J.B. INSTITUTE OF ENGINEERING AND TECHNOLOGY

(UGC AUTONOMOUS)

Bhaskar Nagar, Moinabad Mandal, R.R. District, Hyderabad -500075

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLEGENCE

LECTURE NOTES

R20

B. TECH
III YEAR - I SEM

Prepared & Compiled by

G. Swapna

ASSISTANT PROFESSOR, DEPARTMENT OF CSEJ.B.I.E.T

Bhaskar Nagar, Yenkapally(V), Moinabad(M), Ranga Reddy(D), Hyderabad - 500 075, Telangana, India.

AY 2020-21 onwards	J. B. Institute of Engineering and Technology (UGC Autonomous)	B.Tech: CSE III Year – I Sem			
Course Code: J31M1	**ARTIFICIAL INTELLIGENCE** (Common to CSE, IT, EEE& ECM in III-I) and (Common to ECE, CIVIL, ME & MIE in III-II)	**L**	**T**	**P**	**D**
Credits: 0		**2**	**0**	**0**	**0**

Pre-Requisites:

1. Mathematics, Probability and statistics
2. Knowledge in programming Language

Course objectives:

The Student will:

1. Know the AI based problems.
2. Illustrate AI techniques for representing the basic problem.
3. Illustrate Advanced AI techniques to solve the problem.
4. Define Learning and explain various learning techniques.
5. Understand the usage expert system.

Module 1:

Introduction: AI problems, Agents and Environments, Structure of Agents, Problem Solving Agents

Basic Search Strategies: Problem Spaces, Uninformed Search (Breadth-First, Depth-First Search, Depth-first with Iterative Deepening), Heuristic Search (Hill Climbing, Generic Best-First, A*), Constraint Satisfaction (Backtracking, Local Search)

Module 2:

Advanced Search: Constructing Search Trees, Stochastic Search, A* Search Implementation, Minimax Search, Alpha-Beta Pruning

Basic Knowledge Representation and Reasoning: Propositional Logic, First-Order Logic, Forward Chaining and Backward Chaining, Introduction to Probabilistic Reasoning, Bayes Theorem.

Module 3:

Advanced Knowledge Representation and Reasoning: Knowledge Representation Issues, Non- monotonic Reasoning, Other Knowledge Representation Schemes.

Reasoning Under Uncertainty: Basic probability, Acting Under Uncertainty, Bayes' Rule, Representing Knowledge in an Uncertain Domain, Bayesian Networks.

Module 4:

Learning: What Is Learning? Rote Learning, Learning by Taking Advice, Learning in Problem Solving, Learning from Examples, Winston's Learning Program, Decision Trees.

Module 5:

Expert Systems: Representing and Using Domain Knowledge, Shell, Explanation,Knowledge Acquisition.

Text Books:

1. Russell, S. and Norvig, P, Artificial Intelligence: A Modern Approach, ThirdEdition, Prentice- Hall, 2010

References Books:

1. Artificial Intelligence, Elaine Rich, Kevin Knight, Shivasankar B. Nair, TheMcGraw Hill publications, Third Edition, 2009.
2. George F. Luger, Artificial Intelligence: Structures and Strategies for ComplexProblem Solving, Pearson Education, 6th ed., 2009.

E - Resources:

1. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_pdf_version.htm
2. https://www.alljntuworld.in/download/artificial-intelligence-ai-materials-notes/
3. https://drive.google.com/file/d/1mPiI4jy6YkJRDiCT21xgzN0VDNkrW23X/view 4. https://nptel.ac.in/courses/106/105/106105077/

Course outcomes:

The Student will be able to:

1. Identify the AI based problems.
2. Apply AI techniques for representing the basic problem.
3. Apply Advanced AI techniques to solve the problem.
4. Analyze Learning and explain various learning techniques.
5. Illustrate the use of expert system.

UNIT-1

Introduction to AI

What is AI ?

Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by McCarthy in 1956.

There are two ideas in the definition.

1. Intelligence
2. artificial device

What is intelligence?

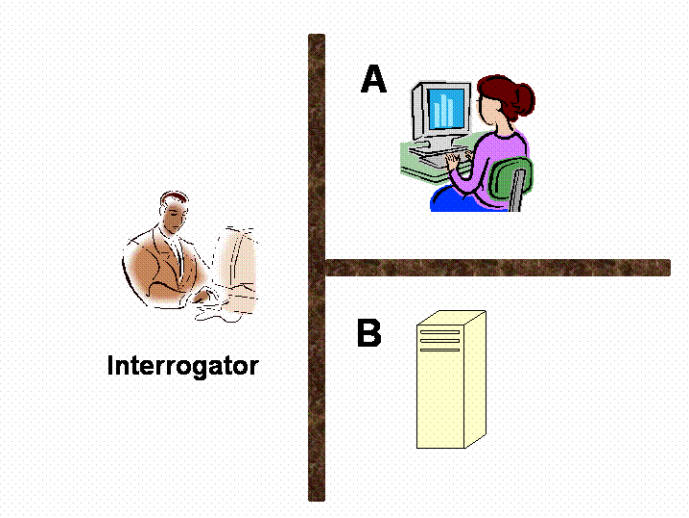
- Is it that which characterize humans? Or is there an absolute standard of judgement?
- Accordingly there are two possibilities:
  - A system with intelligence is expected to behave as intelligently as a human
  - A system with intelligence is expected to behave in the best possible manner
- Secondly what type of behavior are we talking about?
  - Are we looking at the thought process or reasoning ability of the system?
  - Or are we only interested in the final manifestations of the system in terms of its actions?

Given this scenario different interpretations have been used by different researchers as defining the scope and view of Artificial Intelligence.

1. One view is that artificial intelligence is about designing systems that are as intelligent as humans.

   This view involves trying to understand human thought and an effort to build machines that emulate the human thought process. This view is the cognitive science approach to AI.

2. The second approach is best embodied by the concept of the Turing Test. Turing held that in future computers can be programmed to acquire abilities rivaling human intelligence. As part of his argument Turing put forward the idea of an 'imitation game', in which a human being and a computer would be interrogated under conditions where the interrogator would not know which was which, the communication being entirely by textual messages. Turing argued that if the interrogator could not distinguish them by questioning, then it would be unreasonable not to call the computer intelligent. Turing's 'imitation game' is now usually called 'the Turing test' for intelligence.



Turing Test

Consider the following setting. There are two rooms, A and B. One of the rooms contains a computer. The other contains a human. The interrogator is outside and does not know which one is a computer. He can ask questions through a teletype and receives answers from both A and B. The interrogator needs to identify whether A or B are humans. To pass the Turing test, the machine has to fool the interrogator into believing that it is human. For more details on the Turing test visit the site http://cogsci.ucsd.edu/~asaygin/tt/ttest.html

3. Logic and laws of thought deals with studies of ideal or rational thought process and inference. The emphasis in this case is on the inferencing mechanism, and its properties. That is how the system arrives at a conclusion, or the reasoning behind its selection of actions is very important in this point of view. The soundness and completeness of the inference mechanisms are important here.

4. The fourth view of AI is that it is the study of rational agents. This view deals with building machines that act rationally. The focus is on how the system acts and performs, and not so much on the reasoning process. A rational agent is one that acts rationally, that is, is in the best possible manner.

Typical AI problems

While studying the typical range of tasks that we might expect an "intelligent entity" to perform, we need to consider both "common-place" tasks as well as expert tasks.
Examples of common-place tasks include

- *Recognizing* people, objects.
- Communicating (through *natural language*).
- *Navigating* around obstacles on the streets

These tasks are done matter of factly and routinely by people and some other animals.

Expert tasks include:

- Medical diagnosis.
- Mathematical problem solving
- Playing games like chess

These tasks cannot be done by all people, and can only be performed by skilled specialists.

Now, which of these tasks are easy and which ones are hard? Clearly tasks of the first type are easy for humans to perform, and almost all are able to master them. The second range of tasks requires skill development and/or intelligence and only some specialists can perform them well. However, when we look at what computer systems have been able to achieve to date, we see that their achievements include performing sophisticated tasks like medical diagnosis, performing symbolic integration, proving theorems and playing chess.

On the other hand it has proved to be very hard to make computer systems perform many routine tasks that all humans and a lot of animals can do. Examples of such tasks include navigating our way without running into things, catching prey and avoiding predators. Humans and animals are also capable of interpreting complex sensory information. We are able to recognize objects and people from the visual image that we receive. We are also able to perform complex social functions.

Intelligent behaviour

This discussion brings us back to the question of what constitutes intelligent behaviour. Some of these tasks and applications are:

- Perception involving image recognition and computer vision
- Reasoning
- Learning
- Understanding language involving natural language processing, speech processing
- Solving problems
- Robotics

Practical Impact of AI

AI components are embedded in numerous devices e.g. in copy machines for automatic correction of operation for copy quality improvement. AI systems are in everyday use for identifying credit card fraud, for advising doctors, for recognizing speech and in helping complex planning tasks. Then there are intelligent tutoring systems that provide students with personalized attention

Thus AI has increased understanding of the nature of intelligence and found many applications. It has helped in the understanding of human reasoning, and of the nature of intelligence. It has also helped us understand the complexity of modeling human reasoning.

Approaches to AI

Strong AI aims to build machines that can truly reason and solve problems. These machines should be self aware and their overall intellectual ability needs to be indistinguishable from that of a human being. Excessive optimism in the 1950s and 1960s concerning strong AI has given way to an appreciation of the extreme difficulty of the problem. Strong AI maintains that suitably programmed machines are capable of cognitive mental states.

Weak AI: deals with the creation of some form of computer-based artificial intelligence that cannot truly reason and solve problems, but can act as if it were intelligent. Weak AI holds that suitably programmed machines can simulate human cognition.

Applied AI: aims to produce commercially viable "smart" systems such as, for example, a security system that is able to recognise the faces of people who are permitted to enter a particular building. Applied AI has already enjoyed considerable success.

Cognitive AI: computers are used to test theories about how the human mind works--for example, theories about how we recognise faces and other objects, or about how we solve abstract problems.

Limits of AI Today

Today's successful AI systems operate in well-defined domains and employ narrow, specialized knowledge. Common sense knowledge is needed to function in complex, open-ended worlds. Such a system also needs to understand unconstrained natural language. However these capabilities are not yet fully present in today's intelligent systems.

What can AI systems do

Today's AI systems have been able to achieve limited success in some of these tasks.

- In Computer vision, the systems are capable of face recognition
- In Robotics, we have been able to make vehicles that are mostly autonomous.
- In Natural language processing, we have systems that are capable of simple machine translation.
- Today's Expert systems can carry out medical diagnosis in a narrow domain
- Speech understanding systems are capable of recognizing several thousand words continuous speech
- Planning and scheduling systems had been employed in scheduling experiments with

the Hubble Telescope.

- The Learning systems are capable of doing text categorization into about a 1000 topics
- In Games, AI systems can play at the Grand Master level in chess (world champion), checkers, etc.

What can AI systems NOT do yet?

- Understand natural language robustly (e.g., read and understand articles in a newspaper)
- Surf the web
- Interpret an arbitrary visual scene
- Learn a natural language
- Construct plans in dynamic real-time domains
- Exhibit true autonomy and intelligence

AI History

Intellectual roots of AI date back to the early studies of the nature of knowledge and reasoning. The dream of making a computer imitate humans also has a very early history.

The concept of intelligent machines is found in Greek mythology. There is a story in the 8th century A.D about Pygmalion Olio, the legendary king of Cyprus. He fell in love with an ivory statue he made to represent his ideal woman. The king prayed to the goddess Aphrodite, and the goddess miraculously brought the statue to life. Other myths involve human-like artifacts. As a present from Zeus to Europa, Hephaestus created Talos, a huge robot. Talos was made of bronze and his duty was to patrol the beaches of Crete.

Aristotle (384-322 BC) developed an informal system of syllogistic logic, which is the basis of the first formal deductive reasoning system.

Early in the 17th century, Descartes proposed that bodies of animals are nothing more than complex machines.

Pascal in 1642 made the first mechanical digital calculating machine.

In the 19th century, George Boole developed a binary algebra representing (some) "laws of thought."

Charles Babbage & Ada Byron worked on programmable mechanical calculating machines.

In the late 19th century and early 20th century, mathematical philosophers like Gottlob Frege, Bertram Russell, Alfred North Whitehead, and Kurt Gödel built on Boole's initial logic concepts to develop mathematical representations of logic problems.

The advent of electronic computers provided a revolutionary advance in the ability to

study intelligence.

In 1943 McCulloch & Pitts developed a Boolean circuit model of brain. They wrote the paper "A Logical Calculus of Ideas Immanent in Nervous Activity", which explained how it is possible for neural networks to compute.

Marvin Minsky and Dean Edmonds built the SNARC in 1951, which is the first randomly wired neural network learning machine (SNARC stands for Stochastic Neural-Analog Reinforcement Computer).It was a neural network computer that used 3000 vacuum tubes and a network with 40 neurons.

In 1950 Turing wrote an article on "Computing Machinery and Intelligence" which articulated a complete vision of AI.

Turing's paper talked of many things, of solving problems by searching through the space of possible solutions, guided by heuristics. He illustrated his ideas on machine intelligence by reference to chess. He even propounded the possibility of letting the machine alter its own instructions so that machines can learn from experience.

In 1956 a famous conference took place in Dartmouth. The conference brought together the founding fathers of artificial intelligence for the first time. In this meeting the term "Artificial Intelligence" was adopted.

Between 1952 and 1956, Samuel had developed several programs for playing checkers. In 1956, Newell & Simon's Logic Theorist was published. It is considered by many to be the first AI program. In 1959, Gelernter developed a Geometry Engine. In 1961 James Slagle (PhD dissertation, MIT) wrote a symbolic integration program, SAINT. It was written in LISP and solved calculus problems at the college freshman level. In 1963, Thomas Evan's program Analogy was developed which could solve IQ test type analogy problems.

In 1963, Edward A. Feigenbaum & Julian Feldman published Computers and Thought, the first collection of articles about artificial intelligence.

In 1965, J. Allen Robinson invented a mechanical proof procedure, the Resolution Method, which allowed programs to work efficiently with formal logic as a representation language. In 1967, the Dendral program (Feigenbaum, Lederberg, Buchanan, Sutherland at Stanford) was demonstrated which could interpret mass spectra on organic chemical compounds. This was the first successful knowledge-based program for scientific reasoning. In 1969 the SRI robot, Shakey, demonstrated combining locomotion, perception and problem solving.

The years from 1969 to 1979 marked the early development of knowledge-based systems In 1974: MYCIN demonstrated the power of rule-based systems for knowledge representation and inference in medical diagnosis and therapy. Knowledge representation

schemes were developed. These included frames developed by Minski. Logic based languages like Prolog and Planner were developed.

In the 1980s, Lisp Machines developed and marketed.
Around 1985, neural networks return to popularity
In 1988, there was a resurgence of probabilistic and decision-theoretic methods

The early AI systems used general systems, little knowledge. AI researchers realized that specialized knowledge is required for rich tasks to focus reasoning.

The 1990's saw major advances in all areas of AI including the following:

- machine learning, data mining
- intelligent tutoring,
- case-based reasoning,
- multi-agent planning, scheduling,
- uncertain reasoning,
- natural language understanding and translation,
- vision, virtual reality, games, and other topics.

Rod Brooks' COG Project at MIT, with numerous collaborators, made significant progress in building a humanoid robot

The first official Robo-Cup soccer match featuring table-top matches with 40 teams of interacting robots was held in 1997. For details, see the site

In the late 90s, Web crawlers and other AI-based information extraction programs become essential in widespread use of the world-wide-web.

Interactive robot pets ("smart toys") become commercially available, realizing the vision of the 18th century novelty toy makers.

In 2000, the Nomad robot explores remote regions of Antarctica looking for meteorite samples.

We will now look at a few famous AI system that has been developed over the years.

1. ALVINN:

In 1989, Dean Pomerleau at CMU created ALVINN. This is a system which learns to control vehicles by watching a person drive. It contains a neural network whose input is a 30x32 unit two dimensional camera image. The output layer is a representation of the direction the vehicle should travel.

The system drove a car from the East Coast of USA to the west coast, a total of about

2850 miles. Out of this about 50 miles were driven by a human, and the rest solely by the system.

2. Deep Blue

In 1997, the Deep Blue chess program created by IBM, beat the current world chess champion, Gary Kasparov.

3. Machine translation

A system capable of translations between people speaking different languages will be a remarkable achievement of enormous economic and cultural benefit. Machine translation is one of the important fields of endeavour in AI. While some translating systems have been developed, there is a lot of scope for improvement in translation quality.

4. Autonomous agents

In space exploration, robotic space probes autonomously monitor their surroundings, make decisions and act to achieve their goals.

NASA's Mars rovers successfully completed their primary three-month missions in April, 2004. The Spirit rover had been exploring a range of Martian hills that took two months to reach. It is finding curiously eroded rocks that may be new pieces to the puzzle of the region's past. Spirit's twin, Opportunity, had been examining exposed rock layers inside a crater.

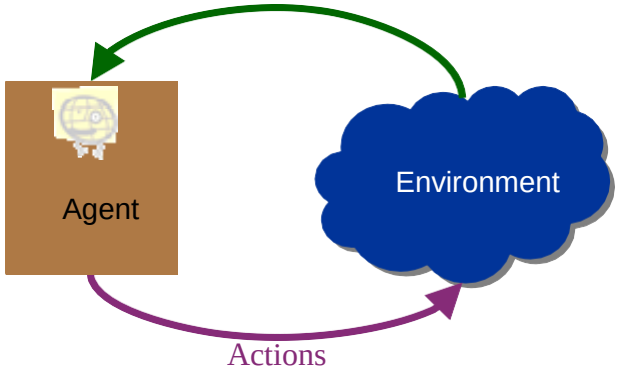
5. Internet agents

The explosive growth of the internet has also led to growing interest in internet agents to monitor users' tasks, seek needed information, and to learn which information is most useful

Introduction to Agents

An agent acts in an environment.

Percepts



An agent perceives its environment through sensors. The complete set of inputs at a given time is called a percept. The current percept, or a sequence of percepts can influence the actions of an agent. The agent can change the environment through actuators or effectors. An operation involving an effector is called an action. Actions can be grouped into action sequences. The agent can have goals which it tries to achieve.

Thus, an agent can be looked upon as a system that implements a mapping from percept sequences to actions.

A performance measure has to be used in order to evaluate an agent.

An autonomous agent decides autonomously which action to take in the current situation to maximize progress towards its goals.

Agent Performance

An agent function implements a mapping from perception history to action. The behaviour and performance of intelligent agents have to be evaluated in terms of the agent function.

The **ideal mapping** specifies which actions an agent ought to take at any point in time.

The **performance measure** is a subjective measure to characterize how successful an agent is. The success can be measured in various ways. It can be measured in terms of speed or efficiency of the agent. It can be measured by the accuracy or the quality of the solutions achieved by the agent. It can also be measured by power usage, money, etc.

Examples of Agents

1. Humans can be looked upon as agents. They have eyes, ears, skin, taste buds, etc. for sensors; and hands, fingers, legs, mouth for effectors.

2. Robots are agents. Robots may have camera, sonar, infrared, bumper, etc. for sensors. They can have grippers, wheels, lights, speakers, etc. for actuators.

   Some examples of robots are Xavier from CMU, COG from MIT, etc.

Xavier Robot (CMU)

Then we have the AIBO entertainment robot from SONY.

Aibo from SONY

3. We also have software agents or softbots that have some functions as sensors and some functions as actuators. Askjeeves.com is an example of a softbot.
4. Expert systems like the Cardiologist is an agent.
5. Autonomous spacecrafts.
6. Intelligent buildings.

Agent Faculties

The fundamental faculties of intelligence are

- Acting
- Sensing
- Understanding, reasoning, learning.

Blind action is not a characterization of intelligence. In order to act intelligently, one must sense. Understanding is essential to interpret the sensory percepts and decide on an action. Many robotic agents stress sensing and acting, and do not have understanding.

Intelligent Agents

An **Intelligent Agent** must sense, must act, must be autonomous (to some extent),. It also must be rational.

AI is about building rational agents. An agent is something that perceives and acts.
A rational agent always does the right thing.

1. What are the functionalities (goals)?
2. What are the components?
3. How do we build them?

Rationality

Perfect Rationality assumes that the rational agent knows all and will take the action that maximizes her utility. Human beings do not satisfy this definition of rationality.
Rational Action is the action that maximizes the expected value of the performance measure given the percept sequence to date.

However, a rational agent is not omniscient. It does not know the actual outcome of its actions, and it may not know certain aspects of its environment. Therefore rationality must take into account the limitations of the agent. The agent has too select the best action to the best of its knowledge depending on its percept sequence, its background knowledge and its feasible actions. An agent also has to deal with the expected outcome of the actions where the action effects are not deterministic.

Bounded Rationality

"Because of the limitations of the human mind, humans must use approximate methods to handle many tasks." Herbert Simon, 1972

Evolution did not give rise to optimal agents, but to agents which are in some senses locally optimal at best. In 1957, Simon proposed the notion of Bounded Rationality:
that property of an agent that behaves in a manner that is nearly optimal with respect to its goals as its resources will allow.

Under these promises an intelligent agent will be expected to act optimally to the best of its abilities and its resource constraints.

Agent Environment

Environments in which agents operate can be defined in different ways. It is helpful to view the following definitions as referring to the way the environment appears from the point of view of the agent itself.

Observability

In terms of observability, an environment can be characterized as fully observable or partially observable.

In a fully observable environment all of the environment relevant to the action being considered is observable. In such environments, the agent does not need to keep track of the changes in the environment. A chess playing system is an example of a system that operates in a fully observable environment.

In a partially observable environment, the relevant features of the environment are only partially observable. A bridge playing program is an example of a system operating in a partially observable environment.

Determinism

In deterministic environments, the next state of the environment is completely described by the current state and the agent's action. Image analysis systems are examples of this kind of situation. The processed image is determined completely by the current image and the processing operations.

If an element of interference or uncertainty occurs then the environment is stochastic. Note that a deterministic yet partially observable environment will *appear* to be stochastic to the agent. Examples of this are the automatic vehicles that navigate a terrain, say, the Mars rovers robot. The new environment in which the vehicle is in is stochastic in nature.

If the environment state is wholly determined by the preceding state and the actions of *multiple* agents, then the environment is said to be strategic. Example: Chess. There are two agents, the players and the next state of the board is strategically determined by the players' actions.

Episodicity

An **episodic** environment means that subsequent episodes do not depend on what actions occurred in previous episodes.

In a **sequential** environment, the agent engages in a series of connected episodes.

Dynamism

Static Environment: does not change from one state to the next while the agent is considering

its course of action. The only changes to the environment are those caused by the agent itself.

- A **static** environment does not change while the agent is thinking.
- The passage of time as an agent deliberates is irrelevant.
- The agent doesn't need to observe the world during deliberation.

A Dynamic Environment changes over time independent of the actions of the agent -- and thus if an agent does not respond in a timely manner, this counts as a choice to do nothing

Continuity

If the number of distinct percepts and actions is limited, the environment is **discrete**, otherwise it is **continuous**.

Presence of Other agents

Single agent/ Multi-agent

A multi-agent environment has other agents. If the environment contains other intelligent agents, the agent needs to be concerned about strategic, game-theoretic aspects of the environment (for either cooperative *or* competitive agents)

Most engineering environments do not have multi-agent properties, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.

Agent architectures

We will next discuss various agent architectures.

Table based agent

In table based agent the action is looked up from a table based on information about the agent's percepts. A table is simple way to specify a mapping from percepts to actions. The mapping is implicitly defined by a program. The mapping may be implemented by a rule based system, by a neural network or by a procedure.

There are several disadvantages to a table based system. The tables may become very large. Learning a table may take a very long time, especially if the table is large. Such systems usually have little autonomy, as all actions are pre-determined.

Percept based agent or reflex agent

In percept based agents,

1. information comes from **sensors - percepts**
2. changes the agents current **state of the world**
3. triggers **actions** through the **effectors**

considering its course of action. The only changes to the environment are those caused by the agent itself.

- A **static** environment does not change while the agent is thinking.
- The passage of time as an agent deliberates is irrelevant.
- The agent doesn't need to observe the world during deliberation.

A Dynamic Environment changes over time independent of the actions of the agent -- and thus if an agent does not respond in a timely manner, this counts as a choice to do nothing

Continuity

If the number of distinct percepts and actions is limited, the environment is **discrete**, otherwise it is **continuous**.

Presence of Other agents

Single agent/ Multi-agent

A multi-agent environment has other agents. If the environment contains other intelligent agents, the agent needs to be concerned about strategic, game-theoretic aspects of the environment (for either cooperative *or* competitive agents)

Most engineering environments do not have multi-agent properties, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.

Such agents are called reactive agents or stimulus-response agents. Reactive agents have no notion of history. The current state is as the sensors see it right now. The action is based on the current percepts only.

The following are some of the characteristics of percept-based agents.

- Efficient
- No internal representation for reasoning, inference.
- No strategic planning, learning.
- Percept-based agents are not good for multiple, opposing, goals.

Subsumption Architecture

We will now briefly describe the subsumption architecture (Rodney Brooks, 1986). This

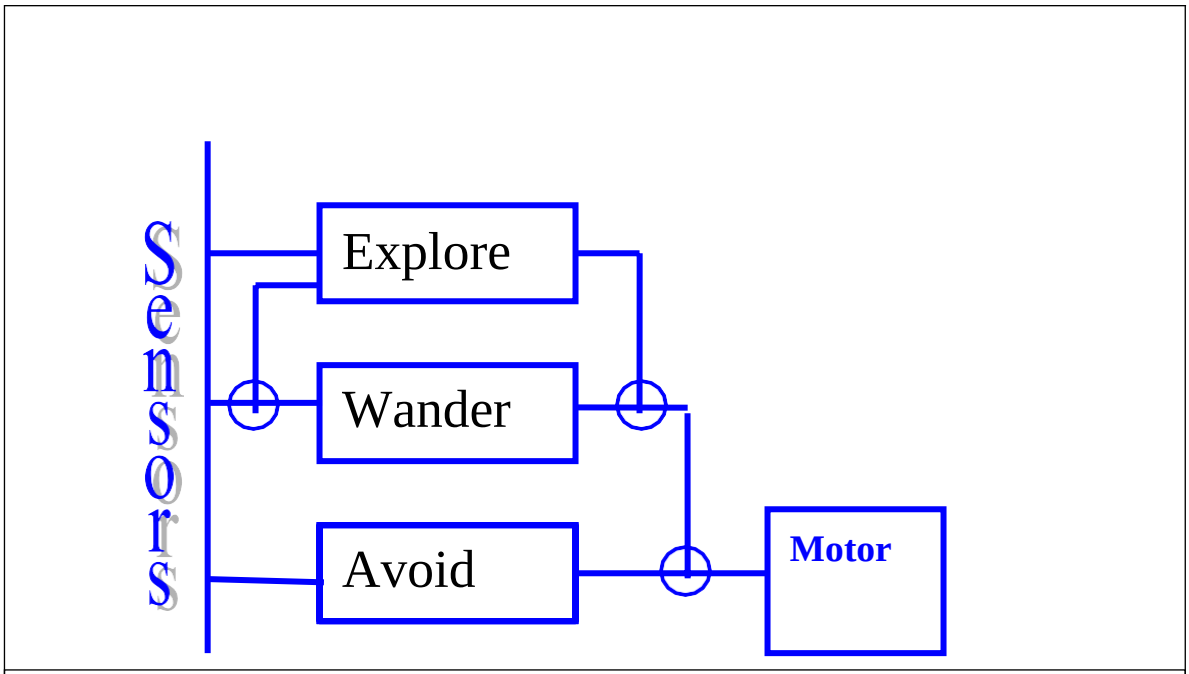
architecture is based on reactive systems. Brooks notes that in lower animals there is no deliberation and the actions are based on sensory inputs. But even lower animals are capable of many complex tasks. His argument is to follow the evolutionary path and build simple agents for complex worlds.

The main features of Brooks' architecture are.

- There is no explicit knowledge representation
- Behaviour is distributed, not centralized
- Response to stimuli is reflexive
- The design is bottom up, and complex behaviours are fashioned from the combination of simpler underlying ones.
- Individual agents are simple

The Subsumption Architecture built in layers. There are different layers of behaviour. The higher layers can override lower layers. Each activity is modeled by a finite state machine.

The subsumption architecture can be illustrated by Brooks' Mobile Robot example.



Subsumption Architecture

The system is built in three layers.

1. Layer 0: Avoid Obstacles
2. Layer1: Wander behaviour
3. Layer 2: Exploration behaviour

Layer 0 (Avoid Obstacles) has the following capabilities:

- Sonar: generate sonar scan
- Collide: send HALT message to forward
- Feel force: signal sent to run-away, turn

Layer1 (Wander behaviour)

- Generates a random heading
- Avoid reads repulsive force, generates new heading, feeds to turn and forward

Layer2 (Exploration behaviour)

- Whenlook notices idle time and looks for an interesting place.
- Pathplan sends new direction to avoid.
- Integrate monitors path and sends them to the path plan.

State-based Agent or model-based reflex agent

State based agents differ from percept based agents in that such agents maintain some sort of state based on the percept sequence received so far. The state is updated regularly based on what the agent senses, and the agent's actions. Keeping track of the state requires that

the agent has knowledge about how the world evolves, and how the agent's actions affect the world.

Thus a state based agent works as follows:

- information comes from **sensors** - **percepts**
- based on this, the agent changes the current **state of the world**
- based on **state of the world** and **knowledge (memory)**, it triggers **actions** through the **effectors**

Goal-based Agent

The goal based agent has some goal which forms a basis of its actions.
Such agents work as follows:

- information comes from **sensors** - **percepts**
- changes the agents current **state of the world**
- based on **state of the world** and **knowledge (memory)** and **goals/intentions**, it chooses **actions** and does them through the **effectors**.

Goal formulation based on the current situation is a way of solving many problems and search is a universal problem solving mechanism in AI. The sequence of steps required to solve a problem is not known a priori and must be determined by a systematic exploration of the alternatives.

Utility-based Agent

Utility based agents provides a more general agent framework. In case that the agent has multiple goals, this framework can accommodate different preferences for the different goals.
Such systems are characterized by a utility function that maps a state or a sequence of states to a real valued utility. The agent acts so as to maximize expected utility

Learning Agent

Learning allows an agent to operate in initially unknown environments. The learning element modifies the performance element. Learning is required for true autonomy

Conclusion

In conclusion AI is a truly fascinating field. It deals with exciting but hard problems. A goal of AI is to build intelligent agents that act so as to optimize performance.

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **autonomous agent** uses its own experience rather than built-in knowledge of the environment by the designer.
- An agent program maps from percept to action and updates its internal state.

    - Reflex agents respond immediately to percepts.
    - Goal-based agents act in order to achieve their goal(s).
    - Utility-based agents maximize their own utility function.
- Representing knowledge is important for successful agent design.
- The most challenging environments are partially observable, stochastic, sequential, dynamic, and continuous, and contain multiple intelligent agents.

Introduction to State Space Search

State space search

- Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states .
- A state is defined by the specification of the values of all attributes of interest in the world
- An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The initial state is where you start
- The goal state is the partial description of the solution

Goal Directed Agent

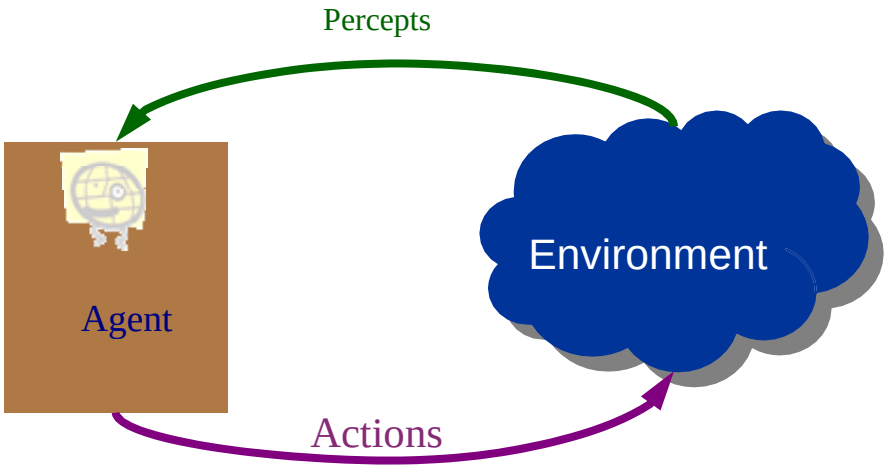


Figure 1

We have earlier discussed about an intelligent agent. Today we will study a type of intelligent agent which we will call a goal directed agent.

A goal directed agent needs to achieve certain goals. Such an agent selects its actions based on the goal it has. Many problems can be represented as a set of states and a set of rules of how one state is transformed to another. Each state is an abstract representation of the agent's environment. It is an abstraction that denotes a configuration of the agent.
Initial state : The description of the starting configuration of the agent
An action/ operator takes the agent from one state to another state. A state can have a number of successor states.
A plan is a sequence of actions.

A goal is a description of a set of desirable states of the world. Goal states are often specified by a goal test which any goal state must satisfy.

Let us look at a few examples of goal directed agents.

1. 15-puzzle: The goal of an agent working on a 15-puzzle problem may be to reach a configuration which satisfies the condition that the top row has the tiles 1, 2 and 3. The details of this problem will be described later.
2. The goal of an agent may be to navigate a maze and reach the HOME position.

The agent must choose a sequence of actions to achieve the desired goal.

State Space Search Notations

Let us begin by introducing certain terms.
An initial state is the description of the starting configuration of the agent
An action or an operator takes the agent from one state to another state which is called a successor state. A state can have a number of successor states.
A plan is a sequence of actions. The cost of a plan is referred to as the path cost. The path cost is a positive number, and a common path cost may be the sum of the costs of the steps in the path.

Now let us look at the concept of a search problem.
Problem formulation means choosing a relevant set of states to consider, and a feasible set of operators for moving from one state to another.
Search is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.

Search Problem

We are now ready to formally describe a search problem.
A search problem consists of the following:

- S: the full set of states
- s_0 : the initial state
- A:S→S is a set of operators
- G is the set of final states. Note that G □S

These are schematically depicted in Figure 2.

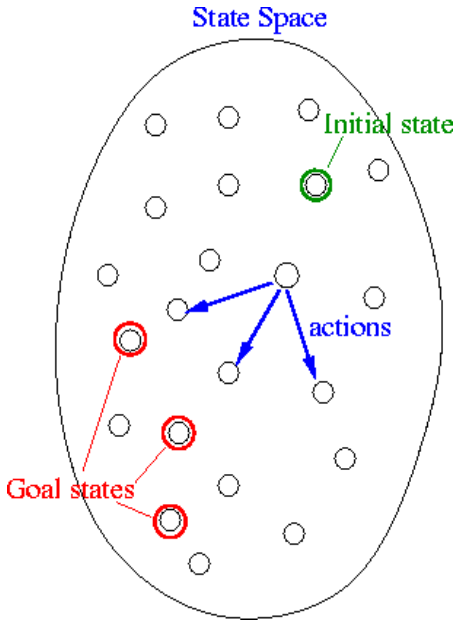


Figure 2

The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state g□G. A search problem is represented by a 4-tuple {S, s_0, A, G}.

S: set of states

s_0 □ S : initial state

A: S□ S operators/ actions that transform one state to another state

G : goal, a set of states. G □ S

This sequence of actions is called a solution plan. It is a path from the initial state to a goal state. A *plan* P is a sequence of actions.

P = {a_0, a_1, … , a_N} which leads to traversing a number of states {s_0, s_1, … , s_{N+1}□G}. A sequence of states is called a path. The cost of a path is a positive number. In many cases the path cost is computed by taking the sum of the costs of each action.

Representation of search problems

A search problem is represented using a directed graph.

- The states are represented as nodes.
- The allowed actions are represented as arcs.

Searching process

The generic searching process can be very simply described in terms of the following steps:

Do until a solution is found or the state space is exhausted.

1. Check the current state
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state
   If it is not, the new state becomes the current state and the process is repeated

Examples

Illustration of a search process

We will now illustrate the searching process with the help of an example. Consider the problem depicted in Figure 3.

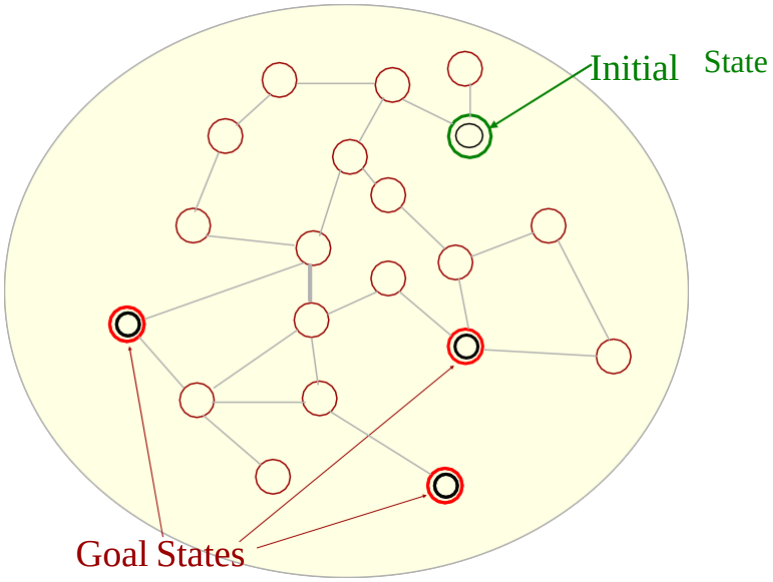


Figure 3

s_0 is the initial state.
The successor states are the adjacent states in the graph.
There are three goal states.

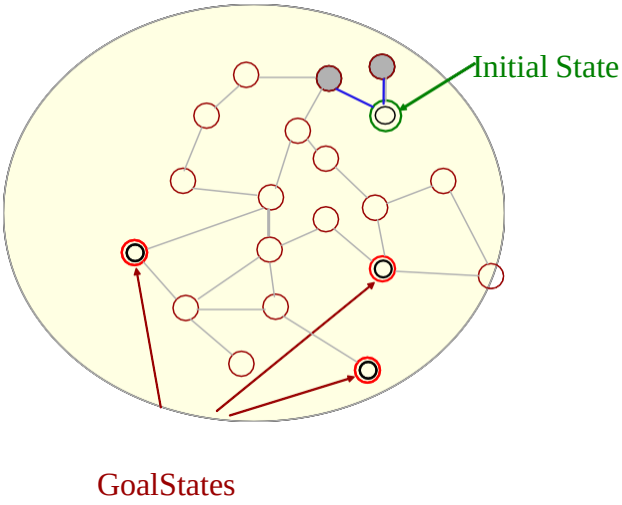


Figure 4

The two successor states of the initial state are generated.

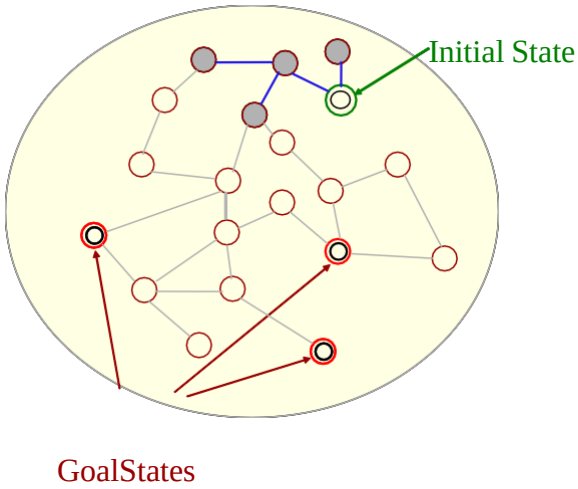


Figure 5

The successors of these states are picked and their successors are generated.

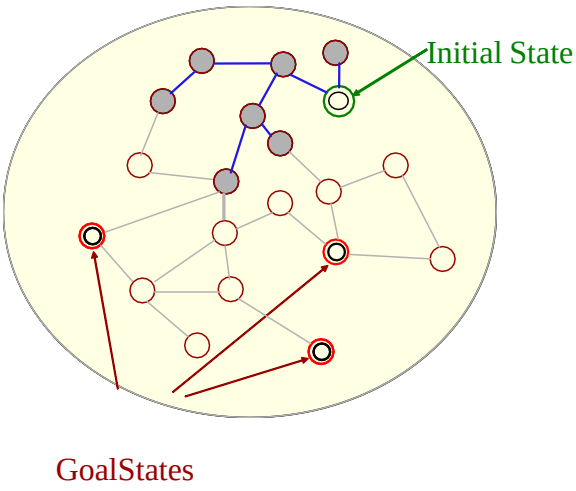


Figure 6

Successors of all these states are generated.

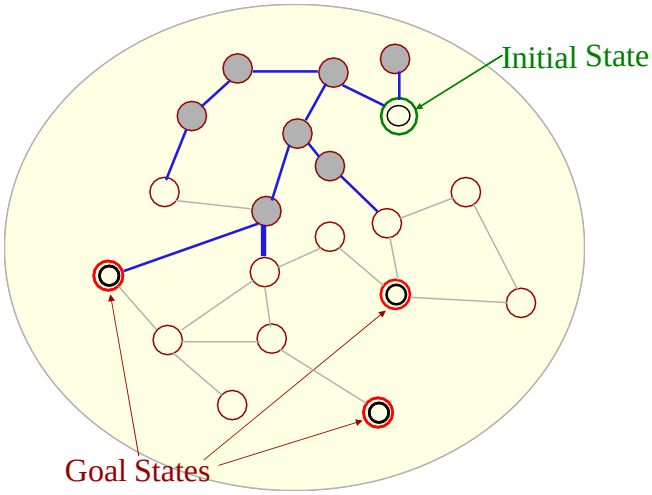


Figure 7

The successors are generated.

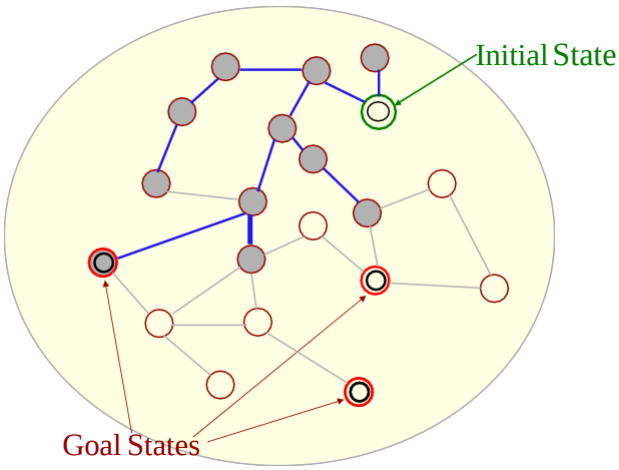


Figure 8

A goal state has been found.

The above example illustrates how we can start from a given state and follow the successors, and be able to find solution paths that lead to a goal state. The grey nodes define the search tree. Usually the search tree is extended one node at a time. The order in which the search tree is extended depends on the search strategy.

We will now illustrate state space search with one more example – the pegs and disks problem. We will illustrate a solution sequence which when applied to the initial state takes us to a goal state.

Example problem: Pegs and Disks problem

Consider the following problem. We have 3 pegs and 3 disks.
Operators: one may move the topmost disk on any needle to the topmost position to any other needle

In the goal state all the pegs are in the needle B as shown in the figure below..

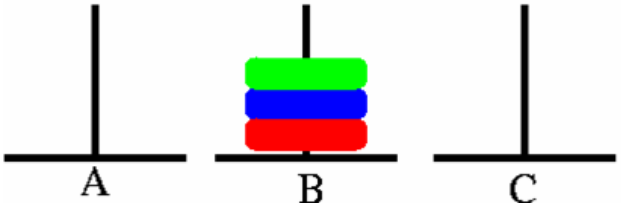


Figure 9

The initial state is illustrated below.

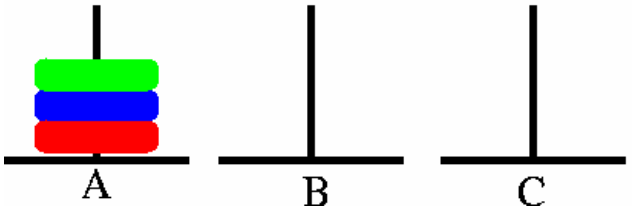


Figure 10

Now we will describe a sequence of actions that can be applied on the initial state.

