detail later) to guess the position of the cheese and choose the best from the alternative choices. As the smell gets stronger, the mouse knows that the cheese is closer. Hence the mouse is informed about the cheese through the smell and thus performs an informed search in the maze. For now you might think that the informed search will always give us a better solution and will always solve our problem. This might not be true as you will find out when we discuss the word heuristic in detail later.



When solving the maze search problem, we saw that the mouse can reach the cheese from different paths. In the diagram above two possible paths are shown.

In any-path/non optimal searches we are concerned with finding any one solution to our problem. As soon as we find a solution, we stop, without thinking that there might as well be a better way to solve the problem which might take lesser time or fewer operators.

Contrary to this, in optimal path searches we try to find the best solution. For example, in the diagram above the optimal path is the blue one because it is smaller and requires lesser operators. Hence in optimal searches we find solutions that are least costly, where cost of the solution may be different for each problem.

2.9 Simple Search Algorithm

Let us now state a simple search algorithm that will try to give you an idea about the sort of data structures that will be used while searching, and the stop criteria for your search. The strength of the algorithm is such that we will be able to use this algorithm for both Depth First Search (DFS) and Breadth First Search (BFS).

Let S be the start state

1. **Initialize Q with the start node Q=(S) as the only entry; set Visited = (S)**
2. **If Q is empty, fail. Else pick node X from Q**
3. **If X is a goal, return X, we've reached the goal**
4. **(Otherwise) Remove X from Q**
5. **Find all the children of state X not in Visited**
6. **Add these to Q; Add Children of X to Visited**
7. **Go to Step 2**

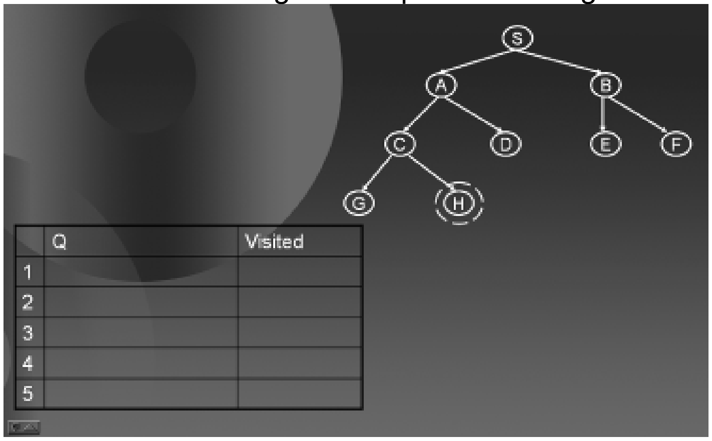
Here Q represents a priority queue. The algorithm is simple and doesn't need much explanation. We will use this algorithm to implement blind and uninformed searches. The algorithm however can be used to implement informed searches as well. The critical step in the Simple Search Algorithm is picking of a node X from Q according to a priority function. Let us call this function P(n). While using this algorithm for any of the techniques, our priority will be to reduce the value of P(n) as much as we can. In other words, the node with the highest priority will have the smallest value of the function P(n) where n is the node referred to as X in the algorithm.

2.10 Simple Search Algorithm Applied to Depth First Search

Depth First Search dives into a tree deeper and deeper to fine the goal state. We will use the same Simple Search Algorithm to implement DFS by keeping our priority function as

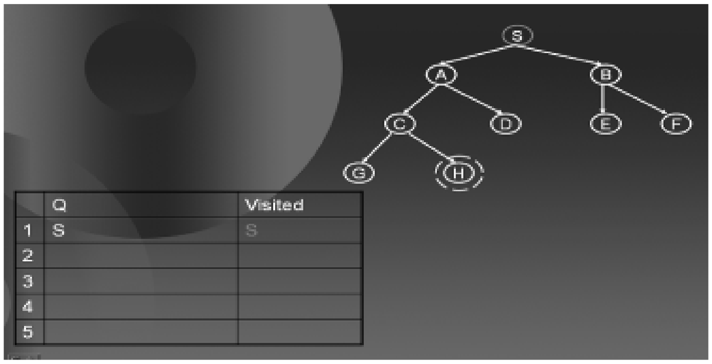
$$P(n) = \frac{1}{height(n)}$$

As mentioned previously we will give priority to the element with minimum P(n) hence the node with the largest value of height will be at the maximum priority to be picked from Q. The following sequence of diagrams will show you how DFS works on a tree using the Simple Search Algorithm.



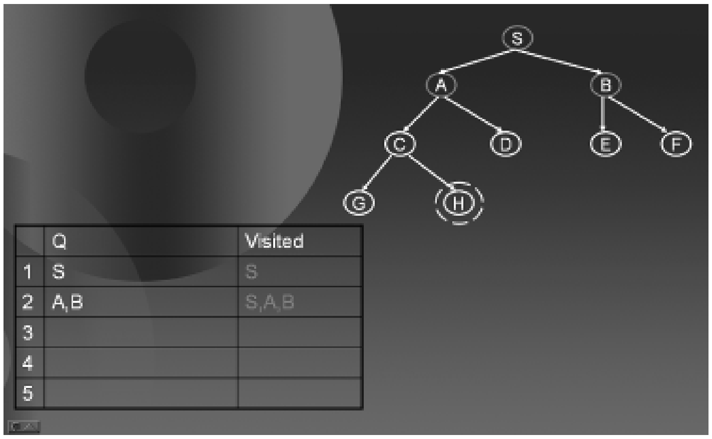
	Q	Visited
1		
2		
3		
4		
5		

We start with a tree containing nodes S, A, B, C, D, E, F, G, and H, with H as the goal node. In the bottom left table we show the two queues Q and Visited. According to the Simple Search Algorithm, we initialize Q with the start node S, shown below.



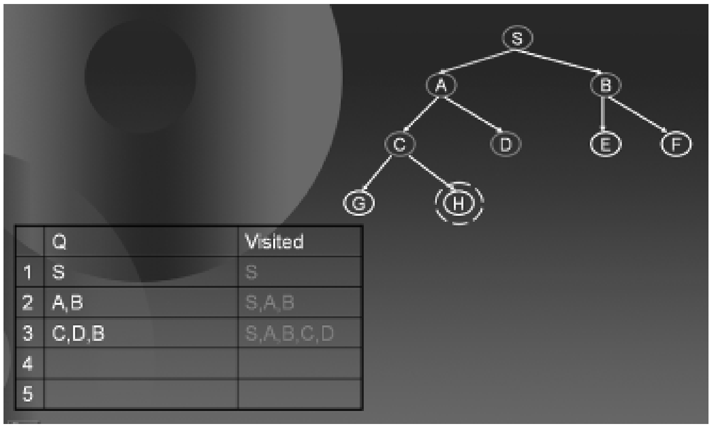
	Q	Visited
1	S	S
2		
3		
4		
5		

If Q is not empty, pick the node X with the minimum P(n) (in this case S), as it is the only node in Q. Check if X is goal, (in this case X is not the goal). Hence find all the children of X not in Visited and add them to Q and Visited. Goto Step 2.



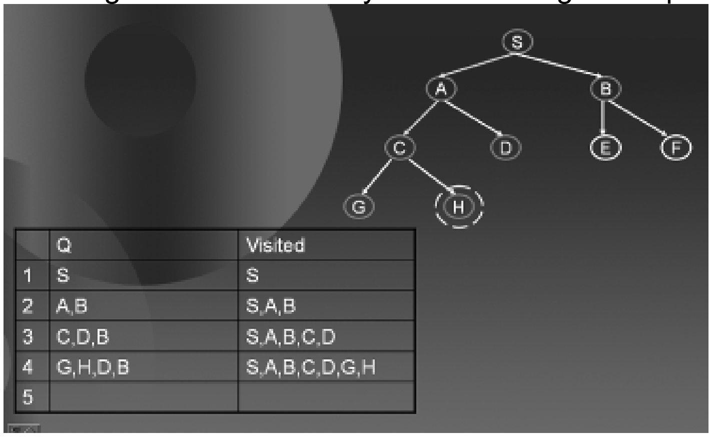
	Q	Visited
1	S	S
2	A,B	S,A,B
3		
4		
5		

Again check if Q is not empty, pick the node X with the minimum P(n) (in this case either A or B), as both of them have the same value for P(n). Check if X is goal, (in this case A is not the goal). Hence, find all the children of A not in Visited and add them to Q and Visited. Go to Step 2.

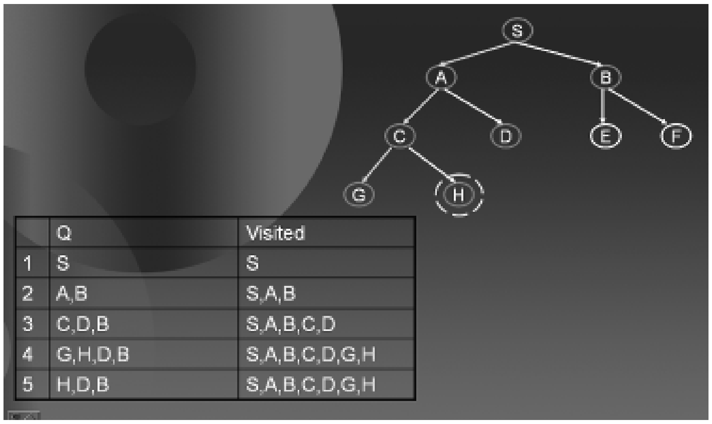


	Q	Visited
1	S	S
2	A,B	S,A,B
3	C,D,B	S,A,B,C,D
4		
5		

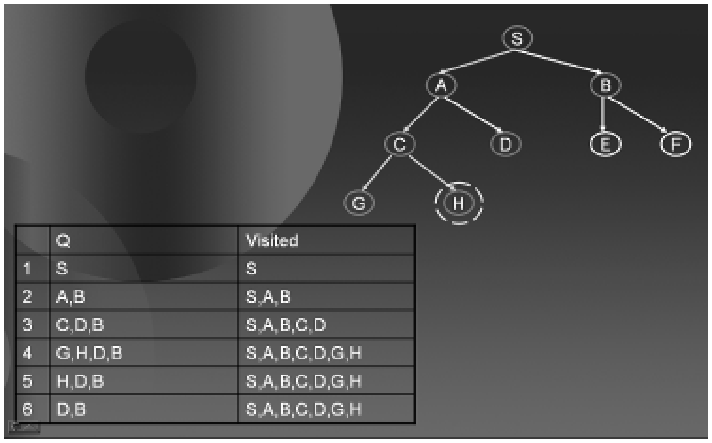
Go on following the steps in the Simple Search Algorithm till you find a goal node. The diagrams below show you how the algorithm proceeds.



	Q	Visited
1	S	S
2	A,B	S,A,B
3	C,D,B	S,A,B,C,D
4	G,H,D,B	S,A,B,C,D,G,H
5		

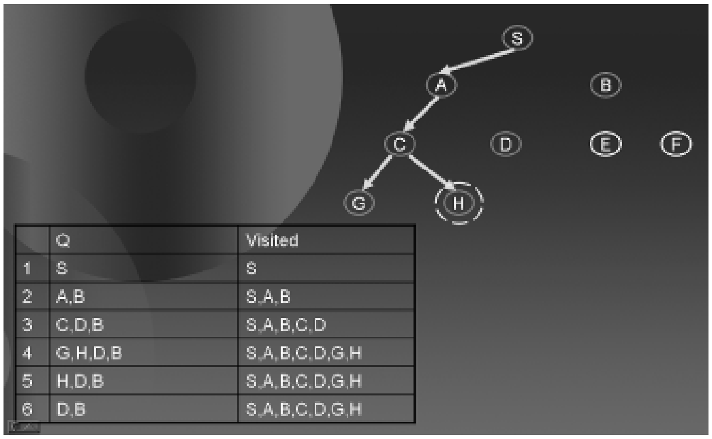


	Q	Visited
1	S	S
2	A,B	S,A,B
3	C,D,B	S,A,B,C,D
4	G,H,D,B	S,A,B,C,D,G,H
5	H,D,B	S,A,B,C,D,G,H



	Q	Visited
1	S	S
2	A,B	S,A,B
3	C,D,B	S,A,B,C,D
4	G,H,D,B	S,A,B,C,D,G,H
5	H,D,B	S,A,B,C,D,G,H
6	D,B	S,A,B,C,D,G,H

Here, from the 5th row of the table when we remove H and check if it's the goal, the algorithm says YES and hence we return H as we have reached the goal state. The path followed by the DFS is shown by green arrows at each step. The diagram below also shows that DFS didn't have to search the entire search space, rather only by traveling in half the tree, the algorithm was able to search the solution.



	Q	Visited
1	S	S
2	A,B	S,A,B
3	C,D,B	S,A,B,C,D
4	G,H,D,B	S,A,B,C,D,G,H
5	H,D,B	S,A,B,C,D,G,H
6	D,B	S,A,B,C,D,G,H

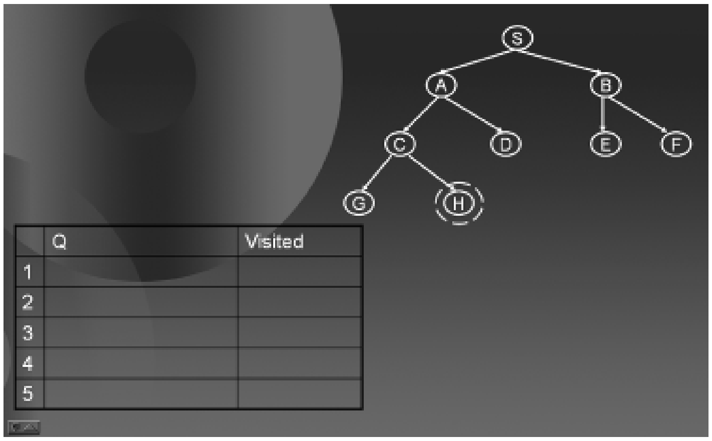
Hence simply by selecting a specific P(n) our Simple Search Algorithm was converted to a DFS procedure.

2.11 Simple Search Algorithm Applied to Breadth First Search

Breadth First Search explores the breadth of the tree first and progresses downward level by level. Now, we will use the same Simple Search Algorithm to implement BFS by keeping our priority function as

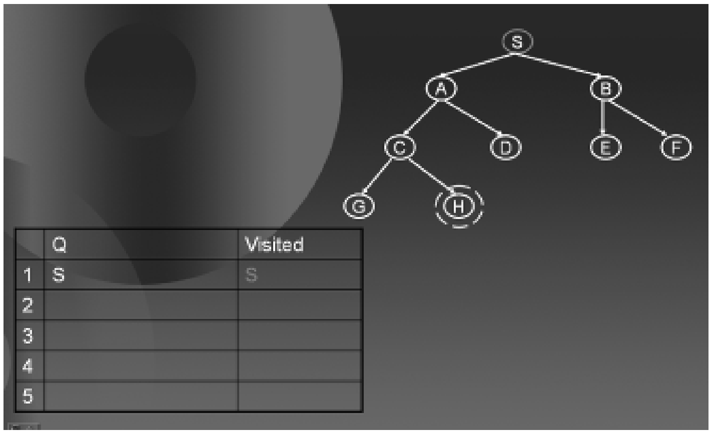
$$P(n) = height(n)$$

As mentioned previously, we will give priority to the element with minimum P(n) hence the node with the largest value of height will be at the maximum priority to be picked from **Q**. In other words, greater the depth/height greater the priority. The following sequence of diagrams will show you how BFS works on a tree using the Simple Search Algorithm.



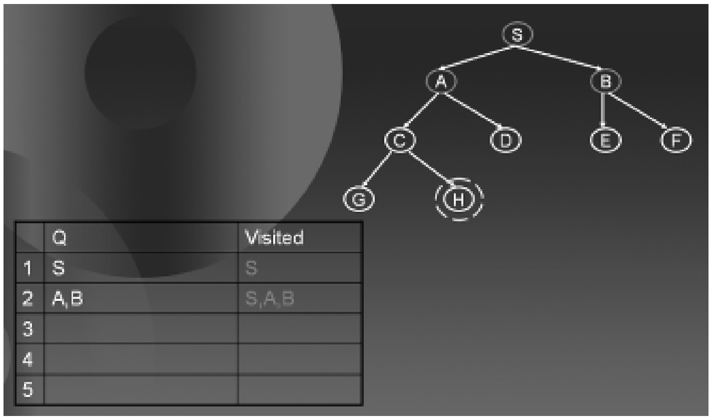
	Q	Visited
1		
2		
3		
4		
5		

We start with a tree containing nodes S, A, B, C, D, E, F, G, and H, with H as the goal node. In the bottom left table we show the two queues Q and Visited. According to the Simple Search Algorithm, we initialize Q with the start node S.



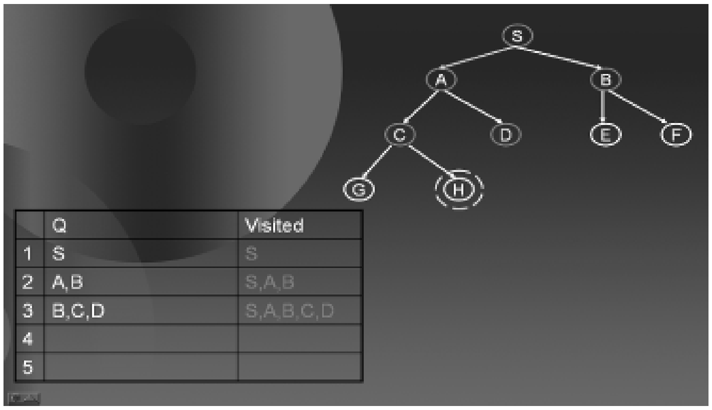
	Q	Visited
1	S	S
2		
3		
4		
5		

If Q is not empty, pick the node X with the minimum P(n) (in this case S), as it is the only node in Q. Check if X is goal, (in this case X is not the goal). Hence find all the children of X not in Visited and add them to Q and Visited. Goto Step 2.



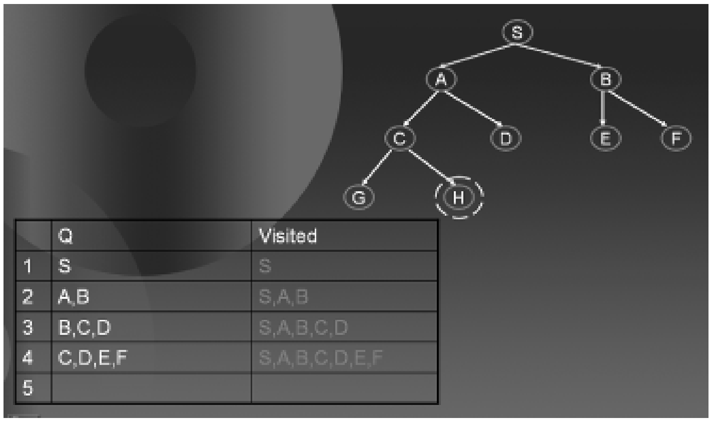
	Q	Visited
1	S	S
2	A,B	S,A,B
3		
4		
5		

Again, check if Q is not empty, pick the node X with the minimum P(n) (in this case either A or B), as both of them have the same value for P(n). Remember, n refers to the node X. Check if X is goal, (in this case A is not the goal). Hence find all the children of A not in Visited and add them to Q and Visited. Go to Step 2.

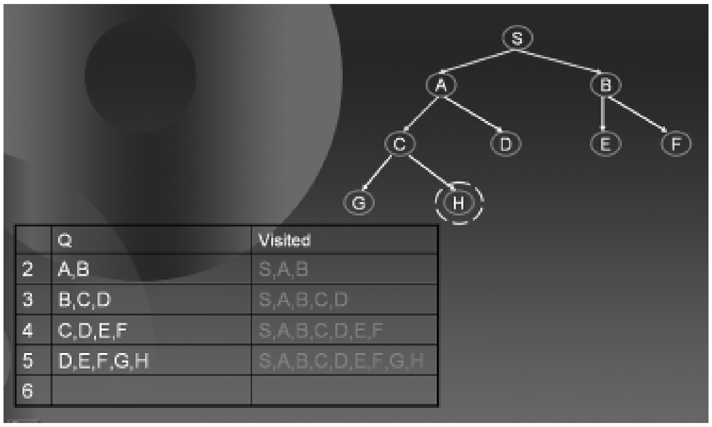


	Q	Visited
1	S	S
2	A,B	S,A,B
3	B,C,D	S,A,B,C,D
4		
5		

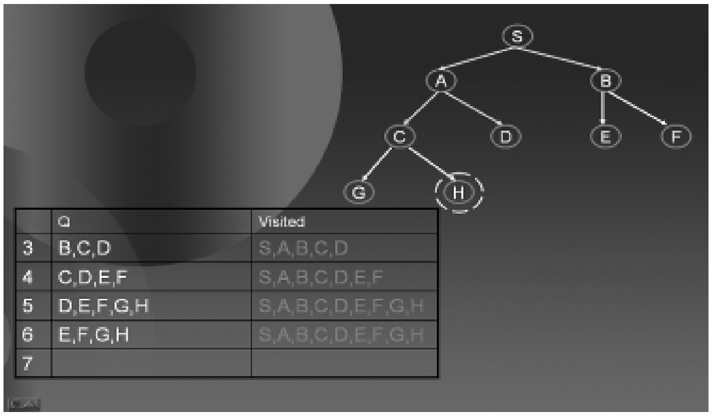
Now, we have B, C and D in the list Q. B has height 1 while C and D are at a height 2. As we are to select the node with the minimum P(n) hence we will select B and repeat. The following sequence of diagram tells you how the algorithm proceeds till it reach the goal state.



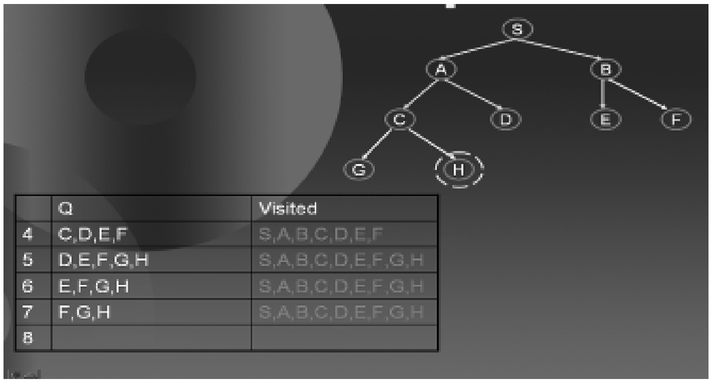
	Q	Visited
1	S	S
2	A,B	S,A,B
3	B,C,D	S,A,B,C,D
4	C,D,E,F	S,A,B,C,D,E,F
5		



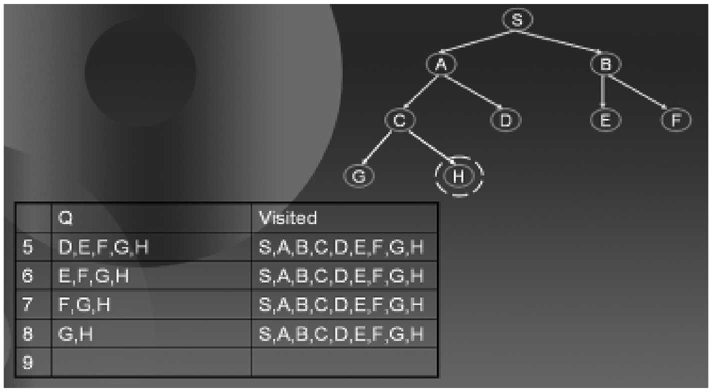
	Q	Visited
2	A,B	S,A,B
3	B,C,D	S,A,B,C,D
4	C,D,E,F	S,A,B,C,D,E,F
5	D,E,F,G,H	S,A,B,C,D,E,F,G,H
6		



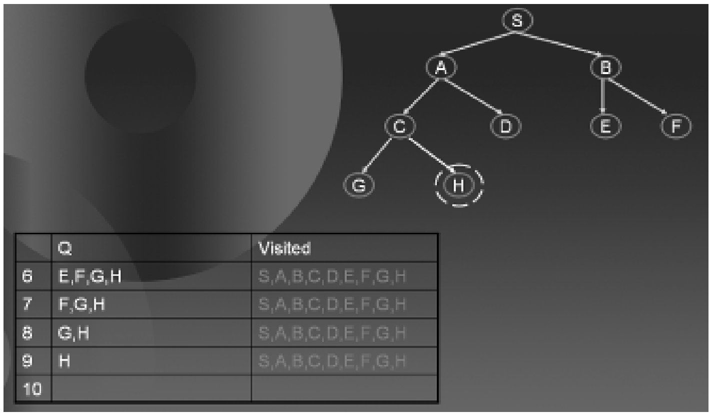
	Q	Visited
3	B,C,D	S,A,B,C,D
4	C,D,E,F	S,A,B,C,D,E,F
5	D,E,F,G,H	S,A,B,C,D,E,F,G,H
6	E,F,G,H	S,A,B,C,D,E,F,G,H
7		



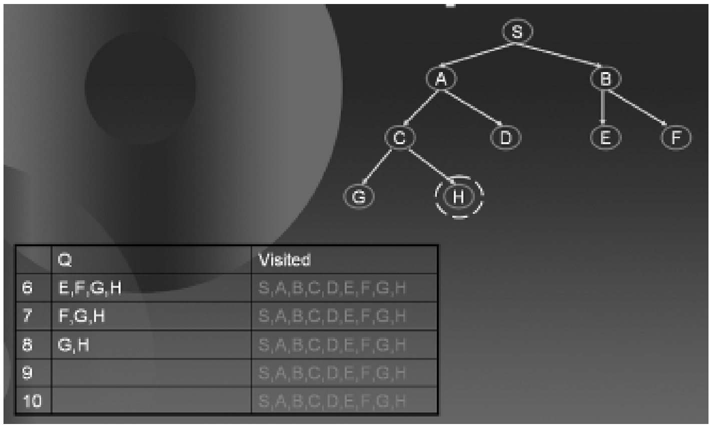
	Q	Visited
4	C,D,E,F	S,A,B,C,D,E,F
5	D,E,F,G,H	S,A,B,C,D,E,F,G,H
6	E,F,G,H	S,A,B,C,D,E,F,G,H
7	F,G,H	S,A,B,C,D,E,F,G,H
8		



	Q	Visited
5	D,E,F,G,H	S,A,B,C,D,E,F,G,H
6	E,F,G,H	S,A,B,C,D,E,F,G,H
7	F,G,H	S,A,B,C,D,E,F,G,H
8	G,H	S,A,B,C,D,E,F,G,H
9		

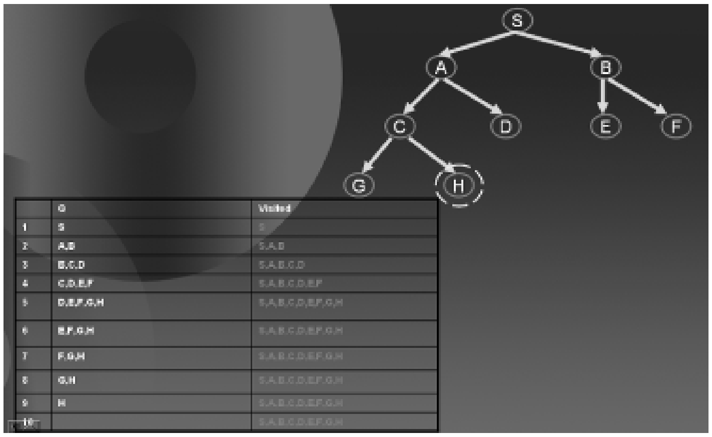


	Q	Visited
6	E,F,G,H	S,A,B,C,D,E,F,G,H
7	F,G,H	S,A,B,C,D,E,F,G,H
8	G,H	S,A,B,C,D,E,F,G,H
9	H	S,A,B,C,D,E,F,G,H
10		



	Q	Visited
6	E,F,G,H	S,A,B,C,D,E,F,G,H
7	F,G,H	S,A,B,C,D,E,F,G,H
8	G,H	S,A,B,C,D,E,F,G,H
9		S,A,B,C,D,E,F,G,H
10		S,A,B,C,D,E,F,G,H

When we remove H from the 9th row of the table and check if it's the goal, the algorithm says YES and hence we return H since we have reached the goal state. The path followed by the BFS is shown by green arrows at each step. The diagram below also shows that BFS travels a significant area of the search space if the solution is located somewhere deep inside the tree.



	Q	Visited
1	S	S
2	A,B	S,A,B
3	B,C,D	S,A,B,C,D
4	C,D,E,F	S,A,B,C,D,E,F
5	D,E,F,G,H	S,A,B,C,D,E,F,G,H
6	E,F,G,H	S,A,B,C,D,E,F,G,H
7	F,G,H	S,A,B,C,D,E,F,G,H
8	G,H	S,A,B,C,D,E,F,G,H
9	H	S,A,B,C,D,E,F,G,H
10		S,A,B,C,D,E,F,G,H

Hence, simply by selecting a specific P(n) our Simple Search Algorithm was converted to a BFS procedure.

2.12 Problems with DFS and BFS

Though DFS and BFS are simple searching techniques which can get us to the goal state very easily yet both of them have their own problems.

DFS has small space requirements (linear in depth) but has major problems:

- DFS can run forever in search spaces with infinite length paths
- DFS does not guarantee finding the shallowest goal

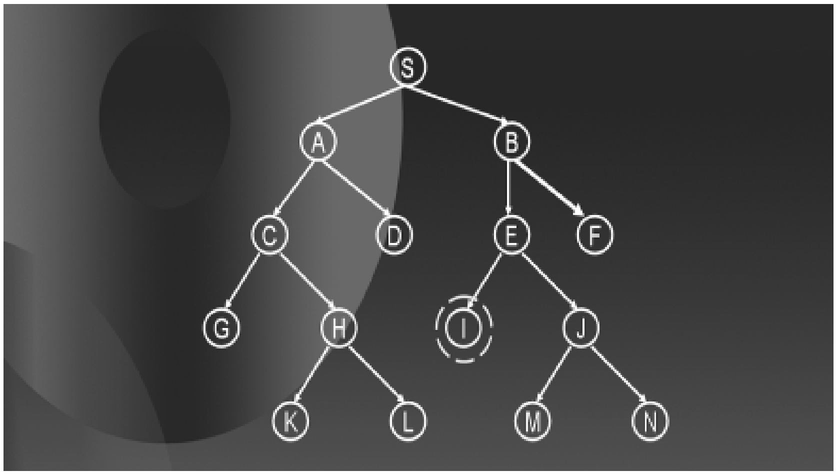
BFS guarantees finding the shallowest path even in presence of infinite paths, but it has one great problem

- BFS requires a great deal of space (exponential in depth)

We can still come up with a better technique which caters for the drawbacks of both these techniques. One such technique is progressive deepening.

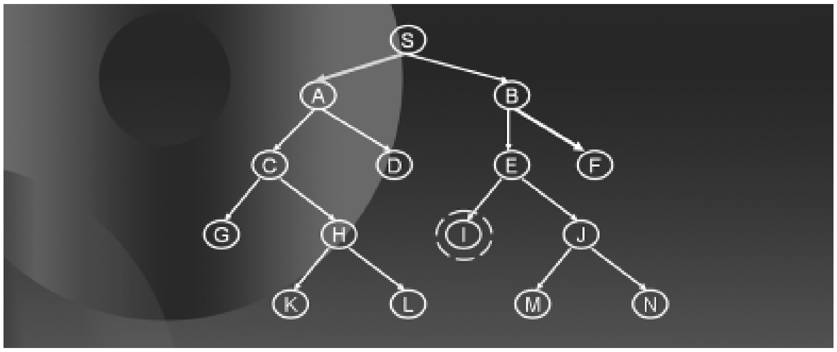
2.13 Progressive Deepening

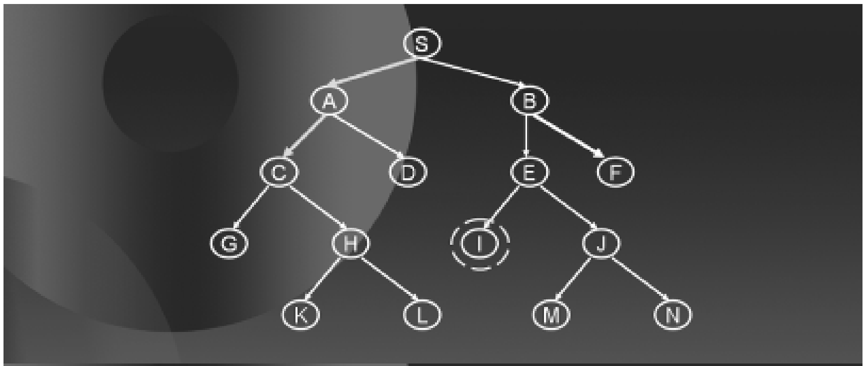
Progressive deepening actually emulates BFS using DFS. The idea is to simply apply DFS to a specific level. If you find the goal, exit, other wise repeat DFS to the next lower level. Go on doing this until you either reach the goal node or the full height of the tree is explored. For example, apply a DFS to level 2 in the tree, if it reaches the goal state, exit, otherwise increase the level of DFS and apply it again until you reach level 4. You can increase the level of DFS by any factor. An example will further clarify your understanding.

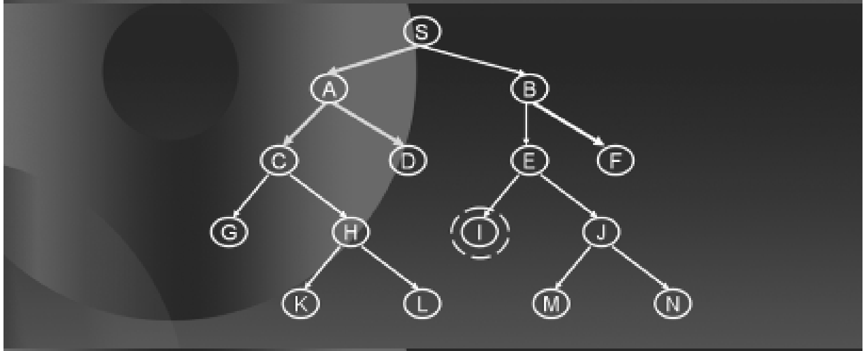


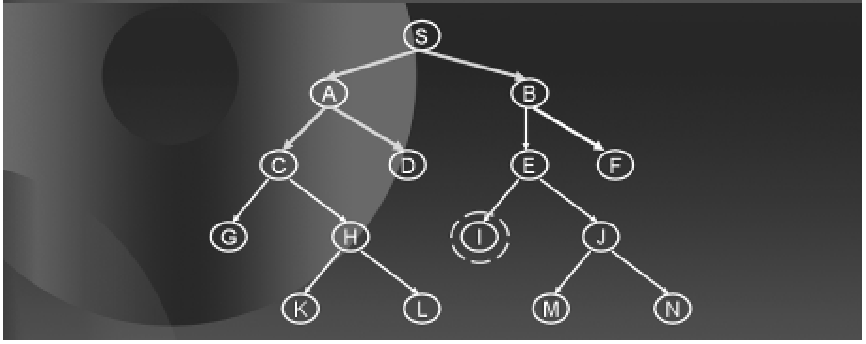
Consider the tree on the previous page with nodes from S … to N, where I is the goal node.

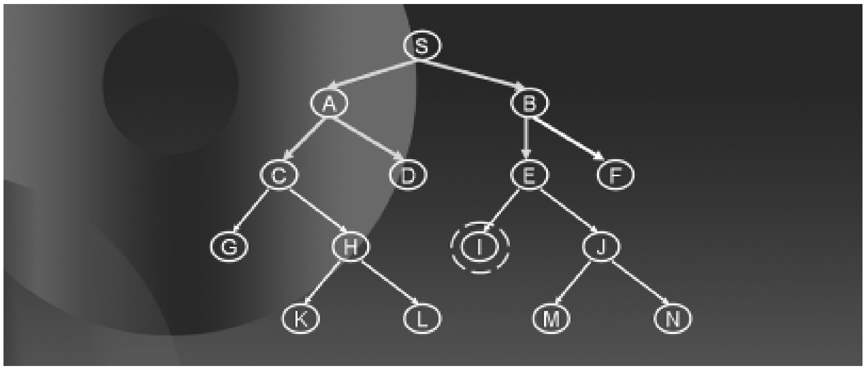
Apply DFS to level 2 in the tree. The green arrows in the diagrams below show how DFS will proceed to level 2.

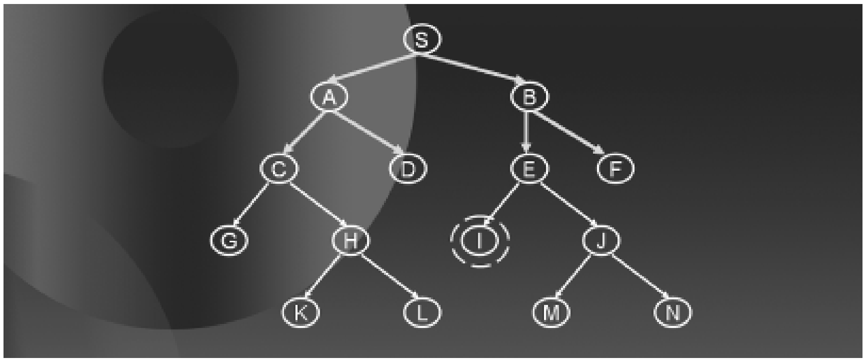

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N


S
A
B
C
D
E
F
G
H
I
J
K
L
M
N

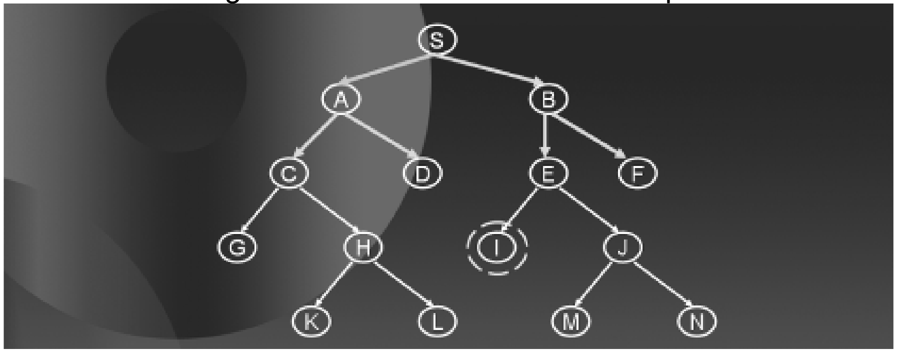

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N

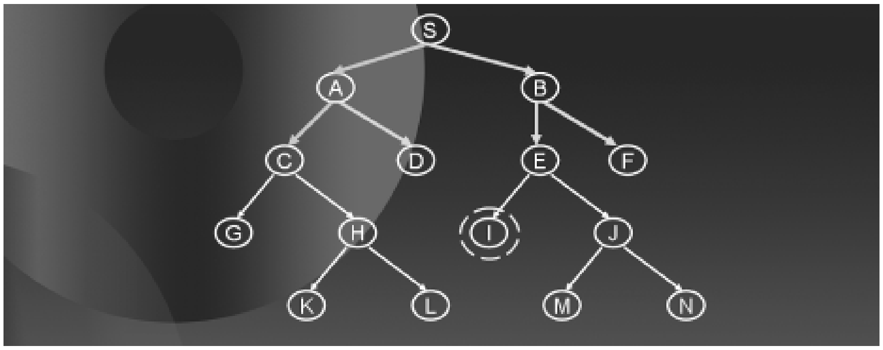

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N

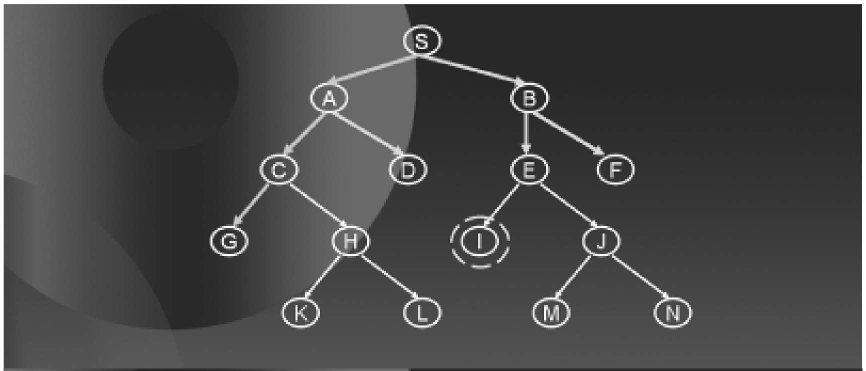


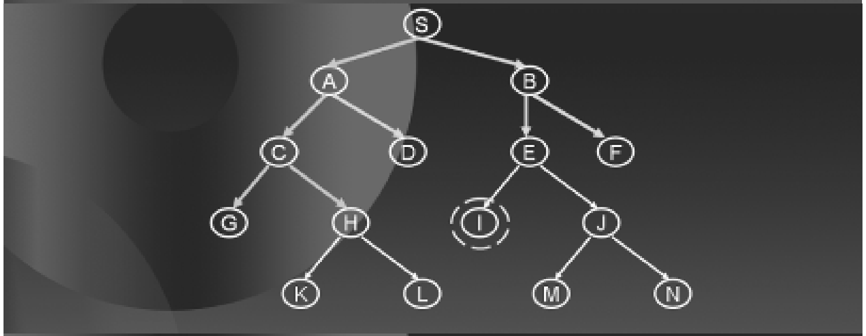


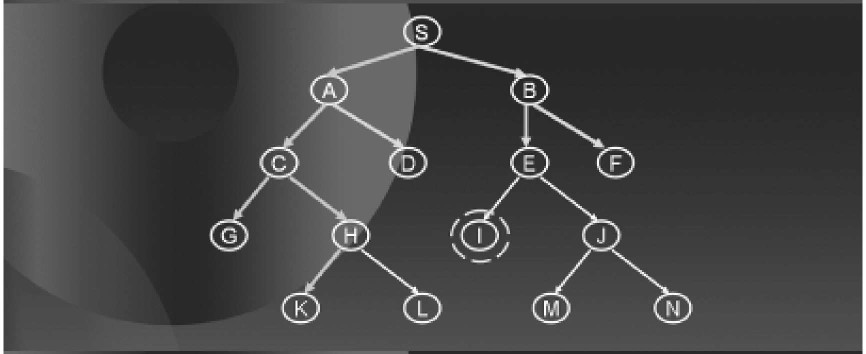
After exploring to level 2, the progressive deepening procedure will find out that the goal state has still not been reached. Hence, it will increment the level by a factor of, say 2, and will now perform a DFS in the tree to depth 4. The blue arrows in the diagrams below show how DFS will proceed to level 4.

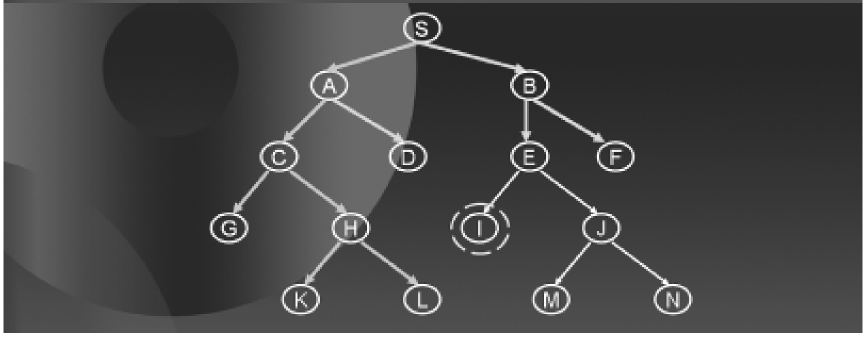


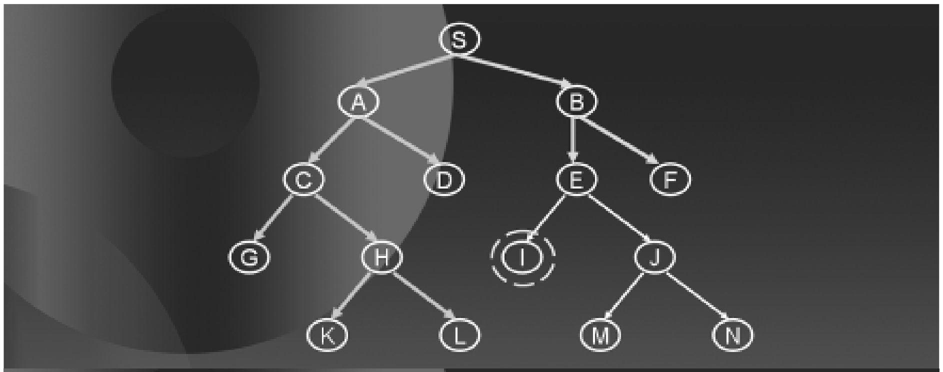


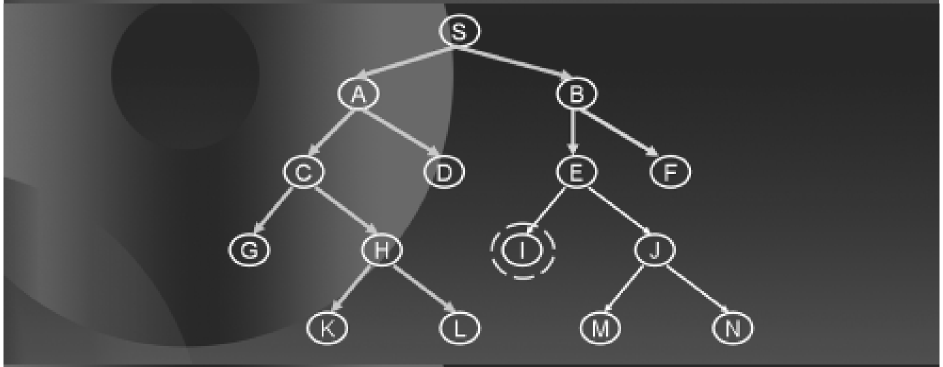

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N

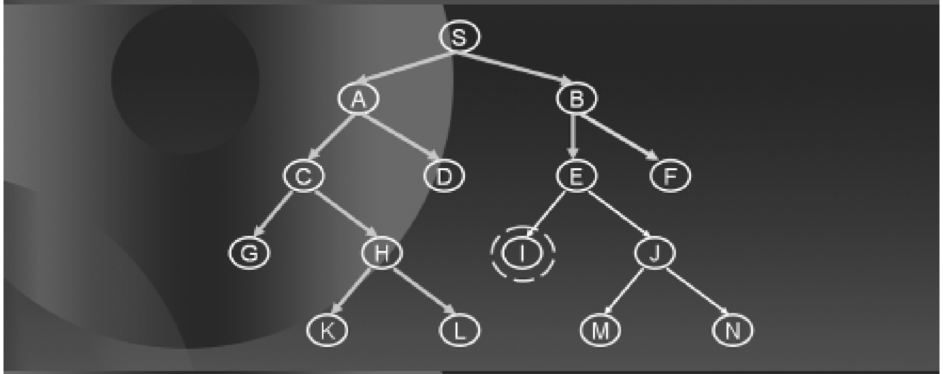

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N

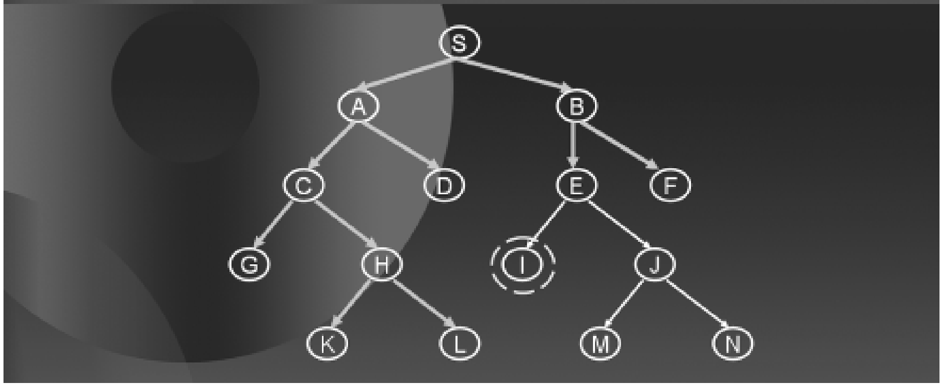

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N

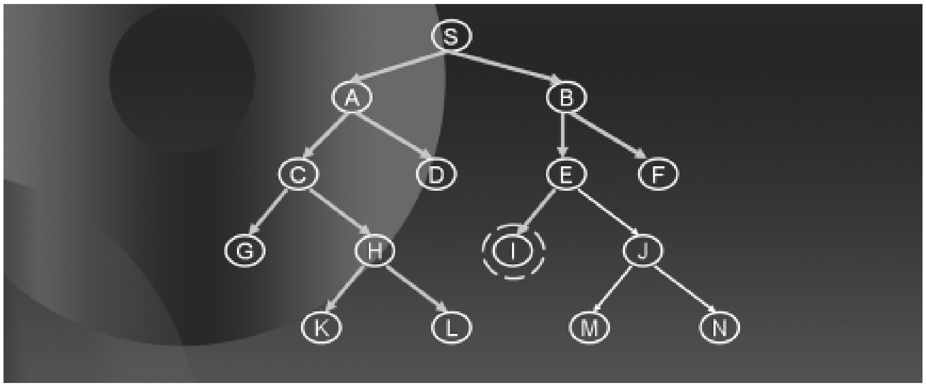

S
A
B
C
D
E
F
G
H
I
J
K
L
M
N


S
A
B
C
D
E
F
G
H
I
J
K
L
M
N


S
A
B
C
D
E
F
G
H
I
J
K
L
M
N


S
A
B
C
D
E
F
G
H
I
J
K
L
M
N


S
A
B
C
D
E
F
G
H
I
J
K
L
M
N



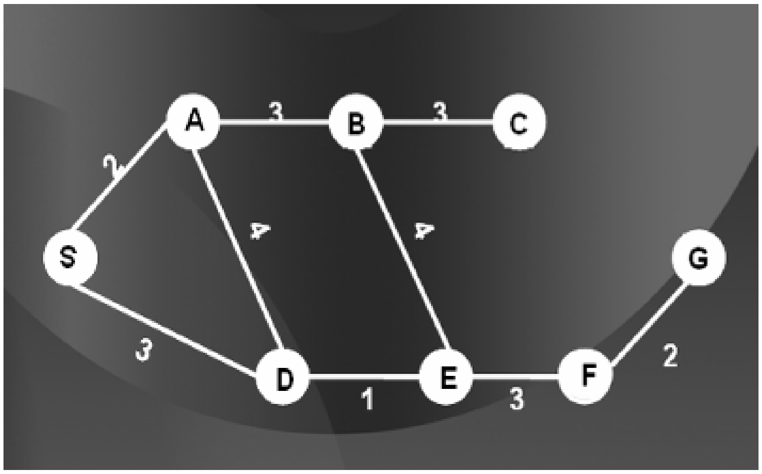
As soon as the procedure finds the goal state it will quit. Notice that it guarantees to find the solution at a minimum depth like BFS. Imagine that there are a number of solutions below level 4 in the tree. The procedure would only travel a small portion of the search space and without large memory requirements, will find out the solution.

2.14 Heuristically Informed Searches

So far we have looked into procedures that search the solution space in an uninformed manner. Such procedures are usually costly with respect to either time, space or both. We now focus on a few techniques that search the solution space in an informed manner using something which is called a heuristic. Such techniques are called heuristic searches. The basic idea of a heuristic search is that rather than trying all possible search paths, you try and focus on paths that seem to be getting you closer to your goal state using some kind of a “guide”. Of course, you generally can't be sure that you are really near your goal state. However, we might be able to use a good guess for the purpose. Heuristics are used to help us make that guess. It must be noted that heuristics don’t always give us the right guess, and hence the correct solutions. In other words educated guesses are not always correct.

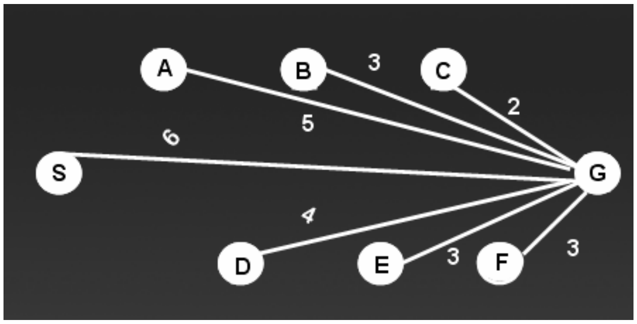
Recall the example of the mouse searching for cheese. The smell of cheese guides the mouse in the maze, in other words the strength of the smell informs the mouse that how far is it from the goal state. Here the smell of cheese is the heuristic and it is quite accurate.

Similarly, consider the diagram below. The graph shows a map in which the numbers on the edges are the distances between cities, for example, the distance between city S and city D is 3 and between B and E is 4.



Suppose our goal is to reach city G starting from S. There can be many choices, we might take S, A, D, E, F, G or travel from S, to A, to E, to F, and to G. At each city, if we were to decide which city to go next, we might be interested in some sort of information which will guide us to travel to the city from which the distance of goal is minimum.

If someone can tell us the straight-line distance of G from each city then it might help us as a heuristic in order to decide our route map. Consider the graph below.



It shows the straight line distances from every city to the goal. Now, cities that are closer to the goal should be our preference. These straight line distances also known as “as the crow flies distance” shall be our heuristic.

It is important to note that heuristics can sometimes misguide us. In the example we have just discussed, one might try to reach city C as it is closest from the goal according to our heuristic, but in the original map you can see that there is no direct link between city C and city G. Even if someone reaches city C using the heuristic, he won’t be able to travel to G from C directly, hence the heuristic can misguide. The catch here is that crow-flight distances do not tell us that the two cities are directly connected.

Similarly, in the example of mouse and cheese, consider that the maze has fences fixed along some of the paths through which the smell can pass. Our heuristic might guide us on a path which is blocked by a fence, hence again the heuristic is misguiding us.

The conclusion then is that heuristics do help us reduce the search space, but it is not at all guaranteed that we'll always find a solution. Still many people use them as most of the time they are helpful. The key lies in the fact that how do we use the heuristic. Consider the notion of a heuristic function.

Whenever we choose a heuristic, we come up with a heuristic function which takes as input the heuristic and gives us out a number corresponding to that heuristic. The search will now be guided by the output of the heuristic function. Depending on our application we might give priority to either larger numbers or smaller numbers.

Hence to every node/ state in our graph we will assign a heuristic value, calculated by the heuristic function. We will start with a basic heuristically informed search which is called Hill Climbing.

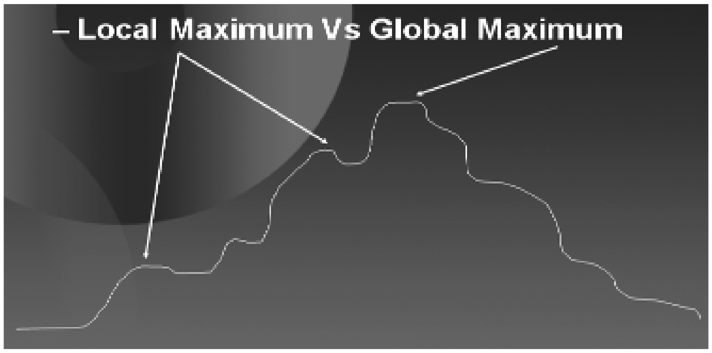
2.15 Hill Climbing

Hill Climbing is basically a depth first search with a measure of quality that is assigned to each node in the tree. The basic idea is: Proceed as you would in DFS except that you order your choices according to some heuristic measurement of the remaining distance to the goal. We will discuss the Hill climbing with an example.

Before going to the actual example, let us give another analogy for which the name Hill Climbing has been given to this procedure. Consider a blind person climbing a hill. He can not see the peak of the hill. The best he can do is that from a given point he takes steps in all possible directions and wherever he finds that a step takes him higher he takes that step and reaches a new, higher point. He goes on doing this until all possible steps in any direction will take him higher and this would be the peak, hence the name hill climbing. Notice that each step that we take, gets us closer to our goal which in this example is the peak of a hill.

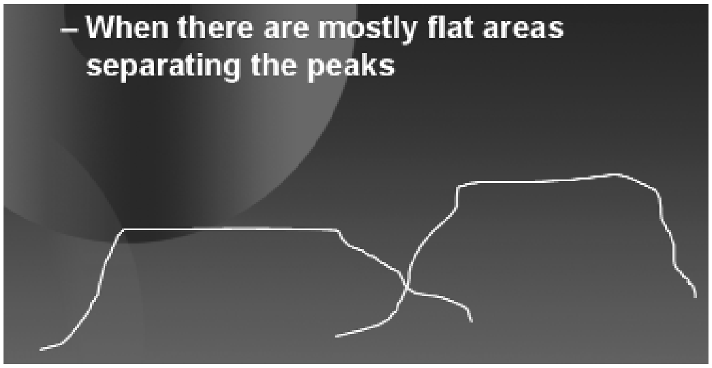
Such a procedure might as well have some problems.

