CHAPTER 1

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

1.1 What is AI?

Artificial Intelligence (AI) is usually defined as the science of making computers do things that require intelligence when done by humans. It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is the branch of computer science concerned with making computers behave like humans.

AI is thus the simulation of human intelligence processes by machines, especially computer systems. These processes include learning (the acquisition of information and rules for using the information), reasoning (using the rules to reach approximate or definite conclusions), and self-correction.

Artificial intelligence includes

- Games playing: programming computers to play games such as chess and checkers.
- Expert systems : programming computers to make decisions in real-life situations (for example, some expert systems help doctors diagnose diseases based on symptoms).
- Natural language : programming computers to understand natural human languages.
- Neural networks : Systems that simulate intelligence by attempting to reproduce the types of physical connections that occur in animal brains.
- Robotics : programming computers to *see* and *hear* and react to other sensory stimuli.

The term *intelligence* covers many cognitive skills, including the ability to solve problems, learn, and understand language; AI addresses all of these. Intelligence therefore is the ability to comprehend; to understand and profit from experience. In other words, iintelligence is the ability to adapt one's behavior to fit new circumstances. Research in AI has focused chiefly on the following components of intelligence: learning, reasoning, problem-solving, perception, and language-understanding.

In summary, AI is a branch of computer science that concerned with the study and creation of computer system that exhibit human like intelligence that

- Can learn new concept,
- Understand natural language and
- Can perform other types of tasks which require human type intelligence.

1.2 Applications of AI

Nowadays, machines are being induced intelligent to perform the tasks using the artificial intelligent. Some particular applications of AI include expert systems, speech recognition, and machine vision and many more. Some of them are described below:

Game playing

Machines can play master level chess. There is some AI in them, but they play well against people mainly through brute force computation--looking at hundreds of thousands of positions. To beat a world champion by brute force and known reliable heuristics requires being able to look at 200 million positions per second.

Speech recognition

In the 1990s, computer speech recognition reached a practical level for limited purposes. Thus United Airlines has replaced its keyboard tree for flight information by a system using speech recognition of flight numbers and city names. It is quite convenient. On the other hand, while it is possible to instruct some computers using speech, most users have gone back to the keyboard and the mouse as still more convenient.

Understanding natural language

Just getting a sequence of words into a computer is not enough. Parsing sentences is not enough either. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

Computer vision

The world is composed of three-dimensional objects, but the inputs to the human eye and computers' TV cameras are two dimensional. Some useful programs can work solely in two dimensions, but full computer vision requires partial three-dimensional information that is not just a set of two-dimensional views. At present there are only limited ways of representing three-dimensional information directly, and they are not as good as what humans evidently use.

Expert systems

A ``knowledge engineer'' interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. When this turned out not to be so, there were many disappointing results. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. Namely, its ontology included bacteria, symptoms, and treatments and did not include patients, doctors, hospitals, death, recovery, and events occurring in time. Its interactions depended on a single patient being considered. Since the experts consulted by the knowledge engineers knew about patients, doctors, death, recovery, etc., it is clear that the knowledge engineers forced what the experts told them into a predetermined framework. In the present state of AI, this has to be true. The usefulness of current expert systems depends on their users having common sense.

Heuristic classification

One of the most feasible kinds of expert system given the present knowledge of AI is to put some information in one of a fixed set of categories using several sources of information. An example is advising whether to accept a proposed credit card purchase. Information is available about the owner of the credit card, his record of payment and also about the item he is buying and about the establishment from which he is buying it (e.g., about whether there have been previous credit card frauds at this establishment).

1.3 Brief History of AI

Date	Events
1950	• Alan Turing proposes the Turing Test as a measure of machine intelligence. • Claude Shannon published a detailed analysis of chess playing as search. • Isaac Asimov published his Three Laws of Robotics.

1951	The first working AI programs were written in 1951 to run on the Ferranti Mark 1 machine of the University of Manchester: a checkers-playing program written by Christopher Strachey and a chess-playing program written by Dietrich Prinz.
1952–1962	Arthur Samuel (IBM) wrote the first game-playing program,[28] for checkers (draughts), to achieve sufficient skill to challenge a respectable amateur. His first checkers-playing program was written in 1952, and in 1955 he created a version that learned to play.
1955	The first Dartmouth College summer AI conference is organized by John McCarthy, Marvin Minsky, Nathan Rochester of IBM and Claude Shannon.
1956	The name artificial intelligence is used for the first time as the topic of the second Dartmouth Conference, organized by John McCarthy The first demonstration of the Logic Theorist (LT) written by Allen Newell, J.C. Shaw and Herbert Simon (Carnegie Institute of Technology, now Carnegie Mellon University). This is often called the first AI program, though Samuel's checkers program also has a strong claim.
1957	The General Problem Solver (GPS) demonstrated by Newell, Shaw and Simon.
1958	John McCarthy (Massachusetts Institute of Technology or MIT) invented the Lisp programming language.
1959	John McCarthy and Marvin Minsky founded the MIT AI Lab.
1962	First industrial robot company, Unimation, was founded.
1965	Edward Feigenbaum initiated Dendral, a ten-year effort to develop software to deduce the molecular structure of organic compounds using scientific instrument data. It was the first expert system.
1972	• Prolog programming language developed by Alain