Step 1: Move A □ C

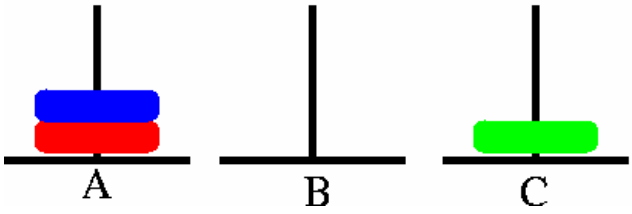


Figure 11

Step 2: Move A □ B

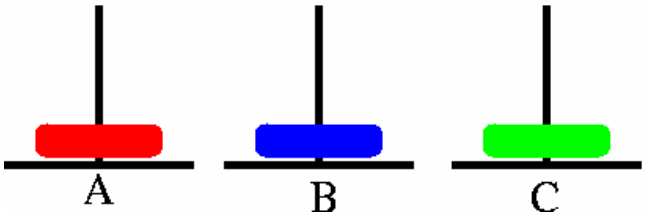


Figure 12

Step 3: Move A □ C

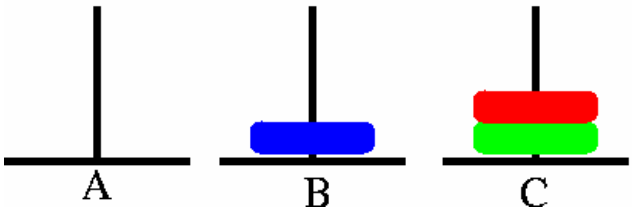


Figure 13

Step 4: Move B□ A

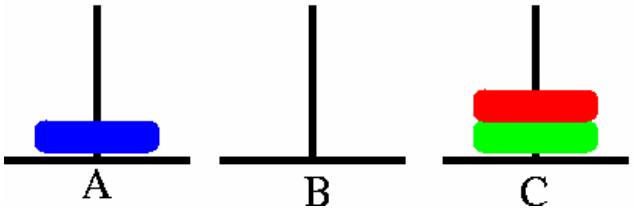


Figure 14

Step 5: Move C □ B

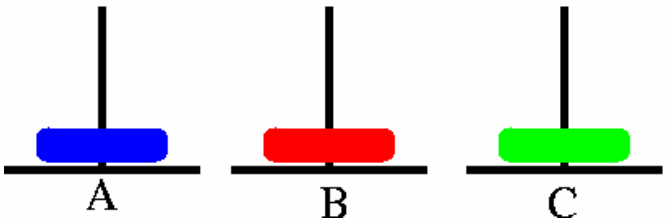


Figure 15

Step 6: Move A □ B

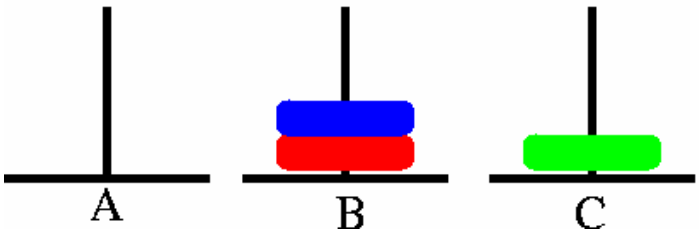


Figure 16

Step 7: Move C□ B

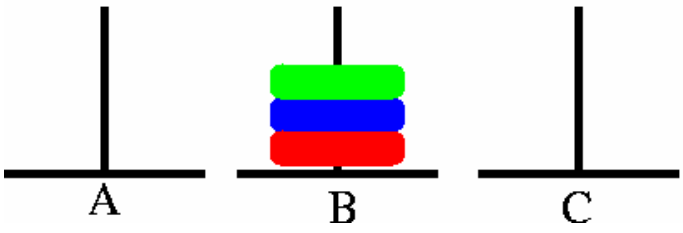


Figure 17

We will now look at another search problem – the 8-queens problem, which can be generalized to the N-queens problem.

8 queens problem

The problem is to place 8 queens on a chessboard so that no two queens are in the same row, column or diagonal

The picture below on the left shows a solution of the 8-queens problem. The picture on the right is not a correct solution, because some of the queens are attacking each other.

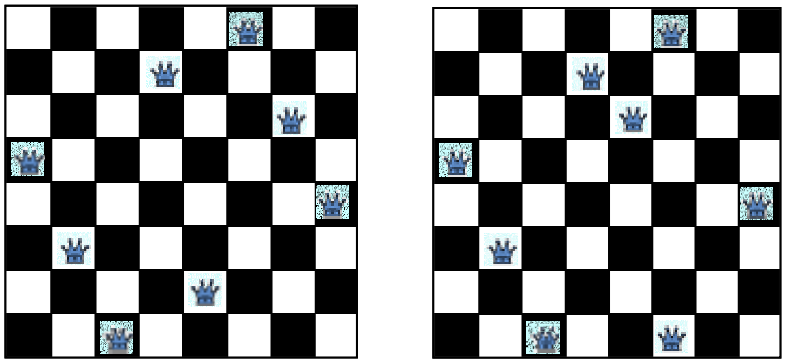


Figure 18

How do we formulate this in terms of a state space search problem? The problem formulation involves deciding the representation of the states, selecting the initial state representation, the description of the operators, and the successor states. We will now show that we can formulate the search problem in several different ways for this problem.

N queens problem formulation 1

- States: Any arrangement of 0 to 8 queens on the board
- Initial state: 0 queens on the board
- Successor function: Add a queen in any square
- Goal test: 8 queens on the board, none are attacked

The initial state has 64 successors. Each of the states at the next level have 63 successors, and so on. We can restrict the search tree somewhat by considering only those successors where no queen is attacking each other. To do that we have to check the new queen against all existing queens on the board. The solutions are found at a depth of 8.

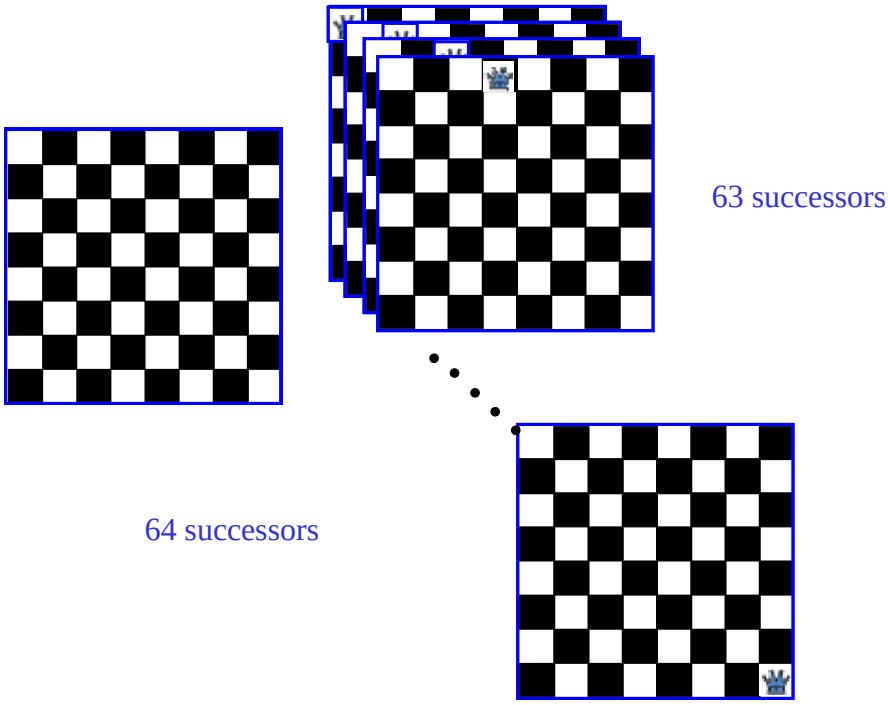


Figure 19

N queens problem formulation 2

- States: Any arrangement of 8 queens on the board
- Initial state: All queens are at column 1
- Successor function: Change the position of any one queen
- Goal test: 8 queens on the board, none are attacked

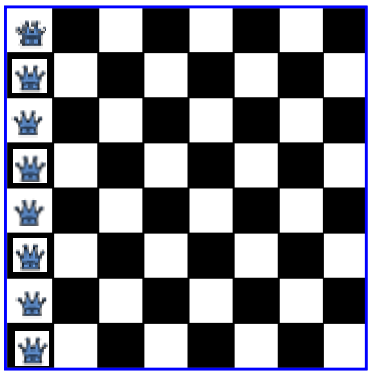


Figure 20

If we consider moving the queen at column 1, it may move to any of the seven remaining columns.

N queens problem formulation 3

- States: Any arrangement of k queens in the first k rows such that none are attacked
- Initial state: 0 queens on the board
- Successor function: Add a queen to the (k+1)th row so that none are attacked.
- Goal test : 8 queens on the board, none are attacked

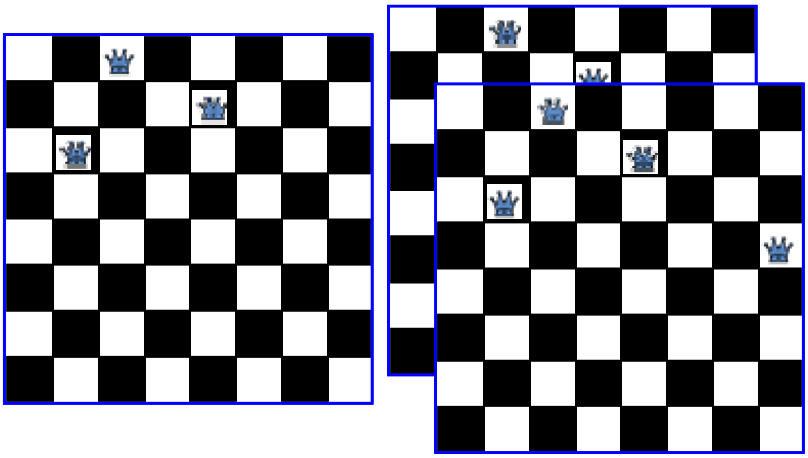


Figure 21

We will now take up yet another search problem, the 8 puzzle.

Problem Definition - Example, 8 puzzle

5	4	
6	1	8
7	3	2

InitialState

1	4	7
2	5	8
3	6	

Goal State

Figure 22

In the 8-puzzle problem we have a 3□3 square board and 8 numbered tiles. The board has one blank position. Bocks can be slid to adjacent blank positions. We can alternatively and equivalently look upon this as the movement of the blank position up, down, left or right. The objective of this puzzle is to move the tiles starting from an initial position and arrive at a given goal configuration.
The 15-puzzle problems is similar to the 8-puzzle. It has a 4□4 square board and 15 numbered tiles

The state space representation for this problem is summarized below:

States: A state is a description of each of the eight tiles in each location that it can occupy.
Operators/Action: The blank moves left, right, up or down
Goal Test: The current state matches a certain state (e.g. one of the ones shown on previous slide)
Path Cost: Each move of the blank costs 1

A small portion of the state space of 8-puzzle is shown below. Note that we do not need to generate all the states before the search begins. The states can be generated when required.

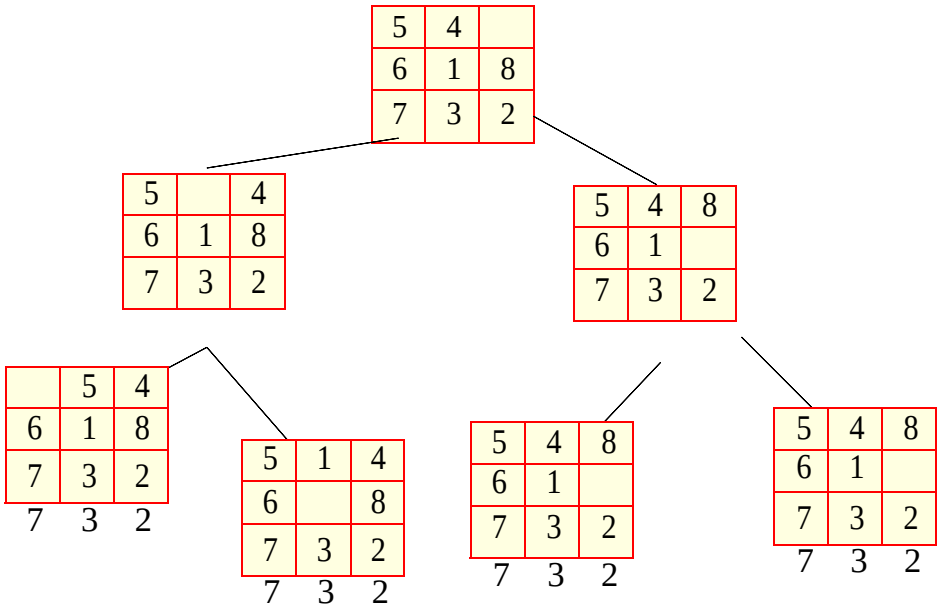


Figure 23

8- puzzle partial state space

Problem Definition - Example, tic-tac-toe

Another example we will consider now is the game of tic-tac-toe. This is a game that involves two players who play alternately. Player one puts a X in an empty position. Player 2 places an O in an unoccupied position. The player who can first get three of his symbols in the same row, column or diagonal wins. A portion of the state space of tic-tac-toe is depicted below.

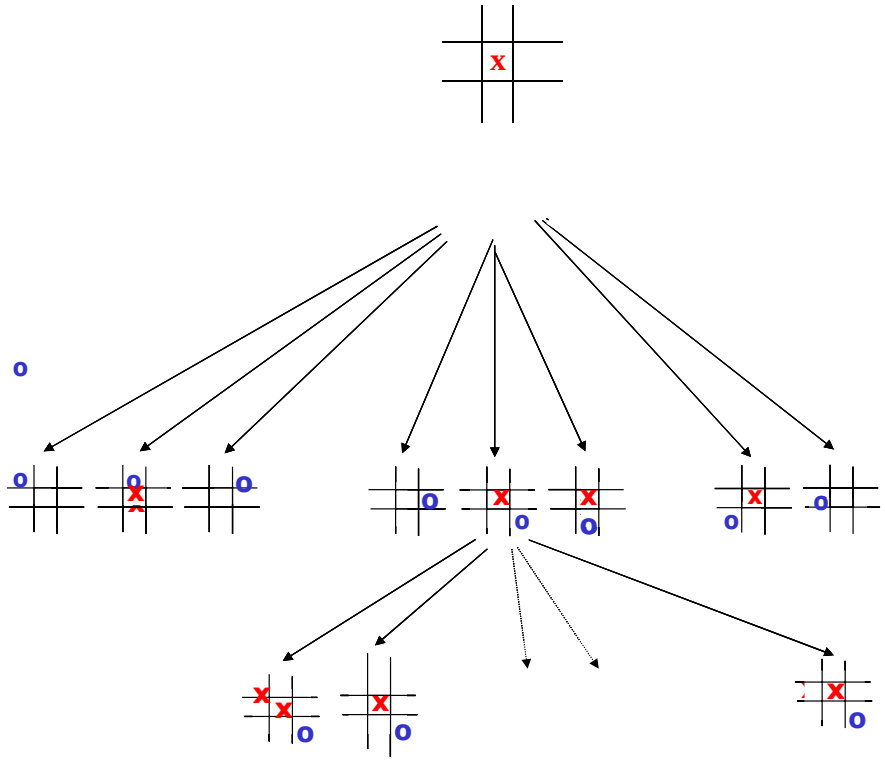


Figure 24

Now that we have looked at the state space representations of various search problems, we will now explore briefly the search algorithms.

Water jug problem

You have three jugs measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You need to measure out exactly one gallon.

- Initial state: All three jugs are empty
- Goal test: Some jug contains exactly one gallon.
- *Successor function:Applying the*
  - **action** *tranfer* to jugs i and j with capacities Ci and Cj and containing Gi and Gj gallons of water, respectively, leaves jug i with max(0, Gi−(Cj −Gj)) gallons of water and jug j with min(Cj, Gi+ Gj) gallons of water.
  - Applying the **action** *fill* to jug i leaves it with Ci gallons ofwater.
- Cost function: Charge one point for each gallon of water transferred and each gallon of water filled

Explicit vs Implicit state space

The state space may be explicitly represented by a graph. But more typically the state space can be implicitly represented and generated when required. To generate the state space implicitly, the agent needs to know

- the initial state
- The operators and a description of the effects of the operators

An operator is a function which "expands" a node and computes the successor node(s). In the various formulations of the N-queen's problem, note that if we know the effects of the operators, we do not need to keep an explicit representation of all the possible states, which may be quite large.

Basic Search Strategies:
Problem Spaces, Uninformed Search

Search

Searching through a state space involves the following:

- A set of states
- Operators and their costs
- Start state
- A test to check for goal state

We will now outline the basic search algorithm, and then consider various variations of this algorithm.

The basic search algorithm

```
Let L be a list containing the initial state (L= the fringe)
Loop
        if L is empty return failure
        Node ⓪ select (L)
          if Node is a goal
                  then return Node
                     (the path from initial state to Node)
        else generate all successors of Node, and
             merge the newly generated states into L
End Loop
```

We need to denote the states that have been generated. We will call these as nodes. The data structure for a node will keep track of not only the state, but also the parent state or the operator that was applied to get this state. In addition the search algorithm maintains a list of nodes called the fringe. The fringe keeps track of the nodes that have been generated but are yet to be explored. The fringe represents the frontier of the search tree generated. The basic search algorithm has been described above.

Initially, the fringe contains a single node corresponding to the start state. In this version we use only the OPEN list or fringe. The algorithm always picks the first node from fringe for expansion. If the node contains a goal state, the path to the goal is returned. The path corresponding to a goal node can be found by following the parent pointers. Otherwise all the successor nodes are generated and they are added to the fringe.

The successors of the current expanded node are put in fringe. We will soon see that the

order in which the successors are put in fringe will determine the property of the search algorithm.

Search algorithm: Key issues

Corresponding to a search algorithm, we get a search tree which contains the generated and the explored nodes. The search tree may be unbounded. This may happen if the state space is infinite. This can also happen if there are loops in the search space. How can we handle loops?

Corresponding to a search algorithm, should we return a path or a node? The answer to this depends on the problem. For problems like N-queens we are only interested in the goal state. For problems like 15-puzzle, we are interested in the solution path.

We see that in the basic search algorithm, we have to select a node for expansion. Which node should we select? Alternatively, how would we place the newly generated nodes in the fringe? We will subsequently explore various search strategies and discuss their properties,

Depending on the search problem, we will have different cases. The search graph may be weighted or unweighted. In some cases we may have some knowledge about the quality of intermediate states and this can perhaps be exploited by the search algorithm. Also depending on the problem, our aim may be to find a minimal cost path or any to find path as soon as possible.

Which path to find?

The objective of a search problem is to find a path from the initial state to a goal state. If there are several paths which path should be chosen? Our objective could be to find any path, or we may need to find the shortest path or least cost path.

Evaluating Search strategies

We will look at various search strategies and evaluate their problem solving performance. What are the characteristics of the different search algorithms and what is their efficiency? We will look at the following three factors to measure this.

1. Completeness: Is the strategy guaranteed to find a solution if one exists?

2. Optimality: Does the solution have low cost or the minimal cost?

3. What is the search cost associated with the time and memory required to find a solution?
    a. Time complexity: Time taken (number of nodes expanded) (worst or average case) to find a solution.
    b. Space complexity: Space used by the algorithm measured in terms of the maximum size of fringe

The different search strategies that we will consider include the following:

1. Blind Search strategies or Uninformed search
    a. Depth first search
    b. Breadth first search
    c. Iterative deepening search
    d. Iterative broadening search
2. Informed Search
3. Constraint Satisfaction Search
4. Adversary Search

Blind Search

In this lesson we will talk about blind search or uninformed search that does not use any extra information about the problem domain. The two common methods of blind search are:

- BFS or Breadth First Search
- DFS or Depth First Search

Search Tree

Consider the explicit state space graph shown in the figure.

One may list all possible paths, eliminating cycles from the paths, and we would get the complete search tree from a state space graph. Let us examine certain terminology associated with a search tree. A search tree is a data structure containing a root node, from where the search starts. Every node may have 0 or more children. If a node X is a child of node Y, node Y is said to be a parent of node X.

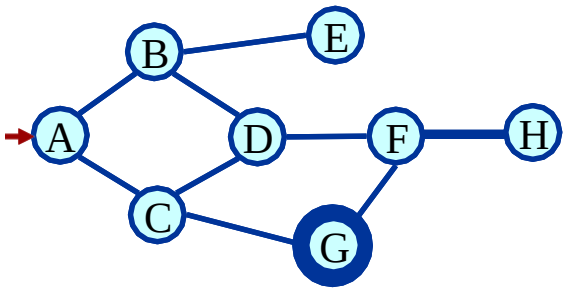


Figure 1: A State Space Graph

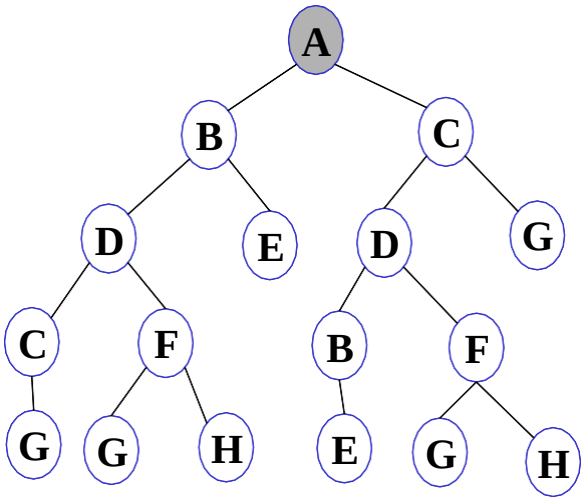


Figure 2: Search tree for the state space graph in Figure 25

Consider the state space given by the graph in Figure 25. Assume that the arcs are bidirectional. Starting the search from state A the search tree obtained is shown in Figure 26.

Search Tree – Terminology

- Root Node: The node from which the search starts.
- Leaf Node: A node in the search tree having no children.
- Ancestor/Descendant: X is an ancestor of Y is either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- Branching factor: the maximum number of children of a non-leaf node in the search tree
- Path: A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

We also need to introduce some data structures that will be used in the search algorithms.

Node data structure

A node used in the search algorithm is a data structure which contains the following:

1. A state description
2. A pointer to the parent of the node
3. Depth of the node
4. The operator that generated this node
5. Cost of this path (sum of operator costs) from the start state

The nodes that the algorithm has generated are kept in a data structure called OPEN or fringe. Initially only the start node is in OPEN.

The search starts with the root node. The algorithm picks a node from OPEN for expanding and generates all the children of the node. Expanding a node from OPEN results in a closed node. Some search algorithms keep track of the closed nodes in a data structure called CLOSED.

A solution to the search problem is a sequence of operators that is associated with a path from a start node to a goal node. The cost of a solution is the sum of the arc costs on the solution path. For large state spaces, it is not practical to represent the whole space. State space search makes explicit a sufficient portion of an implicit state space graph to find a goal node. Each node represents a partial solution path from the start node to the given node. In general, from this node there are many possible paths that have this partial path as a prefix.

The search process constructs a search tree, where

- **root** is the initial state and
- **leaf nodes** are nodes
  - not yet expanded (i.e., in fringe) or
  - having no successors (i.e., "dead-ends")

Search tree may be infinite because of loops even if state space is small

The search problem will return as a solution a path to a goal node. Finding a path is important in problems like path finding, solving 15-puzzle, and such other problems. There are also problems like the N-queens problem for which the path to the solution is not important. For such problems the search problem needs to return the goal state only.

Breadth First Search

Algorithm

Breadth first search
Let *fringe* be a list containing the initial state
Loop
    if *fringe* is empty return failure
    Node □ remove-first (fringe)
    if Node is a goal
        then return the path from initial state to Node
    else generate all successors of Node, and
        (merge the newly generated nodes into *fringe*)
        add generated nodes to the back of *fringe*
End Loop

Note that in breadth first search the newly generated nodes are put at the back of fringe or the OPEN list. What this implies is that the nodes will be expanded in a FIFO (First In

First Out) order. The node that enters OPEN earlier will be expanded earlier. This amounts to expanding the shallowest nodes first.

BFS illustrated

We will now consider the search space in Figure 1, and show how breadth first search works on this graph.

Step 1: Initially fringe contains only one node corresponding to the source state A.

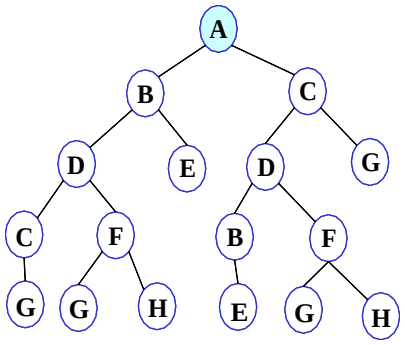


Figure 3

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

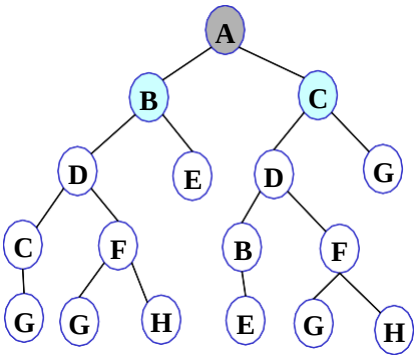


Figure 4

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

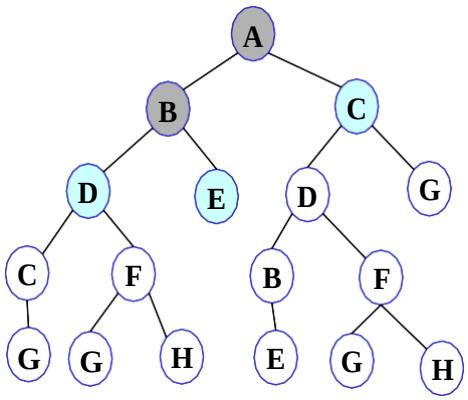


Figure 5

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

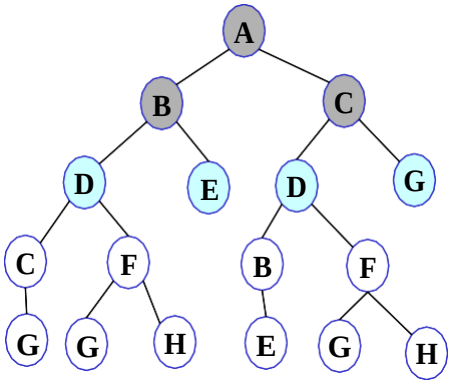


Figure 6

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

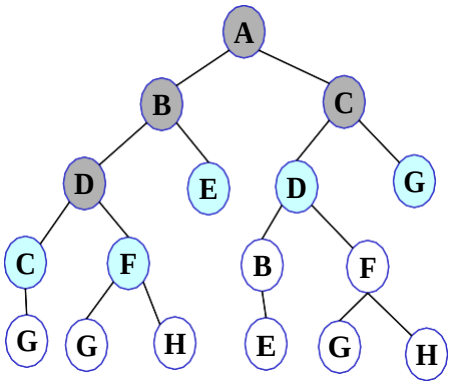
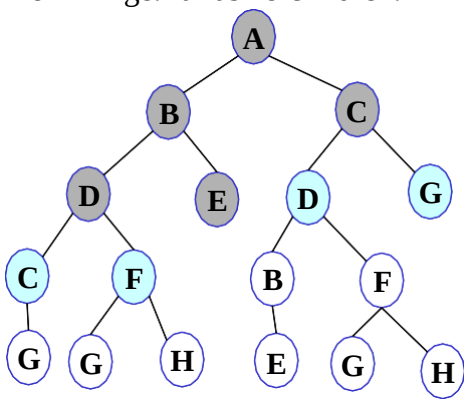


Figure 7

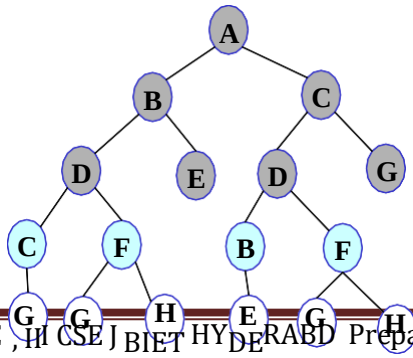
FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.



FRINGE: D G C F

Step 7: D is expanded, B and F are put in OPEN.



Goal!

Figure 8

FRINGE: G C F B F

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

Properties of Breadth-First Search

We will now explore certain properties of breadth first search. Let us consider a model of the search tree as shown in Figure 3. We assume that every non-leaf node has b children. Suppose that d is the depth o the shallowest goal node, and m is the depth of the node found first.

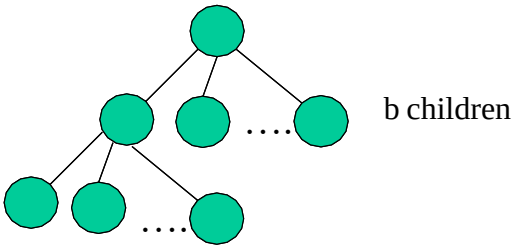


Figure 9: Model of a search tree with uniform branching factor b

Breadth first search is:

- Complete.
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- The algorithm has exponential time and space complexity. Suppose the search tree can be modeled as a b-ary tree as shown in Figure 3. Then the time and space complexity of the algorithm is O(bd) where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node.

A complete search tree of depth d where each non-leaf node has b children, has a total of $1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1)/(b-1)$ nodes

Consider a complete search tree of depth 15, where every node at depths 0 to14 has 10 children and every node at depth 15 is a leaf node. The complete search tree in this case will have $O(10^{15})$ nodes. If BFS expands 10000 nodes per second and each node uses 100 bytes of storage, then BFS will take 3500 years to run in the worst case, and it will use 11100 terabytes of memory. So you can see that the breadth first search algorithm cannot be effectively used unless the search space is quite small. You may also observe that even if you have all the time at your disposal, the search algorithm cannot run because it will run out of memory very soon.

Advantages of Breadth First Search

Finds the path of minimal length to the goal.

Disadvantages of Breadth First Search

Requires the generation and storage of a tree whose size is exponential the the depth of the shallowest goal node

Uniform-cost search

This algorithm is by Dijkstra [1959]. The algorithm expands nodes in the order of their cost from the source.

We have discussed that operators are associated with costs. The path cost is usually taken to be the sum of the step costs.

In uniform cost search the newly generated nodes are put in OPEN according to their path costs. This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in OPEN.
Let g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.
Some properties of this search algorithm are:

- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

Depth first Search

Algorithm

```
Depth First Search
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node □ remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
            merge the newly generated nodes into fringe
            add generated nodes to the front of fringe
End Loop
```

The depth first search algorithm puts newly generated nodes in the front of OPEN. This results in expanding the deepest node first. Thus the nodes in OPEN follow a LIFO order (Last In First Out). OPEN is thus implemented using a stack data structure.

DFS illustrated

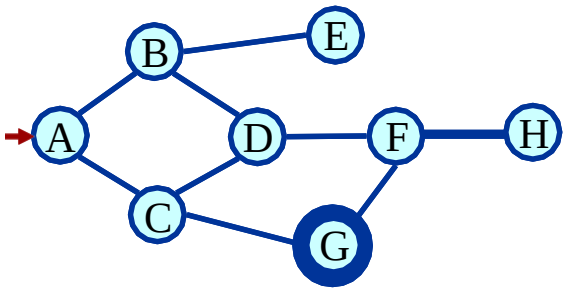


Figure 10

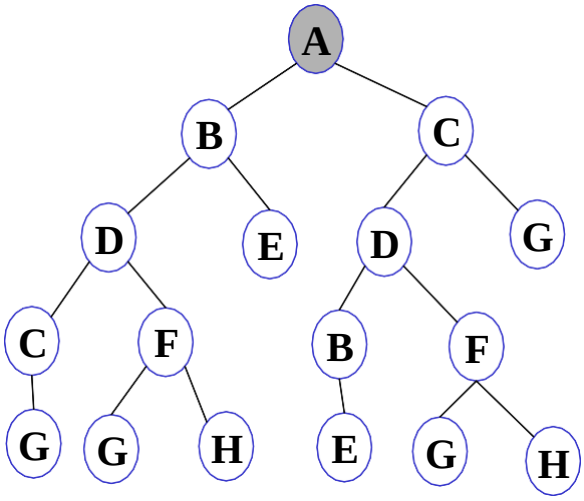
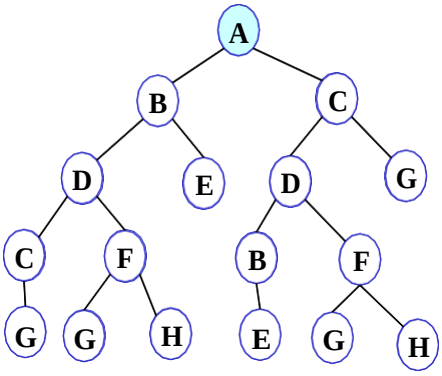


Figure 11: Search tree for the state space graph in Figure 34

Let us now run Depth First Search on the search space given in Figure 34, and trace its progress.

Step 1: Initially fringe contains only the node for A.

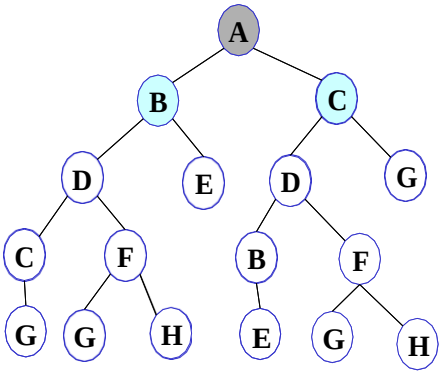


FRINGE: A

FRINGE: A

Figure 12

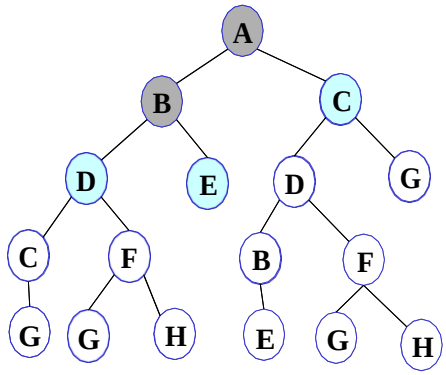
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



FRINGE: B C

Figure 13

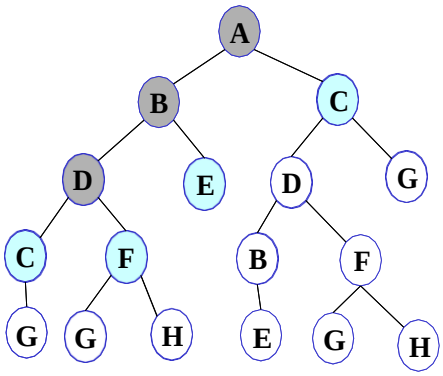
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



FRINGE: D E C

Figure 14

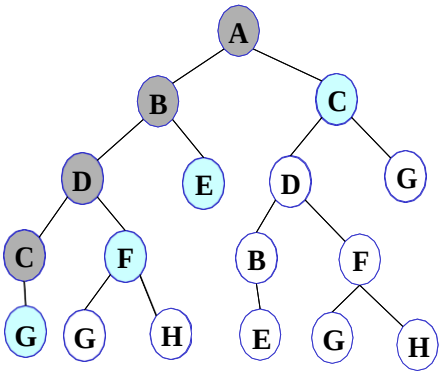
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



FRINGE: C F E C

Figure 15

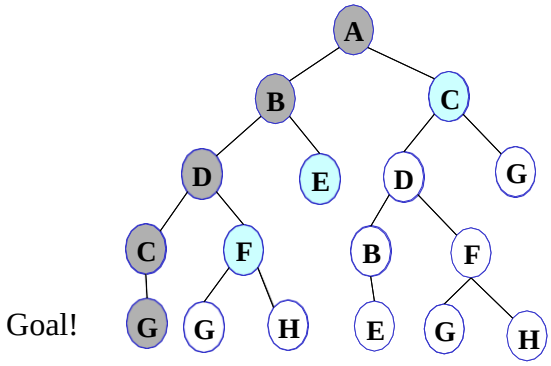
Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



FRINGE: G F E C

Figure 16

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.



FRINGE: G F E C

Figure 17

Properties of Depth First Search

Let us now examine some properties of the DFS algorithm. The algorithm takes exponential time. If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time $O(b^d)$. However the space taken is linear in the depth of the search tree, O(bN).

Note that the time taken by the algorithm is related to the maximum depth of the search tree. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus Depth First Search is not complete.

Depth Limited Search

A variation of Depth First Search circumvents the above problem by keeping a depth bound. Nodes are only expanded if they have depth less than the bound. This algorithm is known as depth-limited search.

Depth limited search (limit)
Let fringe be a list containing the initial state Loop if fringe is empty return failure Node □ remove-first (fringe) if Node is a goal then return the path from initial state to Node else if depth of Node = limit return cutoff else add generated nodes to the front of fringe End Loop

.

Depth-First Iterative Deepening (DFID)

First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

DFID
until solution found do *DFS with depth cutoff c* *c = c+1*

Advantage

- Linear memory requirements of depth-first search
- Guarantee for goal node of minimal depth

Procedure

Successive depth-first searches are conducted – each with depth bounds increasing by 1

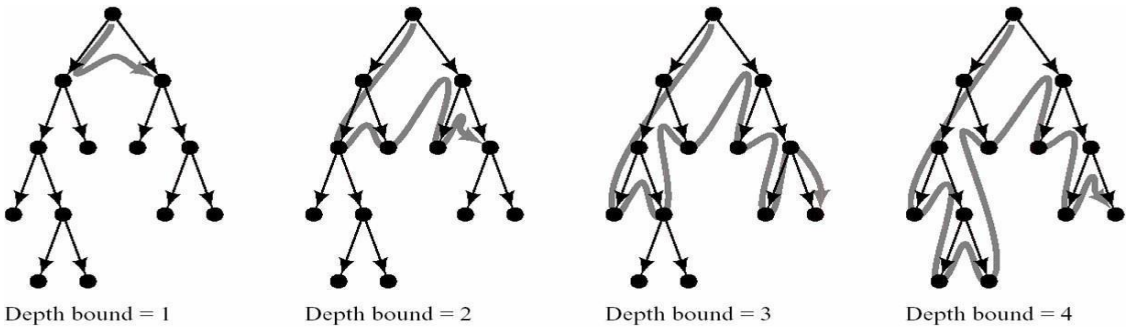


Figure 18: Depth First Iterative Deepening

Properties

For large *d* the ratio of the number of nodes expanded by DFID compared to that of DFS is given by *b/*(*b-1*).
For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative-deepening search than breadth-first search

The algorithm is

- Complete
- Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, $O(b^d)$

If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc.
Hence $b^d + 2b^{(d-1)} + ... + db <= b^d / (1 - 1/b)^2 = O(b^d)$.

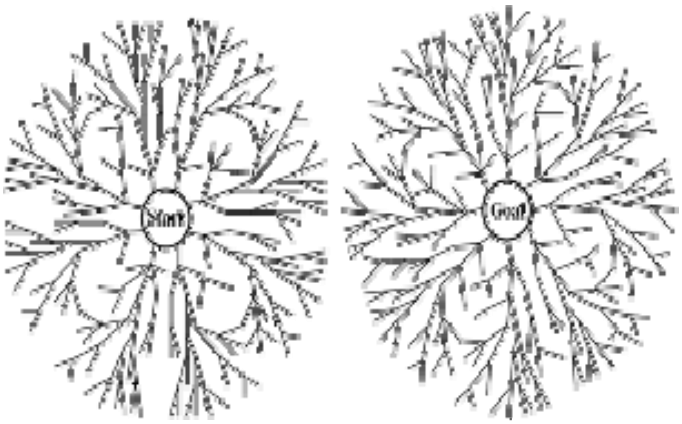
- **Linear space complexity**, O(bd), like DFS

Depth First Iterative Deepening combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly)
This algorithm is generally preferred for **large state spaces** where the **solution depth is unknown.**

There is a related technique called *iterative broadening* is useful when there are many goal nodes. This algorithm works by first constructing a search tree by expanding only one child per node. In the 2nd iteration, two children are expanded, and in the ith iteration I children are expanded.

Bi-directional search

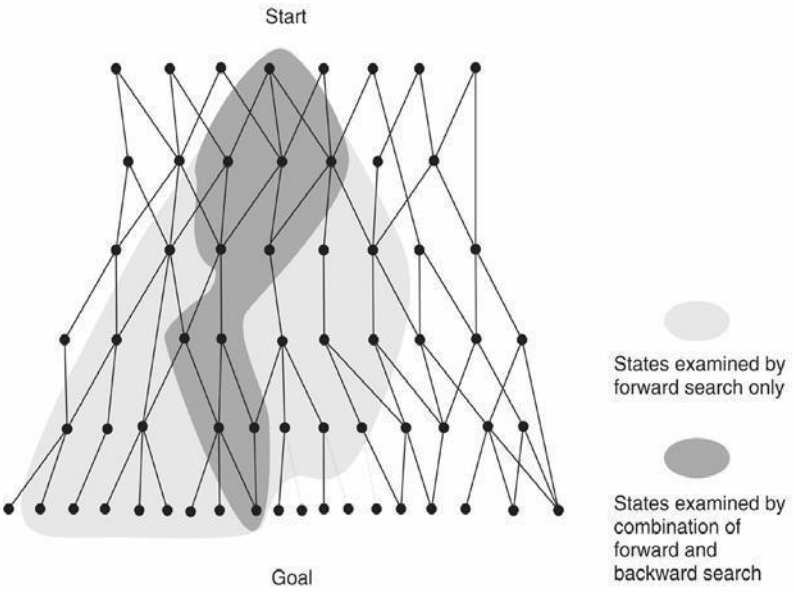
Suppose that the search problem is such that the arcs are bidirectional. That is, if there is an operator that maps from state A to state B, there is another operator that maps from state B to state A. Many search problems have reversible arcs. 8-puzzle, 15-puzzle, path planning etc are examples of search problems. However there are other state space search formulations which do not have this property. The water jug problem is a problem that does not have this property. But if the arcs are reversible, you can see that instead of starting from the start state and searching for the goal, one may start from a goal state and try reaching the start state. If there is a single state that satisfies the goal property, the search problems are identical.
How do we search backwards from goal? One should be able to generate predecessor states. Predecessors of node n are all the nodes that have n as successor. This is the motivation to consider bidirectional search.



Algorithm: Bidirectional search involves alternate searching from the start state toward the goal and from the goal state toward the start. The algorithm stops when the frontiers intersect.

A search algorithm has to be selected for each half. How does the algorithm know when the frontiers of the search tree intersect? For bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.

Bidirectional search can sometimes lead to finding a solution more quickly. The reason can be seen from inspecting the following figure.



Also note that the algorithm works well only when there are unique start and goal states. Question: How can you make bidirectional search work if you have 2 possible goal states?

Time and Space Complexities

Consider a search space with branching factor b. Suppose that the goal is d steps away from the start state. Breadth first search will expand $O(b^d)$ nodes.

If we carry out bidirectional search, the frontiers may meet when both the forward and the backward search trees have depth = d/2. Suppose we have a good hash function to check for nodes in the fringe. IN this case the time for bidirectional search will be $O((b^{d/2})$. Also note that for at least one of the searches the frontier has to be stored. So the space complexity is also $O((b^{d/2})$.

Comparing search strategies:

	Breadth first	Depth first	Iterative deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bm	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes

Complete?	Yes	No	Yes	Yes

Search Graphs

If the search space is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state. It is easy to consider a pathological example to see that the search space size may be exponential in the total number of states.

In many cases we can modify the search algorithm to avoid repeated state expansions. The way to avoid generating the same state again when not required, the search algorithm can be modified to check a node when it is being generated. If another node corresponding to the state is already in OPEN, the new node should be discarded. But what if the state was in OPEN earlier but has been removed and expanded? To keep track of this phenomenon, we use another list called CLOSED, which records all the expanded nodes. The newly generated node is checked with the nodes in CLOSED too, and it is put in OPEN if the corresponding state cannot be found in CLOSED. This algorithm is outlined below:

```
Graph search algorithm
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
        if fringe is empty return failure
        Node ® remove-first (fringe)
        if Node is a goal
            then return the path from initial state to Node
        else put Node in closed
                generate all successors of Node S
                for all nodes m in S
                    if m is not in fringe or closed
                        merge m into fringe
End Loop
```

But this algorithm is quite expensive. Apart from the fact that the CLOSED list has to be maintained, the algorithm is required to check every generated node to see if it is already there in OPEN or CLOSED. Unless there is a very efficient way to index nodes, this will require additional overhead for every node.