Foothill Problem: Consider the diagram of a mountain below. Before reaching the global maxima, that is the highest peak, the blind man will encounter local maxima that are the intermediate peaks and before reaching the maximum height. At each of these local maxima, the blind man gets the perception of having reached the global maxima as none of the steps takes him to a higher point. Hence he might just reach local maxima and think that he has reached the global maxima. Thus getting stuck in the middle of searching the solution space.



– Local Maximum Vs Global Maximum

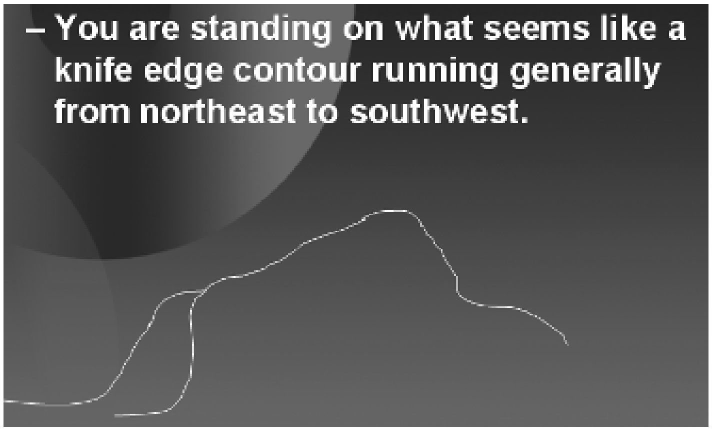
Plateau Problem: Similarly, consider another problem as depicted in the diagram below. Mountains where flat areas called plateaus are frequently encountered the blind person might again get stuck.



– When there are mostly flat areas separating the peaks

When he reaches the portion of a mountain which is totally flat, whatever step he takes gives him no improvement in height hence he gets stuck.

Ridge Problem: Consider another problem; you are standing on what seems like a knife edge contour running generally from northeast to southwest. If you take step in one direction it takes you lower, on the other hand when you step in some other direction it gives you no improvement.



– You are standing on what seems like a knife edge contour running generally from northeast to southwest.

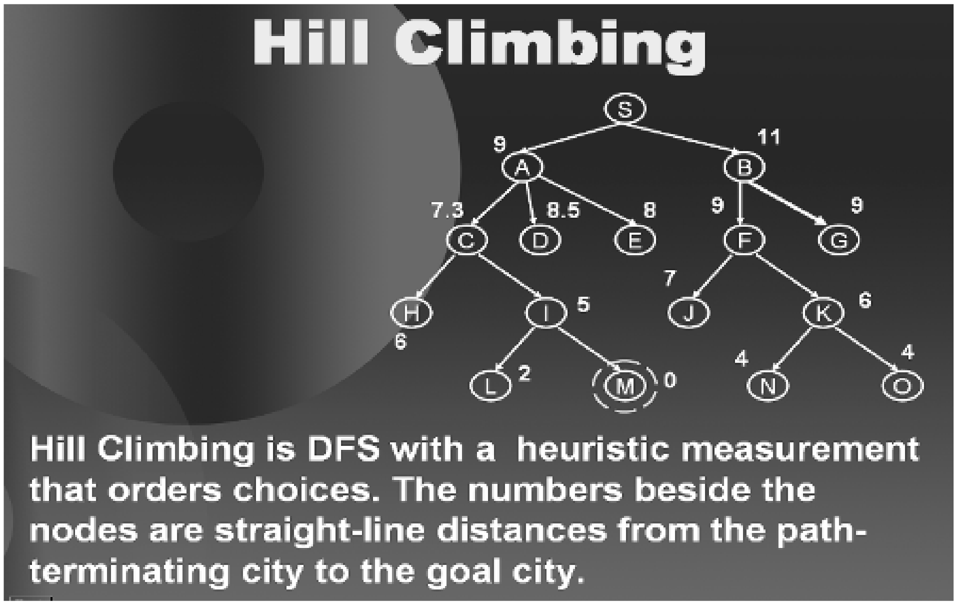
All these problems can be mapped to situations in our solution space searching. If we are at a state and the heuristics of all the available options take us to a lower value, we might be at local maxima. Similarly, if all the available heuristics take us to no improvement we might be at a plateau. Same is the case with ridge as we can encounter such states in our search tree.

The solution to all these problems is randomness. Try taking random steps in random direction of random length and you might get out of the place where you are stuck.

Example

Let us now take you through an example of searching a tree using hill climbing to end out discussion on hill climbing.

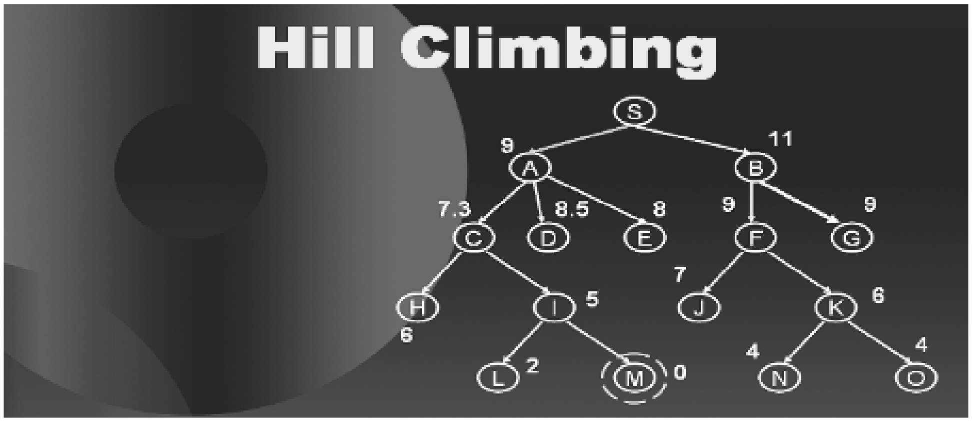
Consider the diagram below. The tree corresponds to our problem of reaching city M starting from city S. In other words our aim is to find a path from S to M. We now associate heuristics with every node, that is the straight line distance from the path-terminating city to the goal city.



Hill Climbing

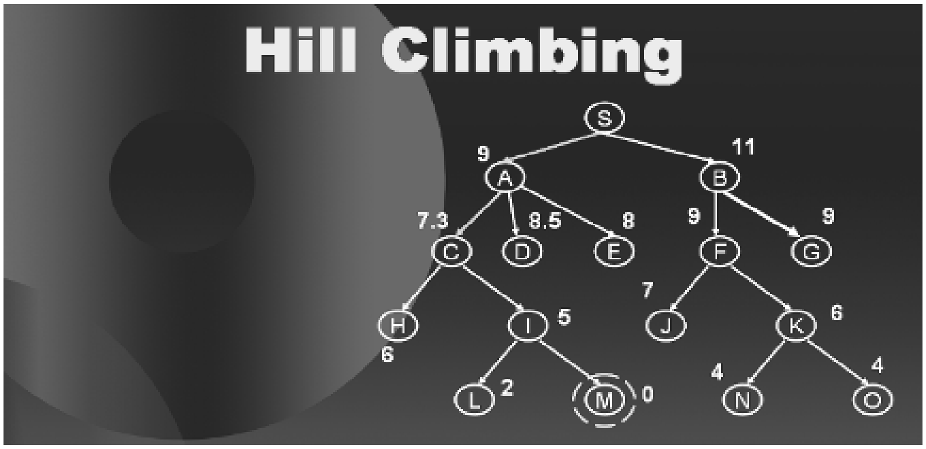
Hill Climbing is DFS with a heuristic measurement that orders choices. The numbers beside the nodes are straight-line distances from the path-terminating city to the goal city.

When we start at S we see that if we move to A we will be left with 9 units to travel. As moving on A has given us an improvement in reaching our goal hence we move to A. Exactly in the same manner as the blind man moves up a step that gives him more height.



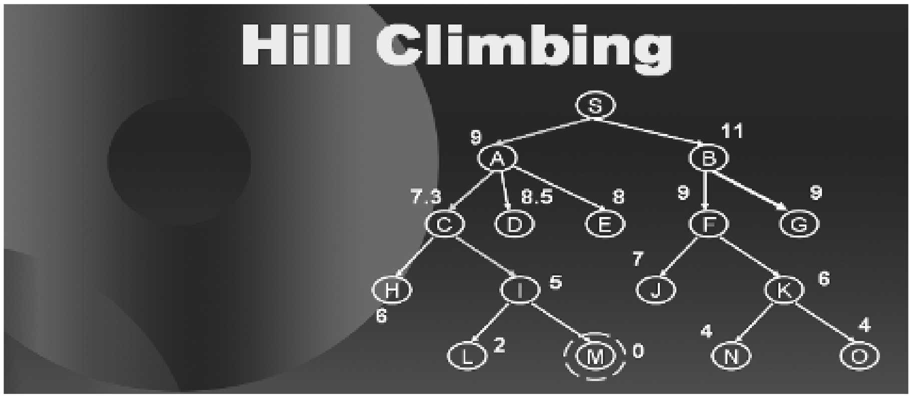
Hill Climbing

Standing on A we see that C takes us closer to the goal hence we move to C.

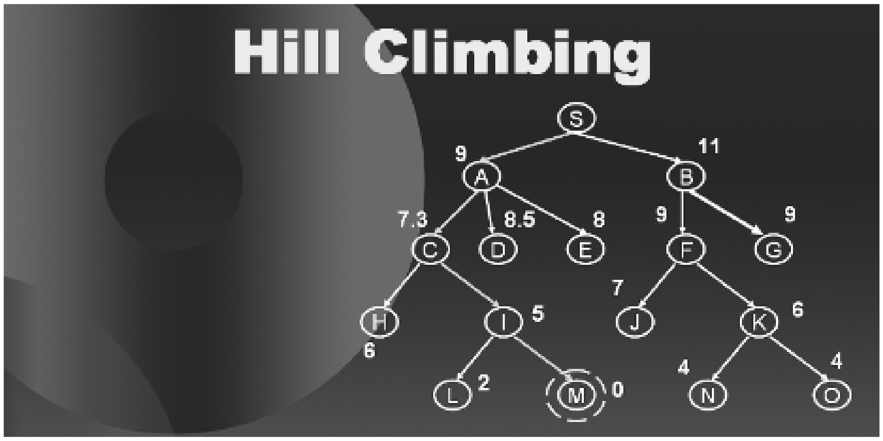


Hill Climbing

From C we see that city I give us more improvement hence we move to I and then finally to M.



Hill Climbing

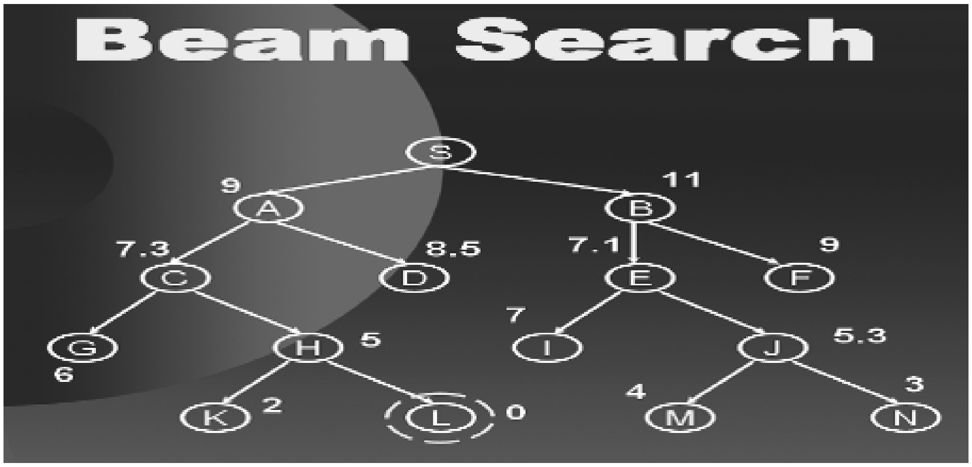


Notice that we only traveled a small portion of the search space and reached our goal. Hence the informed nature of the search can help reduce space and time.

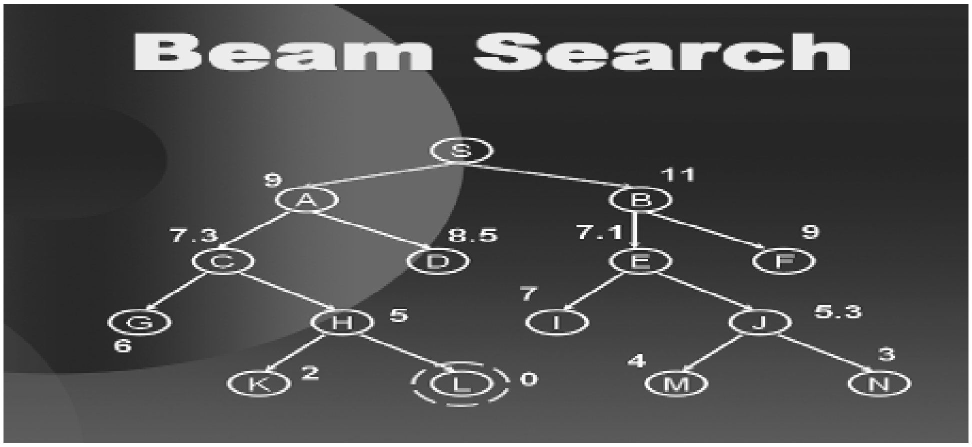
2.16Beam Search

You just saw how hill climbing procedure works through the search space of a tree. Another procedure called beam search proceeds in a similar manner. Out of n possible choices at any level, beam search follows only the best k of them; k is the parameter which we set and the procedure considers only those many nodes at each level.

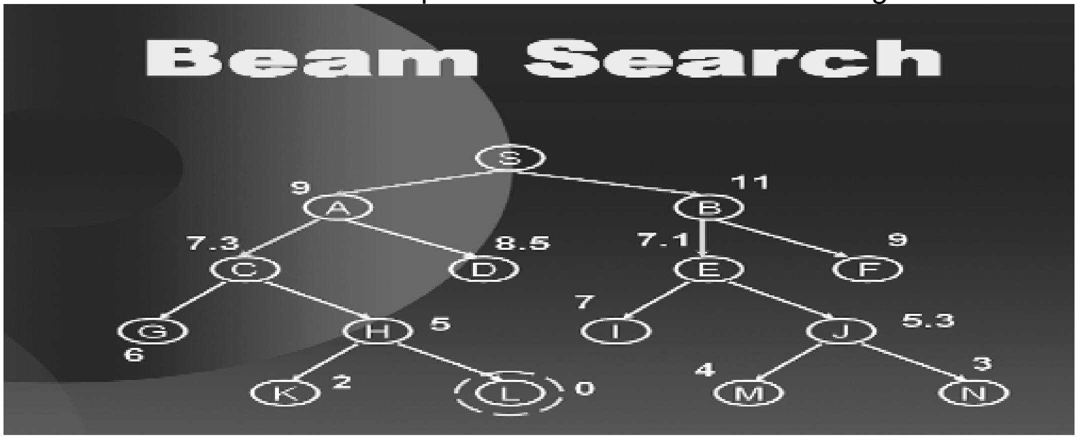
The following sequence of diagrams will show you how Beam Search works in a search tree.



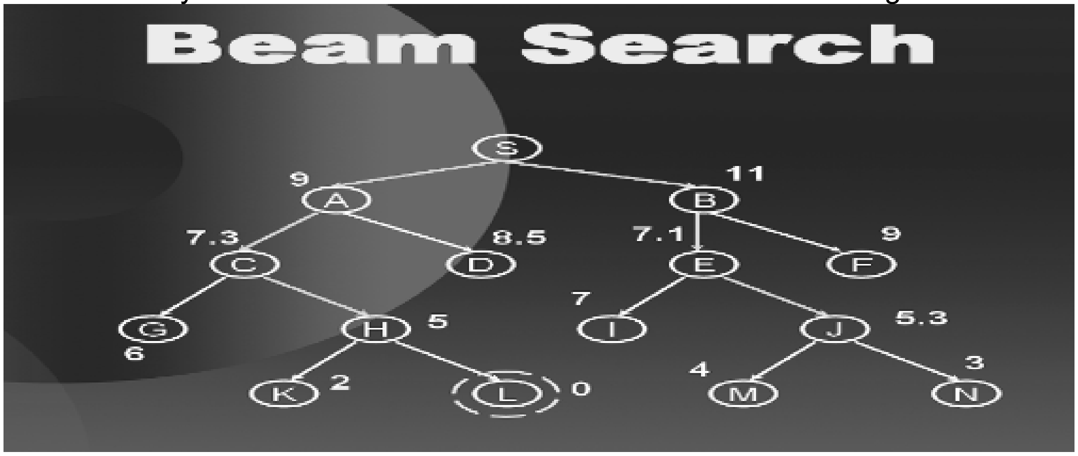
We start with a search tree with L as goal state and k=2, that is at every level we will only consider the best 2 nodes. When standing on S we observe that the only two nodes available are A and B so we explore both of them as shown below.

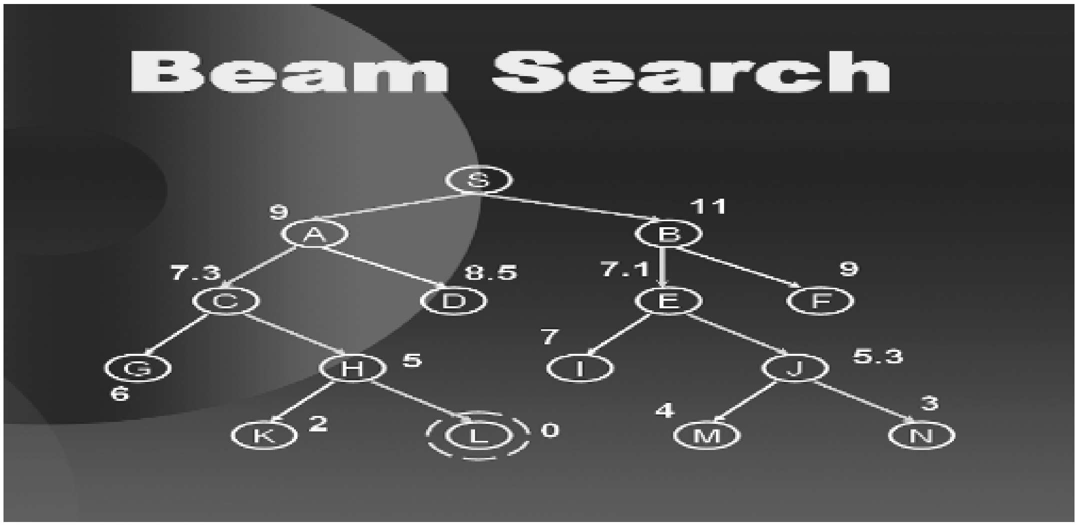


From here we have C, D, E and F as the available options to go. Again, we select the two best of them and we explore C and E as shown in the diagram below.



From C and E we have G, H, I and J as the available options so we select H and J and similarly at the last level we select L and N of which L is the goal.

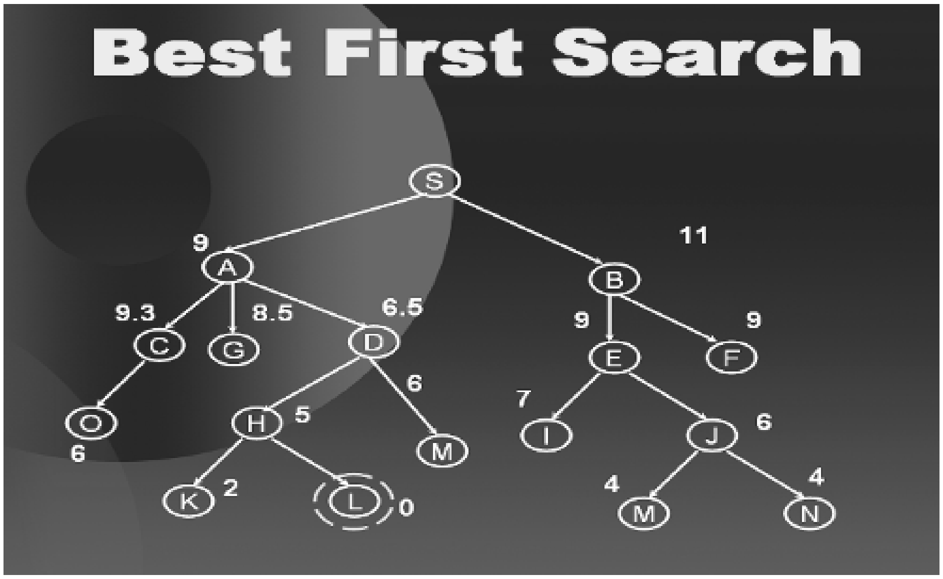




Beam Search

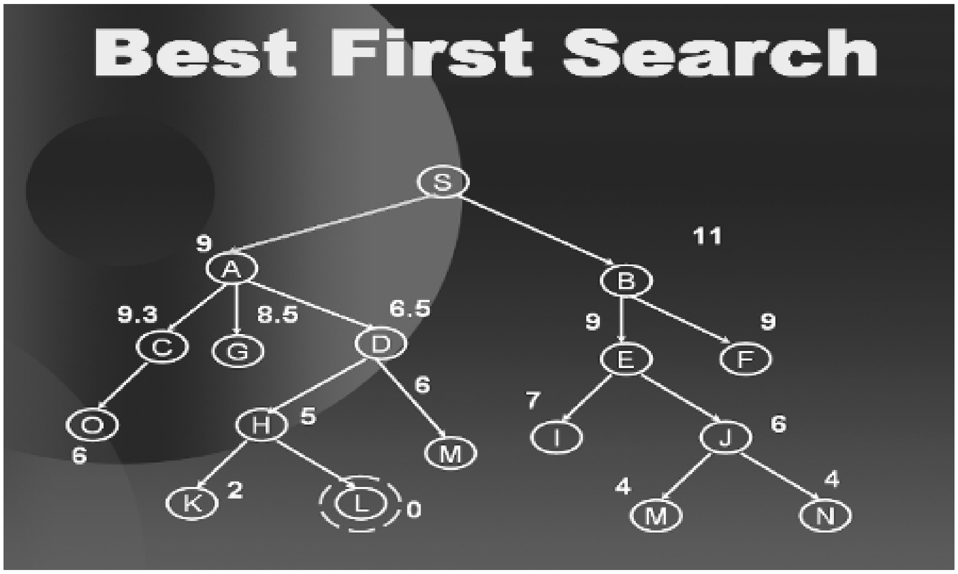
2.17 Best First Search

Just as beam search considers best k nodes at every level, best first search considers all the open nodes so far and selects the best amongst them. The following sequence of diagrams will show you how a best first search procedure works in a search tree.



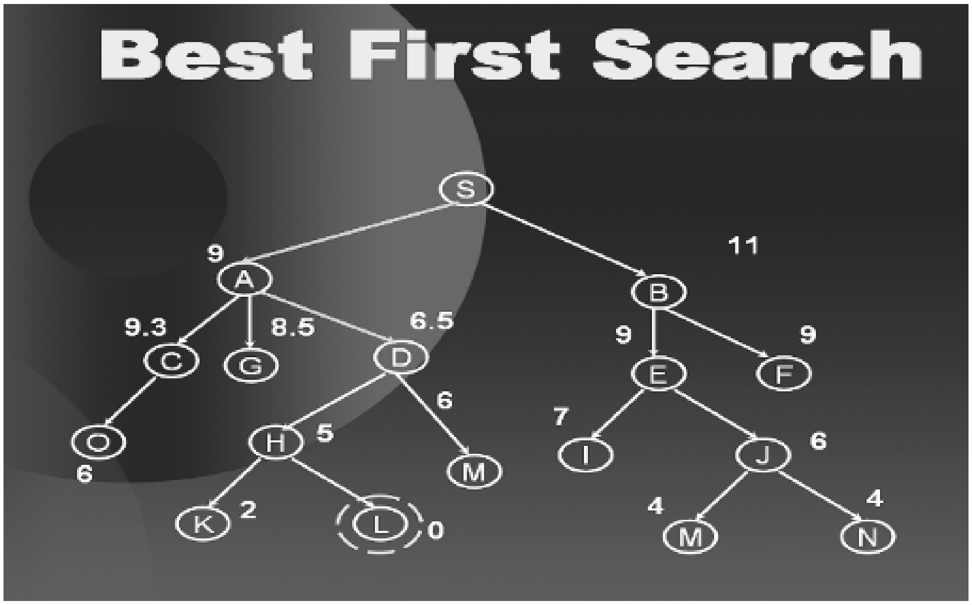
Best First Search

We start with a search tree as shown above. From S we observe that A is the best option so we explore A.

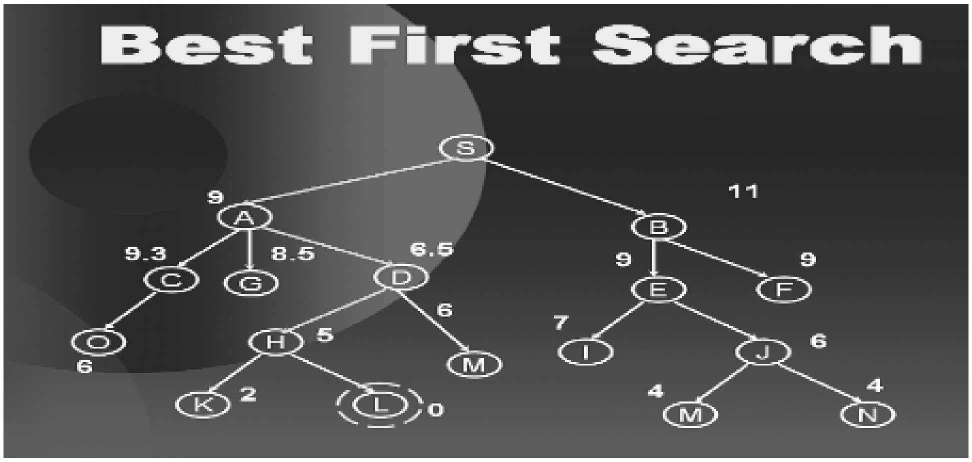


Best First Search

At A we now have C, G, D and B as the options. We select the best of them which is D.

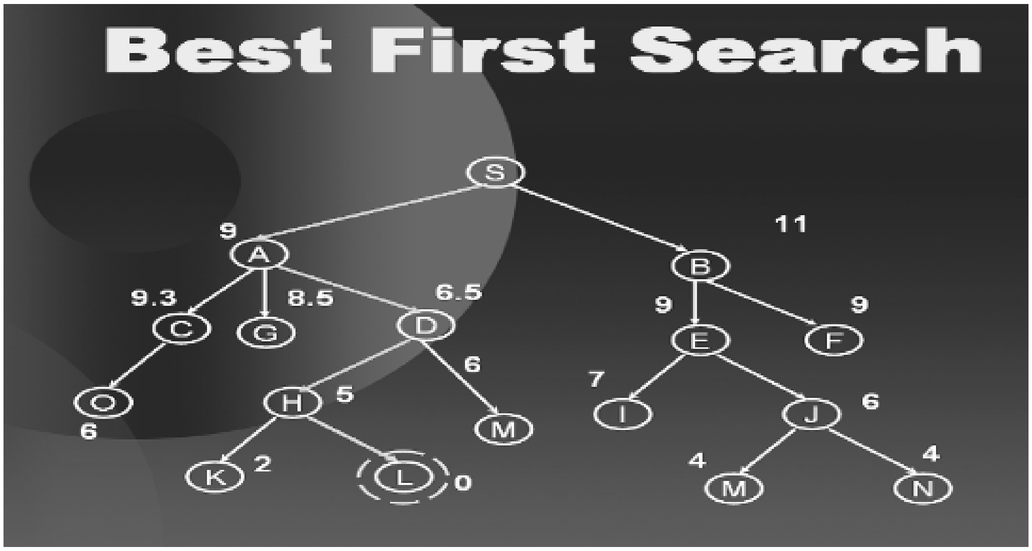


Best First Search



Best First Search

At D we have S, G, B, H, M and J as the options. We select H which is the best of them.



Best First Search

At last from H we find L as the best. Hence best first search is a greedy approach will looks for the best amongst the available options and hence can sometimes reduce the searching time. All these heuristically informed procedures are considered better but they do not guarantee the optimal solution, as they are dependent on the quality of heuristic being used.

2.18 Optimal Searches

So far we have looked at uninformed and informed searches. Both have their advantages and disadvantages. But one thing that lacks in both is that whenever they find a solution they immediately stop. They never consider that their might be more than one solution to the problem and the solution that they have ignored might be the optimal one.

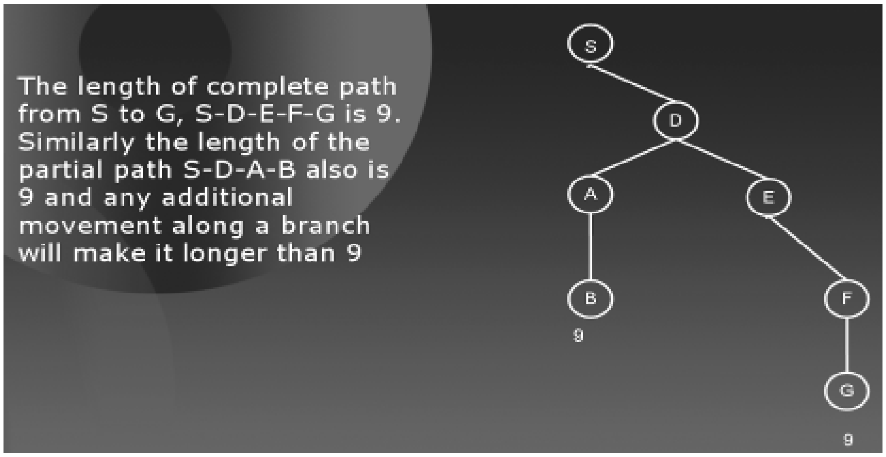
A simplest approach to find the optimal solution is this; find all the possible solutions using either an uninformed search or informed search and once you have searched the whole search space and no other solution exists, then choose the most optimal amongst the solutions found. This approach is analogous to the brute force method and is also called the British museum procedure.

But in reality, exploring the entire search space is never feasible and at times is not even possible, for instance, if we just consider the tree corresponding to a game of chess (we will learn about game trees later), the effective branching factor is 16 and the effective depth is 100. The number of branches in an exhaustive survey would be on the order of 10^{120}. Hence a huge amount of computation power and time is required in solving the optimal search problems in a brute force manner.

2.19 Branch and Bound

In order to solve our problem of optimal search without using a brute force technique, people have come up with different procedures. One such procedure is called branch-and-bound method.

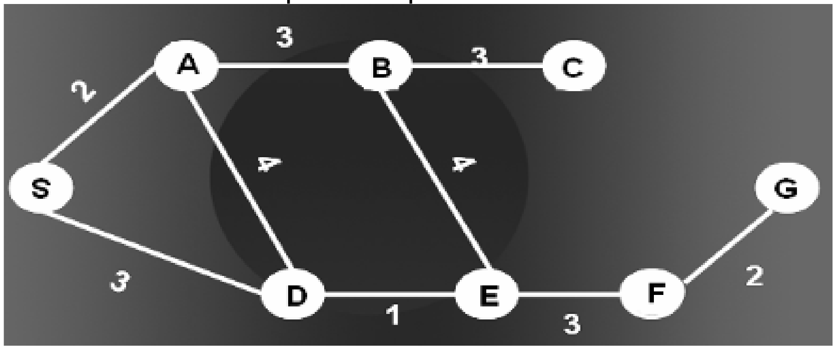
The simple idea of branch and bound is the following:



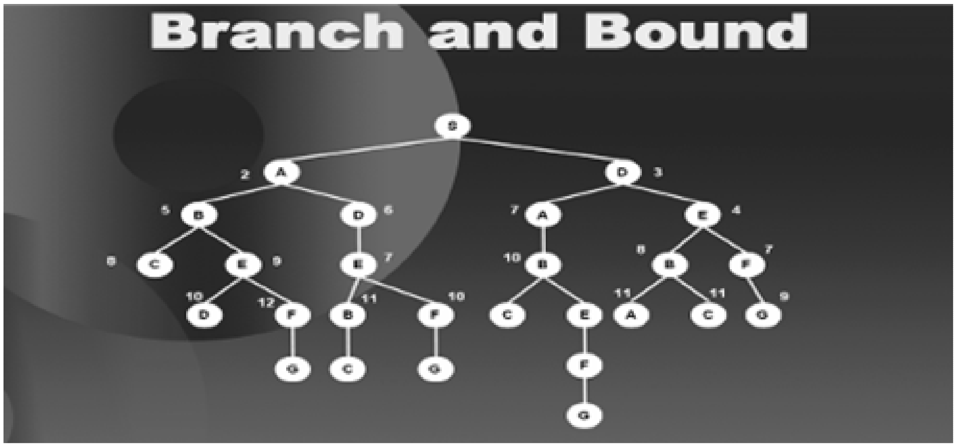
The length of complete path from S to G, S-D-E-F-G is 9. Similarly the length of the partial path S-D-A-B also is 9 and any additional movement along a branch will make it longer than 9

The length of the complete path from S to G is 9. Also note that while traveling from S to B we have already covered a distance of 9 units. So traveling further from S D A B to some other node will make the path longer. So we ignore any further paths ahead of the path S D A B.

We will show this with a simple example.

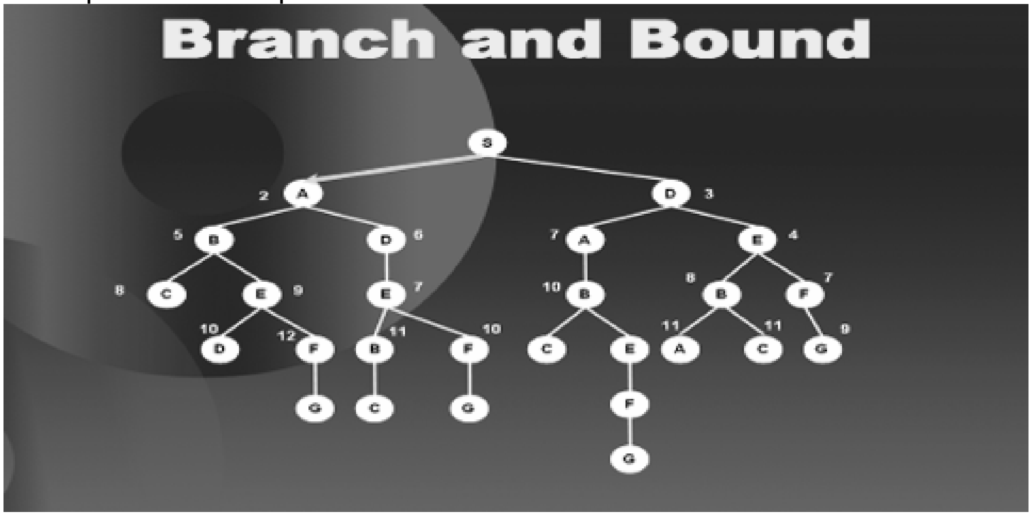


The diagram above shows the same city road map with distance between the cities labels on the edges. We convert the map to a tree as shown below.



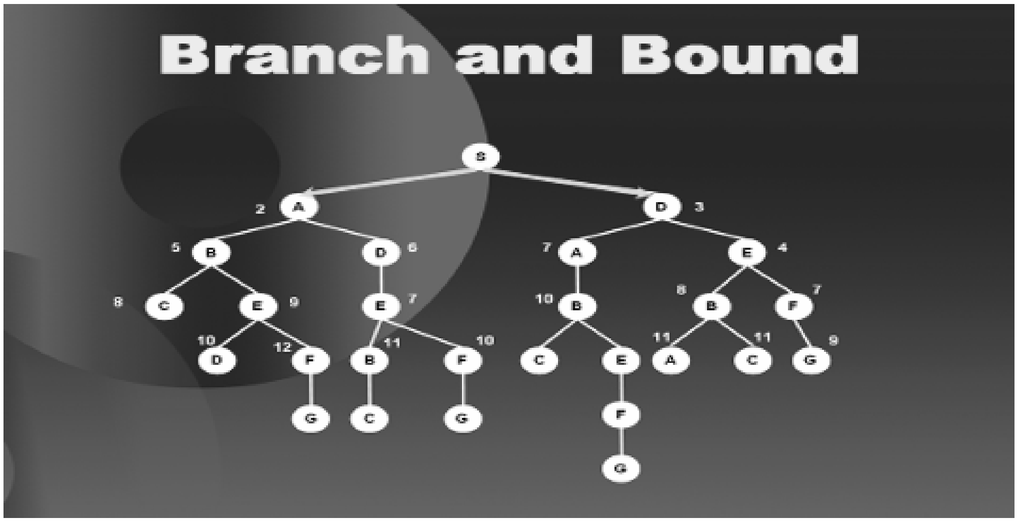
Branch and Bound

We proceed in a Best First Search manner. Starting at S we see that A is the best option so we explore A.



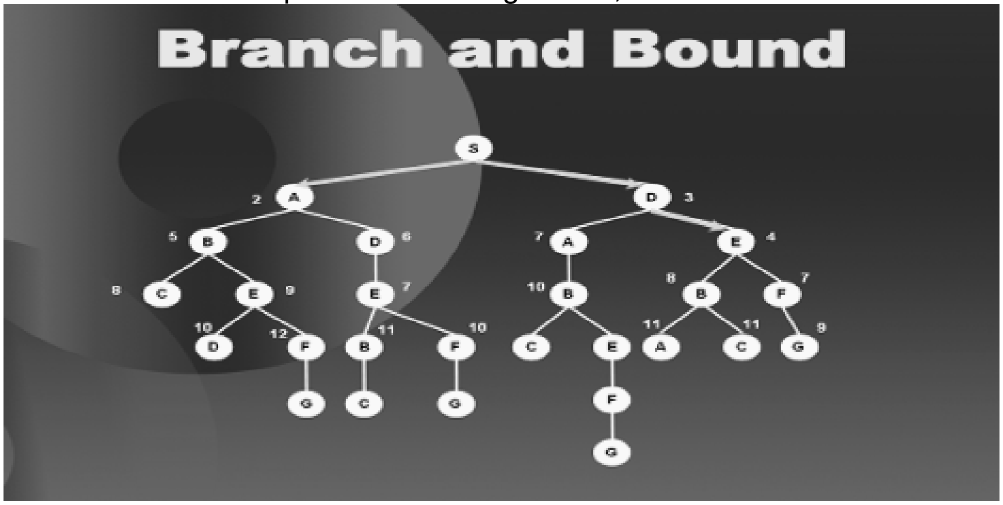
Branch and Bound

From S the options to travel are B and D, the children of A and D the child of S. Among these, D the child of S is the best option. So we explore D.



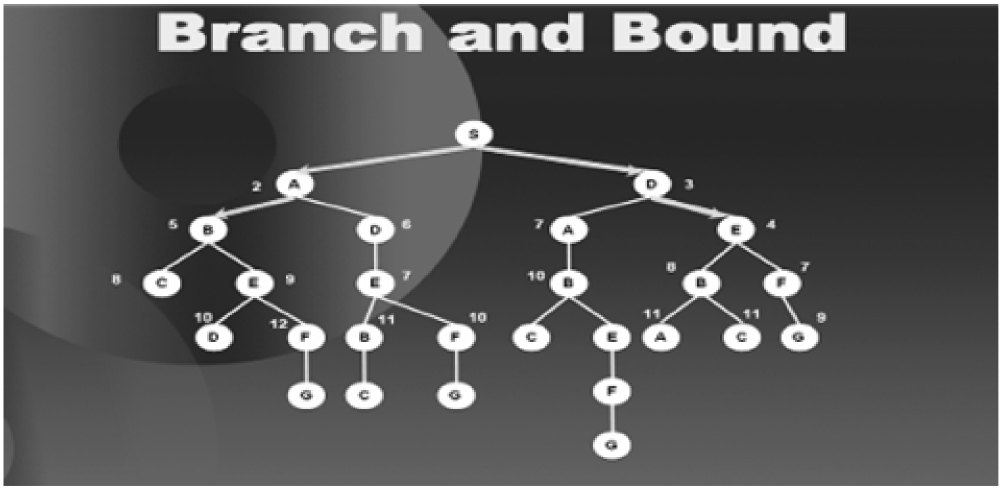
Branch and Bound

From here the best option is E so we go there,



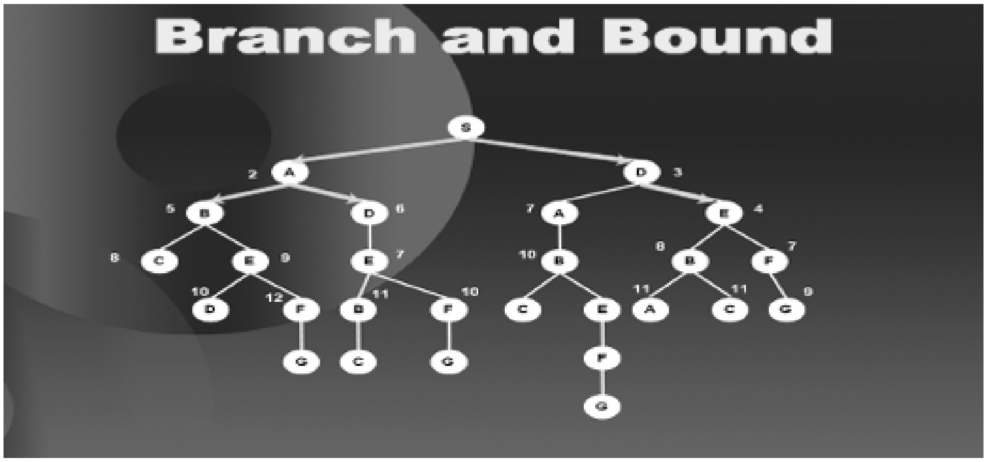
Branch and Bound

then B,



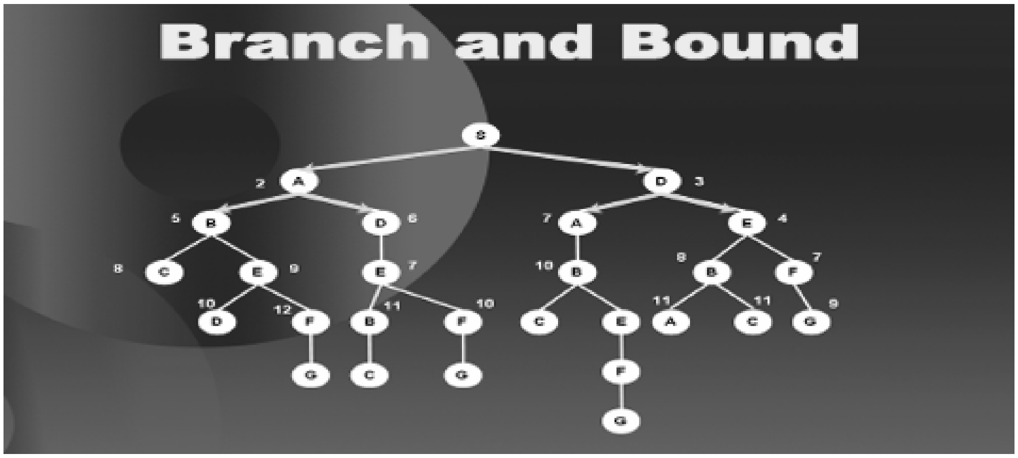
Branch and Bound

then D,



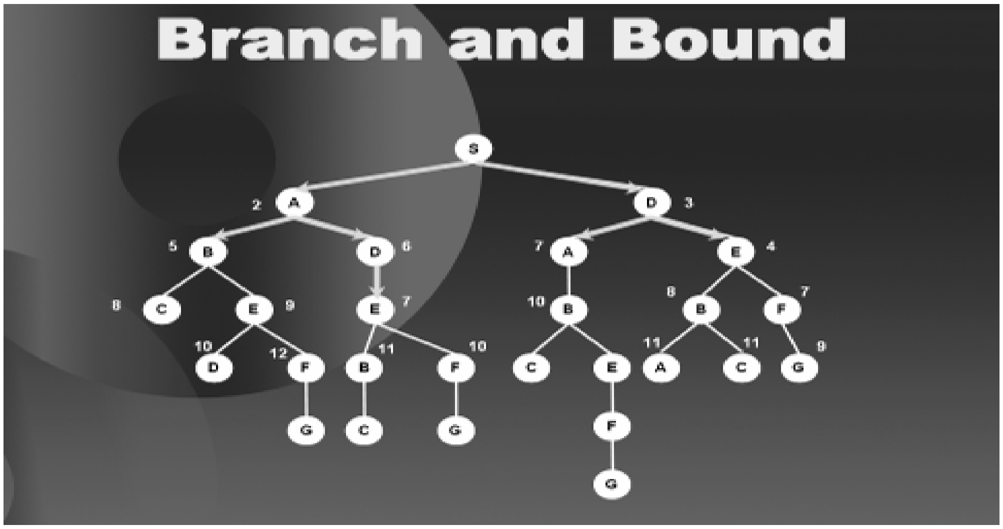
Branch and Bound

Here we have E, F and A as equally good options so we select arbitrarily and move to say A,



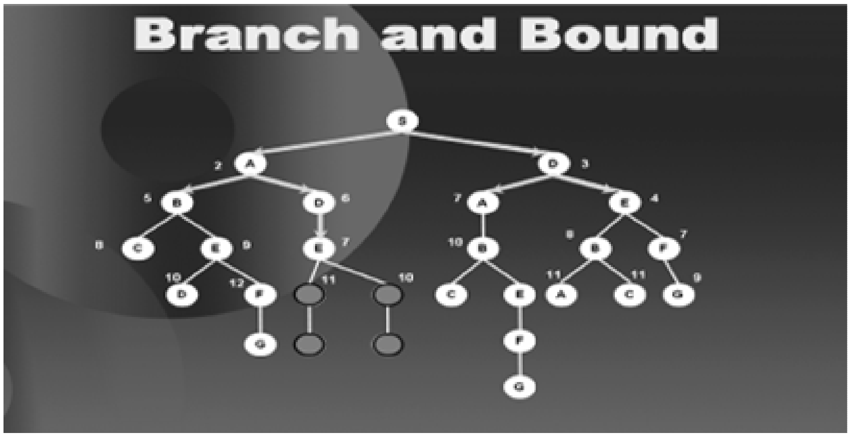
Branch and Bound

then E.



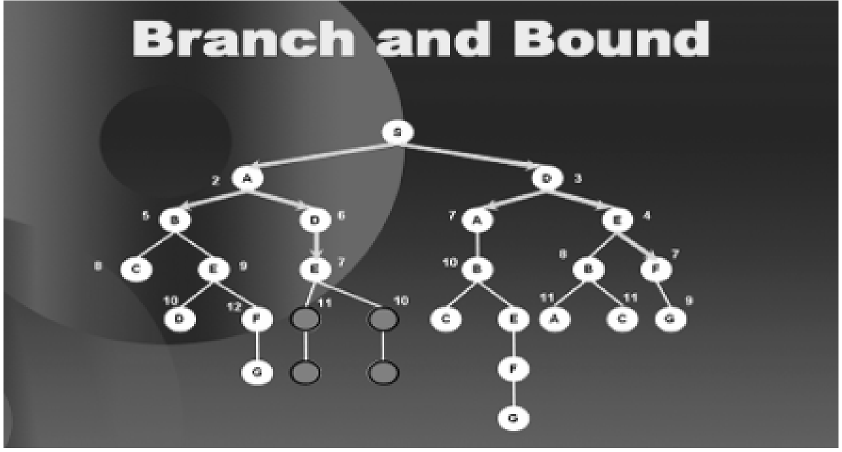
Branch and Bound

When we explore E we find out that if we follow this path further, our path length will increase beyond 9 which is the distance of S to G. Hence we block all the further sub-trees along this path, as shown in the diagram below.



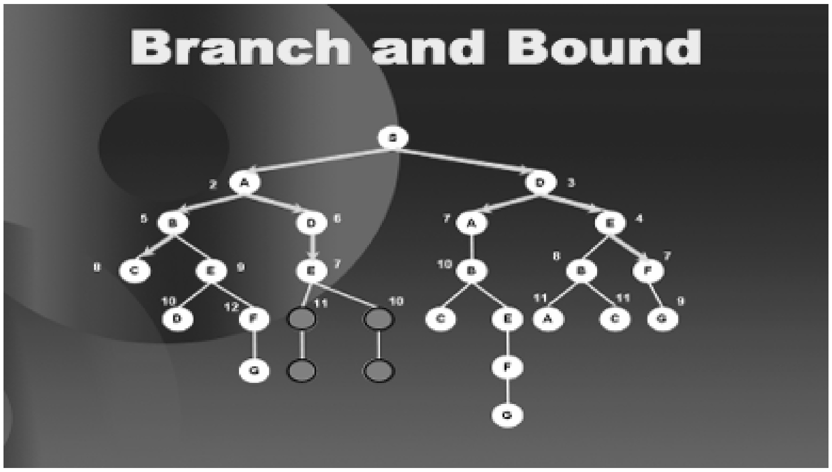
Branch and Bound

We then move to F as that is the best option at this point with a value 7.



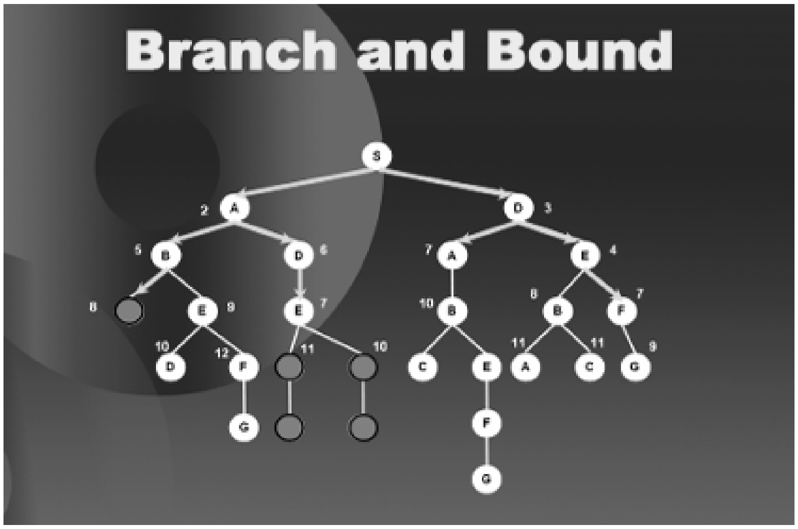
Branch and Bound

then C,



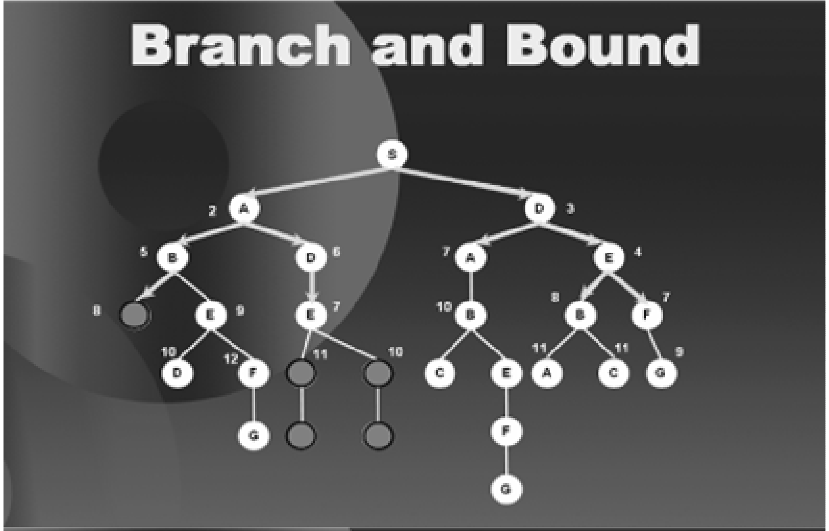
Branch and Bound

We see that C is a leaf node so we bind C too as shown in the next diagram.

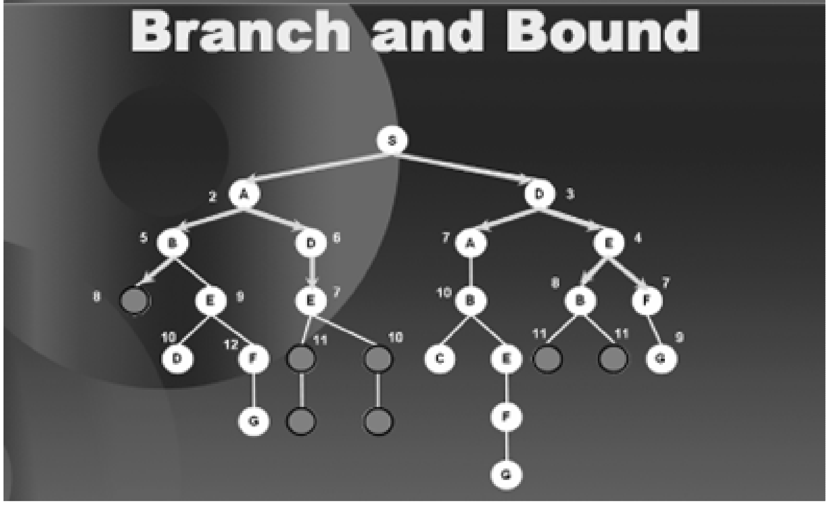


Branch and Bound

Then we move to B on the right hand side of the tree and bind the sub trees ahead of B as they also exceed the path length 9.

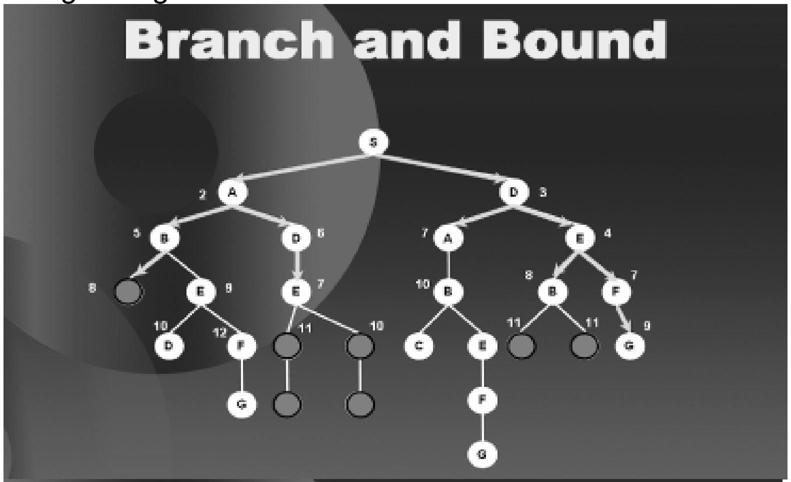


Branch and Bound

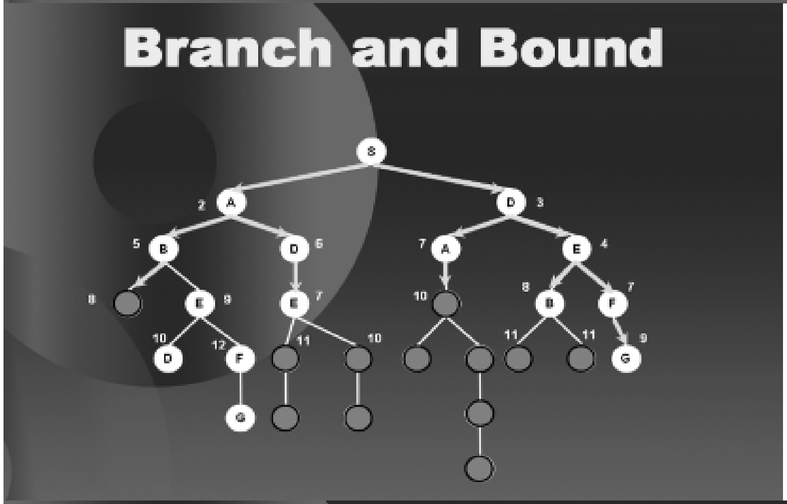


Branch and Bound

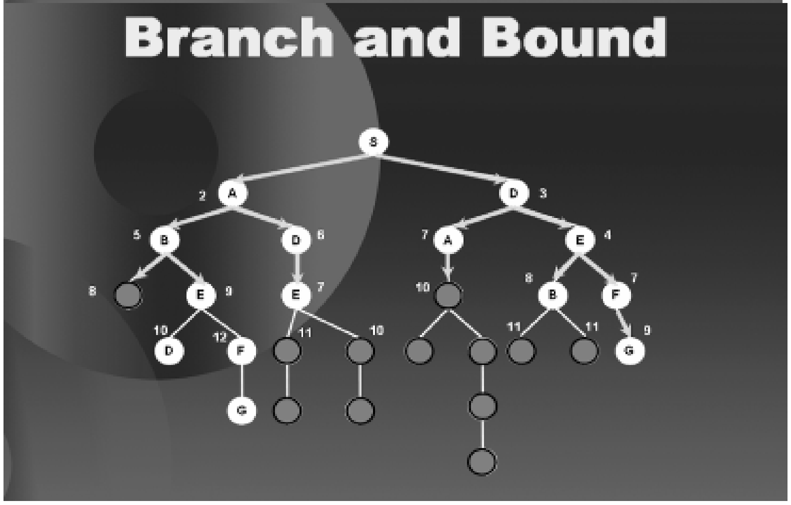
We go on proceeding in this fashion, binding the paths that exceed 9 and hence we are saved from traversing a considerable portion of the tree. The subsequent diagrams complete the search until it has found all the optimal solution, that is along the right hand branch of the tree.