	Colmerauer.
1980	• First National Conference of the American Association for Artificial Intelligence (AAAI) held at Stanford.
Mid 1980s	• Neural Networks become widely used with the Backpropagation algorithm (first described by Paul Werbos in 1974).
2007	• Checkers is solved by a team of researchers at the University of Alberta.
2011	• AI received much public attention in February, 2011, with the Jeopardy exhibition match during which IBM's Watson soundly defeated the two greatest Jeopardy champions, Brad Rutter and Ken Jennings.

1.4 Some Terminologies

Fact

A **fact** (derived from the Latin *Factum*) is something that has really occurred or is actually the case. A fact can be referred to:

- Something that actually exists; reality; truth: *Your fears have no basis in fact.*
- Something known to exist or to have happened: *Space travel is now a fact.*
- A truth known by actual experience or observation; something known to be true: *Scientists gather facts about plant growth.*

The difference between fact and opinion is that a fact is something that is empirically true and can be supported by evidence, while an opinion is a belief that may or may not be backed up with some type of evidence. A fact can safely be defined as a true belief. Belief is synonymous with opinion

Knowledge

Knowledge is a familiarity with someone or something, which can include information, facts, descriptions, or skills acquired through experience or education. It can refer to the theoretical or practical understanding of a subject.

In normal conversation we use knowledge to mean:

- Knowing that (facts and information) and Knowing how (the ability to do something)
- The state or fact of knowing.
- Familiarity, awareness, or understanding gained through experience or study.
- The sum or range of what has been perceived, discovered, or learned.
- Learning; erudition: *teachers of great knowledge.*
- Specific information about something.

Belief

Belief is the psychological state in which an individual holds a proposition or premise to be true. It is something believed; an opinion or conviction: *a belief that the earth is flat.*

A belief is an assumed truth. It may refer to:

- Confidence in the truth or existence of something not immediately susceptible to rigorous proof: *a statement unworthy of belief.*
- Confidence; faith; trust: *a child's belief in his parents.*

Data

Data is a collection of facts, such as values or measurements. It can be numbers, words, measurements, observations or even just descriptions of things. Data can be qualitative or quantitative.

- **Qualitative data** is descriptive information (it *describes* something)
- **Quantitative data** is numerical information (numbers).

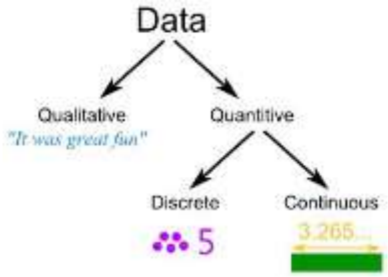


Figure 1.1: Types of knowledge

Quantitative data can also be Discrete or Continuous:

- **Discrete data** can only take certain values (like whole numbers)
- **Continuous data** can take any value (within a range)

Notes on: Data, Information and Knowledge

1. **Data** represents a fact or statement of event without relation to other things.

   For example: It is raining.

2. **Information** embodies the understanding of a relationship of some sort, possibly cause and effect.

For example: The temperature dropped 15 degrees and then it started raining.

3. **Knowledge** represents a pattern that connects and generally provides a high level of predictability as to what is described or what will happen next.

   For example: If the humidity is very high and the temperature drops substantially the atmospheres is often unlikely to be able to hold the moisture so it rains.

Hypothesis

A **hypothesis** is an explanation for a phenomenon which can be tested in some way which ideally either proves or disproves the hypothesis. For the duration of testing, the hypothesis is taken to be true, and the goal of the researcher is to rigorously test the terms of the hypothesis.

When someone formulates a hypothesis, he or she does so with the intention of testing it, and he or she should not know the outcome of potential tests before the hypothesis is made. When formulating a hypothesis, the ideals of the scientific method are often kept in mind, so the hypothesis is designed to be testable in a way which could be replicated by other people. It is also kept clear and simple, and the hypothesis relies on known information and reasoning.

A **hypothesis** is a proposed explanation for a phenomenon. A hypothesis may be proven correct or wrong, and must be capable of refutation. If it remains un-refuted by facts, it is said to be verified.

1.5 Knowledge

Knowledge can be defined as the body of facts and principles accumulated by human kind or the act, fact or state of knowing. In a broad sense, knowledge means having a familiarity with language concept, procedures, rules, idea, information, facts, skills acquired through experience or education. It can refer to both the theoretical or practical understanding of a subject.

1.5.1 Types of Knowledge

Knowledge can be classifies into the following categories:

a) Procedural knowledge
b) Declarative knowledge
c) Heuristic knowledge
d) Relational knowledge
e) Inheritable knowledge
f) Inferential knowledge

a) Procedural Knowledge

Procedural knowledge is the type of knowledge someone has and demonstrates through the procedure of doing something. For example, the steps used to solve algebraic equations.