In many search problems, we can adopt some less computationally intense strategies. Such strategies do not stop duplicate states from being generated, but are able to reduce many of such cases.

The simplest strategy is to not return to the state the algorithm just came from. This simple strategy avoids many node re-expansions in 15-puzzle like problems.

A second strategy is to check that you do not create paths with cycles in them. This algorithm only needs to check the nodes in the current path so is much more efficient than the full checking algorithm. Besides this strategy can be employed successfully with depth first search and not require additional storage.

The third strategy is as outlined in the table. Do not generate any state that was ever created before.

Which strategy one should employ must be determined by considering the frequency of "loops" in the state space.

Iterative-Deepening A*

IDA* Algorithm

Iterative deepening A* or IDA* is similar to iterative-deepening depth-first, but with the following modifications:
The depth bound modified to be an f-limit

1. Start with limit = h(start)
2. Prune any node if f(node) > f-limit
3. Next f-limit=minimum cost of any node pruned

The cut-off for nodes expanded in an iteration is decided by the f-value of the nodes.

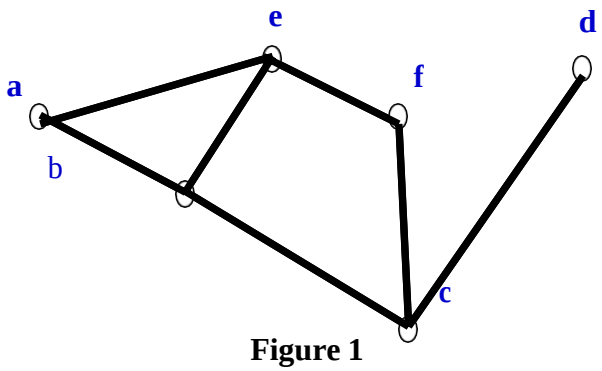
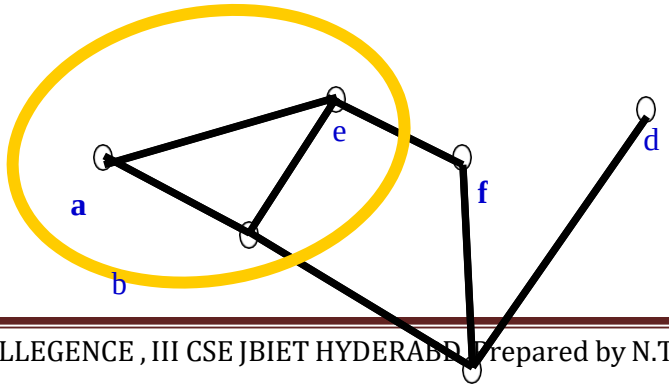


Figure 1

Consider the graph in Figure 3. In the first iteration, only node a is expanded. When a is expanded b and e are generated. The f value of both are found to be 15.

For the next iteration, a f-limit of 15 is selected, and in this iteration, a, b and c are expanded. This is illustrated in Figure 4.



c

Figure 2: f-limit = 15

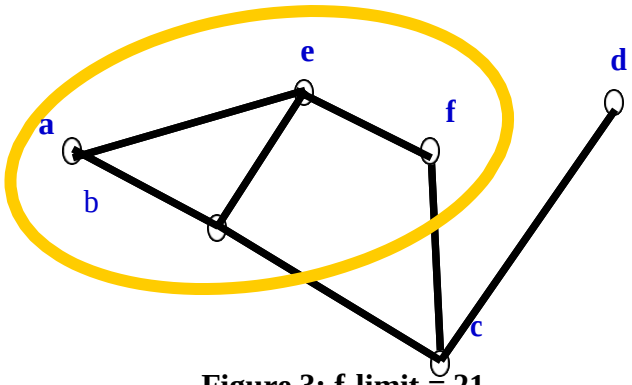


Figure 3: f-limit = 21

IDA* Analysis

IDA* is complete & optimal Space usage is linear in the depth of solution. Each iteration is depth first search, and thus it does not require a priority queue.

The number of nodes expanded relative to A* depends on # unique values of heuristic function. The number of iterations is equal tit h number of distinct f values less than or equal to C*.

- In problems like 8 puzzle using the Manhattan distance heuristic, there are few possible f values (f values are only integral in this case.). Therefore the number of node expansions in this case is close to the number of nodes A* expands.
- But in problems like traveling salesman (TSP) using real valued costs, : each f value may be unique, and many more nodes may need to be expanded. In the worst case, if all f values are distinct, the algorithm will expand only one new node per iteration, and thus if A* expands N nodes, the maximum number of nodes expanded by IDA* is 1+2+…+ N = O(N2)

Why do we use IDA*? In the case of A*, it I usually the case that for slightly larger problems, the algorithm runs out of main memory much earlier than the algorithm runs out of time. IDA* can be used in such cases as the space requirement is linear. In fact 15-puzzle problems can be easily solved by IDA*, and may run out of space on A*.

IDA* is not thus suitable for TSP type of problems. Also IDA* generates duplicate nodes in cyclic graphs. Depth first search strategies are not very suitable for graphs containing too many cycles.

Space required : O(bd)

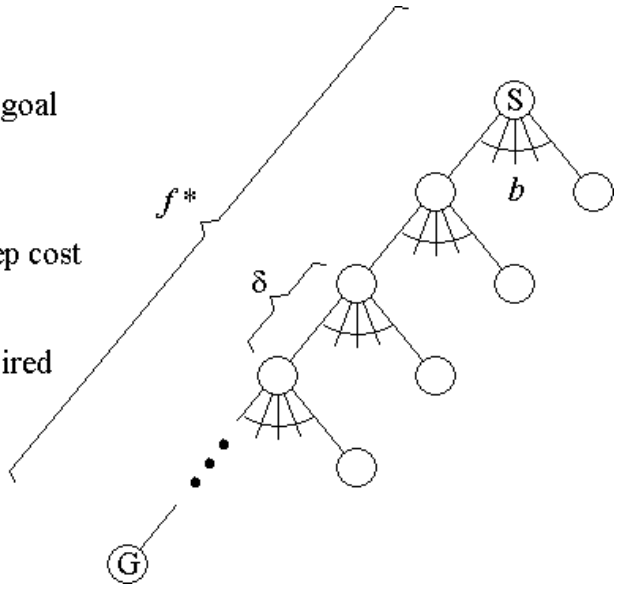
IDA* is complete, optimal, and optimally efficient (assuming a consistent, admissible heuristic), and requires only a polynomial amount of storage in the worst case:

f^* = optimal path cost to a goal

b = branching factor

δ = minimum operator step cost

$\frac{bf^*}{\delta}$ nodes of storage required



Other Memory limited heuristic search

IDA* uses very little memory
Other algorithms may use more memory for more efficient search.

RBFS: Recursive Breadth First Search

RBFS uses only linear space.
It mimics best first search.
It keeps track of the f-value of the best alternative path available from any ancestor of the current node.
If the current node exceeds this limit, the alternative path is explored.
RBFS remembers the f-value of the best leaf in the forgotten sub-tree.

```
RBFS (node: N, value: F(N), bound: B)

   IF f(N)>B, RETURN f(N)
   IF N is a goal, EXIT algorithm
   IF N has no children, RETURN infinity
   FOR each child Ni of N,
         IF f(N)<F(N), F[i] := MAX(F(N),f(Ni))
         ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
   IF only one child, F[2] := infinity
   WHILE (F[1] <= B and F[1] < infinity)
            F[1] := RBFS(N1, F[1], MIN(B, F[2]))
      insert Ni and F[1] in sorted order
RETURN F[1]
```

MA* and SMA*

MA* and SMA* are restricted memory best first search algorithms that utilize all the memory available.
The algorithm executes best first search while memory is available.
When the memory is full the worst node is dropped but the value of the forgotten node is backed up at the parent.

Local Search

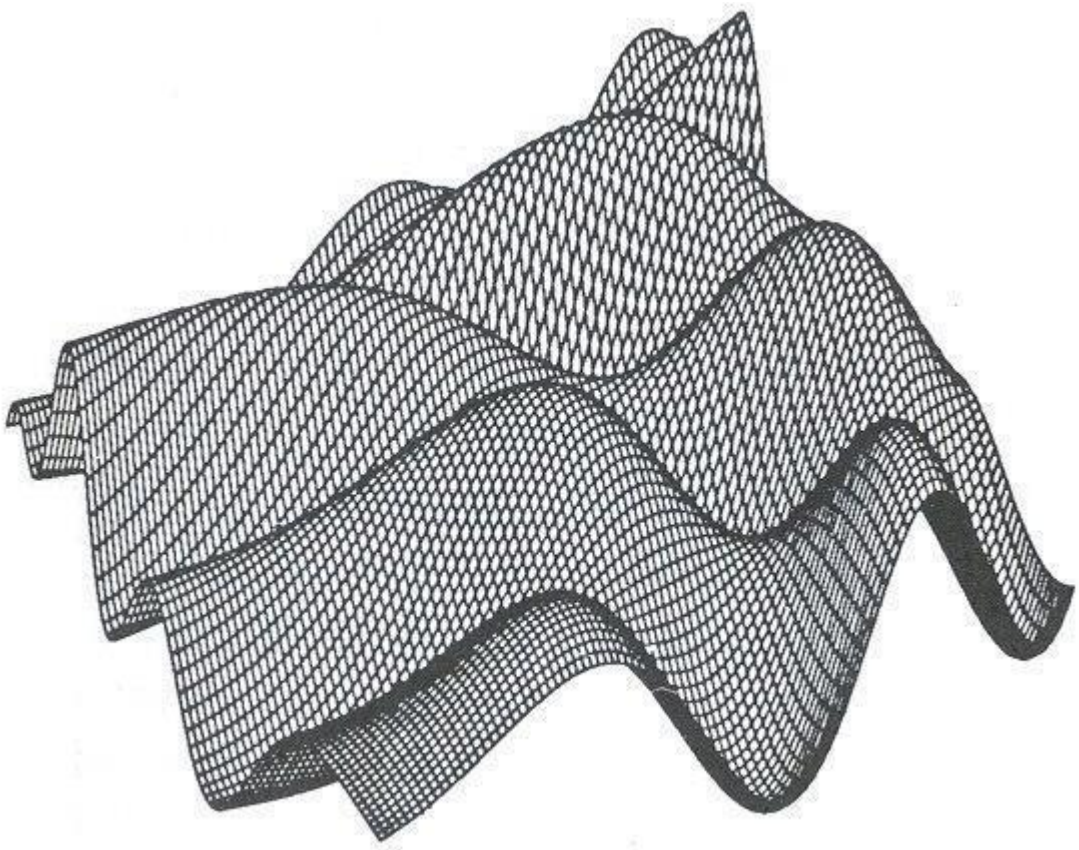
Local search methods work on complete state formulations. They keep only a small number of nodes in memory.
Local search is useful for solving optimization problems:

- Often it is easy to find a solution
- But hard to find the best solution

Algorithm goal:
find optimal configuration (e.g., TSP),

- Hill climbing
- Gradient descent
- Simulated annealing
- For some problems the state description contains all of the information relevant for a solution. Path to the solution is unimportant.
- Examples:
  - map coloring
  - 8-queens
  - cryptarithmetic

- Start with a state configuration that violates some of the constraints for being a solution, and make gradual modifications to eliminate the violations.
- One way to visualize iterative improvement algorithms is to imagine every possible state laid out on a landscape with the height of each state corresponding to its goodness. Optimal solutions will appear as the highest points. Iterative improvement works by moving around on the landscape seeking out the peaks by looking only at the local vicinity.



Iterative improvement

In many optimization problems, the path is irrelevant; the goal state itself is the solution. An example of such problem is to find configurations satisfying constraints (e.g., n-queens).

Algorithm:

- Start with a solution
- Improve it towards a good solution

Example:

N queens
Goal: Put n chess-game queens on an n x n board, with no two queens on the same row, column, or diagonal.

Example:

Chess board reconfigurations

Here, goal state is initially unknown but is specified by constraints that it must satisfy

Hill climbing (or gradient ascent/descent)

Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.

Note: minimizing a "value" function v(n) is equivalent to maximizing –v(n),

thus both notions are used interchangeably.

Hill climbing – example

Complete state formulation for 8 queens
Successor function: move a single queen to another square in the same column
Cost: number of pairs that are attacking each other.
Minimization problem

Hill climbing (or gradient ascent/descent)

- *Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.*

Note: minimizing a "value" function v(n) is equivalent to maximizing –v(n), thus both notions are used interchangeably.

- Algorithm:
    1. determine successors of current state
    2. choose successor of maximum goodness (break ties randomly)
    3. if goodness of best successor is less than current state's goodness, stop
    4. otherwise make best successor the current state and go to step 1
- No search tree is maintained, only the current state.
- Like greedy search, but only states directly reachable from the current state are considered.
- Problems:

Local maxima

Once the top of a hill is reached the algorithm will halt since every possible step leads down.

Plateaux

If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.

Ridges

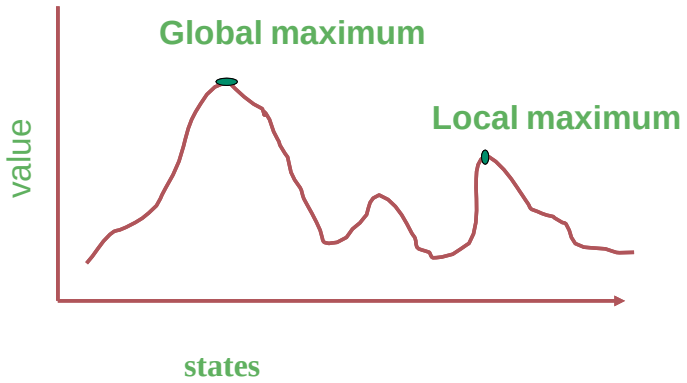
If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search.

- Shape of state space landscape strongly influences the success of the search process. A very spiky surface which is flat in between the spikes will be very difficult to solve.
- Can be combined with nondeterministic search to recover from local maxima.
- **Random-restart hill-climbing** is a variant in which reaching a local maximum causes the current state to be saved and the search restarted from a random point. After several restarts, return the best state found. With enough restarts, this method will find the optimal solution.
- **Gradient descent** is an inverted version of hill-climbing in which better states are represented by lower *cost* values. Local *minima* cause problems instead of local maxima.

Hill climbing - example

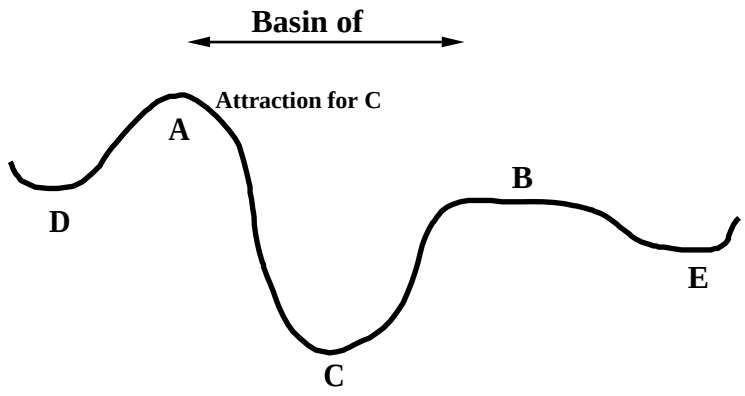
- *Complete state formulation for 8 queens*
  - Successor function: move a single queen to another square in the same column
  - Cost: number of pairs that are attacking each other.
- *Minimization problem*

- *Problem: depending on initial state, may get stuck in local extremum.*



Minimizing energy

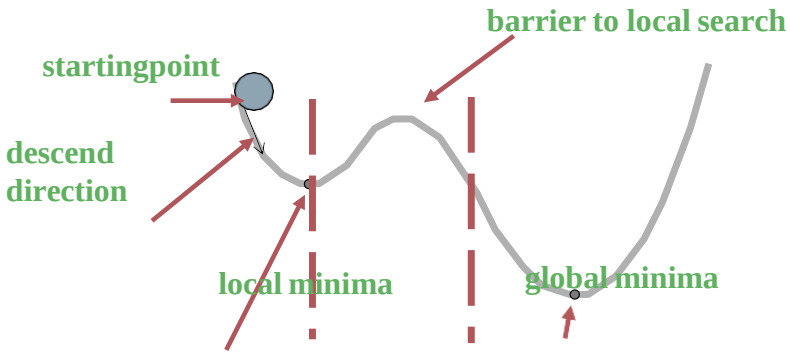
- Compare our state space to that of a physical system that is subject to natural interactions
- Compare our value function to the overall potential energy E of the system.
- On every updating, we have DE 0



Hence the dynamics of the system tend to move E toward a minimum.

We stress that there may be different such states — they are *local* minima. Global minimization is not guaranteed.

- *Question: How do you avoid this local minima?*



Consequences of Occasional Ascents

Simulated annealing: basic idea

- *From current state, pick a random successor state;*
- *If it has better value than current state, then "accept the transition," that is, use successor state as current state;*

Simulated annealing: basic idea

- *Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).*
- *So we accept to sometimes "un-optimize" the value function a little with a non-zero probability.*

- Instead of restarting from a random point, we can allow the search to take some downhill steps to try to escape local maxima.
- Probability of downward steps is controlled by **temperature** parameter.
- High temperature implies high chance of trying locally "bad" moves, allowing nondeterministic exploration.
- Low temperature makes search more deterministic (like hill-climbing).
- Temperature begins high and gradually decreases according to a predetermined

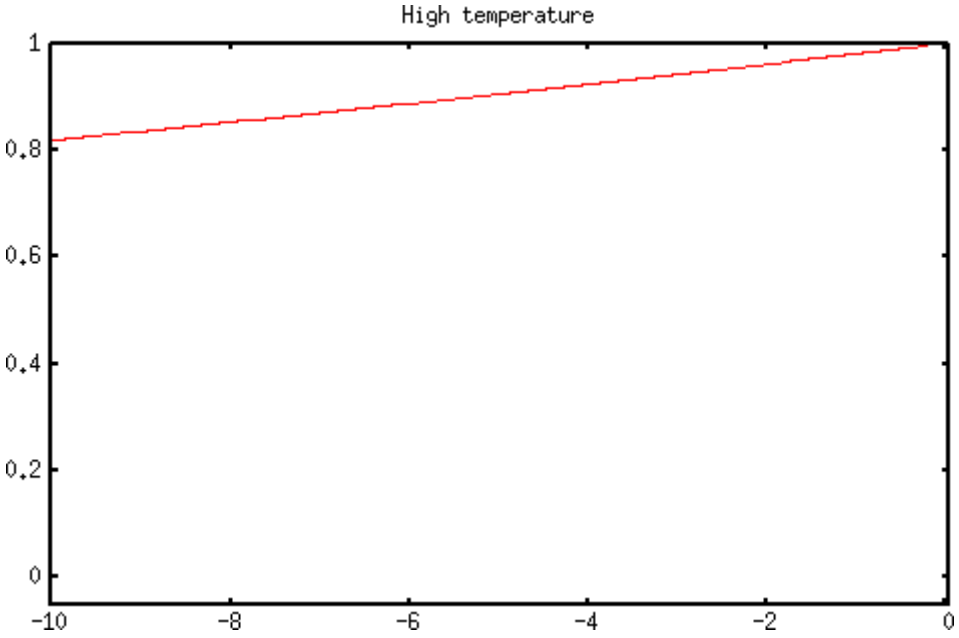
annealing schedule.

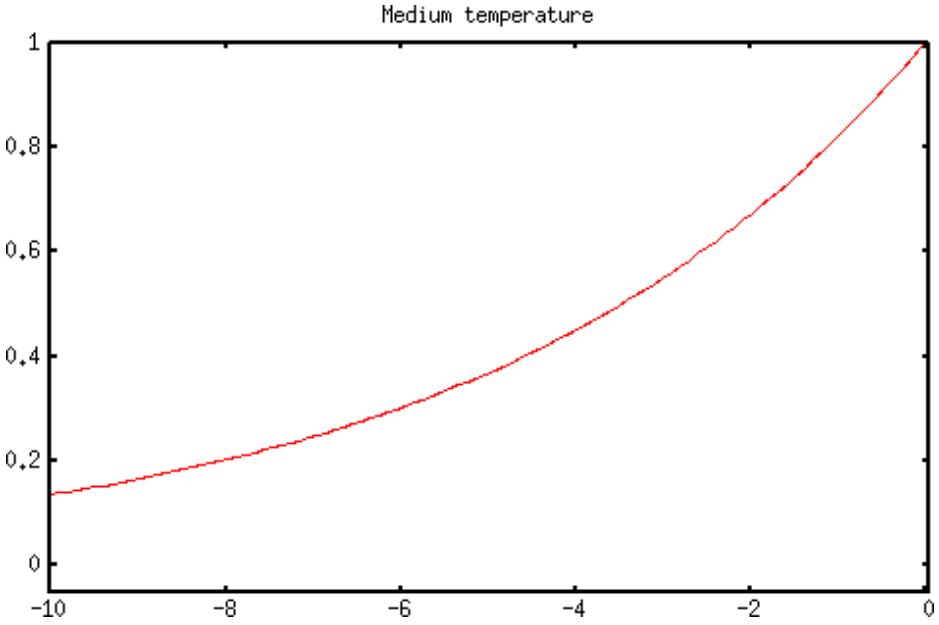
- Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
- If temperature is lowered slowly enough, an optimal solution will be found.
- In practice, this schedule is often too slow and we have to accept suboptimal solutions.

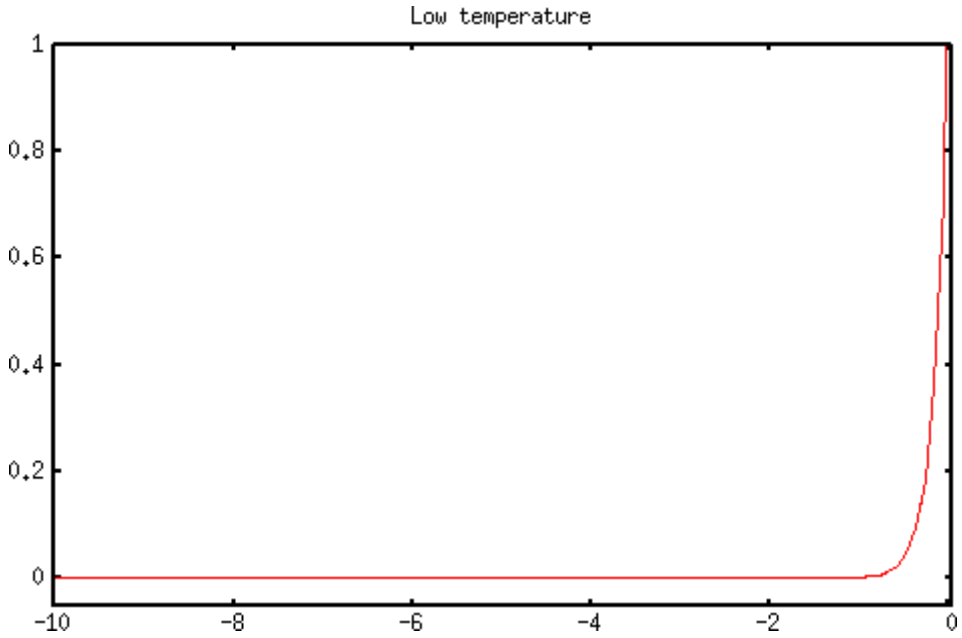
Algorithm:

```
set current to start state
for time = 1 to infinity {
   set Temperature to annealing_schedule[time]
   if Temperature = 0 {
      return current
   }
   randomly pick a next state from successors of current
   set ΔE to value(next) - value(current)
   if ΔE > 0 {
      set current to next
   } else {
      set current to next with probability e^(ΔE/Temperature)
   }
}
```

- Probability of moving downhill for negative ΔE values at different temperature ranges:


High temperature
1
0.8
0.6
0.4
0.2
0
-10
-8
-6
-4
-2
0


Medium temperature
1
0.8
0.6
0.4
0.2
0
-10
-8
-6
-4
-2
0



Other local search methods

- *Genetic Algorithms*

Questions for Lecture 6

1. Compare IDA* with A* in terms of time and space complexity.
2. Is hill climbing guaranteed to find a solution to the n-queens problem ?
3. Is simulated annealing guaranteed to find the optimum solution of an optimization problem like TSP ?

1. Suppose you have the following search space:

State	next	cost
A	B	4
A	C	1
B	D	3
B	E	8
C	C	0
C	D	2
C	F	6
D	C	2
D	E	4
E	G	2
F	G	8

UNIT-II

Adversarial Search

We will set up a framework for formulating a multi-person game as a search problem. We will consider games in which the players alternate making moves and try respectively to maximize and minimize a scoring function (also called utility function). To simplify things a bit, we will only consider games with the following two properties:

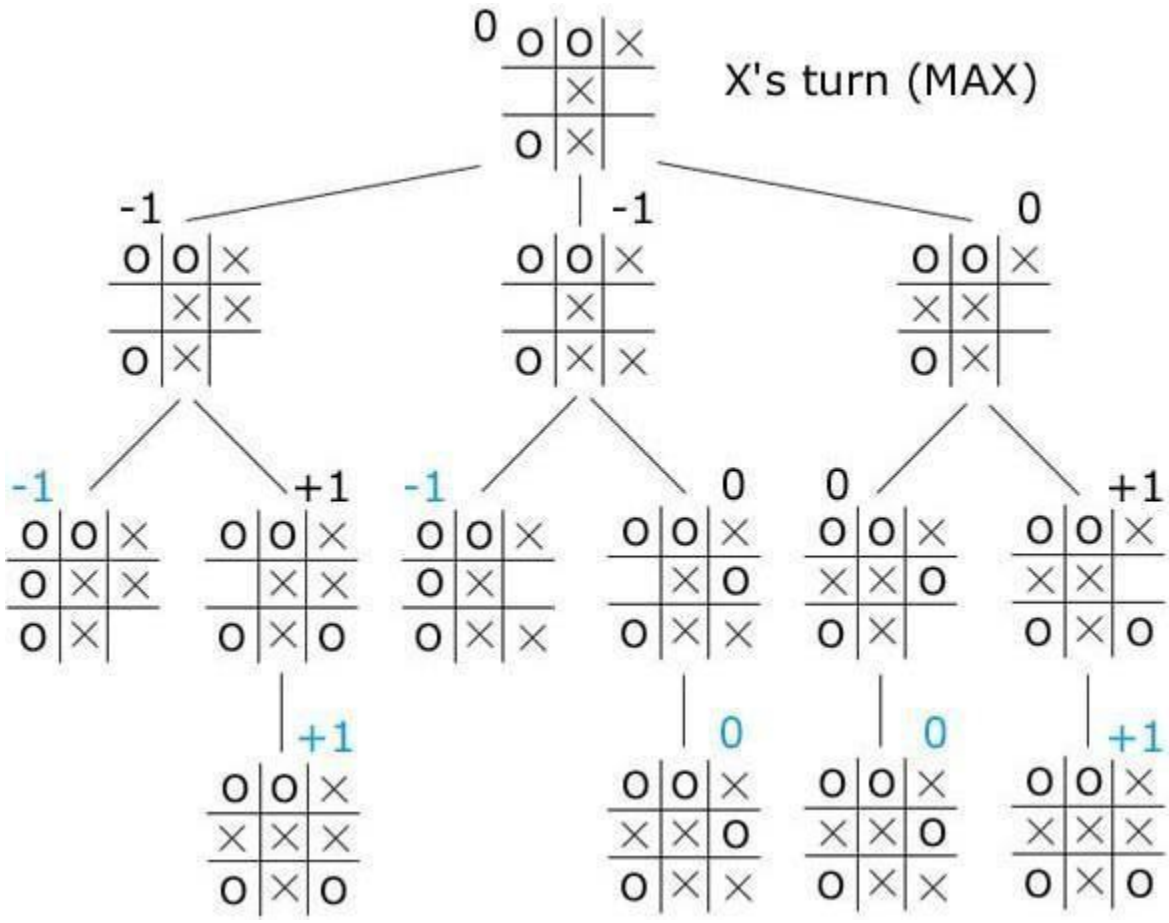
- Two player - we do not deal with coalitions, etc.
- Zero sum - one player's win is the other's loss; there are no cooperative victories

We also consider only perfect information games.

Game Trees

The above category of games can be represented as a tree where the nodes represent the current state of the game and the arcs represent the moves. The game tree consists of all possible moves for the current players starting at the root and all possible moves for the next player as the children of these nodes, and so forth, as far into the future of the game as desired. Each individual move by one player is called a "ply". The leaves of the game tree represent terminal positions as one where the outcome of the game is clear (a win, a loss, a draw, a payoff). Each terminal position has a score. High scores are good for one of the player, called the MAX player. The other player, called MIN player, tries to minimize the score. For example, we may associate 1 with a win, 0 with a draw and -1 with a loss for MAX.

Example : Game of Tic-Tac-Toe



Above is a section of a game tree for tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position. To score each position, we will give each position which is favorable for player 1 a positive number (the more positive, the more favorable). Similarly, we will give each position which is favorable for player 2 a negative number (the more negative, the more favorable). In our tic tac toe example, player 1 is 'X', player 2 is 'O', and the only three scores we will have are +1 for a win by 'X', -1 for a win by 'O', and 0 for a draw. Note here that the blue scores are the only ones that can be computed by looking at the current position.

Minimax Algorithm

Now that we have a way of representing the game in our program, how do we compute our optimal move? We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly. One algorithm for computing the best move is the minimax algorithm:

```
minimax(player,board)
    if(game over in current board position) return
        winner
    children = all legal moves for player from this board if(max's
    turn)
        return maximal score of calling minimax on all the children else (min's
    turn)
        return minimal score of calling minimax on all the children
```

If the game is over in the given position, then there is nothing to compute; minimax will simply return the score of the board. Otherwise, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where 'best' is the move leading to the board with the most positive score for player 1, and the board with the most negative score for player 2.

How long does this algorithm take? For a simple game like tic tac toe, not too long - it is certainly possible to search all possible positions. For a game like Chess or Go however, the running time is prohibitively expensive. In fact, to completely search either of these games, we would first need to develop interstellar travel, as by the time we finish analyzing a move the sun will have gone nova and the earth will no longer exist. Therefore, all real computer games will search, not to the end of the game, but only a few moves ahead. Of course, now the program must determine whether a certain board position is 'good' or 'bad' for a certainly player. This is often done using an *evaluation function*. This function is the key to a strong computer game. The depth bound search may stop just as things get interesting (e.g. in the middle of a piece exchange in chess. For this reason, the depth bound is usually extended to the end of an exchange to an quiescent state. The search may also tend to postpone bad news until after the depth bound leading to the *horizon effect*.

a. Draw the state space of this problem.
b. Assume that the initial state is **A** and the goal state is **G**. Show how each of the following search strategies would create a search tree to find a path from the initial state to the goal state:

   i. Uniform cost search
   ii. Greedy search
   iii. A* search

At each step of the search algorithm, show which node is being expanded, and the content of *fringe*. Also report the eventual solution found by each algorithm, and the solution cost.

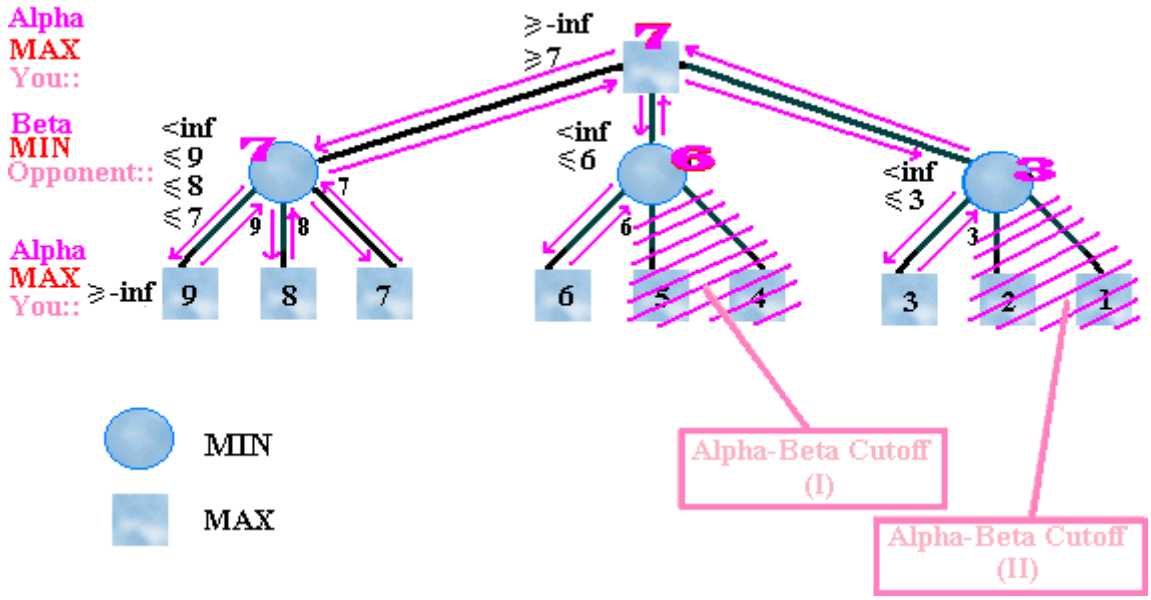
Alpha-Beta Pruning

ALPHA-BETA pruning is a method that reduces the number of nodes explored in Minimax strategy. It reduces the time required for the search and it must be restricted so that no time is to be wasted searching moves that are obviously bad for the current player. The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.

We proceed in the same (preorder) way as for the minimax algorithm. For the **MIN** nodes, the score computed starts with **+infinity** and decreases with time. For **MAX** nodes, scores computed starts with **-infinity** and increase with time.

The efficiency of the *Alpha-Beta* procedure depends on the order in which successors of a node are examined. If we were lucky, at a MIN node we would always consider the nodes in order from low to high score and at a MAX node the nodes in order from high to low score. In general it can be shown that in the most favorable circumstances the alpha-beta search opens as many leaves as minimax on a game tree with double its depth.

Here is an example of Alpha-Beta search:



Alpha-Beta algorithm:

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially alpha is negative infinity and beta is positive infinity. As the recursion progresses the "window" becomes smaller. When beta becomes less than alpha, it means that the current position cannot be the result of best play by both players and hence need not be explored further.

Pseudocode for the alpha-beta algorithm is given below.

```
evaluate (node, alpha, beta) if node
    is a leaf
        return the heuristic value of node if node
    is a minimizing node
        for each child of node
            beta = min (beta, evaluate (child, alpha, beta)) if beta <=
            alpha
                return beta
        return beta
    if node is a maximizing node for
        each child of node
            alpha = max (alpha, evaluate (child, alpha, beta)) if beta <=
            alpha
                return alpha
        return alpha
```

Constraint Satisfaction Problems

Constraint satisfaction problems or **CSP**s are mathematical problems where one must find states or objects that satisfy a number of *constraints* or criteria. A constraint is a restriction of the feasible solutions in an optimization problem.

Many problems can be stated as constraints satisfaction problems. *Here are some examples:*

Example 1: The n-Queen problem is the problem of putting n chess queens on an $n \times n$ chessboard such that none of them is able to capture any other using the standard chess queen's moves. The colour of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

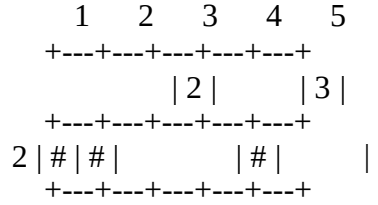
The problem was originally proposed in 1848 by the chess player Max Bazzel, and over the years, many mathematicians, including Gauss have worked on this puzzle. In 1874, S. Gunther proposed a method of finding solutions by using determinants, and J.W.L. Glaisher refined this approach.

The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **unique** solutions. The following table gives the number of solutions for *n* queens, both unique and distinct.

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	285,053
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

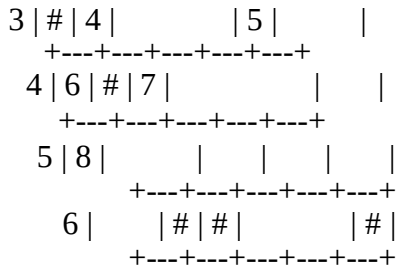
Note that the 6 queens puzzle has, interestingly, fewer solutions than the 5 queens puzzle!

Example 2: A crossword puzzle: We are to complete the puzzle



Given the list of words:

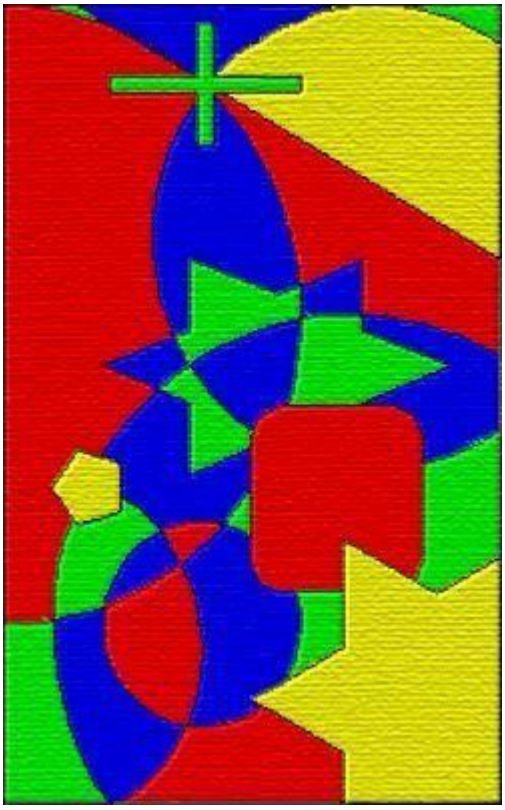
AFT	LASER
ALE	LEE
EEL	LINE
HEEL	SAILS



HIKE SHEET
HOSES STEER
KEEL TIE
KNOT

The numbers 1,2,3,4,5,6,7,8 in the crossword puzzle correspond to the words that will start at those locations.

Example 3: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using k colors, so that no two neighboring countries have the same color. Example of a four color map is shown below:



The **four color theorem** states that given any plane separated into regions, such as a political map of the countries of a state, the regions may be colored using no more than four colors in such a way that no two adjacent regions receive the same color. Two regions are called *adjacent* if they share a border segment, not just a point. Each region must be contiguous: that is, it may not consist of separate sections like such real countries as Angola, Azerbaijan, and the United States.

It is obvious that three colors are inadequate: this applies already to the map with one region surrounded by three other regions (even though with an even number of surrounding countries three colors are enough) and it is not at all difficult to prove that five colors are sufficient to color a map.

The four color theorem was the first major theorem to be proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand. Ultimately, one has to have faith in the correctness of the compiler and hardware executing the program used for the proof. The lack of

mathematical elegance was another factor, and to paraphrase comments of the time, "a good mathematical proof is like a poem — this is a telephone directory!"

Example 4: The **Boolean satisfiability problem (SAT)** is a decision problem considered in complexity theory. An instance of the problem is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true?

In mathematics, a formula of propositional logic is said to be **satisfiable** if truth-values can be assigned to its variables in a way that makes the formula true. The class of satisfiable propositional formulas is NP-complete. The propositional satisfiability problem (SAT), which decides whether or not a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design and verification.

The problem can be significantly restricted while still remaining NP-complete. By applying De Morgan's laws, we can assume that NOT operators are only applied directly to variables, not expressions; we refer to either a variable or its negation as a *literal*. For example, both x_1 and not(x_2) are literals, the first a *positive* literal and the second a *negative* literal. If we OR together a group of literals, we get a *clause*, such as (x_1 *or* not(x_2)). Finally, let us consider formulas that are a conjunction (AND) of clauses. We call this form conjunctive normal form. Determining whether a formula in this form is satisfiable is still NP-complete, even if each clause is limited to at most three literals. This last problem is called 3CNFSAT, 3SAT, or 3-satisfiability.

On the other hand, if we restrict each clause to at most two literals, the resulting problem, 2SAT, is in P. The same holds if every clause is a Horn clause; that is, it contains at most one positive literal.

Example 5: A cryptarithmetic problem: In the following pattern

```
  S E N D
  M O R
  E
========
= M O N E
Y
```

we have to replace each letter by a distinct digit so that the resulting sum is correct.

All these examples and other real life problems like time table scheduling, transport scheduling, floor planning etc are instances of the same pattern, captured by the following definition:

A **Constraint Satisfaction Problem** (CSP) is characterized by:

- a *set of variables* {x1, x2, .., xn},
- for each variable xi a *domain* Di with the possible values for that variable, and

- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied.

A CSP can easily be stated as a sentence in first order logic, of the form:

(exist x1)..(exist xn) (D1(x1) & .. Dn(xn) => C1..Cm)

Representation of CSP

A CSP is usually represented as an undirected graph, called ***Constraint Graph*** where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs.

A constraint can affect any number of variables form 1 to n (n is the number of variables in the problem). If all the constraints of a CSP are binary, the variables and constraints can be represented in a constraint graph and the constraint satisfaction algorithm can exploit the graph search techniques.

The conversion of arbitrary CSP to an equivalent binary CSP is based on the idea of introducing a new variable that encapsulates the set of constrained variables. This newly introduced variable, we call it an encapsulated variable, has assigned a domain that is a Cartesian product of the domains of individual variables. Note, that if the domains of individual variables are finite than the Cartesian product of the domains, and thus the resulting domain, is still finite.

Now, arbitrary n-ary constraint can be converted to equivalent unary constraint that constrains the variable which appears as an encapsulation of the original individual variables. As we mentioned above, this unary constraint can be immediately satisfied by reducing the domain of encapsulated variable. Briefly speaking, n-ary constraint can be substituted by an encapsulated variable with the domain corresponding to the constraint.

This is interesting because any constraint of higher arity can be expressed in terms of binary constraints. Hence, binary CSPs are representative of all CSPs.

Example 2 revisited: We introduce a variable to represent each word in the puzzle. So we have the variables:

VARIABLE	STARTING CELL	DOMAIN
1ACROSS	1	{HOSES, LASER, SAILS, SHEET, STEER}
4ACROSS	4	{HEEL, HIKE, KEEL, KNOT, LINE}
7ACROSS	7	{AFT, ALE, EEL, LEE, TIE}
8ACROSS	8	{HOSES, LASER, SAILS, SHEET, STEER}
2DOWN	2	{HOSES, LASER, SAILS, SHEET, STEER}
3DOWN	3	{HOSES, LASER, SAILS, SHEET, STEER}
5DOWN	5	{HEEL, HIKE, KEEL, KNOT, LINE}
6DOWN	6	{AFT, ALE, EEL, LEE, TIE}

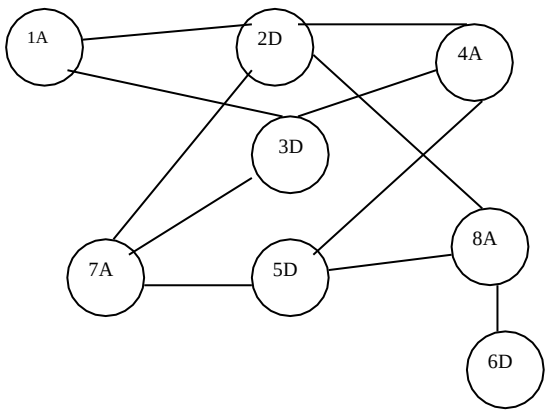
The domain of each variable is the list of words that may be the value of that variable. So, variable 1ACROSS requires words with five letters, 2DOWN requires words with five letters, 3DOWN requires words with four letters, etc. Note that since each domain has 5 elements and there are 8 variables, the total number of states to consider in a naive approach is 5^8 = 390,625.

The constraints are all binary constraints:

1ACROSS[3] = 2DOWN[1] i.e. the third letter of 1ACROSS must be equal to the first letter of 2DOWN

1ACROSS[5] = 3DOWN[1]
4ACROSS[2] = 2DOWN[3]
4ACROSS[3] = 5DOWN[1]
4ACROSS[4] = 3DOWN[3]
7ACROSS[1] = 2DOWN[4]
7ACROSS[2] = 5DOWN[2]
7ACROSS[3] = 3DOWN[4]
8ACROSS[1] = 6DOWN[2]
8ACROSS[3] = 2DOWN[5]
8ACROSS[4] = 5DOWN[3]
8ACROSS[5] = 3DOWN[5]

The corresponding graph is:



Solving CSPs

Next we describe four popular solution methods for CSPs, namely, *Generate-and-Test*, *Backtracking*, *Consistency Driven, and Forward Checking*.