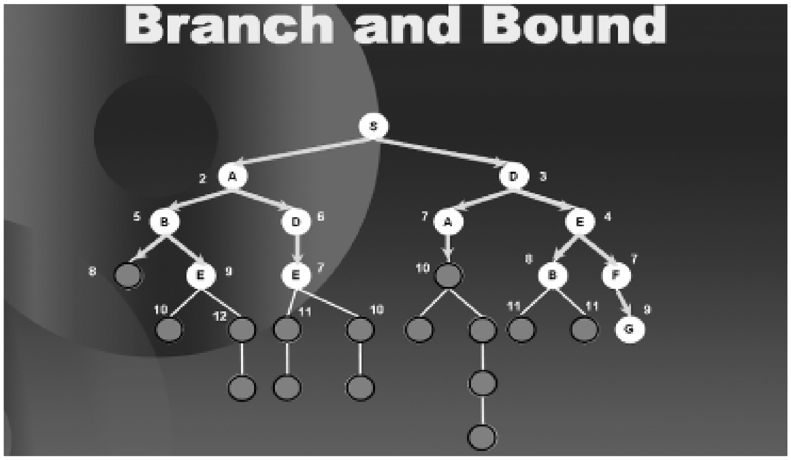
Branch and Bound



Branch and Bound



Branch and Bound



Branch and Bound

Notice that we have saved ourselves from traversing a considerable portion of the tree and still have found the optimal solution. The basic idea was to reduce the search space by binding the paths that exceed the path length from S to G.

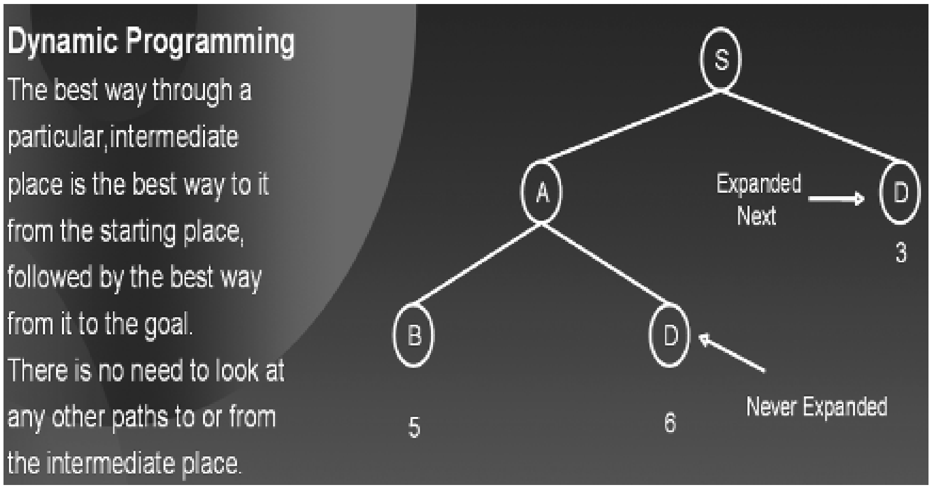
2.20 Improvements in Branch and Bound

The above procedure can be improved in many different ways. We will discuss the two most famous ways to improve it.

1. Estimates
2. Dynamic Programming

The idea of estimates is that we can travel in the solution space using a heuristic estimate. By using “guesses” about remaining distance as well as facts about distance already accumulated we will be able to travel in the solution space more efficiently. Hence we use the estimates of the remaining distance. A problem here is that if we go with an overestimate of the remaining distance then we might loose a solution that is somewhere nearby. Hence we always travel with underestimates of the remaining distance. We will demonstrate this improvement with an example.

The second improvement is dynamic programming. The simple idea behind dynamic programming is that if we can reach a specific node through more than one different path then we shall take the path with the minimum cost.



Dynamic Programming

The best way through a particular, intermediate place is the best way to it from the starting place, followed by the best way from it to the goal. There is no need to look at any other paths to or from the intermediate place.

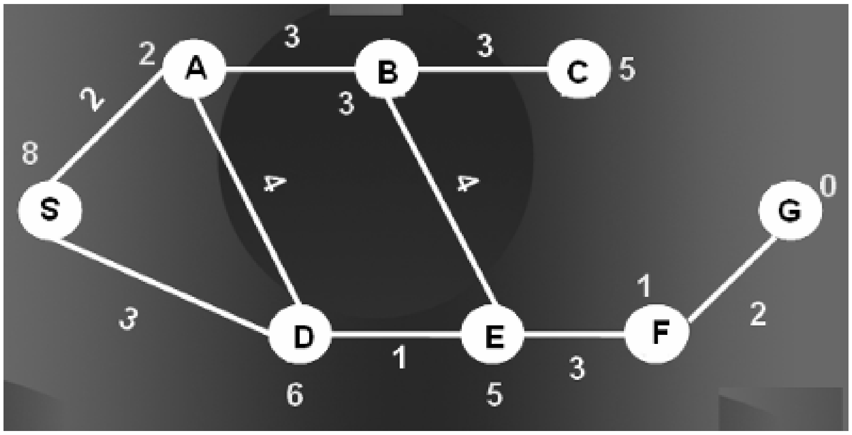
In the diagram you can see that we can reach node D directly from S with a cost of 3 and via S A D with a cost of 6 hence we will never expand the path with the larger cost of reaching the same node.

When we include these two improvements in branch and bound then we name it as a different technique known as A* Procedure.

2.21 A* Procedure

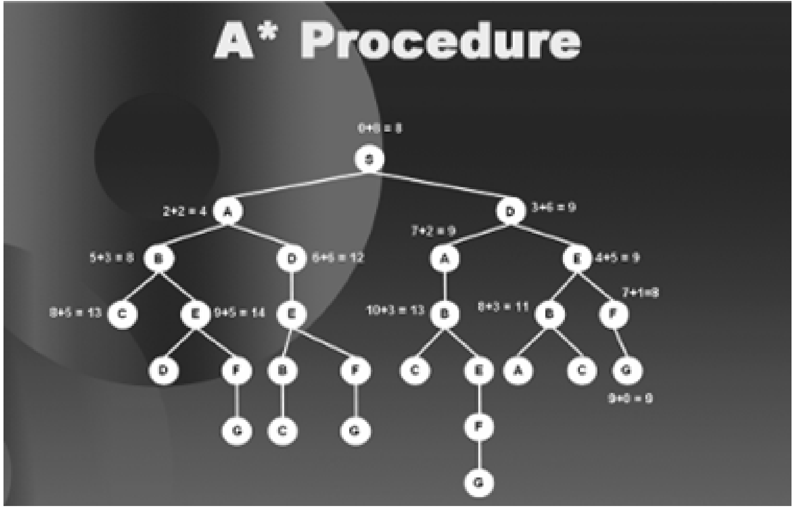
This is actually branch and bound technique with the improvement of underestimates and dynamic programming.

We will discuss the technique with the same example as that in branch-and-bound.



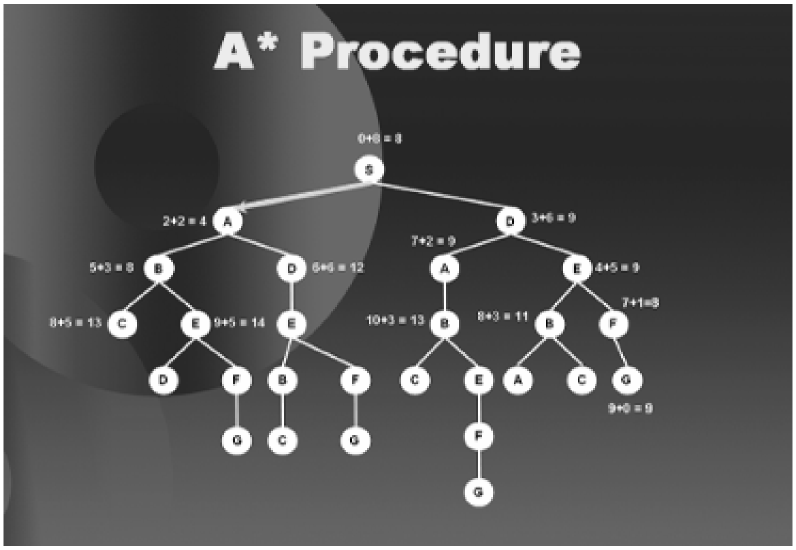
The values on the nodes shown in yellow are the underestimates of the distance of a specific node from G. The values on the edges are the distance between two adjacent cities.

Our measure of goodness and badness of a node will now be decided by a combination of values that is the distance traveled so far and the estimate of the remaining distance. We construct the tree corresponding to the graph above.
We start with a tree with goodness of every node mentioned on it.



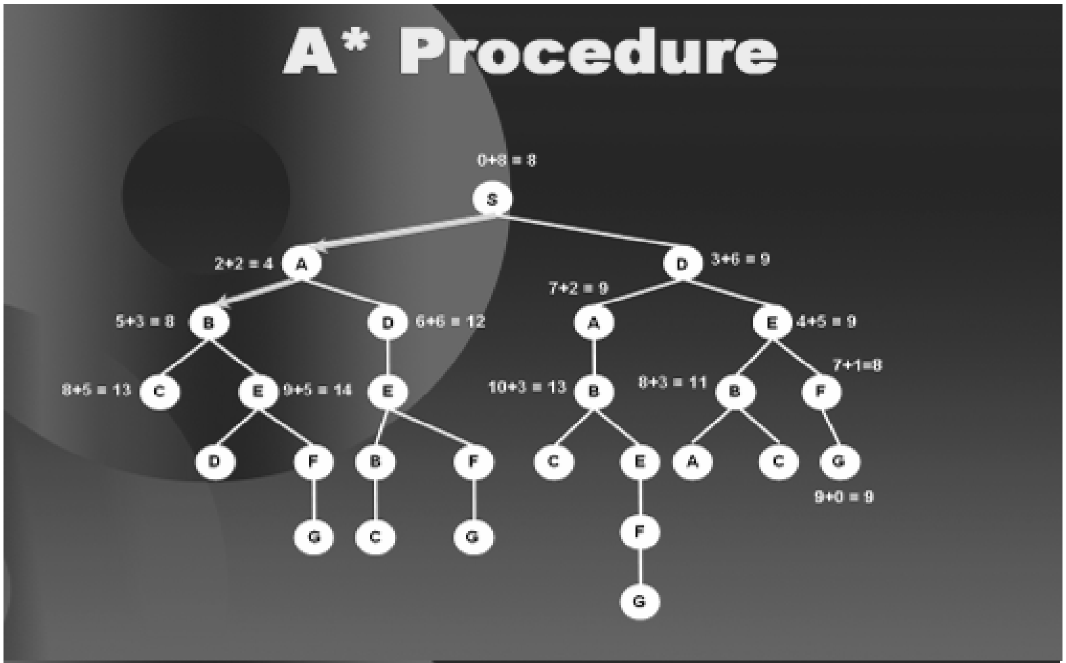
A* Procedure

Standing at S we observe that the best node is A with a value of 4 so we move to 4.

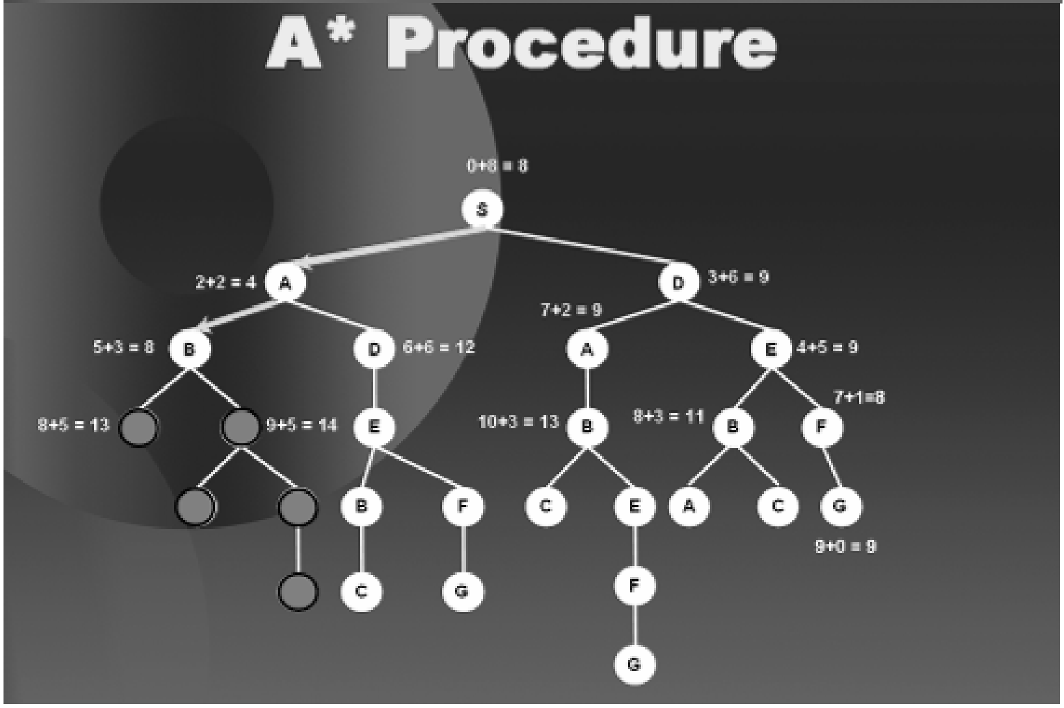


A* Procedure

Then B. As all the sub-trees emerging from B make our path length more than 9 units so we bound this path, as shown in the next diagram.

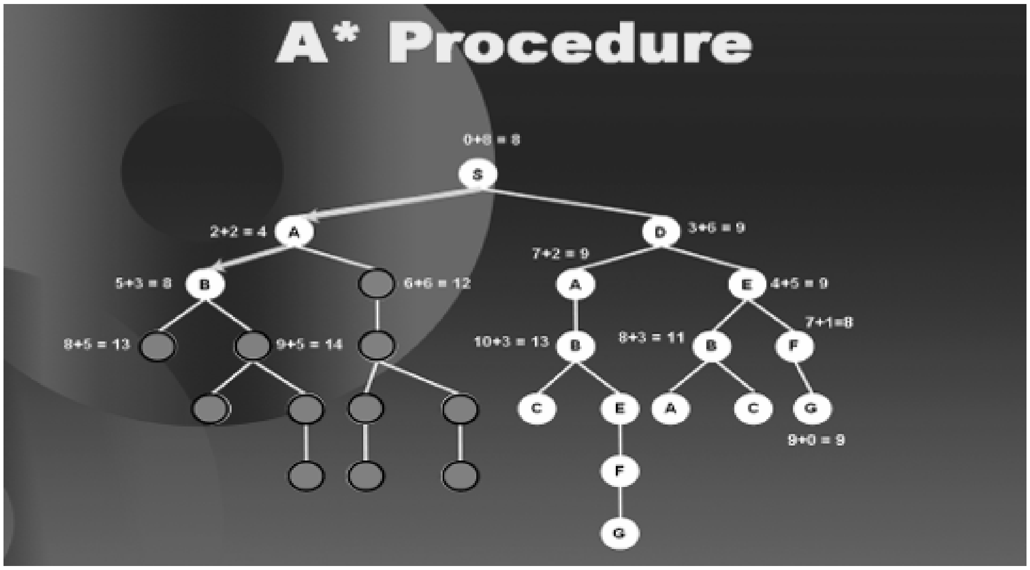


A* Procedure



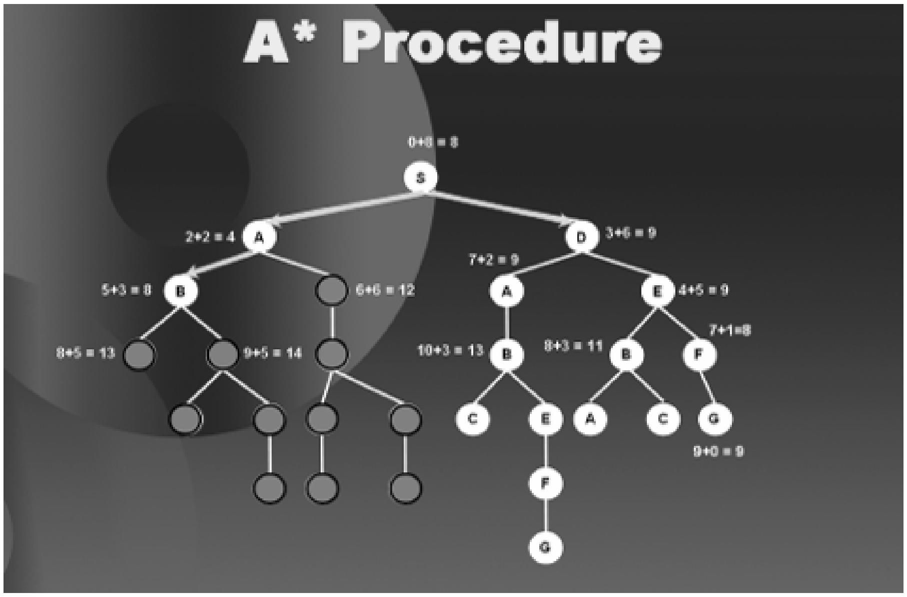
A* Procedure

Now observe that to reach node D that is the child of A we can reach it either with a cost of 12 or we can directly reach D from S with a cost of 9. Hence using dynamic programming we will ignore the whole sub-tree beneath D (the child of A) as shown in the next diagram.



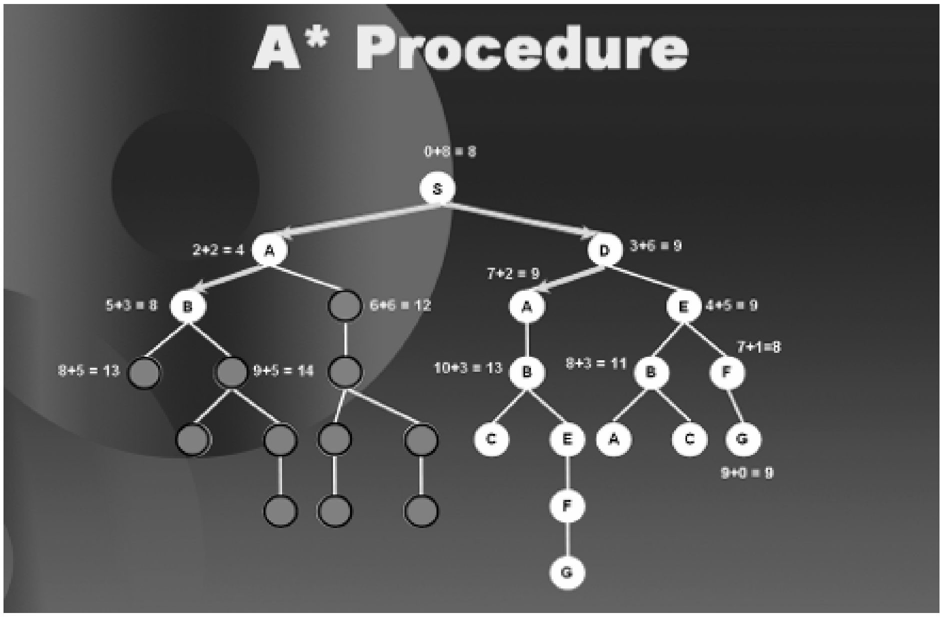
A* Procedure

Now we move to D from S.



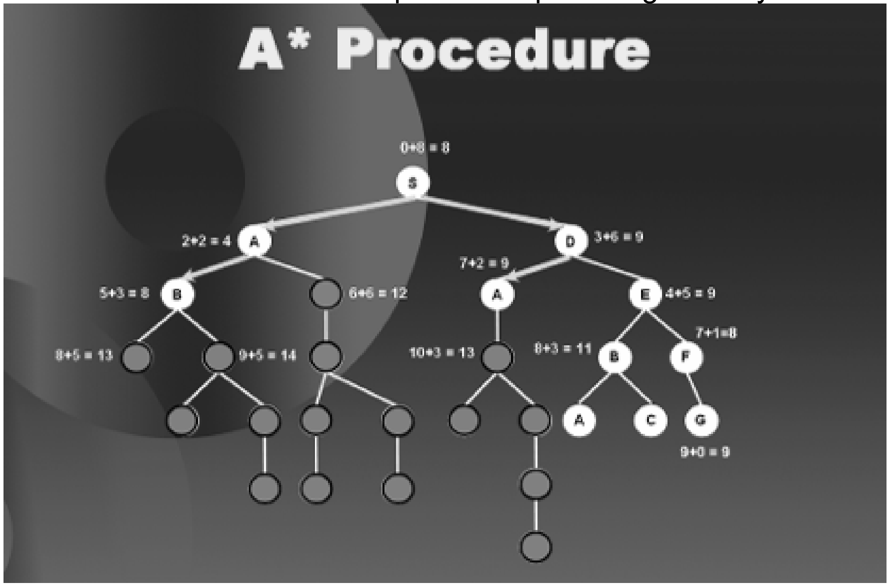
A* Procedure

Now A and E are equally good nodes so we arbitrarily choose amongst them, and we move to A.



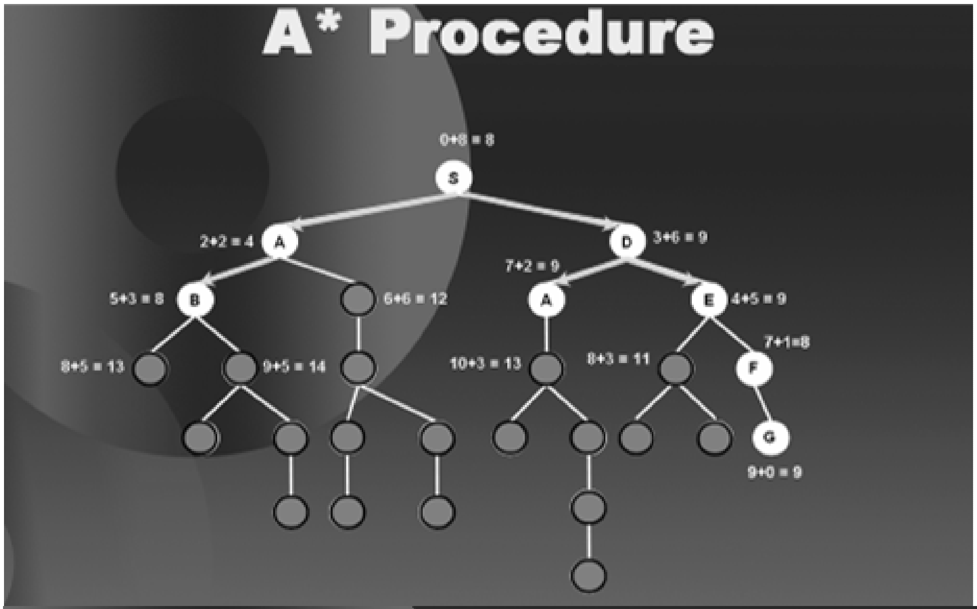
A* Procedure

As the sub-tree beneath A expands the path length is beyond 9 so we bind it.

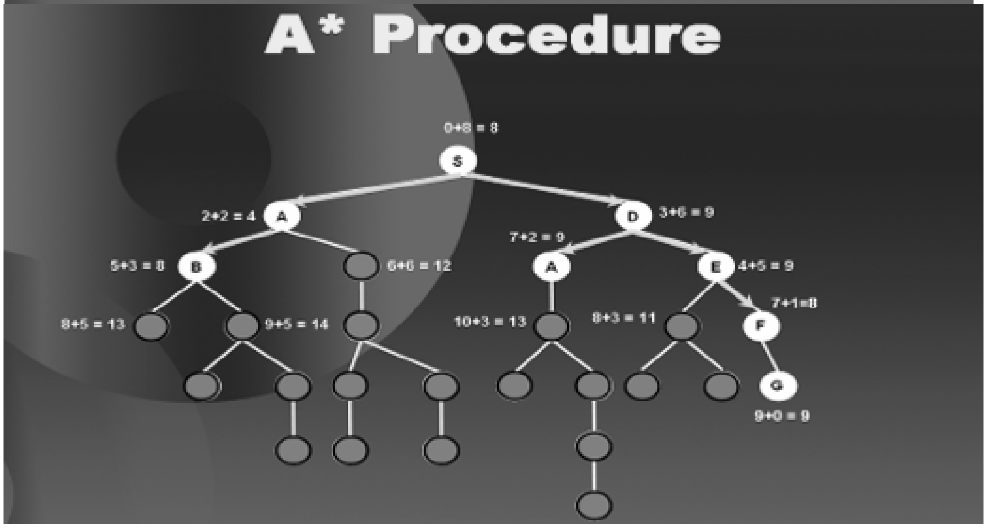


A* Procedure

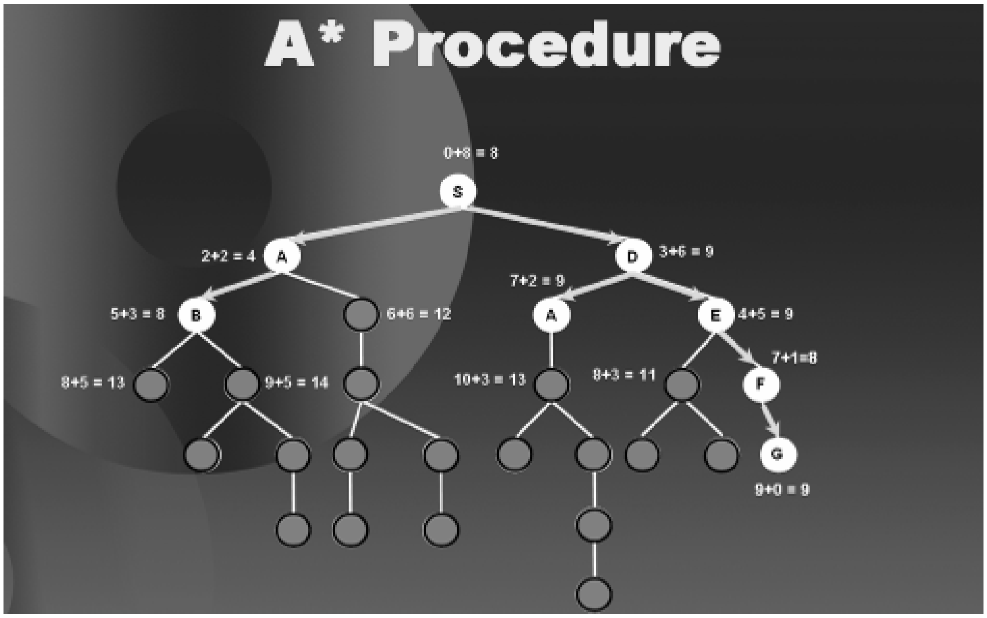
We proceed in this manner. Next we visit E, then we visit B the child of E, we bound the sub-tree below B. We visit F and finally we reach G as shown in the subsequent diagrams.



A* Procedure



A* Procedure



A* Procedure

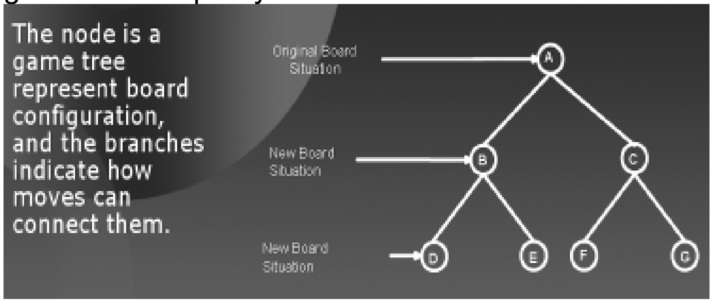
Notice that by using underestimates and dynamic programming the search space was further reduced and our optimal solution was found efficiently.

2.22 Adversarial Search

Up until now all the searches that we have studied there was only one person or agent searching the solution space to find the goal or the solution. In many applications there might be multiple agents or persons searching for solutions in the same solution space.

Such scenarios usually occur in game playing where two opponents also called adversaries are searching for a goal. Their goals are usually contrary to each other. For example, in a game of tic-tac-toe player one might want that he should complete a line with crosses while at the same time player two wants to complete a line of zeros. Hence both have different goals. Notice further that if player one puts a cross in any box, player-two will intelligently try to make a move that would leave player-one with minimum chance to win, that is, he will try to stop player-one from completing a line of crosses and at the same time will try to complete his line of zeros.

Many games can be modeled as trees as shown below. We will focus on board games for simplicity.



The node is a game tree represent board configuration, and the branches indicate how moves can connect them.

Searches in which two or more players with contrary goals are trying to explore the same solution space in search of the solution are called adversarial searches.

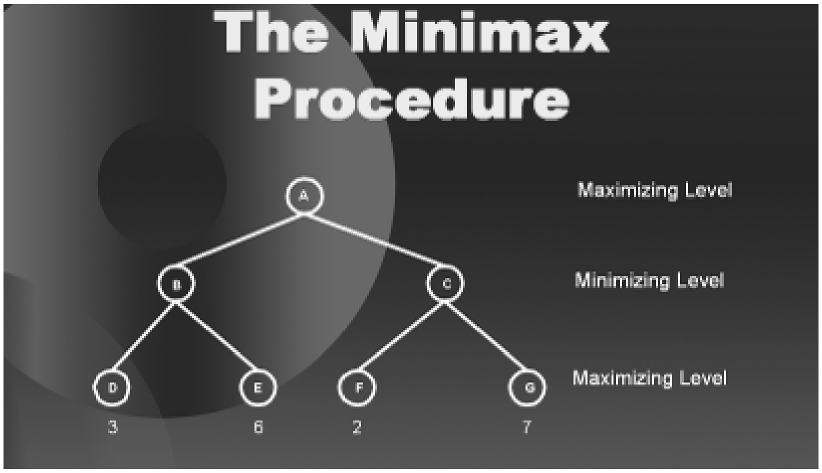
2.23Minimax Procedure

In adversarial searches one player tries to cater for the opponent's moves by intelligently deciding that what will be the impact of his own move on the over all configuration of the game. To develop this stance he uses a look ahead thinking strategy. That is, before making a move he looks a few levels down the game tree to see that what can be the impact of his move and what options will be open to the opponent once he has made this move.

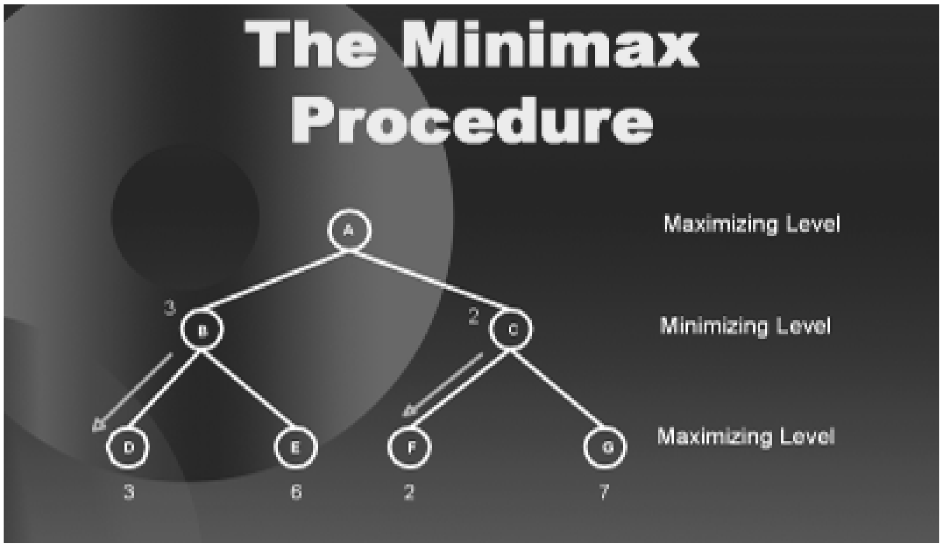
To clarify the concept of adversarial search let us discuss a procedure called the minimax procedure.

Here we assume that we have a situation analyzer that converts all judgments about board situations into a single, over all quality number. This situation analyzer is also called a static evaluator and the score/ number calculated by the evaluator is called the static evaluation of that node. Positive numbers, by convention indicate favor to one player. Negative numbers indicate favor to the other player. The player hoping for positive numbers is called maximizing player or maximizer. The other player is called minimizing player or minimizer. The maximizer has to keep in view that what choices will be available to the minimizer on the next step. The minimizer has to keep in view that what choices will be available to the maximizer on the next step.

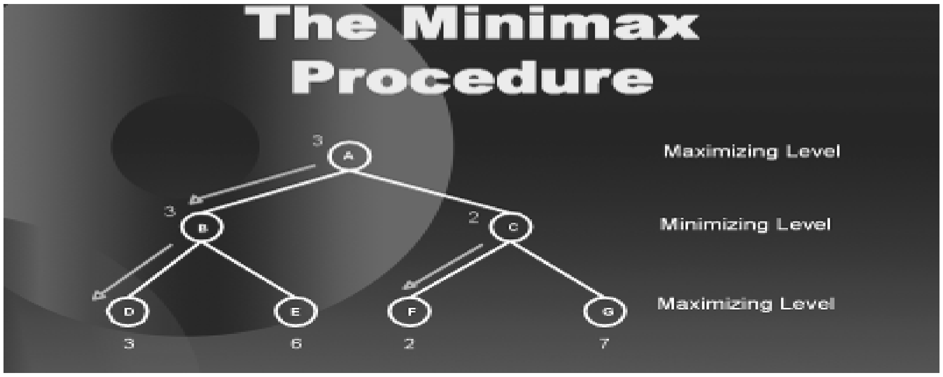
Consider the following diagram.



The Minimax Procedure



The Minimax Procedure



The Minimax Procedure

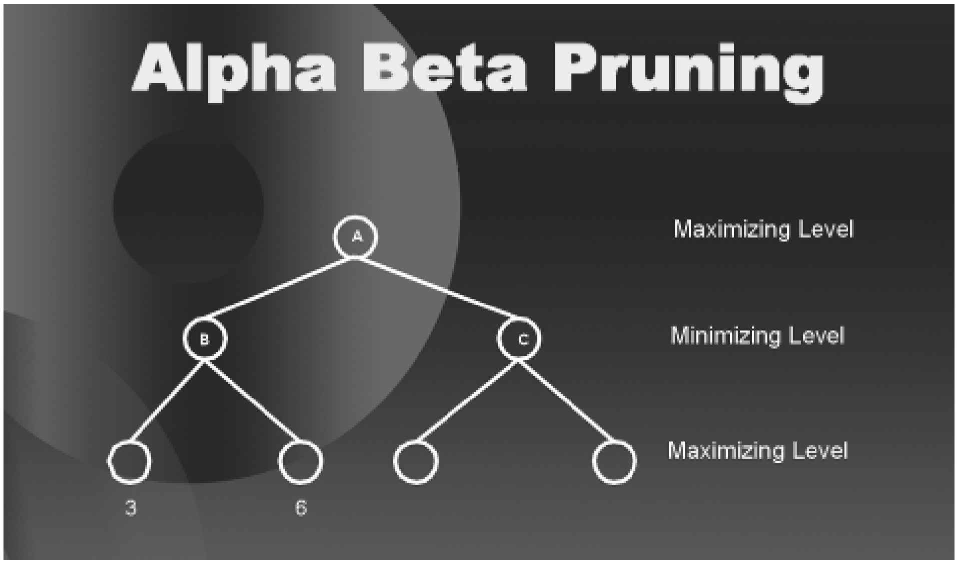
Standing at node A the maximizer wants to decide which node to visit next, that is, choose between B or C. The maximizer wishes to maximize the score so apparently 7 being the maximum score, the maximizer should go to C and then to G. But when the maximizer will reach C the next turn to select the node will be of the minimizer, which will force the game to reach configuration/node F with a score of 2. Hence maximizer will end up with a score of 2 if he goes to C from A. On the other hand, if the maximizer goes to B from A the worst which the minimizer can do is that he will force the maximizer to a score of 3. Now, since the choice is between scores of 3 or 2, the maximizer will go to node B from A.

2.24Alpha Beta Pruning

In Minimax Procedure, it seems as if the static evaluator must be used on each leaf node. Fortunately there is a procedure that reduces both the tree branches that must be generated and the number of evaluations. This procedure is called Alpha Beta pruning which “prunes” the tree branches thus reducing the number of static evaluations.

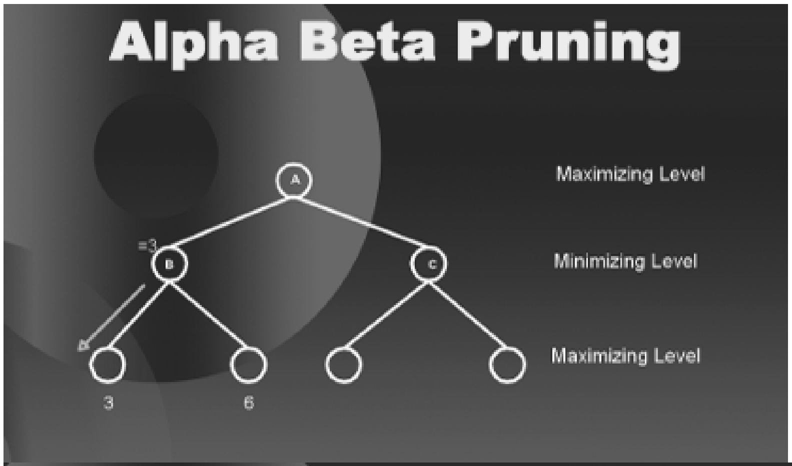
We use the following example to explain the notion of Alpha Beta Pruning.

Suppose we start of with a game tree in the diagram below. Notice that all nodes/situations have not yet been previously evaluated for their static evaluation score. Only two leaf nodes have been evaluated so far.

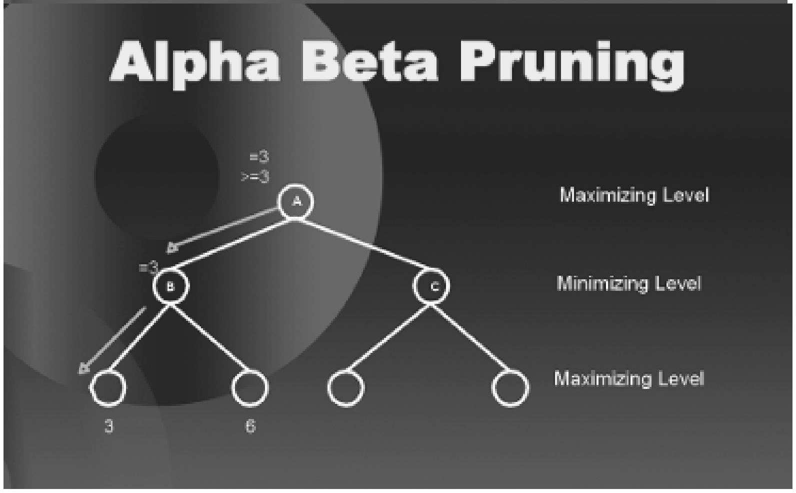


Alpha Beta Pruning

Sitting at A, the player-one will observe that if he moves to B the best he can get is 3.

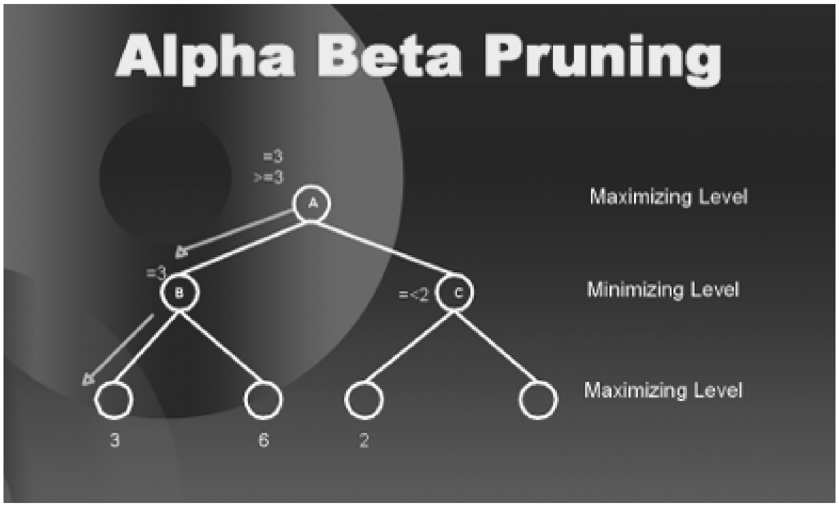


Alpha Beta Pruning



Alpha Beta Pruning

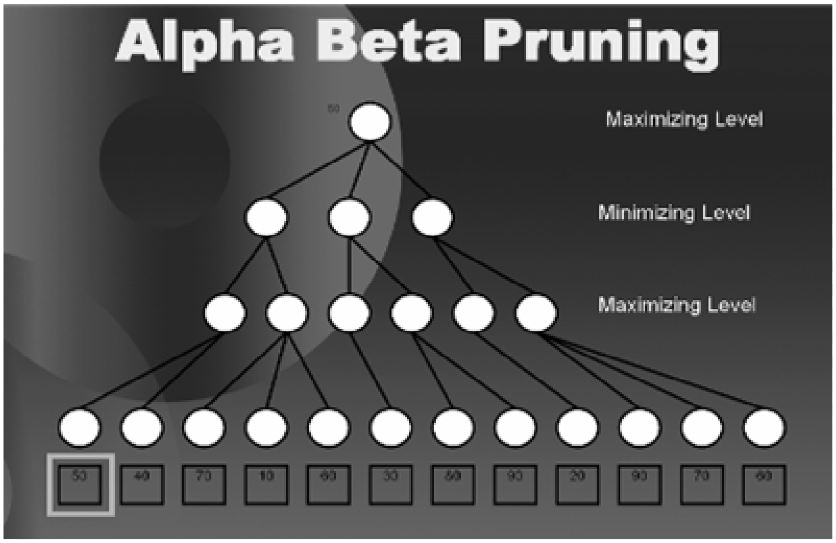
So the value three travels to the root A. Now after observing the other side of the tree, this score will either increase or will remain the same as this level is for the maximizer.



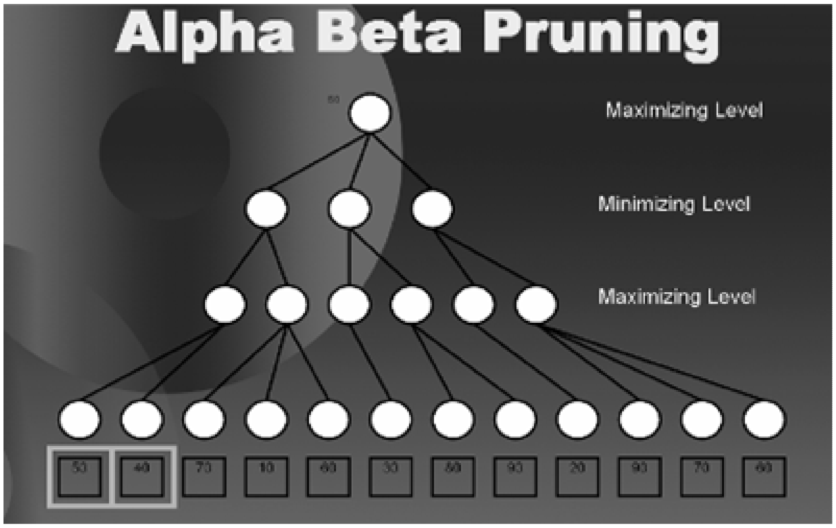
Alpha Beta Pruning

When he evaluates the first leaf node on the other side of the tree, he will see that the minimizer can force him to a score of less than 3 hence there is no need to fully explore the tree from that side. Hence the right most branch of the tree will be pruned and won't be evaluated for static evaluation.

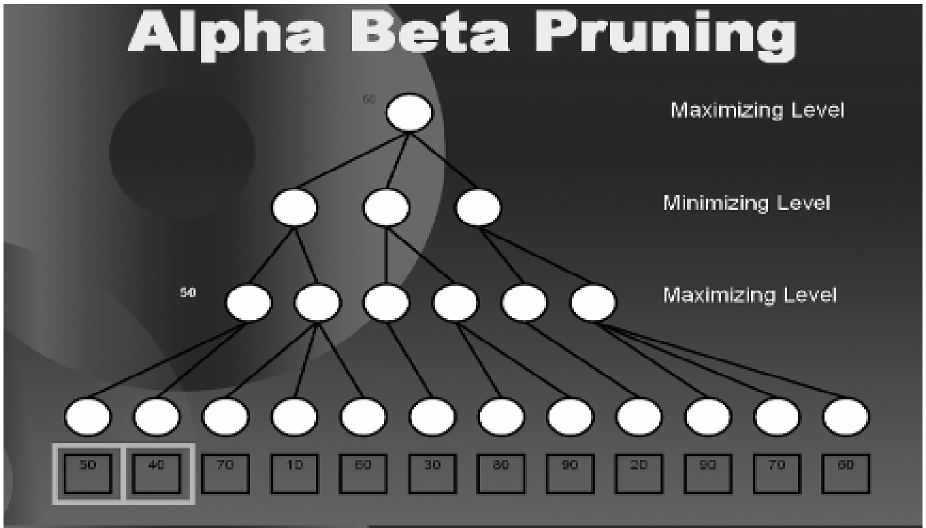
We have discussed a detailed example on Alpha Beta Pruning in the lectures. We have shown the sequence of steps in the diagrams below. The readers are required to go through the last portion of Lecture 10 for the explanation of this example, if required.