Thus procedural knowledge (the knowledge of how, and especially how best, to perform some task), is related to the procedure to carry an action out. Knowledge about "how to do something" is procedural knowledge. It is instruction-oriented. It focuses on how to obtain a result.

Basic idea:

- It is the knowledge encoded in some procedures
  - Small programs that know how to do specific things, how to proceed.
  - *For example,* a parser in a natural language has the knowledge that a *noun phrase* may contain articles, adjectives and nouns. It is represented by calls to routines that know how to process articles, adjectives and nouns.

Advantages:

- *Heuristic* or domain specific knowledge can be represented.
- *Extended logical inferences*, such as default reasoning facilitated.
- *Side effects* of actions may be modeled. Some rules may become false in time. Keeping track of this in large systems may be tricky.

Disadvantages:

- Completeness -- not all cases may be represented.
- Consistency -- not all deductions may be correct.

  For example, if we know that *Fred is a bird* we might deduce that *Fred can fly.* Later we might discover that *Fred is an emu.*

- Modularity is sacrificed. Changes in knowledge base might have far-reaching effects.
- Cumbersome control information.

People, who claim to know how to drive, are not simply claiming that they understand the theory involved in driving. Rather, they are claiming that actually possess the skills involved, that they are able to do drive.

b) Declarative Knowledge

Declarative knowledge is the type of knowledge that is expressed in declarative sentences or indicative propositions. Declarative knowledge is factual knowledge. For example, knowing that "A cathode ray tube is used to project a picture in most televisions." is a declarative knowledge.

When we say things like "I know that the internal angles of a triangle add up to 180 degrees" or "I know that it was you that ate my sandwich", we are claiming to have propositional knowledge.

It is also known as descriptive knowledge or propositional knowledge. It is the knowledge which describes how things are. It is assertion[1]-oriented and focuses on describing the properties.

c) Heuristic Knowledge

Heuristic knowledge is the less rigorous (exact), more experiential, more judgmental knowledge of performance. In contrast to factual knowledge, heuristic knowledge is rarely discussed, and is largely individualistic. It is the knowledge of good practice, good judgment, and plausible (probable) reasoning in the field. It is the knowledge that underlies the "art of good guessing."

Heuristic (Greek **meaning: "find"** or **"discover"**) refers to experience-based techniques for problem solving, learning, and discovery. Where an exhaustive search is impractical, heuristic methods are used to speed up the process of finding a satisfactory solution.

[1] Assertion: the act of stating something.

The most fundamental heuristic is trial and error, which can be used in everything from matching nuts and bolts to finding the values of variables in algebra problems.

Thus heuristic is the process of gaining knowledge or some desired result by intelligent guesswork rather than by following some pre-established formula. For example, human chess players use heuristic knowledge. It is the knowledge of 1) approaches that are likely to work or 2) properties that are likely to be true (but not guaranteed).

A heuristic method is a problem solving approach characterized by exploration and trial and error.

d) Relational Knowledge

The simplest way of storing facts is to use a relational method where each fact about a set of objects is set out systematically in columns. This representation gives little opportunity for inference, but it can be used as the knowledge basis for inference engines.

- Simple way to store facts.
- Each fact about a set of objects is set out systematically in columns (Figure).
- Little opportunity for inference.
- Knowledge basis for inference engines.

Class	Roll No	Name of Student	Address	Contact No
5	501	Ram Poudel	Pokhara	061886788
5	504	Hari Baral	Kathmandu	014895463
6	603	Sujan Chhetri	Pokhara	061866734

Figure 1.2: Simple Relational

Knowledge We can ask questions like:

- How many students are from Pokhara?
- What is the name of student whose Roll No is 504?

Relational knowledge is made up of objects consisting of attributes and corresponding associated values. This sort of representation is popular in database systems.

e) Inheritable knowledge

One of the most important forms of inference is property inheritance in which elements of specific classes inherit attributes and values from more general classes in which they are included.

- Property inheritance
    - Elements inherit values from being members of a class.
    - Data must be organized into a hierarchy of classes (Figure 1.3).

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy.