Generate and Test

We generate one by one all possible complete variable assignments and for each we test if it satisfies all constraints. The corresponding program structure is very simple, just nested loops, one per variable. In the innermost loop we test each constraint. In most situation this method is intolerably slow.

Backtracking

We order the variables in some fashion, trying to place first the variables that are more highly constrained or with smaller ranges. This order has a great impact on the efficiency of solution algorithms and is examined elsewhere. We start assigning values to variables. We check constraint satisfaction at the earliest possible time and extend an assignment if the constraints involving the currently bound variables are satisfied.

Example 2 Revisited: In our crossword puzzle we may order the variables as follows: 1ACROSS, 2DOWN, 3DOWN, 4ACROSS, 7ACROSS, 5DOWN, 8ACROSS, 6DOWN. Then we start the assignments:

```
1ACROSS     <- HOSES
2DOWN       <- HOSES      => failure, 1ACROSS[3] not equal to
2DOWN[1]
            <- LASER      => failure
            <- SAILS
3DOWN       <- HOSES      => failure
            <- LASER      => failure
            <- SAILS
4ACROSS     <- HEEL       => failure
            <- HIKE       => failure
            <- KEEL       => failure
            <- KNOT       => failure
            <- LINE       => failure, backtrack
3DOWN       <- SHEET
4ACROSS     <- HEEL

7ACROSS <- AFT        => failure
.....
```

What we have shown is called *Chronological Backtracking*, whereby variables are unbound in the inverse order to the the order used when they were bound. *Dependency Directed Backtracking* instead recognizes the cause of failure and backtracks to one of the causes of failure and skips over the intermediate variables that did not cause the failure.

The following is an easy way to do dependency directed backtracking. We keep track at each variable of the variables that precede it in the backtracking order and to which it is connected directly in the constraint graph. Then, when instantiation fails at a variable, backtracking goes in order to these variables skipping over all other intermediate variables.

Notice then that we will backtrack at a variable up to as many times as there are preceding neighbors. [This number is called the *width* of the variable.] The time complexity of the backtracking algorithm grows when it has to backtrack often. Consequently there is a real gain when the variables are ordered so as to minimize their largest width.

Consistency Driven Techniques

Consistency techniques effectively rule out many inconsistent labeling at a very early stage, and thus cut short the search for consistent labeling. These techniques have since proved to be effective on a wide variety of hard search problems. The consistency techniques are deterministic, as opposed to the search which is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to done. Nevertheless, the consistency techniques are rarely used alone to solve constraint satisfaction problem completely (but they could).

In binary CSPs, various consistency techniques for constraint graphs were introduced to prune the search space. The consistency-enforcing algorithm makes any partial solution of a small subnetwork extensible to some surrounding network. Thus, the potential inconsistency is detected as soon as possible.

Node Consistency

The simplest consistency technique is refered to as node consistency and we mentioned it in the section on binarization of constraints. The node representing a variable V in constraint graph is node consistent if for every value x in the current domain of V, each unary constraint on V is satisfied.

If the domain D of a variable V containts a value "a" that does not satisfy the unary constraint on V, then the instantiation of V to "a" will always result in immediate failure. Thus, the node inconsistency can be eliminated by simply removing those values from the domain D of each variable V that do not satisfy unary constraint on V.

Arc Consistency

If the constraint graph is node consistent then unary constraints can be removed because they all are satisfied. As we are working with the binary CSP, there remains to ensure consistency of binary constraints. In the constraint graph, binary constraint corresponds to arc, therefore this type of consistency is called arc consistency.

Arc (V_i,V_j) is **arc consistent** if for every value x the current domain of V_i there is some value y in the domain of V_j such that V_i=x and V_j=y is permitted by the binary constraint between V_i and V_j. Note, that the concept of arc-consistency is directional, i.e., if an arc (V_i,V_j) is consistent, than it does not automatically mean that (V_j,V_i) is also consistent.

Clearly, an arc (V_i,V_j) can be made consistent by simply deleting those values from the domain of V_i for which there does not exist corresponding value in the domain of D_j such that the binary constraint between V_i and V_j is satisfied (note, that deleting of such values does not eliminate any solution of the original CSP).

The following algorithm does precisely that.

Algorithm REVISE

```
procedure REVISE(Vi,Vj)
   DELETE <- false;
   for each X in Di do
      if there is no such Y in Dj such that (X,Y) is consistent, then
           delete X from Di;
           DELETE <- true;
      endif;
   endfor;
   return DELETE;
end REVISE
```

To make every arc of the constraint graph consistent, it is not sufficient to execute REVISE for each arc just once. Once REVISE reduces the domain of some variable V_i, then each previously revised arc (V_j,V_i) has to be revised again, because some of the members of the domain of V_j may no longer be compatible with any remaining members of the revised domain of V_i. The following algorithm, known as **AC-1**, does precisely that.

Algorithm AC-1

```
procedure AC-1
   Q <- {(Vi,Vj) in arcs(G),i#j}; repeat
      CHANGE <- false;
      for each (Vi,Vj) in Q do
         CHANGE <- REVISE(Vi,Vj) or CHANGE;
      endfor
   until not(CHANGE)
end AC-1
```

This algorithm is not very efficient because the succesfull revision of even one arc in some iteration forces all the arcs to be revised again in the next iteration, even though only a small number of them are really affected by this revision. Visibly, the only arcs

affected by the reduction of the domain of V_k are the arcs (V_i,V_k). Also, if we revise the

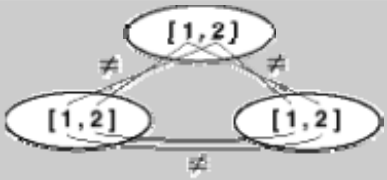
arc (V_k,V_m) and the domain of V_k is reduced, it is not necessary to re-revise the arc (V_m,V_k) because non of the elements deleted from the domain of V_k provided support for any value in the current domain of V_m. The following variation of arc consistency algorithm, called AC-3, removes this drawback of AC-1 and performs re-revision only for those arcs that are possibly affected by a previous revision.

Algorithm AC-3

```
procedure AC-3
   Q <- {(Vi,Vj) in arcs(G),i#j}; while
   not Q empty
      select and delete any arc (Vk,Vm) from Q; if
      REVISE(Vk,Vm) then
         Q <- Q union {(Vi,Vk) such that (Vi,Vk) in
arcs(G),i#k,i#m}
      endif
   endwhile
end AC-3
```

When the algorithm AC-3 revises the edge for the second time it re-tests many pairs of values which are already known (from the previous iteration) to be consistent or inconsistent respectively and which are not affected by the reduction of the domain. As this is a source of potential inefficiency, the algorithm **AC-4** was introduced to refine handling of edges (constraints). The algorithm works with indiviual pairs of values as the following example shows.

Example:



First, the algorithm AC-4 initializes its internal structures which are used to remember pairs of consistent (inconsistent) values of incidental variables (nodes) - structure $S_{i,a}$. This initialization also counts "supporting" values from the domain of incindental variable - structure $counter_{(i,j),a}$ - and it removes those values which have no support. Once the value is removed from the domain, the algorithm adds the pair <Variable,Value> to the list Q for re-revision of affected values of corresponding variables.

Algorithm INITIALIZE

```
procedure INITIALIZE
   Q <- {};
   S <- {};        % initialize each element of structure S
   for each (Vi,Vj) in arcs(G) do                % (Vi,Vj) and (Vj,Vi) are
same elements
      for each a in Di do total
         <- 0;
         for each b in Dj do
            if (a,b) is consistent according to the constraint (Vi,Vj) then
               total <- total+1;
               Sj,b <- Sj,b union {<i,a>}; endif
         endfor;
         counter[(i,j),a] <- total; if
         counter[(i,j),a]=0 then
            delete a from Di;
            Q <- Q union {<i,a>};
         endif;
      endfor;
   endfor;
   return Q;
end INITIALIZE
```

After the initialization, the algorithm AC-4 performs re-revision only for those pairs of values of incindental variables that are affected by a previous revision.

Algorithm AC-4

```
procedure AC-4
   Q <- INITIALIZE;
   while not Q empty
      select and delete any pair <j,b> from Q; for each
      <i,a> from Sj,b do
         counter[(i,j),a] <- counter[(i,j),a] - 1;
         if counter[(i,j),a]=0 & a is still in Di then delete a from
            Di;
            Q <- Q union {<i,a>};
         endif
      endfor
   endwhile
end AC-4
```

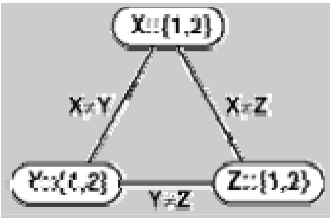
Both algorithms, AC-3 and AC-4, belong to the most widely used algorithms for maintaining arc consistency. It should be also noted that there exist other algorithms AC-5, AC-6, AC-7 etc. but their are not used as frequently as AC-3 or AC-4.

Maintaining arc consistency removes many inconsistencies from the constraint graph but

is any (complete) instantiation of variables from current (reduced) domains a solution to the CSP? If the domain size of each variable becomes one, then the CSP has exactly one solution which is obtained by assigning to each variable the only possible value in its

domain. Otherwise, the answer is no in general. The following example shows such a case where the constraint graph is arc consistent, domains are not empty but there is still no solution satisfying all constraints.

Example:



This constraint graph is arc consistent but there is no solution that satisfies all the constraints.

Path Consistency (K-Consistency)

Given that arc consistency is not enough to eliminate the need for backtracking, is there another stronger degree of consistency that may eliminate the need for search? The above example shows that if one extends the consistency test to two or more arcs, more inconsistent values can be removed.

A graph is **K-consistent** if the following is true: Choose values of any K-1 variables that satisfy all the constraints among these variables and choose any K^{th} variable. Then there exists a value for this Kth variable that satisfies all the constraints among these K variables. A graph is **strongly K-consistent** if it is J-consistent for all J<=K.

Node consistency discussed earlier is equivalent to strong 1-consistency and arc-consistency is equivalent to strong 2-consistency (arc-consistency is usually assumed to include node-consistency as well). Algorithms exist for making a constraint graph strongly K-consistent for K>2 but in practice they are rarely used because of efficiency issues. The exception is the algorithm for making a constraint graph strongly 3-consistent that is usually refered as **path consistency**. Nevertheless, even this algorithm is too hungry and a weak form of path consistency was introduced.

A node representing variable V_i is **restricted path consistent** if it is arc-consistent, i.e., all arcs from this node are arc-consistent, and the following is true: For every value a in the domain D_i of the variable V_i that has *just one supporting value* b from the domain of incidental variable V_j there exists a value c in the domain of other incidental variable V_k such that (a,c) is permitted by the binary constraint between V_i and V_k, and (c,b) is permitted by the binary constraint between V_k and V_j.

The algorithm for making graph restricted path consistent can be naturally based on AC-4 algorithm that counts the number of supporting values. Although this algorithm removes more inconsistent values than any arc-consistency algorithm it does not eliminate the need for search in general. Clearly, if a constraint graph containing n nodes is strongly n-consistent, then a solution to the CSP can be found without any search. But the worst-case complexity of the algorithm for obtaining n-consistency in a n-node constraint graph

is also exponential. If the graph is (strongly) K-consistent for K<n, then in general, backtracking cannot be avoided, i.e., there still exist inconsistent values.

Forward Checking

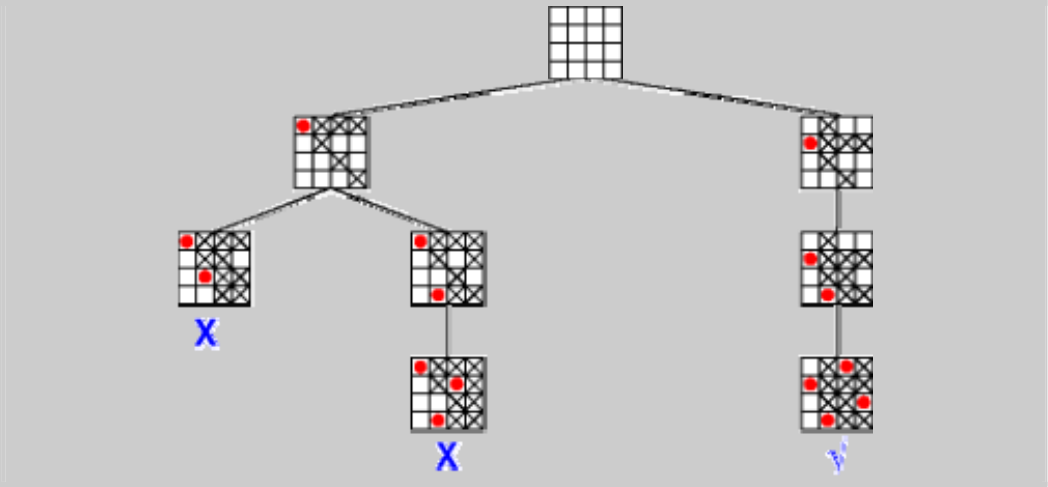
Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

Algorithm AC-3 for Forward Checking

```
procedure AC3-FC(cv)
   Q <- {(Vi,Vcv) in arcs(G),i>cv};
   consistent <- true;
   while not Q empty & consistent
      select and delete any arc (Vk,Vm) from Q; if
      REVISE(Vk,Vm) then
         consistent <- not Dk empty
      endif
   endwhile
   return consistent end
AC3-FC
```

Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and (hopefully) the overall amount of work done. But it should be noted that forward checking does more work when each assignment is added to the current partial solution.

Example: (4-queens problem and FC)



Forward checking is almost always a much better choice than simple backtracking.

Look Ahead

Forward checking checks only the constraints between the current variable and the future variables. So why not to perform full arc consistency that will further reduces the domains and removes possible conflicts? This approach is called (full) look ahead or maintaining arc consistency (MAC).

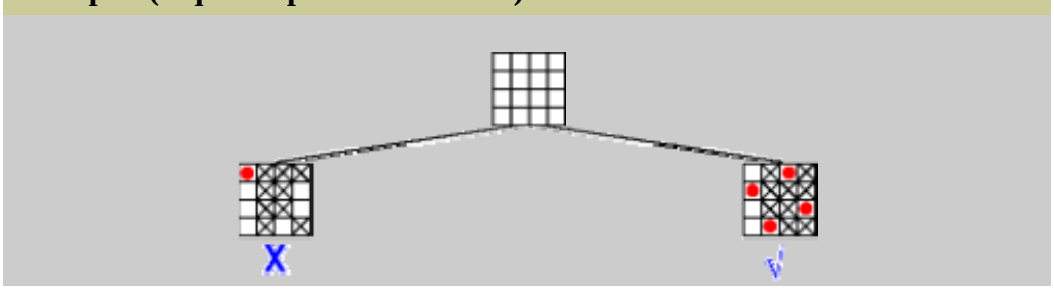
The advantage of look ahead is that it detects also the conflicts between future variables and therefore allows branches of the search tree that will lead to failure to be pruned earlier than with forward checking. Also as with forward checking, whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no necessary.

Algorithm AC-3 for Look Ahead

```
procedure AC3-LA(cv)
   Q <- {(Vi,Vcv) in arcs(G),i>cv};
   consistent <- true;
   while not Q empty & consistent
      select and delete any arc (Vk,Vm) from Q; if
      REVISE(Vk,Vm) then
         Q <- Q union {(Vi,Vk) such that (Vi,Vk) in
arcs(G),i#k,i#m,i>cv}
         consistent <- not Dk empty
      endif
   endwhile
   return consistent end
AC3-LA
```

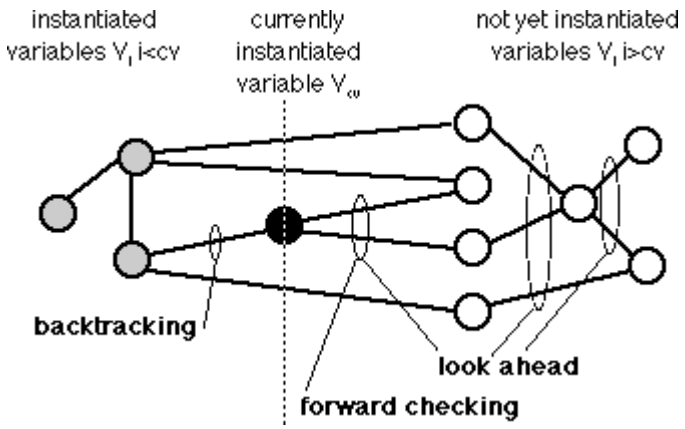
Look ahead prunes the search tree further more than forward checking but, again, it should be noted that look ahead does even more work when each assignment is added to the current partial solution than forward checking.

Example: (4-queens problem and LA)



Comparison of Propagation Techniques

The following figure shows which constraints are tested when the above described propagation techniques are applied.



More constraint propagation at each node will result in the search tree containing fewer nodes, but the overall cost may be higher, as the processing at each node will be more expensive. In one extreme, obtaining strong n-consistency for the original problem would completely eliminate the need for search, but as mentioned before, this is usually more expensive than simple backtracking. Actually, in some cases even the full look ahead may be more expensive than simple backtracking. That is the reason why forward checking and simple backtracking are still used in applications.

Variable and Value Ordering

A search algorithm for constraint satisfaction requires the order in which variables are to be considered to be specified as well as the order in which the values are assigned to the

variable on backtracking. Choosing the right order of variables (and values) can noticeably improve the efficiency of constraint satisfaction.

Variable Ordering

Experiments and analysis of several researchers have shown that the ordering in which variables are chosen for instantiation can have substantial impact on the complexity of backtrack search. The ordering may be either

- a *static ordering*, in which the order of the variables is specified before the search begins, and it is not changed thereafter, or
- a *dynamic ordering*, in which the choice of next variable to be considered at any point depends on the current state of the search.

Dynamic ordering is not feasible for all search algorithms, e.g., with simple backtracking there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, with forward checking, the current state includes the domains of the variables as they have been pruned by the current set of instantiations, and so it is possible to base the choice of next variable on this information.

Several heuristics have been developed and analyzed for selecting variable ordering. The most common one is based on the "**first-fail**" principle, which can be explained as

"*To succeed, try first where you are most likely to fail.*"

In this method, the variable with the fewest possible remaining alternatives is selected for instantiation. Thus the order of variable instantiations is, in general, different in different branches of the tree, and is determined dynamically. This method is based on assumption that any value is equally likely to participate in a solution, so that the more values there are, the more likely it is that one of them will be a successful one.

The first-fail principle may seem slightly misleading, after all, we do not want to fail. The reason is that if the current partial solution does not lead to a complete solution, then the sooner we discover this the better. Hence encouraging early failure, if failure is inevitable, is beneficial in the long term. On the other end, if the current partial solution can be extended to a complete solution, then every remaining variable must be instantiated and the one with smallest domain is likely to be the most difficult to find a value for (instantiating other variables first may further reduce its domain and lead to a failure). Hence the principle could equally well be stated as:

"*Deal with hard cases first: they can only get more difficult if you put them off.*"

This heuristic should reduce the average depth of branches in the search tree by triggering early failure.

Another heuristic, that is applied when all variables have the same number of values, is to choose the variable which participates in most constraints (in the absence of more specific information on which constraints are likely to be difficult to satisfy, for instance). This heuristic follows also the principle of dealing with hard cases first.

There is also a heuristic for static ordering of variables that is suitable for simple backtracking. This heuristic says: choose the variable which has the largest number of constraints with the past variables. For instance, during solving graph coloring problem, it is reasonable to assign color to the vertex which has common arcs with already colored vertices so the conflict is detected as soon as possible.

Value Ordering

Once the decision is made to instantiate a variable, it may have several values available. Again, the order in which these values are considered can have substantial impact on the time to find the first solution. However, if all solutions are required or there are no solutions, then the value ordering is indifferent.

A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to death ends. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed, in that case, every value for the current variable will eventually have to be considered, and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we want to choose a value which is likely to succeed, and unlikely to lead to a conflict. So, we apply the **"succeed first"** principle.

One possible heuristic is to prefer those values that maximize the number of options available. Visibly, the algorithm AC-4 is good for using this heuristic as it counts the supporting values. It is possible to count "promise" of each value, that is the product of the domain sizes of the future variables after choosing this value (this is an upper bound on the number of possible solutions resulting from the assignment). The value with highest promise should be chosen. Is also possible to calculate the percentage of values in future domains which will no longer be usable. The best choice would be the value with lowest cost.

Another heuristic is to prefer the value (from those available) that leads to an easiest to solve CSP. This requires to estimate the difficulty of solving a CSP. One method propose to convert a CSP into a tree-structured CSP by deleting a minimum number of arcs and then to find all solutions of the resulting CSP (higher the solution count, easier the CSP).

For randomly generated problems, and probably in general, the work involved in assessing each value is not worth the benefit of choosing a value which will on average be more likely to lead to a solution than the default choice. In particular problems, on the other hand, there may be information available which allows the values to be ordered according to the principle of choosing first those most likely to succeed.

Heuristic Search in CSP

In the last few years, greedy local search strategies became popular, again. These algorithms incrementally alter inconsistent value assignments to all the variables. They use a "repair" or "hill climbing" metaphor to move towards more and more complete solutions. To avoid getting stuck at "local optima" they are equipped with various heuristics for randomizing the search. Their stochastic nature generally voids the guarantee of "completeness" provided by the systematic search methods.

The local search methodology uses the following terms:

- **state (configuration):** one possible assignment of all variables; the number of states is equal to the product of domains' sizes
- **evaluation value:** the number of constraint violations of the state (sometimes weighted)
- **neighbor:** the state which is obtained from the current state by changing one variable value
- **local-minimum:** the state that is not a solution and the evaluation values of all of its neighbors are larger than or equal to the evaluation value of this state
- **strict local-minimum:** the state that is not a solution and the evaluation values of all of its neighbors are larger than the evaluation value of this state
- **non-strict local-minimum:** the state that is a local-minimum but not a strict local-minimum.

Hill-Climbing

Hill-climbing is probably the most known algorithm of local search. The idea of hill-climbing is:

1. start at randomly generated state
2. move to the neighbor with the best evaluation value
3. if a strict local-minimum is reached then restart at other randomly generated state.

This procedure repeats till the solution is found. In the algorithm, that we present here, the parameter Max_Flips is used to limit the maximal number of moves between restarts which helps to leave non-strict local-minimum.

Algorithm Hill-Climbing

```
procedure hill-climbing(Max_Flips)
    restart: s <- random valuation of variables; for j:=1 to
    Max_Flips do
       if eval(s)=0 then return s endif; if s is a
       strict local minimum then
             goto restart
       else
          s <- neighborhood with smallest evaluation value endif
    endfor
   goto restart end
hill-climbing
```

Note, that the hill-climbing algorithm has to explore all neighbors of the current state before choosing the move. This can take a lot of time.

Min-Conflicts

To avoid exploring all neighbors of the current state some heuristics were proposed to find a next move. *Min-conflicts* heuristics chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints).

Algorithm Min-Conflicts

```
procedure MC(Max_Moves)
    s <- random valuation of variables;
    nb_moves <- 0;
    while eval(s)>0 & nb_moves<Max_Moves do choose
       randomly a variable V in conflict;
       choose a value v' that minimizes the number of conflicts for V; if v' # current
       value of V then
          assign v' to V; nb_moves <-
          nb_moves+1;
       endif
    endwhile
    return s
end MC
```

Note, that the pure min-conflicts algorithm presented above is not able to leave local-minimum. In addition, if the algorithm achieves a strict local-minimum it does not

perform any move at all and, consequently, it does not terminate.

GSAT

GSAT is a greedy local search procedure for satisfying logic formulas in a conjunctive normal form (CNF). Such problems are called SAT or k-SAT (*k* is a number of literals in each clause of the formula) and are known to be NP-c (each NP-hard problem can be transformed to NP-complex problem).

The procedure starts with an arbitrary instantiation of the problem variables and offers to reach the highest satisfaction degree by succession of small transformations called repairs or flips (flipping a variable is a changing its value).

Algorithm GSAT

```
procedure GSAT(A,Max_Tries,Max_Flips)
    A: is a CNF formula
    for i:=1 to Max_Tries do
        S <- instantiation of variables for j:=1
        to Max_Iter do
            if A satisfiable by S then return
                S
            endif
           V <- the variable whose flip yield the most important raise in the number of
satisfied clauses;
            S <- S with V flipped;
        endfor
    endfor
   return the best instantiation found end
GSAT
```

Knowledge Representation and Reasoning

Intelligent agents should have capacity for:

- **Perceiving**, that is, acquiring information from environment,
- **Knowledge Representation**, that is, representing its understanding of the world,
- **Reasoning**, that is, inferring the implications of what it knows and of the choices it has, and
- **Acting**, that is, choosing what it want to do and carry it out.

Representation of knowledge and the reasoning process are central to the entire field of artificial intelligence. The primary component of a knowledge-based agent is its knowledge-base. A knowledge-base is a set of sentences. Each sentence is expressed in a language called the knowledge representation language. Sentences represent some assertions about the world. There must mechanisms to derive new sentences from old ones. This process is known as inferencing or reasoning. Inference must obey the primary requirement that the new sentences should follow logically from the previous ones.

Logic is the primary vehicle for representing and reasoning about knowledge. Specifically, we will be dealing with formal logic. The advantage of using formal logic as a language of AI is that it is precise and definite. This allows programs to be written which are declarative - they describe what is true and not how to solve problems. This also allows for automated reasoning techniques for general purpose inferencing.

This, however, leads to some severe limitations. Clearly, a large portion of the reasoning carried out by humans depends on handling knowledge that is uncertain. Logic cannot represent this uncertainty well. Similarly, natural language reasoning requires inferring hidden state, namely, the intention of the speaker. When we say, "One of the wheel of the car is flat.", we know that it has three wheels left. Humans can cope with virtually infinite variety of utterances using a finite store of commonsense knowledge. Formal logic has difficulty with this kind of ambiguity.

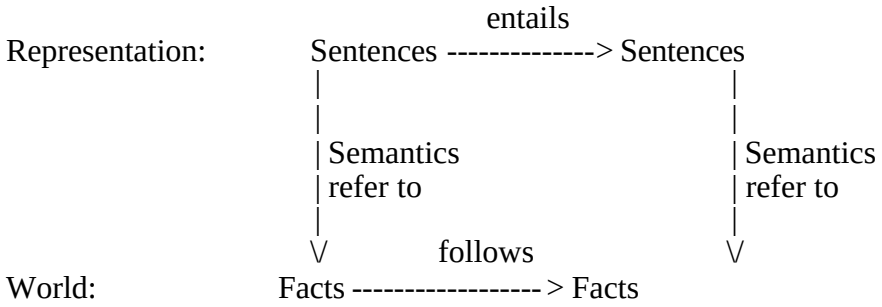
A logic consists of two parts, a language and a method of reasoning. The logical language, in turn, has two aspects, syntax and semantics. Thus, to specify or define a particular logic, one needs to specify three things:

Syntax: The atomic symbols of the logical language, and the rules for constructing well-formed, non-atomic expressions (symbol structures) of the logic. Syntax specifies the symbols in the language and how they can be combined to form sentences. Hence facts about the world are represented as sentences in logic.

Semantics: The meanings of the atomic symbols of the logic, and the rules for determining the meanings of non-atomic expressions of the logic. It specifies what facts in the world a sentence refers to. Hence, also specifies how you assign a truth value to a sentence based on its meaning in the world. A **fact** is a claim about the world, and may be true or false.

Syntactic Inference Method: The rules for determining a subset of logical expressions, called theorems of the logic. It refers to mechanical method for computing (deriving) new (true) sentences from existing sentences.

Facts are claims about the world that are True or False, whereas a **representation** is an expression (sentence) in some language that can be encoded in a computer program and stands for the objects and relations in the world. We need to ensure that the representation is consistent with reality, so that the following figure holds:



There are a number of logical systems with different syntax and semantics. We list below a few of them.

- Propositional logic

  All objects described are fixed or unique

  "John is a student" student(john)

  Here John refers to one unique person.

- First order predicate logic

  Objects described can be unique or variables to stand for a unique object

  "All students are poor"

  ForAll(S) [student(S) -> poor(S)]

  Here S can be replaced by many different unique students.

  This makes programs much more compact:

  eg. ForAll(A,B)[brother(A,B) -> brother (B,A)]

  replaces half the possible statements about brothers

- Temporal

  Represents truth over time.

- Modal

  Represents doubt

- Higher order logics

  Allows variable to represent many relations between objects

- Non-monotonic

  Represents defaults

Propositional is one of the simplest systems of logic.

Propositional Logic

In propositional logic (PL) an user defines a set of propositional symbols, like *P* and *Q*. User defines the semantics of each of these symbols. For example,

  - P means "It is hot"
  - Q means "It is humid"
  - R means "It is raining"
  - o

- A **sentence** (also called a formula or well-formed formula or wff) is defined as:
  1. A symbol
  2. If S is a sentence, then ~S is a sentence, where "~" is the "not" logical operator
  3. If S and T are sentences, then (S v T), (S ^ T), (S => T), and (S <=> T) are sentences, where the four logical connectives correspond to "or," "and," "implies," and "if and only if," respectively
  4. A finite number of applications of (1)-(3)

- Examples of PL sentences:
  - (P ^ Q) => R (here meaning "If it is hot and humid, then it is raining")
  - Q => P (here meaning "If it is humid, then it is hot")
  - Q (here meaning "It is humid.")

- Given the truth values of all of the constituent symbols in a sentence, that sentence can be "evaluated" to determine its truth value (True or False). This is called an **interpretation** of the sentence.

- A **model** is an interpretation (i.e., an assignment of truth values to symbols) of a set of sentences such that each sentence is True. A model is just a formal mathematical structure that "stands in" for the world.

- A **valid** sentence (also called a **tautology**) is a sentence that is True under *all* interpretations. Hence, no matter what the world is actually like or what the semantics is, the sentence is True. For example "It's raining or it's not raining."

- An **inconsistent** sentence (also called **unsatisfiable** or a **contradiction**) is a sentence that is False under *all* interpretations. Hence the world is never like what it describes. For example, "It's raining and it's not raining."

- Sentence P **entails** sentence Q, written P |= Q, means that whenever P is True, so is Q. In other words, all models of P are also models of Q

Entailment ($\models$): Given 2 *sentences* p and q we say p *entails* q, written $p \models q$, if q holds in every model that p holds.

Example: Entailment

Show that: $p \wedge (p \Rightarrow q) \models q$

Proof: For any model M in which $p \wedge (p \Rightarrow q)$ holds then we know that *p* holds in M and $p \Rightarrow q$ holds in M. Since *p* holds in M then since $p \Rightarrow q$ holds in M, *q* must hold in M. Therefore *q* holds in every model that $p \wedge (p \Rightarrow q)$ holds and so $p \wedge (p \Rightarrow q) \models q$.

As we have noted models affect equivalence and so we repeat the definition again and give an example of a proof of equivalence.

Equivalence: Two sentences are *equivalent* if they hold in exactly the same models.

Example: Equivalence

Show that: $p \Rightarrow q \equiv \neg p \vee q$

Proof: We need to provide two proofs as above for
$p \Rightarrow q \models \neg p \vee q$

- For any model M in which $p \Rightarrow q$ holds then we know that either p holds in M and so q holds in M, or p does not hold in M and so $\neg p$ holds in M. Since either q holds in M or $\neg p$ holds in M, then $\neg p \vee q$ holds in M.

and $\neg p \vee q \models p \Rightarrow q$

- For any model M in which $\neg p \vee q$ holds then we know that either $\neg p$ holds in M or q holds in M. If $\neg p$ holds in M then $p \Rightarrow q$ holds in M. Otherwise, if q holds in M then $p \Rightarrow q$ holds in M. Therefore $p \Rightarrow q$ holds in M.

Knowledge based programming relies on concluding new knowledge from existing knowledge. Entailment is a required justification; i.e. if $p_1, \ldots, p_n$ is known then there is justification to conclude q if

$$p_1 \wedge \ldots \wedge p_n \models q$$

In some circumstances we insist on this strong form of justification; i.e. we cannot conclude q unless the entailment holds. Reasoning like this is the equivalent for knowledge based programs of running a piece of conventional software.

Note: Entailment ($\models$) is concerned with *truth* and is determined by considering the truth of the sentences in all models.

Propositional Logic Inference

Let KB = { S1, S2,..., SM } be the set of all sentences in our Knowledge Base, where each Si is a sentence in Propositional Logic. Let { X1, X2, ..., XN } be the set of all the symbols (i.e., variables) that are contained in all of the M sentences in KB. Say we want to know if a goal (aka query, conclusion, or theorem) sentence G follows from KB.

Model Checking

Since the computer doesn't know the interpretation of these sentences in the world, we don't know whether the constituent symbols represent facts in the world that are True or False. So, instead, consider *all* possible combinations of truth values for all the symbols, hence enumerating all logically distinct cases:

X1	X2	...	XN	S1	S2	...	SM	S1 ^ S2 ^...^ SM	G	(S1 ^...^ SM) => G
F	F	...	F							
F	F	...	T							
...										
T	T	...	T							

- There are 2^N rows in the table.
- Each row corresponds to an equivalence class of worlds that, under a given interpretation, have the truth values for the N symbols assigned in that row.
- The **models** of KB are the rows where the third-to-last column is *true*, i.e., where all of the sentences in KB are *true*.
- A sentence R is **valid** if and only if it is true under all possible interpretations, i.e., if the entire column associated with R contains all *true* values.
- Since we don't know the semantics and therefore whether each symbol is True or False, to determine if a sentence G is **entailed** by KB, we must determine if **all** models of KB are also models of G. That is, whenever KB is true, G is true too. In other words, whenever the third-to-last column has a T, the same row in the second-to-last column also has a T. But this is logically equivalent to saying that the sentence (KB => G) is valid (by definition of the "implies" connective). In other words, if the last column of the table above contains only *True*, then **KB entails G**; or conclusion G logically follows from the premises in KB, no matter what the interpretations (i.e., semantics) associated with all of the sentences!
- The truth table method of inference is **complete** for PL (Propositional Logic) because we can always enumerate all 2^*n* rows for the *n* propositional symbols that occur. But this is exponential in *n*. In general, it has been shown that the problem of checking if a set of sentences in PL is satisfiable is NP-complete. (The truth table method of inference is *not* complete for FOL (First-Order Logic).)

Example

Using the "weather" sentences from above, let KB = (((P ^ Q) => R) ^ (Q => P) ^ Q)

corresponding to the three facts we know about the weather: (1) "If it is hot and humid, then it is raining," (2) "If it is humid, then it is hot," and (3) "It is humid." Now let's ask the query "Is it raining?" That is, is the query sentence R entailed by KB? Using the truth- table approach to answering this query we have:

P	Q	R	(P ^ Q) => R	Q => P	Q	KB	R	KB => R
T	T	T	T	T	T	T	T	T
T	T	F	F	T	T	F	F	T
T	F	T	T	T	F	F	T	T
T	F	F	T	T	F	F	F	T
F	T	T	T	F	T	F	T	T
F	T	F	T	F	T	F	F	T
F	F	T	T	T	F	F	T	T
F	F	F	T	T	F	F	F	T

Hence, in this problem there is only one model of KB, when P, Q, and R are all True. And in this case R is also True, so R is entailed by KB. Also, you can see that the last column is all True values, so the sentence KB => R is valid.

Instead of an exponential length proof by truth table construction, is there a faster way to implement the inference process? Yes, using a **proof procedure** or **inference procedure** that uses **sound rules of inference** to deduce (i.e., derive) new sentences that are true in all cases where the premises are true. For example, consider the following:

P	Q	P	P => Q	P ^ (P => Q)	Q	(P ^ (P => Q)) => Q
F	F	F	T	F	F	T
F	T	F	T	F	T	T
T	F	T	F	F	F	T
T	T	T	T	T	T	T

Since whenever P and P => Q are both true (last row only), Q is true too, Q is said to be **derived** from these two premise sentences. We write this as KB |- Q. This local pattern referencing only two of the M sentences in KB is called the **Modus Ponens** inference rule. The truth table shows that this inference rule is **sound**. It specifies how to make one kind of step in deriving a conclusion sentence from a KB.

Therefore, given the sentences in KB, construct a **proof** that a given conclusion sentence can be derived from KB by applying a sequence of sound inferences using either sentences in KB or sentences derived earlier in the proof, until the conclusion sentence is derived. This method is called the **Natural Deduction** procedure. (Note: This step-by- step, local proof process also relies on the **monotonicity** property of PL and FOL. That is, adding a new sentence to KB does not affect what can be entailed from the original KB and does not invalidate old sentences.)

Rules of Inference

Here are some examples of sound rules of inference. Each can be shown to be sound once and for all using a truth table. The left column contains the premise sentence(s), and the right column contains the derived sentence. We write each of these derivations as A |- B , where A is the premise and B is the derived sentence.

Name	**Premise(s)**	**Derived Sentence**
Modus Ponens	A, A => B	B
And Introduction	A, B	A ^ B
And Elimination	A ^ B	A
Double Negation	~~A	A
Unit Resolution	A v B, ~B	A
Resolution	A v B, ~B v C	A v C

In addition to the above rules of inference one also requires a set of equivalences of propositional logic like “A /\ B” is equivalent to “B /\ A”. A number of such equivalences were presented in the discussion on propositional logic.

Using Inference Rules to Prove a Query/Goal/Theorem

A proof is a sequence of sentences, where each sentence is either a premise or a sentence derived from earlier sentences in the proof by one of the rules of inference. The last sentence is the query (also called goal or theorem) that we want to prove.

Example for the "weather problem" given above.

1.	Q	Premise
2.	Q => P	Premise
3.	P	Modus Ponens(1,2)
4.	(P ^ Q) => R	Premise
5.	P ^ Q	And Introduction(1,3)
6.	R	Modus Ponens(4,5)

Inference vs Entailmant

There is a subtle difference between entailment and inference.

Inference ($\vdash$): Given 2 *sentences* p and q we say q *is inferred from* p, written $p \vdash q$, if there is a sequence of rules of inference that apply to p and allow q to be added.

Notice that inference is not directly related to truth; i.e. we can infer a sentence provided we have rules of inference that produce the sentence from the original sentences.

However, if rules of inference are to be useful we wish them to be related to entailment. Ideally we would like:

$p \vdash q$ iff $p \models q$

but this equivalence may fail in two ways:

- $p \vdash q$ but $p \not\models q$

  We have inferred ***q*** by applying rules of inference to ***p***, but there is some model in which p holds but q does not hold. In this case the rules of inference have inferred ``too much".

- $p \models q$ but $p \not\vdash q$

  q is a sentence which holds in all models in which ***p*** holds, but we cannot find rules of inference that will infer ***q*** from ***p***. In this case the rules of inference are insufficient to infer the things we want to be able to infer.

Soundness and Completeness

These notions are so important that there are 2 properties of logics associated with them.

Soundness: An *inference procedure* $\vdash$ is *sound* if whenever $p \vdash q$ then it is also the case that $p \models q$.

``A sound inference procedure infers things that are valid consequences"

Completeness: An *inference procedure* $\vdash$ is *complete* if whenever $p \models q$ then it is also the case that $p \vdash q$.

``A complete inference procedure is able to infer anything that is that is a valid consequence"

The ``best" inference procedures are both sound and complete, but gaining completeness is often computationally expensive. Notice that even if inference is not complete it is desirable that it is sound.

Propositional Logic and Predicate Logic each with Modus Ponens as their inference produce are sound but not complete. We shall see that we need further (sound) rules of inference to achieve completeness. In fact we shall see that we shall even restrict the language in order to achieve an effective inference procedure that is sound and complete for a subset of First Order Predicate Logic.

The notion of soundness and completeness is more generally applied than in logic. Whenever we create a knowledge based program we use the syntax of the knowledge representation language, we assign a semantics in some way and the reasoning mechanism defines the inference procedures. The semantics will define what entailment means in this representation and we will be interested in how well the reasoning mechanism achieves entailment.

Decidability

Determining whether $p \models q$ is computationally hard. If q is a consequence then if $\vdash$ is complete then we know that $p \vdash q$ and we can apply the rules of inference exhaustively knowing that we will eventually find the sequence of rules of inference. It may take a long time but it is finite.

However if q is not a consequence (remember the task is *whether or not* $p \models q$) then we can happily apply rules of inference generating more and more irrelevant consequences. So the procedure is guaranteed to eventually stop if q is derivable, but may not terminate otherwise.

Decidability: A *problem* is *decidable* if there is a procedure that is guaranteed to terminate having determined whether the answer is "yes" or "no".

Semi-Decidability: A *problem* is only *semi-decidable* if there is a procedure that is guaranteed to terminate in one of these cases but not both.

Entailment in Propositional Logic is decidable since truth tables can be applied in a finite number of steps to determine whether or not $p \models q$.

Entailment in Predicate Logic is only semi-decidable; it is only guaranteed to terminate when q is a consequence. One result of this semi-decidability is that many problems are not decidable; if they rely on failing to prove some sentence. Planning is typically not decidable. A common reaction to a non-decidable problem is to *assume* the answer after some reasoning time threshold has been reached. Another reaction to the semi- decidability of Predicate Logic is to restrict attention to subsets of the logic; however even if its entailment is decidable the procedure may be computationally expensive.

First Order Logic

Syntax

Let us first introduce the symbols, or alphabet, being used. Beware that there are all sorts of slightly different ways to define FOL.

Alphabet

- Logical Symbols: These are symbols that have a standard meaning, like: AND, OR, NOT, ALL, EXISTS, IMPLIES, IFF, FALSE, =.
- Non-Logical Symbols: divided in:
    - Constants:
        - Predicates: 1-ary, 2-ary, .., n-ary. These are usually just identifiers.
        - Functions: 0-ary, 1-ary, 2-ary, .., n-ary. These are usually just identifiers. 0-ary functions are also called **individual constants**.

      Where predicates return true or false, functions can return any value.

    - Variables: Usually an identifier.

  One needs to be able to distinguish the identifiers used for predicates, functions, and variables by using some appropriate convention, for example, capitals for function and predicate symbols and lower cases for variables.

Terms

A Term is either an individual constant (a 0-ary function), or a variable, or an n-ary function applied to n terms: F(t1 t2 ..tn)
[We will use both the notation F(t1 t2 ..tn) and the notation (F t1 t2 .. tn)]

Atomic Formulae

An Atomic Formula is either FALSE or an n-ary predicate applied to n terms: P(t1 t2 .. tn). In the case that "=" is a logical symbol in the language, (t1 = t2), where t1 and t2 are terms, is an atomic formula.

Literals

A Literal is either an atomic formula (a **Positive Literal**), or the negation of an atomic formula (a **Negative Literal**). A **Ground Literal** is a variable-free literal.

Clauses

A Clause is a disjunction of literals. A **Ground Clause** is a variable-free clause. A **Horn Clause** is a clause with at most one positive literal. A **Definite Clause** is a Horn Clause with exactly one positive Literal.

Notice that implications are equivalent to Horn or Definite clauses:
(A IMPLIES B) is equivalent to ((NOT A) OR B)
(A AND B IMPLIES FALSE) is equivalent to ((NOT A) OR (NOT B)).

Formulae

A Formula is either:

- an atomic formula, or
- a **Negation**, i.e. the NOT of a formula, or
- a **Conjunctive Formula**, i.e. the AND of formulae, or
- a **Disjunctive Formula**, i.e. the OR of formulae, or
- an **Implication**, that is a formula of the form (formula1 IMPLIES formula2), or
- an **Equivalence**, that is a formula of the form (formula1 IFF formula2), or
- a **Universally Quantified Formula**, that is a formula of the form (ALL variable formula). We say that occurrences of variable are **bound** in formula [we should be more precise]. Or
- a **Existentially Quantified Formula**, that is a formula of the form (EXISTS variable formula). We say that occurrences of variable are **bound** in formula [we should be more precise].

An occurrence of a variable in a formula that is not bound, is said to be **free**. A formula where all occurrences of variables are bound is called a **closed formula**, one where all variables are free is called an **open formula**.

A formula that is the disjunction of clauses is said to be in **Clausal Form**. We shall see that there is a sense in which every formula is equivalent to a clausal form.

Often it is convenient to refer to terms and formulae with a single name. **Form** or **Expression** is used to this end.

Substitutions

- Given a term s, the result [**substitution instance**] of **substituting a term t in s for a variable x**, s[t/x], is:
  - t, if s is the variable x
  - y, if s is the variable y different from x
  - F(s1[t/x] s2[t/x] .. sn[t/x]), if s is F(s1 s2 .. sn).