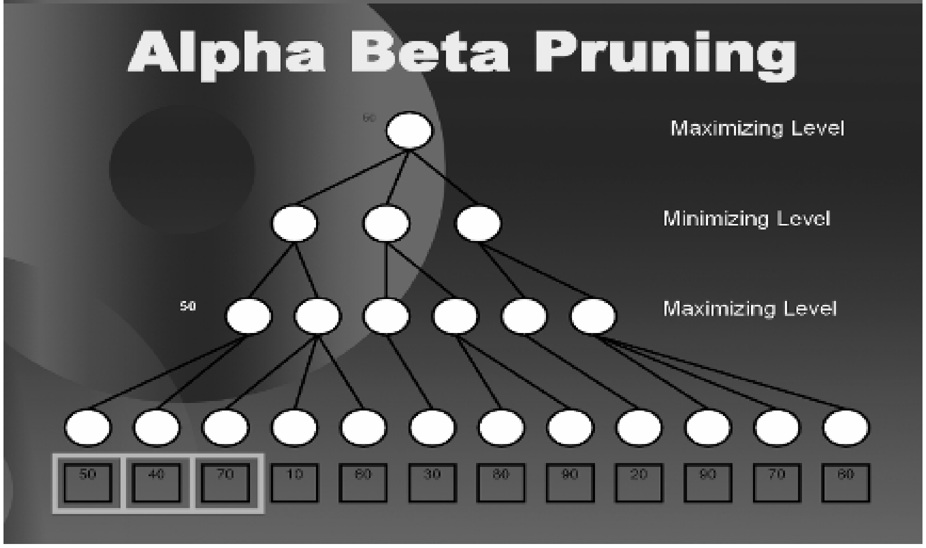
Alpha Beta Pruning



Alpha Beta Pruning

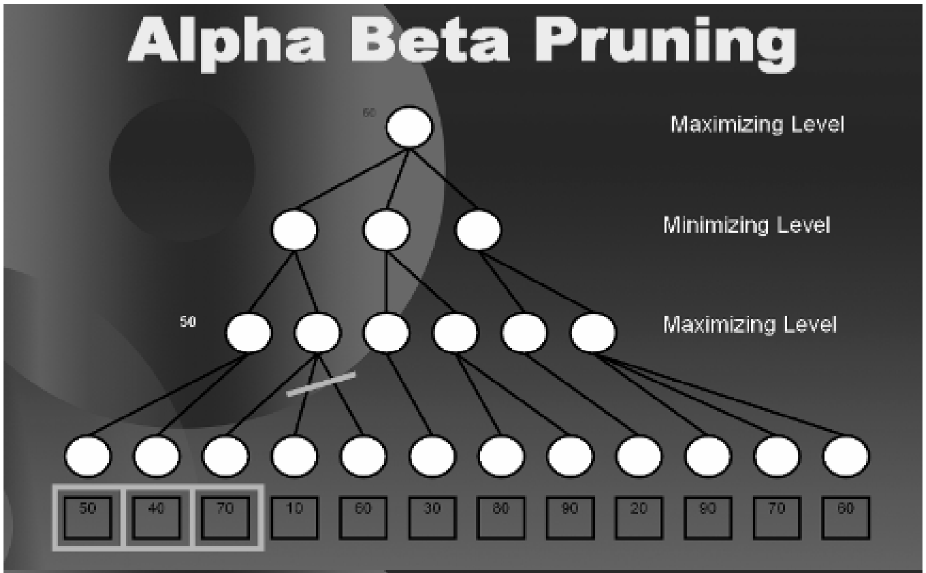


Alpha Beta Pruning

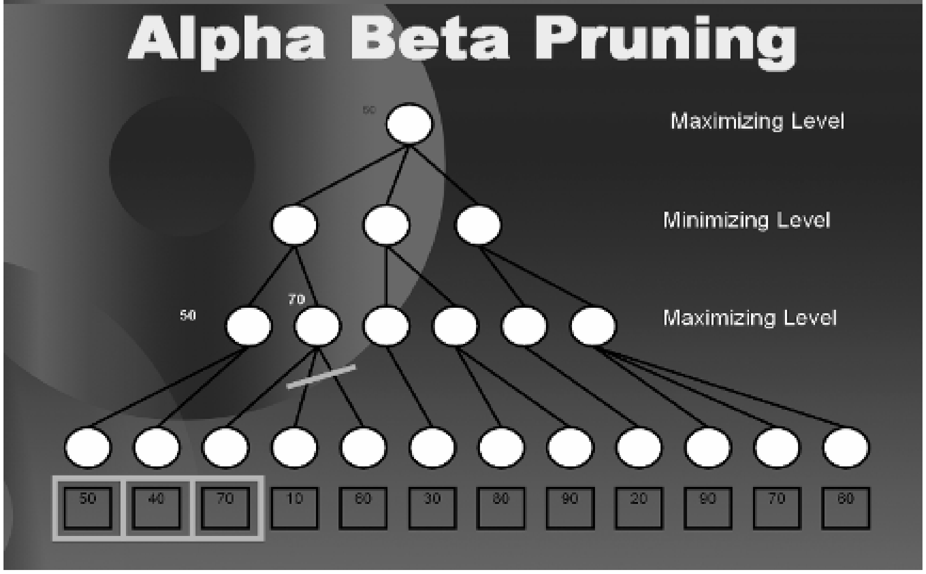


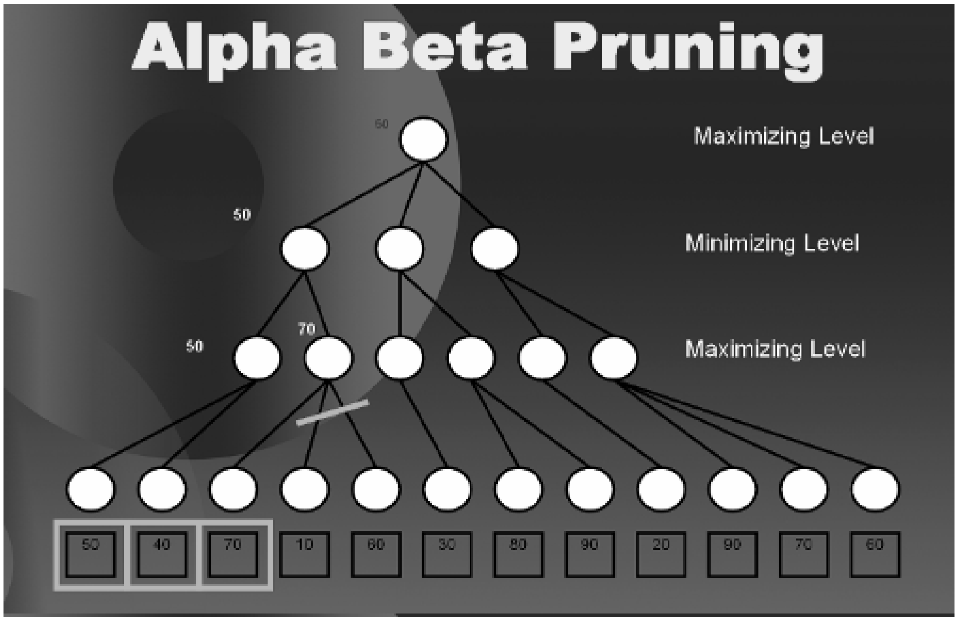
Alpha Beta Pruning

Alpha Beta Pruning

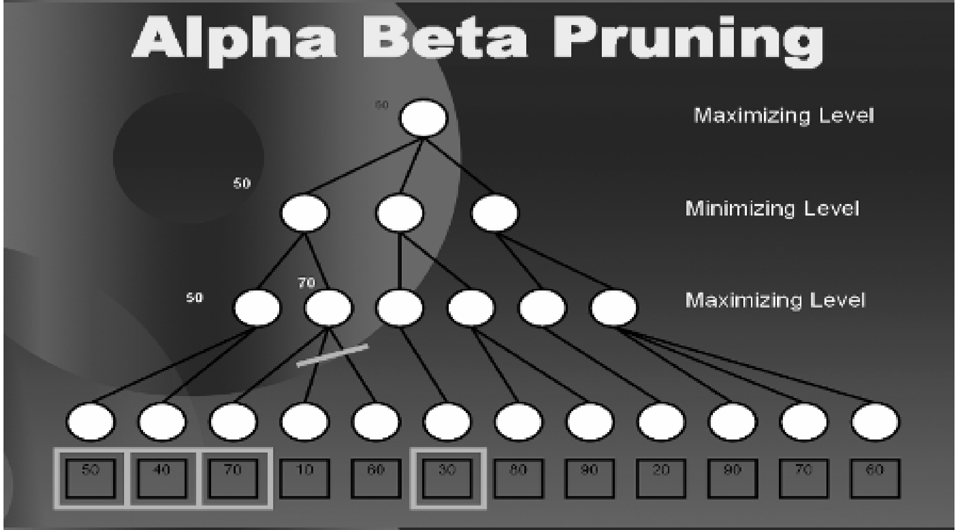


Alpha Beta Pruning

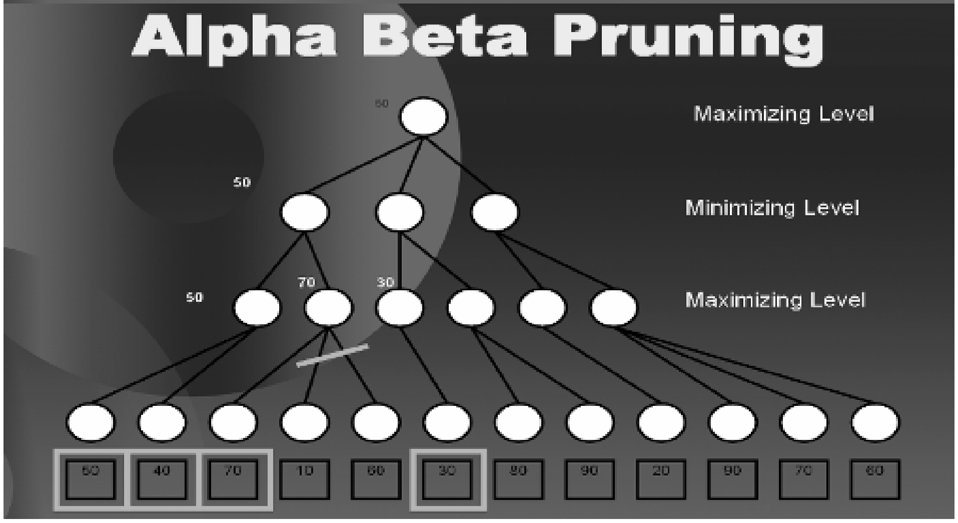




Alpha Beta Pruning

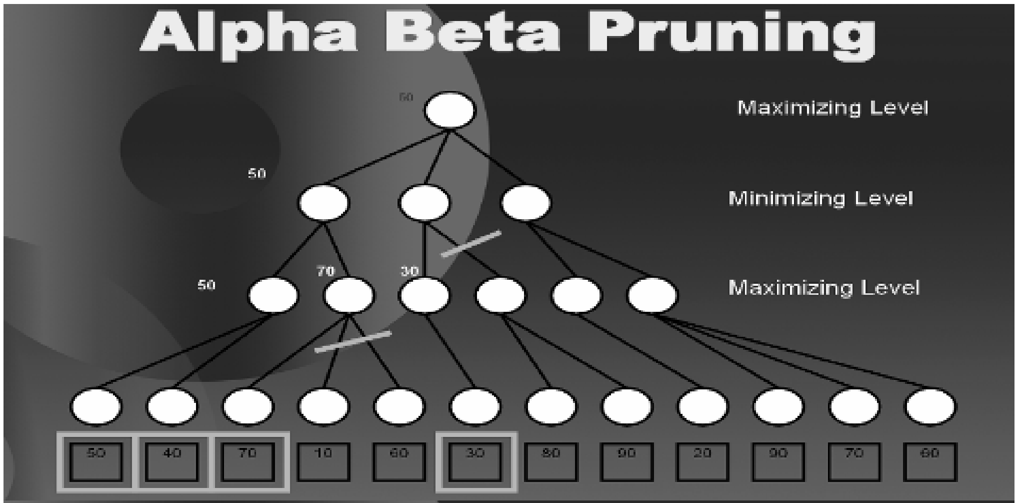


Alpha Beta Pruning



Alpha Beta Pruning

Alpha Beta Pruning

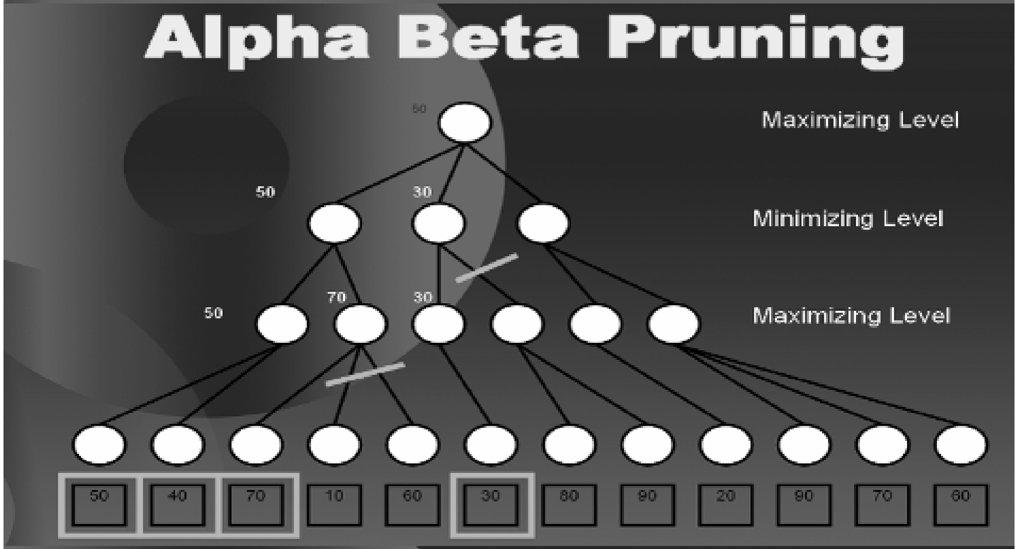


Maximizing Level

Minimizing Level

Maximizing Level

Alpha Beta Pruning

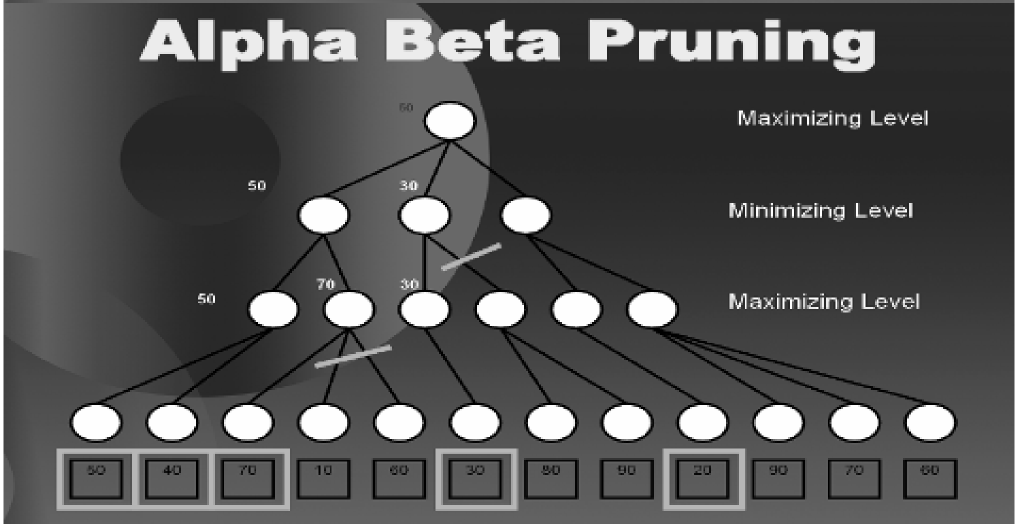


Maximizing Level

Minimizing Level

Maximizing Level

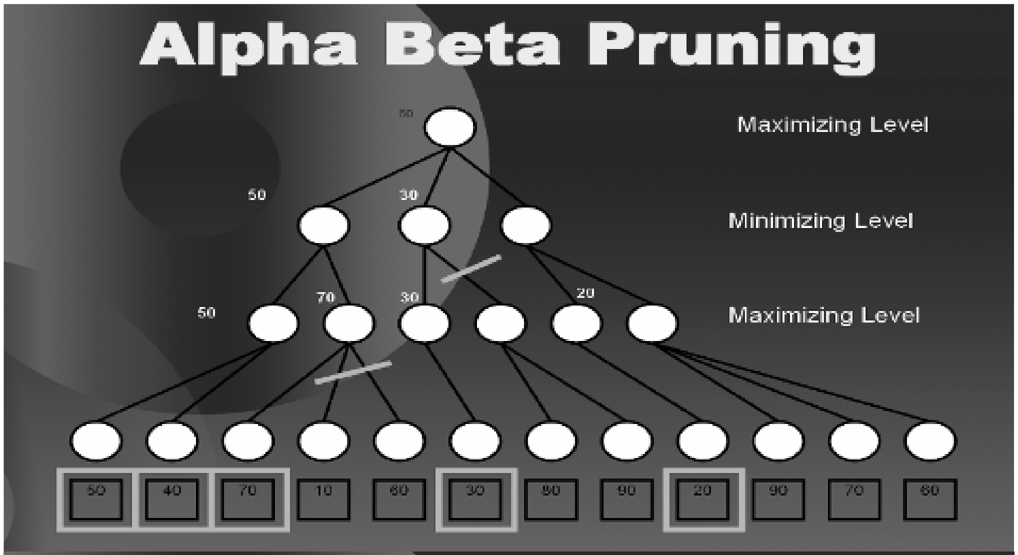
Alpha Beta Pruning



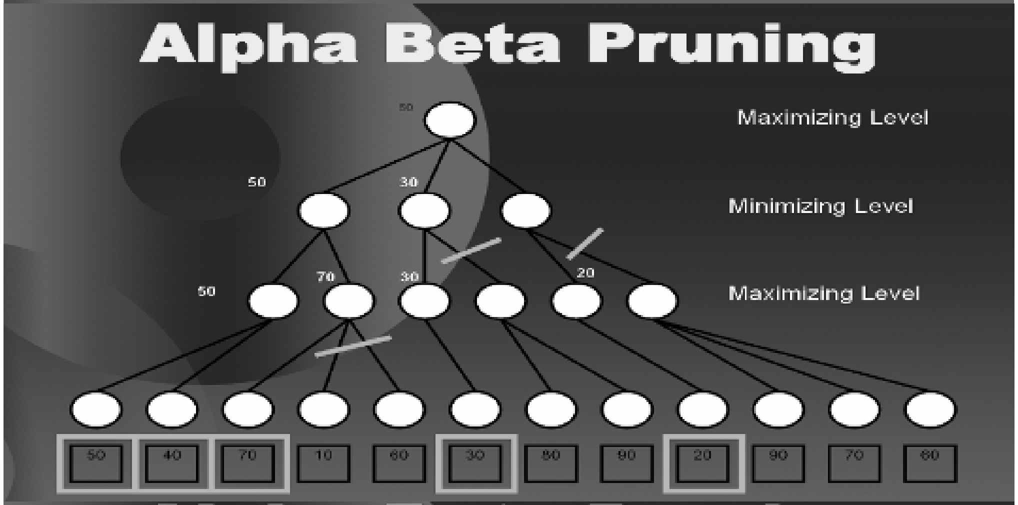
Maximizing Level

Minimizing Level

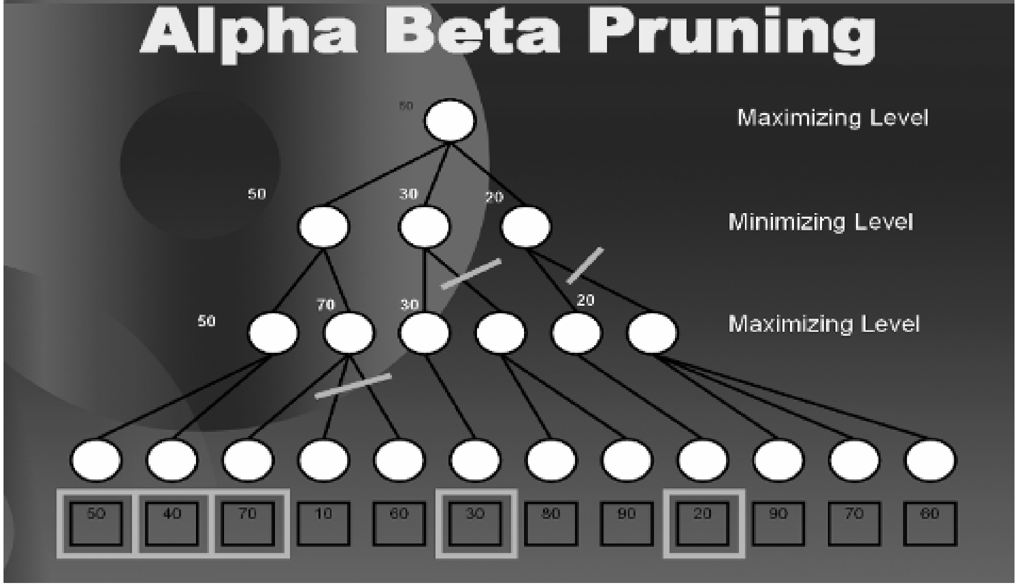
Maximizing Level



Alpha Beta Pruning



Alpha Beta Pruning



Alpha Beta Pruning

2.25 Summary

- People used to think that one who can solve more problems is more intelligent

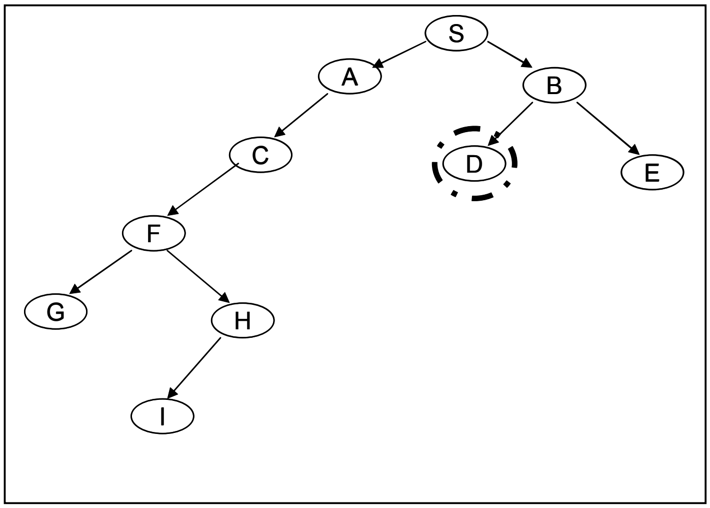
- Generate and test is the classical approach to solving problems
- Problem representation plays a key role in problem solving
- The components of problem solving include
    - Problem Statement
    - Operators
    - Goal State
    - Solution Space
- Searching is a formal mechanism to explore alternatives
- Searches can be blind or uninformed, informed, heuristic, non-optimal and optional.
- Different procedures to implement different search strategies form the major content of this chapter

2.26 Problems

Q1 Consider that a person has never been to the city air port. Its early in the morning and assume that no other person is awake in the town who can guide him on the way. He has to drive on his car but doesn't know the way to air port. Clearly identify the four components of problem solving in the above statement, i.e. problem statement, operators, solution space, and goal state. Should he follow blind or heuristic search strategy? Try to model the problem in a graphical representation.

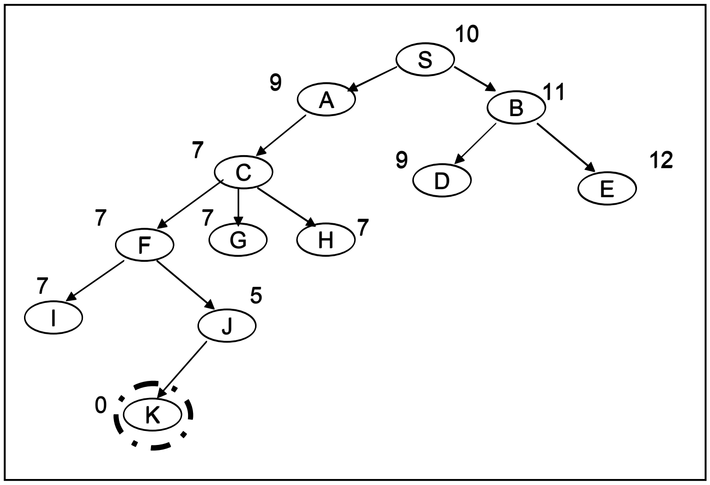
Q2 Clearly identify the difference between WSP (Well-Structured Problems) and ISP (Ill- Structured) problems as discussed in the lecture. Give relevant examples.

Q3 Given the following tree. Apply DFS and BFS as studied in the chapter. Show the state of the data structure **Q** and the visited list clearly at every step. S is the initial state and D is the goal state.



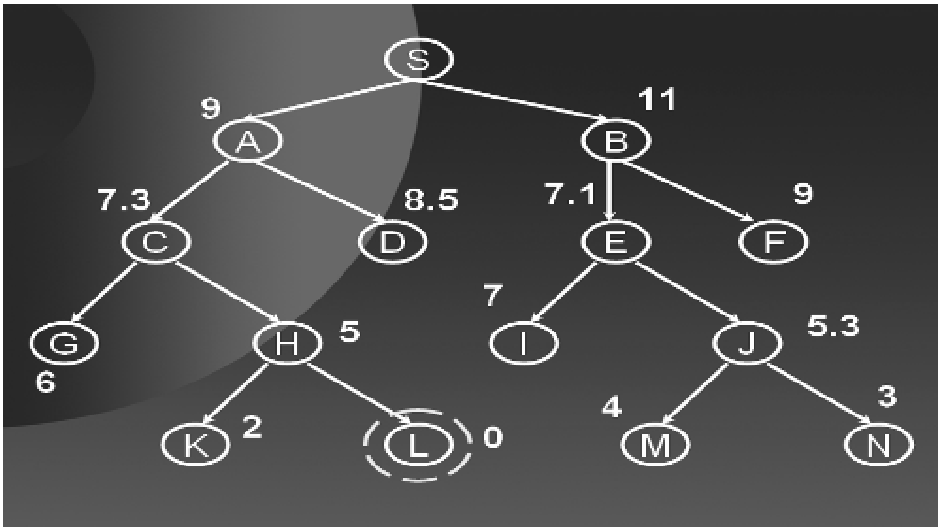
Q4 Discuss how progressive deepening uses a mixture of DFS and BFS to eliminate the disadvantages of both and at the same time finds the solution is a given tree. Support your answer with examples of a few trees.

Q5 Discuss the problems in Hill Climbing. Suggest solutions to the commonly encountered problems that are local maxima, plateau problem and ridge problem. Given the following tree, use the hill climbing procedure to climb up the tree. Use your suggested solutions to the above mention problems if any of them are encountered. K is the goal state and numbers written on each node is the estimate of remaining distance to the goal.

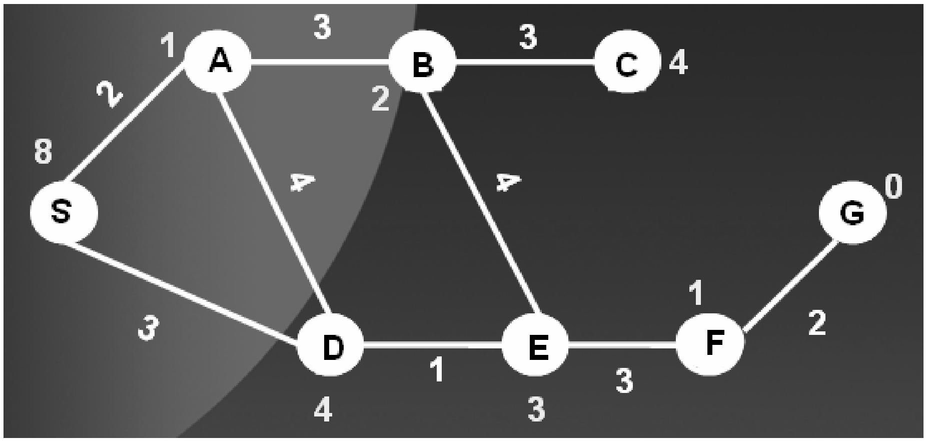


Q6 Discuss how best first search works in a tree. Support your answer with an example tree. Is best first search always the best strategy? Will it always guarantee the best solution?

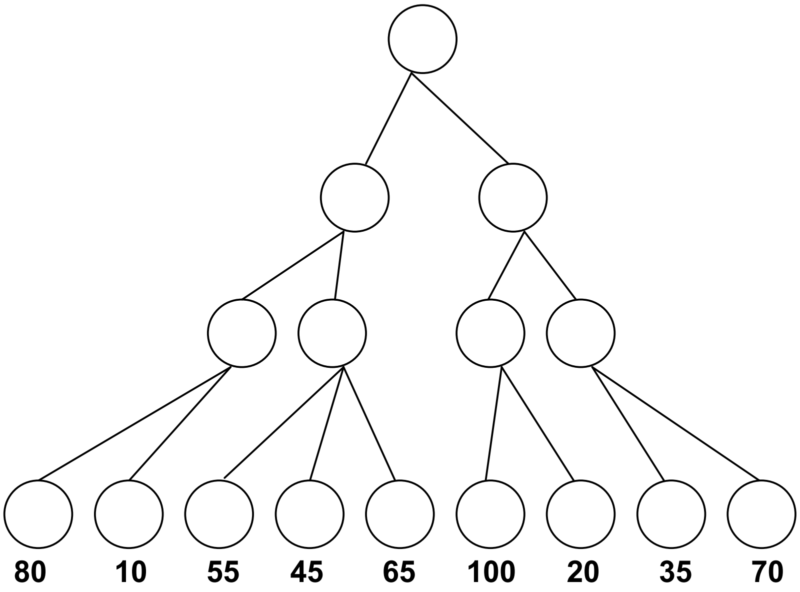
Q7 Discuss how beam search with degree of the search = 3 propagates in the given search tree. Is it equal to best first search when the degree = 1.



Q8 Discuss the main concept behind branch and bound search strategy. Suggest Improvements in the Algorithm. Simulate the algorithm on the given graph below. The values on the links are the distances between the cities. The numbers on the nodes are the estimated distance on the node from the goal state.

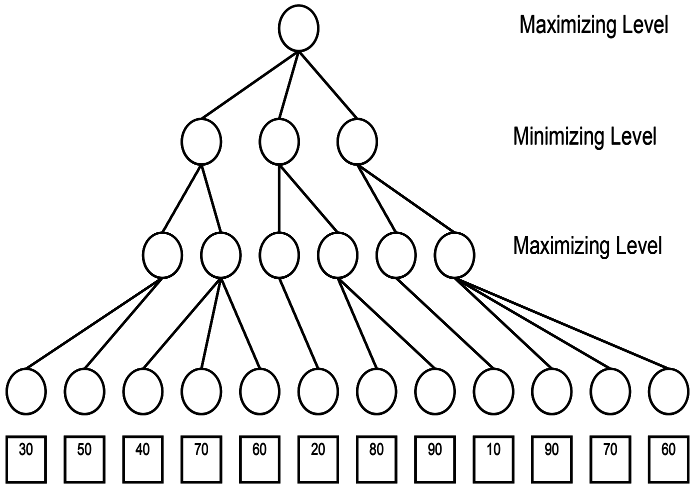


Q9. Run the MiniMax procedure on the given tree. The static evaluation scores for each leaf node are written under it. For example the static evaluation scores for the left most leaf node is 80.



Q10 Discuss how Alpha Beta Pruning minimizes the number of static evaluations at the leaf nodes by pruning branches. Support your answer with small examples of a few trees.

Q11 Simulate the Minimax procedure with Alpha Beta Pruning algorithm on the following search tree.



Adapted from: Artificial Intelligence, Third Edition by Patrick Henry Winston

Lecture No. 11-13

3 Genetic Algorithms

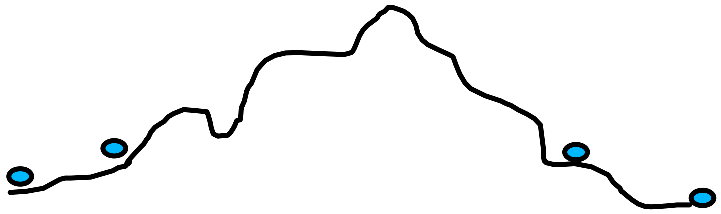
3.1 Discussion on Problem Solving

In the previous chapter we studied problem solving in general and elaborated on various search strategies that help us solve problems through searching in problem trees. We kept the information about the tree traversal in memory (in the queues), thus we know the links that have to be followed to reach the goal. At times we don't really need to remember the links that were followed. In many problems where the size of search space grows extremely large we often use techniques in which we don't need to keep all the history in memory. Similarly, in problems where requirements are not clearly defined and the problem is ill-structured, that is, we don't exactly know the initial state, goal state and operators etc, we might employ such techniques where our objective is to find the solution not how we got there.

Another thing we have noticed in the previous chapter is that we perform a sequential search through the search space. In order to speed up the techniques we can follow a parallel approach where we start from multiple locations (states) in the solution space and try to search the space in parallel.

3.2 Hill Climbing in Parallel

Suppose we were to climb up a hill. Our goal is to reach the top irrespective of how we get there. We apply different operators at a given position, and move in the direction that gives us improvement (more height). What if instead of starting from one position we start to climb the hill from different positions as indicated by the diagram below.



In other words, we start with different independent search instances that start from different locations to climb up the hill.

Further think that we can improve this using a collaborative approach where these instances interact and evolve by sharing information in order to solve the problem. You will soon find out that what we mean by interact and evolve.

However, it is possible to implement parallelism in the sense that the instances can interact and evolve to solve the solution. Such implementations and algorithms are motivated from the biological concept of evolution of our genes, hence the name **Genetic Algorithms**, commonly terms as GA.

3.3 Comment on Evolution

Before we discuss Genetic Algorithms in detail with examples lets go through some basic terminology that we will use to explain the technique. The genetic algorithm technology comes from the concept of human evolution. The following paragraph gives a brief overview of evolution and introduces some terminologies to the extent that we will require for further discussion on GA. Individuals (animals or plants) produce a number of offspring (children) which are almost, but not entirely, like themselves. Variation may be due to mutation (random changes), or due to inheritance (offspring/children inherit some characteristics from each parent). Some of these offspring may survive to produce offspring of their own—some will not. The "better adapted" individuals are more likely to survive. Over time, generations become better and better adapted to survive.

3.4 Genetic Algorithm

Genetic Algorithms is a search method in which multiple search paths are followed in parallel. At each step, current states of different pairs of these paths are combined to form new paths. This way the search paths don't remain independent, instead they share information with each other and thus try to improve the overall performance of the complete search space.

3.5 Basic Genetic Algorithm

A very basic genetic algorithm can be stated as below.

```
Start with a population of randomly generated, (attempted) solutions to a problem

Repeatedly do the following:
        Evaluate each of the attempted solutions
        Keep the "best" solutions
        Produce next generation from these solutions (using "inheritance" and "mutation")

Quit when you have a satisfactory solution (or you run out of time)
```

The two terms introduced here are inheritance and mutation. Inheritance has the same notion of having something or some attribute from a parent while mutation refers to a small random change. We will explain these two terms as we discuss the solution to a few problems through GA.

3.6 Solution to a Few Problems using GA

3.6.1 Problem 1:

- **Suppose your "individuals" are 32-bit computer words**
- **You want a string in which all the bits in these words are ones**
- **Here's how you can do it:**
    - **Create 100 randomly generated computer words**
    - **Repeatedly do the following:**
        - **Count the 1 bits in each word**
        - **Exit if any of the words have all 32 bits set to 1**
        - **Keep the ten words that have the most 1s (discard the rest)**
        - **From each word, generate 9 new words as follows:**
            - **Pick a random bit in the word and toggle (change) it**
- **Note that this procedure does not guarantee that the next "generation" will have more 1 bits, but it's likely**

As you can observe, the above solution is totally in accordance with the basic algorithm you saw in the previous section. The table on the next page shows which steps correspond to what.

Terms	**Basic GA**	**Problem 1 Solution**
Initial Population	Start with a population of randomly generated attempted solutions to a problem	Create 100 randomly generated computer words
Evaluation Function	Evaluate each of the attempted solutions. Keep the "best" solutions	Count the 1 bits in each word. Exit if any of the words have all 32 bits set to 1 Keep the ten words that have the most 1s (discard the rest)
Mutation	Produce next generation from these solutions (using "inheritance" and "mutation")	From each word, generate 9 new words as follows: Pick a random bit in the word and toggle (change) it

For the sake of simplicity we only use mutation for now to generate the new individuals. We will incorporate inheritance later in the example. Let's introduce the concept of an evaluation function. An evaluation function is the criteria that check various individuals/ solutions for being better than others in the population. Notice that mutation can be as simple as just flipping a bit at random or any number of bits.

We go on repeating the algorithm until we either get our required word that is a 32-bit number with all ones, or we run out of time. If we run out of time, we either present the best possible solution (the one with most number of 1-bits) as the answer or we can say that the solution can't be found. Hence GA is at times used to get optimal solution given some parameters.

3.6.2 Problem 2:

- **Suppose you have a large number of data points (x, y), e.g., (1, 4), (3, 9), (5, 8), ...**
- **You would like to fit a polynomial (of up to degree 1) through these data points**
    - **That is, you want a formula y = mx + c that gives you a reasonably good fit to the actual data**
    - **Here's the usual way to compute goodness of fit of the polynomial on the data points:**
        - **Compute the sum of (actual y – predicted $y)^2$ for all the data points**
        - **The lowest sum represents the best fit**
- **You can use a genetic algorithm to find a "pretty good" solution**

By a pretty good solution we simply mean that you can get reasonably good polynomial that best fits the given data.

- **Your formula is y = mx + c**
- **Your unknowns are m and c; where m and c are integers**
- **Your representation is the array [m, c]**
- **Your evaluation function for one array is:**
    - **For every actual data point (x, y)**
        - **Compute ý = mx + c**
        - **Find the sum of $(y – ý)^2$ over all x**
        - **The sum is your measure of "badness" (larger numbers are worse)**
    - **Example: For [5, 7] and the data points (1, 10) and (2, 13):**
        - **ý = 5x + 7 = 12 when x is 1**
        - **ý = 5x + 7 = 17 when x is 2**
        - **$(10 - 12)^2 + (13 – 17)^2 = 2^2 + 4^2 = 20$**
        - **If these are the only two data points, the "badness" of [5, 7] is 20**

- **Your algorithm might be as follows:**
  - **Create two-element arrays of random numbers**
  - **Repeat 50 times (or any other number):**
    - **For each of the arrays, compute its badness (using all data points)**
    - **Keep the best arrays (with low badness)**
    - **From the arrays you keep, generate new arrays as follows:**
      - **Convert the numbers in the array to binary, toggle one of the bits at random**
  - **Quit if the badness of any of the solution is zero**
  - **After all 50 trials, pick the best array as your final answer**

Let us solve this problem in detail. Consider that the given points are as follows.

- **(x, y) : {(1,5) (3, 9)}**

We start will the following initial population which are the arrays representing the solutions (m and c).

- **[2 7][1 3]**

Compute badness for [2 7]

- **ý = 2x + 7 = 9 when x is 1**
- **ý = 2x + 7 = 13 when x is 3**
- **$(5 – 9)^2 + (9 – 13)^2 = 4^2 + 4^2 = 32$**

- **ý = 1x + 3 = 4 when x is 1**
- **ý = 1x + 3 = 6 when x is 3**
- **$(5 – 4)^2 + (9 – 6)^2 = 1^2 + 3^2 = 10$**

- **Lets keep the one with low "badness" [1 3]**
- **Representation [001 011]**
- **Apply mutation to generate new arrays [011 011]**
- **Now we have [1 3] [3 3] as the new population considering that we keep the two best individuals**

Second iteration

- **(x, y) : {(1,5) (3, 9)}**
- **[1 3][3 3]**
  - **ý = 1x + 3 = 4 when x is 1**
  - **ý = 1x + 3 = 6 when x is 3**
  - **$(5 – 4)^2 + (9 – 6)^2 = 1^2 + 3^2 = 10$**

  - **ý = 3x + 3 = 6 when x is 1**
  - **ý = 3x + 3 = 12 when x is 3**

        - **$(5 – 6)^2 + (9 – 12)^2 = 1 + 9 = 10$**

- **Lets keep the [3 3]**
- **Representation [011 011]**
- **Apply mutation to generate new arrays [010 011]**
- **Now we have [3 3] [2 3] as the new population**

Third Iteration

- **(x, y) : {(1,5) (3, 9)}**
- **[3 3][2 3]**
    - **ý = 3x + 3 = 6 when x is 1**
    - **ý = 3x + 3 = 12 when x is 3**
    - **$(5 – 6)^2 + (9 – 12)^2 = 1 + 9 = 10$**

    - **ý = 2x + 3 = 5 when x is 1**
    - **ý = 2x + 3 = 9 when x is 3**
    - **$(5 – 5)^2 + (9 – 9)^2 = 0^2 + 0^2 = 0$**

- **Solution found [2 3]**
- **y = 2x+3**

So you see that how by going through the iteration of a GA one can find a solution to the given problem. It is not necessary in the above example that you get a solution that gives 0 badness. In case we go on doing iterations and we run out of time, we might just present the solution that has the least badness as the most optimal solution given these number of iterations on this data.

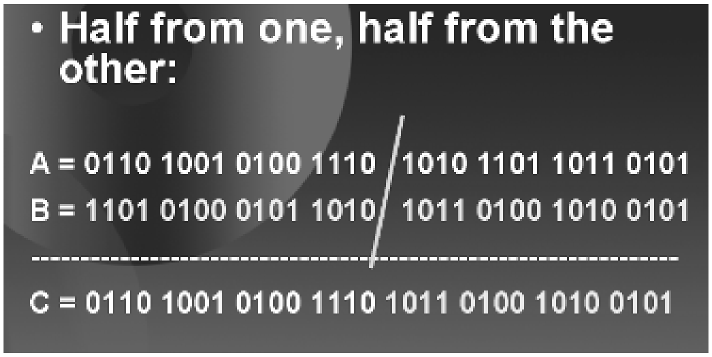
In the examples so far, each “Individual” (or “solution”) had only one parent. The only way to introduce variation was through mutation (random changes). In Inheritance or Crossover, each “Individual” (or “solution”) has two parents. Assuming that each organism has just one chromosome, new offspring are produced by forming a new chromosome from parts of the chromosomes of each parent.

Let us repeat the 32-bit word example again but this time using crossover instead of mutation.

- **Suppose your “organisms” are 32-bit computer words, and you want a string in which all the bits are ones**
- **Here’s how you can do it:**
    - **Create 100 randomly generated computer words**
    - **Repeatedly do the following:**
        - **Count the 1 bits in each word**
        - **Exit if any of the words have all 32 bits set to 1**
        - **Keep the ten words that have the most 1s (discard the rest)**
        - **From each word, generate 9 new words as follows:**
            - **Choose one of the other words**

- **Take the first half of this word and combine it with the second half of the other word**

Notice that we are generating new individuals from the best ones by using crossover. The simplest way to perform this crossover is to combine the head of one individual to the tail of the other, as shown in the diagram below.



- Half from one, half from the other:

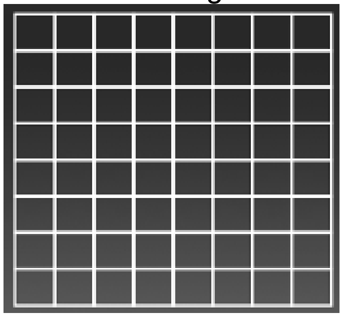
```
A = 0110 1001 0100 1110 /1010 1101 1011 0101
B = 1101 0100 0101 1010/ 1011 0100 1010 0101
-------------------------------------------------------------
C = 0110 1001 0100 1110 1011 0100 1010 0101
```

In the 32-bit word problem, the (two-parent, no mutation) approach, if it succeeds, is likely to succeed much faster because up to half of the bits change each time, not just one bit. However, with no mutation, it may not succeed at all. By pure bad luck, maybe none of the first (randomly generated) words have (say) bit 17 set to 1. Then there is no way a 1 could ever occur in this position. Another problem is lack of genetic diversity. Maybe some of the first generation did have bit 17 set to 1, but none of them were selected for the second generation. The best technique in general turns out to be a combination of both, i.e., crossover with mutation.

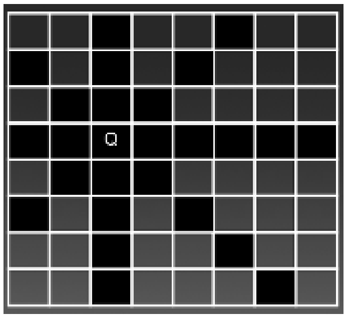
3.7 Eight Queens Problem

Let us now solve a famous problem which will be discussed under GA in many famous books in AI. Its called the Eight Queens Problem.

The problem is to place 8 queens on a chess board so that none of them can attack the other. A chess board can be considered a plain board with eight columns and eight rows as shown below.

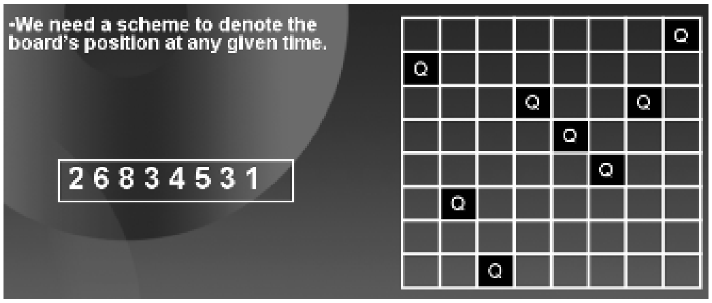


The possible cells that the Queen can move to when placed in a particular square are shown (in black shading)



We now have to come up with a representation of an individual/ candidate solution representing the board configuration which can be used as individuals in the GA.

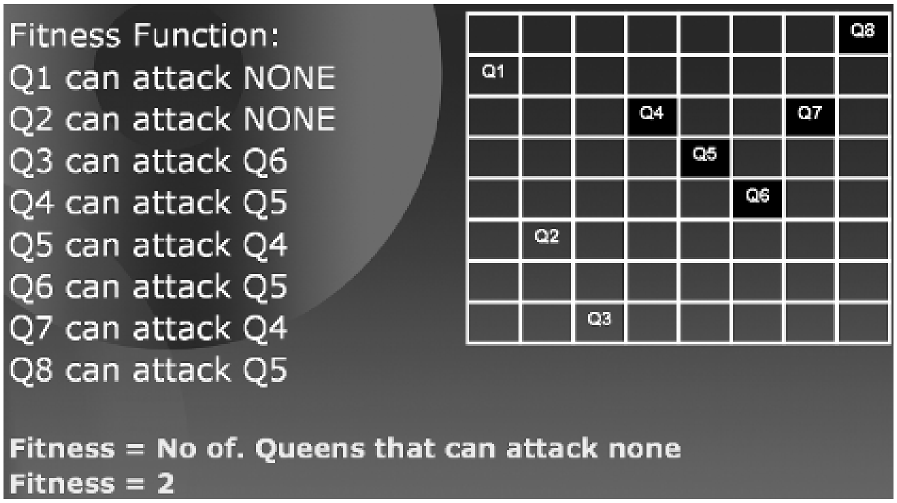
We will use the representation as shown in the figure below.


-We need a scheme to denote the board's position at any given time.

2 6 8 3 4 5 3 1

Where the 8 digits for eight columns specify the index of the row where the queen is placed. For example, the sequence 2 6 8 3 4 5 3 1 tells us that in first column the queen is placed in the second row, in the second column the queen is in the 6th row so on till in the 8th column the queen is in the 1st row.

Now we need a fitness function, a function by which we can tell which board position is nearer to our goal. Since we are going to select best individuals at every step, we need to define a method to rate these board positions or individuals. One fitness function can be to count the number of pairs of Queens that are not attacking each other. An example of how to compute the fitness of a board configuration is given in the diagram on the next page.



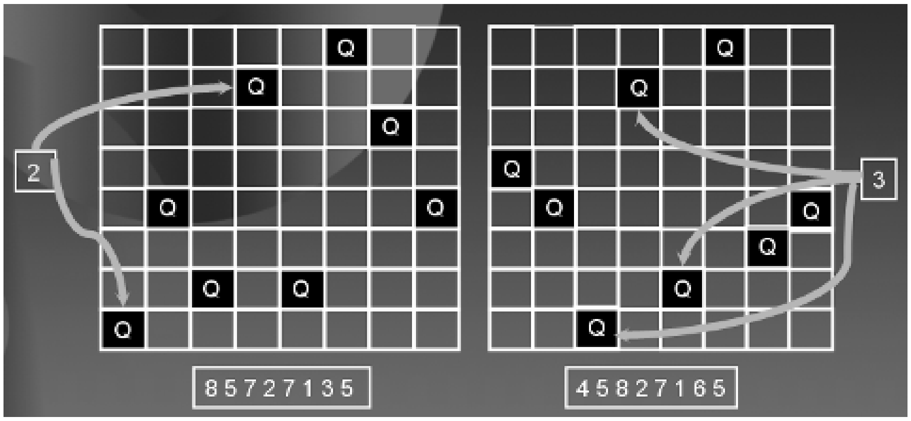
Fitness Function:
Q1 can attack NONE
Q2 can attack NONE
Q3 can attack Q6
Q4 can attack Q5
Q5 can attack Q4
Q6 can attack Q5
Q7 can attack Q4
Q8 can attack Q5

Fitness = No of. Queens that can attack none
Fitness = 2

So once representation and fitness function is decided, the solution to the problem is simple.

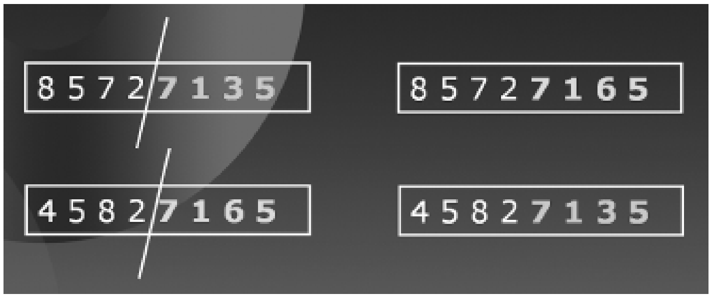
- **Choose initial population**
- **Evaluate the fitness of each individual**
- **Choose the best individuals from the population for crossover**

Let us quickly go through an example of how to solve this problem using GA. Suppose individuals (board positions) chosen for crossover are:



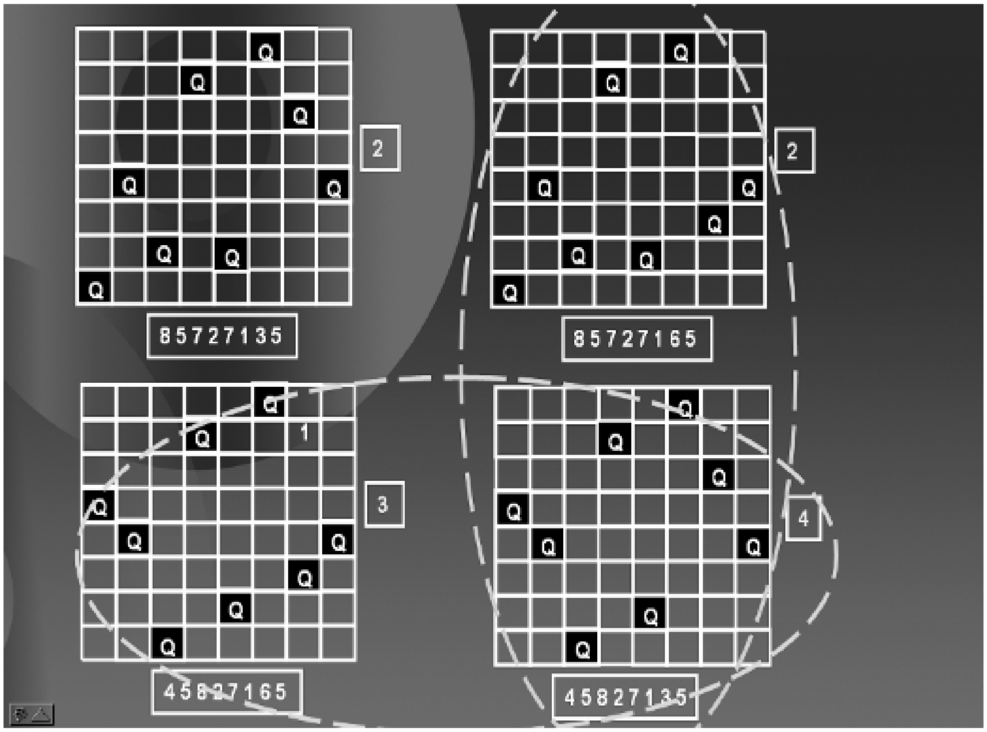
Where the numbers 2 and 3 in the boxes to the left and right show the fitness of each board configuration and green arrows denote the queens that can attack none.

The following diagram shows how we apply crossover:

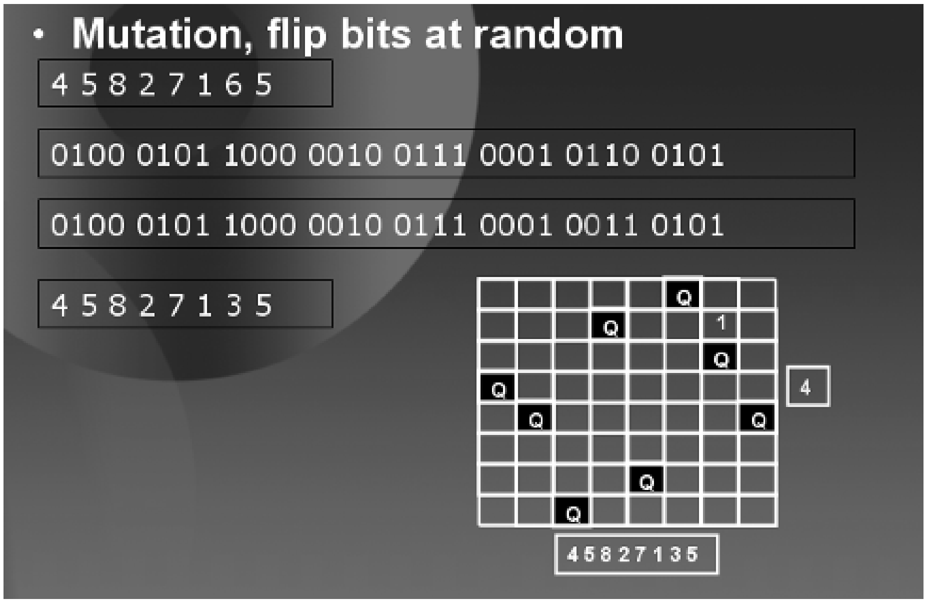


The individuals in the initial population are shown on the left and the children generated by swapping their tails are shown on the right. Hence we now have a total of 4 candidate solutions. Depending on their fitness we will select the best two.

The diagram below shows where we select the best two on the bases of their fitness. The vertical over shows the children and the horizontal oval shows the selected individuals which are the fittest ones according to the fitness function.



Similarly, the mutation step can be done as under.



- Mutation, flip bits at random

4 5 8 2 7 1 6 5

0100 0101 1000 0010 0111 0001 0110 0101

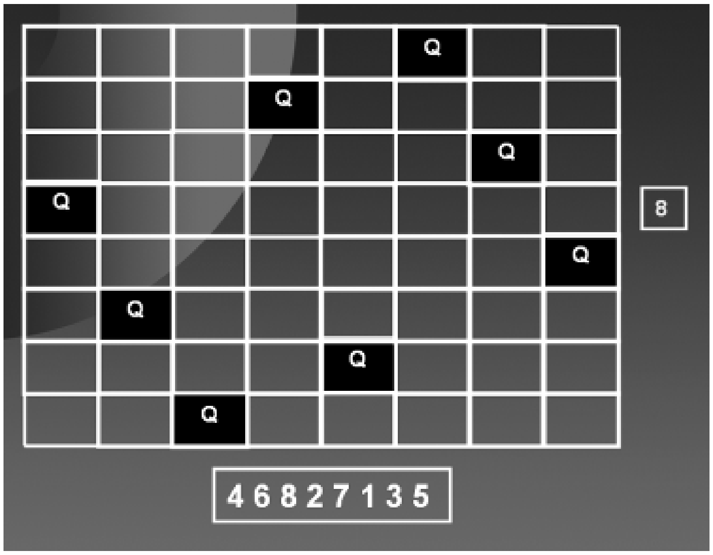
0100 0101 1000 0010 0111 0001 0011 0101

4 5 8 2 7 1 3 5

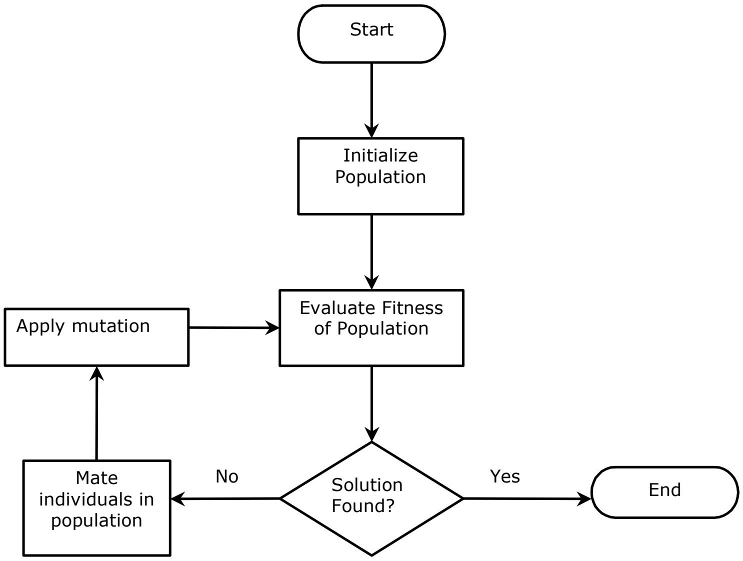
That is, we represent the individual in binary and we flip at random a certain number of bits. You might as well decide to flip 1, 2, 3 or k number of bits, at random position. Hence GA is totally a random technique.

This process is repeated until an individual with required fitness level is found. If no such individual is found, then the process is repeated till the overall fitness of the population or any of its individuals gets very close to the required fitness level. An upper limit on the number of iterations is usually used to end the process in finite time.

One of the solutions to the problem is shown as under whose fitness value is 8.



The following flow chart summarizes the Genetic Algorithm.



You are encouraged to explore the internet and other books to find more applications of GA in various fields like:

- Genetic Programming
- Evolvable Systems
- Composing Music
- Gaming
- Market Strategies
- Robotics
- Industrial Optimization

and many more.

3.8 Problems

Q1 what type of problems can be solved using GA. Give examples of at least 3 problems from different fields of life. Clearly identify the initial population, representation, evaluation function, mutation and cross over procedure and exit criteria.

Q2 Given pairs of (x, y) coordinates, find the best possible m, c parameters of the line y = mx + c that generates them. Use mutation only. Present the best possible solution given the data after at least three iterations of GA or exit if you find the solution earlier.

- **(x, y) : {(1,2.5) (2, 3.75)}**
- **Initial population [2 0][3 1]**

Q3 Solve the 8 Queens Problem on paper. Use the representations and strategy as discussed in the chapter.

Lecture No. 14 -17

4 Knowledge Representation and Reasoning

Now that have looked at general problem solving, lets look at knowledge representation and reasoning which are important aspects of any artificial intelligence system and of any computer system in general. In this section we will become familiar with classical methods of knowledge representation and reasoning in AI.

4.1 The AI Cycle

Almost all AI systems have the following components in general:

- Perception
- Learning
- Knowledge Representation and Reasoning
- Planning
- Execution

Figure 1 shows the relationship between these components.

An AI system has a perception component that allows the system to get information from its environment. As with human perception, this may be visual, audio or other forms of sensory information. The system must then form a meaningful and useful representation of this information internally. This knowledge representation maybe static or it may be coupled with a learning component that is adaptive and draws trends from the perceived data.

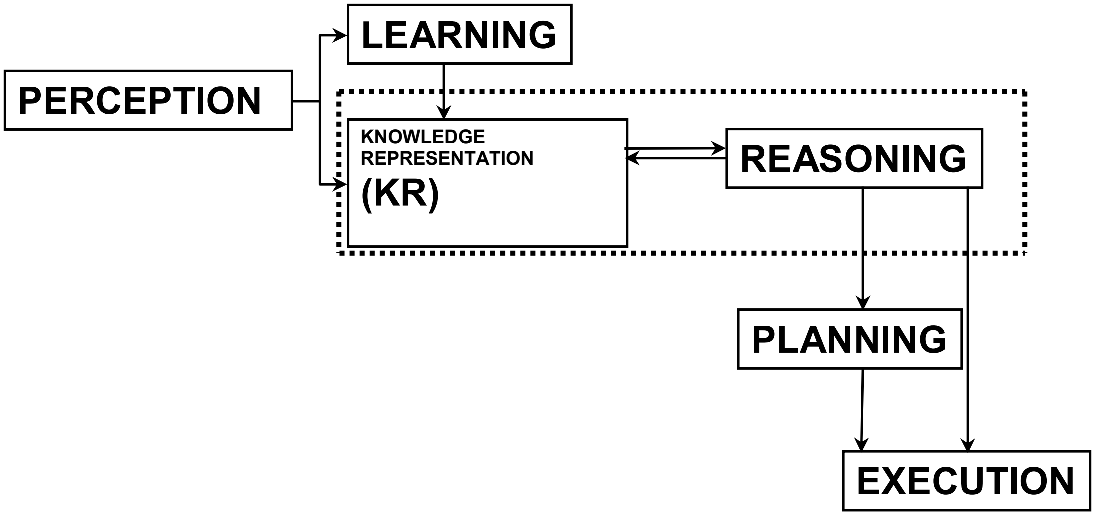


Figure 1: The AI Cycle

Knowledge representation (KR) and reasoning are closely coupled components; each is intrinsically tied to the other. A representation scheme is not meaningful on its own; it must be useful and helpful in achieve certain tasks. The same information may be represented in many different ways, depending on how you want to use that information. For example, in mathematics, if we want to solve problems about ratios, we would most likely use algebra, but we could also use simple hand drawn symbols. To say half of something, you could use 0.5x or you could draw a picture of the object with half of it colored differently. Both would convey the same information but the former is more compact and useful in complex scenarios where you want to perform reasoning on the information. It is important at this point to understand how knowledge representation and reasoning are interdependent components, and as AI system designer, you have to consider this relationship when coming up with any solution.

4.2 The dilemma

The key question when we begin to think about knowledge representation and reasoning is how to approach the problem ----should we try to emulate the human brain completely and exactly as it is? Or should we come up with something new?

Since we do not know how the KR and reasoning components are implemented in humans, even though we can see their manifestation in the form of intelligent behavior, we need a synthetic (artificial) way to model the knowledge representation and reasoning capability of humans in computers.

4.3 Knowledge and its types

Before we go any further, lets try to understand what ‘knowledge’ is. Durkin refers to it as the “Understanding of a subject area”. A well-focused subject area is referred to as a knowledge domain, for example, medical domain, engineering domain, business domain, etc..

If we analyze the various types of knowledge we use in every day life, we can broadly define knowledge to be one of the following categories:

- Procedural knowledge: Describes how to do things, provides a set of directions of how to perform certain tasks, e.g., how to drive a car.

- Declarative knowledge: It describes objects, rather than processes. What is known about a situation, e.g. it is sunny today, and cherries are red.

- Meta knowledge: Knowledge about knowledge, e.g., the knowledge that blood pressure is more important for diagnosing a medical condition than eye color.

- Heuristic knowledge: Rule-of-thumb, e.g. if I start seeing shops, I am close to the market.
    - Heuristic knowledge is sometimes called shallow knowledge.
    - Heuristic knowledge is empirical as opposed to deterministic

- Structural knowledge: Describes structures and their relationships. e.g. how the various parts of the car fit together to make a car, or knowledge structures in terms of concepts, sub concepts, and objects.

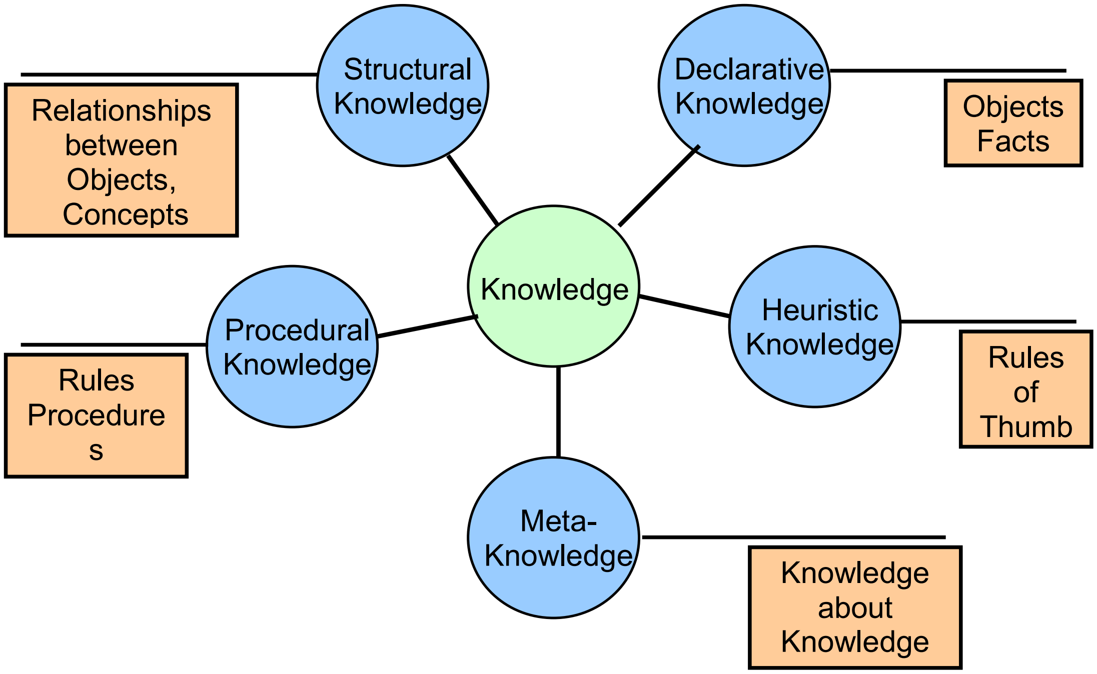


Fig 2: Types of Knowledge

4.4 Towards Representation

There are multiple approaches and schemes that come to mind when we begin to think about representation

- Pictures and symbols. This is how the earliest humans represented knowledge when sophisticated linguistic systems had not yet evolved
- Graphs and Networks
- Numbers

4.4.1 Pictures

Each type of representation has its benefits. What types of knowledge is best represented using pictures? , e.g. can we represent the relationship between individuals in a family using a picture? We could use a series of pictures to store procedural knowledge, e.g. how to boil an egg. But we can easily see that pictures are best suited for recognition tasks and for representing structural information. However, pictorial representations are not very easily translated to useful information in computers because computers cannot interpret pictures directly with out complex reasoning. So even though pictures are useful for human understanding, because they provide a high level view of a concept to be obtained readily, using them for representation in computers is not as straight forward.

4.4.2 Graphs and Networks

Graphs and Networks allow *relationships* between objects/entities to be incorporated, e.g., to show family relationships, we can use a graph.

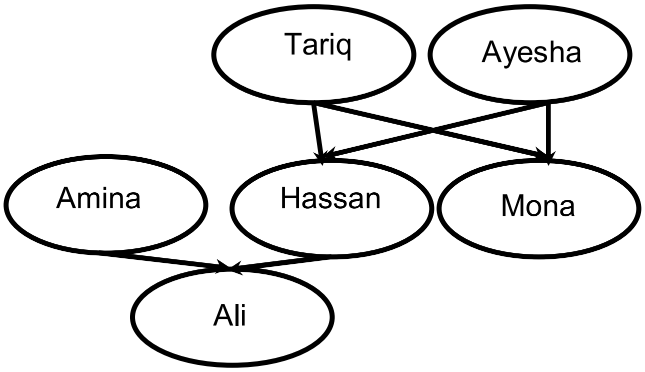


Fig 3: Family Relationships

We can also represent procedural knowledge using graphs, e.g. How to start a car?