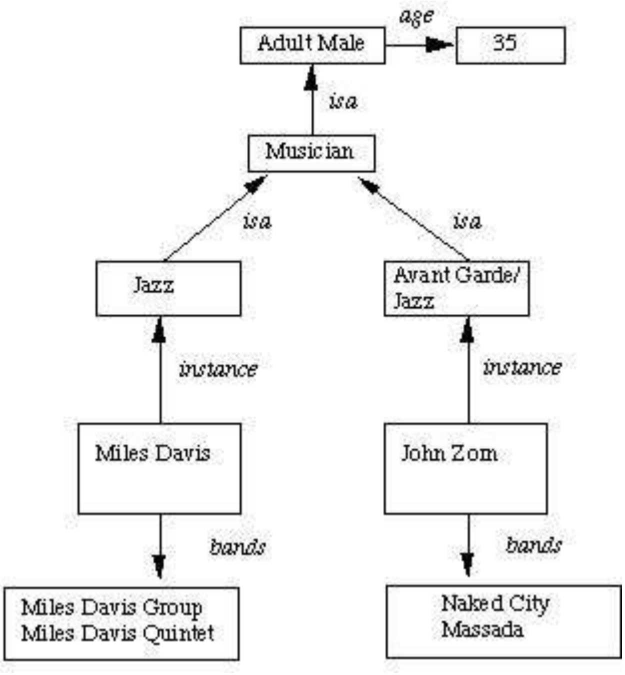


Figure 1.3: Property Inheritance Hierarchy

- Lines represents attributes
- Boxed nodes represent objects and values of attributes of objects. These values can be viewed as objects with attributes and values and so on.
- Arrows on the lines point from object to its value.
- Isa Relation: Musician is an adult male.
- Instance: Miles Devis is an instance of Jazz.

This structure is known as a slot and filler structure, semantic network or a collection of frames.

f) Inferential Knowledge

Inference knowledge represents knowledge as *formal logic*. For example, let us consider a statement: ***All dogs have tails.*** This statement is represented in predicate logic as:

$$\forall x: dog(x) \rightarrow hasatail(x)$$

Advantages:

- *A set of strict rules.*
  - Can be used to derive more facts.
  - Truths of new statements can be verified. ○ Guaranteed correctness.
- Popular in AI systems. *For example,* automated theorem proving.

1.6 Turing Test

The **Turing test** is a test of a machine's ability to exhibit intelligent behavior. A human judge engages in a natural language conversation with one human and one machine, each emulating human responses. All participants are separated from one another. If the judge cannot reliably tell the machine from the human, the machine is said to have passed the test.

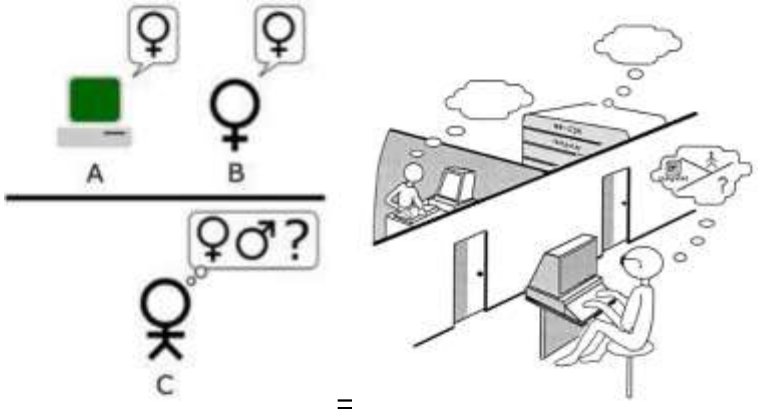


Figure 1.4: Turing Test. The "standard interpretation" of the Turing Test, in which player C, the interrogator, is tasked with trying to determine which player A or B is a computer and which is a human. The interrogator is limited to using the responses to written questions in order to make the determination.

The Turing Test was proposed by Alan Turing in 1950, with the aim to provide a satisfactory operational definition of intelligence. Turing defined intelligent behavior as the ability to achieve human-level performance in all cognitive[2] tasks, sufficient to fool an interrogator.

The test he proposed is that *the computer should be interrogated by a human via a teletype, and passes the test if the interrogator cannot tell if there is a computer or a human at the other end*. The computer would need to possess the following capabilities to pass the Turing test:

- **Natural language processing** to enable it to communicate successfully in English (or some other human language);
- **Knowledge representation** to store information provided before or during the interrogation;
- **Automated reasoning** to use the stored information to answer questions and to draw new conclusions;
- **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

1.7 Intelligent Agents

Agent

An **agent is** anything that can be viewed as **perceiving** its environment through **sensors and acting** upon that environment through **actuators**[3]. ***Agents*** interact with environment through sensors and actuators. This situation is illustrated in figure 1.5.

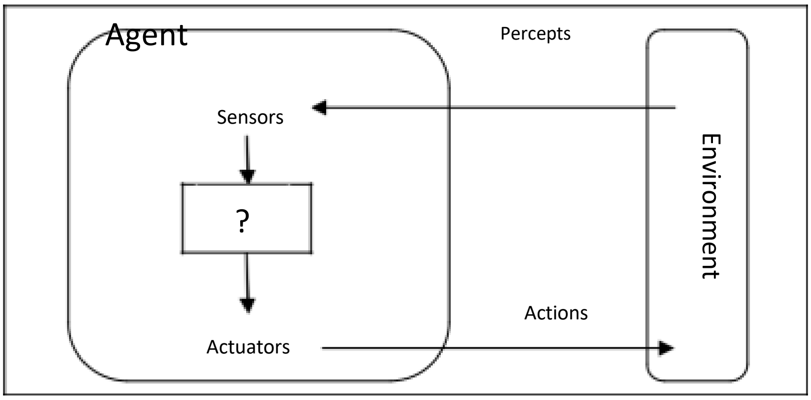


Figure 1.5 *Agents interact with environments through sensors and actuators*

[2] Cognition: The psychological result of perception and learning and reasoning.