- Given a formula A, the result (**substitution instance**) of **substituting a term t in A for a variable x**, A[t/x], is:
  - FALSE, if A is FALSE,
  - P(t1[t/x] t2[t/x] .. tn[t/x]), if A is P(t1 t2 .. tn),
  - (B[t/x] AND C[t/x]) if A is (B AND C), and similarly for the other connectives,
  - (ALL x B) if A is (ALL x B), (similarly for EXISTS),
  - (ALL y B[t/x]), if A is (ALL y B) and y is different from x (similarly for EXISTS).

The substitution [t/x] can be seen as a map from terms to terms and from formulae to formulae. We can define similarly [t1/x1 t2/x2 .. tn/xn], where t1 t2 .. tn are terms and x1 x2 .. xn are variables, as a map, the **[simultaneous] substitution of x1 by t1, x2 by t2, .., of xn by tn**. [If all the terms t1 .. tn are variables, the substitution is called an **alphabetic variant**, and if they are ground terms, it is called a **ground substitution**.] Note that a simultaneous substitution is not the same as a sequential substitution.

Unification

- Given two substitutions S = [t1/x1 .. tn/xn] and V = [u1/y1 .. um/ym], the **composition** of S and V, S . V, is the substitution obtained by:
  - Applying V to t1 .. tn [the operation on substitutions with just this property is called **concatenation**], and
  - adding any pair uj/yj such that yj is not in {x1 .. xn}.

  For example: [G(x y)/z].[A/x B/y C/w D/z] is [G(A B)/z A/x B/y C/w].

  Composition is an operation that is associative and non commutative

- A set of forms f1 .. fn is **unifiable** iff there is a substitution S such that f1.S = f2.S = .. = fn.S. We then say that S is a **unifier** of the set.
  For example {P(x F(y) B) P(x F(B) B)} is unified by [A/x B/y] and also unified by [B/y].

- A **Most General Unifier** (MGU) of a set of forms f1 .. fn is a substitution S that unifies this set and such that for any other substitution T that unifies the set there is a substitution V such that S.V = T. The result of applying the MGU to the forms is called a **Most General Instance** (MGI). Here are some examples:

FORMULAE	MGU	MGI
(P x), (P A)	[A/x]	(P A)
(P (F x) y (G y)), (F x) z (G x))	[x/y x/z]	(P (F x) x (G x)) (P

(F x (G y)), [(G u)/x y/z] (F (G u) (G y))

(F (G u) (G z))	
(F x (G y)), (F (G u) (H z))	Not Unifiable
(F x (G x) x), (F (G u) (G (G z)) z)	Not Unifiable

This last example is interesting: we first find that (G u) should replace x, then that (G z) should replace x; finally that x and z are equivalent. So we need x->(G z) and x->z to be both true. This would be possible only if z and (G z) were equivalent. That cannot happen for a finite term. To recognize cases such as this that do not allow unification [we cannot replace z by (G z) since z occurs in (G z)], we need what is called an Occur Test . Most Prolog implementation use Unification extensively but do not do the occur test for efficiency reasons.

The determination of Most General Unifier is done by the **Unification Algorithm**. Here is the pseudo code for it:

FUNCTION Unify WITH PARAMETERS form1, form2, and assign RETURNS MGU, where form1 and form2 are the forms that we want to unify, and assign is initially nil.

1. Use the Find-Difference function described below to determine the first elements where form1 and form2 differ and one of the elements is a variable. Call difference-set the value returned by Find- Difference. This value will be either the atom Fail, if the two forms cannot be unified; or null, if the two forms are identical; or a pair of the form (Variable Expression).

2. If Find-Difference returned the atom Fail, Unify also returns Fail and we cannot unify the two forms.

3. If Find-Difference returned nil, then Unify will return assign as MGU.

4. Otherwise, we replace each occurrence of Variable by Expression in form1 and form2; we compose the given assignment assign with the assignment that maps Variable into Expression, and we repeat the process for the new form1, form2, and assign.

FUNCTION Find-Difference WITH PARAMETERS form1 and form2 RETURNS pair, where form1 and form2 are e-expressions.

1. If form1 and form2 are the same variable, return nil.

2. Otherwise, if either form1 or form2 is a variable, and it does not appear anywhere in the other form, then return the pair (Variable Other-Form), otherwise return Fail.

3. Otherwise, if either form1 or form2 is an atom then if they are the same atom then return nil otherwise return Fail.

4. Otherwise both form1 and form2 are lists.
   Apply the Find-Difference function to corresponding elements of the two lists until either a call returns a non-null value or the two lists are simultaneously exhausted, or some elements are left over in one list.

   In the first case, that non-null value is returned; in the second,
   nil is returned; in the third, Fail is returned

Semantics

Before we can continue in the "syntactic" domain with concepts like Inference Rules and Proofs, we need to clarify the Semantics, or meaning, of First Order Logic.

An **L-Structure** or **Conceptualization** for a language L is a structure M= (U,I), where:

- U is a non-empty set, called the **Domain**, or **Carrier**, or **Universe of Discourse** of M, and
- I is an **Interpretation** that associates to each n-ary function symbol F of L a map

  I(F): UxU..xU -> U

  and to each n-ary predicate symbol P of L a subset of UxU..xU.

The set of functions (predicates) so introduced form the **Functional Basis (Relational Basis)** of the conceptualization.

Given a language L and a conceptualization (U,I), an **Assignment** is a map from the variables of L to U. An **X-Variant** of an assignment s is an assignment that is identical to s everywhere except at x where it differs.

Given a conceptualization M=(U,I) and an assignment s it is easy to extend s to map each term t of L to an individual s(t) in U by using induction on the structure of the term.

Then

- **M satisfies a formula A under s** iff
  - A is atomic, say P(t1 .. tn), and (s(t1) ..s(tn)) is in I(P).
  - A is (NOT B) and M does not satisfy B under s.
  - A is (B OR C) and M satisfies B under s, or M satisfies C under s. [Similarly for all other connectives.]
  - A is (ALL x B) and M satisfies B under all x-variants of s.
  - A is (EXISTS x B) and M satisfies B under some x-variants of s.
- **Formula A is satisfiable in M** iff there is an assignment s such that M satisfies A under s.
- **Formula A is satisfiable** iff there is an L-structure M such that A is satisfiable in M.

- **Formula A is valid or logically true in M** iff M satisfies A under any s. We then say that M is a **model** of A.
- **Formula A is Valid or Logically True** iff for any L-structure M and any assignment s, M satisfies A under s.

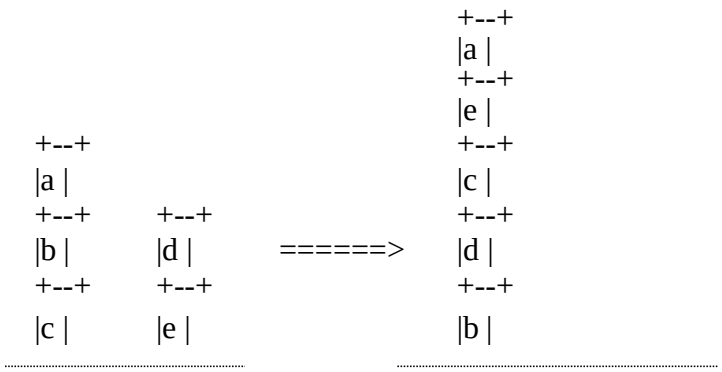
Some of these definitions can be made relative to a set of formulae GAMMA:

- **Formula A is a Logical Consequence of GAMMA in M** iff M satisfies A under any s that also satisfies all the formulae in GAMMA.
- **Formula A is a Logical Consequence of GAMMA** iff for any L-structure M, A is a logical consequence of GAMMA in M. At times instead of "A is a logical consequence of GAMMA" we say "GAMMA entails A".

We say that formulae A and B are (logically) **equivalent** iff A is a logical consequence of {B} and B is a logical consequence of {A}.

EXAMPLE 1: A Block World

Here we look at a problem and see how to represent it in a language. We consider a simple world of blocks as described by the following figures:



We see two possible states of the world. On the left is the current state, on the right a desired new state. A robot is available to do the transformation. To describe these worlds we can use a structure with domain U = {a b c d e}, and with predicates {ON, ABOVE, CLEAR, TABLE} with the following meaning:

- ON: (ON x y) iff x is immediately above y.
  The interpretation of ON in the left world is {(a b) (b c) (d e)}, and in the right world is {(a e) (e c) (c d) (d b)}.
- ABOVE: (ABOVE x y) iff x is above y.
  The interpretation of ABOVE [in the left world] is {(a b) (b c) (a c) (d e)} and in the right world is {(a e) (a c) (a d) (a b) (e c) (e d) (e b) (c d) (c b) (d b)}
- CLEAR: (CLEAR x) iff x does not have anything above it.
  The interpretation of CLEAR [in the left world] is {a d} and in the right world is

{a}

- TABLE: (TABLE x) iff x rests directly on the table.
  The interpretation of TABLE [in the left world] is {c e} and in the right world id {b}.

Examples of formulae true in the block world [both in the left and in the right state] are [these formulae are known as **Non-Logical Axioms**]:

- (ON x y) IMPLIES (ABOVE x y)
- ((ON x y) AND (ABOVE y z)) IMPLIES (ABOVE x z)
- (ABOVE x y) IMPLIES (NOT (ABOVE y x))
- (CLEAR x) IFF (NOT (EXISTS y (ON y x)))
- (TABLE x) IFF (NOT (EXISTS y (ON x y)))

Note that there are things that we cannot say about the block world with the current functional and predicate basis unless we use equality. Namely, we cannot say as we would like that a block can be ON at most one other block. For that we need to say that if x is ON y and x is ON z then y is z. That is, we need to use a logic with equality.

Not all formulae that are true on the left world are true on the right world and viceversa. For example, a formula true in the left world but not in the right world is (ON a b). Assertions about the left and right world can be in contradiction. For example (ABOVE b c) is true on left, (ABOVE c b) is true on right and together they contradict the non-logical axioms. This means that the theory that we have developed for talking about the block world can talk of only one world at a time. To talk about two worlds simultaneously we would need something like the Situation Calculus that we will study later.

First Order Logic - II
INFERENCE IN FOL

Herbrand Universe

It is a good exercise to determine for given formulae if they are satisfied/valid on specific L-structures, and to determine, if they exist, models for them. A good starting point in this task, and useful for a number of other reasons, is the **Herbrand Universe** for this set of formulae. Say that {F01 .. F0n} are the individual constants in the formulae [if there are no such constants, then introduce one, say, F0]. Say that {F1 .. Fm} are all the non 0-ary function symbols occurring in the formulae. Then the set of (constant) terms obtained starting from the individual constants using the non 0-ary functions, is called the Herbrand Universe for these formulae.

For example, given the formula (P x A) OR (Q y), its Herbrand Universe is just {A}. Given the formulae (P x (F y)) OR (Q A), its Herbrand Universe is {A (F A) (F (F A)) (F (F (F A))) ...}.

Reduction to Clausal Form

In the following we give an algorithm for deriving from a formula an equivalent clausal form through a series of truth preserving transformations.

We can state an (unproven by us) theorem:

Theorem: Every formula is equivalent to a clausal form

We can thus, when we want, restrict our attention only to such forms.

Deduction

An **Inference Rule** is a rule for obtaining a new formula [the **consequence**] from a set of given formulae [the **premises**].

A most famous inference rule is **Modus Ponens**:

{A, NOT A OR B} / B

For example:

{Sam is tall,
if Sam is tall then Sam is unhappy} / Sam is unhappy

When we introduce inference rules we want them to be **Sound**, that is, we want the consequence of the rule to be a logical consequence of the premises of the rule. Modus Ponens is sound. But the following rule, called **Abduction** , is not:

{B, NOT A OR B} / A

is not. For example:

John is wet

If it is raining then John is wet It is

raining

gives us a conclusion that is usually, but not always true [John takes a shower even when it is not raining].

A **Logic** or **Deductive System** is a language, plus a set of inference rules, plus a set of logical axioms [formulae that are valid].

A **Deduction or Proof or Derivation** in a deductive system D, given a set of formulae GAMMA, is a a sequence of formulae B1 B2 .. Bn such that:

- for all i from 1 to n, Bi is either a logical axiom of D, or an element of GAMMA, or is obtained from a subset of {B1 B2 .. Bi-1} by using an inference rule of D.

In this case we say that Bn is **Derived** from GAMMA in D, and in the case that GAMMA is empty, we say that Bn is a **Theorem** of D.

Soundness, Completeness, Consistency, Satisfiability

A Logic D is **Sound** iff for all sets of formulae GAMMA and any formula A:

- if A is derived from GAMMA in D, then A is a logical consequence of GAMMA

A Logic D is **Complete** iff for all sets of formulae GAMMA and any formula A:

- If A is a logical consequence of GAMMA, then A can be derived from GAMMA in D.

A Logic D is **Refutation Complete** iff for all sets of formulae GAMMA and any formula A:

- If A is a logical consequence of GAMMA, then the union of GAMMA and (NON A) is inconsistent

Note that if a Logic is Refutation Complete then we can enumerate all the logical consequences of GAMMA and, for any formula A, we can reduce the question if A is or not a logical consequence of GAMMA to the question: the union of GAMMA and NOT A is or not consistent.

We will work with logics that are both Sound and Complete, or at least Sound and Refutation Complete.

A **Theory** T consists of a logic and of a set of Non-logical axioms. For convenience, we may refer, when not ambiguous, to the logic of T, or the non-logical axioms of T, just as T.

The common situation is that we have in mind a well defined "world" or set of worlds. For example we may know about the natural numbers and the arithmetic operations and relations. Or we may think of the block world. We choose a language to talk about these worlds. We introduce function and predicate symbols as it is appropriate. We then introduce formulae, called **Non-Logical Axioms**, to characterize the things that are true in the worlds of interest to us. We choose a logic, hopefully sound and (refutation) complete, to derive new facts about the worlds from the non-logical axioms.

A **Theorem** in a theory T is a formula A that can be derived in the logic of T from the non-logical axioms of T.

A Theory T is **Consistent** iff there is no formula A such that both A and NOT A are theorems of T; it is **Inconsistent** otherwise.
If a theory T is inconsistent, then, for essentially any logic, any formula is a theorem of T. [Since T is inconsistent, there is a formula A such that both A and NOT A are theorems of T. It is hard to imagine a logic where from A and (NOT A) we cannot infer FALSE, and from FALSE we cannot infer any formula. We will say that a logic that is at least this powerful is **Adeguate**.]

A Theory T is **Unsatisfiable** if there is no structure where all the non-logical axioms of T are valid. Otherwise it is **Satisfiable**.

Given a Theory T, a formula A is a **Logical Consequence of T** if it is a logical consequence of the non logical axioms of T.

Theorem: If the logic we are using is sound then:

1. If a theory T is satisfiable then T is consistent
2. If the logic used is also adequate then if T is consistent then T is satisfiable
3. If a theory T is satisfiable and by adding to T the non-logical axiom (NOT A) we get a theory that is not satisfiable Then A is a logical consequence of T.
4. If a theory T is satisfiable and by adding the formula (NOT A) to T we get a theory that is inconsistent, then A is a logical consequence of T.

Resolution

We have introduced the inference rule Modus Ponens. Now we introduce another inference rule that is particularly significant, Resolution.

Since it is not trivial to understand, we proceed in two steps. First we introduce Resolution in the Propositional Calculus, that is, in a language with only truth valued variables. Then we generalize to First Order Logic.

Resolution in the Propositional Calculus

In its simplest form Resolution is the inference rule:

{A OR C, B OR (NOT C)}

A OR B

More in general the **Resolution Inference Rule** is:

- Given as premises the clauses C1 and C2, where C1 contains the literal L and C2 contains the literal (NOT L), infer the clause C, called the **Resolvent** of C1 and C2, where C is the union of (C1 - {L}) and (C2 -{(NOT L)})

In symbols:

{C1, C2}

(C1 - {L}) UNION (C2 - {(NOT L)})

Example:

The following set of clauses is inconsistent:

1. (P OR (NOT Q))
2. ((NOT P) OR (NOT S))
3. (S OR (NOT Q))
4. Q

In fact:

5. ((NOT Q) OR (NOT S)) from 1. and 2.
6. (NOT Q) from 3. and 5.
7. FALSE from 4. and 6.

Notice that 7. is really the empty clause [why?].

Theorem: The Propositional Calculus with the Resolution Inference Rule is sound and Refutation Complete.

NOTE: This theorem requires that clauses be represented as sets, that is, that each element of the clause appear exactly once in the clause. This requires some form of membership test when elements are added to a clause.

C1 = {P P}

C2 = {(NOT P) (NOT P)}
C = {P (NOT P)}

From now on by resolution we just get again C1, or C2, or C.

Resolution in First Order Logic

Given clauses C1 and C2, a clause C is a **RESOLVENT of C1 and C2**, if

1. There is a subset C1' = {A1, .., Am} of C1 of literals of the same sign, say positive, and a subset C2' = {B1, .., Bn} of C2 of literals of the opposite sign, say negative,
2. There are substitutions s1 and s2 that replace variables in C1' and C2' so as to have new variables,
3. C2" is obtained from C2 removing the negative signs from B1 .. Bn
4. There is an Most General Unifier s for the union of C1'.s1 and C2".s2

and C is
((C1 - C1').s1 UNION (C2 - C2').s2).s

In symbols this **Resolution** inference rule becomes:

{C1, C2}

C

If C1' and C2' are singletons (i.e. contain just one literal), the rule is called **Binary Resolution**.

Example:

C1 = {(P z (F z)) (P z A)}
C2 = {(NOT (P z A)) (NOT (P z x)) (NOT (P x z))
C1' = {(P z A)}
C2' = {(NOT (P z A)) (NOT (P z x))}
C2" = {(P z A) (P z x)}
s1 = [z1/z]
s2 = [z2/z]
C1'.s1 UNION C2'.s2 = {(P z1 A) (P z2 A) (P z2 x)}
s = [z1/z2 A/x]
C = {(NOT (P A z1)) (P z1 (F z1))}

Notice that this application of Resolution has eliminated more than one literal from C2, i.e. it is not a binary resolution.

Theorem: First Order Logic, with the Resolution Inference Rule, is sound and refutation complete.

We will not develop the proof of this theorem. We will instead look at some of its steps, which will give us a wonderful opportunity to revisit Herbrand. But before that let's observe that in a sense, if we replace in this theorem "Resolution" by "Binary Resolution", then the theorem does not hold and Binary Resolution is not Refutation Complete. This is the case when in the implementation we do not use sets but instead use bags.This can be shown using the same example as in the case of propositional logic.

Given a clause C, a subset D of C, and a substitution s that unifies D, we define C.s to be a **Factor** of C.
The **Factoring Inference Rule** is the rule with premise C and as consequence C.s.

Theorem: For any set of clauses S and clause C, if C is derivable from S using Resolution, then C is derivable from S using Binary Resolution and Factoring.

When doing proofs it is efficient to have as few clauses as possible. The following definition and rule are helpful in eliminating redundant clauses:

A clause C1 **Subsumes** a clause C2 iff there is a substitution s such that C1.s is a subset of C2.
Subsumption Elimination Rule: If C1 subsumes C2 then eliminate C2.

Herbrand Revisited

We have presented the concept of Herbrand Universe H_S for a set of clauses S. Here we meet the concept of **Herbrand Base**, H(S), for a set of clauses S.
H(S) is obtained from S by considering the ground instances of the clauses of S under all the substitutions that map all the variables of S into elements of the Herbrand universe of S. Clearly, if in S occurs some variable and the Herbrand universe of S is infinite then H(S) is infinite.
[NOTE: Viceversa, if S has no variables, or S has variables and possibly individual constants, but no other function symbol, then H(S) is finite. If H(S) is finite then we can, as we will see, decide if S is or not satisfiable.]
[NOTE: it is easy to determine if a finite subset of H(S) is satisfiable: since it consists of ground clauses, the truth table method works now as in propositional cases.]

The importance of the concepts of Herbrand Universe and of Herbrand Base is due to the following theorems:

Herbrand Theorem: If a set S of clauses is unsatisfiable then there is a finite subset of H(S) that is also unsatisfiable.

Because of the theorem, when H(S) is finite we will be able to decide is S is or not satisfiable. Herbrand theorem immediately suggests a general refutation complete proof procedure:

given a set of clauses S, enumerate subsets of H(S) until you find one that is

unsatisfiable.

But, as we shall soon see, we can come up with a better refutation complete proof procedure.

Refutation Completeness of the Resolution Proof Procedure

Given a set of clauses S, the **Resolution Closure** of S, R(S), is the smallest set of clauses that contains S and is closed under Resolution. In other words, it is the set of clauses obtained from S by applying repeatedly resolution.

Ground Resolution Theorem: If S is an unsatisfiable set of ground clauses, then R(S) contains the clause FALSE.

In other words, there is a resolution deduction of FALSE from S.

Lifting Lemma: Given clauses C1 and C2 that have no variables in common, and ground instances C1' and C2', respectively, of C1 and C2, if C' is a resolvent of C1' and C2', then there is a clause C which is a resolvent of C1 and C2 which has C' as a ground instance

With this we have our result, that the Resolution Proof procedure is Refutation Complete. Note the crucial role played by the Herbrand Universe and Basis. Unsatisfiability of S is reduced to unsatisfiability for a finite subset $H_S(S)$ of H(S), which in turn is reduced to the problem of finding a resolution derivation for FALSE in $H_S(S)$, derivation which can be "lifted" to a resolution proof of FALSE from S.

Dealing with Equality

Up to now we have not dealt with equality, that is, the ability to recognize terms as being equivalent (i.e. always denoting the same individual) on the basis of some equational information. For example, given the information that

S(x) = x+1

then we can unify:

F(S(x) y) and F(x+1, 3).

There are two basic approaches to dealing with this problem.

- The first is to add inference rules to help us replace terms by equal terms. One such rule is the **Demodulation Rule**: Given terms t1, t2, and t3 where t1 and t2 are unifiable with MGU s, and t2 occurs in a formula A, then

{t1 = t3, A(... t2 ...)}

.....A(... t3.s

...)

Another more complex, and useful, rule is **Paramodulation**.

- The second approach is not to add inference rules and instead to add non-logical axioms that characterize equality and its effect for each non logical symbol. We first establish the reflexive, symmetric and transitive properties of "=":

  x=x
  x=y IMPLIES y=x
  x=y AND y=z IMPLIES x=z

  Then for each unary function symbol F we add the equality axiom

  x=y IMPLIES F(x)=F(y)

  Then for each binary function symbol F we add the equality axiom

  x=z AND y=w IMPLIES F(x y)=F(z w)

  And similarly for all other function symbols.

  The treatment of predicate symbols is similar. For example, for the binary predicate symbol P we add

  x=z AND y=w IMPLIES (P(x y) IFF P(z w))

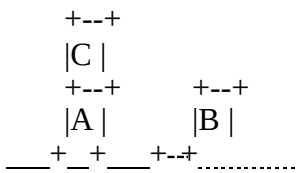
Whatever approach is chosen, equality substantially complicates proofs.

Answering True/False Questions

If we want to show that a clause C is derivable from a set of clauses S={C1 C2 .. Cn}, we add to S the clauses obtained by negating C, and apply resolution to the resulting set S' of clauses until we get the clause FALSE.

Example:

We are back in the Block World with the following state



which gives us the following State Clauses:

- ON(C A)
- ONTABLE(A)
- ONTABLE(B)
- CLEAR(C)
- CLEAR(B)

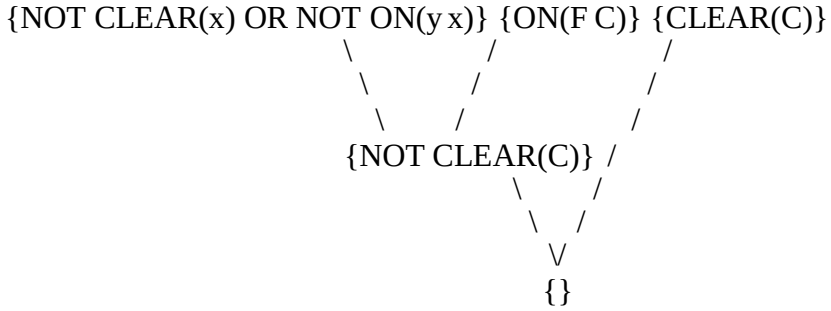
In addition we consider the non-logical axiom:

- (ALL x (CLEAR(x) IMPLIES (NOT (EXISTS y ON(y x)))))

which in clause form becomes

- NOT CLEAR(x) OR NOT ON(y x)

If we now ask whether (NOT (EXISTS y (ON(y C)))), we add to the clauses considered above the clause ON(F C) and apply resolution:



Example:

We are given the following predicates:

- S(x) : x is Satisfied
- H(x) : x is Healthy
- R(x) : x is Rich
- P(x) : x is Philosophical

The premises are the non-logical axioms:

- S(x) IMPLIES (H(x) AND R(x))
- EXISTS x (S(x) and P(x))

The conclusion is

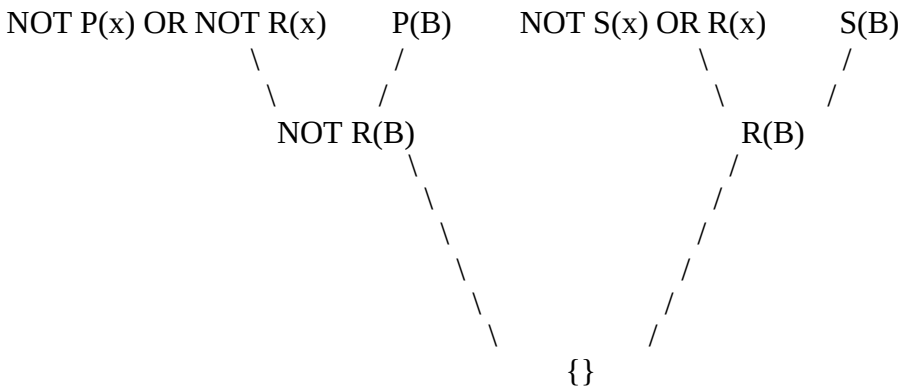
- EXISTS x (P(x) AND R(x))

The corresponding clauses are:

1. NOT S(x) OR H(x)
2. NOT S(x) OR R(x)
3. S(B)
4. P(B)
5. NOT P(x) OR NOT R(x)

where B is a Skolem constant.

The proof is then:



Answering Fill-in-Blanks Questions

We now determine how we can identify individual(s) that satisfy specific formulae.

EXAMPLE:

NON-LOGICAL SYMBOLS:

- SW(x y): x is staying with y
- A(x y): x is at place y
- R(x y): x can be reached at phone number y
- PH(x): the phone number for place x
- Sally, Morton, UnionBldg: Individuals

NON-LOGICAL AXIOMS:

1. SW(Sally Morton)
2. A(Morton UnionBlidg)
3. SW(x y) AND A(y z) IMPLIES A(x z), which is equivalent to the clause
    1. NOT SW(x y) OR NOT A(y z) OR A(x z)
4. A(x y) IMPLIES R(x PH(y)), which is equivalent to the clause
    1. NOT A(u v) OR R(u PH(v))

GOAL: Determine where to call Sally

- NOT EXISTS x R(Sally x), which is equivalent to the clause
    1. NOT R(Sally w).

To this clause we add as a disjunct the literal, **Answer Literal**, Ans(w) to obtain the clause :

5. Ans(w) OR NOT R(Sally w).

PROOF

6. Ans(v) OR NOT A(Sally v). from 4. and 5.
7. Ans(v) OR NOT SW(Sally y) OR NOT A(y v), from 6. and 3.
8. Ans(v) OR NOT A(Morton v), from 7. and 1.
9. Ans(UnionBldg), from 8. and 2.

The proof procedure terminates when we get a clause that is an instance of the Answer Literal. 9. and gives us the place where we can call Sally.

General Method

If A is the Fill-In-Blanks question that we need to answer and x1 .. xn are the free variables occurring in A, then we add to the Non-Logical axioms and Facts GAMMA the clause

NOT A OR ANS(x1 .. xn)

We terminate the proof when we get a clause of the form

ANS(t1 .. tn)

t1 .. tn are terms that denote individuals that simultaneously satisfy the query for, respectively x1 .. xn.

We can obtain all the individuals that satisfy the original query by continuing the proof looking for alternative instantiations for the variables x1 .. xn.

If we build the proof tree for ANS(t1 .. tn) and consider the MGUs used in it, the composition of these substitutions, restricted to x1 .. xn, gives us the individuals that answer the original Fill-In-Blanks question.

Inference in FOL - II

Proof as Search

Up to now we have exhibited proofs as if they were found miraculously. We gave formulae and showed proofs of the intended results. We did not exhibit how the proofs were derived.

We now examine how proofs can actually be found. In so doing we stress the close ties between theorem proving and search.

A General Proof Procedure

We use binary resolution [we represent clauses as sets] and we represent the proof as a tree, the **Proof Tree**. In this tree nodes represent clauses. We start with two disjoint sets

of clauses INITIAL and OTHERS.

1. We create a node START and introduce a hyperarc from START to new nodes, each representing a distinct element of INITIAL. We put in a set OPEN all these new nodes. These nodes are called **AND Nodes**.
2. If OPEN is empty, we terminate. Otherwise we remove an element N from OPEN using a selection function SELECT.
3. If N is an AND node, we connect N with a single hyperarc to new nodes N1 ... Nm, one for each literal in the clause C represented at N. These nodes are also labeled with C and are called **OR Nodes**. All of these nodes are placed into OPEN.
   [NOTE 1: In the case that C consists of a single literal, we can just say that N is now an OR node.]

   [NOTE 2: One may decide not to create all the nodes N1 .. Nm in a single action and instead to choose one of the literals of C and to create a node Ni to represent C with this choice of literal. Ni is inserted into OPEN. N also goes back into OPEN if not all of its literals have been considered. The rule used to choose the literal in C is called a **Selection Rule**]

   Repeat from 2.

4. If N is an OR node, say, corresponding to the ith literal of a clause C, we consider all the possible ways of applying binary resolution between C and clauses from the set OTHERS, resolving on the ith literal of C.

   Let D1 .. Dp be the resulting clauses. We represent each of these clauses Dj by an AND node N(Dj) as in 1. above. We put an arc from N to N(Dj). We set OPEN to NEXT-OPEN(OPEN, C, {D1, .., Dp}). We set OTHERS to NEXT-OTHERS(OTHERS, C, {D1, .., Dp}). If in the proof tree we have, starting at START, a hyperpath (i.e. a path that may include hyperarcs) whose leaves have all label {}, we terminate with success.

[NOTE 3: One may decide, instead of resolving C on its ith literal with all possible element of OTHERS, to select one compatible element of OTHERS and to resolve C with it, putting this resolvent and C back into OPEN. We would call a rule for selecting an element of OTHERS a **Search Rule**.]

Repeat from 2.

In this proof procedure we have left indetermined:

- The sets INITIAL and OTHERS of clauses
- The function SELECT by which we choose which node to expand
- The function NEXT-OPEN for computing the next value of OPEN
- The function NEXT-OTHERS for computing the next value of OTHERS

There is no guaranty that for any choice of INITIAL, OTHERS, SELECT, NEXT-OPEN and NEXT-OTHERS the resulting theorem prover will be "complete", i.e. that everything that is provable will be provable with this theorem prover.

Example

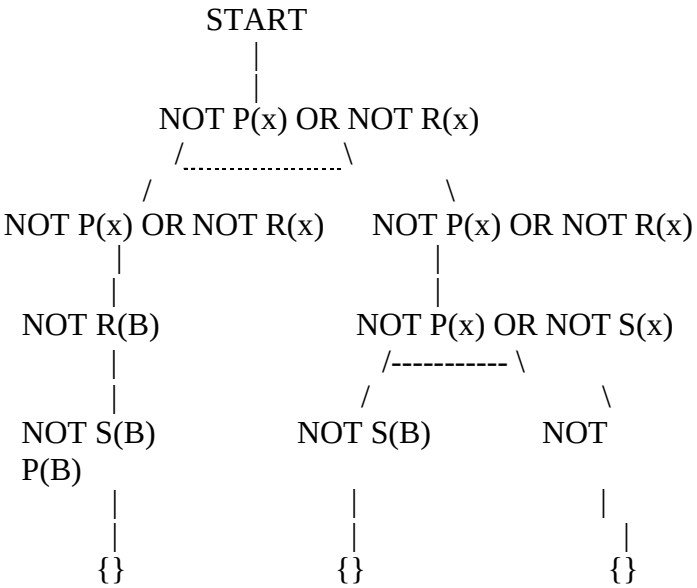
Suppose that we want to prove that

1. NOT P(x) OR NOT R(x)
   is inconsistent with the set of clauses:
2. NOT S(x) OR H(x)
3. NOT S(x) OR R(x)
4. S(b)
5. P(b)

The following are possible selections for the indeterminates:

INITIAL: {1.}, that is, it consists of the clauses representing the negation of the goal.
OTHERS: {2. 3. 4. 5.}, that is, it consists of the non-logical axioms.
SELECT: We use OPEN as a FIFO queue, i.e. we do breadth-first search.
NEXT-OPEN: It sets NEXT-OPEN(OPEN, C, {D1, .., Dp}) to the union of OPEN, {C}, and {D1, .., Dp}.
NEXT-OTHERS: It leaves OTHERS unchanged

The Proof Tree is then (we underline AND nodes and all their outgoing arcs are assumed to form a single hyperlink)



At an OR node we apply resolution between the current clause at its selected literal and all the compatible elements of OTHERS.

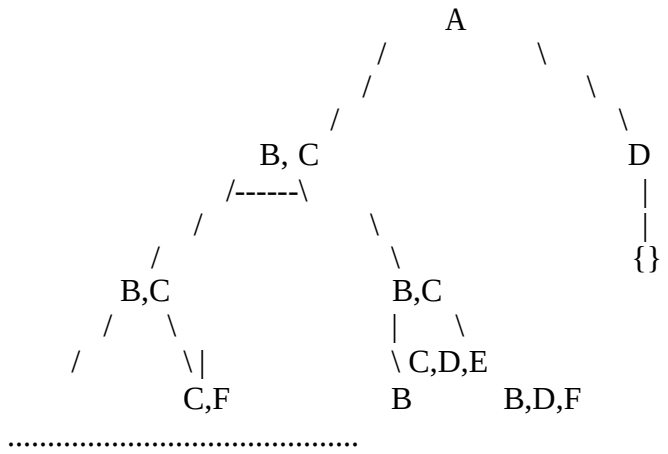
Example

The following example uses Horn clauses in propositional logic. I use the notation that is common in Prolog: we represent the implication:
A1 AND A2 AND .. AND An IMPLIES A
as
A < = A1, A2, .. , An

Our problem is:

1. A, This is what we want to prove
2. A < = B, C
3. A < = D
4. B < = D, E
5. B < = F
6. C < =
7. C < = D, F
8. D < =
9. F < =

We now partially represent the proof tree. We do not apply breadth first because we want to see how great is the impact of the choice of SELECT.



You can keep on expanding the tree and see how depth first would generate a large tree while breadth first rapidly derives D from A, and {} from D.
In other circumstances other strategies would be appropriate as we see below.

Some Proof Strategies

From the early sixties people have looked for strategies that would simplify the search problem in theorem proving. Here are some of the strategies suggested.

Unit Preference

When we apply resolution if one of the premises is a unit clause (it has a single literal), the resolvent will have one less literal than the largest of the premises, thus getting closer to the desired goal {}. Thus it appears desirable to use resolution between clauses one of which is the unit clause. This unit preference is applied both when selecting from the OPEN set (i.e. at the leaves of the tree) and when we at an OR node we select an element of OTHERS to resolve with it.

Set of Support Strategy

When we use the Set of Support Strategy we have:
NEXT-OPEN(OPEN, C, {D1, .., Dp}) is the union of OPEN, {C}, and {D1,..,Dp}
NEXT-OTHERS(OTHERS, C, {D1,..,Dp}) is OTHERS.

In simple words, the set of support strategy uses the OPEN set as its set of support. Each application of resolution has as a premise an element of the set of support and adds that premise and the resolvent to the set of support.

Usually the INITIAL set (i.e. the initial set of support) consists of all the clauses obtained by negating the intended "theorem".
The set of support strategy is complete if

- The OTHERS set of clauses is satisfiable. That is the case when we are given a satisfiable set of non-logical axioms and we use it as OTHERS.

Input Resolution

In Input Resolution:

- INITIAL consists of the union of the negation of the "theorem" and of the set of non-logical axioms.
- OTHERS usually is the set of non-logical axioms.
- NEXT-OPEN(OPEN, C, {D1,..,Dp}) is the union of OPEN and {C}, that is, OPEN does not change in successive iterations
- NEXT-OTHERS(OTHERS, C, {D1,..,Dp}) is the union of OTHERS, {C}, and {D1,..,Dp}.

In other words, in each resolution one of the premises is one of the original clauses.

In general Input Resolution is incomplete, as it can be seen with the following unsatisfiable set of clauses (from Nilsson) from which we are unable to derive FALSE using Input Resolution:

- Q(u) OR P(A)
- NOT Q(w) OR P(w)
- NOT Q(x) OR NOT P(x)
- Q(y) OR NOT P(y)

[You can use this set of clauses as both OPEN and OTHERS.]

Input resolution is complete in the case that all the clauses are Horn clauses.

Linear Resolution

Linear Resolution, also known as Ancestry-Filtered Form Strategy, is a generalization of Input Resolution. The generalization is that now in each resolution one of the clauses is one of the original clauses or it is an ancestor in the proof tree of the second premise.

Linear Resolution is complete.

SLD-Resolution

Selected Linear Resolution for Definite Clauses, or simply SLD-Resolution, is the basis for a proof procedure for theorems that are in the form of conjunctions of positive literals (this conjunction is called a **Goal**) given non-logical axioms that are definite clauses.

More precisely:

- A Goal is the conjunction of positive literals
- A **Selection Rule** is a method for selecting one of the literals of a goal (the first literal, or the last literal, etc.)

- In SLD-Resolution, given a goal G = A1 .. An, a definite clause C: A < = B1 .. Bm, and a subgoal Ai selected from G using a Selection Rule S, where Ai and A are unifiable with MGU s, the **Resolvent of G and C under S** is

(A1 .. Ai-1 B1 ..Bm Ai+1..An).s

- An **SLD-Derivation** of a goal G given a selection rule S and a set P of definite clauses, is a sequence of triples (Gi, Ci, si), for i from 0 to k, where
  - G0 is G
  - for each i > 0, Ci is obtained from a clause in P by replacement of all of its variables with new variables
  - Gi+1 is the SLD-Resolvent of Gi and Ci by use of S and with MGU si.
- A **Refutation** of a goal G given definite clauses P and selection rule S, is a finite SLD-derivation of G given S and P whose last goal is the null clause.
  If s1 .. sk are the MGUs used in the refutation, then s = si.s2... sk is a substitution that, restricted to the variables of G, makes G true whenever the clauses of P are true.
- The goal G **succeeds** for given selection rule S and set of definite clauses P if it has a refutation for P and S; otherwise it **Fails**.

Theorem: SLD-Resolution is Sound and Complete for conjunctive positive goals and definite clauses.

An important consequence of this theorem is that it remains true no matter the selection rule we use to select literals in goals. Thus we can select literals as we please, for instance left-to right. An other important aspect is that the substitution s = s1.s2. sn gives us a method for finding the individuals that satisfy the goal in the structures that satisfy the clauses in P.

Nothing has been said in SLD-Resolution about what rule should be used to select the clause of P to resolve with the current literal of the current goal (such a rule is called a **Search rule**).

Example

Suppose that we are given the following clauses:

- WOMAN(MOTHER(v)) ; Every mother is a woman
- GOOD(HUSBAND(ANN)) ; The husband of Ann is good
- GOOD(z) < = LIKES(MOTHER(z), z); if z likes his mother then z is good

and we want to find out a woman that likes's someone's husband.

The goal can be stated as:

- WOMAN(x),LIKES(x,HUSBAND(y))
  [NOTE: the variables x and y are implicitly existentially quantified.]

The SLD-Derivation is:
((WOMAN(x),LIKES(x,HUSBAND(y))), WOMAN(MOTHER(v)), [MOTHER(v)/x])
((LIKES(MOTHER(v),HUSBAND(y))), GOOD(z) < = LIKES(MOTHER(z), z),
[HUSBAND(y)/v,
HUSBAND(y)/z]) ((GOOD(HUSBAND(y))),
GOOD(HUSBAND(ANN)), [ANN/y])
{}
and the substitution is:
[MOTHER(HUSBAND(ANN))/x, ANN/y]

Non-Monotonic Reasoning

First order logic and the inferences we perform on it is an example of monotonic reasoning.

In monotonic reasoning if we enlarge at set of axioms we cannot retract any existing assertions or axioms.

Humans do not adhere to this monotonic structure when reasoning:

- we need to jump to conclusions in order to plan and, more basically, survive.
  - we cannot anticipate all possible outcomes of our plan.
  - we must make assumptions about things we do not specifically know about.

Default Reasoning

This is a very common from of non-monotonic reasoning. Here *We want to draw conclusions based on what is most likely to be true.*

We have already seen examples of this and possible ways to represent this knowledge.

We will discuss two approaches to do this:

- Non-Monotonic logic.
- Default logic.

DO NOT get confused about the label *Non-Monotonic* and *Default* being applied to reasoning and a particular logic. Non-Monotonic reasoning is generic descriptions of a class of reasoning. Non-Monotonic logic is a specific theory. The same goes for Default reasoning and Default logic.

Non-Monotonic Logic

This is basically an extension of first-order predicate logic to include a *modal* operator, *M*. The purpose of this is to allow for consistency.

For example: $\forall x$: plays_instrument(x) $\wedge M$ improvises(x) $\rightarrow$ jazz_musician(x)

states that for all x is x plays an instrument and if the fact that x can improvise is consistent with all other knowledge then we can conclude that x is a jazz musician.

How do we define *consistency*?

One common solution (consistent with PROLOG notation) is

to show that fact P is true attempt to prove $\neg P$. If we fail we may say that P is consistent (since $\neg P$ is false).

However consider the famous set of assertions relating to President Nixon.

$\forall x$: Republican(x) $\wedge M \neg$ Pacifist(x) $\rightarrow \neg$ Pacifist(x)

$\forall x$: Quaker(x) $\wedge M$ Pacifist(x) $\rightarrow$ Pacifist(x)

Now this states that Quakers tend to be pacifists and Republicans tend not to be.

BUT Nixon was both a Quaker and a Republican so we could assert:

Quaker(Nixon)

Republican(Nixon)

This now leads to our total knowledge becoming inconsistent.

Default Logic

Default logic introduces a new inference rule:

$$\frac{A : B}{C}$$

which states if A is deducible and it is consistent to assume B then conclude C.

Now this is similar to Non-monotonic logic but there are some distinctions:

- New inference rules are used for computing the set of plausible extensions. So in the Nixon example above Default logic can support both assertions since is does not say anything about how choose between them -- it will depend on the inference being made.
- In Default logic any nonmonotonic expressions are rules of inference rather than expressions

Circumscription

Circumscription is a rule of conjecture that allows you to jump to the conclusion that the objects you can show that posses a certain property, *p*, are in fact all the objects that posses that property.

Circumscription can also cope with default reasoning.

Suppose we know: bird(tweety)

$\forall x$: penguin(x) $\rightarrow$ bird(x)

$\forall x$: penguin(x) $\rightarrow \neg$flies(x)

and we wish to add the fact that *typically, birds fly*.

In circumscription this phrase would be stated as:

A bird will fly if it is not abnormal

and can thus be represented by:

$\forall x$: bird(x) $\wedge \neg$abnormal(x) $\rightarrow$flies(x).

However, this is not sufficient

We cannot conclude

flies(tweety)

since we cannot prove

$\neg$ abnormal(tweety).

This is where we apply circumscription and, in this case,

we will assume that those things that are shown to be abnormal are the only things to be

abnormal

Thus we can rewrite our *default rule* as:

$\forall x$: bird(x) $\wedge \neg$ flies(x) $\rightarrow$ abnormal(x)

and add the following

$\forall x$: abnormal(x)

since there is nothing that cannot be shown to be abnormal.

If we now add the fact:

penguin(tweety)

Clearly we can prove

abnormal(tweety).

If we circumscribe abnormal now we would add the sentence,

a penguin (tweety) is the abnormal thing:

$\forall x$: abnormal(x) $\rightarrow$ penguin(x).