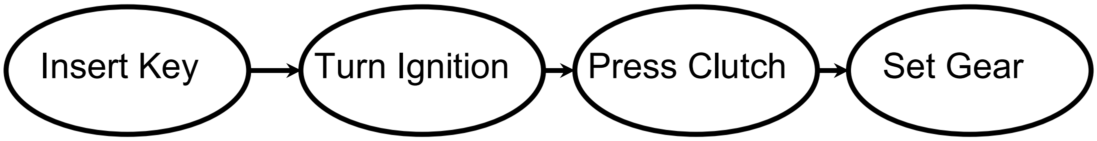


Fig 4: Graph for procedural knowledge

4.4.3 Numbers

Numbers are an integral part of knowledge representation used by humans. Numbers translate easily to computer representation. Eventually, as we know, every representation we use gets translated to numbers in the computers internal representation.

4.4.4 An Example

In the context of the above discussion, let’s look at some ways to represent the knowledge of a family

Using a picture

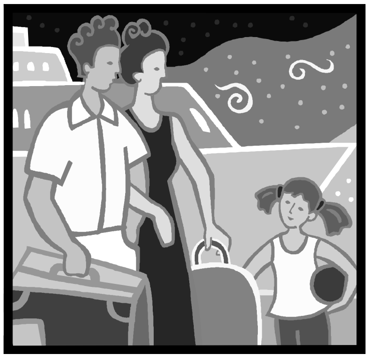


Fig 5: Family Picture

As you can see, this kind of representation makes sense readily to humans, but if we give this picture to a computer, it would not have an easy time figuring out the relationships between the individuals, or even figuring out how many individuals are there in the picture. Computers need complex computer vision algorithms to understand pictures.

Using a graph

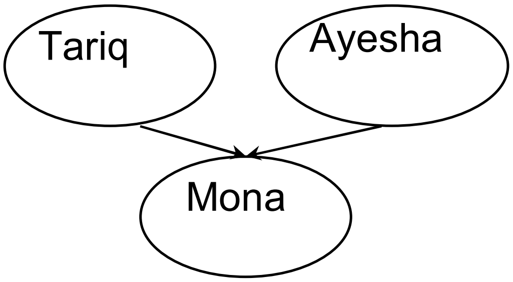


Fig 6: Family graph

This representation is more direct and highlights relationships.

Using a description in words

For the family above, we could say in words

- Tariq is Mona’s Father
- Ayesha is Mona’s Mother
- Mona is Tariq and Ayesha’s Daughter

This example demonstrates the fact that each knowledge representation scheme has its own strengths and weaknesses.

4.5 Formal KR techniques

In the examples above, we explored intuitive ways for knowledge representation. Now, we will turn our attention to formal KR techniques in AI. While studying these techniques, it is important to remember that each method is suited to representing a certain type of knowledge. Choosing the proper representation is important because it must help in reasoning. As the saying goes ‘Knowledge is Power’.

4.6 Facts

Facts are a basic block of knowledge (the atomic units of knowledge). They represent declarative knowledge (they declare knowledge about objects). A *proposition* is the statement of a fact. Each proposition has an associated *truth value*. It may be either true or false. In AI, to represent a fact, we use a proposition and its associated truth value, e.g.

–Proposition A: It is raining
–Proposition B: I have an umbrella
–Proposition C: I will go to school

4.6.1 Types of facts

Single-valued or multiple –valued

Facts may be single-valued or multi-valued, where each fact (attribute) can take one or more than one values at the same time, e.g. an individual can only have one eye color, but may have many cars. So the value of attribute cars may contain more than one value.

Uncertain facts

Sometimes we need to represent uncertain information in facts. These facts are called uncertain facts, e.g. it will probably be sunny today. We may chose to store *numerical certainty values* with such facts that tell us how much uncertainty there is in the fact.

Fuzzy facts

Fuzzy facts are *ambiguous* in nature, e.g. the book is heavy/light. Here it is unclear what heavy means because it is a subjective description. Fuzzy representation is used for such facts. While defining fuzzy facts, we use certainty factor values to specify value of “truth”. We will look at fuzzy representation in more detail later.

Object-Attribute-Value triplets

Object-Attribute Value Triplets or OAV triplets are a type of fact composed of three parts; object, attribute and value. Such facts are used to assert a particular property of some object, e.g.

Ali’s eye color is brown.

- Object: Ali
- Attribute: eye color

- Value: brown

Ahmed’s son is Ali

- Object: Ahmed
- Attribute: son
- Value: Ali

OAV Triplets are also defined as in figure below

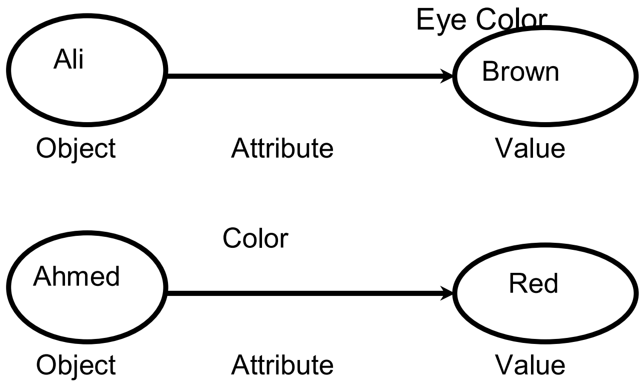


Figure: OAV Triplets

4.7 Rules

Rules are another form of knowledge representation. Durkin defines a rule as “A knowledge structure that relates some known information to other information that can be concluded or inferred to be true.”

4.7.1 Components of a rule

A Rule consists of two components

- Antecedent or premise or the IF part
- Consequent or conclusion or the THEN part

For example, we have a rule: IF it is raining THEN I will not go to school
Premise: It is raining
Conclusion: I will not go to school.

4.7.2 Compound Rules

Multiple premises or antecedents may be joined using AND (conjunctions) and OR (disjunctions), e.g.

IF it is raining AND I have an umbrella
THEN I will go to school.

IF it is raining OR it is snowing
THEN I will not go to school

4.7.3 Types of rules

Relationship

Relationship rules are used to express a direct occurrence relationship between two events, e.g. IF you hear a loud sound THEN the silencer is not working

Recommendation

Recommendation rules offer a recommendation on the basis of some known information, e.g.

IF it is raining
THEN bring an umbrella

Directive

Directive rules are like recommendations rule but they offer a specific line of action, as opposed to the ‘advice’ of a recommendation rule, e.g.

IF it is raining AND you don’t have an umbrella
THEN wait for the rain to stop

Variable Rules

If the same type of rule is to be applied to multiple objects, we use variable rules, i.e. rules with variables, e.g.

If X is a Student
AND X’s GPA>3.7
THEN place X on honor roll.

Such rules are called pattern-matching rules. The rule is matched with known facts and different possibilities for the variables are tested, to determine the truth of the fact.

Uncertain Rules

Uncertain rules introduce uncertain facts into the system, e.g.
IF you have never won a match
THEN you will most probably not win this time.

Meta Rules

Meta rules describe how to use other rules, e.g.
IF you are coughing AND you have chest congestion
THEN use the set of respiratory disease rules.

Rule Sets

As in the previous example, we may group rules into categories in our knowledge representation scheme, e.g. the set of respiratory disease rules

4.8 Semantic networks

Semantic networks are graphs, with nodes representing objects and arcs representing relationships between objects. Various types of relationships may be defined using semantic networks. The two most common types of relationships are

–IS-A (Inheritance relation)

–HAS (Ownership relation)

Let's consider an example semantic network to demonstrate how knowledge in a semantic network can be used

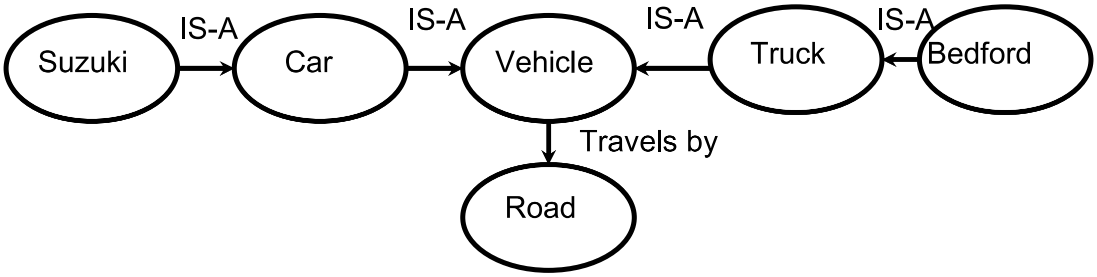


Figure: Vehicle Semantic Network

Network Operation

To infer new information from semantic networks, we can ask questions from nodes

- Ask node vehicle: 'How do you travel?'
  - This node looks at arc and replies: road
- Ask node Suzuki: 'How do you travel?'
  - This node does not have a link to travel therefore it asks other nodes linked by the IS-A link
  - Asks node Car (because of IS-A relationship)
  - Asks node Vehicle (IS-A relationship)
  - Node Vehicle Replies: road

Problems with Semantic Networks

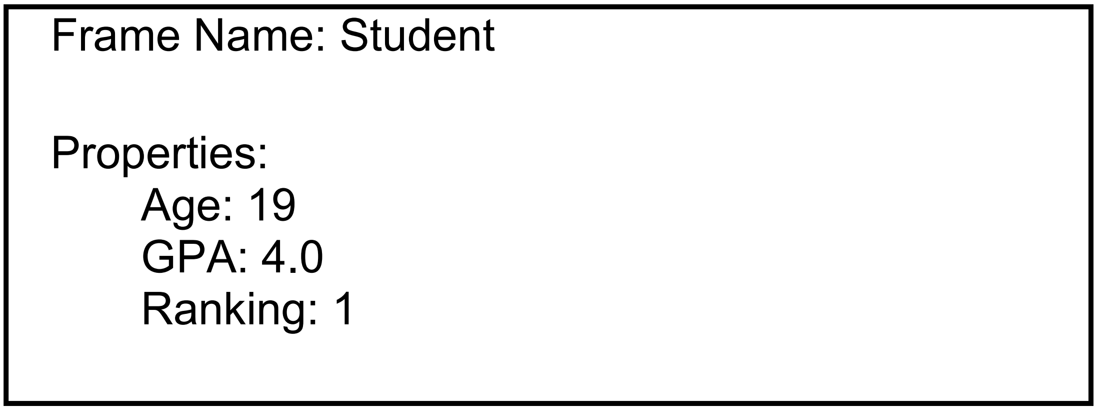
- Semantic networks are computationally expensive at run-time as we need to traverse the network to answer some question. In the worst case, we may need to traverse the entire network and then discover that the requested info does not exist.
- They try to model human associative memory (store information using associations), but in the human brain the number of neurons and links are

in the order of 10^{15}. It is not practical to build such a large semantic network, hence this scheme is not feasible for this type of problems.

- Semantic networks are logically inadequate as they do not have any equivalent quantifiers, e.g., for all, for some, none.

4.9 Frames

“Frames are data structures for representing stereotypical knowledge of some concept or object” according to Durkin, a frame is like a schema, as we would call it in a database design. They were developed from semantic networks and later evolved into our modern-day Classes and Objects. For example, to represent a student, we make use of the following frame:



Frame Name: Student

Properties:
Age: 19
GPA: 4.0
Ranking: 1

Figure: Student Frame

The various components within the frame are called slots, e.g. Frame Name slot.

4.9.1 Facets

A slot in a frame can hold more that just a value, it consists of metadata and procedures also. The various aspects of a slot are called facets. They are a feature of frames that allows us to put constraints on frames. e.g. IF-NEEDED Facets are called when the data of a particular slot is needed. Similarly, IF-CHANGED Facets are when the value of a slot changes.

4.10 Logic

Just like algebra is a type of formal logic that deals with numbers, e.g. 2+4 = 6, propositional logic and predicate calculus are forms of formal logic for dealing with propositions. We will consider two basic logic representation techniques:
–Propositional Logic
–Predicate Calculus

4.10.1 Propositional logic

A proposition is the statement of a fact. We usually assign a symbolic variable to represent a proposition, e.g.

p = It is raining
q = I carry an umbrella

A proposition is a sentence whose truth values may be determined. So, each proposition has a truth value, e.g.
–The proposition ‘A rectangle has four sides’ is true
–The proposition ‘The world is a cube’ is false.

4.10.1.1 Compound statements

Different propositions may be logically related and we can form compound statements of propositions using logical connectives. Common logical connectives are:

∧	AND (Conjunction)
∨	OR (Disjunction)
¬	NOT (Negation)
→	If … then (Conditional)
⇔	If and only if (bi-conditional)

The table below shows the logic of the above connectives

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Figure: Truth Table of Binary Logical Connectives

4.10.1.2 Limitations of propositional logic

- Propositions can only represent knowledge as complete sentences, e.g. a = the ball’s color is blue.
- Cannot analyze the internal structure of the sentence.

- No quantifiers are available, e.g. for-all, there-exists
- Propositional logic provides no framework for proving statements such as:
  All humans are mortal
  All women are humans
  Therefore, all women are mortals

  This is a limitation in its representational power.

4.10.2 Predicate calculus

Predicate Calculus is an extension of propositional logic that allows the structure of facts and sentences to be defined. With predicate logic, we can use expressions like

Color(ball, blue)

This allows the relationship of sub-sentence units to be expressed, e.g. the relationship between color, ball and blue in the above example. Due to its greater representational power, predicate calculus provides a mechanism for proving statements and can be used as a logic system for proving logical theorems.

4.10.2.1 Quantifiers

Predicate calculus allows us to use quantifiers for statements. Quantifiers allow us to say things about some or all objects within some set. The logical quantifiers used in basic predicate calculus are universal and existential quantifiers.

The Universal quantifier

The symbol for the universal quantifier is $\forall$. It is read as “for every” or “for all” and used in formulae to assign the same truth value to all variables in the domain, e.g. in the domain of numbers, we can say that ($\forall$x) (x + x = 2x). In words this is: for every x (where x is a number), x + x = 2x is true. Similarly, in the domain of shapes, we can say that ($\forall$x) (x = square $\rightarrow$ x = polygon), which is read in words as: every square is a polygon. In other words, for every x (where x is a shape), if x is a square, then x is a polygon (it implies that x is a polygon).

Existential quantifier

The symbol for the existential quantifier is $\exists$. It is read as “there exists”, “ for some”, “for at least one”, “there is one”, and is used in formulae to say that something is true for at least one value in the domain, e.g. in the domain of persons, we can say that

($\exists$x) (Person (x) $\wedge$ father (x, Ahmed)). In words this reads as: there exists some person, x who is Ahmed’s father.

4.10.2.2 First order predicate logic

First order predicate logic is the simplest form of predicate logic. The main types of symbols used are

–Constants are used to name specific objects or properties, e.g. Ali, Ayesha, blue, ball.

–Predicates: A fact or proposition is divided into two parts

Predicate: the assertion of the proposition

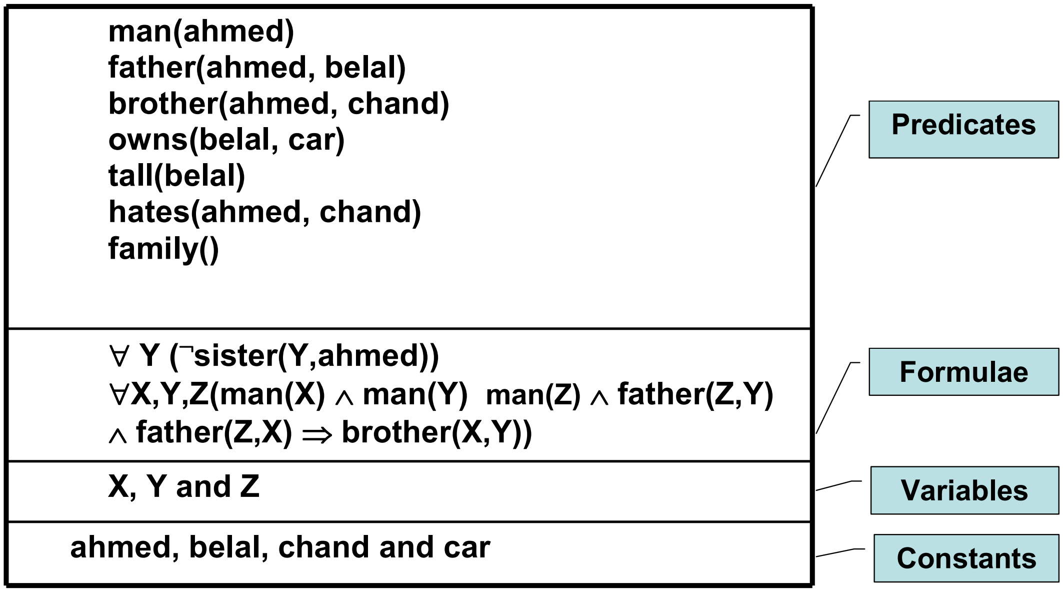
Argument: the object of the proposition

For example, the proposition "Ali likes bananas" can be represented in predicate logic as Likes (Ali, bananas), where Likes is the predicate and Ali and bananas are the arguments.

–Variables: Variables are used to a represent general class of objects/properties, e.g. in the predicate likes (X, Y), X and Y are variables that assume the values X=Ali and Y=bananas

–Formulae: Formulas combine predicates and quantifiers to represent information

Lets us illustrate these symbols using an example



man(ahmed)
father(ahmed, belal)
brother(ahmed, chand)
owns(belal, car)
tall(belal)
hates(ahmed, chand)
family()

$\forall Y (\neg sister(Y,ahmed))$

$\forall X,Y,Z(man(X) \wedge man(Y)\ \ man(Z) \wedge father(Z,Y) \wedge father(Z,X) \Rightarrow brother(X,Y))$

X, Y and Z

ahmed, belal, chand and car

Figure : Predicate Logic Example

The predicate section outlines the known facts about the situation in the form of predicates, i.e. predicate name and its arguments. So, man(ahmed) means that ahmed is a man, hates(ahmed, chand) means that ahmed hates chand.

The formulae sections outlines formulae that use universal quantifiers and variables to define certain rules. $\forall$ Y ($\neg$sister(Y,ahmed)) says that there exists no Y such that Y is the sister of ahmed, i.e. ahmed has no sister. Similarly, $\forall$X,Y,Z(man(X) $\wedge$ man(Y) man(Z) $\wedge$ father(Z,Y) $\wedge$ father(Z,X) $\Rightarrow$ brother(X,Y)) means that if there are three men, X, Y and Z, and Z is the father of both X and Y, then X and Y are bothers. This expresses the rule for the two individuals being brothers.

4.11 Reasoning

lec number 16

Now that we have looked at knowledge representation, we will look at mechanisms to reason on the knowledge once we have represented it using some logical scheme. Reasoning is the process of deriving logical conclusions from given facts. Durkin defines reasoning as 'the process of working with knowledge, facts and problem solving strategies to draw conclusions'.

Throughout this section, you will notice how representing knowledge in a particular way is useful for a particular kind of reasoning.

4.12 Types of reasoning

We will look at some broad categories of reasoning

4.12.1.1 Deductive reasoning

Deductive reasoning, as the name implies, is based on deducing new information from logically related known information. A deductive argument offers assertions that lead automatically to a conclusion, e.g.

–If there is dry wood, oxygen and a spark, there will be a fire

Given: There is dry wood, oxygen and a spark

We can deduce: There will be a fire.

–All men are mortal. Socrates is a man.

We can deduce: Socrates is mortal

4.12.2 Inductive reasoning

Inductive reasoning is based on forming, or inducing a 'generalization' from a limited set of observations, e.g.

–Observation: All the crows that I have seen in my life are black.

–Conclusion: All crows are black

Comparison of deductive and inductive reasoning

We can compare deductive and inductive reasoning using an example. We conclude what will happen when we let a ball go using both each type of reasoning in turn

–The inductive reasoning is as follows: By experience, every time I have let a ball go, it falls downwards. Therefore, I conclude that the next time I let a ball go, it will also come down.
–The deductive reasoning is as follows: I know Newton's Laws. So I conclude that if I let a ball go, it will certainly fall downwards.

Thus the essential difference is that inductive reasoning is based on experience while deductive reasoning is based on rules, hence the latter will always be correct.

4.12.3 Abductive reasoning

Deduction is exact in the sense that deductions follow in a logically provable way from the axioms. Abduction is a form of deduction that allows for plausible inference, i.e. the conclusion might be wrong, e.g.

Implication: She carries an umbrella if it is raining
Axiom: she is carrying an umbrella
Conclusion: It is raining

This conclusion might be false, because there could be other reasons that she is carrying an umbrella, e.g. she might be carrying it to protect herself from the sun.

4.12.4 Analogical reasoning

Analogical reasoning works by drawing analogies between two situations, looking for similarities and differences, e.g. when you say driving a truck is just like driving a car, by analogy you know that there are some similarities in the driving mechanism, but you also know that there are certain other distinct characteristics of each.

4.12.5 Common-sense reasoning

Common-sense reasoning is an informal form of reasoning that uses rules gained through experience or what we call rules-of-thumb. It operates on heuristic knowledge and heuristic rules.

4.12.6 Non-Monotonic reasoning

Non-Monotonic reasoning is used when the facts of the case are likely to change after some time, e.g.

Rule:
IF the wind blows
THEN the curtains sway

When the wind stops blowing, the curtains should sway no longer. However, if we use monotonic reasoning, this would not happen. The fact that the curtains are swaying would be retained even after the wind stopped blowing. In non-monotonic reasoning, we have a ‘truth maintenance system’. It keeps track of what caused a fact to become true. If the cause is removed, that fact is removed (retracted) also.

4.12.7 Inference

Inference is the process of deriving new information from known information. In the domain of AI, the component of the system that performs inference is called an *inference engine*. We will look at inference within the framework of ‘logic’, which we introduced earlier

4.12.7.1 Logic

Logic, which we introduced earlier, can be viewed as a formal language. As a language, it has the following components: syntax, semantics and proof systems.

Syntax

Syntax is a description of valid statements, the expressions that are legal in that language. We have already looked at the syntax of two type of logic systems called propositional logic and predicate logic. The syntax of proposition gives us ways to use propositions, their associated truth value and logical connectives to reason.

Semantics

Semantics pertain to what expressions mean, e.g. the expression ‘the cat drove the car’ is syntactically correct, but semantically non-sensible.

Proof systems

A logic framework comes with a proof system, which is a way of manipulating given statements to arrive at new statements. The idea is to derive ‘new’ information from the given information.

Recall proofs in math class. You write down all you know about the situation and then try to apply all the rules you know repeatedly until you come up with the statement you were supposed to prove. Formally, a proof is a sequence of statements aiming at inferring some information. While doing a proof, you usually proceed with the following steps:

–You begin with initial statements, called premises of the proof (or knowledge base)
–Use rules, i.e. apply rules to the known information
–Add new statements, based on the rules that match

Repeat the above steps until you arrive at the statement you wished to prove.

4.12.7.1.1 Rules of inference

Rules of inference are logical rules that you can use to prove certain things. As you look at the rules of inference, try to figure out and convince yourself that the rules are logically sound, by looking at the associated truth tables. The rules we will use for propositional logic are:

Modus Ponens
Modus Tolens
And-Introduction
And-Elimination

Modus ponens

“Modus ponens" means "affirming method“. Note: From now on in our discussion of logic, anything that is written down in a proof is a statement that is true.

$$\begin{array}{l} \alpha \rightarrow \beta \\ \underline{\alpha \qquad\;} \\ \beta \end{array}$$

Modus-
Ponens

Modus Ponens says that if you know that alpha implies beta, and you know alpha to be true, you can automatically say that beta is true.

Modus Tolens

Modus Tolens says that "alpha implies beta" and "not beta" you can conclude "not alpha". In other words, if Alpha implies beta is true and beta is known to be not true, then alpha could not have been true. Had alpha been true, beta would automatically have been true due to the implication.

$$\begin{array}{l} \alpha \rightarrow \beta \\ \underline{\neg\beta \qquad} \\ \neg\alpha \end{array}$$

Modus -
Tolens

And-Introduction and and-Elimination

And-introduction say that from "Alpha" and from "Beta" you can conclude "Alpha and Beta". That seems pretty obvious, but is a useful tool to know upfront. Conversely, and-elimination says that from "Alpha and Beta" you can conclude "Alpha".

$\begin{array}{l}\alpha \\ \underline{\beta} \\ \alpha \wedge \beta\end{array}$	$\dfrac{\alpha \wedge \beta}{\alpha}$
And-Introduction	And-elimination

The table below gives the four rules of inference together:

$\begin{array}{l}\alpha \rightarrow \beta \\ \underline{\alpha \qquad} \\ \beta\end{array}$	$\begin{array}{l}\alpha \rightarrow \beta \\ \underline{\neg\beta \qquad} \\ \neg\alpha\end{array}$	$\begin{array}{l}\alpha \\ \underline{\beta} \\ \alpha \wedge \beta\end{array}$	$\dfrac{\alpha \wedge \beta}{\alpha}$
Modus Ponens	Modus Tolens	And-Introduction	And-elimination

Figure : Table of Rules of Inference

4.12.7.2 Inference example

Now, we will do an example using the above rules. Steps 1, 2 and 3 are added initially, they are the given facts. The goal is to prove D. Steps 4-8 use the rules of inference to reach at the required goal from the given rules.

Step	Formula	Derivation
1	A ∧ B	Given
2	A→C	Given
3	(B ∧ C) →D	Given
4	A	1 And-elimination
5	C	4, 2 Modus Ponens
6	B	1 And-elimination
7	B ∧C	5, 6 And-introduction
8	D	7, 3 Modus Ponens

Note: The numbers in the derivation reference the statements of other step numbers.

4.12.7.3 Resolution rule 17 lec

The deduction mechanism we discussed above, using the four rules of inference may be used in practical systems, but is not feasible. It uses a lot of inference rules that introduce a large branch factor in the search for a proof. An alternative is approach is called resolution, a strategy used to determine the truth of an assertion, using only one resolution rule:

$$\begin{array}{l} \alpha \vee \beta \\ \underline{\neg\beta \vee \gamma} \\ \alpha \vee \gamma \end{array}$$

To see how this rule is logically correct, look at the table below:

A	β	Γ	$\neg\beta$	$\alpha \vee \beta$	$\neg\beta \vee \gamma$	$\alpha \vee \gamma$
F	F	F	T	F	T	F
F	F	T	T	F	T	T
F	T	F	F	T	F	F
F	T	T	F	T	T	T
T	F	F	T	T	T	T
T	F	T	T	T	T	T
T	T	F	F	T	F	T
T	T	T	F	T	T	T

You can see that the rows where the premises of the rule are true, the conclusion of the rule is true also.

To be able to use the resolution rule for proofs, the first step is to convert all given statements into the conjunctive normal form.

4.12.7.4 Conjunctive normal form

Resolution requires all sentences to be converted into a special form called conjunctive normal form (CNF). A statement in conjunctive normal form (CNF) consists of ANDs of Ors. A sentence written in CNF looks like

$$\begin{array}{l} (A \vee B) \wedge (B \vee \neg C) \wedge (D) \\ note : D = (D \vee \neg D) \end{array}$$

The outermost structure is made up of conjunctions. Inner units called clauses are made up of disjunctions. The components of a statement in CNF are clauses and literals. A clause is the disjunction of many units. The units that make up a clause are called literals. And a literal is either a variable or the negation of a variable. So you get an expression where the negations are pushed in as tightly as possible, then you have ORs, then you have ANDs. You can think of each clause as a requirement. Each clause has to be satisfied individually to satisfy the entire statement.

4.12.7.5 Conversion to CNF

1. Eliminate arrows (implications)

$$A \rightarrow B = \neg A \vee B$$

2. Drive in negations using De Morgan's Laws, which are given below

$$\neg(A \vee B) = (\neg A \wedge \neg B)$$

$$\neg(A \wedge B) = (\neg A \vee \neg B)$$

3. Distribute OR over AND

$$A \vee (B \wedge C)$$
$$= (A \vee B) \wedge (A \vee C)$$

4.12.7.6 Example of CNF conversion

$$(A \vee B) \rightarrow (C \rightarrow D)$$

$$1. \neg(A \vee B) \vee (\neg C \vee D)$$

$$2. (\neg A \wedge \neg B) \vee (\neg C \vee D)$$

$$3. (\neg A \vee \neg C \vee D) \wedge (\neg B \vee \neg C \vee D)$$

4.12.7.7 Resolution by Refutation

Now, we will look at a proof strategy called resolution refutation. The steps for proving a statement using resolution refutation are:

- Write all sentences in CNF
- Negate the desired conclusion
- Apply the resolution rule until you derive a contradiction or cannot apply the rule anymore.
- If we derive a contradiction, then the conclusion follows from the given axioms
- If we cannot apply anymore, then the conclusion cannot be proved from the given axioms

4.12.8 Resolution refutation example 1

Ste	Formula	Derivation
1	$A \lor B$	Given
2	$\neg A \lor C$	Given
3	$\neg B \lor C$	Given
4	$\neg C$	Negated Conclusion
5	$B \lor C$	1, 2
6	$\neg A$	2, 4
7	$\neg B$	3, 4
8	C	5, 7 Contradiction!

Prove C

1	$A \lor B$
2	$A \rightarrow C$
3	$B \rightarrow C$

The statements in the table on the right are the given statements. These are converted to CNF and are included as steps 1, 2 and 3. Our goal is to prove C. Step 4 is the addition of the negation of the desired conclusion. Steps 5-8 use the resolution rule to prove C.

Note that you could have come up with multiple ways of proving R:

Step	Formula	
1	$A \lor B$	Given
2	$\neg A \lor C$	Given
3	$\neg B \lor C$	Given
4	$\neg C$	
5	$\neg B$	3,4
6	A	1,5
7	**C**	2,6

Step	Formula	
1	$A \lor B$	Given
2	$\neg A \lor C$	Given
3	$\neg B \lor C$	Given
4	$\neg C$	
5	$B \lor C$	1,2
6	$\neg A$	2,4
7	$\neg B$	3,4
8	**C**	5,7

4.12.9 Resolution Refutation Example 2

1. (A→B) →B
2. A→C
3. ¬C → ¬B
Prove C

Convert to CNF

$$1.(A \rightarrow B) \rightarrow B$$
$$= (\neg A \vee B) \rightarrow B$$
$$= \neg(\neg A \vee B) \vee B$$
$$= (A \wedge \neg B) \vee B$$
$$= (A \vee B) \wedge (\neg B \vee B)$$
$$= (A \vee B)$$
$$2.A \rightarrow C = \neg A \vee C$$
$$3.\neg C \rightarrow \neg B = C \vee \neg B$$

Proof

Step	Formula	Derivation	Step	Formula	Derivation
1	B ∨ A	Given	1	B ∨ A	Given
2	¬ A ∨ C	Given	2	¬ A ∨ C	Given
3	C ∨ ¬B	Given	3	C ∨ ¬B	Given
4	¬C	Negation of conclusion	4	¬C	Negation of conclusion
5	A	2,4	5	¬B	3,4
6	C	2,5	6	A	1,5
			7	C	2,6

4.12.9.1 Proof strategies

As you can see from the examples above, it is often possible to apply more than one rule at a particular step. We can use several strategies in such cases. We may apply rules in an arbitrary order, but there are some rules of thumb that may make the search more efficient

- Unit preference: prefer using a clause with one literal. Produces shorter clauses
- Set of support: Try to involve the thing you are trying to prove. Chose a resolution involving the negated goal. These are relevant clauses. We move 'towards solution'

Lecture No. 18-28

5 Expert Systems

Expert Systems (ES) are a popular and useful application area in AI. Having studied KRR, it is instructive to study ES to see a practical manifestation of the principles learnt there.

5.1 What is an Expert?

Before we attempt to define an expert system, we have look at what we take the term 'expert' to mean when we refer to human experts. Some traits that characterize experts are:

- They possess specialized knowledge in a certain area
- They possess experience in the given area
- They can provide, upon elicitation, an explanation of their decisions
- The have a skill set that enables them to translate the specialized knowledge gained through experience into solutions.

Try to think of the various traits you associate with experts you might know, e.g. skin specialist, heart specialist, car mechanic, architect, software designer. You will see that the underlying common factors are similar to those outlined above.

5.2 What is an expert system?

According to Durkin, an expert system is "A computer program designed to model the problem solving ability of a human expert". With the above discussion of experts in mind, the aspects of human experts that expert systems model are the experts:

- Knowledge
- Reasoning

5.3 History and Evolution

Before we begin to study development of expert systems, let us get some historical perspective about the earliest practical AI systems. After the so-called dark ages in AI, expert systems were at the forefront of rebirth of AI. There was a realization in the late 60's that the general framework of problem solving was not enough to solve all kinds of problem. This was augmented by the realization that specialized knowledge is a very important component of practical systems. People observed that systems that were designed for well-focused problems and domains out performed more ‘general’ systems. These observations provided the motivation for expert systems. Expert systems are important historically as the earliest AI systems and the most used systems practically. To highlight the utility of expert systems, we will look at some famous expert systems, which served to define the paradigms for the current expert systems.

5.3.1 Dendral (1960’s)

Dendral was one of the pioneering expert systems. It was developed at Stanford for NASA to perform chemical analysis of Martian soil for space missions. Given mass spectral data, the problem was to determine molecular structure. In the laboratory, the ‘generate and test’ method was used; possible hypothesis about molecular structures were generated and tested by matching to actual data. There was an early realization that experts use certain heuristics to rule out certain options when looking at possible structures. It seemed like a good idea to encode that knowledge in a software system. The result was the program Dendral, which gained a lot of acclaim and most importantly provided the important distinction that Durkin describes as: ‘Intelligent behavior is dependent, not so much on the methods of reasoning, but on the knowledge one has to reason with’.

5.3.2 MYCIN (mid 70s)

MYCIN was developed at Stanford to aid physicians in diagnosing and treating patients with a particular blood disease. The motivation for building MYCIN was that there were few experts of that disease, they also had availability constraints. Immediate expertise was often needed because they were dealing with a life-threatening condition. MYCIN was tested in 1982. Its diagnosis on ten selected cases was obtained, along with the diagnosis of a panel of human experts. MYCIN compositely scored higher than human experts!

MYCIN was an important system in the history of AI because it demonstrated that expert systems could be used for solving practical problems. It was pioneering work on the structure of ES (separate knowledge and control), as opposed to Dendral, MYCIN used the same structure that is now formalized for expert systems.

5.3.3 R1/XCON (late 70’s)

R1/XCON is also amongst the most cited expert systems. It was developed by DEC (Digital Equipment Corporation), as a computer configuration assistant. It was one of the most successful expert systems in routine use, bringing an estimated saving of $25million per year to DEC. It is a classical example of how an ES can increase productivity of organization, by assisting existing experts.

5.4 Comparison of a human expert and an expert system

The following table compares human experts to expert systems. While looking at these, consider some examples, e.g. doctor, weather expert.

Issues	**Human Expert**	**Expert System**
Availability	Limited	Always
Geographic location	Locally available	Anywhere
Safety considerations	Irreplaceable	Can be replaced
Durability	Depends on individual	Non-perishable
Performance	Variable	High
Speed	Variable	High
Cost	High	Low
Learning Ability	Variable/High	Low
Explanation	Variable	Exact

5.5 Roles of an expert system

An expert system may take two main roles, relative to the human expert. It may replace the expert or assist the expert

Replacement of expert

This proposition raises many eyebrows. It is not very practical in some situations, but feasible in others. Consider drastic situations where safety or location is an issue, e.g. a mission to Mars. In such cases replacement of an expert may be the only feasible option. Also, in cases where an expert cannot be available at a particular geographical location e.g. volcanic areas, it is expedient to use an expert system as a substitute.

An example of this role is a France based oil exploration company that maintains a number of oil wells. They had a problem that the drills would occasionally become stuck. This typically occurs when the drill hits something that prevents it from turning. Often delays due to this problem cause huge losses until an expert can arrive at the scene to investigate. The company decided to deploy an expert system so solve the problem. A system called ‘Drilling Advisor’ (Elf-Aquitane 1983) was developed, which saved the company from huge losses that would be incurred otherwise.

Assisting expert

Assisting an expert is the most commonly found role of an ES. The goal is to aid an expert in a routine tasks to increase productivity, or to aid in managing a complex situation by using an expert system that may itself draw on experience of other (possibly more than one) individuals. Such an expert system helps an expert overcome shortcomings such as recalling relevant information.
XCON is an example of how an ES can assist an expert.

5.6 How are expert systems used?

Expert systems may be used in a host of application areas including diagnosis, interpretation, prescription, design, planning, control, instruction, prediction and simulation.

Control applications

In control applications, ES are used to adaptively govern/regulate the behavior of a system, e.g. controlling a manufacturing process, or medical treatment. The ES obtains data about current system state, reasons, predicts future system states and recommends (or executes) adjustments accordingly. An example of such a system is VM (Fagan 1978). This ES is used to monitor patient status in the intensive care unit. It analyses heart rate, blood pressure and breathing measurements to adjust the ventilator being used by the patient.

Design

ES are used for design applications to configure objects under given design constraints, e.g. XCON. Such ES often use non-monotonic reasoning, because of implications of steps on previous steps. Another example of a design ES is PEACE (Dincbas 1980), which is a CAD tool to assist in design of electronic structures.

Diagnosis and Prescription

An ES can serve to identify system malfunction points. To do this it must have knowledge of possible faults as well as diagnosis methodology extracted from technical experts, e.g. diagnosis based on patient’s symptoms, diagnosing malfunctioning electronic structures. Most diagnosis ES have a prescription subsystem. Such systems are usually interactive, building on user information to narrow down diagnosis.

Instruction and Simulation

ES may be used to guide the instruction of a student in some topic. Tutoring applications include GUIDON (Clancey 1979), which instructs students in diagnosis of bacterial infections. Its strategy is to present user with cases (of which it has solution). It then analyzes the student’s response. It compares the students approach to its own and directs student based on differences.

Simulation

ES can be used to model processes or systems for operational study, or for use along with tutoring applications

Interpretation

According to Durkin, interpretation is 'Producing an understanding of situation from given information'. An example of a system that provides interpretation is FXAA (1988). This ES provides financial assistance for a commercial bank. It looks at a large number of transactions and identifies irregularities in transaction trends. It also enables automated audit.

Planning and prediction

ES may be used for planning applications, e.g. recommending steps for a robot to carry out certain steps, cash management planning. SMARTPlan is such a system, a strategic market planning expert (Beeral, 1993). It suggests appropriate marketing strategy required to achieve economic success. Similarly, prediction systems infer likely consequences from a given situation.

Appropriate domains for expert systems

When analyzing a particular domain to see if an expert system may be useful, the system analyst should ask the following questions:

- Can the problem be effectively solved by conventional programming? If not, an ES may be the choice, because ES are especially suited to ill-structured problems.
- Is the domain well-bounded? e.g. a headache diagnosis system may eventually have to contain domain knowledge of many areas of medicine because it is not easy to limit diagnosis to one area. In such cases where the domain is too wide, building an ES may be not be a feasible proposition.
- What are the practical issues involved? Is some human expert willing to cooperate? Is the expert's knowledge especially uncertain and heuristic? If so, ES may be useful.

5.7 Expert system structure lec 19

Having discussed the scenarios and applications in which expert systems may be useful, let us delve into the structure of expert systems. To facilitate this, we use the analogy of an expert (say a doctor) solving a problem. The expert has the following:

- Focused area of expertise
- Specialized Knowledge (Long-term Memory, LTM)
- Case facts (Short-term Memory, STM)
- Reasons with these to form new knowledge

- Solves the given problem

Now, we are ready to define the corresponding concepts in an Expert System.

Human Expert	**Expert System**
Focused Area of Expertise	**Domain**
Specialized Knowledge (stored in LTM)	**Domain Knowledge** (stored in **Knowledge Base**)
Case Facts (stored in STM)	Case/Inferred Facts (stored in **Working Memory**)
Reasoning	**Inference Engine**
Solution	**Conclusions**

We can view the structure of the ES and its components as shown in the figure below

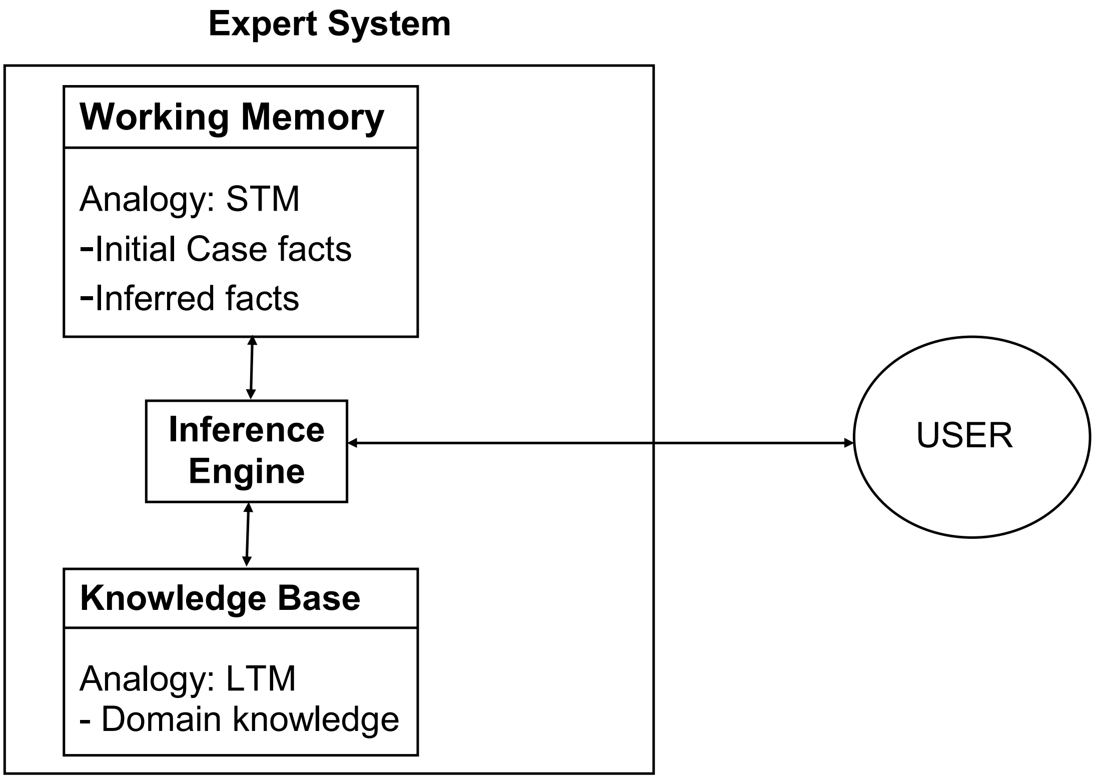


Figure 13: Expert System Structure

5.7.1 Knowledge Base

The knowledge base is the part of an expert system that contains the domain knowledge, i.e.

- Problem facts, rules
- Concepts
- Relationships

As we have emphasised several times, the power of an ES lies to a large extent in its richness of knowledge. Therefore, one of the prime roles of the expert system designer is to act as a knowledge engineer. As a knowledge engineer, the designer must overcome the knowledge acquisition bottleneck and find an effective way to get information from the expert and encode it in the knowledge base, using one of the knowledge representation techniques we discussed in KRR.

As discussed in the KRR section, one way of encoding that knowledge is in the form of IF-THEN rules. We saw that such representation is especially conducive to reasoning.

5.7.2 Working memory

The working memory is the 'part of the expert system that contains the problem facts that are discovered during the session' according to Durkin. One session in the working memory corresponds to one consultation. During a consultation:

- User presents some facts about the situation.
- These are stored in the working memory.
- Using these and the knowledge stored in the knowledge base, new information is inferred and also added to the working memory.

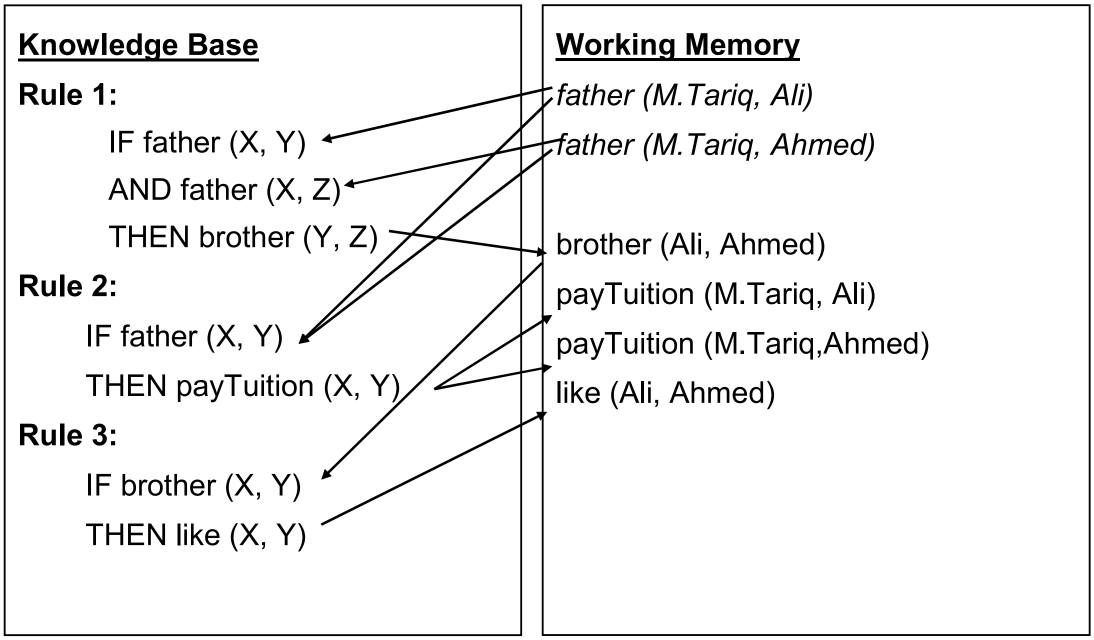
5.7.3 Inference Engine

The inference engine can be viewed as the processor in an expert system that matches the facts contained in the working memory with the domain knowledge contained in the knowledge base, to draw conclusions about the problem. It works with the knowledge base and the working memory, and draws on both to add new facts to the working memory.

If the knowledge of an ES is represented in the form of IF-THEN rules, the Inference Engine has the following strategy: Match given facts in working memory to the premises of the rules in the knowledge base, if match found, 'fire' the conclusion of the rule, i.e. add the conclusion to the working memory. Do this repeatedly, while new facts can be added, until you come up with the desired conclusion.

We will illustrate the above features using examples in the following sections

5.7.4 Expert System Example: Family



Knowledge Base

Rule 1:

IF father (X, Y)
AND father (X, Z)
THEN brother (Y, Z)

Rule 2:

IF father (X, Y)
THEN payTuition (X, Y)

Rule 3:

IF brother (X, Y)
THEN like (X, Y)

Working Memory

father (M.Tariq, Ali)
father (M.Tariq, Ahmed)

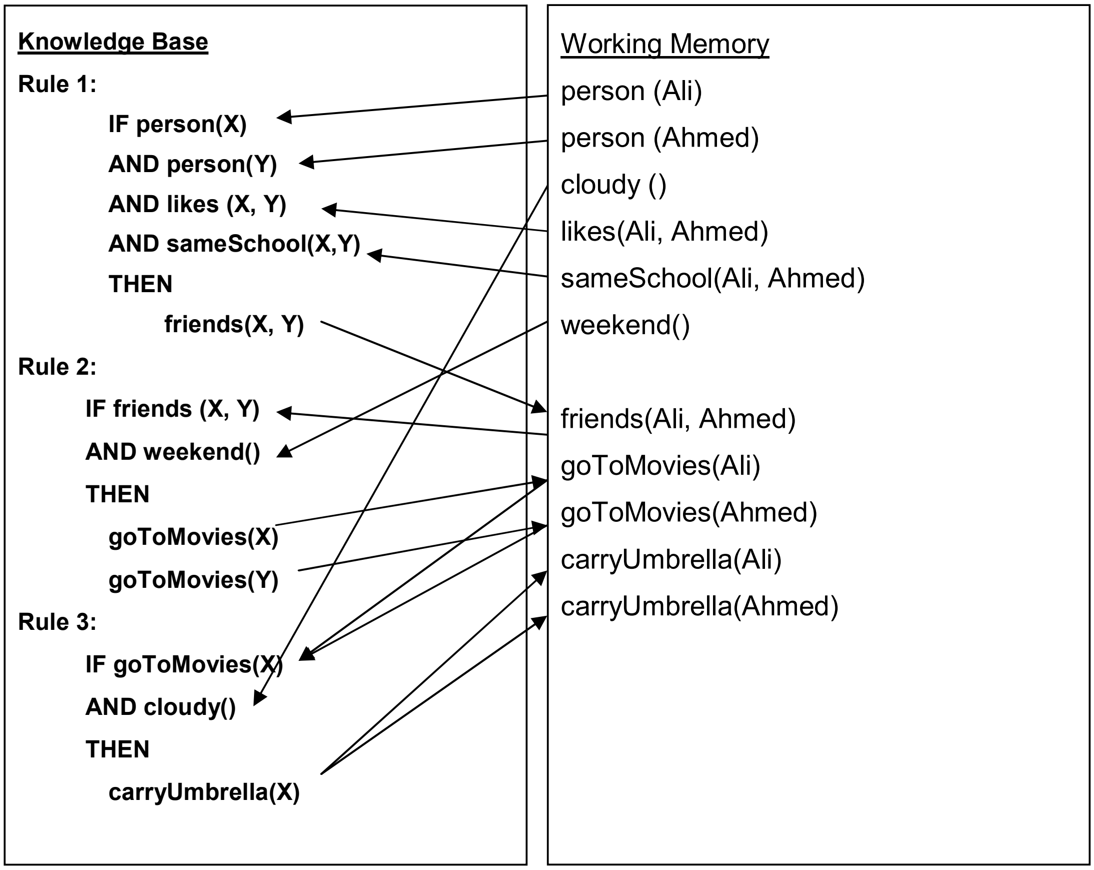
brother (Ali, Ahmed)
payTuition (M.Tariq, Ali)
payTuition (M.Tariq,Ahmed)
like (Ali, Ahmed)

Let's look at the example above to see how the knowledge base and working memory are used by the inference engine to add new facts to the working memory. The knowledge base column on the left contains the three rules of the system. The working memory starts out with two initial case facts:

father (M.Tariq, Ali)
father (M.Tariq, Ahmed)

The inference engine matches each rule in turn with the rules in the working memory to see if the premises are all matched. Once all premises are matched, the rule is fired and the conclusion is added to the working memory, e.g. the premises of rule 1 match the initial facts, therefore it fires and the fact brother(Ali, Ahmed is fired). This matching of rule premises and facts continues until no new facts can be added to the system. The matching and firing is indicated by arrows in the above table.

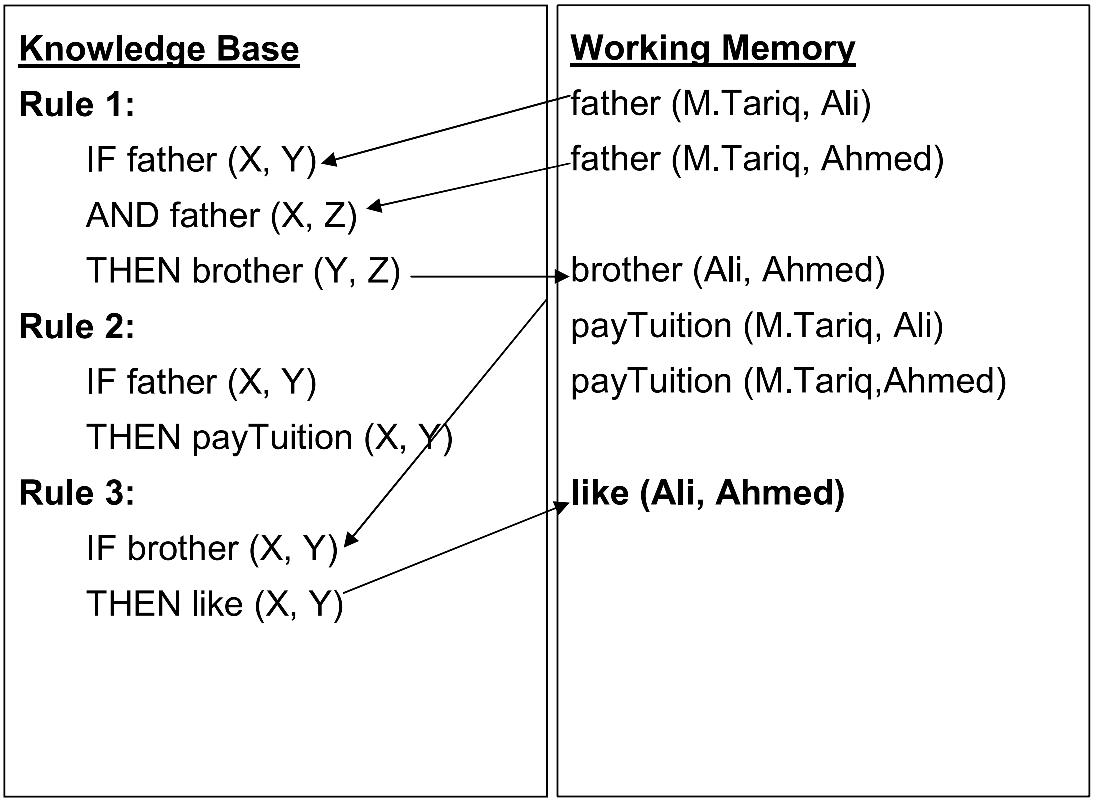
5.7.5 Expert system example: raining



5.7.6 Explanation facility

The explanation facility is a module of an expert system that allows transparency of operation, by providing an explanation of how it reached the conclusion. In the family example above, how does the expert system draw the conclusion that Ali likes Ahmed?

The answer to this is the sequence of reasoning steps as shown with the arrows in the table below.



Knowledge Base

Rule 1:

IF father (X, Y)

AND father (X, Z)

THEN brother (Y, Z)

Rule 2:

IF father (X, Y)

THEN payTuition (X, Y)

Rule 3:

IF brother (X, Y)

THEN like (X, Y)

Working Memory

father (M.Tariq, Ali)

father (M.Tariq, Ahmed)

brother (Ali, Ahmed)

payTuition (M.Tariq, Ali)

payTuition (M.Tariq,Ahmed)

like (Ali, Ahmed)

The arrows above provide the explanation for how the fact like(Ali, Ahmed) was added to the working memory.

5.8 Characteristics of expert systems

Having looked at the basic operation of expert systems, we can begin to outline desirable properties or characteristics we would like our expert systems to possess.

ES have an explanation facility. This is the module of an expert system that allows transparency of operation, by providing an explanation of how the inference engine reached the conclusion. We want ES to have this facility so that users can have knowledge of how it reaches its conclusion.

An expert system is different from conventional programs in the sense that program control and knowledge are separate. We can change one while affecting the other minimally. This separation is manifest in ES structure; knowledge base, working memory and inference engine. Separation of these components allows changes to the knowledge to be independent of changes in control and vice versa.

”There is a clear separation of general knowledge about the problem (the rules forming the knowledge base) from information about the current problem (the input data) and methods for applying the general knowledge to a problem (the rule interpreter).The program itself is only an interpreter (or general reasoning mechanism) and ideally the system can be changed simply by adding or subtracting rules in the knowledge base" (Duda)

Besides these properties, an expert system also possesses expert knowledge in that it embodies expertise of human expert. If focuses expertise because the larger the domain, the more complex the expert system becomes, e.g. a car diagnosis expert is more easily handled if we make separate ES components for engine problems, electricity problems, etc. instead of just designing one component for all problems.

We have also seen that an ES reasons heuristically, by encoding an expert's rules-of-thumb. Lastly, an expert system, like a human expert makes mistakes, but that is tolerable if we can get the expert system to perform at least as well as the human expert it is trying to emulate.

5.9 Programming vs. knowledge engineering

Conventional programming is a sequential, three step process: Design, Code, Debug. Knowledge engineering, which is the process of building an expert system, also involves assessment, knowledge acquisition, design, testing, documentation and maintenance. However, there are some key differences between the two programming paradigms.

Conventional programming focuses on solution, while ES programming focuses on problem. An ES is designed on the philosophy that if we have the right knowledge base, the solution will be derived from that data using a generic reasoning mechanism.

Unlike traditional programs, you don't just program an ES and consider it 'built'. It grows as you add new knowledge. Once framework is made, addition of knowledge dictates growth of ES.

5.10 People involved in an expert system project

The main people involved in an ES development project are the domain expert, the knowledge engineer and the end user.

Domain Expert

A domain expert is 'A person who posses the skill and knowledge to solve a specific problem in a manner superior to others' (Durkin). For our purposes, an expert should have expert knowledge in the given domain, good communication skills, availability and readiness to co-operate.

Knowledge Engineer

A knowledge engineer is 'a person who designs, builds and tests an Expert System' (Durkin). A knowledge engineer plays a key role in identifying, acquiring and encoding knowledge.