[3] Actuator: A mechanism that puts something into automatic action.

For example, a human agent has eyes, ears, and other organs for sensors, and hands, legs, mouth, and other body parts for **actuators**. A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the **actuators**.

The term **percept** refers to the agent's perceptual inputs at any given time. An agent **percept sequence** is the complete history of everything the agent has ever perceived. The **agent behavior** is described by the **agent function** that maps any given percept sequence to an action. The agent function for an artificial agent will be implemented by an **agent program**. The agent function is an abstract mathematical description and agent program is a concrete implementation, running on the agent architecture.

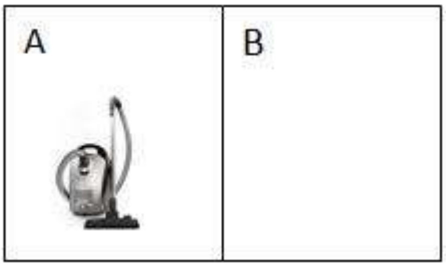


Figure 1.6: Vacuum Cleaner World

Example: The Vacuum cleaner world (Figure 1.6). Let this particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt or do nothing. One very simple agent function is as following: if the current square is dirty, then suck, otherwise move to the other square.

The figure 1.7 shows the tabulated agent function for the vacuum cleaner world:

Percept Sequence	**Action**
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
....	....

Figure 1.7: Partial tabulation of a simple agent function for the vacuum cleaner world.

Performance Measure

The performance measure evaluates the behaviour of the agent in an environment. It defines the agent's degree of success.

A Ration Agent

A ration agent is one that does the right thing. Conceptually speaking, every entry in the table of the agent function is filled out correctly.

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Structure of Agent

The **agent program is** a function that implements the agent mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators, which we will call the **architecture.** The relationship among agents, architectures, and programs can be summed up as follows:

Agent = Architecture + Program

Where, architecture is a computing device running agent program with sensors and actuators and program implements the agent function; perform mapping of percepts to actions.

The agent programs take the current percept as input from the sensors and return an action to the actuators. Note that the agent program takes only the current percept as input while the agent function takes the entire percept history.

```
function TABLE-DRIVEN-AGENT(percept) returns action
        static: percepts, a sequence, initially empty
        table, a table of actions, indexed by percept sequences, initially
        fully specified
        append percept to the end of percepts
        action <— LOOKUP(percepts, table)
return action
```

Figure 1.8 The TABLE-DRIVENAGEBT program is invoked for each new percept and returns an action each time. It keeps track of the percept sequence using its own private data structure.

The figure 1.8 shows an agent program that keeps track of percept sequence and then uses it to index into a table of actions to decide what to do. The table represents the agent function that the agent program embodies.

Examples

1) Agent Program for Vacuum Cleaner Agent

Let us consider the vacuum cleaner world agent described in previous section. An agent program for the vacuum agent is shown in figure 1.9.

```
function REFLEX-VACUUM-AGENT(location, status) returns an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left
```

Figure 1.9: The agent program for a simple reflex agent in two-state vacuum environment. This program implements the agent function tabulated in figure 1.7.

Note: Simple reflex agents are those agents which select actions on the basis of the current percept, ignoring the rest of the percept history.

2) The Taxi Driver Agent

Percepts: input from cameras, speedometer, GPS, sonar, microphone etc

Actions: steer, accelerate, brake, talk to passenger, horn etc

Goals: (a) Make trip safe, fast, legal, and comfortable;

(b) Maximize profits.

Environment: roads, other traffic, pedestrians, customers, weather

Performance Measure: Based on one or more of the following goals:

- get to correct destination;
- minimize fuel consumption, wear and tear;
- minimize trip time;
- maximize passenger safety and comfort;
- maximize profits;

1.8 Assignments

1) Write brief history of Artificial Intelligence.
2) Write notes on Human versus Machine Performance

1.9 Exercise

1) What is Turing Test? What are the capabilities a machine needs to have to pass the Turing Test? Explain. [2011]
2) An agent consists of architecture and an agent program. Sketch the basic architecture of an intelligent agent. Give an example of an agent. [2011]
3) Is it possible to develop human level intelligence in a machine? What could be the challenges of such research? [2011]
4) Define artificial intelligence. Describe the application areas of AI. [2010]
5) What is AI? What are the things to be involved in understanding intelligence? [2008]
6) What are the capabilities a machine needs to have to pass the Turing Test? Explain. [2009], [2007]
7) Describe Turing Test. Do you think the test is an accurate measure of artificial intelligence? Explain. [2007]
8) Define and describe the difference between knowledge, belief, hypothesis and data. [2004]
9) Briefly explain with example: a) Declarative knowledge, b) Procedural Knowledge and c) Heuristic knowledge. [2004]
10) "A dumb machine can be converted into an intelligent machine and that machine can behave as human". Do you agree on this statement? If yes, why, otherwise justify. [2006]
11) Write short notes on:
    a. Relational and procedural knowledge [2011]
    b. Human versus machine performance [2011]
    c. Turing Test [2011]

Chapter 2

Problem Solving

1. DEFINITION OF PROBLEM AND SOLUTION

Problem and Solution

A problem is an opportunity for improvement. A problem results from the recognition of a present imperfect and the belief in the possibility of a better future.