Note the distinction between Default logic and circumscription:

Defaults are sentences in language itself not additional inference rules.

Truth Maintenance Systems

A variety of *Truth Maintenance Systems* (TMS) have been developed as a means of implementing Non-Monotonic Reasoning Systems.

Basically TMSs:

- all do some form of dependency directed backtracking
- assertions are connected via a network of dependencies.

Justification-Based Truth Maintenance Systems (JTMS)

- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- Each supported belief (assertion) in has a justification.
- Each justification has two parts:

    - An *IN-List* -- which supports beliefs held.
    - An *OUT-List* -- which supports beliefs *not* held.

- An assertion is connected to its justification by an arrow.
- One assertion can *feed* another justification thus creating the network.
- Assertions may be labelled with a *belief status*.
- An assertion is *valid* if every assertion in the IN-List is believed and none in the OUT-List are believed.
- An assertion is non-monotonic is the OUT-List is not empty or if any assertion in the IN-List is non-monotonic.

Logic-Based Truth Maintenance Systems (LTMS)

Similar to JTMS except:

- Nodes (assertions) assume no relationships among them except ones explicitly stated in justifications.
- JTMS can represent P and P simultaneously. An LTMS would throw a contradiction here.
- If this happens network has to be reconstructed.

Assumption-Based Truth Maintenance Systems (ATMS)

- JTMS and LTMS pursue a single line of reasoning at a time and backtrack (dependency-directed) when needed -- *depth first search*.
- ATMS maintain alternative paths in parallel -- *breadth-first search*
- Backtracking is avoided at the expense of maintaining multiple contexts.
- However as reasoning proceeds contradictions arise and the ATMS can be *pruned*
    - Simply find assertion with no valid justification.

UNIT - III

Advanced Knowledge Representation and Reasoning:

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

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

What to Represent:

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

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

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

Types of knowledge

Following are the various types of knowledge:

1. Declarative Knowledge:

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

2. Procedural Knowledge

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

3. Meta-knowledge:

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

4. Heuristic knowledge:

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

5. Structural knowledge:

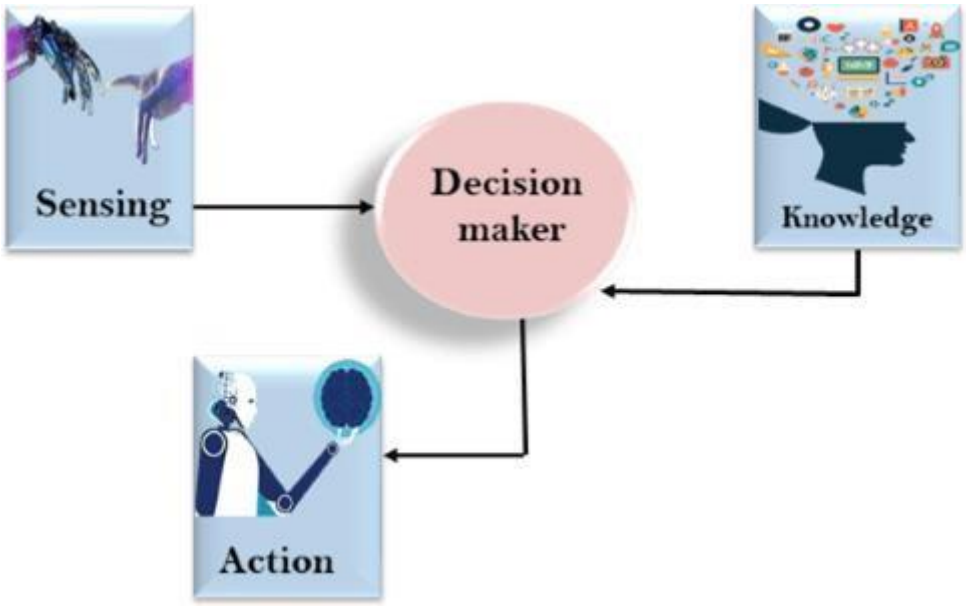
- Structural knowledge is basic knowledge to problem-solving.
- It describes relationships between various concepts such as kind of, part of, and grouping of something.
- It describes the relationship that exists between concepts or objects.

The relation between knowledge and intelligence:

Knowledge of real-worlds plays a vital role in intelligence and same for creating artificial intelligence. Knowledge plays an important role in demonstrating intelligent behavior in AI agents. An agent is only able to accurately act on some input when he has some knowledge or experience about that input.

Let's suppose if you met some person who is speaking in a language which you don't know, then how you will able to act on that. The same thing applies to the intelligent behavior of the agents.

As we can see in below diagram, there is one decision maker which act by sensing the environment and using knowledge. But if the knowledge part will not present then, it cannot display intelligent behavior.

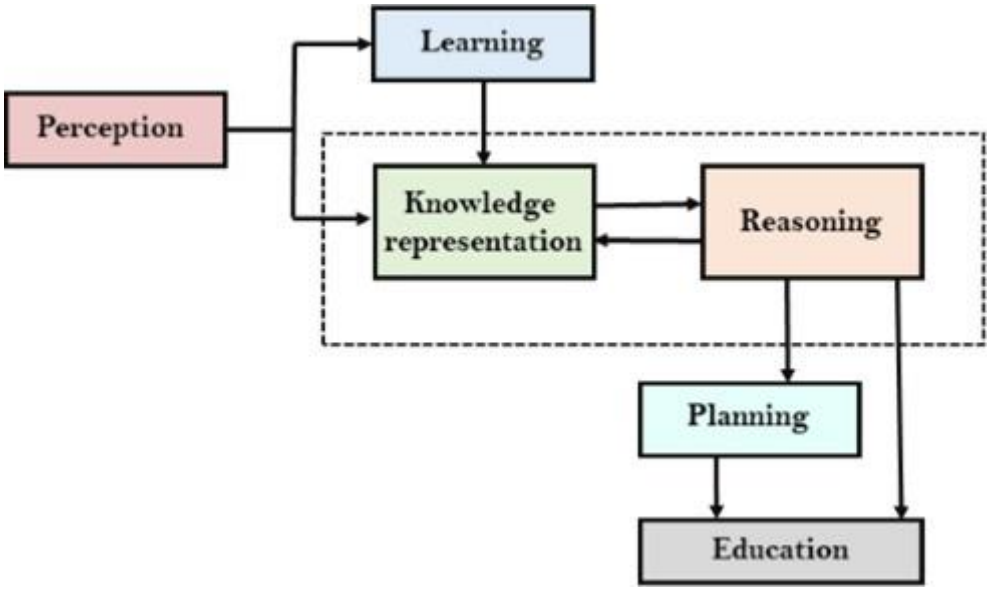


AI knowledge cycle:

An Artificial intelligence system has the following components for displaying intelligent behavior:

- Perception
- Learning
- Knowledge Representation and Reasoning

- Planning
- Execution



The above diagram is showing how an AI system can interact with the real world and what components help it to show intelligence. AI system has Perception component by which it retrieves information from its environment. It can be visual, audio or another form of sensory input. The learning component is responsible for learning from data captured by Perception comportment. In the complete cycle, the main components are knowledge representation and Reasoning. These two components are involved in showing the intelligence in machine-like humans. These two components are independent with each other but also coupled together. The planning and execution depend on analysis of Knowledge representation and reasoning.

Approaches to knowledge representation:

There are mainly four approaches to knowledge representation, which are givenbelow:

1. Simple relational knowledge:

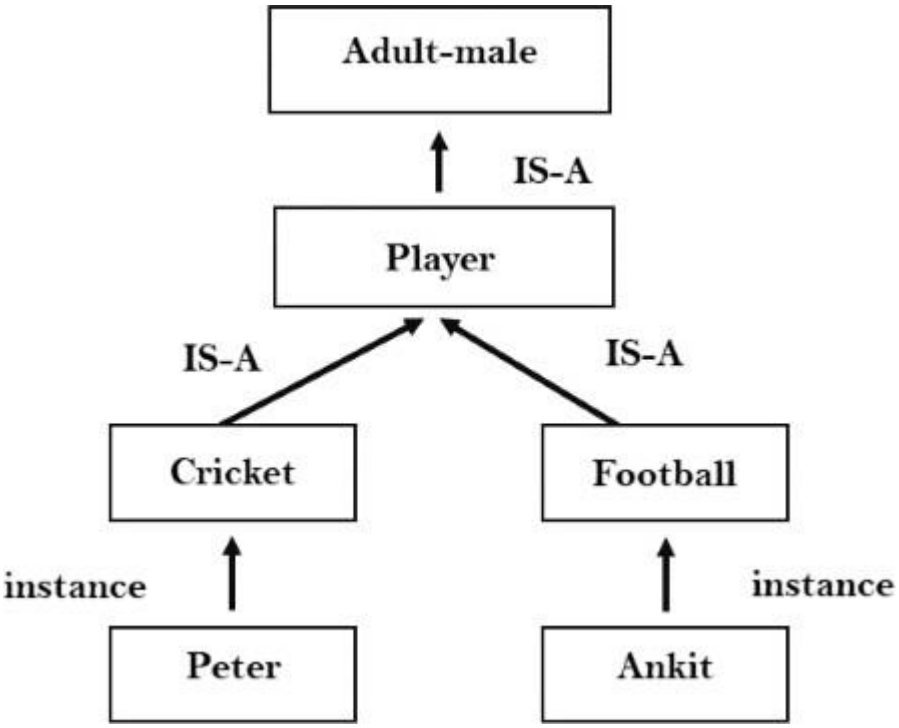
- It is the simplest way of storing facts which uses the relational method, and each fact about a set of the object is set out systematically in columns.
- This approach of knowledge representation is famous in database systems where the relationship between different entities is represented.
- This approach has little opportunity for inference.

Example: The following is the simple relational knowledge representation.

Player	Weight	Age
Player1	65	23
Player2	58	18
Player3	75	24

2. Inheritable knowledge:

- In the inheritable knowledge approach, all data must be stored into a hierarchy of classes.
- All classes should be arranged in a generalized form or a hierarchal manner.
- In this approach, we apply inheritance property.
- Elements inherit values from other members of a class.
- This approach contains inheritable knowledge which shows a relation between instance and class, and it is called instance relation.
- Every individual frame can represent the collection of attributes and its value.
- In this approach, objects and values are represented in Boxed nodes.
- We use Arrows which point from objects to their values.
- **Example:**



3. Inferential knowledge:

- Inferential knowledge approach represents knowledge in the form of formal logics.
- This approach can be used to derive more facts.
- It guaranteed correctness.
- **Example:** Let's suppose there are two statements:
  a. Marcus is a man
  b. All men are mortal
  Then it can represent as;

  man(Marcus)
  ∀x = man (x) ---------- > mortal (x)s

4. Procedural knowledge:

- Procedural knowledge approach uses small programs and codes which describes how to do specific things, and how to proceed.
- In this approach, one important rule is used which is **If-Then rule**.

- In this knowledge, we can use various coding languages such as **LISP language** and **Prolog language**.
- We can easily represent heuristic or domain-specific knowledge using this approach.
- But it is not necessary that we can represent all cases in this approach.

Requirements for knowledge Representation system:

A good knowledge representation system must possess the following properties.

1. **1. Representational Accuracy:**
   KR system should have the ability to represent all kind of required knowledge.
2. **2. Inferential Adequacy:**
   KR system should have ability to manipulate the representational structures to produce new knowledge corresponding to existing structure.
3. **3. Inferential Efficiency:**
   The ability to direct the inferential knowledge mechanism into the most productive directions by storing appropriate guides.
4. **4. Acquisitional efficiency-** The ability to acquire the new knowledge easily using automatic methods.

Reasoning in Artificial intelligence

In previous topics, we have learned various ways of knowledge representation in artificial intelligence. Now we will learn the various ways to reason on this knowledge using different logical schemes.

Reasoning:

The reasoning is the mental process of deriving logical conclusion and making predictions from available knowledge, facts, and beliefs. Or we can say, "**Reasoning is a way to infer facts from existing data**." It is a general process of thinking rationally, to find valid conclusions.

In artificial intelligence, the reasoning is essential so that the machine can also think rationally as a human brain, and can perform like a human.

Types of Reasoning

In artificial intelligence, reasoning can be divided into the following categories:

- Deductive reasoning
- Inductive reasoning
- Abductive reasoning
- Common Sense Reasoning
- Monotonic Reasoning
- Non-monotonic Reasoning

Note: Inductive and deductive reasoning are the forms of propositional logic.

1. Deductive reasoning:

Deductive reasoning is deducing new information from logically related known information. It is the form of valid reasoning, which means the argument's conclusion must be true when the premises are true.

Deductive reasoning is a type of propositional logic in AI, and it requires various rules and facts. It is sometimes referred to as top-down reasoning, and contradictory to inductive reasoning.

In deductive reasoning, the truth of the premises guarantees the truth of the conclusion.

Deductive reasoning mostly starts from the general premises to the specific conclusion, which can be explained as below example.

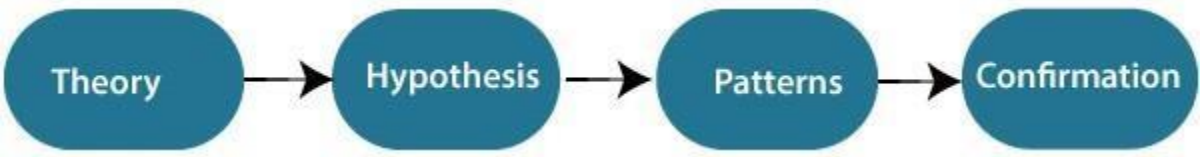
Example:

Premise-1: All the human eats veggies

Premise-2: Suresh is human.

Conclusion: Suresh eats veggies.

The general process of deductive reasoning is given below:



2. Inductive Reasoning:

Inductive reasoning is a form of reasoning to arrive at a conclusion using limited sets of facts by the process of generalization. It starts with the series of specific facts or data and reaches to a general statement or conclusion.

Inductive reasoning is a type of propositional logic, which is also known as cause-effect reasoning or bottom-up reasoning.

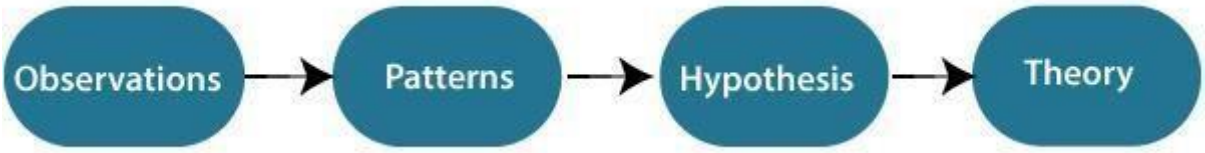
In inductive reasoning, we use historical data or various premises to generate a generic rule, for which premises support the conclusion.

In inductive reasoning, premises provide probable supports to the conclusion, so the truth of premises does not guarantee the truth of the conclusion.

Example:

Premise: All of the pigeons we have seen in the zoo are white.

Conclusion: Therefore, we can expect all the pigeons to be white.



3. Abductive reasoning:

Abductive reasoning is a form of logical reasoning which starts with single or multiple observations then seeks to find the most likely explanation or conclusion for the observation.

Abductive reasoning is an extension of deductive reasoning, but in abductive reasoning, the premises do not guarantee the conclusion.

Example:

Implication: Cricket ground is wet if it is raining

Axiom: Cricket ground is wet.

Conclusion It is raining.

4. Common Sense Reasoning

Common sense reasoning is an informal form of reasoning, which can be gained through experiences.

Common Sense reasoning simulates the human ability to make presumptions about events which occurs on every day.

It relies on good judgment rather than exact logic and operates on **heuristic knowledge** and **heuristic rules**.

Example:

1. **One person can be at one place at a time.**
2. **If I put my hand in a fire, then it will burn.**

The above two statements are the examples of common sense reasoning which a human mind can easily understand and assume.

5. Monotonic Reasoning:

In monotonic reasoning, once the conclusion is taken, then it will remain the same even if we add some other information to existing information in our knowledge base. In monotonic reasoning, adding knowledge does not decrease the set of prepositions that can be derived.

To solve monotonic problems, we can derive the valid conclusion from the available facts only, and it will not be affected by new facts.

Monotonic reasoning is not useful for the real-time systems, as in real time, facts get changed, so we cannot use monotonic reasoning.

Monotonic reasoning is used in conventional reasoning systems, and a logic-based system is monotonic.

Any theorem proving is an example of monotonic reasoning.

Example:

- **Earth revolves around the Sun.**

It is a true fact, and it cannot be changed even if we add another sentence in knowledge base like, "The moon revolves around the earth" Or "Earth is not round," etc.

Advantages of Monotonic Reasoning:

- In monotonic reasoning, each old proof will always remain valid.
- If we deduce some facts from available facts, then it will remain valid for always.

Disadvantages of Monotonic Reasoning:

- We cannot represent the real world scenarios using Monotonic reasoning.
- Hypothesis knowledge cannot be expressed with monotonic reasoning, which means facts should be true.
- Since we can only derive conclusions from the old proofs, so new knowledge from the real world cannot be added.

6. Non-monotonic Reasoning

In Non-monotonic reasoning, some conclusions may be invalidated if we add some more information to our knowledge base.

Logic will be said as non-monotonic if some conclusions can be invalidated by adding more knowledge into our knowledge base.

Non-monotonic reasoning deals with incomplete and uncertain models.

"Human perceptions for various things in daily life, "is a general example of non-monotonic reasoning.

Example: Let suppose the knowledge base contains the following knowledge:

- **Birds can fly**
- **Penguins cannot fly**
- **Pitty is a bird**

So from the above sentences, we can conclude that **Pitty can fly**.

However, if we add one another sentence into knowledge base "**Pitty is a penguin**", which concludes "**Pitty cannot fly**", so it invalidates the above conclusion.

Advantages of Non-monotonic reasoning:

- For real-world systems such as Robot navigation, we can use non-monotonic reasoning.
- In Non-monotonic reasoning, we can choose probabilistic facts or can make assumptions.

Disadvantages of Non-monotonic Reasoning:

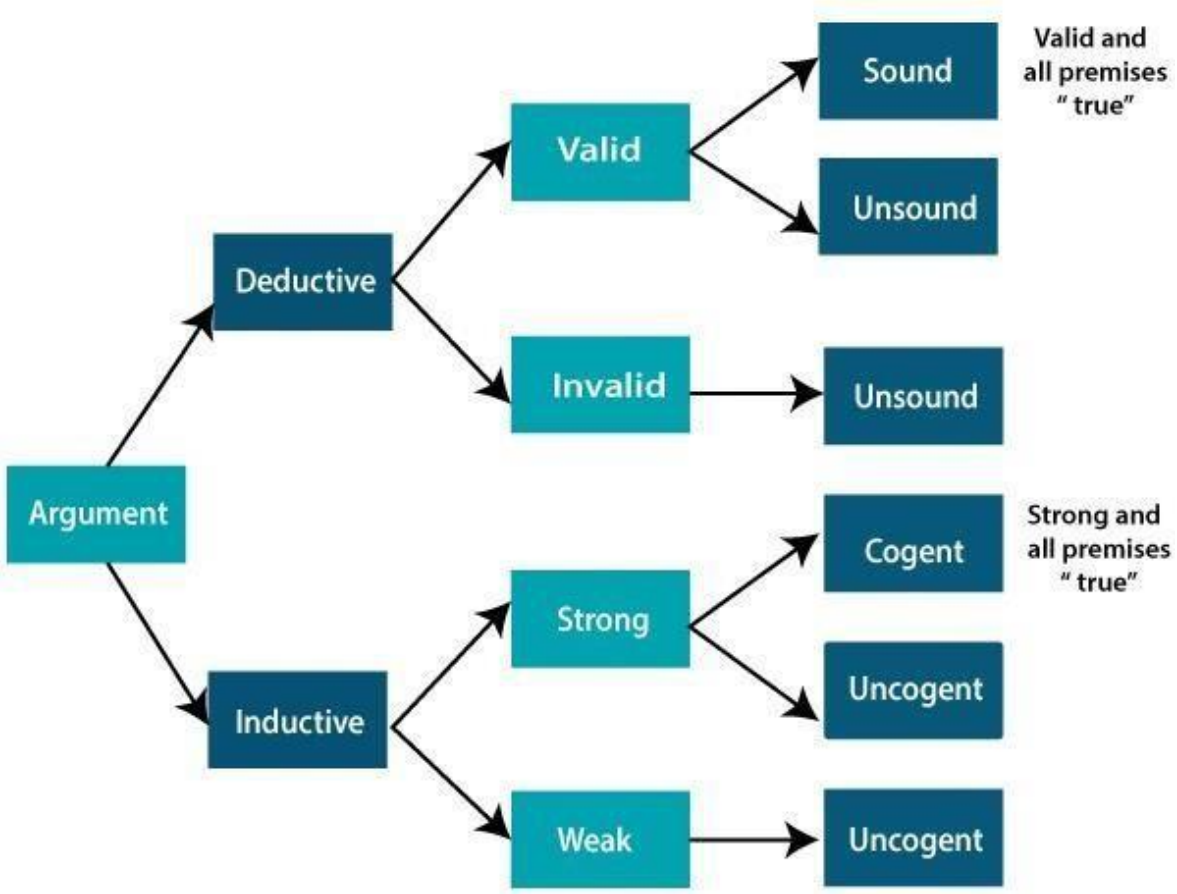
- In non-monotonic reasoning, the old facts may be invalidated by adding new sentences.
- It cannot be used for theorem **proving**.

Difference between Inductive and Deductive reasoning

Reasoning in artificial intelligence has two important forms, Inductive reasoning, and Deductive reasoning. Both reasoning forms have premises and conclusions, but both reasoning are contradictory to each other. Following is a list for comparison between inductive and deductive reasoning:

- Deductive reasoning uses available facts, information, or knowledge to deduce a valid conclusion, whereas inductive reasoning involves making a generalization from specific facts, and observations.
- Deductive reasoning uses a top-down approach, whereas inductive reasoning uses a bottom-up approach.
- Deductive reasoning moves from generalized statement to a valid conclusion, whereas Inductive reasoning moves from specific observation to a generalization.
- In deductive reasoning, the conclusions are certain, whereas, in Inductive reasoning, the conclusions are probabilistic.
- Deductive arguments can be valid or invalid, which means if premises are true, the conclusion must be true, whereas inductive argument can be strong or weak, which means conclusion may be false even if premises are true.

The differences between inductive and deductive can be explained using the below diagram on the basis of arguments:



Comparison Chart:

Basis for comparison	Deductive Reasoning	Inductive Reasoning
Definition	Deductive reasoning is the form of valid reasoning, to deduce new information or conclusion from known related facts and information.	Inductive reasoning arrives at a conclusion by the process of generalization using specific facts or data.
Approach	Deductive reasoning follows a top-down approach.	Inductive reasoning follows a bottom-up approach.
Starts from	Deductive reasoning starts from Premises.	Inductive reasoning starts from the Conclusion.
Validity	In deductive reasoning conclusion must be true if the premises are true.	In inductive reasoning, the truth of premises does not guarantee the truth of conclusions.
Usage	Use of deductive reasoning is difficult, as we need facts which must be true.	Use of inductive reasoning is fast and easy, as we need evidence instead of true facts. We often use it in our daily life.
Process	Theory→ hypothesis→ patterns→confirmation.	Observations-→patterns→hypothesis→Theory.
Argument	In deductive reasoning, arguments may be valid or invalid.	In inductive reasoning, arguments may be weak or strong.
Structure	Deductive reasoning reaches from general facts to specific facts.	Inductive reasoning reaches from specific facts to general facts.

Probabilistic reasoning in Artificial intelligence

Uncertainty:

Till now, we have learned knowledge representation using first-order logic and propositional logic with certainty, which means we were sure about the predicates. With this knowledge representation, we might write A→B, which means if A is true then B is true, but consider a situation where we are not sure about whether A is true or not then we cannot express this statement, this situation is called uncertainty.

So to represent uncertain knowledge, where we are not sure about the predicates, we need uncertain reasoning or probabilistic reasoning.

Causes of uncertainty:

Following are some leading causes of uncertainty to occur in the real world.

1. Information occurred from unreliable sources.
2. Experimental Errors
3. Equipment fault
4. Temperature variation
5. Climate change.

Probabilistic reasoning:

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Need of probabilistic reasoning in AI:

- When there are unpredictable outcomes.
- When specifications or possibilities of predicates becomes too large to handle.
- When an unknown error occurs during an experiment.

In probabilistic reasoning, there are two ways to solve problems with uncertain knowledge:

- **Bayes' rule**
- **Bayesian Statistics**

Note: We will learn the above two rules in later chapters.

As probabilistic reasoning uses probability and related terms, so before understanding probabilistic reasoning, let's understand some common terms:

Probability: Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

1. $0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.
1. P(A) = 0, indicates total uncertainty in an event A.
1. P(A) =1, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.

$$\text{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

- P(¬A) = probability of a not happening event.
- P(¬A) + P(A) = 1.

Event: Each possible outcome of a variable is called an event.

Sample space: The collection of all possible events is called sample space.

Random variables: Random variables are used to represent the events and objects in the real world.

Prior probability: The prior probability of an event is probability computed before observing new information.

Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.

Conditional probability:

Conditional probability is a probability of occurring an event when another event has already happened.

Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

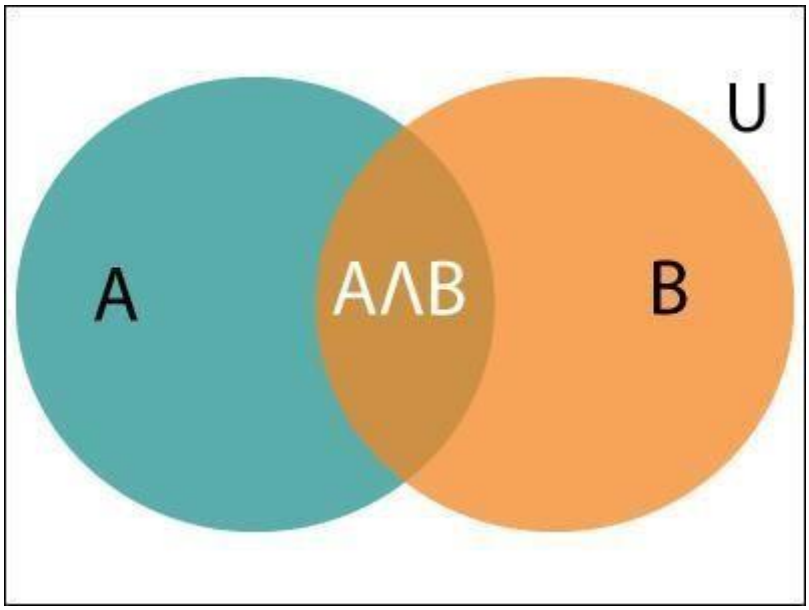
Where $P(A \wedge B)$= Joint probability of a and B

P(B)= Marginal probability of B.

If the probability of A is given and we need to find the probability of B, then it will be given as:

$$P(B|A) = \frac{P(A \wedge B)}{P(A)}$$

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of **P(A∧*B*) by P(B)**.



Example:

In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?

Solution:

Let, A is an event that a student likes Mathematics

B is an event that a student likes English.

$$P(A|B) = \frac{P(A \wedge B)}{P(B)} = \frac{0.4}{0.7} = 57\%$$

Hence, 57% are the students who like English also like Mathematics.

Bayes' theorem in Artificial intelligence

Bayes' theorem:

Bayes' theorem is also known as **Bayes' rule, Bayes' law**, or **Bayesian reasoning**, which determines the probability of an event with uncertain knowledge.

In probability theory, it relates the conditional probability and marginal probabilities of two random events.

Bayes' theorem was named after the British mathematician **Thomas Bayes**. The **Bayesian inference** is an application of Bayes' theorem, which is fundamental to Bayesian statistics.

It is a way to calculate the value of P(B|A) with the knowledge of P(A|B).

Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world.

Example: If cancer corresponds to one's age then by using Bayes' theorem, we can determine the probability of cancer more accurately with the help of age.

Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:

As from product rule we can write:

1. $P(A \wedge B) = P(A|B)\, P(B)$ or

Similarly, the probability of event B with known event A:

1. $P(A \wedge B) = P(B|A)\, P(A)$

Equating right hand side of both the equations, we will get:

$$P(A|B) = \frac{P(B|A)\, P(A)}{P(B)} \quad \text{....(a)}$$

The above equation (a) is called as **Bayes' rule** or **Bayes' theorem**. This equation is basic of most modern AI systems for **probabilistic inference**.

It shows the simple relationship between joint and conditional probabilities. Here,

P(A|B) is known as **posterior**, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B.

P(B|A) is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.

P(A) is called the **prior probability**, probability of hypothesis before considering the evidence

P(B) is called **marginal probability**, pure probability of an evidence.

In the equation (a), in general, we can write P (B) = P(A)*P(B|Ai), hence the Bayes' rule can be written as:

$$P(A_i|B) = \frac{P(A_i) * P(B|A_i)}{\sum_{i=1}^{k} P(A_i) * P(B|A_i)}$$

Where A_1, A_2, A_3,..... , A_n is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

Bayes' rule allows us to compute the single term P(B|A) in terms of P(A|B), P(*B*), and P(A). This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one. Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes:

$$\mathbf{P(cause|effect)} = \frac{P(effect|cause)\,P(cause)}{P(effect)}$$

Example-1:

Question: what is the probability that a patient has diseases meningitis with a stiff neck?

Given Data:

A doctor is aware that disease meningitis causes a patient to have a stiff neck, and it occurs 80% of the time. He is also aware of some more facts, which are given as follows:

- The Known probability that a patient has meningitis disease is 1/30,000.
- The Known probability that a patient has a stiff neck is 2%.

Let a be the proposition that patient has stiff neck and b be the proposition that patient has meningitis. , so we can calculate the following as:

P(a|b) = 0.8

P(b) = 1/30000

P(a)= .02

$$\mathbf{P(b|a)} = \frac{P(a|b)P(b)}{P(a)} = \frac{0.8*(\frac{1}{30000})}{0.02} = 0.001333333.$$

Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

Example-2:

Question: From a standard deck of playing cards, a single card is drawn. The probability that the card is king is 4/52, then calculate posterior probability P(King|Face), which means the drawn face card is a king card.

Solution:

$$P(king|face) = \frac{P(Face|king)*P(King)}{P(Face)} \quad .....(i)$$

P(king): probability that the card is King= 4/52= 1/13

P(face): probability that a card is a face card= 3/13

P(Face|King): probability of face card when we assume it is a king = 1

Putting all values in equation (i) we will get:

$$P(king|face) = \frac{1*(\frac{1}{13})}{(\frac{3}{13})} = 1/3,$$ **it is a probability that a face card is a king card.**

Application of Bayes' theorem in Artificial intelligence:

Following are some applications of Bayes' theorem:

- It is used to calculate the next step of the robot when the already executed step is given.
- Bayes' theorem is helpful in weather forecasting.
- It can solve the Monty Hall problem.

Bayesian Belief Network in artificial intelligence

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a **Bayes network, belief network, decision network**, or **Bayesian model**.

Bayesian networks are probabilistic, because these networks are built from a **probability distribution**, and also use probability theory for prediction and anomaly detection.

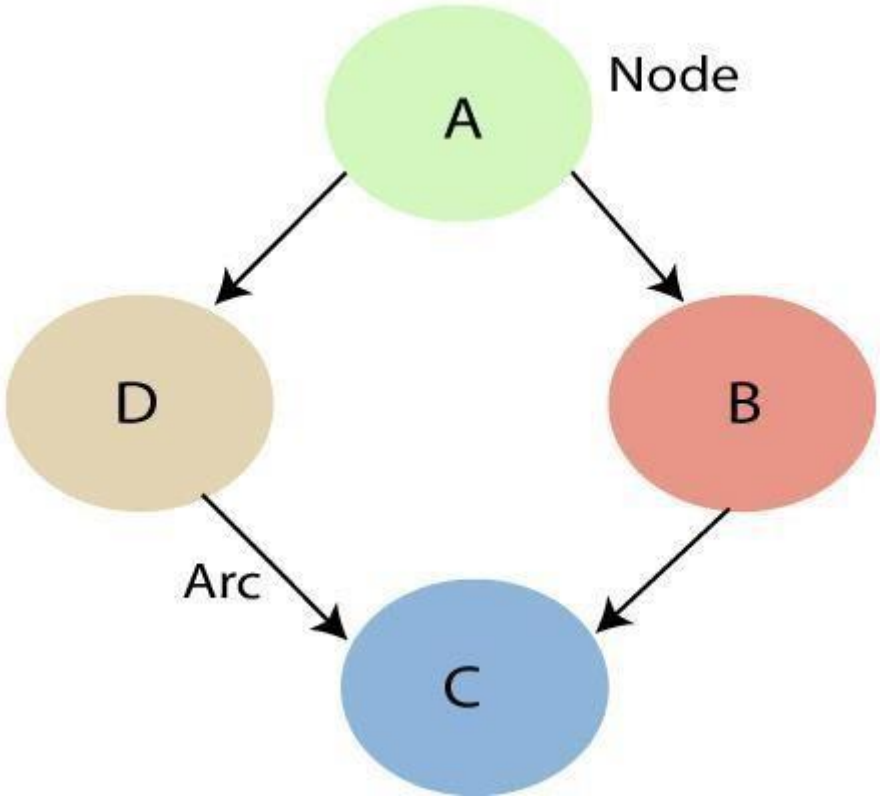
Real world applications are probabilistic in nature, and to represent the relationship between multiple events, we need a Bayesian network. It can also be used in various tasks including **prediction, anomaly detection, diagnostics, automated insight, reasoning, time series prediction**, and **decision making under uncertainty**.

Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

- **Directed Acyclic Graph**
- **Table of conditional probabilities.**

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an **Influence diagram**.

A Bayesian network graph is made up of nodes and Arcs (directed links), where:



- Each **node** corresponds to the random variables, and a variable can be **continuous** or **discrete**.
- **Arc or directed arrows** represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph. These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other
  - **In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.**
  - **If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.**
  - **Node C is independent of node A.**

Note: The Bayesian network graph does not contain any cyclic graph. Hence, it is known as a directed acyclic graph or DAG**.**

The Bayesian network has mainly two components:

- **Causal Component**
- **Actual numbers**

Each node in the Bayesian network has condition probability distribution **$P(X_i | Parent(X_i))$** , which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables x1, x2, x3, ..., xn, then the probabilities of a different combination of x1, x2, x3.. xn, are known as Joint probability distribution.

$P[x_1, x_2, x_3, ..., x_n]$, it can be written as the following way in terms of the joint probability distribution.

$= P[x_1| x_2, x_3,....., x_n]P[x_2, x_3, ..., x_n]$

$= P[x_1| x_2, x_3,....., x_n]P[x_2|x_3,....., x_n]....P[x_{n-1}|x_n]P[x_n].$

In general for each variable Xi, we can write the equation as:

$P(X_i|X_{i-1}, ..., X_1) = P(X_i |Parents(X_i))$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

- The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.
- The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.
- The conditional distributions for each node are given as conditional probabilities table or CPT.
- Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.
- In CPT, a boolean variable with k boolean parents contains 2^K probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

- **Burglary (B)**
- **Earthquake(E)**
- **Alarm(A)**

- **David Calls(D)**
- **Sophia calls(S)**

We can write the events of problem statement in the form of probability: **P[D, S, A, B, E]**, can rewrite the above probability statement using joint probability distribution:

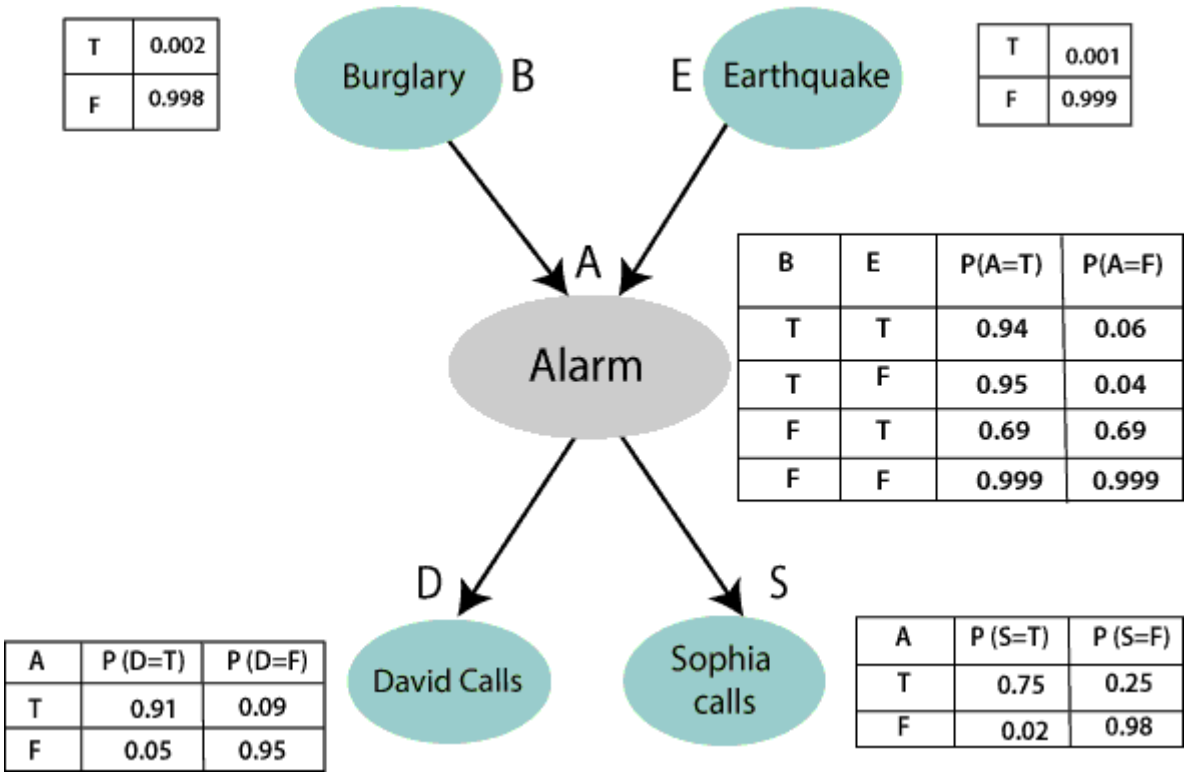
P[D, S, A, B, E]= P[D | S, A, B, E]. P[S, A, B, E]

=P[D | S, A, B, E]. P[S | A, B, E]. P[A, B, E]

= P [D| A]. P [S| A, B, E]. P[A, B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B, E]

= P[D | A]. P[S | A]. P[A| B, E]. P[B |E]. P[E]



T	0.002
F	0.998

T	0.001
F	0.999

B	E	P(A=T)	P(A=F)
T	T	0.94	0.06
T	F	0.95	0.04
F	T	0.69	0.69
F	F	0.999	0.999

A	P (D=T)	P (D=F)
T	0.91	0.09
F	0.05	0.95

A	P (S=T)	P (S=F)
T	0.75	0.25
F	0.02	0.98

Let's take the observed probability for the Burglary and earthquake component:

P(B= True) = 0.002, which is the probability of burglary.

P(B= False)= 0.998, which is the probability of no burglary.

P(E= True)= 0.001, which is the probability of a minor earthquake

P(E= False)= 0.999, Which is the probability that an earthquake not occurred.

We can provide the conditional probabilities as per the below tables:

Conditional probability table for Alarm A:

The Conditional probability of Alarm A depends on Burglar and earthquake:

B	E	P(A= True)	P(A= False)
True	True	0.94	0.06
True	False	0.95	0.04
False	True	0.31	0.69
False	False	0.001	0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A	P(D= True)	P(D= False)
True	0.91	0.09
False	0.05	0.95

Conditional probability table for Sophia Calls:

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

A	P(S= True)	P(S= False)
True	0.75	0.25
False	0.02	0.98

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

P(S, D, A, ¬B, ¬E) = P (S|A) *P (D|A)*P (A|¬B ^ ¬E) *P (¬B) *P (¬E).

= 0.75* 0.91* 0.001* 0.998*0.999

= 0.00068045.

Hence, a Bayesian network can answer any query about the domain by using Joint distribution.

The semantics of Bayesian Network:

There are two ways to understand the semantics of the Bayesian network, which is given below:

1. To understand the network as the representation of the Joint probability distribution.

It is helpful to understand how to construct the network.

2. To understand the network as an encoding of a collection of conditional independence statements.

It is helpful in designing inference procedure.

UNIT-IV

What is learning?

- According to **Herbert Simon**, learning denotes changes in a system that enable a system to do the same task more efficiently the next time.
- **Arthur Samuel stated that**, "Machine learning is the subfield of computer science, that gives computers the ability to learn without being explicitly programmed ".
- In 1997, **Mitchell** proposed that, " A computer program is said to learn from experience **'E'** with respect to some class of tasks **'T'** and performance measure **'P'**, if its performance at tasks in **'T'**, as measured by **'P'**, improves with experience **E** ".
- The main purpose of machine learning is to study and design the algorithms that can be used to produce the predicates from the given dataset.
- Besides these, the machine learning includes the agents percepts for acting as well as to improve their future performance.

The following tasks must be learned by an agent.

- To predict or decide the result state for an action.
- To know the values for each state(understand which state has high or low vale).
- To keep record of relevant percepts.

Why do we require machine learning?

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

Various forms of learnings are explained below:

1. Rote learning

- Rote learning is possible on the basis of memorization.
- This technique mainly focuses on memorization by avoiding the inner complexities. So, it becomes possible for the learner to recall the stored knowledge.

For example: When a learner learns a poem or song by reciting or repeating it, without knowing the actual meaning of the poem or song.

2. Induction learning (Learning by example).

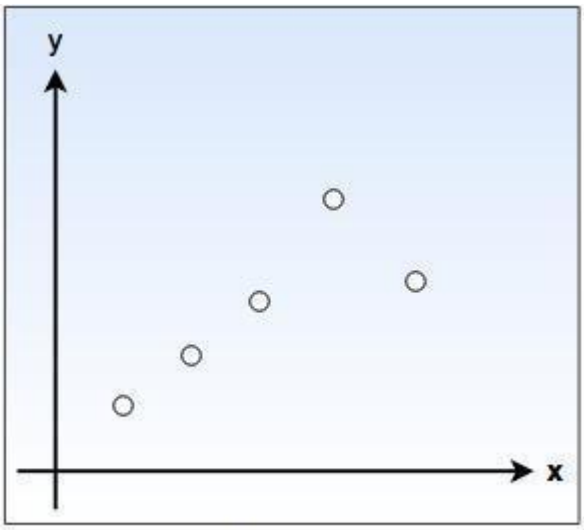
- Induction learning is carried out on the basis of supervised learning.
- In this learning process, a general rule is induced by the system from a set of observed instance.
- However, class definitions can be constructed with the help of a classification method.

For Example:

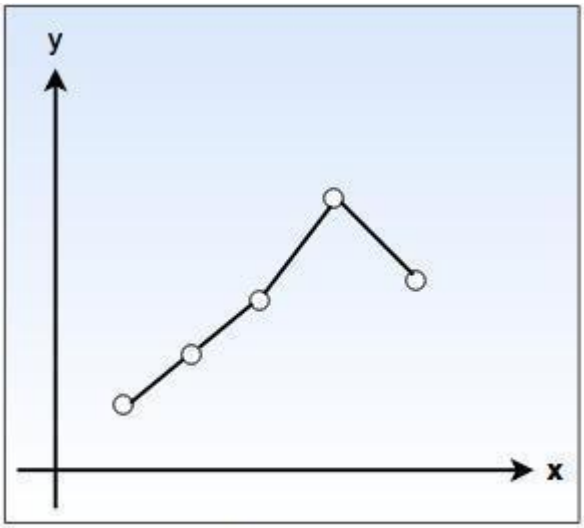
Consider that **'*f*'** is the target function and example is a pair (x $f(x)$), where 'x' is input and $f(x)$ is the output function applied to 'x'.