End-user

The end users are the people who will use the expert system. Correctness, usability and clarity are important ES features for an end user.

5.11 Inference mechanisms

In the examples we have looked at so far, we have looked informally at how the inference engine adds new facts to the working memory. We can see that many different sequences for matching are possible and that we can have multiple strategies for inferring new information, depending upon our goal. If we want to look for a specific fact, it makes no sense to add all possible facts to the working memory. In other cases, we might actually need to know all possible facts about the situation. Guided by this intuition, we have two formal inference mechanisms; forward and backward chaining.

5.11.1 Forward Chaining

Let's look at how a doctor goes about diagnosing a patient. He asks the patient for symptoms and then infers diagnosis from symptoms. Forward chaining is based on the same idea. It is an "inference strategy that begins with a set of known facts, derives new facts using rules whose premises match the known facts, and continues this process until a goal sate is reached or until no further rules have premises that match the known or derived facts" (Durkin). As you will come to appreciate shortly, it is a data-driven approach.

Approach

1. Add facts to working memory (WM)
2. Take each rule in turn and check to see if any of its premises match the facts in the WM
3. When matches found for all premises of a rule, place the conclusion of the rule in WM.
4. Repeat this process until no more facts can be added. Each repetition of the process is called a pass.

We will demonstrate forward chaining using an example.

Doctor example (forward chaining)

Rules

Rule 1
IF The patient has deep cough
AND We suspect an infection
THEN The patient has Pneumonia

Rule 2
IF The patient's temperature is above 100
THEN Patient has fever

Rule 3	
IF	The patient has been sick for over a fortnight
AND	The patient has a fever
THEN	We suspect an infection

Case facts

- Patients temperature= 103
- Patient has been sick for over a month
- Patient has violent coughing fits

First Pass

Rule, premise	Status	Working Memory
1, 1 Deep cough	True	Temp= 103 Sick for a month Coughing fits
1, 2 Suspect infection	Unknown	Temp= 103 Sick for a month Coughing fits
2, 1 Temperature>100	True, fire rule	Temp= 103 Sick for a month Coughing fits Patient has fever

Second Pass

Rule, premise	Status	Working Memory
1, 1 Deep cough	True	Temp= 103 Sick for a month Coughing fits Patient has fever
1, 2 Suspect infection	Unknown	Temp= 103 Sick for a month Coughing fits Patient has fever
3, 1 Sick for over fortnight	True	Temp= 103 Sick for a month Coughing fits Patient has fever
3, 2 Patient has fever	True, fire	Temp= 103 Sick for a month Coughing fits Patient has fever Infection

Third Pass

Rule, **premise**	**Status**	**Working Memory**
1, 1 Deep cough	True	Temp= 103 Sick for a month Coughing fits Patient has fever Infection
1, 2 Suspect infection	True, fire	Temp= 103 Sick for a month Coughing fits Patient has fever Infection **Pneumonia**

Now, no more facts can be added to the WM. Diagnosis: Patient has Pneumonia.

Issues in forward chaining

Undirected search

There is an important observation to be made about forward chaining. The forward chaining inference engine infers all possible facts from the given facts. It has no way of distinguishing between important and unimportant facts. Therefore, equal time spent on trivial evidence as well as crucial facts. This is draw back of this approach and we will see in the coming section how to overcome this.

Conflict resolution

Another important issue is *conflict resolution.* This is the question of what to do when the premises of two rules match the given facts. Which should be fired first? If we fire both, they may add conflicting facts, e.g.

IF you are bored
    AND you have no cash
    THEN go to a friend’s place
IF you are bored
    AND you have a credit card
    THEN go watch a movie

If both rules are fired, you will add conflicting recommendations to the working memory.

Conflict resolution strategies

To overcome the conflict problem stated above, we may choose to use on of the following conflict resolution strategies:

- Fire first rule in sequence (rule ordering in list). Using this strategy all the rules in the list are ordered (the ordering imposes prioritization). When more than one rule matches, we simply fire the first in the sequence

- Assign rule priorities (rule ordering by importance). Using this approach we assign explicit priorities to rules to allow conflict resolution.

- More specific rules (more premises) are preferred over general rules. This strategy is based on the observation that a rule with more premises, in a sense, more evidence or votes from its premises, therefore it should be fired in preference to a rule that has less premises.

- Prefer rules whose premises were added more recently to WM (time-stamping). This allows prioritizing recently added facts over older facts.

- Parallel Strategy (view-points). Using this strategy, we do not actually resolve the conflict by selecting one rule to fire. Instead, we branch out our execution into a tree, with each branch operation in parallel on multiple threads of reasoning. This allows us to maintain multiple view-points on the argument concurrently

5.11.2 Backward chaining lec 21

Backward chaining is an inference strategy that works backward from a hypothesis to a proof. You begin with a hypothesis about what the situation might be. Then you prove it using given facts, e.g. a doctor may suspect some disease and proceed by inspection of symptoms. In backward chaining terminology, the hypothesis to prove is called the goal.

Approach

1. Start with the goal.
2. Goal may be in WM initially, so check and you are done if found!
3. If not, then search for goal in the THEN part of the rules (match conclusions, rather than premises). This type of rule is called *goal rule*.
4. Check to see if the goal rule's premises are listed in the working memory.
5. Premises not listed become **sub-goals** to prove.
6. Process continues in a **recursive** fashion until a premise is found that is not supported by a rule, i.e. a premise is called a **primitive**, if it cannot be concluded by any rule
7. When a primitive is found, ask user for information about it. Back track and use this information to prove sub-goals and subsequently the goal.

As you look at the example for backward chaining below, notice how the approach of backward chaining is like depth first search.

Backward chaining example

Consider the same example of doctor and patient that we looked at previously

Rules

Rule 1
IF The patient has deep cough
AND We suspect an infection
THEN The patient has Pneumonia

Rule 2
IF The patient’s temperature is above 100
THEN Patient has fever

Rule 3
IF The patient has been sick for over a fortnight
AND The patient has fever
THEN We suspect an infection

Goal

Patient has Pneumonia

Step	Description	Working Memory
1	Goal: Patient has pneumonia. Not in working memory	
2	Find rules with goal in conclusion: Rule 1	
3	See if rule 1, premise 1 is known, “the patient has a deep cough	
4	Find rules with this statement in conclusion. No rule found. “The patient has a deep cough” is a primitive. Prompt patient. Response: Yes.	Deep cough
5	See if rule 1, premise 2 is known, “We suspect an infection”	Deep cough
6	This is in conclusion of rule 3. See if rule 3, premise 1 is known, “The patient has been sick for over a fortnight”	Deep cough
7	This is a primitive. Prompt patient. Response: Yes	Deep cough Sick over a month
8	See if rule 3, premise 2 is known, “The patient has a fever”	Deep cough Sick over a month
9	This is conclusion of rule 2. See if rule 2, premise 1 is known, “Then patients temperature is above 100”	Deep cough Sick over a month

10	This is a primitive. Prompt patient. Response: Yes. Fire Rule	Deep cough Sick over a month Fever
11	Rule 3 fires	Deep cough Sick over a month Fever Infection
12	Rule 1 fires	Deep cough Sick over a month Fever Infection **Pneumonia**

5.11.3 Forward vs. backward chaining

The exploration of knowledge has different mechanisms in forward and backward chaining. Backward chaining is more focused and tries to avoid exploring unnecessary paths of reasoning. Forward chaining, on the other hand is like an exhaustive search.

In the figures below, each node represents a statement. Forward chaining starts with several facts in the working memory. It uses rules to generate more facts. In the end, several facts have been added, amongst which one or more may be relevant. Backward chaining however, starts with the goal state and tries to reach down to all primitive nodes (marked by '?'), where information is sought from the user.

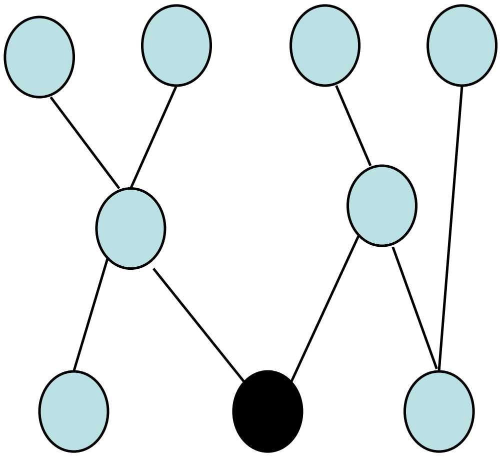


Figure: Forward chaining

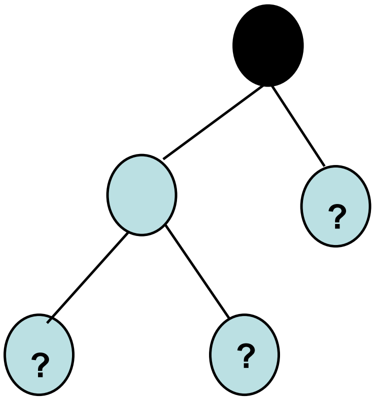


Figure: Backward Chaining

5.12 Design of expert systems

We will now look at software engineering methodology for developing practical ES. The general stages of the expert system development lifecycle or ESDLC are

- Feasibility study
- Rapid prototyping
- Alpha system (in-house verification)
- Beta system (tested by users)
- Maintenance and evolution

Linear model

The Linear model (Bochsler 88) of software development has been successfully used in developing expert systems. A linear sequence of steps is applied repeatedly in an iterative fashion to develop the ES. The main phases of the linear sequence are

- Planning
- Knowledge acquisition and analysis
- Knowledge design
- Code
- Knowledge verification
- System evaluation

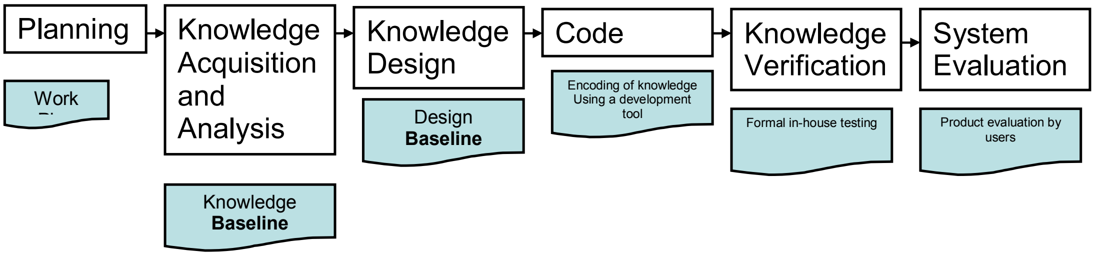


Figure: Linear Model for ES development

5.12.1 Planning phase

This phase involves the following steps

- Feasibility assessment
- Resource allocation
- Task phasing and scheduling
- Requirements analysis

5.12.2 Knowledge acquisition

This is the most important stage in the development of ES. During this stage the knowledge engineer works with the domain expert to acquire, organize and analyze the domain knowledge for the ES. 'Knowledge acquisition is the bottleneck in the construction of expert systems' (Hayes-Roth et al.). The main steps in this phase are

- Knowledge acquisition from expert
- Define knowledge acquisition strategy (consider various options)
- Identify concrete knowledge elements
- Group and classify knowledge. Develop hierarchical representation where possible.
- Identify knowledge source, i.e. expert in the domain
    - Identify potential sources (human expert, expert handbooks/ manuals), e.g. car mechanic expert system's knowledge engineer may chose a mix of interviewing an expert mechanic and using a mechanics trouble-shooting manual.
      Tip: Limit the number of knowledge sources (experts) for simple domains to avoid scheduling and view conflicts. However, a single expert approach may only be applicable to restricted small domains.
    - Rank by importance
    - Rank by availability
    - Select expert/panel of experts
    - If more than one expert has to be consulted, consider a blackboard system, where more than one knowledge source (kept partitioned), interact through an interface called a Blackboard

5.12.3 Knowledge acquisition techniques

- Knowledge elicitation by interview
- Brainstorming session with one or more experts. Try to introduce some structure to this session by defining the problem at hand, prompting for ideas and looking for converging lines of thought.
- Electronic brainstorming
- On-site observation
- Documented organizational expertise, e.g. troubleshooting manuals

5.12.4 Knowledge elicitation

Getting knowledge from the expert is called *knowledge elicitation vs.* the broader term *knowledge acquisition.* Elicitation methods may be broadly divided into:

- Direct Methods
    - Interviews
        - Very good at initial stages
        - Reach a balance between structured (multiple choice, rating scale) and un-structured interviewing.
        - Record interviews (transcribe or tape)
        - Mix of open and close ended questions

  - Informal discussions (gently control digression, but do not offend expert by frequent interruption)
- Indirect methods
  - Questionnaire

Problems that may be faced and have to be overcome during elicitation include

- Expert may not be able to effectively articulate his/her knowledge.
- Expert may not provide relevant information.
- Expert may provide incomplete knowledge
- Expert may provide inconsistent or incorrect knowledge

5.12.5 Knowledge analysis

The goal of knowledge analysis is to analyze and structure the knowledge gained during the knowledge acquisition phase. The key steps to be followed during this stage are

- Identify specific knowledge elements, at the level of concepts, entities, etc.
- From the notes taken during the interview sessions, extract specific
  - Identify strategies (as a list of points)
  - Translate strategies to rules
  - Identify heuristics
  - Identify concepts
  - Represent concepts and their relationships using some visual mechanism like cognitive maps

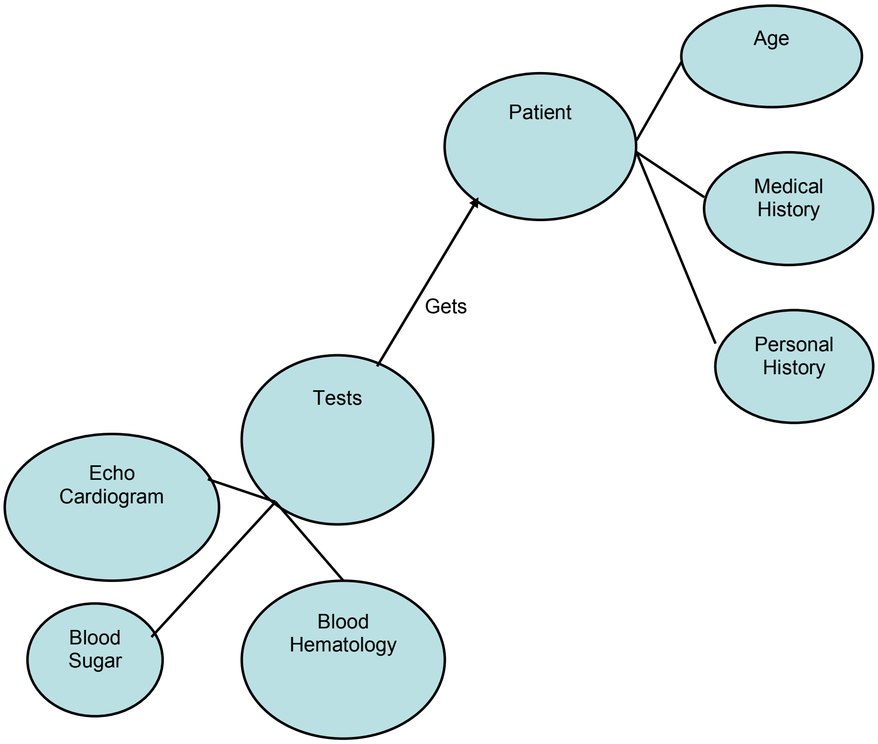


Figure: Cognitive Map example

The example cognitive map for the domain of medicine shows entities and their relationships. Concepts and sub-concepts are identified and grouped together to understand the structure of the knowledge better. Cognitive maps are usually used to represent static entities.

Inference networks

Inference networks encode the knowledge of rules and strategies.

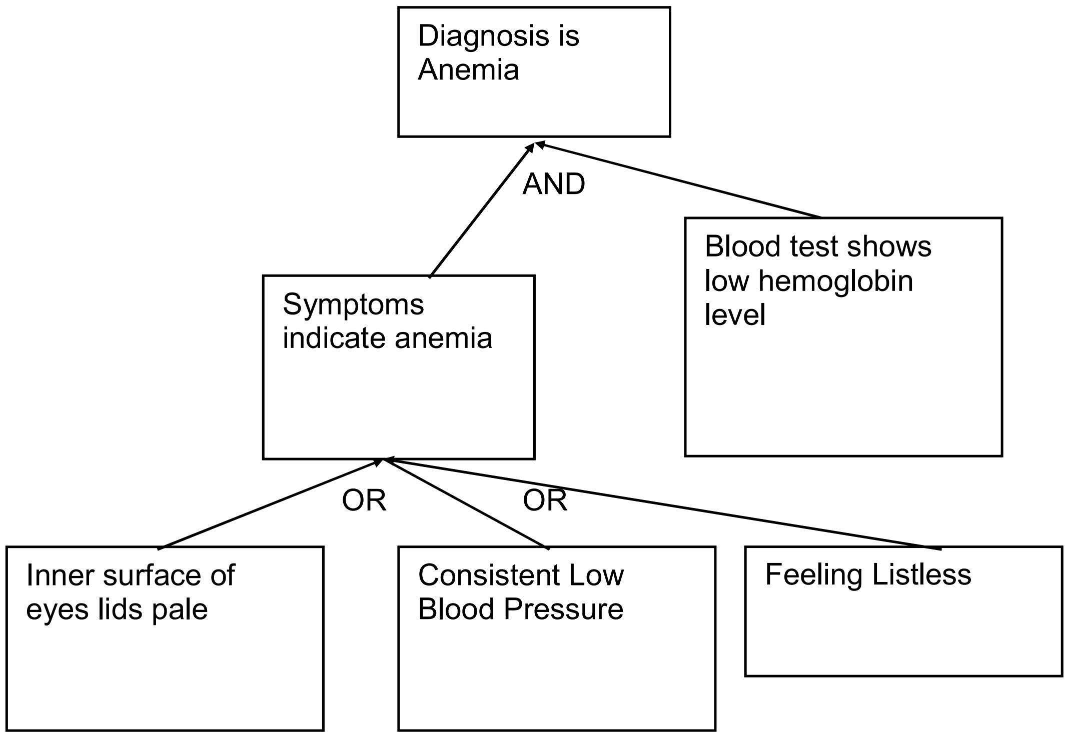

Diagnosis is Anemia
AND
Blood test shows low hemoglobin level
Symptoms indicate anemia
OR
OR
Inner surface of eyes lids pale
Consistent Low Blood Pressure
Feeling Listless

Figure: Inference Network Example

Flowcharts

Flow charts also capture knowledge of strategies. They may be used to represent a sequence of steps that depict the order of application of rule sets. Try making a flow chart that depicts the following strategy. The doctor begins by asking symptoms. If they are not indicative of some disease the doctor will not ask for specific tests. If it is symptomatic of two or three potential diseases, the doctor decides which disease to check for first and rules out potential diagnoses in some heuristic sequence.

5.12.6 Knowledge design

After knowledge analysis is done, we enter the knowledge design phase. At the end of this phase, we have

- Knowledge definition
- Detailed design
- Decision of how to represent knowledge
    - Rules and Logic
    - Frames
- Decision of a development tool. Consider whether it supports your planned strategy.
- Internal fact structure
- Mock interface

5.12.7 Code

This phase occupies the least time in the ESDLC. It involves coding, preparing test cases, commenting code, developing user's manual and installation guide. At the end of this phase the system is ready to be tested.

5.12.8 CLIPS lec 22

We will now look at a tool for expert system development. CLIPS stands for C Language Integrated Production System. CLIPS is an expert system tool which provides a complete environment for the construction of rule and object based expert systems.Download CLIPS for windows (CLIPSWin.zip) from: http://www.ghg.net/clips/download/executables/pc/. Also download the complete documentation including the programming guide from: http://www.ghg.net/clips/download/documentation/

The guides that you download will provide comprehensive guidance on programming using CLIP. Here are some of the basics to get you started

Entering and Exiting CLIPS

When you start executable, you will see prompt

```
CLIPS>
```

Commands can be entered here

To leave CLIPS, enter

```
(exit)
```

All commands use ( ) as delimiters, i.e. all commands are enclosed in brackets.
A simple command example for adding numbers

```
CLIPS> (+ 3 4)
```

Fields

Fields are the main types of fields/tokens that can be used with clips. They can be:

- Numeric fields: consist of sign, value and exponent
    - Float .e.g. 3.5e-10
    - Integer e.g. -1 , 3
- Symbol: ASCII characters, ends with delimiter. e.g. family
- String: Begins and ends with double quotation marks, “Ali is Ahmed’s brother”

Remember that CLIPS is case sensitive

The Deftemplate construct

Before facts can be added, we have to define the format for our relations.Each relation consists of: relation name, zero or more slots (arguments of the relation)
The Deftemplate construct defines a relation’s structure

```
        (deftemplate <relation-name> [<optional   comment>] <slot-definition>
       e.g.
   CLIPS> ( deftemplate  father “Relation father”
                         (slot fathersName)
                         (slot sonsName) )
```

Adding facts

Facts are added in the predicate format. The deftemplate construct is used to inform CLIPS of the structure of facts. The set of all known facts is called the *fact list.* To add facts to the fact list, use the assert command, e.g.
Facts to add:
man(ahmed)
father(ahmed, belal)
brother(ahmed, chand)

```
CLIPS> (assert ( man ( name “Ahmed” ) ) )

CLIPS>(assert ( father ( fathersName “Ahmed”) (sonsName “Belal”) ) )
```

Viewing fact list

After adding facts, you can see the fact list using command: (facts). You will see that a fact index is assigned to each fact, starting with 0. For long fact lists, use the format
(facts [<start> [<end>]])
For example:
(facts 1 10) lists fact numbers 1 through 10

Removing facts

The retract command is used to remove or retract facts. For example:
(retract 1) removes fact 1

(retract 1 3) removes fact 1 and 3

Modifying and duplicating facts

We add a fact:
CLIPS>(assert (father (fathersName “Ahmed”) (sonsName “Belal”)))

To modify the fathers name slot, enter the following:

```
CLIPS> (modify 2 ( fathersName “Ali Ahmed”))
```

Notice that a new index is assigned to the modified fact
To duplicate a fact, enter:

```
CLIPS> (duplicate 2 (name “name”) )
```

The WATCH command

The WATCH command is used for debugging programs. It is used to view the assertion and modification of facts. The command is

```
CLIPS> (watch facts)
```

After entering this command, for subsequent commands, the whole sequence of events will be shown. To turn off this option, use:
(unwatch facts)

The DEFFACTS construct

These are a set of facts that are automatically asserted when the (reset) command is used, to set the working memory to its initial state. For example:

```
CLIPS> (deffacts myFacts “My known facts
( man ( name “Ahmed” ) )
(father
        (fathersName“Ahmed”)
        (sonsName “Belal”) ) )
```

The Components of a rule

The Defrule construct is used to add rules. Before using a rule the component facts need to be defined. For example, if we have the rule

IF Ali is Ahmed’s father
THEN Ahmed is Ali’s son

We enter this into CLIPS using the following construct:

```
        ;Rule header
        (defrule isSon “An example rule”
        ; Patterns
        (father (fathersName “ali”) (sonsName “ahmed”)
        ;THEN
        =>
        ;Actions
        (assert (son (sonsName “ahmed”) (fathersName “ali”)))
        )
```

CLIPS attempts to match the pattern of the rules against the facts in the fact list. If all patterns of a rule match, the rule is activated, i.e. placed on the agenda.

Agenda driven control and execution

The *agenda* is the list of activated rules. We use the run command to run the agenda. Running the agenda causes the rules in the agenda to be fired.

```
CLIPS>(run)
```

Displaying the agenda

To display the set of rules on the agenda, enter the command

```
(agenda)
```

Watching activations and rules

You can watch activations in the agenda by entering

```
(watch activations)
```

You can watch rules firing using

```
(watch rules)
```

All subsequent activations and firings will be shown until you turn the watch off using the unwatch command.

Clearing all constructs

(clear) clears the working memory

The PRINTOUT command

Instead of asserting facts in a rule, you can print out messages using

```
(printout t “Ali is Ahmed’s son” crlf)
```

The SET-BREAK command

This is a debugging command that allows execution of an agenda to halt at a specified rule (breakpoint)

```
        (set-break isSon)
```

Once execution stops, run is used to resume it again.

(remove-break isSon) is used to remove the specified breakpoint.

Use (show-breaks) to view all breakpoints.

Loading and saving constructs

Commands cannot be loaded from a file; they have to be entered at the command prompt. However constructs like deftemplate, deffacts and defrules can be loaded from a file that has been saved using .clp extension. The command to load the file is:
(load “filename.clp”)

You can write out constructs in file editor, save and load. Also (save “filename.clp”) saves all constructs currently loaded in CLIPS to the specified file.

Pattern matching

Variables in CLIPS are preceded by ?, e.g.
?speed
?name
Variables are used on left hand side of a rule. They are bound to different values and once bound may be referenced on the right hand side of a rule. Multi-field wildcard variables may be bound to one or more field of a pattern. They are preceded by $? e.g. $?name will match to entire name(last, middle and first)

Below are some examples to help you see the above concept in practice:

Example 1

```
;This is a comment, anything after a semicolon is a comment
;Define initial facts
(deffacts startup (animal dog) (animal cat) (animal duck) (animal turtle)(animal horse) (warm-
blooded dog) (warm-blooded cat) (warm-blooded duck) (lays-eggs duck) (lays-eggs turtle) (child-
of dog puppy) (child-of cat kitten) (child-of turtle hatchling))

;Define a rule that prints animal names
(defrule animal (animal ?x) => (printout t "animal found: " ?x crlf))

;Define a rule that identifies mammals
(defrule mammal
  (animal ?name)
  (warm-blooded ?name)
  (not (lays-eggs ?name))
  =>
  (assert (mammal ?name))
  (printout t ?name " is a mammal" crlf))

;Define a rule that adds mammals
(defrule mammal2
  (mammal ?name)
  (child-of ?name ?young)
  =>
  (assert (mammal ?young))
  (printout t ?young " is a mammal" crlf))
```

```
;Define a rule that removes mammals from fact list
;(defrule remove-mammals
;  ?fact <- (mammal ?)
;  =>
;  (printout t "retracting " ?fact crlf)
;  (retract ?fact))
;

;Define rule that adds child’s name after asking user
(defrule what-is-child
       (animal ?name)
       (not (child-of ?name ?))
=>
       (printout t "What do you call the child of a " ?name "?")
       (assert (child-of ?name (read))))
```

Example 2

```
;OR example
;note: CLIPS operators use prefix notation
(deffacts startup (weather raining))

(defrule take-umbrella
  (or (weather raining)
     (weather snowing))
  =>
  (assert (umbrella required)))
```

These two are very basic examples. You will find many examples in the CLIPS documentation that you download. Try out these examples.

Below is the code for the case study we discussed in the lectures, for the automobile diagnosis problem discussion that is given in Durkin’s book. This is an implementation of the solution. (The solution is presented by Durkin as rules in your book).

```
;Helper functions for asking user questions
(deffunction ask-question (?question $?allowed-values)
   (printout t ?question)
   (bind ?answer (readline))

   (while (and (not (member ?answer ?allowed-values)) (not(eq ?answer "q"))) do
            (printout t ?question)
         (bind ?answer (readline)))
   (if (eq ?answer "q")
          then (clear))
?answer)

(deffunction yes-or-no-p (?question)
   (bind ?response (ask-question ?question "yes" "no" "y" "n"))
   (if (or (eq ?response "yes") (eq ?response "y"))
      then TRUE
      else FALSE))

;startup rule
```

```
(deffacts startup (task begin))

(defrule startDiagnosis
        ?fact <- (task begin)
        =>
        (retract ?fact)
        (assert (task test_cranking_system))
        (printout t "Auto Diagnostic Expert System" crlf)
)

;---------------------------------
;Test Display Rules
;---------------------------------
(defrule testTheCrankingSystem
        ?fact <- (task test_cranking_system)
        =>
        (printout t "Cranking System Test" crlf)
        (printout t "--------------------" crlf)
        (printout t "I want to first check out the major components of the cranking system. This
includes such items as the battery, cables, ignition switch and starter. Usually, when a car does
not start the problem can be found with one of     these components" crlf)
        (printout t "Steps: Please turn on the ignition switch to energize the starting motor" crlf)
        (bind ?response
                (ask-question "How does your engine turn:  (slowly or not at all/normal)? "
            "slowly or not at all" "normal") )
        (assert(engine_turns ?response))
)

(defrule testTheBatteryConnection
        ?fact <- (task test_battery_connection)
        =>
        (printout t "Battery Connection Test" crlf)
        (printout t "-----------------------" crlf)
        (printout t "I next want to see if the battery connections are good. Often, a bad connection
will appear like a bad battery" crlf)
        (printout t "Steps: Insert a screwdriver between the battery post and the cable clamp.
Then turn the headlights on high beam and observe the lights as the screwdriver is twisted." crlf)
        (bind ?response
                (ask-question "What happens to the lights:  (brighten/don't brighten/not on)? "
            "brighten" "don't brighten" "not on") )
        (assert(screwdriver_test_shows_that_lights ?response))
)

(defrule testTheBattery
        ?fact <- (task test_battery)
        =>
        (printout t "Battery Test" crlf)
        (printout t "------------" crlf)
        (printout t "The state of the battery can be checked with a hydrometer. This is a good test
to determine the amount of charge in the battery and is better than a simple voltage
measurement" crlf)
        (printout t "Steps: Please test each battery cell with the hydrometer and note each cell's
specific gravity reading." crlf)
        (bind ?response
                (ask-question "Do all cells have a reading above 1.2:  (yes/no)? "
            "yes" "no" "y" "n") )
        (assert(battery_hydrometer_reading_good ?response))
)

(defrule testTheStartingSystem
        ?fact <- (task test_starting_system)
        =>
        (printout t "Starting System Test" crlf)
        (printout t "--------------------" crlf)
        (printout t "Since the battery looks good, I want to next test the starter and solenoid" crlf)
        (printout t "Steps: Please connect a jumper from the battery post of the solenoid to the
starter post of hte solenoid. Then turn the ignition key." crlf)
        (bind ?response
                (ask-question "What happens after you make this connection and turn the key:
(engine turns normally/starter buzzes/engine turns slowly/nothing)? "
            "engine turns normally" "starter buzzes" "engine turns slowly" "nothing" ))
        (assert(starter ?response))
)

(defrule testTheStarterOnBench
        ?fact <- (task test_starter_on_bench)
        =>
        (bind ?response
                (ask-question "Check your starter on bench:  (meets specifications/doesn't meet
specifications)? "
            "meets specifications" "doesn't meet specifications") )
        (assert(starter_on_bench ?response))
)

(defrule testTheIgnitionOverrideSwitch
        ?fact <- (task test_ignition_override_switches)
        =>
        (bind ?response
                (ask-question "Check the ignition override switches:  starter(operates/doesn't
operate)? "
            "operates" "doesn't operate") )
        (assert(starter_override ?response))
)

(defrule testTheIgnitionSwitch
        ?fact <- (task test_ignition_switch)
        =>
        (bind ?response
                (ask-question "Test your ignition swich. The voltmeter:  (moves/doesn't move)? "
"moves" "doesn't move") )
        (assert(voltmeter ?response))
)
(defrule testEngineMovement
        ?fact <- (task test_engine_movement)
        =>
        (bind ?response
                (ask-question "Test your engine movement:  (doesn't move/moves freely)? "
            "doesn't move" "moves freely") )
        (assert(engine_turns ?response))
)
;--------------------------------
;Test Cranking System Rules
;--------------------------------
(defrule crankingSystemIsDefective
        ?fact <- (task test_cranking_system)
        (engine_turns "slowly or not at all")
        =>
        (assert(cranking_system defective))
        (retract ?fact)
        ;(bind ?response )
        (printout t "It seems like the cranking system is defective! I will now identify the problem
with the cranking system" crlf)
        (assert (task test_battery_connection))
)

(defrule crankingSystemIsGood
        ?fact <- (task test_cranking_system)
        (engine_turns "normal")
        =>
        (assert( cranking_system "good"))
        (retract ?fact)
        (printout t "Your Cranking System Appears to be Good" crlf)
        (printout t "I will now check your ignition system" crlf)
        (assert(task test_ignition_switch))        ;in complete system, replace this with
test_ignition_system
)

;---------------------------------
;Test Battery Connection Rules
;---------------------------------
(defrule batteryConnectionIsBad
        ?fact <- (task test_battery_connection)
        (or  (screwdriver_test_shows_that_lights  "brighten")(screwdriver_test_shows_that_lights
"not on"))

        =>
        (assert( problem bad_battery_connection))
        (printout t "The problem is a bad battery connection" crlf)
        (retract ?fact)
        (assert (task done))
)

(defrule batteryConnectionIsGood
        ?fact <- (task test_battery_connection)
        (screwdriver_test_shows_that_lights "don't brighten")
        =>
        (printout t "The problem does not appear to be a bad battery connection." crlf)
        (retract ?fact)
        (assert(task test_battery))
)

;---------------------------------
;Test Battery Rules
;---------------------------------
(defrule batteryChargeIsBad
        ?fact <- (task test_battery)
        (battery_hydrometer_reading_good "no")
        =>
        (assert( problem bad_battery))
        (printout t "The problem is a bad battery." crlf)
        (retract ?fact)
        (assert (task done))

)

(defrule batteryChargeIsGood
        ?fact <- (task test_battery)
        (battery_hydrometer_reading_good "yes")
        =>
        (retract ?fact)
        (printout t "The problem does not appear to be a bad battery." crlf)
        (assert(task test_starting_system))
)

;--------------------------------
;Test Starter Rules
;--------------------------------
(defrule RunStarterBenchTest
        ?fact <- (task test_starting_system)
        (or (starter "starter buzzes")(starter "engine turns slowly"))
        =>
        (retract ?fact)
        (assert (task test_starter_on_bench))
)

(defrule solenoidBad
        ?fact <- (task test_starting_system)
        (starter "nothing")
        =>
        (retract ?fact)
        (assert (problem bad_solenoid))
        (printout t "The problem appears to be a bad solenoid." crlf)
        (assert(task done))
)

(defrule starterTurnsEngineNormally
        ?fact <- (task test_starting_system)
        (starter "engine turns normally")
        =>
        (retract ?fact)
        (printout t "The problem does not appears to be a bad solenoid." crlf)
        (assert(task test_ignition_override_switches))
)

;--------------------------------
;Starter Bench Test Rules
;--------------------------------
(defrule starterBad
        ?fact <- (task test_starter_on_bench)
        (starter_on_bench "doesn't meet specifications")
        =>
        (assert( problem bad_starter))
        (printout t "The problem is a bad starter." crlf)
        (retract ?fact)
        (assert (task done))

)

(defrule starterGood
        ?fact <- (task test_starter_on_bench)
        (starter_on_bench "meets specifications")
        =>
        (retract ?fact)
        (printout t "The problem does not appear to be with starter." crlf)
        (assert(task test_engine_movement))
)

;--------------------------------
;Override Swich Test Rules
;--------------------------------
```

```
(defrule overrideSwitchBad
        ?fact <- (task test_ignition_override_switches)
        (starter_override "operates")
        =>
        (assert( problem bad_override_switch))
        (printout t "The problem is a bad override switch." crlf)
        (retract ?fact)
        (assert (task done))

)

(defrule starterWontOperate
        ?fact <- (task test_ignition_override_switches)
        (starter_override "doesn't operate")
        =>
        (retract ?fact)
        (printout t "The problem does not appear to be with override switches." crlf)
        (assert(task test_ignition_switch))
)

;--------------------------------
;Engine Movement Test
;--------------------------------
(defrule engineBad
        ?fact <- (task test_engine_movement)
        (engine_turns "doesn't move")
        =>
        (assert( problem bad_engine))
        (printout t "The problem is a bad engine." crlf)
        (retract ?fact)
        (assert (task done))

)

(defrule engineMovesFreely
        ?fact <- (task test_engine_movement)
        (engine_turns "moves freely")
        =>
        (retract ?fact)
        (printout t "The problem does not appear to be with the engine." crlf)
        (printout t "Test your engine timing. That is beyond my scope for now" crlf) ; actual test
goes here in final system.
        (assert(task perform_engine_timing_test))
)


;--------------------------------
;Ignition Switch Test
;--------------------------------
;these reluts for the ignition system are not complete, they are added only to test the control flow.

(defrule ignitionSwitchConnectionsBad
        ?fact <- (task test_ignition_switch)
        (voltmeter "doesn't move")
        =>
        (assert( problem bad_ignition_switch_connections))
        (printout t "The problem is bad ignition switch connections." crlf)
        (retract ?fact)
        (assert (task done))
)

(defrule ignitionSwitchBad
        ?fact <- (task test_ignition_switch)
        (voltmeter "moves")
        =>
        (assert( problem bad_ignition_switch))
        (printout t "The problem is a bad ignition switch." crlf)
        (retract ?fact)
        (assert (task done))
)
```

6 Handling uncertainty with fuzzy systems

6.1 Introduction

Ours is a vague world. We humans, talk in terms of ‘maybe’, ‘perhaps’, things which cannot be defined with cent percent authority. But on the other hand, conventional computer programs cannot understand natural language as computers cannot work with vague concepts. Statements such as: “Umar is tall”, are difficult for computers to translate into definite rules. On the other hand, “Umar’s height is 162 cm”, doesn’t explicitly state whether Umar is tall or short.

We’re driving in a car, and we see an old house. We can easily classify it as an old house. But what exactly is an old house? Is a 15 years old house, an old house? Is 40 years old house an old house? Where is the dividing line between the old and the new houses? If we agree that a 40 years old house is an old house, then how is it possible that a house is considered new when it is 39 years, 11 months and 30 days old only. And one day later it has become old all of a sudden? That would be a bizarre world, had it been like that for us in all scenarios of life.

Similarly human beings form vague groups of things such as ‘short men’, ‘warm days’, ‘high pressure’. These are all groups which don’t appear to have a well defined boundary but yet humans communicate with each other using these terminologies.

6.2 Classical sets

A classical set is a container, which wholly includes or wholly excludes any given element. It’s called classical merely because it has been around for quite some time. It was Aristotle who came up with the ‘Law of the Excluded Middle’, which states that any element X, must be either in set A or in set not-A. It cannot be in both. And these two sets, set A and set not-A should contain the entire universe between them.

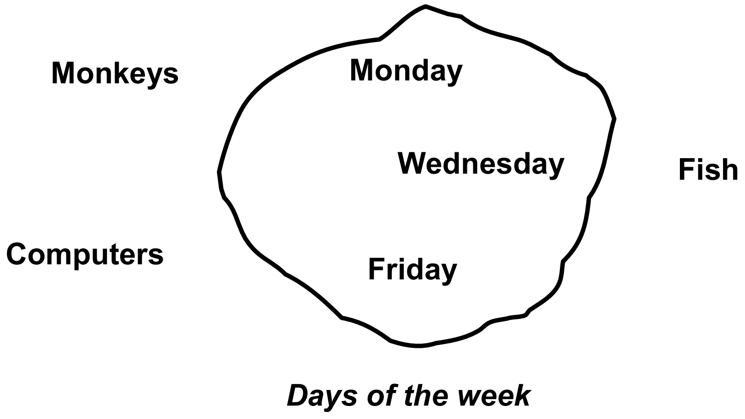


Figure : Classical Set

Let’s take the example of the set ‘Days of the week’. This is a classical set in which all the 7 days from Monday up until Sunday belong to the set, and everything possible other than that that you can think of, monkeys, computers, fish, telephone, etc, are definitely not a part of this set. This is a binary classification system, in which everything must be asserted or denied. In the case of Monday, it will be asserted to be an element of the set of ‘days of the week’, but tuna fish will not be an element of this set.

6.3 Fuzzy sets

Fuzzy sets, unlike classical sets, do not restrict themselves to something lying wholly in either set A or in set not-A. They let things sit on the fence, and are thus closer to the human world. Let us, for example, take into consideration ‘days of the week**end**’. The classical set would say strictly that only Saturday and Sunday are a part of weekend, whereas most of us would agree that we do feel like it’s a weekend somewhat on Friday as well. Actually we’re more excited about the weekend on a Friday than on Sunday, because on Sunday we know that the next day is a working day. This concept is more vividly shown in the following figure.

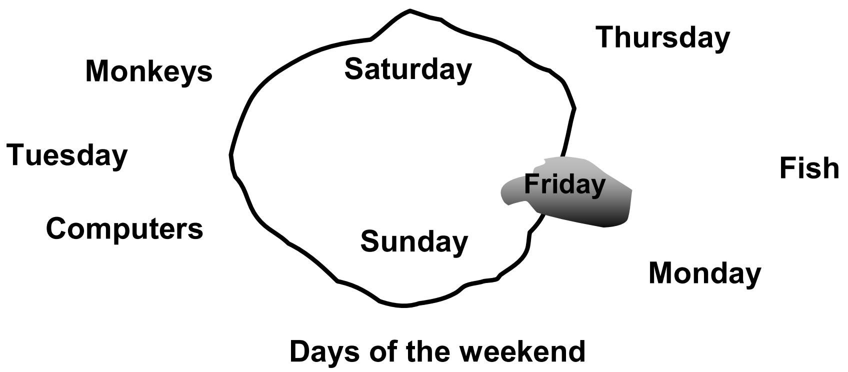


Figure : Fuzzy Sets

Another diagram that would help distinguish between crisp and fuzzy representation of days of the weekend is shown below.

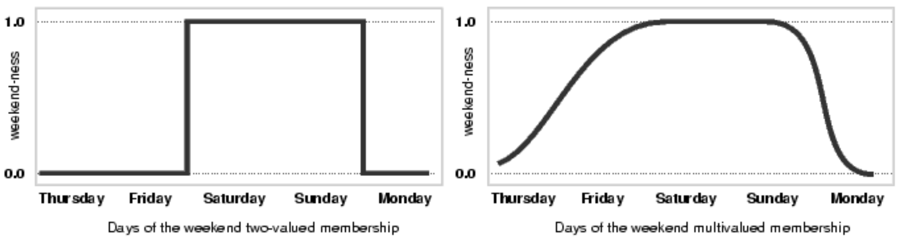


Figure : Crisp v/s Fuzzy

The left side of the above figure shows the crisp set ‘days of the weekend’, which is a Boolean two-valued function, so it gives a value of 0 for all week days except Saturday and Sunday where it gives an abrupt 1 and then back to 0 as soon as Sunday ends. On the other hand, Fuzzy set is a multi-valued function, which in this case is shown by a smoothly rising curve for the weekend, and even Friday has a good membership in the set ‘days of the weekend’.

Same is the case with seasons. There are four seasons in Pakistan: Spring, Summer, Fall and Winter. The classical/crisp set would mark a hard boundary between the two adjacent seasons, whereas we know that this is not the case in reality. Seasons gradually change from one into the next. This is more clearly explained in the figure below.

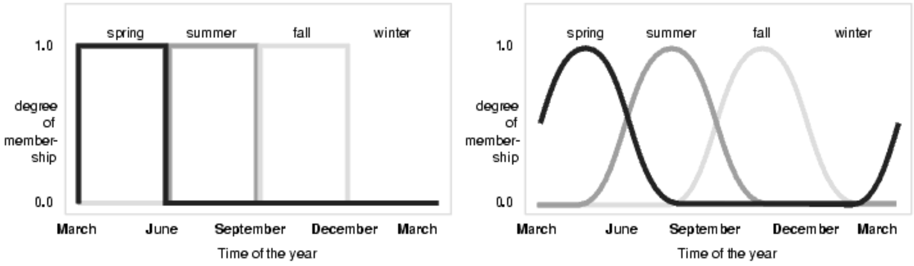


Figure: Seasons [Left: Crisp] [Right: Fuzzy]

This entire discussion brings us to a question: What is fuzzy logic?

6.4 Fuzzy Logic

Fuzzy logic is a superset of conventional (Boolean) logic that has been extended to handle the concept of partial truth -- truth values between "completely true" and "completely false".

Dr. Lotfi Zadeh of UC/Berkeley introduced it in the 1960's as a means to model the uncertainty of natural languages. He was faced with a lot of criticism but today the vast number of fuzzy logic applications speak for themselves:

- Self-focusing cameras
- Washing machines that adjust themselves according to the dirtiness of the clothes
- Automobile engine controls
- Anti-lock braking systems
- Color film developing systems
- Subway control systems
- Computer programs trading successfully in financial markets

6.4.1 Fuzzy logic represents partial truth

Any statement can be fuzzy. The tool that fuzzy reasoning gives is the ability to reply to a yes-no question with a not-quite-yes-or-no answer. This is the kind of thing that humans do all the time (think how rarely you get a straight answer to a seemingly simple question; what time are you coming home? Ans: soon. Q: are you coming? Ans: I might) but it's a rather new trick for computers.

How does it work? Reasoning in fuzzy logic is just a matter of generalizing the familiar yes-no (Boolean) logic. If we give "true" the numerical value of 1 and "false" the numerical value of 0, we're saying that fuzzy logic also permits in-between values like 0.2 and 0.7453.

"In fuzzy logic, the truth of any statement becomes matter of degree"

We will understand the concept of degree or partial truth by the same example of days of the weekend. Following are some questions and their respective answers:

– Q: Is Saturday a weekend day?

- A: 1 (yes, or true)
- Q: Is Tuesday a weekend day?
- A: 0 (no, or false)
- Q: Is Friday a weekend day?
- A: 0.7 (for the most part yes, but not completely)
- Q: Is Sunday a weekend day?
- A: 0.9 (yes, but not quite as much as Saturday)

6.4.2 Boolean versus fuzzy

Let's look at another comparison between boolean and fuzzy logic with the help of the following figures. There are two persons. Person A is standing on the left of person B. Person A is definitely shorter than person B. But if boolean gauge has only two readings, 1 and 0, then a person can be either tall or short. Let's say if the cut off point is at 5 feet 10 inches then all the people having a height greater than this limit are taller and the rest are short.

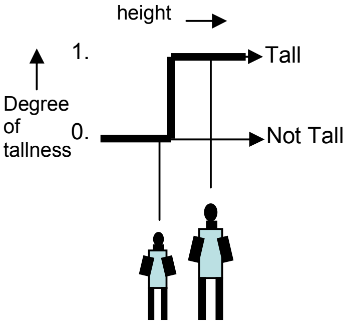


Figure: Boolean Logic

On the other hand, in fuzzy logic, you can define any function represented by any mathematical shape. The output of the function can be discreet or continuous. The output of the function defines the membership of the input or the degree of truth. As in this case, the same person A is termed as 'Not very tall'. This isn't absolute 'Not tall' as in the case of boolean. Similarly, person B is termed as 'Quite Tall' as apposed to the absolute 'Tall' classification by the boolean parameters. In short, fuzzy logic lets us define more realistically the true functions that define real world scenarios.

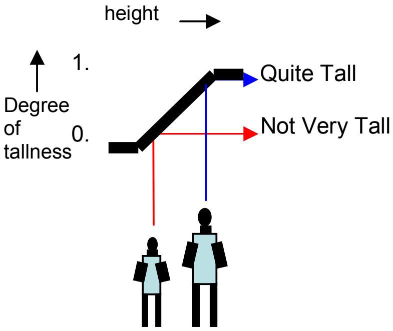


Figure: Fuzzy Logic

6.4.3 Membership Function (μ)

The degree of truth that we have been talking about, is specifically driven out by a function called the membership function. It can be any function ranging from a simple linear straight line to a complicated spline function or a polynomial of a higher degree.

Some characteristics of the membership functions are:

- It is represented by the Greek symbol *μ*
- Truth values range between 0.0 and 1.0
    - Where 0.0 normally represents absolute falseness
    - And 1.0 represent absolute truth

Consider the following sentence:
“Amma ji is old”

In (crisp) set terminology, Amma ji belongs to the set of old people. We define *μ*OLD, the membership function operating on the fuzzy set of old people. *μ*OLD takes as input one variable, which is age, and returns a value between 0.0 and 1.0.

- If Amma ji’s age is 75 years
    - We might say *μ*OLD(Amma ji’s age) = 0.75
        - Meaning Amma ji is quite old
- For Amber, a 20 year old:
    - We might say *μ*OLD(Amber’s age) = 0.2
        - Meaning that Amber is not very old

For this particular age, the membership function is defined by a linear line with positive slope.

6.4.4 Fuzzy vs. probability

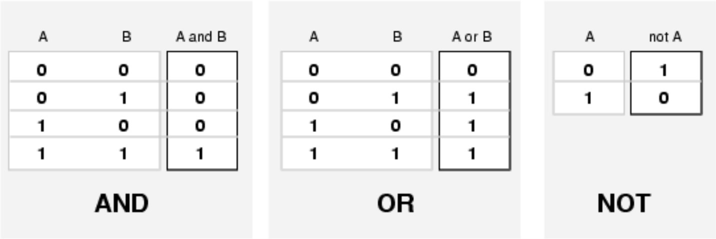
It’s important to distinguish at this point the difference between probability and fuzzy, as both operate over the same range [0.0 to 1.0]. To understand their differences lets take into account the following case, where Amber is a 20 years old girl.
*μ*OLD(Amber) = 0.2

In probability theory:
There is a 20% *chance* that Amber belongs to the set of old people, there’s an 80% *chance* that she doesn’t belong to the set of old people.

In fuzzy terminology:
Amber is definitely not old or some other term corresponding to the value 0.2. But there are certainly no chances involved, no guess work left for the system to classify Amber as young or old.

6.4.5 Logical and fuzzy operators

Before we move on, let’s take a look at the logical operators. What these operators help us see is that fuzzy logic is actually a superset of conventional boolean logic. This might appear to be a startling remark at first, but look at Table 1 below.



A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

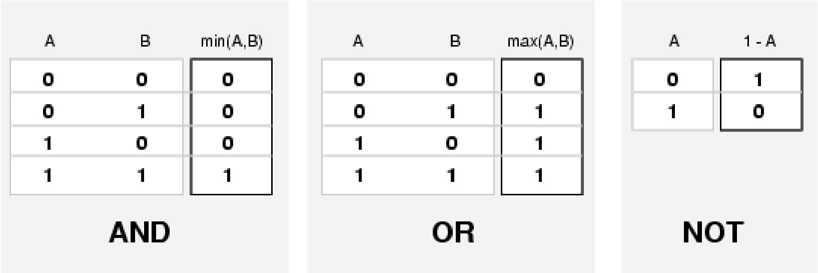
OR

A	not A
0	1
1	0

NOT

Table: Logical Operators

The table above lists down the AND, OR and NOT operators and their respective values for the boolean inputs. Now for fuzzy systems we needed the exact operators which would act exactly the same way when given the extreme values of 0 and 1, and that would in addition also act on other real numbers between the ranges of 0.0 to 1.0. If we choose min (minimum) operator in place for AND, we get the same output, similarly max (maximum) operator replaces OR, and 1-A replaces NOT of A.



A	B	min(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	max(A,B)
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	1 - A
0	1
1	0

NOT

Table: Fuzzy Operators

In a lot of ways these operators seem to make sense. When we are ANDing two domains, A and B, we do want to have the intersection as a result, and intersection gives us the minimum overlapping area, hence both are equivalent. Same is the case with max and 1-A.

The figure below explains these logical operators in a non-tabular form. If we allow the fuzzy system to take on only two values, 0 and 1, then it becomes boolean logic, as can be seen in the figure, top row.

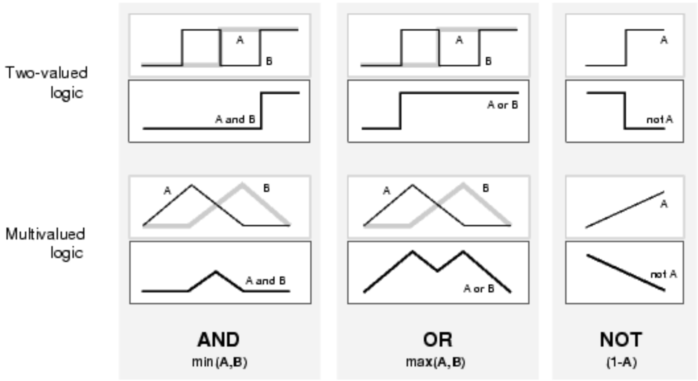


Figure: Logical vs Fuzzy Operators

It would be interesting to mention here that the graphs for A and B are nothing more than a distribution, for instance if A was the set of short men, then the graph A shows the entire distribution of short men where the horizontal axis is the increasing height and the vertical axis shows the membership of men with different heights in the function 'short men'. The men who would be taller would have little or 0 membership in the function, whereas they would have a significant membership in set B, considering it to be the distribution of tall men.

6.4.6 Fuzzy set representation

Usually a triangular graph is chosen to represent a fuzzy set, with the peak around the mean, which is true in most real world scenarios, as majority of the population lies around the average height. There are fewer men who are exceptionally tall or short, which explains the slopes around both sides of the triangular distribution. It's also an approximation of the Gaussian curve, which is a more general function in some aspects.

Apart from this graphical representation, there's also another representation which is more handy if you were to write down some individual members along with their membership. With this representation, the set of Tall men would be written like follows:

- Tall = (0/5, 0.25/5.5, 0.8/6, 1/6.5, 1/7)
  - Numerator: membership value
  - Denominator: actual value of the variable

For instance, the first element is 0/5 meaning, that a height of 5 feet has 0 membership in the set of tall people, likewise, men who are 6.5 feet or 7 feet tall have a membership value of maximum 1.

6.4.7 Fuzzy rules

First of all, let us revise the concept of simple If-Then rules. The rule is of the form:
If x is A then y is B
Where x and y are variables and A and B are some distributions/fuzzy sets. For example:

If *hotel service* is *good* then tip is *average*

Here hotel service is a linguistic variable, which when given to a real fuzzy system would have a certain crisp value, maybe a rating between 0 and 10. This rating would have a membership value in the fuzzy set of ‘good’. We shall evaluate this rule in more detail in the case study that follows.

Antecedents can have multiple parts:

- If wind is mild and racquets are good then playing badminton is fun

In this case all parts of the antecedent are resolved simultaneously and resolved to a single number using logical operators

The consequent can have multiple parts as well

- if temperature is cold then hot water valve is open and cold water valve is shut

How is the consequent affected by the antecedent? The consequent specifies that a fuzzy set be assigned to the output. The ***implication function*** then modifies that fuzzy set to the degree specified by the antecedent. The most common ways to modify the output fuzzy set are truncation using the min function (where the fuzzy set is "chopped off“).

Consider the following figure, which demonstrates the working of fuzzy rule system on one rule, which states:
“If **service is excellent** *or* **food is delicious** then **tip is generous”**

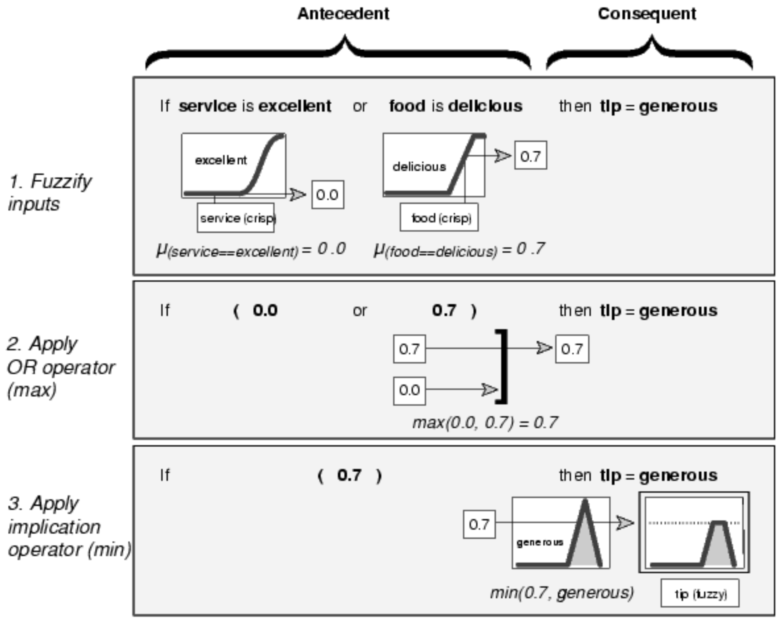

Antecedent
Consequent
1. Fuzzify inputs
If service is excellent or food is delicious then tip = generous
excellent
0.0
service (crisp)
delicious
0.7
food (crisp)
μ(service==excellent) = 0.0
μ(food==delicious) = 0.7
2. Apply OR operator (max)
If (0.0 or 0.7) then tip = generous
0.7
0.0
0.7
max(0.0, 0.7) = 0.7
3. Apply implication operator (min)
If (0.7) then tip = generous
0.7
generous
min(0.7, generous)
tip (fuzzy)

Figure: Fuzzy If-Then Rule

Fuzzify inputs: Resolve all fuzzy statements in the antecedent to a degree of membership between 0 and 1. If there is only one part to the antecedent, this is the degree of support for the rule. In the example, the user gives a rating of 3 to the service, so its membership in the fuzzy set 'excellent' is 0. Likewise, the user gives a rating of 8 to the food, so it has a membership of 0.7 in the fuzzy set of delicious.