A solution is defined as the management of a problem in a way that successfully meets the goals established for treating it.

Problem Solving

Problem solving is a mental process which is part of the larger problem process that includes problem finding and problem shaping. Considered the most complex of all intellectual functions, problem solving has been defined as higher-order cognitive process that requires the modulation and control of more routine or fundamental skills.

Problem solving occurs when an organism or an artificial intelligence system needs to move from a given state to a desired goal state.

2. STEPS TO SOLVE A PROBLEM

There are four things that are to be followed in order to solve a problem:

1. Define the problem.

A precise definition that must include precise specifications of what the initial situation(s) will be as well as what final situations constitute acceptable solutions to the problem.

2. Problem must be analysed.

Some of the important features land up having an immense impact on the appropriateness of various possible techniques for solving the problem.

3. Isolate and represent the task knowledge that is necessary to solve the problem.

4. Amongst the available ones choose the best problem-solving technique(s) and apply the same to the particular problem.

3. DEFINING PROBLEM AS A STATE SPACE SEARCH

A state space represents a problem in terms of states and operators that change states. A state space consists of:

- **A representation of the states** the system can be in. In a chess game, for example, any board position represents the current state of game.
- **A set of operators** that can change one state into another state. In a chess game, the operators are the legal moves from any given state. Often the operators are representation operators for representing new state.
- **An initial state** is a state that is a start state
- **A set of final states** which are terminal states, some of these may be desirable, whereas others may be undesirable. This set is often represented implicitly by, a program that detects, that identifies terminal states as a scenario.

Example: Tic-Tac Toe as a State Space

State spaces are good representations for the board games such as Tic-Tac-Toe. The states of a game can be described by the contents of the board and the player whose turn is next. The board can be represented as an array of 9 cells, each of which may contain an „X" or „O" or may be empty.

State:

- Play to move the next „X" or „O"
- The Board configuration

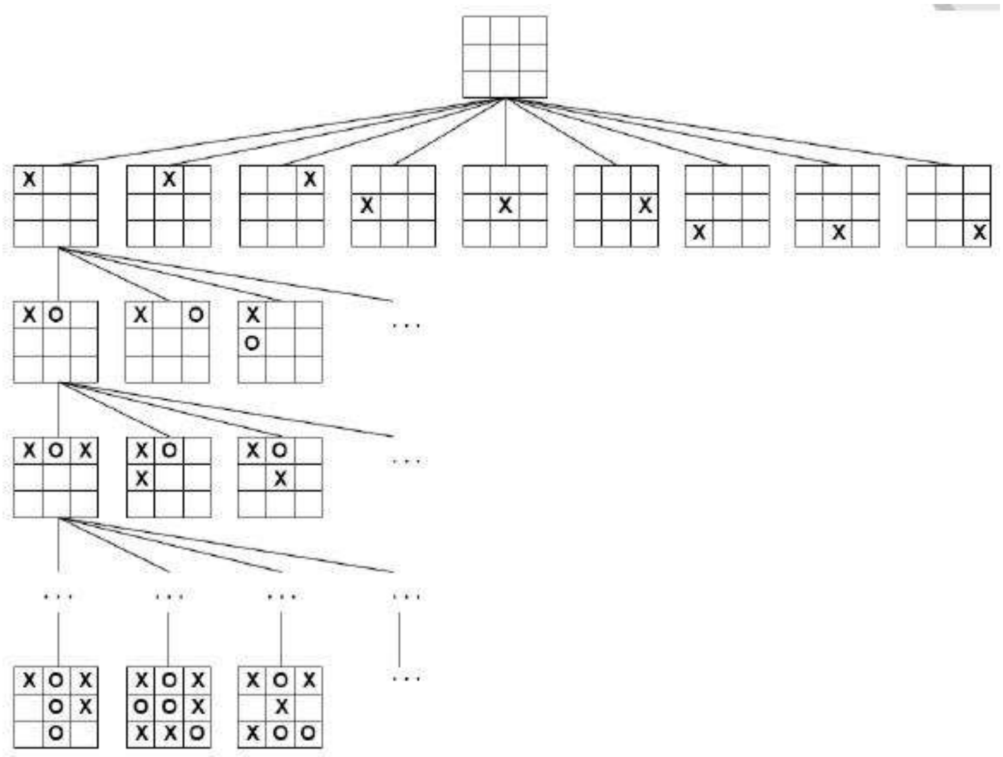


Figure 2.1: States in Tic-Tac-Toe Problems

Operators:

An operator transforms one state into another state. The operators can change an empty cell to an „X‟ or to a „O‟. That is;

- Change empty cell to „X‟
- Change empty cell to „O‟

Start State:

- Board empty
- X‟s turn

Terminal States:

- Three X‟s in a row;
- Three O‟s in a row; and
- All cells are full with no cells empty.