Given problem: Find hypothesis h such as h $\approx f$

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

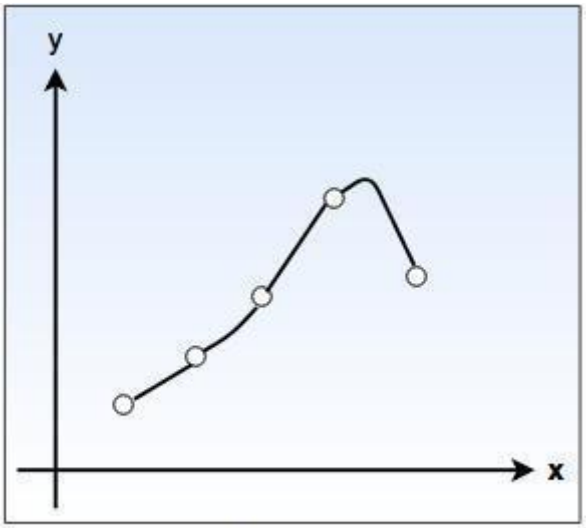


Fig(a)



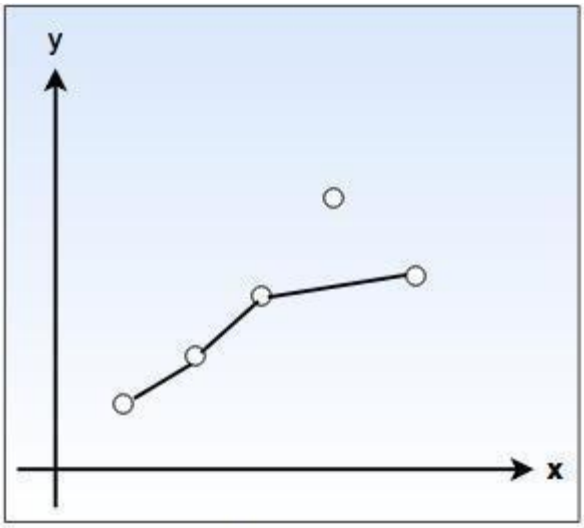
Fig(b)

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



Fig(c)

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



Fig(d)

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

3. Learning by taking advice

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

Explanation based learning

- Explanation-based learning (EBL) deals with an idea of single-example learning.
- This type of learning usually requires a substantial number of training instances but there are two difficulties in this:
  I. it is difficult to have such a number of training instances
  ii. Sometimes, it may help us to learn certain things effectively, specially when we have enough knowledge.
  Hence, it is clear that instance-based learning is more data-intensive, data-driven while EBL is more knowledge-intensive, knowledge-driven.
- Initially, an EBL system accepts a training example.
- On the basis of the given goal concept, an operationality criteria and domain theory, it "generalizes" the training example to describe the goal concept and to satisfy the operationality criteria (which are usually a set of rules that describe relationships between objects and actions in a domain).
- Thus, several applications are possible for the knowledge acquisition and engineering aspects.
  Learning in Problem Solving
- Humans have a tendency to learn by solving various real world problems.
- The forms or representation, or the exact entity, problem solving principle is based on reinforcement learning.
- Therefore, repeating certain action results in desirable outcome while the action is avoided if it results into undesirable outcomes.
- As the outcomes have to be evaluated, this type of learning also involves the definition of a utility function. This function shows how much is a particular outcome worth?
- There are several research issues which include the identification of the learning rate, time and algorithm complexity, convergence, representation (frame and qualification problems), handling of uncertainty (ramification problem), adaptivity and "unlearning" etc.
- In reinforcement learning, the system (and thus the developer) know the desirable outcomes but does not know which actions result into desirable outcomes.
- In such a problem or domain, the effects of performing the actions are usually compounded with side-effects. Thus, it becomes impossible to specify the actions to be performed in accordance to the given parameters.
- Q-Learning is the most widely used reinforcement learning algorithm.
- The main part of an algorithm is a simple value iteration update. For each state 'S', from the state set S, and for each action, a, from the action set 'A', it is possible to calculate an update to its expected reduction reward value, with the following expression:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t (s_t, a_t) [r_t + \gamma \max_a Q (s_{t+1}, a) - Q(s_t, a_t)]$$

- where r_t is a real reward at time t, $\alpha_t(s,a)$ are the learning rates such that $0 \leq \alpha_t(s,a) \leq 1$, and γ is the discount factor such that $0 \leq \gamma < 1$.
- Artificial Neural Network in AI

What is neural network?

Definition:

According to Dr. Robert Hecht-Nielsen, a neural network is defined as, "A computing system made up of a number of simple, highly interconnected processing elements, which process information by their

dynamic state response to external inputs".

- A neural network has the capability of representing the human brain artificially in such a way that it performs the stimulation of its learning process.
- Human brain comprises billions of nerve cells called neurons. These neurons are connected with other cells called as **axons.**
- In neural network, the stimulation process is carried out with the help of **dendrites.**
- These dendrites are connected to sensory organs. Thus, the inputs are accepted from external environment by the sensory organs and are passed to dendrites. These inputs create electric impulses and travel through the neural network.
- A neural network consists of a number of nodes, which are connected by links. Each link is associated with a numeric weight.
- Hence, the learning takes place by updating the weights.
- The units which are connected to the external environment can be used as input or output units.
- The simple model shown in the diagram.

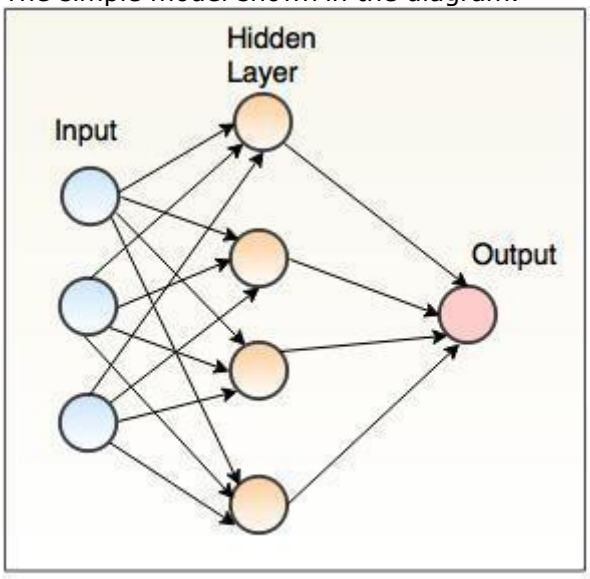


Fig: Artificial Neural Network

In a simple model, the first layer is the input layer, followed by one hidden layer, and lastly by an output layer. Each layer can contain one or more neurons.

- **Winston's Program**
  - Winston's program followed 3 basic steps in concept formulation:

    1. Select one known instance of the concept. Call this the concept definition.
    2. Examine definitions of other known instance of the concept. Generalize the definition to include them.
    3. Examine descriptions of near misses. Restrict the definition to exclude these.

  - Both steps 2 and 3 of this procedure rely heavily on comparison process by which similarities and differences between structures can be detected.
  - Winston's program can be similarly applied to learn other concepts such as "ARCH".

13

Version Spaces

A Version Space is a hierarchical representation of knowledge that keeps track of all the useful information supplied by a sequence of learning examples without remembering any of the examples. The version space method is a concept learning process. It is another approach to learning by Mitchell(1977).

- **Version Space Characteristics**
  - Represents all the alternative plausible descriptions of a heuristic. A plausible description is one that is applicable to all known +ve examples and no known -ve example.
  - A version space description consists of two complementary trees:
    - one, contains nodes connected to overly general models, and
    - other, contains nodes connected to overly specific models.
  - Node values/attributes are discrete.
- **Fundamental Assumptions**
  - The data is correct ie no erroneous instances.
  - A correct description is a conjunction of some attributes with values.

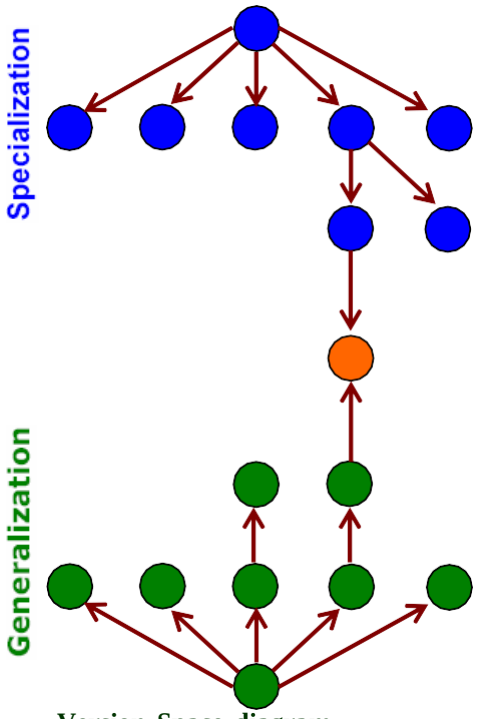
14

- **Version Space Methods**

A version space is a hierarchical representation of knowledge.

Version space method is a concept learning process accomplished by managing multiple models within a versionspace.

It is specialization of general models and generalization of specific models. **Version Space diagram** described below consists : a Specialization tree (colored Blue) and a Generalization tree (colored green) .



Top 1st row : the top of the tree, we have the most general hypotheses.

Top 2nd row : This row is an expanded version of the first. This row of hypotheses is slightly more specific than the root node.

Top 3rd row : As training data (positive examples) is processed, the inconsistent nodes are removed from the general specification.

Finally, we converge to a solution.

Bottom 3rd row : Any hypothesis that is inconsistent with the training data (negative instances) is removed from the tree.

Bottom 2nd row : The specific hypothesis is expanded to form more nodes that are slightly more general.

Bottom 1st row : This is the most specific hypothesis.

Version Space diagram

Tree nodes :

‡ each node is connected to a model.

‡ nodes in the generalization tree are connected to a model that matches everything in its sub-tree.

‡ nodes in the specialization tree are connected to a model that matches only one thing in its sub-tree.

Links between nodes :

‡ denotes generalization relations in a generalization tree, and

‡ denotes specialization relations ina specialization tree.

15

- **Version Space Convergence**

  Generalization and Specialization leads to Version Space convergence.

  The key idea in version space learning is that specialization of the general models and generalization of the specific models may ultimately lead to just one correct model that matches all observed positive examples and does not match any negative examples. This means :

  - Each time a +ve example is used to specialize the general models, then those specific models that match the -ve example are eliminated.
  - Each time a -ve example is used to generalize the specific models, those general models that fail to match the +ve example are eliminated.
  - Finally, the positive and negative examples may be such that only one general model and one identical specific modelsurvive.

16

- **Version Space Learning Algorithm:** Candidate – Elimination

  The Candidate – Elimination algorithm finds all describable hypotheses that are consistent with the observed training examples.

  The version space method handles +ve and -ve examples symmetrically.

  - **Given :** A representation language and a set of positive and negative examples expressed in that language.
  - **Compute :** A concept description that is consistent with all the positive examples and none of the negative examples.

  Note : The methodology / algorithm is explained in the next slide.

[Continued from previous slide - version space]

- **Version Space Search Algorithm**

‡ Let **G** be the set of maximally general hypotheses.

Initialize **G**, to contain most general pattern, a nulldescription.

‡ Let **S** be the set of maximally specific hypotheses.

Initialize **S**, to contain one element : the first positiveexample.

‡ **Accept a new training example.**

◊ For each new positive training example **p** do :

1. Delete all members of **G** that fail to match **p**;
2. For every **s** in **S** do

   If **s** does not match **p**, then replace **s** with its maximally

   specific generalizations that match **p**;
3. Delete from **S** any hypothesis that is subsumed, a more comprehensive, by other hypothesis in **S**;
4. Delete from **S** any hypothesis more general than some hypothesis in **G**.

◊ For each new negative training example **n** do :

1. Delete all members of **S** that match **n**;
2. For each **g** in **G** do

   If **g** match **n**, then replace **g** with its maximally

   general specializations that do not match **n**;
3. Delete from **G** any hypothesis more specific than some other hypothesis in **G**;
4. Delete from **G** any hypothesis that is not more general than some hypothesis in **S**;

◊ If **S** and **G** are both singleton sets, then

1. If they are identical, output their value and halt.
2. If they are different, the training cases were inconsistent; output this result and halt.
3. Else continue accepting new training examples.

‡ The algorithm stops when it runs out of data;

1. If 0 , no consistent description for the data in thelanguage.
2. If 1 , answer (version space converges).
3. If 2 , all descriptions in the language are implicitlyincluded.

- **Version Space – Problem 1**

  Learning the concept of "Japanese Economy Car"

  [Ref: E. Rich, K. Knight, "Artificial Intelligence", McGraw Hill, Second Edition]

Examples 1 to 5 of features

Country of Origin, Manufacturer, Color, Decade, Type

Example	Origin	Manufacturer	Color	Decade	Type	Example Type
1.	Japan	Honda	Blue	1980	Economy	Positive
2.	Japan	Toyota	Green	1970	Sports	Negative
3.	Japan	Toyota	Blue	1990	Economy	Positive
4.	USA	Chrysler	Red	1980	Economy	Negative
5.	Japan	Honda	White	1980	Economy	Positive

Step by step solution : Symbol for Prune 

1. **Initialize :** to maximally general and maximally specific hypotheses

‡ Initialize **G** to a null description, means all features arevariables.

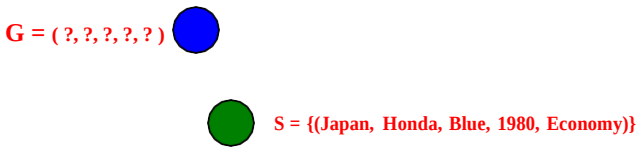
G = { (?, ?, ?, ?, ?) }

‡ Initialize **S** to a set of maximally specific hypotheses; Apply a positive example;

Apply Example 1 (Japan, Honda, Blue, 1980, Economy)

S = { (Japan, Honda, Blue, 1980, Economy) }

‡ Resulting Specialization (blue) and Generalization (green) tree nodes



These two models represent the most general and the most specific heuristics one might learn.

The actual heuristic to be learned, "Japanese economy car", probably lies between them somewhere within the version space.

19

2. **Apply Example 2** (Japan, Toyota, Green, 1970, Sports) It is a negative example.

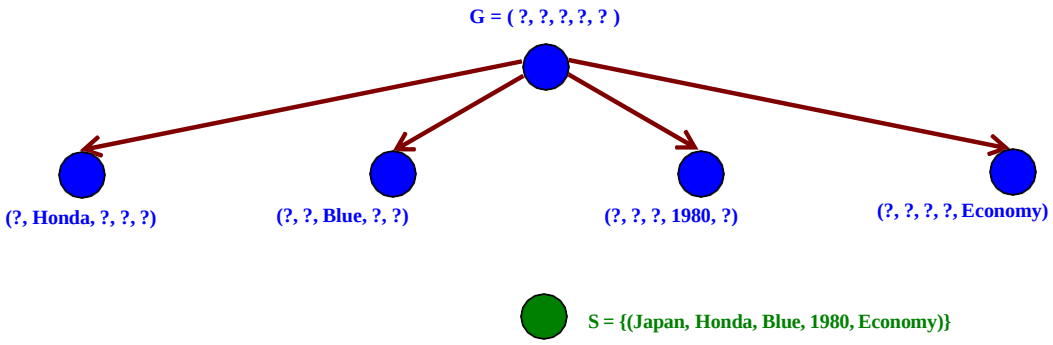
‡ Specialize **G**, to exclude the negative example;

i.e., **G** must be specialized in such a way that the negative example is no longer in the version space.

The available specializations are :

G = { (?, Honda, ?, ?, ?), (?, ?, Blue, ?, ?), (?, ?, ?, 1980, ?),
(?, ?, ?, ?, Economy) }

‡ Resulting Specialization (blue) and Generalization (green) tree nodes and links



20

3. **Apply Example 3** (Japan, Toyota, Blue, 1990, Economy) It is a positive example.

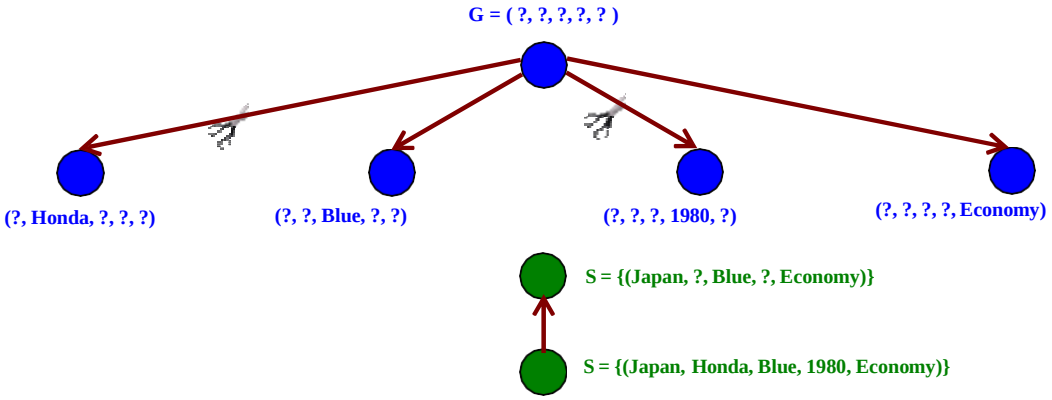
‡ Prune **G ,** to exclude descriptions inconsistent with the positive example; i.e., remove from the **G** set any descriptions that are inconsistent with the positive example. The new **G** set is

G = { (?, ?, Blue, ?, ?), (?, ?, ?, ?, Economy) }

‡ Generalize **S**, to include the new example. The new **S** set is

S = { (Japan, ? , Blue, ? , Economy) }

‡ Resulting Specialization (blue) and Generalization (green) tree nodes and links :



‡ At this point, the new **G** and **S** sets specify a version space that can be translated roughly in to English as : The concept may be as specific as "Japanese, blue economycar".

21

4. **Apply Example 4** (USA, Chrysler, Red, 1980, Economy) It is another negative example.

‡ Specialize **G** to exclude the negative example; (but stay consistent with **S**) i.e.;

G set must avoid covering this new example.

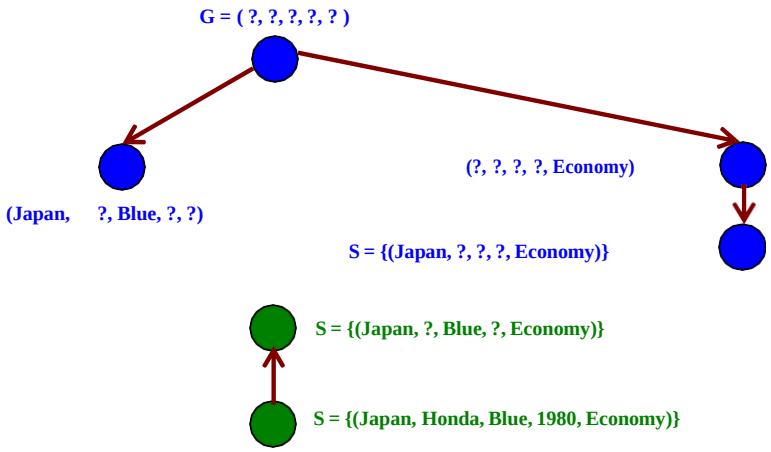
The new **G** set is

G = { (?, ?, Blue, ?, ?), (Japan, ?, ?, ?, Economy) }

‡ Prune away all the specific models that match the negative example. No match found therefore no change in **S**

S = { (Japan, ?, Blue, ?, Economy) }

‡ Resulting Specialization (blue) and Generalization (green) tree nodes and links :



‡ It is now clear that the car must be Japanese, because all description in the version space contain Japan as origin.

22

5. **Apply Example 5** (Japan, Honda, White, 1980, Economy) It is positive example.

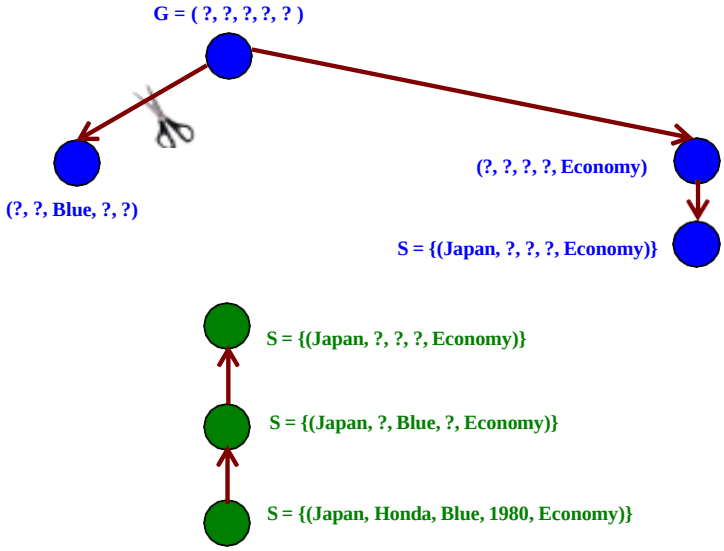
‡ Prune **G** to exclude descriptions inconsistent with positive example.

G = { (Japan, ?, ?, ?, Economy) }

‡ Generalize **S** to include this positive example.

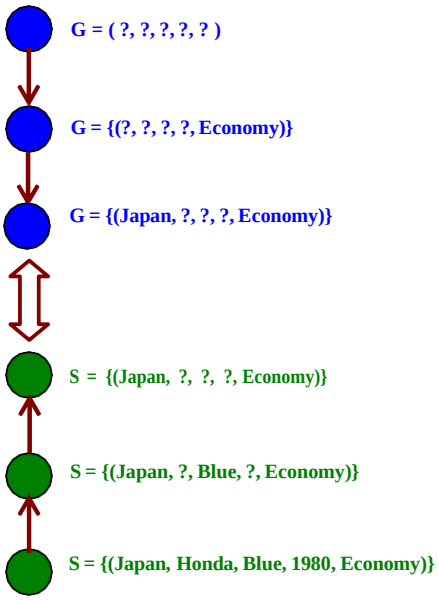
S = { (Japan, ?, ?, ?, Economy) }

‡ Resulting Specialization (blue) and Generalization (green) tree nodes and links :



‡ Thus, finally **G** and **S** are singleton sets and **S = G**. The algorithm has converged. No more examples needed.

Algorithmstops.

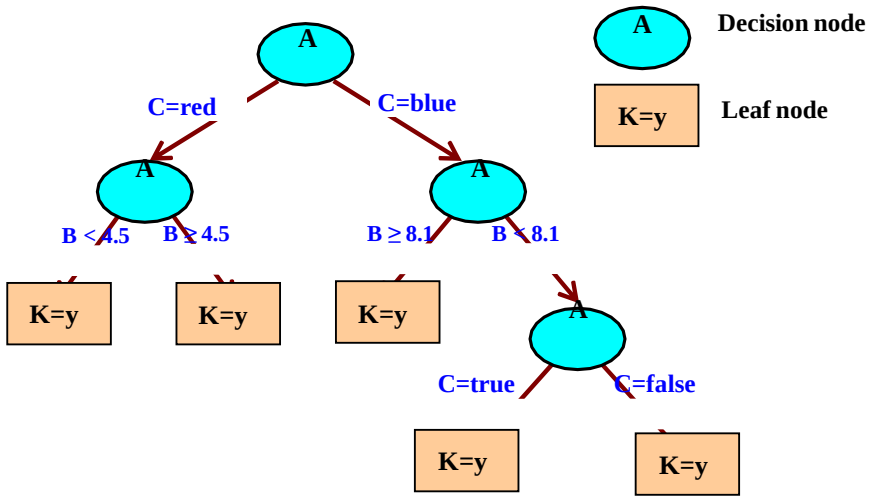


Decision Trees

Decision trees are powerful tools for classification and prediction.

Decision trees represent rules. Rules are easily expressed so that humans can understand them or even directly use in a database access language like SQL so that records falling into a particular category may be retrieved.

- **Description**
  - Decision tree is a classifier in the form of a tree structure where each node is either a leaf or decision node.
    - ‡ **leaf node** – indicates the target attribute (class) values of examples.
    - ‡ **decision node** – specify test to be carried on an attribute-value.
  - Decision tree is a typical inductive approach to learn knowledge on classification. The conditions are :
    - ‡ **Attribute-value description:** Object or case must be expressible as a fixed collection of properties or attributes having discrete values.
    - ‡ **Predefined classes :** Categories to which examples are to be assigned must already be defined (ie supervised data).
    - ‡ Discrete classes: Classes must be sharply delineated; continuous classes broken up into vague categories as "hard", "flexible", "soft".
    - ‡ Sufficient data: Enough training cases to distinguish valid patterns.
- **Example :** A simple decision tree



24

- **Algorithm to generate a Decision Tree**

  A decision tree program constructs a decision tree **T** from a set of training cases. The advantage of a decision tree learning is that a program, rather than a knowledge engineer, elicits knowledge from an expert.

  ID3 Algorithm (Iterative Dichotomiser 3)

  ID3 is based on the Concept Learning System (CLS) algorithm, developed J. Ross Quinlan, at the University of Sydney in 1975.

  - **Description**

    ‡ ID3 builds a decision tree from a set of "examples". The "examples" have several attributes. They belong to a class (yes or no).

    ‡ Leaf nodes contain the class name.

    ‡ Decision node is an attribute test .

25

- **Attribute Selection**

How does ID3 decide which attribute is the best?

There are different criteria to select which attribute will become a test attribute in a given branch of a tree. A well known criteria is information gain. It uses a log function with a base of 2, to determine the number of bits necessary to represent a piece of information.

‡ **Entropy** measures information content in anattribute.

Shannon defines information entropy in terms of a discrete random variable **X** , with possible states (or outcomes) $x_1 \ldots x_n$ as:

$$H(X) = \sum_{i=1}^{n} p(xi) \cdot \log_2 (1/p(xi)) = - \sum_{i=1}^{n} p(xi) \cdot \log_2 p(xi)$$

where $p(xi) = P(X = xi)$ is the probability of the i^{th} outcome of **X**.

‡ **Information gain** of an attribute **X** with respect to class attribute **Y**.

Let **Y** and **X** are discrete variables that take values in $\{y_1 \ldots y_i \ldots y_k\}$ and $\{x_1 \ldots x_j \ldots x_l\}$.

The information gain of a given attribute **X** with respect to theclass attribute **Y** is the reduction in uncertainty about the value of **Y** when we know the value of **X**, is called **I(Y ; X)**.

Uncertainty

Uncertainty about the value of **Y** is measured by its **entropy**, **H(Y)**. uncertainty about the value of **Y** when we know the value of **X** is given by the **conditional entropy** of **Y** given **X**, i.e., **H(Y |X)**.

Information Gain of a given attribute **X** with respect to the class attribute **Y** is given by : **I(Y ; X) = H(Y) - H(Y |X)**. where

$H(Y) = - \sum_{yi=1}^{k} p(yi) \cdot \log_2 p(yi)$ is the initial entropy in **Y**,

$H(Y|X) = \sum_{xj=1}^{l} p(xj) \cdot H(Y|xj)$ is conditional entropy of **Y** given **X** where $H(Y|xj) = - \sum_{xj=1}^{l} p(yi \mid xj) \cdot \log_2 p(yi \mid xj)$ is the entropy in **Y** given a particular answer x_j .

- **Example**

  Decide if the weather is amenable to play Golf or Baseball.

  ‡ **Training Data** Collected are Over 2 weeks

Day	Outlook	Temp	Humidity	wind	Play
1	sunny	85	85	week	no
2	sunny	80	90	strong	no
3	cloudy	83	78	week	yes
4	rainy	70	96	week	yes
5	rainy	68	80	week	yes
6	rainy	65	70	strong	no
7	cloudy	64	65	strong	yes
8	sunny	72	95	week	no
9	sunny	69	70	week	yes
10	rainy	75	80	week	yes
11	sunny	75	70	strong	yes
12	cloudy	72	90	strong	yes
13	cloudy	81	75	week	yes
14	rainy	71	85	strong	no

  ‡ **Learning set**

  In the above example, two attributes, the Temperature and Humidity have continuous ranges. ID3 requires them to be discrete like hot, medium, cold, high, normal. Table below indicates the acceptable values.

Attribute	Possible Values		
Outlook	Sunny	Cloudy	Rainy
Temperature	Hot	Medium	Cold
Humidity Wind	High	Normal	
Class	Strong	Week	
Decision	play	no play	
	n (negative)	p (positive)	

27

‡ Assign discrete values to the Attributes

Partition the continuous attribute values to make them discrete, following the key mentioned below.

Temperature : Hot (H) 80 to 85 Medium (M) 70 to 75 Cold (C) 64 to 69

Humidity : High (H) 81 to 96 Normal (N) 65 to 80

Class : Yes (Y) play No (N) no play

Day	Outlook	Temp		Humidity		Wind	Class (play)
1	Sunny	85	Hot	85	High	week	no
2	Sunny	80	Hot	90	High	strong	no
3	Cloudy	83	Hot	78	High	week	yes
4	Rainy	70	Medium	96	High	week	yes
5	Rainy	68	Cold	80	Normal	week	yes
6	Rainy	65	Cold	70	Normal	strong	no
7	Cloudy	64	Cold	65	Normal	strong	yes
8	Sunny	72	Medium	95	High	week	no
9	Sunny	69	Cold	70	Normal	week	yes
10	Rainy	75	Medium	80	Normal	week	yes
11	Sunny	75	Medium	70	Normal	strong	yes
12	Cloudy	72	Medium	90	High	strong	yes
13	Cloudy	81	Hot	75	Normal	week	yes
14	Rainy	71	Medium	85	High	strong	no

28

‡ Attribute Selection

By applying definitions of Entropy and Information gain

◊ **Entropy** : From the definition, When **Y** is a discrete variables that take values in **{y_1 . . . y_i . . . y_k}** then the entropy of **Y** is given by **H(Y)**

$$H(Y) = \sum_{i=1}^{k} p(yi) \bullet \log_2 (1/p(yi)) = - \sum_{i=1}^{k} p(yi) \bullet \log_2 p(yi)$$

where **p(yi)** is the probability of the **i^{th}** outcome of **Y**.

Apply his definition to the example stated before :

Given, **S** is a collection of 14 examples with 9 YES and 5 NO then **Entropy(S) = - (9/14) Log_2 (9/14) - (5/14) Log_2 (5/14)**

= 0.940

Note that **S** is not an attribute but the entire sample set.

Entropy range from 0 ("perfectly classified") to 1 ("totally random").

[Continued in next slide]

29

[Continued from previous slide – Attribute election]

◊

Information Gain : From the definition, a given attribute **X** with respect to the class attribute **Y** is given by :

$I(Y ; X) = H(Y) - H(Y|X)$ where

$H(Y) = -\sum_{yi=1}^{k} p(y_i) \cdot \log_2 p(y_i)$ is the initial entropy in **Y**,

$H(Y|X) = \sum_{xj=1}^{l} p(x_j) \cdot H(Y|x_j)$ is conditional entropy of **Y** given **X** where $H(Y|x_j) = -\sum_{xj=1}^{l} p(y_i | x_j) \cdot \log_2 p(y_i | x_j)$ is entropy in **Y** given a particular answer x_j .

Apply this definition to the example stated above:

Information **Gain (S, A)** of the example set **S** on attribute **A** is

Gain(S, A) = Entropy(S) - ∑ ((|Sv| / |S|) * Entropy(Sv)) where

∑ sum over each value **v** of all possible values of attribute **A** **Sv** is subset of **S** for which attribute **A** has value **v**

|Sv| is number of elements in **Sv**

|S| is number of elements in **S**

Given, **S** is a set of 14 examples, in which one of the attributes is wind speed. The values of Wind can be Weak or Strong. The classification of these 14 examples are 9 YES and 5 NO. Given, for attribute **wind**:

8 occurrences of **wind = weak** and

6 occurrences of **wind = strong**.

For **wind = weak**, **6** of the examples are **YES** and **2** are **NO**. For **wind = strong**, **3** of the examples are **YES** and **3** are **NO**. **Gain(S, wind) = Entropy(S) - (8/14) • Entropy(S_{weak})**

– (6/14) • Entropy(S_{strong})

= 0.940 - (8/14) • 0.811 - (6/14) • 1.00 = 0.048

Entropy(S_{weak}) = - (6/8) • $\log_2$(6/8) - (2/8) • $\log_2$(2/8) = 0.811 Entropy(S_{strong}) = - (3/6) • $\log_2$(3/6) - (3/6) • $\log_2$(3/6) = 1.00 For each attribute, the gain is calculated and the highest gain is used in the decision node.

30

‡ Step-by-Step Calculations : *Ref. Training data stated before*

◊ Step 01 : "example" set S

The set S of 14 examples is with 9 ‘yes’ and 5 ‘no’ then

Entropy(S) = – (9/14) Log2 (9/14) – (5/14) Log2 (5/14) = 0.940

◊ Step 02 : Attribute Outlook

Outlook value can be sunny, cloudy orrainy. Outlook =

sunny is of occurrences 5 Outlook = cloudy is of

occurrences 4 Outlook = rainy is of

occurrences 5

Outlook = sunny, 2 of the examples are ‘yes’ and 3 are ‘no’ Outlook = cloudy, 4 of the examples are ‘yes’ and 0 are ‘no’ Outlook = rainy, 3 of the examples are ‘yes’ and 2 are ‘no’

Entropy(Ssunny) = – (2/5) x log2(2/5) – (3/5) x log2(3/5) = 0.970950

Entropy(Scloudy) = – (4/4) x log2(4/4) – (0/4) x log2(0/4) = 0

Entropy(Srainy) = – (3/5) x log2(3/5) – (2/5) x log2(2/5) = 0.970950

Gain(S, Outlook) = Entropy(S) – (5/14) x Entropy(Sunny)

– (4/14) xEntropy(Scloud)

– (5/14) xEntropy(Srainy)

= 0.940 - (5/14) x 0.97095059 – (4/14) x 0

- (5/14) x 0.97095059

= 0.940 - 0.34676 - 0 - 0.34676

= 0.246

31

◊ **Step 03 : Attribute Temperature** Temp value can be hot, medium or cold. Temp = hot is of occurrences 4

Temp = medium is of occurrences 6 Temp = cold is of occurrences 4

Temp = hot, 2 of the examples are 'yes' and 2 are 'no' Temp = medium, 4 of the examples are 'yes' and 2 are 'no' Temp = cold, 3 of the examples are 'yes' and 1 are 'no'

Entropy(Shot) = – (2/4)x log2(2/4) – (2/4) x log2(2/4) = -0.99999999

Entropy(Smedium) = – (4/6)x log2(4/6) – (2/6) x log2(2/6) = -0.91829583

Entropy(Scold) = – (3/4)x log2(3/4) – (1/4) x log2(1/4) = -0.81127812

Gain(S, Temp) = Entropy(S) – (4/14) x Entropy(Shot)
– (6/14) x Entropy(Smedium)
– (4/14) x Entropy(Scold)

= 0.940 - (4/14) x 0.99999 – (6/14) x 0.91829583
- (4/14) x 0.81127812

= 0.940 - 0.2857142 -0.393555 - 0.2317937

= 0 . 0289366072

32

◊ **Step 04 : Attribute Humidity** Humidity value can be high, normal. Humidity = high is of occurrences 7 Humidity = normal is of occurrences 7

Humidity = high, 3 of the examples are 'yes' and 4 are 'no' Humidity = normal, 6 of the examples are 'yes' and 1 are'no'

Entropy(Shigh) = – (3/7) x log2(3/7) – (4/7) x log2(4/7) = -0.9852281

Entropy(Snormal) = – (6/7) x log2(6/7) – (1/7) x log2(1/7) = -0.5916727

Gain(S, Humidity) = Entropy(S) – (7/14) x Entropy(Shigh)
– (7/14) x Entropy(Snomal)
= 0.940 - (7/14) x 0.9852281 – (7/14) x 0.5916727
= 0.940 - 0.49261405 - 0.29583635
= 0 . 1515496

33

◊ **Step 05 : Attribute wind**

Wind value can be weak or strong. Wind =

week is of occurrences 8 Wind

= strong is of occurrences 6

Wind = weak, 6 of the examples are 'yes' and 2 are'no' Wind = strong, 3 of the examples are 'yes' and 3 are 'no'

Entropy(S_{weak}) = – (6/8) x log2(6/8) – (2/8) x log2(2/8) = 0.811

Entropy(S_{strong}) = – (3/6) x log2(3/6) – (3/6) x log2(3/6) = 1.00

Gain(S, Wind) = Entropy(S) – (8/14) x Entropy(S_{weak}) – (6/14) xEntropy(S_{strong})

= 0.940 – (8/14) x 0.811 – (6/14)*1.00

= 0.048

34

◊ **Step 06 : Summary of results are**

Entropy(S) = 0.940

Gain(S, Outlook) = **0.246**

Gain(S, Temp) = 0.0289366072

Gain(S, Humidity) = 0.1515496

Gain(S, Wind) = 0.048

◊ **Step 07 : Find which attribute is the root node.**

Gain(S, Outlook) = 0 .246 is highest.

Therefore "Outlook" attribute is the decision attribute in the root node. "Outlook" as root node has three possible values - **sunny, cloudy, rain**.

S.No	Day	Outlook	Temp		Humidity		Wind	Play
1	11	sunny	75	Medium	70	Normal	strong	yes
2	2	sunny	80	Hot	90	High	strong	no
3	1	sunny	85	Hot	85	High	week	no
4	8	sunny	72	Medium	95	High	week	no
5	9	sunny	69	Cold	70	Normal	week	yes
6	12	cloudy	72	Medium	90	High	strong	yes
7	3	cloudy	83	Hot	78	High	week	yes
8	7	cloudy	64	Cold	65	Normal	strong	yes
9	13	cloudy	81	Hot	75	Normal	week	yes
10	14	rainy	71	Medium	85	High	strong	no
11	6	rainy	65	Cold	70	Normal	strong	no
12	10	rainy	75	Medium	80	Normal	week	yes
13	5	rainy	68	Cold	80	Normal	week	yes
14	4	rainy	70	Medium	96	High	week	yes

◊ **Step 08 : Find which attribute is next decision node.**

Outlook has three possible values,

so root node has three branches (sunny, cloudy, rain).

Ssunny = {D1, D2, D8, D9, D11} = 5 "examples". "D" represent "Days". With outlook = sunny

Gain(Ssunny, Humidity) = **0.970**

Gain(Ssunny, Temp) = 0.570

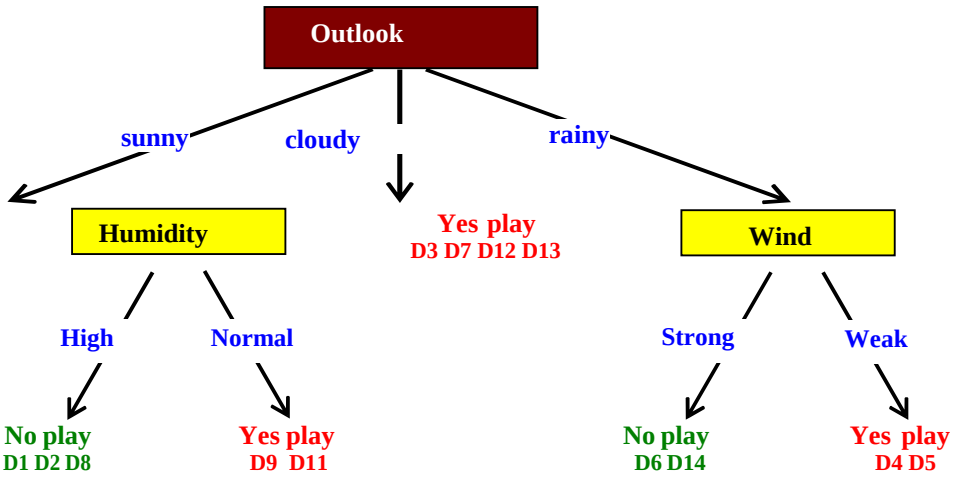
Gain(Ssunny, Wind) = 0.019

Humidity has the highest gain 0.970 is next decision node.

◊ Step 09 : This process goes on until all days data are classified perfectly or run out of attributes.

36

- Thus, the classification tree built using ID3 algorithm is shown below. It tells if the weather was amenable to play?



Decision tree

37

- **Applications**

  ‡ ID3 is easy to use.

  ‡ Its primary use is replacing the expert who would normally build a classification tree by hand.

  ‡ ID3 has been incorporated in a number of commercial rule-induction packages.

  ‡ Some specific applications include

  ◊ medical diagnosis,

  ◊ credit risk assessment of loan applications,

  ◊ equipment malfunctions by their cause,

  ◊ classification of plant diseases, and

  ◊ web search classification.

38

4. Explanation Based Learning (EBL)

Humans appear to learn quite a lot from oneexample.

Human learning is accomplished by examining particular situations and relating them to the background knowledge in the form of known general principles.

This kind of learning is called "Explanation Based Learning (EBL)".

General Approach

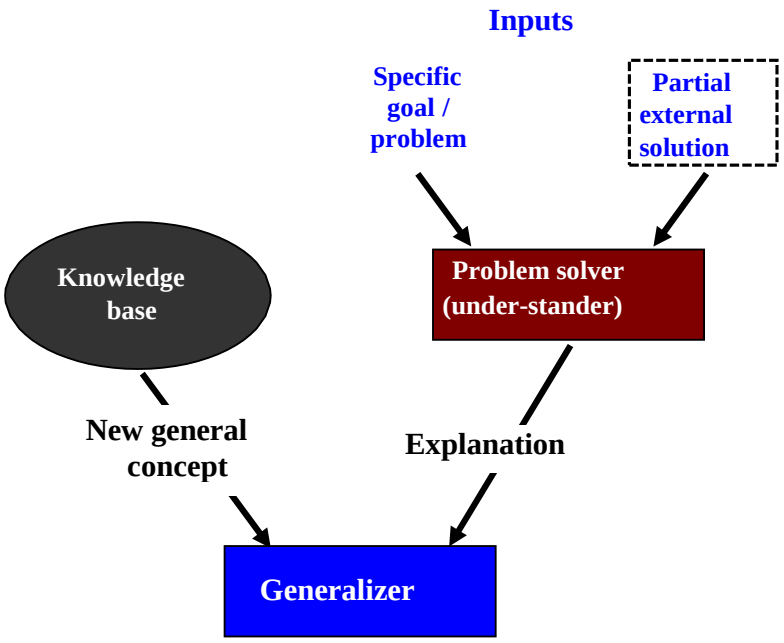
EBL is abstracting a general concept from a particular training example.

EBL is a technique to formulate general concepts on the basis of a specific training example. EBL analyses the specific training example in terms of domain knowledge and the goal concept. The result of EBL is an explanation structure, that explains why the training example is an instance of the goal concept. The explanation-structure is then used as the basis for formulating the general concept.

Thus, EBL provides a way of generalizing a machine-generated explanation of a situation into rules that apply not only to the current situation but to similar ones as well.

EBL Architecture

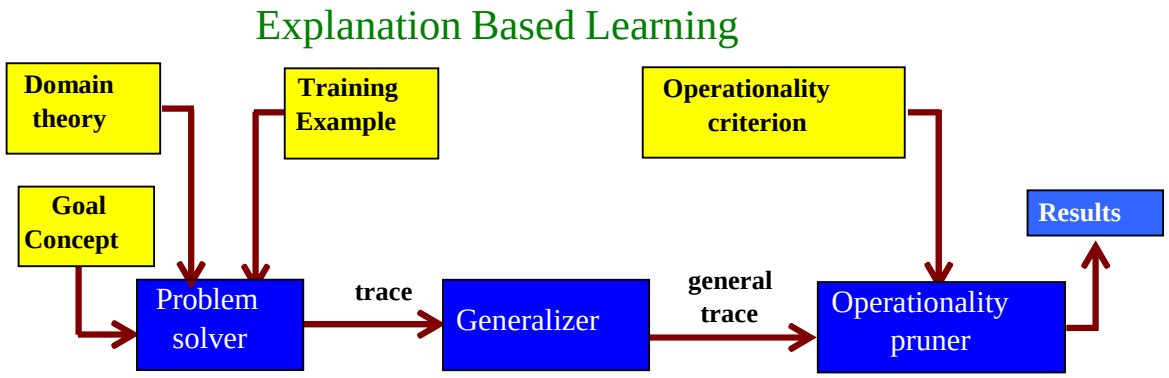
The overall architecture of the EBL learning method.



39

EBL System Schematic

The schematic below shows explanation based learning.



- **Input to EBL :**

The blocks color yellow are external to EBL.

They are the 4 different kinds of input that EBL algorithm accepts.

1. **Training example** - Situation description, facts .

   An example is a set of facts describing an instance of the goal concept.

2. **Goal concept** - Some predicate calculus statements .

   A high level description of concept that the program is supposed to learn; e.g., the definition of an object **"Cup"** in function-structure involves functional properties **(lift-able, stable, open-vessel . . .)** rather than structural features (light, part of, pointing . . .).

3. **Domain theory** - Inference rules .