Apply fuzzy operator to multiple part antecedents: If there are multiple parts to the antecedent, apply fuzzy logic operators and resolve the antecedent to a single number between 0 and 1. This is the degree of support for the rule. In the example, there are two parts to the antecedent, and they have an OR operator in between them, so they are resolved using the max operator and max(0,0,0.7) is 0.7. That becomes the output of this step.

Apply implication method: Use the degree of support for the entire rule to shape the output fuzzy set. The consequent of a fuzzy rule assigns an entire fuzzy set to the output. This fuzzy set is represented by a membership function that is chosen to indicate the qualities of the consequent. If the antecedent is only partially true, (i.e., is assigned a value less than 1), then the output fuzzy set is truncated according to the implication method.

In general, one rule by itself doesn't do much good. What's needed are two or more rules that can play off one another. The output of each rule is a fuzzy set. The output fuzzy sets for each rule are then *aggregated* into a single output fuzzy set. Finally the resulting set is *defuzzified*, or resolved to a single number. The next section shows how the whole process works from beginning to end for a particular type of fuzzy inference system.

6.5 Fuzzy inference system

Fuzzy inference system (FIS) is the process of formulating the mapping from a given input to an output using fuzzy logic. This mapping then provides a basis from which decisions can be made, or patterns discerned

Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multidisciplinary nature, fuzzy inference systems are associated with a number of names, such as fuzzy-rule-based systems, fuzzy expert systems, fuzzy modeling, fuzzy associative memory, fuzzy logic controllers, and simply (and ambiguously !!) fuzzy systems. Since the terms used to describe the various parts of the fuzzy inference process are far from standard, we will try to be as clear as possible about the different terms introduced in this section.

Mamdani's fuzzy inference method is the most commonly seen fuzzy methodology. Mamdani's method was among the first control systems built using fuzzy set theory. It was proposed in 1975 by Ebrahim Mamdani as an attempt to control a steam engine and boiler combination by synthesizing a set of linguistic control rules obtained from experienced human operators. Mamdani's effort was based on Lotfi Zadeh's 1973 paper on fuzzy algorithms for complex systems and decision processes.

6.5.1 Five parts of the fuzzy inference process

- Fuzzification of the input variables
- Application of fuzzy operator in the antecedent (premises)
- Implication from antecedent to consequent
- Aggregation of consequents across the rules
- Defuzzification of output

To help us understand these steps, let's do a small case study.

6.5.2 Case Study: dinner for two

We present a small case study in which two people go for a dinner to a restaurant. Our fuzzy system will help them decide the percentage of tip to be given to the waiter (between 5 to 25 percent of the total bill), based on their rating of service and food. The rating is between 0 and 10. The system is based on three fuzzy rules:

Rule1:
If service is poor or food is rancid then tip is cheap

Rule2:
If service is good then tip is average

Rule3:
If service is excellent or food is delicious then tip is generous

Based on these rules and the input by the diners, the Fuzzy inference system gives the final output using all the inference steps listed above. Let's take a look at those steps one at a time.

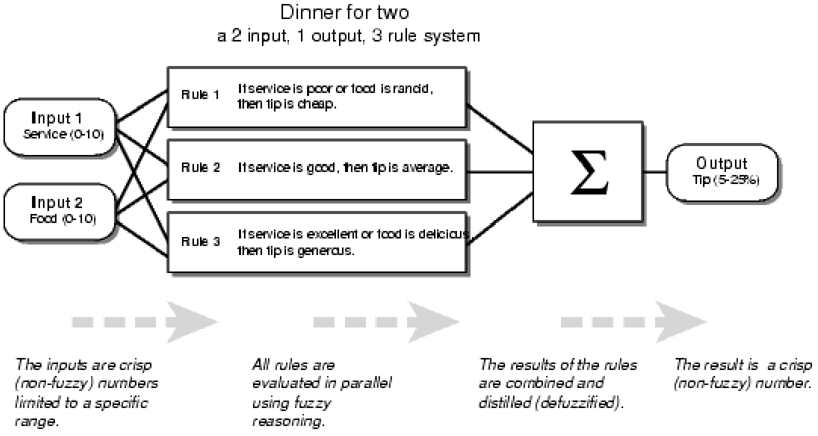

Dinner for two
a 2 input, 1 output, 3 rule system
Input 1
Service (0-10)
Input 2
Food (0-10)
Rule 1
If service is poor or food is rancid, then tip is cheap.
Rule 2
If service is good, then tip is average.
Rule 3
If service is excellent or food is delicious then tip is generous.
Σ
Output
Tip (5-25%)
The inputs are crisp (non-fuzzy) numbers limited to a specific range.
All rules are evaluated in parallel using fuzzy reasoning.
The results of the rules are combined and distilled (defuzzified).
The result is a crisp (non-fuzzy) number.

Figure: Dinner for Two

6.5.2.1 Fuzzify Inputs

The first step is to take the inputs and determine the degree to which they belong to each of the appropriate fuzzy sets via membership functions. The input is always a crisp numerical value limited to the universe of discourse of the input variable (in this case the interval between 0 and 10) and the output is a fuzzy degree of membership in the qualifying linguistic set (always the interval between 0 and 1). Fuzzification of the input amounts to either a table lookup or a function evaluation.

The example we're using in this section is built on three rules, and each of the rules depends on resolving the inputs into a number of different fuzzy linguistic sets: service is poor, service is good, food is rancid, food is delicious, and so on. Before the rules can be evaluated, the inputs must be fuzzified according to each of these linguistic sets. For example, to what extent is the food really delicious? The figure below shows how well the food at our hypothetical restaurant (rated on a scale of 0 to 10) qualifies, (via its membership function), as the linguistic variable "delicious." In this case, the diners rated the food as an 8, which, given our graphical definition of delicious, corresponds to μ = 0.7 for the "delicious" membership function.

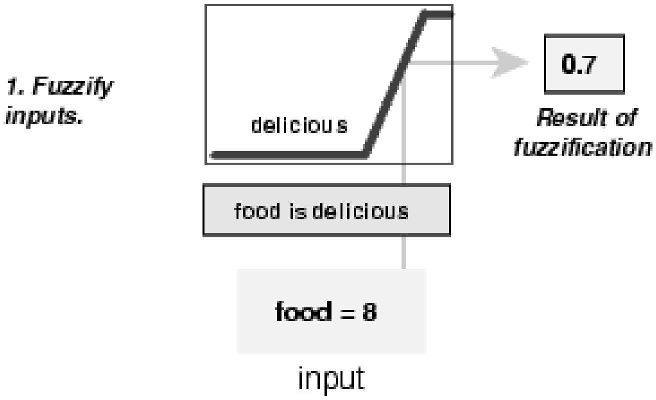


Figure: Fuzzify Input

6.5.2.2 Apply fuzzy operator

Once the inputs have been fuzzified, we know the degree to which each part of the antecedent has been satisfied for each rule. If the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the antecedent for that rule. This number will then be applied to the output function. The input to the fuzzy operator is two or more membership values from fuzzified input variables. The output is a single truth value.

Shown below is an example of the OR operator *max* at work. We're evaluating the antecedent of the rule 3 for the tipping calculation. The two different pieces of the antecedent (service is excellent and food is delicious) yielded the fuzzy membership values 0.0 and 0.7 respectively. The fuzzy OR operator simply selects the maximum of the two values, 0.7, and the fuzzy operation for rule 3 is complete.

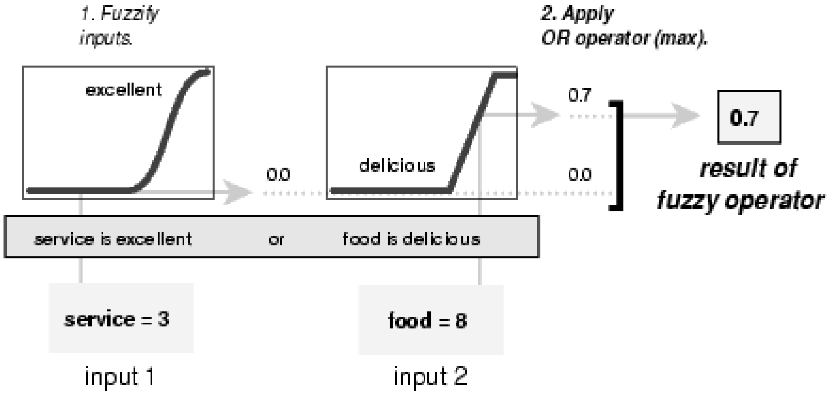


Figure: Apply Fuzzy Operator

6.5.2.3 Apply implication method

Before applying the implication method, we must take care of the rule's weight. Every rule has a *weight* (a number between 0 and 1), which is applied to the number given by the antecedent. Generally this weight is 1 (as it is for this example) and so it has no effect at all on the implication process. From time to time you may want to weigh one rule relative to the others by changing its weight value to something other than 1.

Once proper weightage has been assigned to each rule, the implication method is implemented. A consequent is a fuzzy set represented by a membership function, which weighs appropriately the linguistic characteristics that are attributed to it. The consequent is reshaped using a function associated with the antecedent (a single number). The input for the implication process is a single number given by the antecedent, and the output is a fuzzy set. Implication is implemented for each rule. We will use the *min* (minimum) operator to perform the implication, which truncates the output fuzzy set, as shown in the figure below.

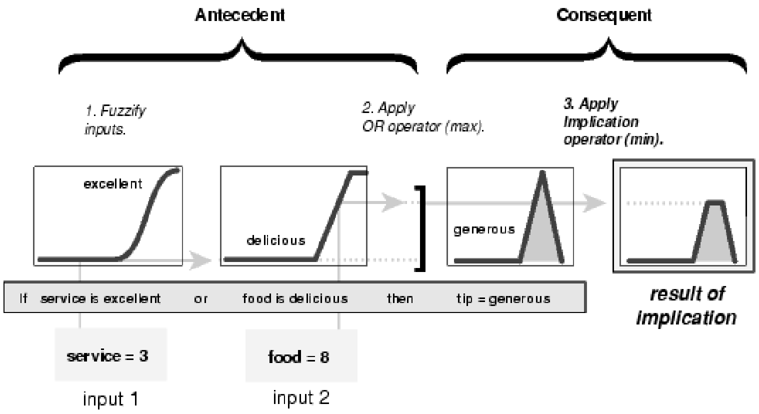


Figure: Apply Implication Method

6.5.2.4 Aggregate all outputs

Since decisions are based on the testing of all of the rules in an FIS (fuzzy inference system), the rules must be combined in some manner in order to make a decision. Aggregation is the process by which the fuzzy sets that represent the outputs of each rule are combined into a single fuzzy set. Aggregation only occurs once for each output variable, just prior to the fifth and final step, defuzzification. The input of the aggregation process is the list of truncated output functions returned by the implication process for each rule. The output of the aggregation process is one fuzzy set for each output variable.

Notice that as long as the aggregation method is commutative (which it always should be), then the order in which the rules are executed is unimportant. Any logical operator can be used to perform the aggregation function: *max* (maximum), *probor* (probabilistic OR), and *sum* (simply the sum of each rule's output set).

In the diagram below, all three rules have been placed together to show how the output of each rule is combined, or aggregated, into a single fuzzy set whose membership function assigns a weighting for every output (tip) value.

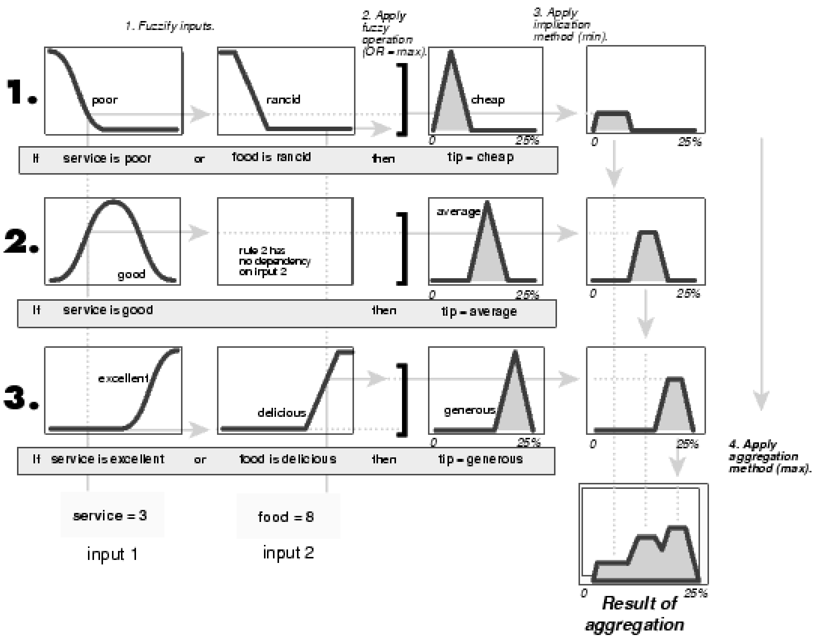


Figure: Aggregate all outputs

6.5.2.5 Defuzzify

The input for the defuzzification process is a fuzzy set (the aggregate output fuzzy set) and the output is a single number. As much as fuzziness helps the rule evaluation during the intermediate steps, the final desired output for each variable is generally a single number. However, the aggregate of a fuzzy set encompasses a range of output values, and so must be defuzzified in order to resolve a single output value from the set.

Perhaps the most popular defuzzification method is the centroid calculation, which returns the center of area under the curve. There are other methods in practice: centroid, bisector, middle of maximum (the average of the maximum value of the output set), largest of maximum, and smallest of maximum.

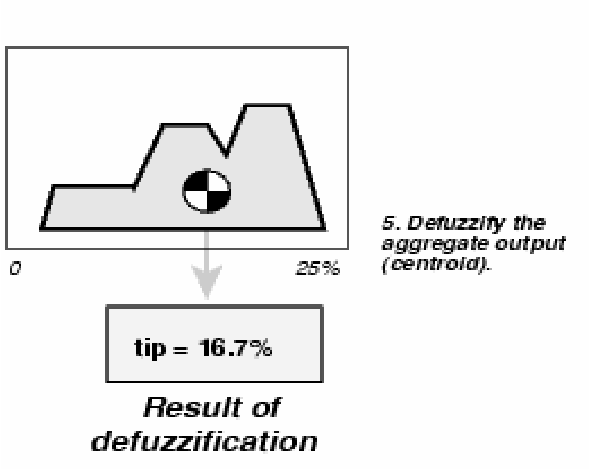

0
25%
5. Defuzzify the aggregate output (centroid).
tip = 16.7%
Result of defuzzification

Figure: Defuzzification

Thus the FIS calculates that in case the food has a rating of 8 and the service has a rating of 3, then the tip given to the waiter should be 16.7% of the total bill.

6.6 Summary

Fuzzy system maps more realistically, the everyday concepts, like age, height, temperature etc. The variables are given fuzzy values. Classical sets, either wholly include something or exclude it from the membership of a set, for instance, in a classical set, a man can be either young or old. There are crisp and rigid boundaries between the two age sets, but in Fuzzy sets, there can be partial membership of a man in both the sets.

6.7 Exercise

1) Think of the membership functions for the following concepts, from the famous quote: “Early to bed, and early to rise, makes a man healthy, wealthy and wise.”
    a. Health
    b. Wealth
    c. Wisdom
2) What do you think would be the implication of using a different shaped curve for a membership function? For example, a triangular, gaussian, square etc
3) Try to come up with at least 5 more rules for the tipping system(Dinner for two case study), such that the system would be a more realistic and complete one.

7 Introduction to learning

7.1 Motivation

Artificial Intelligence (AI) is concerned with programming computers to perform tasks that are presently done better by humans. AI is about human behavior, the discovery of techniques that will allow computers to learn from humans. One of the most often heard criticisms of AI is that machines cannot be called Intelligent until they are able to learn to do new things and adapt to new situations, rather than simply doing as they are told to do. There can be little question that the ability to adapt to new surroundings and to solve new problems is an important characteristic of intelligent entities. Can we expect such abilities in programs? Ada Augusta, one of the earliest philosophers of computing, wrote: "The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform." This remark has been interpreted by several AI critics as saying that computers cannot learn. In fact, it does not say that at all. Nothing prevents us from telling a computer how to interpret its inputs in such a way that its performance gradually improves. Rather than asking in advance whether it is possible for computers to "learn", it is much more enlightening to try to describe exactly what activities we mean when we say "learning" and what mechanisms could be used to enable us to perform those activities. [Simon, 1993] stated "changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time".

7.2 What is learning ?

Learning can be described as normally a relatively permanent change that occurs in behavior as a result of experience. Learning occurs in various regimes. For example, it is possible to learn to open a lock as a result of trial and error; possible to learn how to use a word processor as a result of following particular instructions.

Once the internal model of what ought to happen is set, it is possible to learn by practicing the skill until the performance converges on the desired model. One begins by paying attention to what needs to be done, but with more practice, one will need to monitor only the trickier parts of the performance.

Automatic performance of some skills by the brain points out that the brain is capable of doing things in parallel i.e. one part is devoted to the skill whilst another part mediates conscious experience.

There's no decisive definition of learning but here are some that do justice:

- "Learning denotes changes in a system that ... enables a system to do the same task more efficiently the next time." --Herbert Simon
- "Learning is constructing or modifying representations of what is being experienced." --Ryszard Michalski
- "Learning is making useful changes in our minds." --Marvin Minsky

7.3 What is machine learning ?

It is a very difficult to define precisely what machine learning is. We can best enlighten ourselves by exactly describing the activities that we want a machine to do when we say learning and by deciding on the best possible mechanism to enable us to perform those activities. Generally speaking, the goal of *machine learning* is to build computer systems that can learn from their experience and adapt to their environments. Obviously, learning is an important aspect or component of intelligence. There are both theoretical and practical reasons to support such a claim. Some people even think intelligence is nothing but the ability to learn, though other people think an intelligent system has a separate "learning mechanism" which improves the performance of other mechanisms of the system.

7.4 Why do we want machine learning

One response to the idea of AI is to say that computers can not think because they only do what their programmers tell them to do. However, it is not always easy to tell what a particular program will do, but given the same inputs and conditions it will always produce the same outputs. If the program gets something right once it will always get it right. If it makes a mistake once it will always make the same mistake every time it runs. In contrast to computers, humans learn from their mistakes; attempt to work out why things went wrong and try alternative solutions. Also, we are able to notice similarities between things, and therefore can generate new ideas about the world we live in. Any intelligence, however artificial or alien, that did not learn would not be much of an intelligence. So, machine learning is a prerequisite for any mature programme of artificial intelligence.

7.5 What are the three phases in machine learning?

Machine learning typically follows three phases according to Finlay, [Janet Finlay, 1996]. They are as follows:

1. Training: a training set of examples of correct behavior is analyzed and some representation of the newly learnt knowledge is stored. This is often some form of rules.

2. Validation: the rules are checked and, if necessary, additional training is given. Sometimes additional test data are used, but instead of using a human to validate the rules, some other automatic knowledge based component may be used. The role of tester is often called the critic.

3. Application: the rules are used in responding to some new situations.

These phases may not be distinct. For example, there may not be an explicit validation phase; instead, the learning algorithm guarantees some form of correctness. Also in some circumstances, systems learn "on the job", that is, the training and application phases overlap.

7.5.1 Inputs to training

There is a continuum between knowledge-rich methods that use extensive domain knowledge and those that use only simple domain-independent knowledge. The domain-independent knowledge is often implicit in the algorithms; e.g. inductive learning is based on the knowledge that if something happens a lot it is likely to be generally true. Where examples are provided, it is important to know the source. The examples may be simply measurements from the world, for example, transcripts of grand master tournaments. If so, do they represent "typical" sets of behavior or have they been filtered to be "representative"? If the former is true then it is possible to infer information about the relative probability from the frequency in the training set. However, unfiltered data may also be noisy, have errors, etc., and examples from the world may not be complete, since infrequent situations may simply not be in the training set.

Alternatively, the examples may have been generated by a teacher. In this case, it can be assumed that they are a helpful set which cover all the important cases. Also, it is advisable to assume that the teacher will not be ambiguous.

Finally the system itself may be able to generate examples by performing experiments on the world, asking an expert, or even using the internal model of the world.

Some form of representation of the examples also has to be decided. This may partly be determined by the context, but more often than not there will be a choice. Often the choice of representation embodies quite a lot of the domain knowledge.

7.5.2 Outputs of training

Outputs of learning are determined by the application. The question that arises is 'What is it that we want to do with our knowledge?'. Many machine learning systems are classifiers. The examples they are given are from two or more classes, and the purpose of learning is to determine the common features in each class. When a new unseen example is presented, the system uses the common features to determine which class the new example belongs to. For example:
If example satisfies condition
Then assign it to class X
This sort of job classification is often termed as concept learning. The simplest case is when there are only two classes, of which one is seen as the desired "concept" to be learnt and the other is everything else. The "then" part of the rules is always the same and so the learnt rule is just a predicate describing the concept.

Not all learning is simple classification. In applications such as robotics one wants to learn appropriate actions. In such a case, the knowledge may be in terms of production rules or some similar representation.

An important consideration for both the content and representation of learnt knowledge is the extent to which explanation may be required for future actions. Because of this, the learnt rules must often be restricted to a form that is comprehensible to humans.

7.5.3 The training process

Real learning involves some generalization from past experience and usually some coding of memories into a more compact form. Achieving this generalization needs some form of reasoning. The difference between deductive

reasoning and inductive reasoning is often used as the primary distinction between machine learning algorithms. Deductive learning working on existing facts and knowledge and deduces new knowledge from the old. In contrast, inductive learning uses examples and generates hypothesis based on the similarities between them.
One way of looking at the learning process is as a search process. One has a set of examples and a set of possible rules. The job of the learning algorithm is to find suitable rules that are correct with respect to the examples and existing knowledge.

7.6 Learning techniques available

7.6.1 Rote learning

In this kind of learning there is no prior knowledge. When a computer stores a piece of data, it is performing an elementary form of learning. This act of storage presumably allows the program to perform better in the future. Examples of correct behavior are stored and when a new situation arises it is matched with the learnt examples. The values are stored so that they are not re-computed later. One of the earliest game-playing programs is [Samuel, 1963] checkers program. This program learned to play checkers well enough to beat its creator/designer.

7.6.2 Deductive learning

Deductive learning works on existing facts and knowledge and deduces new knowledge from the old. This is best illustrated by giving an example. For example, assume:
A = B
B = C
Then we can deduce with much confidence that:
C = A
Arguably, deductive learning does not generate "new" knowledge at all, it simply memorizes the logical consequences of what is known already. This implies that virtually all mathematical research would not be classified as learning "new" things. However, regardless of whether this is termed as new knowledge or not, it certainly makes the reasoning system more efficient.

7.6.3 Inductive learning

Inductive learning takes examples and generalizes rather than starting with existing knowledge. For example, having seen many cats, all of which have tails, one might conclude that all cats have tails. This is an unsound step of reasoning but it would be impossible to function without using induction to some extent. In many areas it is an explicit assumption. There is scope of error in inductive reasoning, but still it is a useful technique that has been used as the basis of several successful systems.

One major subclass of inductive learning is concept learning. This takes examples of a concept and tries to build a general description of the concept. Very often, the examples are described using attribute-value pairs. The example of inductive learning given here is that of a fish. Look at the table below:

	herring	cat	dog	cod	whale

Swims	yes	no	no	yes	yes
has fins	yes	no	no	yes	yes
has lungs	no	yes	yes	no	yes
is a fish	yes	no	no	yes	no

In the above example, there are various ways of generalizing from examples of fish and non-fish. The simplest description can be that a fish is something that does not have lungs. No other single attribute would serve to differentiate the fish.

The two very common inductive learning algorithms are version spaces and ID3. These will be discussed in detail, later.

7.7 How is it different from the AI we've studied so far?

Many practical applications of AI do not make use of machine learning. The relevant knowledge is built in at the start. Such programs even though are fundamentally limited; they are useful and do their job. However, even where we do not require a system to learn "on the job", machine learning has a part to play.

7.7.1 Machine learning in developing expert systems?

Many AI applications are built with rich domain knowledge and hence do not make use of machine learning. To build such expert systems, it is critical to capture knowledge from experts. However, the fundamental problem remains unresolved, in the sense that things that are normally implicit inside the expert's head must be made explicit. This is not always easy as the experts may find it hard to say what rules they use to assess a situation but they can always tell you what factors they take into account. This is where machine learning mechanism could help. A machine learning program can take descriptions of situations couched in terms of these factors and then infer rules that match expert's behavior.

7.8 Applied learning

7.8.1 Solving real world problems by learning

We do not yet know how to make computers learn nearly as well as people learn. However, algorithms have been developed that are effective for certain types of learning tasks, and many significant commercial applications have begun to appear. For problems such as speech recognition, algorithms based on machine learning outperform all other approaches that have been attempted to date. In other emergent fields like computer vision and data mining, machine learning algorithms are being used to recognize faces and to extract valuable information and knowledge from large commercial databases respectively. Some of the applications that use learning algorithms include:

- Spoken digits and word recognition
- Handwriting recognition
- Driving autonomous vehicles
- Path finders
- Intelligent homes

- Intrusion detectors
- Intelligent refrigerators, tvs, vacuum cleaners
- Computer games
- Humanoid robotics

This is just the glimpse of the applications that use some intelligent learning components. The current era has applied learning in the domains ranging from agriculture to astronomy to medical sciences.

7.8.2 A general model of learning agents, pattern recognition

Any given learning problem is primarily composed of three things:

- Input
- Processing unit
- Output

The input is composed of examples that can help the learner learn the underlying problem concept. Suppose we were to build the learner for recognizing spoken digits. We would ask some of our friends to record their sounds for each digit [0 to 9]. Positive examples of digit '1' would be the spoken digit '1', by the speakers. Negative examples for digit '1' would be all the rest of the digits. For our learner to learn the digit '1', it would need positive and negative examples of digit '1' in order to truly learn the difference between digit '1' and the rest.

The processing unit is the learning agent in our focus of study. Any learning agent or algorithm should in turn have at least the following three characteristics:

7.8.2.1 Feature representation

The input is usually broken down into a number of features. This is not a rule, but sometimes the real world problems have inputs that cannot be fed to a learning system directly, for instance, if the learner is to tell the difference between a good and a not-good student, how do you suppose it would take the input? And for that matter, what would be an appropriate input to the system? It would be very interesting if the input were an entire student named Ali or Umar etc. So the student goes into the machine and it tells if the student it consumed was a good student or not. But that seems like a far fetched idea right now. In reality, we usually associate some attributes or features to every input, for instance, two features that can define a student can be: grade and class participation. So these become the feature set of the learning system. Based on these features, the learner processes each input.

7.8.2.2 Distance measure

Given two different inputs, the learner should be able to tell them apart. The distance measure is the procedure that the learner uses to calculate the difference between the two inputs.

7.8.2.3 Generalization

In the training phase, the learner is presented with some positive and negative examples from which it leans. In the testing phase, when the learner comes across new but similar inputs, it should be able to classify them similarly. This is called generalization. Humans are exceptionally good at generalization. A small child learns to differentiate between birds and cats in the early days of his/her life. Later when he/she sees a new bird, never seen before, he/she can easily tell that it's a bird and not a cat.

7.9 LEARNING: Symbol-based

Ours is a world of symbols. We use symbolic interpretations to understand the world around us. For instance, if we saw a ship, and were to tell a friend about its size, we will not say that we saw a 254.756 meters long ship, instead we'd say that we saw a 'huge' ship about the size of 'Eiffel tower'. And our friend would understand the relationship between the size of the ship and its hugeness with the analogies of the symbolic information associated with the two words used: 'huge' and 'Eiffel tower'.

Similarly, the techniques we are to learn now use symbols to represent knowledge and information. Let us consider a small example to help us see where we're headed. What if we were to learn the concept of a GOOD STUDENT. We would need to define, first of all some attributes of a student, on the basis of which we could tell apart the good student from the average. Then we would require some examples of good students and average students. To keep the problem simple we can label all the students who are "not good" (average, below average, satisfactory, bad) as NOT GOOD STUDENT. Let's say we choose two attributes to define a student: grade and class participation. Both the attributes can have either of the two values: High, Low. Our learner program will require some examples from the concept of a student, for instance:

1. Student (GOOD STUDENT): Grade (High) ^ Class Participation (High)
2. Student (GOOD STUDENT): Grade (High) ^ Class Participation (Low)
3. Student (NOT GOOD STUDENT): Grade (Low) ^ Class Participation (High)
4. Student (NOT GOOD STUDENT): Grade (Low) ^ Class Participation (Low)

As you can see the system is composed of symbolic information, based on which the learner can even generalize that a student is a GOOD STUDENT if his/her grade is high, even if the class participation is low:
Student (GOOD STUDENT): Grade (High) ^ Class Participation (?)
This is the final rule that the learner has learnt from the enumerated examples. Here the '?' means that the attribute class participation can have any value, as long as the grade is high. In this section we will see all the steps the learner has to go through to actually come up with the final conclusion like this.

7.10 Problem and problem spaces

Before we get down to solving a problem, the first task is to understand the problem itself. There are various kinds of problems that require solutions. In theoretical computer science there are two main branches of problems:

- Tractable
- Intractable

Those problems that can be solved in polynomial time are termed as tractable, the other half is called intractable. The tractable problems are further divided into structured and complex problems. Structured problems are those which have defined steps through which the solution to the problem is reached. Complex problems usually don't have well-defined steps. Machine learning algorithms are particularly more useful in solving the complex problems like recognition of patterns in images or speech, for which it's hard to come up with procedural algorithms otherwise.

The solution to any problem is a function that converts its inputs to corresponding outputs. The domain of a problem or the problem space is defined by the elements explained in the following paragraphs. These new concepts will be best understood if we take one example and exhaustively use it to justify each construct.

Example:

Let us consider the domain of HEALTH. The problem in this case is to distinguish between a sick and a healthy person. Suppose we have some domain knowledge; keeping a simplistic approach, we say that two attributes are necessary and sufficient to declare a person as healthy or sick. These two attributes are: Temperature (T) and Blood Pressure (BP). Any patient coming into the hospital can have three values for T and BP: High (H), Normal (N) and Low (L). Based on these values, the person is to be classified as Sick (SK). SK is a Boolean concept, SK = 1 means the person is sick, and SK = 0 means person is healthy. So the ***concept*** to be learnt by the system is of Sick, i.e., SK=1.

7.10.1 Instance space

How many distinct instances can the concept sick have? Since there are two attributes: T and BP, each having 3 values, there can be a total of 9 possible distinct instances in all. If we were to enumerate these, we'll get the following table:

X	T	BP	SK
x_1	L	L	-
x_2	L	N	-
x_3	L	H	-
x_4	N	L	-
x_5	N	N	-
x_6	N	H	-
x_7	H	L	-
x_8	H	N	-
x_9	H	H	-

This is the entire instance space, denoted by X, and the individual instances are denoted by x_i. |X| gives us the size of the instance space, which in this case is 9.
|X| = 9

The set X is the entire data possibly available for any concept. However, sometimes in real world problems, we don't have the liberty to have access to the entire set X, instead we have a subset of X, known as training data, denoted by D, available to us, on the basis of which we make our learner learn the concept.

7.10.2 Concept space

A concept is the representation of the problem with respect to the given attributes, for example, if we're talking about the problem scenario of concept SICK defined over the attributes T and BP, then the concept space is defined by all the combinations of values of SK for every instance x. One of the possible concepts for the concept SICK might be enumerated in the following table:

X	T	BP	SK
x_1	L	L	0
x_2	L	N	0
x_3	L	H	1
x_4	N	L	0
x_5	N	N	0
x_6	N	H	1
x_7	H	L	1
x_8	H	N	1
x_9	H	H	1

But there are a lot of other possibilities besides this one. The question is: how many total concepts can be generated out of this given situation. The answer is: $2^{|X|}$. To see this intuitively, we'll make small tables for each concept and see them graphically if they come up to the number 2^9, since |X| = 9.

The representation used here is that every box in the following diagram is populated using $C(x_i)$, i.e. the value that the concept C gives as output when x_i is given to it as input.

$C(x_3)$	$C(x_6)$	$C(x_9)$
$C(x_2)$	$C(x_5)$	$C(x_8)$
$C(x_1)$	$C(x_4)$	$C(x_7)$

Since we don't know the concept yet, so there might be concepts which can produce 2^9 different outputs, such as:

$$\underset{C_1}{\begin{matrix}0&0&0\\0&0&0\\0&0&0\end{matrix}} \quad \underset{C_2}{\begin{matrix}0&0&0\\0&0&0\\1&0&0\end{matrix}} \quad \underset{C_3}{\begin{matrix}0&0&0\\1&0&0\\0&0&0\end{matrix}} \quad \underset{C_4}{\begin{matrix}0&0&0\\1&0&0\\1&0&0\end{matrix}} \;--\; \underset{C_{2^9}}{\begin{matrix}1&1&1\\1&1&1\\1&1&1\end{matrix}} \qquad \underset{C_{2^9}}{\begin{matrix}1&1&1\\1&1&1\\1&1&1\end{matrix}}$$

Each of these is a different concept, only one of which is the true concept (that we are trying to learn), but the dilemma is that we don't know which one of the 2^9 is the true concept of SICK that we're looking for, since in real world problems we don't have all the instances in the instance space X, available to us for learning. If we had all the possible instances available, we would know the exact concept, but the problem is that we might just have three or four examples of instances available to us out of nine.

D	T	BP	SK
x_1	N	L	1
x_2	L	N	0
x_3	N	N	0

Notice that this is not the instance space X, in fact it is D: the training set. We don't have any idea about the instances that lie outside this set D. The learner is to learn the true concept C based on only these three observations, so that once it has learnt, it could classify the new patients as sick or healthy based on the input parameters.

7.10.3 Hypothesis space

The above condition is typically the case in almost all the real world problems where learning is to be done based on a few available examples. In this situation, the learner has to hypothesize. It would be insensible to exhaustively search over the entire concept space, since there are 2^9 concepts. This is just a toy problem with only 9 possible instances in the instance space; just imagine how huge the concept space would be for real world problems that involve larger attribute sets.

So the learner has to apply some hypothesis, which has either a search or the language bias to reduce the size of the concept space. This reduced concept space becomes the hypothesis space. For example, the most common language bias is that the hypothesis space uses the conjunctions (AND) of the attributes, i.e.

H = <T, BP>

H is the denotive representation of the hypothesis space; here it is the conjunction of attribute T and BP. If written in English it would mean:

H = <T, BP>:

```
IF "Temperature" = T AND "Blood Pressure" = BP
THEN
        H = 1
ELSE
        H = 0
```

Now if we fill in these two blanks with some particular values of T and B, it would form a hypothesis, e.g. for T = N and BP = N:

BP				
H	0	0	0	
N	0	1	0	
L	0	0	0	
	L	N	H	T

For h = <L, L>:

BP				
H	0	0	0	
N	0	0	0	
L	1	0	0	
	L	N	H	T

Notice that this is the C_2 we presented before in the concept space section:

0	0	0
0	0	0
1	0	0

This means that if the true concept of SICK that we wanted to learn was c_2 then the hypothesis h = <L, L> would have been the solution to our problem. But you must still be wondering what's all the use of having separate conventions for hypothesis and concepts, when in the end we reached at the same thing: C_2 = <L, L> = h. Well, the advantage is that now we are not required to look at 2^9 different concepts, instead we are only going to have to look at the maximum of 17 different hypotheses before reaching at the concept. We'll see in a moment how that is possible.

We said H = <T, BP>. Now T and BP here can take three values for sure: L, N and H, but now they can take two more values: ? and Ø. Where ? means that for any value, H = 1, and Ø means that there will be no value for which H will be 1.
For example, h_1 = <?, ?>: [For any value of T or BP, the person is sick]
Similarly h_2 = <?, N>: [For any value of T AND for BP = N, the person is sick]

BP				
H	1	1	1	
N	1	1	1	
L	1	1	1	
	L	N	H	T

BP				
H	0	0	0	
N	1	1	1	
L	0	0	0	
	L	N	H	T

h_3 = < Ø , Ø >: [For no value of T or BP, the person
[is sick]

BP				
H	0	0	0	
N	0	0	0	
L	0	0	0	
	L	N	H	T

Having said all this, how does this still reduce the hypothesis space to 17? Well it's simple, now each attribute T and BP can take 5 values each: L, N, H, ? and Ø. So there are 5 x 5 = 25 total hypotheses possible. This is a tremendous reduction from 2^9 = 512 to 25.

But if we want to represent h_4 = < Ø , L>, it would be the same as h_3, meaning that there are some redundancies within the 25 hypotheses. These redundancies are caused by Ø, so if there's this 'Ø' in the T or the BP or both, we'll have the same hypothesis h_3 as the outcome, all zeros. To calculate the number of semantically distinct hypotheses, we need one hypothesis which outputs all zeros, since it's a distinct hypothesis than others, so that's one, plus we need to know the rest of the combinations. This primarily means that T and BP can now take 4 values instead of 5, which are: L, N, H and ?. This implies that there are now 4 x 4 = 16 different hypotheses possible. So the total distinct hypotheses are: 16 + 1 = 17. This is a wonderful idea, but it comes at a vital cost. What if the true concept doesn't lie in the conjunctive hypothesis space? This is often the case. We can try different hypotheses then. Some prior knowledge about the problem always helps.

7.10.4 Version space and searching

Version space is a set of all the hypotheses that are consistent with all the training examples. When we are given a set of training examples D, it is possible that there might be more than one hypotheses from the hypothesis space that are consistent with all the training examples. By **consistent** we mean $h(x_i) = C(x_i)$. That is, if the true output of a concept [$c(x_i)$] is 1 or 0 for an instance, then the output by our hypothesis [$h(x_i)$] is 1 or 0 as well, respectively. If this is true for every instance in our training set D, we can say that the hypothesis is consistent.

Let us take the following training set D:

D	T	BP	SK
x_1	H	H	1
x_2	L	L	0
x_3	N	N	0

One of the consistent hypotheses can be h_1=<H, H >
But then there are other hypotheses consistent with D, such as h_2 = < H, ? >

BP				
H	0	0	**1**	
N	0	**0**	0	
L	**0**	0	0	
	L	N	H	**T**

Although it classifies some of the unseen instances that are not in the training set

BP				
H	0	0	**1**	
N	0	**0**	1	
L	**0**	0	1	
	L	N	H	**T**

D, different from h_1, but it's still consistent over all the instances in D. Similarly

BP				
H	1	1	**1**	
N	0	**0**	0	
L	**0**	0	0	
	L	N	H	**T**

there's another hypothesis, h_3 = < ?, H >
Notice the change in h_3 as compared to h_2, but this is again consistent with D. Version space is denoted as **$VS_{H,D} = \{h_1, h_2, h_3\}$**. This translates as: Version space is a subset of hypothesis space H, composed of h_1, h_2 and h_3, that is consistent with D.

7.11 Concept learning as search

Now that we are well familiar with most of the terminologies of machine learning, we can define the learning process in technical terms as:

"*We have to assume that the concept lies in the hypothesis space. So we* ***search*** *for a hypothesis belonging to this hypothesis space that best fits the training examples, such that the output given by the hypothesis is same as the true output of concept.*"

In short:-
Assume $C \in H$, **search** for an $h \in H$ that best fits D
Such that $\forall x_i \in D$, $h(x_i) = C(x_i)$.

The stress here is on the word 'search'. We need to somehow search through the hypothesis space.

7.11.1 General to specific ordering of hypothesis space

Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis. To illustrate the general-to-specific ordering, consider two hypotheses:
h1 = < H, H >
h2 = < ?, H >
Now consider the sets of instances that are classified positive by h1 and by h2. Because h2 imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by h1 will also be classified positive by h2. Therefore, we say that h2 is more general than h1.

So all the hypothesis in H can be ordered according to their generality, starting from < ?, ? > which is the most general hypothesis since it always classifies all the instances as positive. On the contrary we have < Ø , Ø > which is the most specific hypothesis, since it doesn't classify a single instance as positive.

7.11.2 FIND-S

FIND-S finds the maximally specific hypothesis possible within the version space given a set of training data. How can we use the general to specific ordering of hypothesis space to organize the search for a hypothesis consistent with the observed training examples? One way is to begin with the most specific possible hypothesis in H, then generalize the hypothesis each time it fails to cover an observed positive training example. (We say that a hypothesis "covers" a positive example if it correctly classifies the example as positive.) To be more precise about how the partial ordering is used, consider the FIND-S algorithm:

```
Initialize h to the most specific hypothesis in H
For each positive training instance x
    For each attribute constraint a_i in h
          If the constraint a_i is satisfied by x
               Then do nothing
          Else
                Replace a_i in h by the next more general
                constraint that is satisfied by x
Output hypothesis h
```

To illustrate this algorithm, let us assume that the learner is given the sequence of following training examples from the SICK domain:

D	T	BP	SK
x_1	H	H	1
x_2	L	L	0
x_3	N	H	1

The first step of FIND-S is to initialize *h* to the most specific hypothesis in *H*:
h = < Ø , Ø >
Upon observing the first training example (< H, H >, 1), which happens to be a positive example, it becomes obvious that our hypothesis is too specific. In particular, none of the "Ø" constraints in *h* are satisfied by this training example, so each Ø is replaced by the next more general constraint that fits this particular example; namely, the attribute values for this very training example:
h = < H , H >
This is our *h* after we have seen the first example, but this *h* is still very specific. It asserts that all instances are negative except for the single positive training example we have observed.

Upon encountering the second example; in this case a negative example, the algorithm makes no change to *h*. In fact, the *FIND-S algorithm simply ignores every negative example*. While this may at first seem strange, notice that in the current case our hypothesis *h* is already consistent with the new negative example (i.e. *h* correctly classifies this example as negative), and hence no revision is needed. In the general case, as long as we assume that the hypothesis space *H* contains a hypothesis that describes the true target concept *c* and that the training data contains no errors and conflicts, then the current hypothesis *h* can never require a revision in response to a negative example.

To complete our trace of FIND-S, the third (positive) example leads to a further generalization of *h*, this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example. The final hypothesis is:
h = < ?, H >
This hypothesis will term all the future patients which have BP = H as SICK for all the different values of T.

There might be other hypotheses in the version space but this one was the maximally specific with respect to the given three training examples. For generalization purposes we might be interested in the other hypotheses but FIND-S fails to find the other hypotheses. Also in real world problems, the training data isn't consistent and void of conflicting errors. This is another drawback of FIND-S, that, it assumes the consistency within the training set.

7.11.3 Candidate-Elimination algorithm

Although FIND-S outputs a hypothesis from H that is consistent with the training examples, but this is just one of many hypotheses from H that might fit the training data equally well. *The key idea in Candidate-Elimination algorithm is to output a description of the set of all hypotheses consistent with the training examples*. This subset of all hypotheses is actually the *version space* with respect to the hypothesis space *H* and the training examples *D*, because it contains all possible versions of the target *concept*.
The Candidate-Elimination algorithm represents the version space by storing only its most general members (denoted by *G*) and its most specific members (denoted by *S*). Given only these two sets *S* and *G*, it is possible to enumerate all members of the version space as needed by generating the hypotheses that lie between these two sets in general-to-specific partial ordering over hypotheses.

Candidate-Elimination algorithm begins by initializing the version space to the set of all hypotheses in *H*; that is by initializing the *G* boundary set to contain the most general hypothesis in *H*, for example for the SICK problem, the G_0 will be:
G_0 = {< ?, ? >}
The *S* boundary set is also initialized to contain the most specific (least general) hypothesis:
S_0 = {< Ø , Ø >}
These two boundary sets (*G* and *S*) delimit the entire hypothesis space, because every other hypothesis in *H* is both more general than S_0 and more specific than G_0. As each training example is observed one by one, the *S* boundary is made more and more general, whereas the *G* boundary set is made more and more specific, to eliminate from the version space any hypotheses found inconsistent with the new training example. After all the examples have been processed, the computed version space contains all the hypotheses consistent with these examples. The algorithm is summarized below:

```
Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

If d is a positive example
  Remove from G any hypothesis inconsistent with d
  For each hypothesis s in S that is inconsistent with d
        Remove s from S
        Add to S all minimal generalization h of s, such that
              h is consistent with d, and some member of G is more general than h
        Remove from S any hypothesis that is more general than another one in S

If d is a negative example
  Remove from S any hypothesis inconsistent with d
  For each hypothesis g in G that is inconsistent with d
        Remove g from G
        Add to G all minimal specializations h of g, such that
              h is consistent with d, and some member of S is more specific than h
        Remove from G any hypothesis that is less general than another one in S
```

The Candidate-Elimination algorithm above is specified in terms of operations. The detailed implementation of these operations will depend on the specific problem and instances and their hypothesis space, however the algorithm can be applied to any concept learning task. We will now apply this algorithm to our designed problem SICK, to trace the working of each step of the algorithm. For comparison purposes, we will choose the exact training set that was employed in FIND-S:

D	T	BP	SK
x_1	H	H	1
x_2	L	L	0
x_3	N	H	1

We know the initial values of G and S:

G_0 = {< ?, ? >}
S_0 = {< Ø, Ø >}

Now the Candidate-Elimination learner starts:
First training observation is: **d_1 = (<H, H>, 1)** [A positive example]

G_1 = G_0 = {< ?, ? >}, since <?, ?> is consistent with d_1; both give positive outputs.

Since S_0 has only one hypothesis that is < Ø, Ø >, which implies $S_0(x_1) = 0$, which is not consistent with d_1, so we have to remove < Ø, Ø > from S_1. Also, we add minimally general hypotheses from H to S_1, such that those hypotheses are consistent with d_1. The obvious choices are like <H,H>, <H,N>, <H,L>,

<N,H>……… <L,N>, <L,L>, but none of these except <H,H> is consistent with d_1. So S_1 becomes:

S_1 = {< H, H >}
G_1 = {< ?, ? >}

Second training example is: **d_2 = (<L, L>, 0)** [A negative example]

S_2 = S_1 = {< H, H>}, since <H, H> is consistent with d_2: both give negative outputs for x_2.

G_1 has only one hypothesis: < ?, ? >, which gives a positive output on x_2, and hence is not consistent, since SK(x_2) = 0, so we have to remove it and add in its place, the hypotheses which are minimally specialized. While adding we have to take care of two things; we would like to revise the statement of the algorithm for the negative examples:

"Add to *G* all minimal specializations *h* of *g*, such that
h* is consistent with *d*, and some member of *S* is more specific than *h"

The immediate one step specialized hypotheses of < ?, ? > are:

{< H, ? >, < N, ? >, < L, ? >, < ?, H >, < ?, N >, < ?, L >}
Out of these we have to get rid of the hypotheses which are not consistent with d_2 = (<L, L>, 0). We see that all of the above listed hypotheses will give a 0 (negative) output on x_2 = < L, L >, except for < L, ? > and < ?, L >, which give a 1 (positive) output on x_2, and hence are not consistent with d_2, and will not be added to G_2. This leaves us with {< H, ? >, < N, ? >, < ?, H >, < ?, N >}. This takes care of the inconsistent hypotheses, but there's another condition in the algorithm that we must take care of before adding all these hypotheses to G_2. We will repeat the statement again, this time highlighting the point under consideration:

"Add to *G* all minimal specializations *h* of *g*, such that
h is consistent with *d*, **and some member of *S* is more specific than *h***"

This is very important condition, which is often ignored, and which results in the wrong final version space. We know the current S we have is S_2, which is: S_2 = {< H, H>}. Now for which hypotheses do you think < H, H > is more specific to, out of {< H, ? >, < N, ? >, < ?, H >, < ?, N >}. Certainly < H, H > is more specific than < H, ? > and < ?, H >, so we remove < N, ? > and < ?, N >to get the final G_2:

G_2 = {< H, ? >, < ?, H >}
S_2 = {< H, H>}

Third and final training example is: **d_3 = (<N, H>, 1)** [A positive example]

We see that in G_2, < H, ? > is not consistent with d_3, so we remove it:
G_3 = {< ?, H >}

We also see that in S_2, < H, H > is not consistent with d_3, so we remove it and add minimally general hypotheses than < H, H >. The two choices we have are: < H, ? > and < ?, H >. We only keep < ?, H >, since the other one is not consistent with d_3. So our final version space is encompassed by S_3 and G_3:

G_3 = {< ?, H >}
S_3 = {< ?, H >}

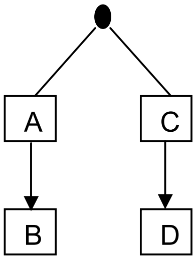
It is only a coincidence that both G and S sets are the same. In bigger problems, or even here if we had more examples, there was a chance that we'd get different but consistent sets. These two sets of G and S outline the version space of a concept. Note that the final hypothesis is the same one that was computed by FIND-S.

7.12 Decision trees learning

Up until now we have been searching in conjunctive spaces which are formed by ANDing the attributes, for instance:
IF Temperature = High **AND** Blood Pressure = High
THEN Person = SICK
But this is a very restrictive search, as we saw the reduction in hypothesis space from 2^9 total possible concepts to 17. This can be risky if we're not sure if the true concept will lie in the conjunctive space. So a safer approach is to relax the searching constraints. One way is to involve OR into the search. Do you think we'll have a bigger search space if we employ OR? Yes, most certainly; consider, for example, the statement:
IF Temperature = High **OR** Blood Pressure = High
THEN Person = SICK
If we could use these kind of OR statements, we'd have a better chance of finding the true concept, if the concept does not lie in the conjunctive space. These are also called disjunctive spaces.

7.12.1 Decision tree representation

Decision trees give us disjunctions of conjunctions, that is, they have the form: (A AND B) OR (C AND D)
In tree representation, this would translate into:



where A, B, C and D are the attributes for the problem. This tree gives a positive output if either A AND B attributes are present in the instance; OR C AND D attributes are present. Through decision trees, this is how we reach the final hypothesis. This is a hypothetical tree. In real problems, every tree has to have a root node. There are various algorithms like ID3 and C4.5 to find decision trees for learning problems.

7.12.2 ID3

ID stands for interactive dichotomizer. This was the 3rd revision of the algorithm which got wide acclaims. The first step of ID3 is to find the root node. It uses a special function GAIN, to evaluate the gain information of each attribute. For example if there are 3 instances, it will calculate the gain information for each. Whichever attribute has the maximum gain information, becomes the root node. The rest of the attributes then fight for the next slots.

7.12.2.1 Entropy

In order to define information gain precisely, we begin by defining a measure commonly used in statistics and information theory, called *entropy*, which characterizes the purity/impurity of an arbitrary collection of examples. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is:

$\text{Entropy}(S) = - p_{+}\log_2 p_{+} - p_{-}\log_2 p_{-}$

where p_{+} is the proportion of positive examples in S and p_{-} is the proportion of negative examples in S. In all calculations involving entropy we define 0log 0 to be 0.

To illustrate, suppose S is a collection of 14 examples of some Boolean concept, including 9 positive and 5 negative examples, then the entropy of S relative to this Boolean classification is:

$$\begin{aligned}\text{Entropy}(S) &= - (9/14)\log_2 (9/14) - (5/14)\log_2 (5/14)\\ &= 0.940\end{aligned}$$

Notice that the entropy is 0, if all the members of S belong to the same class (purity). For example, if all the members are positive ($p_{+} = 1$), then $p_{-} = 0$ and so:

$$\begin{aligned}\text{Entropy}(S) &= - 1\log_2 1 - 0\log_2 0\\ &= - 1\ (0) - 0 \qquad [\text{since } \log_2 1 = 0\text{, also } 0\log_2 0 = 0]\\ &= 0\end{aligned}$$

Note the entropy is 1 when the collection contains equal number of positive and negative examples (impurity). See for yourself by putting p_{+} and p_{-} equal to 1/2. Otherwise if the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1.

7.12.2.2 Information gain

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called *information gain*, is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. That is, if we use the attribute with the maximum information gain as the node, then it will classify some of the instances as positive or negative with 100% accuracy, and this will reduce the entropy for the remaining instances. We will now proceed to an example to explain further.

7.12.2.3 Example

Suppose we have the following hypothetical training data available to us given in the table below. There are three attributes: A, B and E. Attribute A can take three values: a_1, a_2 and a_3. Attribute B can take two values: b_1 and b_2. Attribute E can also take two values: e_1 and e_2. The concept to be learnt is a Boolean concept, so C takes a YES (1) or a NO (0), depending on the values of the attributes.

S	A	B	E	C
d_1	a_1	b_1	e_2	YES
d_2	a_2	b_2	e_1	YES
d_3	a_3	b_2	e_1	NO
d_4	a_2	b_2	e_1	NO
d_5	a_3	b_1	e_2	NO

First step is to calculate the entropy of the entire set S. We know:

E(S) = $- p_+\log_2 p_+ - p_-\log_2 p_-$

$-\frac{2}{5}\log_2\frac{2}{5}-\frac{2}{5}\log_2\frac{3}{5}$ = 0.97

We have the entropy of the entire training set S with us now. We have to calculate the information gain for each attribute A, B, and E based on this entropy so that the attribute giving the maximum information is to be placed at the root of the tree.

The formula for calculating the gain for A is:

$$G(S,A)=E(S)-\frac{|Sa_1|}{|S|}E(Sa_1)-\frac{|Sa_2|}{|S|}E(Sa_2)-\frac{|Sa_3|}{|S|}E(Sa_3)$$

where $|Sa_1|$ is the number of times attribute A takes the value a_1. $E(Sa_1)$ is the entropy of a_1, which will be calculated by observing the proportion of total population of a_1 and the number of times the C is YES or NO within these observation containing a_1 for the value of A.

For example, from the table it is obvious that:

$|S| = 5$

$|Sa_1| = 1$ [since there is only one observation of a_1 which outputs a YES]

$E(Sa_1) = -1\log_2 1 - 0\log_2 0 = 0$ [since log 1 = 0]

$|Sa_2| = 2$ [one outputs a YES and the other outputs NO]

$E(Sa_2) = -\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2} = -\frac{1}{2}(-1)-\frac{1}{2}(-1) = 1$

$|Sa_3| = 1$ [since there is only one observation of a_3 which outputs a NO]

$E(Sa_3) = -0\log_2 0 - 1\log_2 1 = 0$ [since log 1 = 0]

Putting all these values in the equation for *G(S,A)* we get:

$$G(S,A) = 0.97 - \frac{1}{5}(0) - \frac{2}{5}(1) - \frac{1}{5}(0) = 0.57$$

Similarly for B, now since there are only two values observable for the attribute B:

$$G(S,B) = E(S) - \frac{|Sb_1|}{|S|}E(Sb_1) - \frac{|Sb_2|}{|S|}E(Sb_2)$$

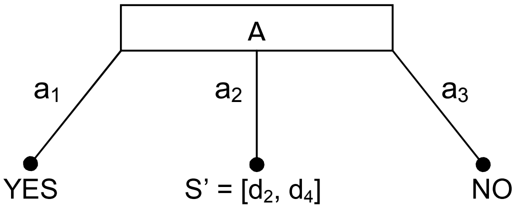
$$G(S,B) = 0.97 - \frac{2}{5}(1) - \frac{3}{5}(-\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3})$$

$$G(S,B) = 0.97 - 0.4 - \frac{3}{5}(0.52 + 0.39) = 0.02$$

Similarly for E

$$G(S,E) = E(S) - \frac{|Se_1|}{|S|}E(Se_1) - \frac{|Se_2|}{|S|}E(Se_2) = 0.02$$

This tells us that information gain for A is the highest. So we will simply choose A as the root of our decision tree. By doing that we'll check if there are any conflicting leaf nodes in the tree. We'll get a better picture in the pictorial representation shown below:



This is a tree of height one, and we have built this tree after only one iteration. This tree correctly classifies 3 out of 5 training samples, based on only one attribute A, which gave the maximum information gain. It will classify every forthcoming sample that has a value of a_1 in attribute A as YES, and each sample having a_3 as NO. The correctly classified samples are highlighted below:

S	A	B	E	C
d_1	a_1	b_1	e_2	YES
d_2	a_2	b_2	e_1	YES
d_3	a_3	b_2	e_1	NO
d_4	a_2	b_2	e_1	NO
d_5	a_3	b_1	e_2	NO

Note that a_2 was not a good determinant for classifying the output C, because it gives both YES and NO for d_2 and d_4 respectively. This means that now we have to look at other attributes B and E to resolve this conflict. To build the tree further we will ignore the samples already covered by the tree above. Our new sample space will be given by S' as given in the table below:

S'	A	B	E	C
d_2	a_2	B_2	e_1	YES
d_4	a_2	B_2	e_2	NO

We'll apply the same process as above again. First we calculate the entropy for this sub sample space S':

E(S') = $- p_+ \log_2 p_+ - p_- \log_2 p_-$

$= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}$ = 1

This gives us entropy of 1, which is the maximum value for entropy. This is also obvious from the data, since half of the samples are positive (YES) and half are negative (NO).