4. PROBLEM SOLVING AGENT

A problem solving agent is a kind of goal-based agent that decide what to do by finding sequences of actions that lead to desirable goal. Intelligent agents can solve problems by searching a state-space.

A problem solving agent first formulates a goal and a problem, searches for a sequence of actions that would solve the problem and then executes the actions one at a time. A problem solving agent solves a given problem in the following four steps:

1. Goal formulation
2. Problem formulation
3. Search
4. Execution

Goal and Goal Formulation

A goal state can be defined as a state that would be acceptable as solution to a problem. It is a state that a problem solving agent is trying to achieve. Goal can be defined as set of world states. Goal is a set of desirable world states in which the objectives have been achieved.

Goal formulation is the first step of problem solving and is the process of finding out what states may be the possible goal states (solutions) of a particular problem. It is the process of limiting the objectives since we may have more than one objective.

In other words, a goal formulation is the process that:

- Specifies the objectives to be achieved. That is, declaring the Goal.
- Ignores the some actions.
- Limits the objective that agent is trying to achieve.

Problem Formulation

After the formulation of the goal, it is the task of problem solving agent to find out which sequence of actions will get it to a goal state. Before doing this, it needs to decide what sorts of actions and states to consider. This process is called problem formulation. This means that a problem formulation is the act of deciding what actions and states are to be considered to achieve its goal.

Search and Solution

After knowing the goal and different actions, an agent chooses a sequence of actions to reach to the goal state from the initial state. This process of looking such a sequence is called a **search.**

There may be different possible set (sequence) of actions and states that can lead to the goal state of a problem. Then it is most desirable that an agent choose the best sequence of actions that lead it to the goal state in terms of path cost. In other word, search is a process that try to find a low cost path (sequences of possible states from initial state to a goal state) to reach to the goal state. It is not always the case that every search algorithm finds a solution with lowest path cost.

A search algorithm takes a problem as input and returns a solution in the form of a sequence of actions. A **solution** is a sequence of actions leading from the initial state to a goal state of a problem. Solution quality is measured by the **path cost function** and an optimal solution has the lowest path cost among all solutions.

Execution

Once a solution is found, the sequence of actions recommended by the search algorithm can be carried out to solve the given problem. This process is called execution. To solve a problem: "Formulate goal and problem, Search a solution, Execute it".

5. WELL DEFINED PROBLEMS AND SOLUTIONS

The exact formulation of problems and solutions are taken as well defined problems and solutions. A problem can be defined formally by four components:

1) **Initial state:**

It is the state the agent starts in. It is also known as start state. For example, if you are at Pokhara and want to go to Kathmandu, then Pokhara is your **initial state** and Kathmandu is the **goal state**.

2) **Actions/Operators:**

An action is a description of the possible actions available to the agent. The most common formulation uses a successor function.

Given a particular state *x*, a **successor function, S(x),** returns the set of states reachable from *x* by any single action.

The initial state and the successor function implicitly define the **state space** of the problem- the set of all states reachable from the initial state.

3) Goal Test

Goal test determines whether a given state is a goal state. It is a test applied to a state which returns true if we have reached a state that solves the problem. A goal state is a state that an agent is trying to achieve to solve the problem. A problem can have a single goal state or a set of goal states

4) Path Cost

A path in a state space is a sequence of states connected by a sequence of actions. A **path cost** function assigns a numeric cost (value) to each path.

In other words, **a path cost** function is a function that determines how much it costs to take a particular sequence of actions.

Examples of Formulating Problems

1) Formulating Vacuum World problem.

Problem Description: There are two squares A and B which are either dirty or clean (Figure 1.6). The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt or do nothing so that both squares will be clean at last.

Problem Formulation:

States: Locations of vacuum squares dirty or clean {1, 2, 3, 4, 5, 6, 7, 8}

Initial State: one of the states {1, 2, 3, 4, 5, 6, 7, 8}

Operators: move left, move right, suck

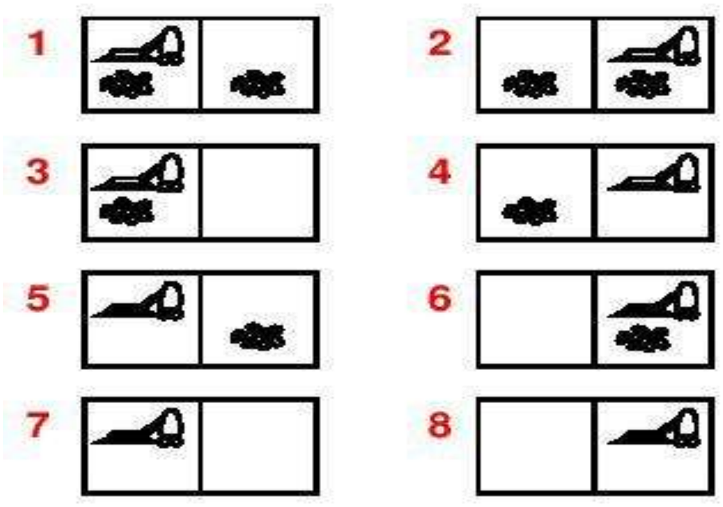


Figure 2.2: Possible states of Vacuum cleaner world

Goal test: all squares clean (goal state: {7, 8})

Path cost: 1 unit per action

2) The 8-Queens Problem:

Problem Description: Arrange the 8 queens on a chess board in such positions so that no queen attacks any other.