   It represents facts and rules that constitute what the learner knows. The facts describe an instance of the goal concept and the rules describe relationships between objects and actions in a domain; e.g., the cup domain includes **facts: concavities, bases, and lugs**, as well as **rules: about lift ability, stability and what makes an open vessel.**

4. **Operational criterion** - description of concepts .

   A description in an appropriate form of the final concept.

   A predicate from domain theory over concept definition, specifying the form in which the learned concept definition must be expressed.

40

- **Description of EBL algorithm**

Given the four inputs just described, the task is to construct an explanation in terms of the domain theory and then determine a set of sufficient conditions under which the explanationholds.

Thus, EBL algorithm consisting of two stages:

1. **Explain:** Construct an explanation in terms of the domain theory that shows how the training example satisfies the goal concept definition. This explanation must be constructed so that each branch of the explanation structure terminates in an expression that satisfies the operationality criterion.
2. **Generalize:** Determine a set of sufficient conditions under which the explanation holds, stated in terms that satisfy the operationality criterion. This is accomplished by regressing (back propagating) the goal concept through the explanation structure. The conjunction of the resulting expressions constitutes the desired concept definition.

Algorithm : The step involved are

1. Given an example, an "explanation" is first constructed by applying a problem solver to relate the example to some general domain theory.
   The result of this operation is a trace or "proof" of the example with respect to the theory.
2. Next, generalizes the explanation using the method of goal regression. This involves traversing the tree from top to bottom, replacing constants by variables, but just those constants that are not embedded in the rules or facts used to create the proof. The result of this operation is the generalized proof tree.
3. The tree is then pruned by removing irrelevant leaf nodes until no operational predicates appear at internal nodes. The result of this operation is that explanation structure satisfies the operationality criterion that applies to other examples too.
4. Finally, the operational rules are extracted from the general explanation. The definition of the target concept is conjunction of the remaining leaf nodes.

41

Example : "CUP" Generalization problem

The EBL methodology described before is applied to concept "CUP".

The training example, domain theory, goal concept, operationality criterion, and a proof tree generalization using a goal regression technique are presented. The logical operators usedare

^ and; → implies; if .. then; ↔ if and only if; iff

Goal Concept

lift-able(x) ^ stable(x) ^ open-vessel(x) ↔ cup(x)

Training example

colour(Obj23, Blue) ^ has-part(Obj23, Handle16) ^ has-part(Obj23, Bottom19) ^ owner(Obj23, Ralph) ^ has-part(Obj23, Concavity12) ^ is(Obj23, Light) ^ is(Ralph, Male) ^ isa(Handle16,Handle) ^ isa(Bottom19, Bottom)^
is(Bottom19, Flat) ^ isa(Concavity12, Concavity) ^
is(Concavity12, Upward-Pointing)

Domain theory

has-part(x,y) ^ isa(y,Concavity) ^ is(y, Upward-Pointing) → is(x, Light) ^ has-part(x,y)^ isa(y,Handle)

open-vessel(x) → liftable(x)

has-part(x,y) ^ isa(y, Bottom)^ is(y,Flat) → stable(x)

Operationality Criterion

The concept definition is expressed in terms of structural features.

The first stage of the EBL process is to show why the training example is an example of a cup. This represents aproof.

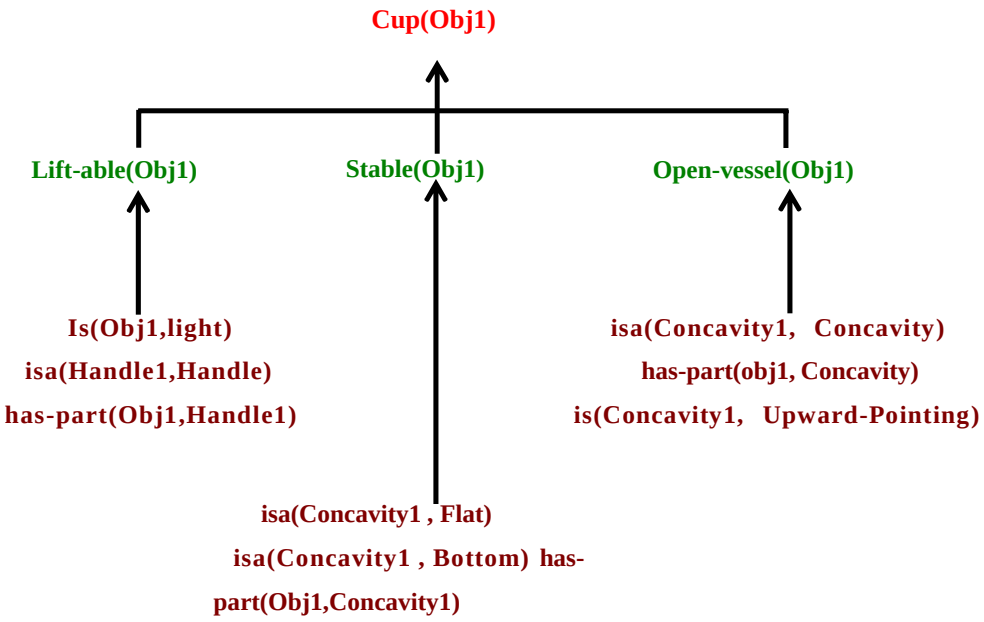
This proof is expressed only in terms of the operationality criterion and irrelevant details discarded relating to the Owner and Color.

Explanation structure of the cup.

42

Explanation Structure of the cup

The structure below represents the first stage of the EBL process. It proofs why the training example is an example of a cup.



Proof tree generalization using a goal regression technique.

The above proof is now generalized by using a goal regression technique. In this example replacing the constants with variables gives the required generalization of the cup.

has-part(x, y) U isa(y, Concavity) U is(y, Upward-Pointing) U has-part(x, z) U isa(z, Bottom) U is(z, Flat) U has-part(x,w) U isa(w, Handle) U is(x, Light)

43

5. Discovery

Simon (1966) first proposed the idea that we might explain scientific discovery in computational terms and automate the processes involved on a computer. Project **DENDRAL** (Feigenbaum 1971) demonstrated this by inferring structures of organic molecules from mass spectra, a problem previously solved only by experienced chemists.

Later, a knowledge based program called **AM** the Automated Mathematician (Lenat 1977) discovered many mathematical concepts.

After this, an equation discovery systems called **BACON** (Langley, 1981) discovered a wide variety of empirical laws such as the ideal gas law. The research continued during the 1980s and 1990s but reduced because the computational biology, bioinformatics and scientific data mining have convinced many researchers to focus on domain-specific methods. But need for research on general principles for scientific reasoning and discovery very much exists.

Discovery system AM relied strongly on **theory-driven** methods of discovery. BACON employed **data-driven** heuristics to direct its search for empirical laws.

These two discovery programs are illustrated in the next few slides.

44

Theory Driven Discovery

The Simon's theory driven science, means AI-modeling for theory building. It starts with an existing theory represented in some or all aspects in form of a symbolic model and one tries to transform the theory to a runable program. One important reason for modeling a theory is scientific discovery in the theory driven approach, this means the discovery of new theoretical conclusions, gaps, or inconsistencies.

Many computational systems have been developed for modeling different types of discoveries. The **Logic Theorist** (1956) was designed to prove theorems in logic when AI did not exist. Among the more recent systems, the Automated Mathematician **AM** (Lenat, 1979) is a good example in modeling mathematical discovery.

- **AM** (Automated Mathematician)
  AM is a heuristic driven program that discovers concepts in elementary mathematics and set theory. AM has 2 inputs:
  (a) description of some concepts of set theory: e.g. union, intersection;
  (b) information on how to perform mathematics. e.g. functions.

  AM have successively rediscovered concepts such as :
  (a) Integers , Natural numbers, Prime Numbers;
  (b) Addition, Multiplication, Factorization theorem ;
  (c) Maximally divisible numbers, e.g. 12 has six divisors 1, 2, 3, 4, 6, 12.

  [AM is described in the next slide.]

45

- **How does AM work ?**

  AM employs many general-purpose AI techniques.

  The system has around 115 basic elements such as sets, lists, elementary relations. The mathematical concepts are represented as frames. Around 250 heuristic rules are attached to slots in the concepts. The rules present hints as how to employ functions, create new concepts, generalization etc. about activities that might lead to interesting discoveries.

  The system operates from an agenda of tasks. It selects the most interesting task as determined by a set of over 50 heuristics. It then performs all heuristics it can find which should help in executing it. The heuristics represented as operators are used to generalize, to specialize or to combine the elementary concepts or relations to make more complex ones. Heuristics can fill in concept slots, check the content of the slots, create new concepts, modify task agenda, interestingness levels, etc. Because it selects the most interesting task to perform at all times, AM is performing the best-first search in a space of mathematical concepts. However, its numerous heuristics (over 200) guide its search very effectively, limiting the number of concepts it creates and improving their mathematical quality.

Data Driven Discovery

Data driven science, in contrast to theory driven, starts with empirical data or the input-output behavior of the real system without an explicitly given theory. The modeler tries to write a computer program which generates the empirical data or input-output behavior of the system. Typically, models are produced in a generate-and-test-procedure. Generate-and-test means writing program code which tries to model the i-o-behavior of the real system first approximately and then improve as long as the i-o-behavior does not correspond to the real system. A family of such discovery models are known as **BACON programs**.

- **BACON System**

Equation discovery is the area of machine learning that develops methods for automated discovery of quantitative laws, expressed in the form of equations, in collections of measured data.

BACON is pioneer among equation discovery systems. BACON is a family of algorithms for discovering scientific laws fromdata.

BACON.1 discovers simple numeric laws.

BACON.3 is a knowledge based system, has discovered simple empirical laws like physicists and shown its generality by rediscovering the Ideal gas law, Kepler's third law, Ohm's law and more.

The next few slides shows how BACON1 rediscovers Kepler's third Law and BACON3 rediscovers Ideal Gas Law.

47

- **BACON.1 :** Discovers simple numeric laws.
  Given a set of observed values about two variables **X** and **Y**, BACON.1 finds a function **Y = f(X)** using four heuristics:
  - Heuristic 1 : If **Y** has value **V** in several observed cases, make the hypothesis that **Y = V** in all cases.
  - Heuristic 2 : If **X** and **Y** are linearly related with slope **S** and intercept **I** in several observed cases, then make the hypothesis that **Y = S · X + I** is in all cases.
  - Heuristic 3 : If **X** increases as **Y** decreases, and **X** and **Y** are not linearly related define a new term **T** as the product of **X** and **Y** ie., **T = X · Y**
  - Heuristic 4 : If **X** increases as **Y** increases, and **X** and **Y** are not linearly related, define a new term **T** as the division of **X** by **Y** ie., **T = X / Y**

  Note : BACON1 iteratively applies these 4 heuristics until a scientific law is discovered. Heuristics 1 and 2 detect linear relationships. Heuristics 3 and 4 detect simple non-linear relationships. Heuristics 1 and 2 produce scientific laws.
  Heuristics 3 and 4 are intermediate steps.

  Example : Rediscovering Kepler's third Law

  Kepler's third Law is stated below. Assume the law is not discovered or known. "The square of the orbital period **T** is proportional to the cube of the mean distance a from the Sun." **ie., $T^2 = k\ a^3$,** **k** is constant number, is same for all planets. If we measure **T** in years and all distances in "astronomical units AUs" with 1 AU the mean distance between the Earth and the Sun, then if **a = 1 AU**, **T** is one year, and **k** with these units just equals 1, i.e. $T^2=a^3$.

  - **Input :** Planets, Distance from Sun (**D**) , orbit time Period (**P**)

Planet	D	P
Mercury	0.382	0.241
Venus	0.724	0.616
Earth	1.0	1.0
Mars	1.524	1.881
Jupiter	5.199	11.855
Saturn	9.539	29.459

[continued in next slide]

48

■

Try heuristic 1:	not applicable,	neither **D** nor **P** is constant
Try heuristic 2:	not applicable,	no linear relationship
Try heuristic 3:	not applicable,	**D** increasing **P** not decreasing
Try heuristic 4:	**applicable,**	**D** increases as **P** increase, so add new variable **D/P** to the data set.

Adding new variable **D/P** to the data set:

Planet	D	P	D/P
Mercury	**0.382**	**0.241**	**1.607**
Venus	**0.724**	**0.616**	**1.175**
Earth	**1.0**	**1.0**	**1.0**
Mars	**1.524**	**1.881**	**0.810**
Jupiter	**5.199**	**11.855**	**0.439**
Saturn	**9.539**	**29.459**	**0.324**

■ **Apply heuristics 1 to 4 :** (Iteration 2)

Try heuristic 1**:** not applicable, **D/P** is not constant

Try heuristic 2**:** not applicable, no linear relationship between **D/P** and **D** or **P**

Try heuristic 3**: applicable,** **D/P** decreases as **D** increases and as **P** increases, so the system could add two new variables: **D × (D/P) = D^2/P or P × (D/P) = D** but **D** already exists, so add new variable **D^2/P**

Adding new variable **D^2/P** to the data set:

Planet	D	P	D/P	D^2/P
Mercury	**0.382**	**0.241**	**1.607**	**0.622**
Venus	**0.724**	**0.616**	**1.175**	**0.851**
Earth	**1.0**	**1.0**	**1.0**	**1.0**
Mars	**1.524**	**1.881**	**0.810**	**1.234**
Jupiter	**5.199**	**11.855**	**0.439**	**2.280**
Saturn	**9.539**	**29.459**	**0.324**	**3.088**

49

- **Apply heuristics 1 to 4 :** (Iteration 3)

Try heuristic 1**:** not applicable, D^2/P is not constant

Try heuristic 2**:** not applicable, D^2/P is not linearly related with any other variable

Try heuristic 3**: applicable,** D^2/P decreases as D/P increases, so add the new variable: $(D^2/P) \times (D/P) = D^3/P^2$

Adding new variable D^3/P^2 to the data set:

Planet	D	P	D/P	D^2/P	D^3/P^2
Mercury	0.382	0.241	1.607	0.622	1.0
Venus	0.724	0.616	1.175	0.851	1.0
Earth	1.0	1.0	1.0	1.0	1.0
Mars	1.524	1.881	0.810	1.234	1.0
Jupiter	5.199	11.855	0.439	2.280	1.0
Saturn	9.539	29.459	0.324	3.088	1.0

- **Apply heuristics 1 to 4 :** (Iteration 4)

Try heuristic1**: applicable,,** D^3/P^2 isconstant

- **Conclusion :** D^3/P^2

This is Kepler'sthird law. (took about 20 years to discover it !)

- **A limitation of BACON.1**

It works only for target equation relating at most two variable.

UNIT-V

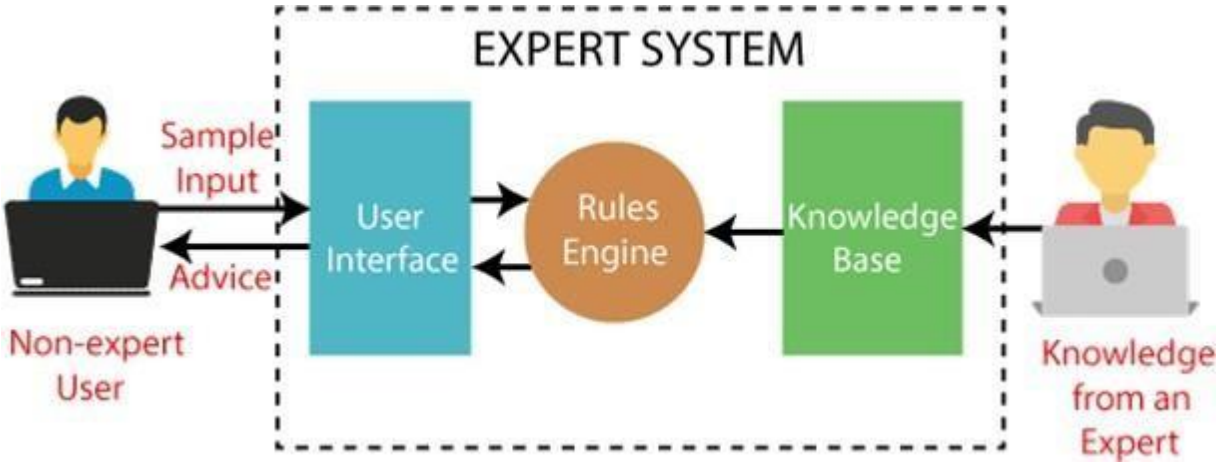
Expert System

An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert. It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.

The expert system is a part of AI, and the first ES was developed in the year 1970, which was the first successful approach of artificial intelligence. It solves the most complex issue as an expert by extracting the knowledge stored in its knowledge base. The system helps in decision making for compsex problems using **both facts and heuristics like a human expert**. It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain. These systems are designed for a specific domain, such as **medicine, science,** etc.

The performance of an expert system is based on the expert's knowledge stored in its knowledge base. The more knowledge stored in the KB, the more that system improves its performance. One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.

Below is the block diagram that represents the working of an expert system:



Note: It is important to remember that an expert system is not used to replace the human experts; instead, it is used to assist the human in making a complex decision. These systems do not have human capabilities of thinking and work on the basis of the knowledge base of the particular domain.

Below are some popular examples of the Expert System:

- **DENDRAL:** It was an artificial intelligence project that was made as a chemical analysis expert system. It was used in organic chemistry to detect unknown organic molecules with the help of their mass spectra and knowledge base of chemistry.
- **MYCIN:** It was one of the earliest backward chaining expert systems that was designed to find the bacteria causing infections like bacteraemia and meningitis. It was also used for the recommendation of antibiotics and the diagnosis of blood clotting diseases.
- **PXDES:** It is an expert system that is used to determine the type and level of lung cancer. To determine the disease, it takes a picture from the upper body, which looks like the shadow. This shadow identifies the type and degree of harm.
- **CaDeT:** The CaDet expert system is a diagnostic support system that can detect cancer at early stages.

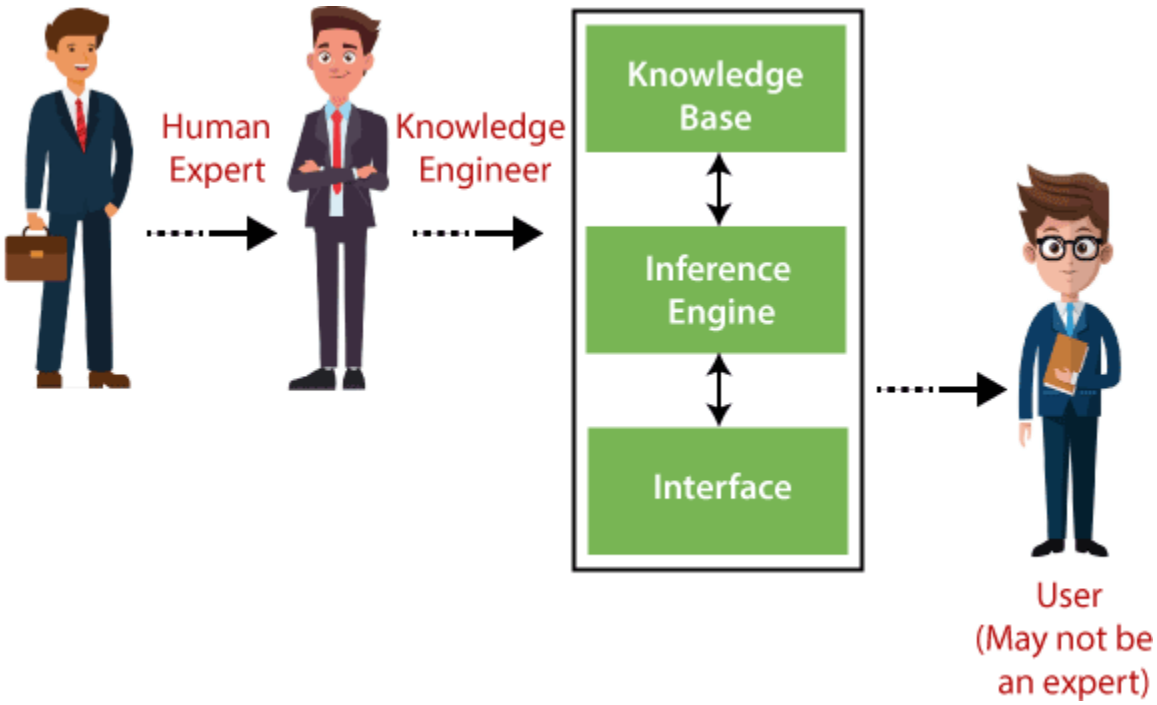
Characteristics of Expert System

- **High Performance:** The expert system provides high performance for solving any type of complex problem of a specific domain with high efficiency and accuracy.
- **Understandable:** It responds in a way that can be easily understandable by the user. It can take input in human language and provides the output in the same way.
- **Reliable:** It is much reliable for generating an efficient and accurate output.
- **Highly responsive:** ES provides the result for any complex query within a very short period of time.

Components of Expert System

An expert system mainly consists of three components:

- **User Interface**
- **Inference Engine**
- **Knowledge Base**



1. User Interface

With the help of a user interface, the expert system interacts with the user, takes queries as an input in a readable format, and passes it to the inference engine. After getting the response from the inference engine, it displays the output to the user. In other words, **it is an interface that helps a non-expert user to communicate with the expert system to find a solution**.

2. Inference Engine(Rules of Engine)

- The inference engine is known as the brain of the expert system as it is the main processing unit of the system. It applies inference rules to the knowledge base to derive a conclusion or deduce new information. It helps in deriving an error-free solution of queries asked by the user.
- With the help of an inference engine, the system extracts the knowledge from the knowledge base.
- There are two types of inference engine:
- **Deterministic Inference engine:** The conclusions drawn from this type of inference engine are assumed to be true. It is based on **facts** and **rules**.
- **Probabilistic Inference engine:** This type of inference engine contains uncertainty in conclusions, and based on the probability.

Inference engine uses the below modes to derive the solutions:

- **Forward Chaining:** It starts from the known facts and rules, and applies the inference rules to add their conclusion to the known facts.
- **Backward Chaining:** It is a backward reasoning method that starts from the goal and works backward to prove the known facts.

3. Knowledge Base

- The knowledgebase is a type of storage that stores knowledge acquired from the different experts of the particular domain. It is considered as big storage of knowledge. The more the knowledge base, the more precise will be the Expert System.
- It is similar to a database that contains information and rules of a particular domain or subject.
- One can also view the knowledge base as collections of objects and their attributes. Such as a Lion is an object and its attributes are it is a mammal, it is not a domestic animal, etc.

Components of Knowledge Base

- **Factual Knowledge:** The knowledge which is based on facts and accepted by knowledge engineers comes under factual knowledge.
- **Heuristic Knowledge:** This knowledge is based on practice, the ability to guess, evaluation, and experiences.

Knowledge Representation: It is used to formalize the knowledge stored in the knowledge base using the If-else rules.

Knowledge Acquisitions: It is the process of extracting, organizing, and structuring the domain knowledge, specifying the rules to acquire the knowledge from various experts, and store that knowledge into the knowledge base.

Development of Expert System

Here, we will explain the working of an expert system by taking an example of MYCIN ES. Below are some steps to build an MYCIN:

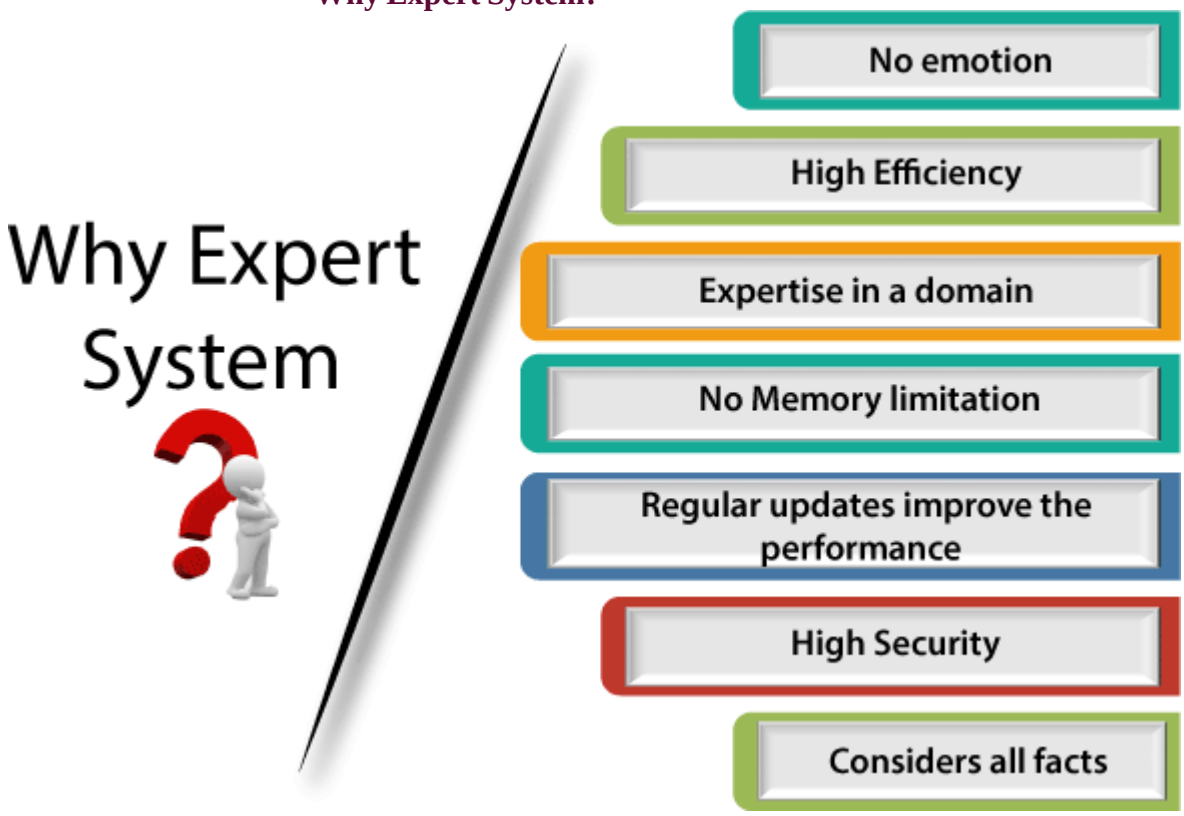
- Firstly, ES should be fed with expert knowledge. In the case of MYCIN, human experts specialized in the medical field of bacterial infection, provide information about the causes, symptoms, and other knowledge in that domain.
- The KB of the MYCIN is updated successfully. In order to test it, the doctor provides a new problem to it. The problem is to identify the presence of the bacteria by inputting the details of a patient, including the symptoms, current condition, and medical history.
- The ES will need a questionnaire to be filled by the patient to know the general information about the patient, such as gender, age, etc.
- Now the system has collected all the information, so it will find the solution for the problem by applying if-then rules using the inference engine and using the facts stored within the KB.
- In the end, it will provide a response to the patient by using the user interface.

Participants in the development of Expert System

There are three primary participants in the building of Expert System:

1. **Expert:** The success of an ES much depends on the knowledge provided by human experts. These experts are those persons who are specialized in that specific domain.
2. **Knowledge Engineer:** Knowledge engineer is the person who gathers the knowledge from the domain experts and then codifies that knowledge to the system according to the formalism.
3. **End-User:** This is a particular person or a group of people who may not be experts, and working on the expert system needs the solution or advice for his queries, which are complex.

Why Expert System?



Before using any technology, we must have an idea about why to use that technology and hence the same for the ES. Although we have human experts in every field, then what is the need to develop a computer-based system. So below are the points that are describing the need of the ES:

1. **No memory Limitations:** It can store as much data as required and can memorize it at the time of its application. But for human experts, there are some limitations to memorize all things at every time.
2. **High Efficiency:** If the knowledge base is updated with the correct knowledge, then it provides a highly efficient output, which may not be possible for a human.
3. **Expertise in a domain:** There are lots of human experts in each domain, and they all have different skills, different experiences, and different skills, so it is not easy to get a final output for the query. But if we put the knowledge gained from human experts into the expert system, then it provides an efficient output by mixing all the facts and knowledge
4. **Not affected by emotions:** These systems are not affected by human emotions such as fatigue, anger, depression, anxiety, etc.. Hence the performance remains constant.
5. **High security:** These systems provide high security to resolve any query.
6. **Considers all the facts:** To respond to any query, it checks and considers all the available facts and provides the result accordingly. But it is possible that a human expert may not consider some facts due to any reason.
7. **Regular updates improve the performance:** If there is an issue in the result provided by the expert systems, we can improve the performance of the system by updating the knowledge base.

Capabilities of the Expert System

Below are some capabilities of an Expert System:

- **Advising:** It is capable of advising the human being for the query of any domain from the particular ES.

- **Provide decision-making capabilities:** It provides the capability of decision making in any domain, such as for making any financial decision, decisions in medical science, etc.
- **Demonstrate a device:** It is capable of demonstrating any new products such as its features, specifications, how to use that product, etc.
- **Problem-solving:** It has problem-solving capabilities.
- **Explaining a problem:** It is also capable of providing a detailed description of an input problem.
- **Interpreting the input:** It is capable of interpreting the input given by the user.
- **Predicting results:** It can be used for the prediction of a result.
- **Diagnosis:** An ES designed for the medical field is capable of diagnosing a disease without using multiple components as it already contains various inbuilt medical tools.

Advantages of Expert System

- These systems are highly reproducible.
- They can be used for risky places where the human presence is not safe.
- Error possibilities are less if the KB contains correct knowledge.
- The performance of these systems remains steady as it is not affected by emotions, tension, or fatigue.
- They provide a very high speed to respond to a particular query.

Limitations of Expert System

- The response of the expert system may get wrong if the knowledge base contains the wrong information.
- Like a human being, it cannot produce a creative output for different scenarios.

- Its maintenance and development costs are very high.
- Knowledge acquisition for designing is much difficult.
- For each domain, we require a specific ES, which is one of the big limitations.
- It cannot learn from itself and hence requires manual updates.

Applications of Expert System

- **In designing and manufacturing domain**

  It can be broadly used for designing and manufacturing physical devices such as camera lenses and automobiles.

- **In the knowledge domain**

  These systems are primarily used for publishing the relevant knowledge to the users. The two popular ES used for this domain is an advisor and a tax advisor.

- **In the finance domain**

  In the finance industries, it is used to detect any type of possible fraud, suspicious activity, and advise bankers that if they should provide loans for business or not.

- **In the diagnosis and troubleshooting of devices**

  In medical diagnosis, the ES system is used, and it was the first area where these systems were used.

- **Planning and Scheduling**

  The expert systems can also be used for planning and scheduling some particular tasks for achieving the goal of that task.

What is Prolog?

- Prolog stands for **Programming in logic.** It is used in artificial intelligence programming.
- Prolog is a **declarative** programming language.

  For example: While implementing the solution for a given problem, instead of specifying the ways

to achieve a certain goal in a specific situation, user needs to specify about the situation (rules and facts) and the goal (query). After these stages, Prolog interpreter derives the solution.

- Prolog is useful in AI, NLP, databases but useless in other areas such as graphics or numerical algorithms.

Prolog facts

- A fact is something that seems to be true.

For example: It's raining.

- In Prolog, facts are used to form the statements. Facts consist of a specific item or relation between two or more items.

How to convert English to prolog facts using facts and rules?

It is very simple to convert English sentence into Prolog facts. Some examples are explained in the following table.

English Statements	Prolog Facts
Dog is barking	barking(dog)
Jaya likes food if it is delicious.	likes(Jaya, Food):-delicious(Food)

In the above table, the statement '**Dog is barking**' is a **fact,** while the statement '**Jaya likes food if it is delicious**' is called **rule.** In this statement, variable like 'Food' has a first letter in capital, because its value came from previous fact. The symbol '**:-**' is used to denote that "Jaya likes delicious food".

Arithmetic Operations in Prolog

- Prolog provides the facility for arithmetic operations.
- As per the requirement of the user, arithmetic operations can be divided into some special purpose integer predicates and a series of general predicates for integer, floating point and rational arithmetic.
- The general arithmetic predicates are handled by the expressions.
- An expression is either a function or a simple number.
- Prolog arithmetic is slightly different than other programming languages.

For example:

```
?- X is 2 + 1.
X = 3 ?
yes
```

In the above example, **'is'** is used as a special predefined operator.

The basic arithmetic operators are given in the following table:

Sr. No	Operator	Explanation
1	X+Y	The sum of 'X' and 'Y'

2	X-Y	the difference of 'X' and 'Y'
3	X*Y	The product of 'X' and 'Y'
4	X/Y	The quotient of 'X' and 'Y'
5	X^Y	'X' to the power of 'Y'
6	-X	Negation of 'X'
7	abs(X)	Absolute value of 'X'
8	sqrt(X)	The square root of X
9	sin(X)	The sine of X
10	cos(X)	The cos of X

Operator precedence

- If there is more than one operator in the arithmetic expression such as A-B*C+D, then the prolog decides an order in which the operator should be applied.
- Prolog gives numerical value for each operator, operators with high precedence like **'*'** and **'/'** are applied before operators with relatively low precedence values like **'+'** and **'-'**.
- Operator with same precedence value ('*' or '/') and ('+' or '-') should be applied from left to right.
- So, the expression A-B*C+D can be written as **A-(B*C)+D**

Matching and Unification in Prolog

Definition: The two terms are said to be matched, if they are equal or if they consist of variables representing the resulting equal terms.

Prolog matches expressions in structural way. So,

?- 3 + 2= 5

no

Note: In prolog '=' means matches with.

Consider the following example,?- X + 3 = 2 * Y

But the following expressions will match because they have same structure.

Expression 1:

?- X + Y = 2 + 3

X = 2

Y = 3

Expression 2:

?- 2 + Y = X + 3

X = 2

Y = 3

Prolog Lists:

- Lists are the finite sequence of elements.
- Prolog uses […] to build a list.
- The notation [X|Y] represents that the first element is X and second element is Y (X is head and Y is tail).
- Prolog has some special notation for lists:
  I) [a] [honda, maruti, renault]
  ii) [a,b,c] [pen, pencil, notebook]
  iii) [] represents the empty list.

Example 1: Pattern Matching in Lists

```
?- [a,b] = [a,X]
X = b
but:
?- [a,b] = [X]
no
```

Example 2:

Consider the following lists:

[a, b, c, d, e, f, g]
[apple, pear, bananas, breadfruit]
[ ] this is an empty list

Now, consider some comparisons of lists:

[a,b,c] matches with [Head|Tail] resulting in Head=a and Tail=[b,c]
[a] matches with [H|T] resulting in H=a and T=[]
[a,b,c] matches with [a|T] resulting in T=[b,c]
[a,b,c] doesn't match with [b|T]
[] doesn't match with [H|T]
[] match with []. Hence, two empty lists get matched with each other.

Backtracking in Prolog

What is backtracking?

If a person reaches a point where a goal cannot be matched, so he can come back (backtrack) to the last spot, where the choice of matching a specific fact or rule was formed. If this process fails, a person again goes to the nearest previous place where a choice was made. So, this procedure is followed until the goal is achieved.

Example:

Consider the following facts,

```
bird(type (sparrow) name (steve)))
```

bird(type (penguin) name (sweety)))
bird(type (penguin)name (jones)))

consider the following query:

?- bird(type (penguin)name(X)

So, prolog will try to match the first query, but this query will not match because sparrow doesn't match with penguin. Then, it will try to find next query to match the fact and succeed with X = sweety. Later, if the query or subgoals are failed, it will go to the saved option and look for more solutions. For example: X = jones

Genetic Learning

Genetic algorithms are the parts of evolutionary computing and are implemented as a computer simulation.

What is biological evolution?

- Every organism has a set of rules which describes, how that organism is built, and encoded in the genes of an organism.
- The chromosomes contain hundreds to thousands of genes.
- Features of the organisms depend on the genes and they have several settings. For example: hair color gene may be brown or black.
- The genes and their settings are referred to as a genotype of an organism.
- When two organisms mate, they share their genes, so that the offspring gets half genes from one parent and half genes from the other parent. Hence, this process is called crossover.
- It is possible that gene is mutated in the organism as a completely new feature.
- The genetic algorithm follows the process of nature to solve the problems such as selection, crossover, mutation, and acceptance for evolution.

Following are the steps of genetic algorithm:

Step 1: Generate random population of 'n' chromosomes.
Step 2: Evaluate the fitness f(x) of each chromosome 'x' present in the population.
Step 3: Follow these steps to create a new population:
a) Selection: From population, select two parents fitness according to their fitness.
b) Crossover: Perform crossover using properties of crossover to form a new offspring.
c) Mutation: Mutate the offspring at each position in chromosome.
d) Accept: Place a new offspring in the new population.
Step 4: Replace: Use new generated population to run an algorithm.
Step 5: Test and Stop, if end condition is satisfied and return the best solution in the current

-

population.

Step 6: Go to step 2.

Knowledge Acquisition

Knowledge acquisition refers to the process of extracting, structuring, and organizing domain knowledge from domain experts into a program. A **knowledge engineer** is an expert in AI language and knowledge representation who investigates a particular problem domain, determines important concepts, and creates correct and efficient representations of the objects and relations in the domain. Capturing domain knowledge of a problem domain is the first step in building an expert system. In general, the knowledge acquisition process through a knowledge engineer can be divided into four phases:

1. **Planning:** The goal is to understand the problem domain, identify domain experts, analyze various knowledge acquisition techniques, and design proper procedures.
2. **Knowledge extraction:** The goal is to extract knowledge from experts by applying various knowledge acquisition techniques.
3. **Knowledge analysis:** The outputs from the knowledge extraction phase, such as concepts and heuristics, are analyzed and represented in formal forms, including heuristic rules, frames, objects and relations, semantic networks, classification schemes, neural networks, and fuzzy logic sets. These representations are used in implementing a prototype expert system.
4. **Knowledge verification:** The prototype expert system containing the formal representation of the heuristics and concepts is verified by the experts. If the knowledge base is incomplete or insufficient to solve the problem, alternative knowledge acquisition techniques may be applied, and additional knowledge acquisition process may be conducted.

Many knowledge acquisition techniques and tools have been developed with various strengths and limitations. Commonly used techniques include interviewing, protocol analysis, repertory grid analysis, and observation.

Interviewing is a technique used for eliciting knowledge from domain experts and design requirements. The basic form involves free-form or unstructured question–answer sessions between the domain expert and the knowledge engineer. The major problem of this approach results from the inability of domain experts to explicitly describe their reasoning process and the biases involved in human reasoning. A more effective form of interviewing is called **structured interviewing,** which is goal-oriented and directed by a series of clearly stated goals. Here, experts either fill out a set of carefully designed questionnaire cards or answer questions carefully designed based on an established domain model of the problem-solving process. This technique reduces the interpretation problem inherent in the unstructured interviewing as well as the distortion caused by domain expert subjectivity.

As an example, let's look at the interviewing process used in constructing GTE's COMPASS system (Prerau, 1990). COMPASS is an expert system that examines error messages derived from a telephone switch's self-test routines and suggests running of additional tests or replacing a particular component. The interviewing process in building COMPASS has an elicit–document–test cycle as follows:

1. Elicit knowledge from an expert.
2. Document the elicited knowledge in rules and procedures.
3. Test the new knowledge using a set of data:
   (a) Have the expert analyze a new set of data.
   (b) Analyze the same set of data using the documented knowledge.
   (c) Compare the two results.
   (d) If the results differ, find the rules or procedures that lead to the discrepancy and return to step 1 to elicit more knowledge to resolve the problem.

Protocol analysis is another technique of data analysis originated in clinical psychology. In this approach, an expert is asked to talk about his or her thinking process while solving a given problem. The difference from interviewing is that experts find it much easier to talk about specific problem instances than to talk in abstract terms. The problem-solving process being described is then analyzed to produce a structured model of the expert's knowledge, including objects of significance, important attributes of the objects, relationships among the objects, and inferences drawn from the relationships. The advantage of protocol analysis is the accurate description of the specific actions and rationales as the expert solves the problem.

Repertory grid analysis investigates the expert's mental model of the problem domain. First, the expert is asked to identify the objects in the problem domain and the traits that differentiate them. Then, a rating grid is formed by rating the objects according to the traits.

Observation involves observing how an expert solves a problem. It enables the expert to continuously work on a problem without being interrupted while the knowledge is obtained. A major limitation of this technique is that the underlying reasoning process of an expert may not be revealed in his or her actions.

Knowledge acquisition is a difficult and time-consuming task that often becomes the bottleneck in expert system development (Hayes-Roth *et al*., 1983). Various techniques have been developed to automate the process by using domaintailored environments containing well-defined domain knowledge and specific problem-solving methods (Rothenfluh *et al*., 1996). For example, OPAL is a program that expedites knowledge elicitation for the expert system ONCOCIN (Shortliffe *et al.*, 1981) that constructs treatment plans for cancer patients. OPAL uses a model of the cancer domain to acquire knowledge directly from an expert. OPAL's domain model has four main aspects:

entities and relationships, domain actions, domain predicate, and **procedural knowledge.**

Based on its domain knowledge, OPAL can acquire more knowledge from a human expert and translate it into executable code, such as production rules and finite state tables. Following OPAL, more general-purpose systems called PROTEGE and PROTEGE-II were developed (Musen, 1989). PROTEGE-II contains tools for creating domain ontology and generating OPAL-like knowledge acquisition programs for particular applications PROTEGE-II is a general tool developed by abstraction from a successful application, similar to the process from MYCIN to EMYCIN.
Another example of automated knowledge acquisition is the SALT system (Marcus and McDermott, 1989) associated with an expert system called Vertical Transportation (VT) for designing custom-design elevator systems. SALT assumes a propose-and-revise strategy in the knowledge acquisition process. Domain knowledge is seen as performing one of three roles: (1) proposing an extension to the current design, (2) identifying constraints upon design extension, and (3) repairing constraint violation. SALT automatically acquires these kinds of knowledge by interacting with an expert and then compiles the knowledge into production rules to generate a domain-specific knowledge base. SALT retains the original knowledge in a declarative form as a dependency network, which can be updated and recompiled as necessary.
To build a knowledge base, the knowledge can be either captured through knowledge engineers or be generated automatically by machine learning techniques. For example, rules in rule-based expert systems may be obtained through a knowledge acquisition process involving domain experts and knowledge engineers or may be generated automatically from examples using decision-tree learning algorithms. Casebased reasoning is another example of automated knowledge extraction in which the expert system searches its collection of past cases, finds the ones that are similar to the new problem, and applies the corresponding solutions to the new one. The whole process is fully automatic. An expert system of this type can be built quickly and maintained easily by adding and deleting cases. Automatic knowledge generation is especially good when a large set of examples exist or when no domain expert exists.
In addition to generating knowledge automatically, machine learning methods have also been used to improve the performance of the inference engines by learning the importance of individual rules and better control in reasoning.

UNIT WISE QUESTION BANK

UNIT 1			Bloom's Taxonomy Level
S. No.	Questions	COs	
1.	What is an AI? Write applications and goals of an AI	1	Understand
2.	Explain in details about agents its environment?	1	Understand
3.	Write about DFS and BFS in uniformed search techniques?	1	Apply
4.	Explain in details about heuristic search techniques?	1	Understand
5	Discuss about constrained satisfaction	1	Apply

UNIT II			Bloom's Taxonomy Level
S. No.	Questions	COs	
1.	How to construct a search trees explain in details	1	Understand
2.	Discuss in details about A* search ?	1	Understand
3.	Write about alpha beta pruning ?	1	Apply
4.	Explain in details about proportional logic ?	1	Understand
5	Discuss about Forward chaining and backward chaining	1	Apply

UNIT III			Bloom's Taxonomy Level
S. No.	Questions	COs	
1.		1	Understand
2.	Discuss in details about knowledge representation Issues?	1	Understand
3.	Write about types of knowledge?	1	Apply
4.	Write the differences between deductive and inductive reasoning?	1	Understand
5	Discuss Bayes' Rule Representing Knowledge in an Uncertain Domain	1	Apply

UNIT IV			Bloom's Taxonomy Level
S. No.	Questions	COs	
1.	What is Learning explain forms of learning?	1	Understand
2.	Write Winston's learning program?	1	Understand
3.	Discuss learning by taking advise with examples?	1	Apply
4.	What is decision Tree? Explain with example problem?	1	Understand

UNIT V			Bloom's Taxonomy Level
S. No.	**Questions**	**COs**	
1.	What is expert system explain role of expert system in AI?	1	Understand
2.	Draw the block diagram of Expert system and explain?	1	Analyze
3.	Write short note on shell and explanation in Expert system?	1	Apply
4.	Discuss about knowledge Acquisition in Expert system?	1	Understand