Since our tree already has a node for A, ID3 assumes that the tree will not have the attribute repeated again, which is true since A has already divided the data as much as it can, it doesn't make any sense to repeat A in the intermediate nodes. Give this a thought yourself too. Meanwhile, we will calculate the gain information of B and E with respect to this new sample space S':

$|S'| = 2$

$|S'b_2| = 2$

$$G(S',B) = E(S') - \frac{|S'b_2|}{|S'|}E(S'b_2)$$

$$G(S',B) = 1 - \frac{2}{2}(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2})$$ = 1 - 1 = 0

Similarly for E:

$|S'| = 2$

$|S'e_1| = 1$ [since there is only one observation of e_1 which outputs a YES]

$E(S'e_1) = -1\log_2 1 - 0\log_2 0 = 0$ [since log 1 = 0]

$|S'e_2| = 1$ [since there is only one observation of e_2 which outputs a NO]

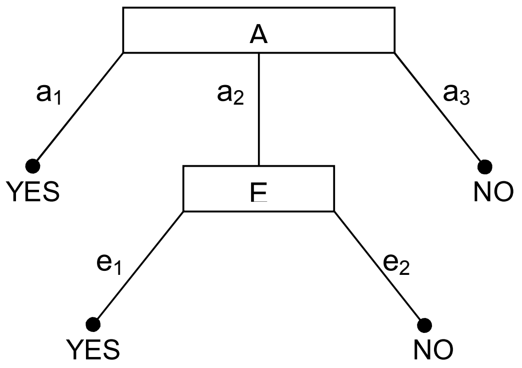
$E(S'e_2) = -0\log_2 0 - 1\log_2 1 = 0$ [since log 1 = 0]

Hence:

$$G(S',E) = E(S') - \frac{|S'e_1|}{|S'|}E(S'e_1) - \frac{|S'e_2|}{|S'|}E(S'e_2)$$

$$G(S',E) = 1 - \frac{1}{2}(0) - \frac{1}{2}(0)$$ = 1 - 0 - 0 = 1

Therefore E gives us a maximum information gain, which is also true intuitively since by looking at the table for S', we can see that B has only one value b_2, which doesn't help us decide anything, since it gives both, a YES and a NO. Whereas, E has two values, e_1 and e_2; e_1 gives a YES and e_2 gives a NO. So we put the node E in the tree which we are already building. The pictorial representation is shown below:



Now we will stop further iterations since there are no conflicting leaves that we need to expand. This is our hypothesis *h* that satisfies each training example.

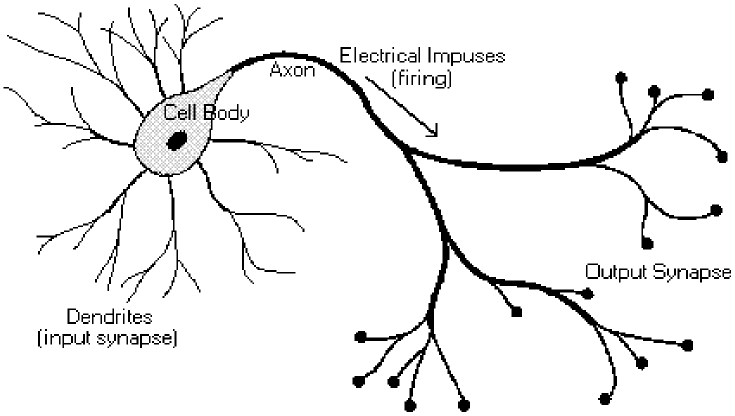
7.13 LEARNING: Connectionist

Although ID3 spanned more of the concept space, but still there is a possibility that the true concept is not simply a mixture of disjunctions of conjunctions, but some more complex arrangement of attributes. (Artificial Neural Networks) ANNs can compute more complicated functions ranging from linear to any higher order quadratic, especially for non-Boolean concepts. This new learning paradigm takes its roots from biology inspired approach to learning. Its primarily a network of parallel distributed computing in which the focus of algorithms is on training rather than explicit programming. Tasks for which connectionist approach is well suited include:

- Classification
    - Fruits – Apple or orange
- Pattern Recognition
    - Finger print, Face recognition
- Prediction
    - Stock market analysis, weather forecast

7.14 Biological aspects and structure of a neuron

The brain is a collection of about 100 billion interconnected neurons. Each neuron is a cell that uses biochemical reactions to receive, process and transmit information. A neuron's dendritic tree is connected to a thousand neighboring neurons. When one of those neurons fire, a positive or negative charge is received by one of the dendrites. The strengths of all the received charges are added together through the processes of spatial and temporal summation. Spatial summation occurs when several weak signals are converted into a single large one, while temporal summation converts a rapid series of weak pulses from one source into one large signal. The aggregate input is then passed to the soma (cell body). The soma and the enclosed nucleus don't play a significant role in the processing of incoming and outgoing data. Their primary function is to perform the continuous maintenance required to keep the neuron functional. The part of the soma that does concern itself with the signal is the axon hillock. If the aggregate input is greater than the axon hillock's threshold value, then the neuron *fires*, and an output signal is transmitted down the axon. The strength of the output is constant, regardless of whether the input was just above the threshold, or a hundred times as great. The output strength is unaffected by the many divisions in the axon; it reaches each terminal button with the same intensity it had at the axon hillock. This uniformity is critical in an analogue device such as a brain where small errors can snowball, and where error correction is more difficult than in a digital system. Each terminal button is connected to other neurons across a small gap called a synapse.



7.14.1 Comparison between computers and the brain

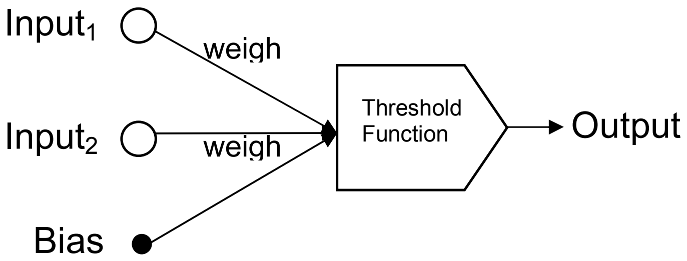
	Biological Neural Networks	Computers
Speed	Fast (nanoseconds)	Slow (milliseconds)
Processing	Superior (massively parallel)	Inferior (Sequential mode)
Size & Complexity	10^{11} neurons, 10^{15} interconnections	Far few processing elements
Storage	Adaptable, interconnection strengths	Strictly replaceable
Fault tolerance	Extremely Fault tolerant	Inherently non fault tolerant
Control mechanism	Distributive control	Central control

While this clearly shows that the human information processing system is superior to conventional computers, but still it is possible to realize an artificial neural network which exhibits the above mentioned properties. We'll start with a single perceptron, pioneering work done in 1943 by McCulloch and Pitts.

7.15 Single perceptron

To capture the essence of biological neural systems, an artificial neuron is defined as follows:

It receives a number of inputs (either from original data, or from the output of other neurons in the neural network). Each input comes via a connection that has a strength (or weight); these weights correspond to synaptic efficacy in a biological neuron. Each neuron also has a single threshold value. The weighted sum of the inputs is formed, and the threshold subtracted, to compose the activation of the neuron. The activation signal is passed through an activation function (also known as a transfer function) to produce the output of the neuron.

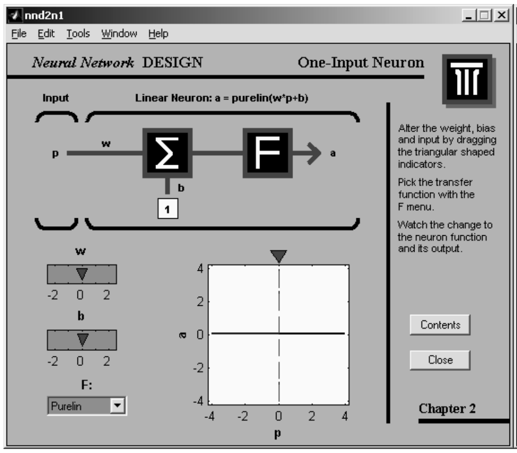


Neuron Firing Rule:

IF (Input1 x weight1) + (Input2 x weight2) + (Bias) satisfies Threshold value
Then Output = 1
Else Output = 0

7.15.1 Response of changing bias

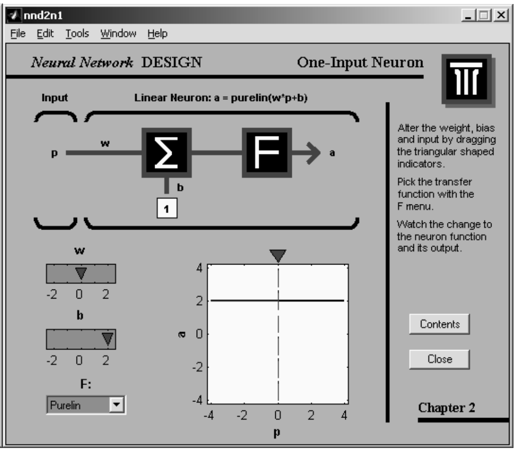
The response of changing the bias of a neuron results in shifting the decision line up or down, as shown by the following figures taken from matlab.



Alter the weight, bias and input by dragging the triangular shaped indicators.

Pick the transfer function with the F menu.

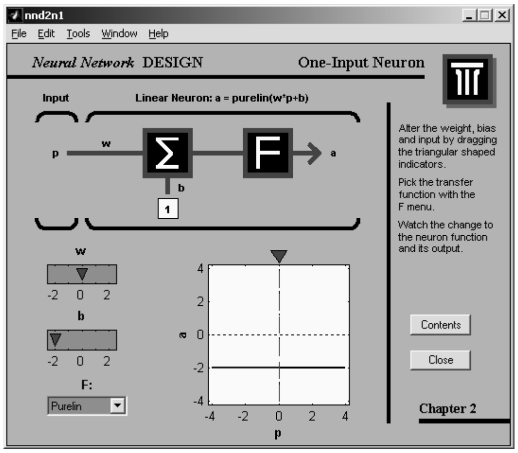
Watch the change to the neuron function and its output.



Alter the weight, bias and input by dragging the triangular shaped indicators.

Pick the transfer function with the F menu.

Watch the change to the neuron function and its output.



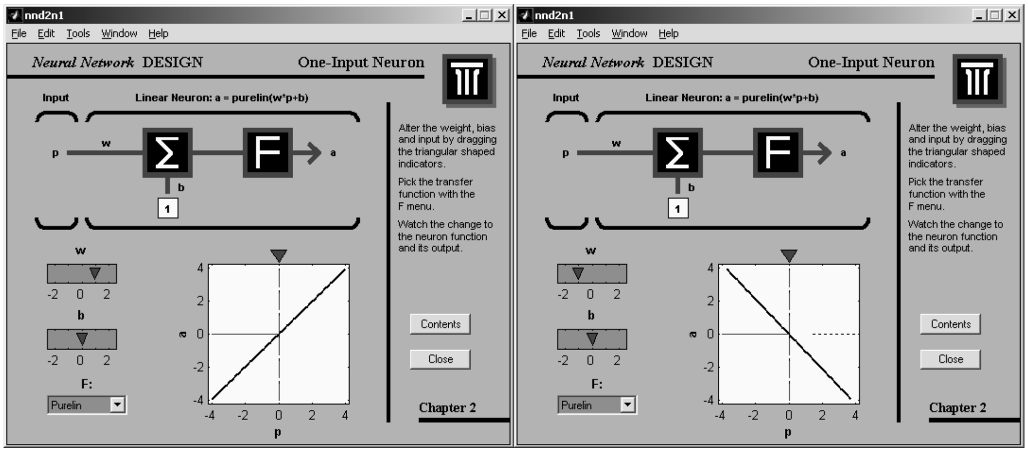
Alter the weight, bias and input by dragging the triangular shaped indicators.

Pick the transfer function with the F menu.

Watch the change to the neuron function and its output.

7.15.2 Response of changing weight

The change in weight results in the rotation of the decision line. Hence this up and down shift, together with the rotation of the straight line can achieve any linear decision.



Alter the weight, bias and input by dragging the triangular shaped indicators.

Pick the transfer function with the F menu.

Watch the change to the neuron function and its output.

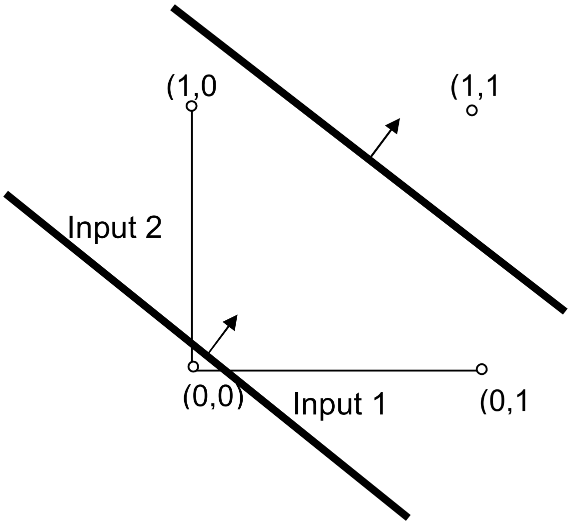
Alter the weight, bias and input by dragging the triangular shaped indicators.

Pick the transfer function with the F menu.

Watch the change to the neuron function and its output.

7.16 Linearly separable problems

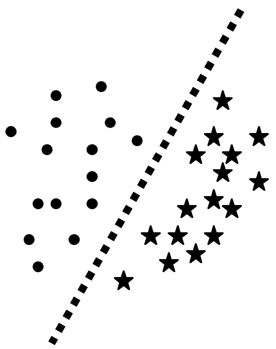
There is a whole class of problems which are termed as linearly separable. This name is given to them, because if we were to represent them in the input space, we could classify them using a straight line. The simplest examples are the logical AND or OR. We have drawn them in their input spaces, as this is a simple 2D problem. The upper sloping line in the diagram shows the decision boundary for AND gate, above which, the output is 1, below is 0. The lower sloping line decides for the OR gate similarly.



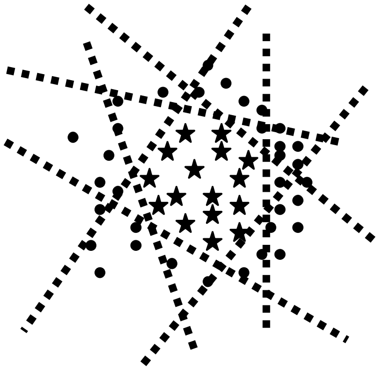
Input 1	Input 2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Input 1	Input 2	OR
0	0	0
0	1	1
1	0	1
1	1	1

A single perceptron simply draws a line, which is a hyper plane when the data is more then 2 dimensional. Sometimes there are complex problems (as is the case in real life). The data for these problems cannot be separated into their respective classes by using a single straight line. These problems are not linearly separable.

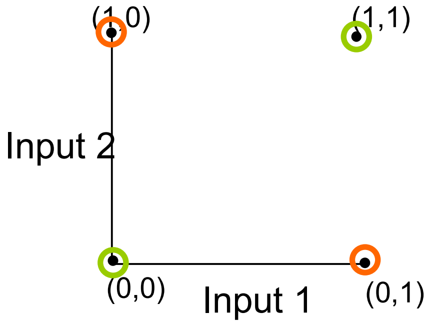


Linearly Separable



Linearly Non Separable /
Non linear decision region

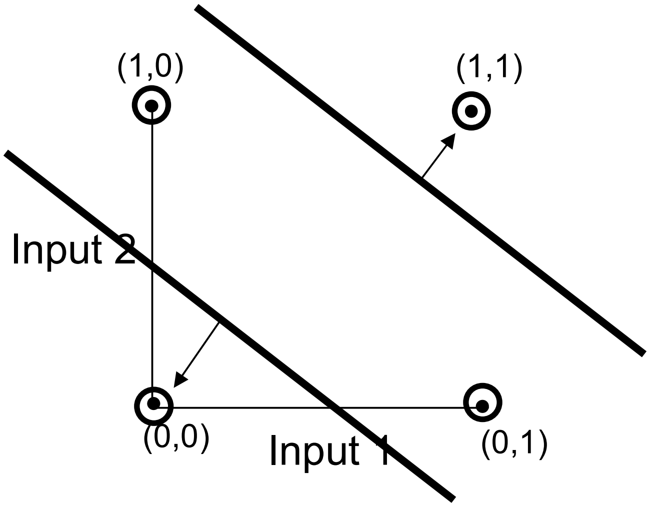
Another example of linearly non-separable problems is the XOR gate (exclusive OR). This shows how such a small data of just 4 rows, can make it impossible to draw one line decision boundary, which can separate the 1s from 0s.



Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	1

Can you draw one line which separates the ones from zeros for the output?

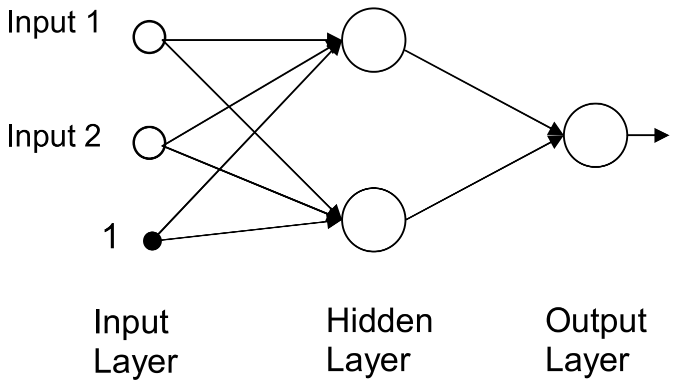
We need two lines:



A single layer perceptron can perform pattern classification only on linearly separable patterns, regardless of the type of non-linearity (hard limiter, signoidal). Papert and Minsky in 1969 illustrated the limitations of Rosenblatt's single layer perceptron (e.g. requirement of linear separability, inability to solve XOR problem) and cast doubt on the viability of neural networks. However, multi-layer perceptron and the back-propagation algorithm overcomes many of the shortcomings of the single layer perceptron.

7.17 Multiple layers of perceptrons

Just as in the previous example we saw that XOR needs two lines to separate the data without incorporating errors. Likewise, there are many problems which need to have multiple decision lines for a good acceptable solution. Multiple layer perceptrons achieve this task by the introduction of one or more hidden layers. Each neuron in the hidden layer is responsible for a different line. Together they form a classification for the given problem.



Each neuron in the hidden layer forms a different decision line. Together all the lines can construct any arbitrary non-linear decision boundaries. These multi-layer perceptrons are the most basic artificial neural networks.

7.18 Artificial Neural Networks: supervised and unsupervised

A neural network is a massively parallel distributed computing system that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

- Knowledge is acquired by the network through a learning process (called training)
- Interneuron connection strengths known as synaptic weights are used to store the knowledge

Knowledge in the artificial neural networks is implicit and distributed.

Advantages

- Excellent for pattern recognition
- Excellent classifiers
- Handles noisy data well
- Good for generalization

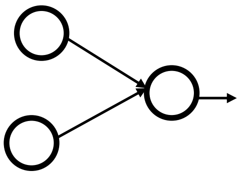
Draw backs

- The power of ANNs lie in their parallel architecture
  - Unfortunately, most machines we have are serial (Von Neumann architecture)
- Lack of defined rules to build a neural network for a specific problem
  - Too many variables, for instance, the learning algorithm, number of neurons per layer, number of layers, data representation etc
- Knowledge is implicit
- Data dependency

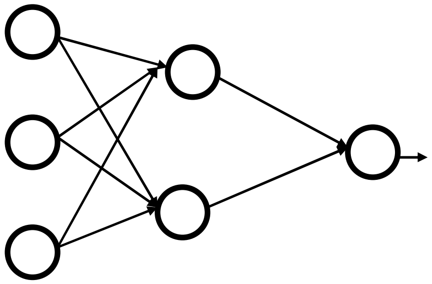
But all these drawbacks doesn't mean that the neural networks are useless artifacts. They are still arguably very powerful general purpose problem solvers.

7.19 Basic terminologies

- Number of layers
  - Single layer network
  - Multilayer networks

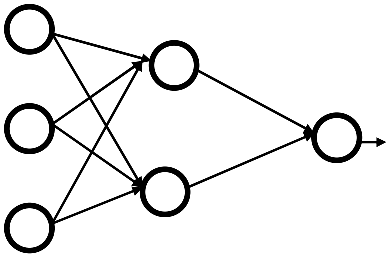


Single Layer
Only one input and
One output layer

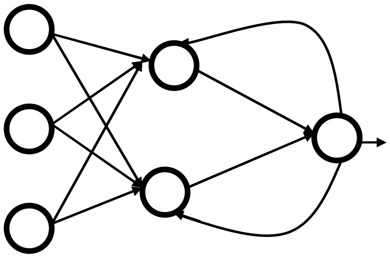


Two Layers
One input ,
One hidden and
One output layer

- Direction of information (signal) flow
    - Feed-forward
    - Recurrent (feed-back)

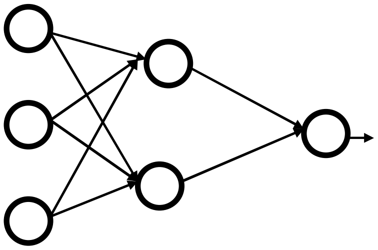


Feed forward

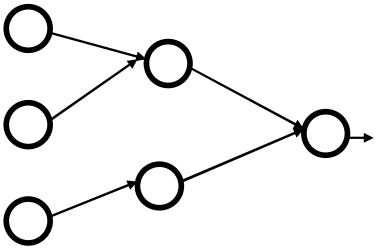


Recurrent Network

- Connectivity
    - Fully connected
    - Partially connected



Fully connected



Partially connected

- Learning methodology
    - Supervised
        - Given a set of example input/output pairs, find a rule that does a good job of predicting the output associated with a new input.
    - Unsupervised
        - Given a set of examples with no labeling, group them into sets called clusters

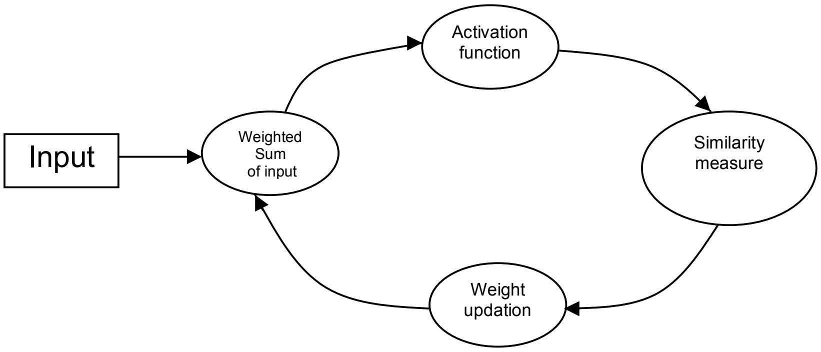
Knowledge is not explicitly represented in ANNs. Knowledge is primarily encoded in the weights of the neurons within the network

7.20 Design phases of ANNs

- Feature Representation
    - The number of features are determined using no of inputs for the problem. In many machine learning applications, there are huge number of features:
        - Text Classification (# of words)
        - Gene Arrays for DNA classification (5,000-50,000)
        - Images (512 x 512)

  - These large feature spaces make algorithms run slower. They also make the training process longer. The solution lies in finding a smaller feature space which is the subset of existing features.
  - Feature Space should show discrimination between classes of the data. Patient's height is not a useful feature for classifying whether he is sick or healthy

- Training
  - Training is either supervised or unsupervised.
  - Remember when we said:
    - We assume that the concept lies in the hypothesis space. So we **search** for a hypothesis belonging to this hypothesis space that best fits the training examples, such that the output given by the hypothesis is same as the true output of concept
    - Finding the right hypothesis is the goal of the training session. So neural networks are doing function approximation, and training stops when it has found the closest possible function that gives the minimum error on all the instances
    - Training is the heart of learning, in which finding the best hypothesis that covers most of the examples is the objective. Learning is simply done through adjusting the weights of the network



- Similarity Measurement
  - A measure to tell the difference between the actual output of the network while training and the desired labeled output
  - The most common technique for measuring the total error in each iteration of the neural network (epoch) is Mean Squared Error (MSE).

- Validation
  - During training, training data is divided into k data sets; k-1 sets are used for training, and the remaining data set is used for cross validation. This ensures better results, and avoids over-fitting.

- Stopping Criteria
    - Done through MSE. We define a low threshold usually 0.01, which if reached stops the training data.
    - Another stopping criterion is the number of epochs, which defines how many maximum times the data can be presented to the network for learning.

- Application Testing
    - A network is said to generalize well when the input-output relationship computed by the network is correct (or nearly so) for input-output pattern (test data) never used in creating and training the network.

7.21 Supervised

Given a set of example input/output pairs, find a rule that does a good job of predicting the output associated with a new input.

7.21.1 Back propagation algorithm

1. Randomize the weights {ws} to small random values (both positive and negative)
2. Select a training instance t, i.e.,
    a. the vector {xk(t)}, i = 1,...,Ninp (a pair of input and output patterns), from the training set
3. Apply the network input vector to network input
4. Calculate the network output vector {zk(t)}, k = 1,...,Nout
5. Calculate the errors for each of the outputs k , k=1,...,Nout, the difference between the desired output and the network output
6. Calculate the necessary updates for weights -ws in a way that minimizes this error
7. Adjust the weights of the network by - ws
8. **8.** Repeat steps for each instance (pair of input–output vectors) in the training set until the error for the entire system

7.22 Unsupervised

- Given a set of examples with no labeling, group them into sets called clusters
- A cluster represents some specific underlying patterns in the data
- Useful for finding patterns in large data sets
- Form clusters of input data
- Map the clusters into outputs
- Given a new example, find its cluster, and generate the associated output

7.22.1 Self-organizing neural networks: clustering, quantization, function approximation, Kohonen maps

1. Each node's weights are initialized

2. A data input from training data (vector) is chosen at random and presented to the cluster lattice
3. Every cluster centre is examined to calculate which weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU)
4. The radius of the neighborhood of the BMU is now calculated. Any nodes found within this radius are deemed to be inside the BMU's neighborhood
5. Each neighboring node's (the nodes found in step 4) weights are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered
6. Repeat steps for N iterations

7.23 Exercise

1) We will change the problem size for SICK a little bit. If T can take on 4 values, and BP can take 5 values. For conjunctive bias, determine the size of instance space and hypothesis space.
2) Is the following concept possible through conjunctive or disjunctive hypothesis? (T AND BP) or (T OR BP)

BP				
H	1	0	0	
N	0	1	0	
L	0	0	1	
	L	N	H	**T**

Appendix – MATLAB CODE

makeTrainData.m

```
trainData = zeros(21, 100);
tempImage = zeros(10,10);

for i = 1:7
   filename = strcat('alif', int2str(i),'.bmp');
   tempImage = imread(filename);
   trainData(i,:) = reshape(tempImage,1,100);
end

for i = 1:7
   filename = strcat('bay', int2str(i),'.bmp');
   tempImage = imread(filename);
   trainData(i+7,:) = reshape(tempImage,1,100);
end

for i = 1:7
   filename = strcat('jeem', int2str(i),'.bmp');
   tempImage = imread(filename);
   trainData(i+14,:) = reshape(tempImage,1,100);
end

targetData = zeros(21,3);

targetData(1:7,1) = 1;
targetData(8:14,2) = 1;
targetData(15:21,3) = 1;
```

```
save 'trainData' trainData targetData ;
```

makeTestData.m

```
testData = zeros(9, 100);
tempImage = zeros(10,10);

for i = 1:3
   filename = strcat('alif', int2str(i),'.bmp');
   tempImage = imread(filename);
   testData(i,:) = reshape(tempImage,1,100);
end

for i = 1:3
   filename = strcat('bay', int2str(i),'.bmp');
   tempImage = imread(filename);
   testData(i+3,:) = reshape(tempImage,1,100);
end

for i = 1:3
   filename = strcat('jeem', int2str(i),'.bmp');
   tempImage = imread(filename);
   testData(i+6,:) = reshape(tempImage,1,100);
end

targetData = zeros(9,3);

targetData(1:3,1) = 1;
targetData(4:6,2) = 1;
targetData(7:9,3) = 1;

save 'testData' testData targetData;
```

trainNN.m

```
load 'trainData.mat';

minMax = [min(trainData) ; max(trainData)]';


bpn = newff(minMax, [10 3],{'tansig' 'tansig'});

bpn.trainParam.epochs = 15;
bpn.trainParam.goal = 0.01;

bpn = train(bpn,trainData',targetData');

save 'bpnNet' bpn;
```

testNN.m

```
load('trainData');
load('bpnNet');
Y = sim(bpn, trainData');

[X,I] = max(Y);

errorCount = 0;
for i = 1 : length(targetData)
   if  ceil(i/7) ~= I(i)
      errorCount = errorCount + 1;
   end
end

percentageAccuracyOnTraining = (1-(errorCount/length(targetData))) * 100

%%%%%%%%%%%%%%%%%%%%%%%%%%%

load('testData');
Y = sim(bpn, testData');

[X,I] = max(Y);

errorCount = 0;
for i = 1 : length(targetData)
   if  ceil(i/3) ~= I(i)
      errorCount = errorCount + 1;
   end
end

percentageAccuracyOnTesting = (1-(errorCount/length(targetData))) * 100
```

8 Planning

8.1 Motivation

We started study of AI with the classical approach of problem solving that founders of AI used to exhibit intelligence in programs. If you look at problem solving again you might now be able to imagine that for realistically complex problems too problem solving could work. But when you think more you might guess that there might be some limitation to this old approach.

Lets take an example. I have just landed on Lahore airport as a cricket-loving tourist. I have to hear cricket commentary live on radio at night at a hotel where I have to reserve a room. For doing that, I have to find the hotel to get my room reserved before its too late, and also I have to find the market to buy the radio from. Now this is a more realistic problem. Is this a tougher problem? Let’s see.

One thing easily visible is that this problem can be broken into multiple problems i.e. is composed of small problems like finding market and finding the hotel. Another observation is that different things are dependent on others like listening to radio is dependent upon the sub-problem of buying the radio or finding the market.

Ignore the observations made above for a moment. If we start formulating this problem as usual, be assured that the state design will have more information in it. There will be more operators. Consequently, the search tree we generate will be much bigger. The poor system that will run this search will have much more load than any of the examples we have studied so far. The search tree will consume more space and it will take more calculations in the process.

A state design and operators for the sample problem formulation could be as shown in figure.

Location
Has radio?
Sells radio?
IsHotel?
IsMarket?
Reservation done?
And maybe more…

Turn right
Turn left
Move forward
Buy radio
Get reservation
Listen radio
Sleep
And maybe more…

Initial state

Operators

Figure – Sample problem formulation

If we apply say, BFS in this problem the tree can easily become something huge like this rough illustration.

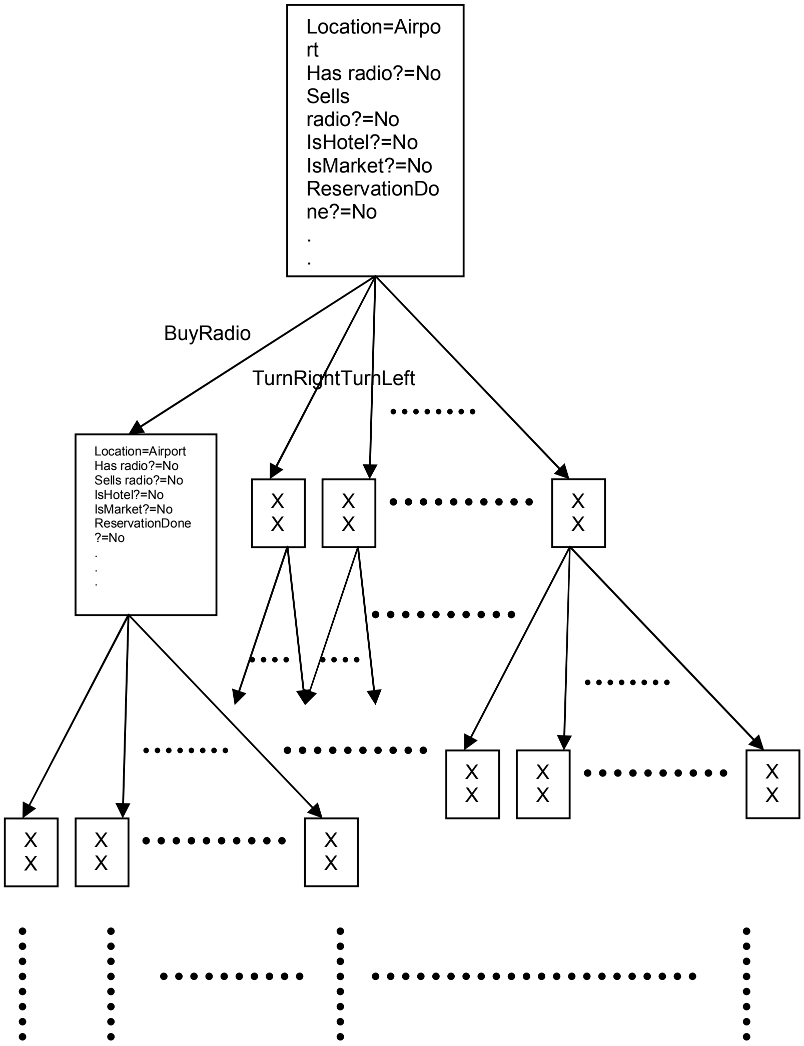


Figure – Search space of a moderate problem

Although this tree is just a depiction of how a search space grows for realistic problems, yet after seeing this tree we can very well imagine for even more complex problems that the search tree could be too big, big enough to trouble us. So the question is, can we make such inefficient problem solving any better?

Good news is that the answer is yes. How? Simply speaking, this 'search' technique could be improved by acting a bit logically instead of blindly. For example not using operators at a state where their usage is illogical. Like operator 'sleeping' should not be even tried to generate children nodes from a state where I am not at the hotel, or even haven't reserved the room.

The field of acting logically to solve problems is known as Planning. Planning is based on logic representation that we have already studied, so you will not find it too difficult and thus we have kept it short.

8.2 Definition of Planning

The key in planning is to use logic in order to solve problem elegantly. People working in AI have devised different techniques and algorithms for planning. We will now introduce a basic definition of planning.

Planning is an advanced form problem solving which generates a sequence of operators that guarantee the goal. Furthermore, such sequence of operators or actions (commonly used in planning literature) is called a plan.

8.3 Planning vs. problem solving

Planning introduces the following improvements with respect to classical problem solving:

- Each state is represented in predicate logic. De-facto representation of a state is the conjunction (AND) of predicates that are true in that state.
- The goal is also represented as states, i.e. conjunction of predicates.
- Each action (or operator) is associated with some logic preconditions that must be true for that action to be applied. Thus a planning system can avoid any action that is just not possible at a particular state.
- Each action is associated with an 'effect' or post-conditions. These post-conditions specify the added and/or deleted predicates when the action is applied.
- The inference mechanism used is that of backward chaining so as to use only the actions and states that are really required to reach goal state.
- Optional: The sequence of actions (plan) is minimally ordered. Only those actions are ordered in a sequence when any other order will not achieve the desired goal. Therefore, planning allows partial ordering i.e. there can be two actions that are not in any order from each other because any particular order used amongst them will achieve the same goal.

8.4 Planning language

STRIPS is one of the founding languages developed particularly for planning. Let us understand planning to a better level by seeing what a planning language can represent.

8.4.1 Condition predicates

Condition predicates are the predicates that define states. For example, a predicate that specifies that we are at location 'X' is given as.

```
at(X)
```

8.4.2 State

State is a conjunction of predicates represented in well-known form, for example, a state where we are at the hotel and do not have either cash or radio is represented as,

```
at(hotel) ∧ ¬have(cash) ∧ ¬have(radio)
```

8.4.3 Goal

Goal is also represented in the same manner as a state. For example, if the goal of a planning problem is to be at the hotel with radio, it is represented as,

```
at(hotel) ∧ have(radio)
```

8.4.4 Action Predicates

Action is a predicate used to change states. It has three components namely, the predicate itself, the pre-condition, and post-condition predicates. For example, the action to buy something item can be represented as,

Action:
`buy(X)`

Pre-conditions:
`at(Place) ∧ sells(Place, X)`

Post-conditions/Effect:
`have(X)`

What this example action says is that to buy any item 'X', you have to be (pre-conditions) at a place 'Place' where 'X' is sold. And when you apply this operator i.e. buy 'X', then the consequence would be that you have item 'X' (post-conditions).

8.5 The partial-order planning algorithm – POP

Now that we know what planning is and how states and actions are represented, let us see a basic planning algorithm POP.

```
POP(initial_state, goal, actions) returns plan
  Begin
  Initialize plan 'p' with initial_state linked to goal state with two
  special actions, start and finish
  Loop until there is not unsatisfied pre-condition
    Find an action 'a' which satisfies an unachieved pre-condition of
    some action 'b' in the plan
    Insert 'a' in plan linked with 'b'
    Reorder actions to resolve any threats
  End
```

If you think over this algorithm, it is quite simple. You just start with an empty plan in which naturally, no condition predicate of goal state is met i.e. pre-conditions of finish action are not met. You backtrack by adding actions that meet these unsatisfied pre-condition predicates. New unsatisfied preconditions will be generated for each newly added action. Then you try to satisfy those by using appropriate actions in the same way as was done for goal state initially. You keep on doing that until there is no unsatisfied precondition.

Now, at some time there might be two actions at the same level of ordering of them one action's effect conflicts with other action's pre-condition. This is called a threat and should be resolved. Threats are resolved by simply reordering such actions such that you see no threat.

Because this algorithm does not order actions unless absolutely necessary it is known as a partial-order planning algorithm.

Let us understand it more by means of the example we discussed in the lecture from [??].

8.6 POP Example

The problem to solve is of shopping a banana, milk and drill from the market and coming back to home. Before going into the dry-run of POP let us reproduce the predicates.

The condition predicates are:

```
At(x)
Has (x)
Sells (s, g)
Path (s, d)
```

The initial state and the goal state for our algorithm are formally specified as under.

Initial State:

```
At(Home) ∧ Sells (HWS, Drill) ∧ Sells (SM, Banana) ∧ Sells (SM, Milk) ∧
Path (home, SM) ∧ path (SM, HWS) ∧ Path (home, HWS)
```

Goal State:

```
At (Home) ∧ Has (Banana) ∧ Has (Milk) ∧ Has (Drill)
```

The actions for this problem are only two i.e. buy and go. We have added the special actions start and finish for our POP algorithm to work. The definitions for these four actions are.

```
Go (x)
Preconditions: at(y) ∧ path(y,x)
Postconditions: at(x) ∧ ~at(y)

Buy (x)
Preconditions: at(s) ∧ sells (s, x)
Postconditions: has(x)

Start ()
Preconditions: nill
Postconditions: At(Home) ∧ Sells (HWS, Drill) ∧ Sells (SM, Banana) ∧
Sells (SM, Milk) ∧ Path (home, SM) ∧ path (SM, HWS) ∧ Path (home, HWS)

Finish ()
Preconditions: At (Home) ∧ Has (Banana) ∧ Has (Milk) ∧ Has (Drill)
```

```
Postconditions: nill
```

Note the post-condition of the start action is exactly our initial state. That is how we have made sure that our end plan starts with the initial state configuration given. Similarly note that the pre-conditions of finish action are exactly the same as the goal state. Thus we can ensure that this plan satisfies all the conditions of the goal state. Also note that naturally there is no pre-condition to start and no post-condition for finish actions.

Now we start the algorithm by just putting the start and finish actions in our plan and linking them. After this first initial step the situation becomes as follows.

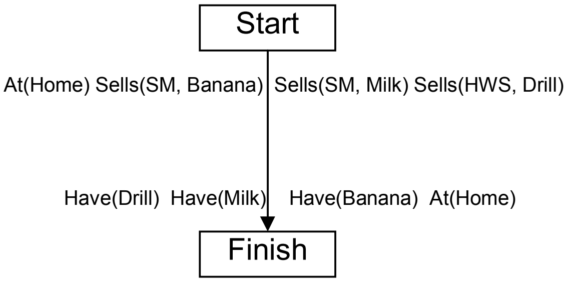

Start
At(Home) Sells(SM, Banana) Sells(SM, Milk) Sells(HWS, Drill)
Have(Drill) Have(Milk) Have(Banana) At(Home)
Finish

Figure – Initial plan scene A

We now enter the main loop of POP algorithm where we iteratively find any unsatisfied pre-condition in our existing plan and then satisfying it by an appropriate action.

At first you see three unsatisfied predicates Have(Drill), Have(Milk) and Have(Banana). Lets take Have(Drill) first. Have(Drill) matches the post-condition Have(X) of action Buy(X), where X becomes Drill in this case. Similarly we can satisfy the other two condition predicates and the resulting plan has three new actions added as shown below.

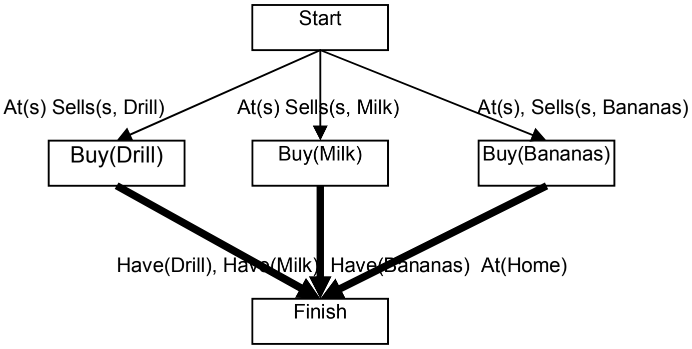


Figure – Plan scene B

There is no threat visible in the current plan, so no re-ordering is required.

The algorithm moves forward. Now if you see the Sells() pre-conditions of the three new actions, they are satisfied with the post-conditions Sells(HWS,Drill), Sells(SM,Banana), and Sells(SM,Milk) of the Start() action with the exact values as shown.

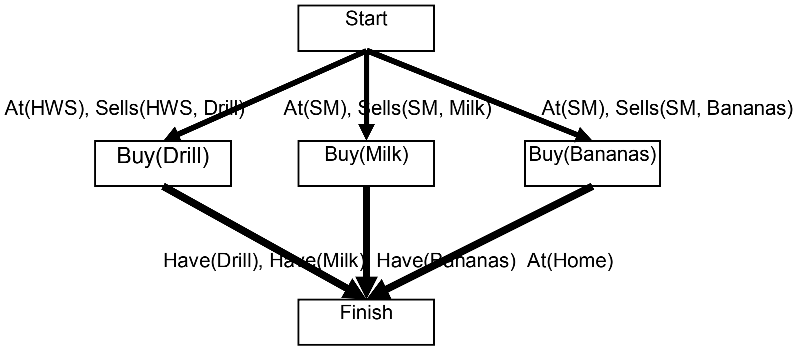

Start
At(HWS), Sells(HWS, Drill)
At(SM), Sells(SM, Milk)
At(SM), Sells(SM, Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas) At(Home)
Finish

Figure – Plan scene C

We now move forward and see what other pre-conditions are not satisfied. At(HWS) is not satisfied in action Buy(Drill). Similarly At(SM) is not satisfied in actions Buy(Milk) and Buy(Banana). Only action Go() has post-conditions that can satisfy these pre-conditions. Adding them one-by-one to satisfy all these pre-conditions our plan becomes,

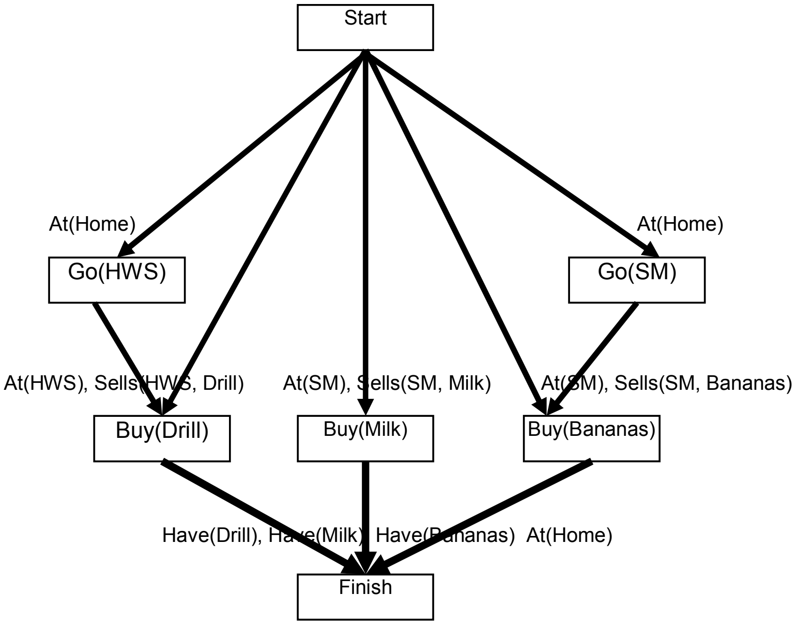

Start
At(Home)
At(Home)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS, Drill)
At(SM), Sells(SM, Milk)
At(SM), Sells(SM, Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill), Have(Milk), Have(Bananas) At(Home)
Finish

Figure – Plan scene D

Now if we check for threats we find that if we go to HWS from Home we cannot go to SM from Home. Meaning, post-condition of Go(HWS) threats the pre-condition At(Home) of Go(SM) and vice versa. So as given in our POP algorithm, we have to resolve the threat by reordering these actions such that no action threat pre-conditions of other action.

That is how POP proceeds by adding actions to satisfy preconditions and reordering actions to resolve any threat in the plan. The final plan using this algorithm becomes.

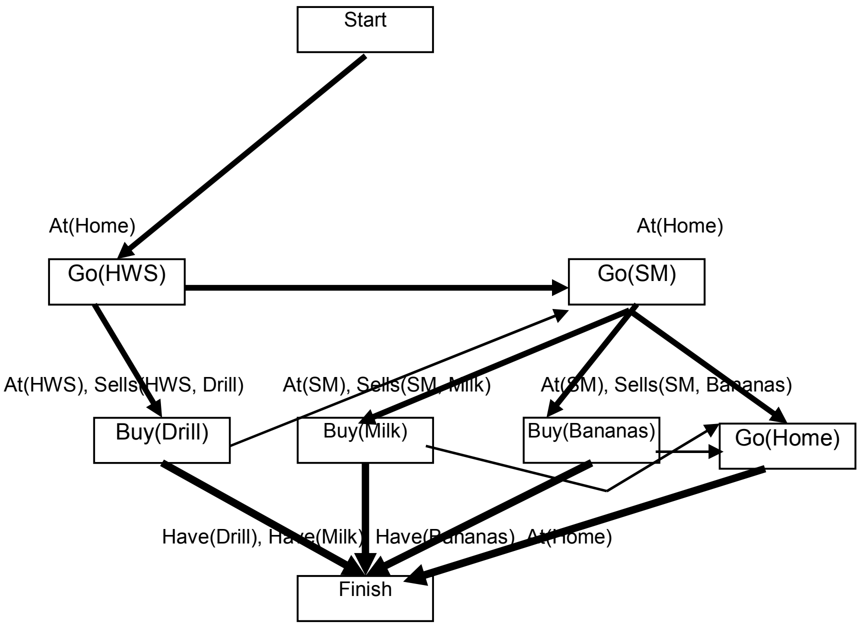


Figure – Plan scene E

You can see how reordering is done from this illustration. For example the threat we observed between Go(HWS) and Go(SM), the link from Start to Go(SM) has been deleted and a new links have been established from Go(HWS) and Buy(Drill) to Go(SM).

To feel more comfortable on the plan we have achieved from this problem, lets narrate our solution in plain English.

“Start by going to hardware store. Then you can buy drill and then go to the super market. At the super market, buy milk and banana in any order and then go home. You are done.”

8.7 Problems

1. A Farmer has a tiger, a goat and a bundle of grass. He is standing at one side of the river with a very week boat which can hold only one of his belongings at a time. His goal is has to take all three of his belongings to the other side. The constraint is that the farmer cannot leave either goat and tiger, or goat and grass, at any side of the river unattended because one of them will eat the other. Using the simple POP algorithm we studied in the lecture, solve this problem. Show all the intermediate and final plans step by step.

2. A robot has three slots available to put the blocks A, B, C. The blocks are initially placed at slot 1, one upon the other (A placed on B placed on C) and it’s goal is to move all three to slot 3 in the same order. The constraint to this robot is that it can only move one block from any slot to any other slot, and it can only pick the top most block from a slot to move. Using the simple POP algorithm we studied in the lecture, solve this problem. Show all the intermediate and final plans step by step.

9 Advanced Topics

9.1 Computer vision

It is a subfield of Artificial Intelligence. The purpose of computer vision is to study algorithms, techniques and applications that help us make machines that can **"understand"** images and videos. In other words, it deals with procedures that extract useful information from static pictures and sequence of images. Enabling a machine to see, percieve and understand exactly as humans see, percieve and understand is the aim of Computer Vision.

Computer vision finds its applications in medicine, military, security and surveillance, quality inspection, robotics, automotive industry and many other areas. Few areas of vision in which research is benig actively conducted throughout the world are as follows:

- The **detection**, **segmentation**, **localisation**, and **recognition** of certain objects in images (e.g., human faces)
- **Tracking** an object through an image sequence
- **Object Extraction** from a video sequence
- **Automated Navigation** of a robot or a vehicle
- **Estimation** of the three-dimensional **pose of humans** and their **limbs**
- **Medical Imaging**, automated analysis of different body scans (CT Scan, Bone Scan, X-Rays)
- **Searching** for digital images by their content (**content-based image retrieval**)
- **Registration** of different views of the same scene or object

Computer vision encompases topics from pattern recognition, machine learning, geometry, image processing, artificial intelligence, linear algebra and other subjects.

Apart from its applications, computer vision is itself interesting to study. Many detailed turorials regarding the field are freely avalible on the internet. Readers of this text are encouraged to read through these tutorials get indepth knowledge about the limits and contents of the field.

Exercise Question

Search through the internet and read about interesting happenings and research going on around the globe in the area of Computer Vision.

http://www.cs.ucf.edu/~vision/

The above link might be useful to explore knowledge about computer vision.

9.2 Robotics

Robotics is the highly advanced and totally hyped field of today. Literally speaking, robotics is the study of robots. Robots are nothing but a complex combination of hardware and intelligence, or mechanics and brains. Thus robotics is truly a multi-disciplinary area, having active contributions from, physics, mechanics, biology, mathematics, computer science, statistics, control thory, philosophy, etc.

The features that constitute a robot are:

- Mobility
- Perception
- Planning
- Searching
- Reasoning
- Dealing with uncertainty
- Vision
- Learning
- Autonomy
- Physical Intelligence

What we can see from the list is that robotics is the most profound manifestation of AI in practice. The most crucial or defining ones from the list above are mobility, autonomy and dealing with uncertainety

The area of robotics have been followed with enthusiasm by masses from fiction, science and industry. Now robots have entered the common household, as robot pets (Sony Aibo entertainment robot), oldage assistant and people carriers (Segway human transporter).
Exercise Question

Search through the internet and read about interesting happenings and reseach going on around the globe in the area of robotics.

http://www.cs.dartmouth.edu/~brd/Teaching/AI/Lectures/Summaries/robotics.html

The above link might be useful to explore knowledge about robotics.

9.2.1 Softcomputing

Softcomputing is a relatively new term coined to encapsulate the emergence of new hybrid area of work in AI. Different technologies including fuzzy systems, genetic algorithms, neural networks and a few statistical methods have been combined together in different orientations to successfully solve today's complex real-world problems.

The most common combinations are of the pairs

- genetic algorithms – fuzzy systems (genetic fuzzy)
- Neural Networks – fuzzy systems (neuro-fuzzy systems)
- Genetic algorithms – Neural Networks (neuro-genetic systems)

Softcomputing is naturally applied in machine learning applications. For example one usage of genetic-fuzzy system is of 'searching' for an acceptable fuzzy system that conforms to the training data. In which, fuzzy sets and rules combined, are encoded as individuals, and GA iterations refine the individuals i.e. fuzzy system, on the basis of their fitness evaluations. The fitness function is usually MSE of the invidual fuzzy system on the training data. Very similar applications have been developed in the other popular neuro-fuzzy systems, in which neural networks are used to find the best fuzzy system for the given data through means of classical ANN learning algorithms.

Genetic algorithms have been employed in finding the optimal initial weights of neural networks.

Exercise Question

Search through the internet and read about interesting happenings and research going on around the globe in the area of softcomputing.

http://www.soft-computing.de/

The above link might be useful to explore knowledge about softcomputing.

9.3 Clustering

Clustering is a form of unsupervised learning, in which the training data is available but without the classification information or class labels. The task of clustering is to identify and group similar individual data elements based on some measure of similarity. So basically using clustering algorithms, classification information can be 'produced' from a training data which has no classification data at the first place. Naturally, there is no supervision of classification in clustering algorithms for their learning/clustering, and hence they fall under the category of unsupervised learning.

The famous clustering algorithms are Self-organizing maps (SOM), k-means, linear vector quantization, Density based data analysis, etc.

Exercise Question

Search through the internet and read about interesting happenings and research going on around the globe in the area of clustering.

http://www.elet.polimi.it/upload/matteucc/Clustering/tutorial_html/

The above link might be useful to explore knowledge about clustering.

10 Conclusion

We have now come to the end of this course and we have tried to cover all the core technologies of AI at the basic level. We hope that the set of topics we have studied so far can give you the essential base to work into specialized, cutting-edge areas of AI.

Let us recap what have we studied and concluded so far. The list of major topics that we covered in the course is:

- Introduction to intelligence and AI
- Classical problem solving
- Genetic algorithms
- Knowledge representation and reasoning
- Expert systems
- Fuzzy systems
- Learning
- Planning
- Advanced topics

Let us review each of them very briefly.

10.1 Intelligence and AI

Intelligence is defined by some characteristics that are common in different intelligent species, including problem solving, uncertainty handling, planning, perception, information processing, recognition, etc.
AI is classified differently by two major schools of thought. One school classifies AI as study of systems that think like humans i.e. strong AI and the other classifies AI as study of systems that act like humans i.e. weak AI. Most of the techniques prevalent today are counted in the latter classification.

10.2 Problem solving

Many people view AI as nothing but problem solving. Early work in AI was done around the generic concept of problem solving, starting with the basic technique of generate and test. Although such classical problem solving did not get extraordinary success but still it provided a conceptual backbone for almost each approach to the systematic exploration of alternatives.
The basic technique used in classical problem solving is searching. There are several algorithms for searching for problem solving, including BFS, DFS, hill climbing, beam search, A* etc. broadly categorized on the basis of completeness, optimality and informed ness. A special branch of problem solving through

searching involved adversarial problems like classical two-player games, handled in classical problem solving by adversarial search algorithms like Minimax.

10.3 Genetic Algorithms

Genetic algorithms is a modern advancement to the hill climbing search based problem solving. Genetic algorithms are inspired by the biological theory of evolution and provide facilities of parallel search agents using collaborative hill climbing. We have seen that many otherwise difficult problems to solve through classical programming or blind search techniques are easily but undeterministically solved using genetic algorithms.

At this point we introduced the cycle of AI to set base for systematic approach to study contemporary techniques in AI.

10.4 Knowledge representation and reasoning

Reasoning has been presented by most researchers in AI as the core ability of an intelligent being. By nature, reasoning is tightly coupled with knowledge representation i.e. the reasoning process must exactly know how the knowledge is kept to manipulate and extract new knowledge from it.

As we are yet to decode the exact representation of knowledge in natural intelligent beings like humans, we have based our knowledge representation and hence reasoning on man-made logical representation namely logic i.e. predicate logic and family.

10.5 Expert systems

The first breakthrough successful application of AI came from the subject of knowledge representation and reasoning and was name expert systems. Based on its components i.e. knowledge base, inference and working memory, expert systems have been successfully applied to diagnosis, interpretation, prescription, design, planning, simulations, etc.

10.6 Fuzzy systems

Predicate logic and the classical and successful expert systems were limited in that they could only deal with perfect boolean logic alone. Fuzzy logic provided the new base of knowledge and logic representation to capture uncertain information and thus fuzzy reasoning systems were developed. Just like expert systems, fuzzy systems have almost recently found exceptional success and are one of the most used AI systems of today, with applications ranging from self-focusing cameras to automatic intelligent stock trading systems.

10.7 Learning

Having covered the core intelligence characteristic of reasoning, we shifted to the other major half contributed to AI i.e. learning or formally machine learning. The KRR and fuzzy systems perform remarkably but they cannot add or improve their knowledge at all, and that is where learning was felt essential i.e. the ability of knowledge based systems to improve through experience.

Learning has been categorized into rote, inductive and deductive learning. Out of these all almost all the prevalent learning techniques are attributed to inductive learning, including concept learning, decision tree learning and neural networks.

10.8 Planning

In the end we have studied a rather specialized part of AI namely planning. Planning is basically advancement to problem solving in which concepts of KRR are fused with the knowledge of classical problem solving to construct advanced systems to solve reasonably complex real world problems with multiple, interrelated and unrelated goals. We have learned that using predicate logic and regression, problems could be elegantly solved which would have been nightmare for machines in case of classical problem solving approach.

10.9 Advanced Topics

You have been given just a hint of where the field of AI is moving by mentioning some of the exciting areas of AI of today including vision, robotics, soft-computing and clustering. Of these we saw robotics as the most comprehensive field in which the other topics like vision can be considered as a sub-part.

Now, it's up to you to take these thoughts and directions along with the basics and move forward into advanced study and true application of the field of Artificial Intelligence.