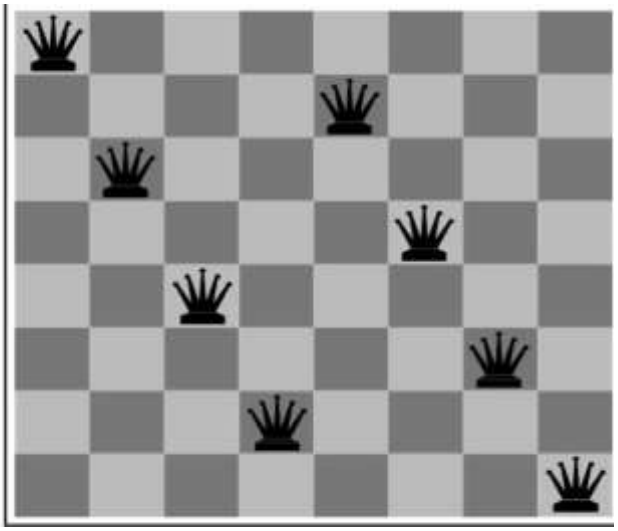


Figure 2.3 goal state of 8-queen problem

Problem Formulation:

States: arrangement of 8 queens on the board

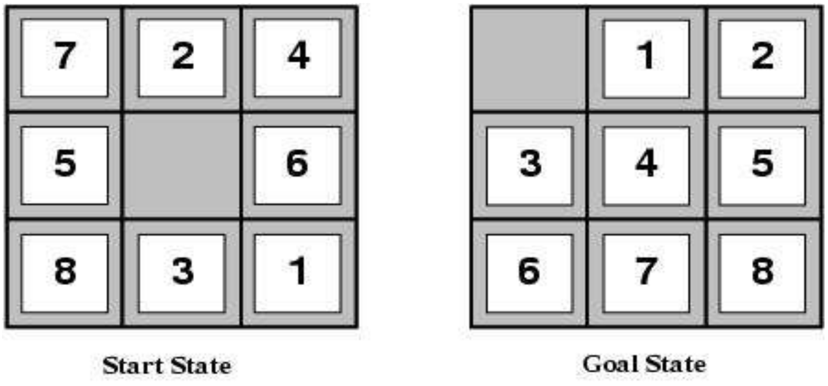
Initial State: no queens on the board

Operators: add queen to any empty square

Goal test: 8 queens on the board such that no queen attacks any other

Path cost: 1 unit per move

3) The 8-puzzle problem



(a) **(b)**

Figure 2.4: Board states of 8-puzzle problem- (a) one of the start state among various possible states (b) a goal state

Problem Description: Arrange the tiles as in Figure 2.4 (b).

Problem Formulation:

States: locations of tiles

Initial state: state given in Figure 2.4 (a)

Actions: move blank left, right, up, down

Goal test: state given in Figure 2.4 (b)

Path cost: 1 per move

4) A Water Jug Problem

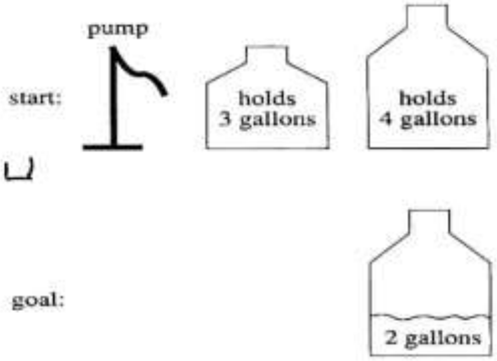


Figure 2.5: A water jug problem

Problem Description: You have a 4-gallon and a 3-gallon water jug and a faucet with an unlimited amount of water. You need to get exactly 2 gallons in 4-gallon jug.

Problem Formulation:

State representation: (x, y) where x: Contents of four gallon and y: Contents of three gallon

Start state: (0, 0)

Goal state: (2, n)

Operators:

Fill 3-gallon from faucet; fill 4-gallon from faucet

Fill 3-gallon from 4-gallon; fill 4-gallon from 3-gallon

Empty 3-gallon into 4-gallon, empty 4-gallon into 3-gallon

Dump 3-gallon down drain, dump 4-gallon down drain

One Solution to the Water Jug Problem

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug
0	0
0	3
3	0
3	3
4	2
0	2
2	0

5) Problem Formulation - Romania

Problem Description: On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest. The agent needs to research Bucharest by tomorrow. Formulate the problem of getting to Bucharest from Arad. *Find solution: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest.*

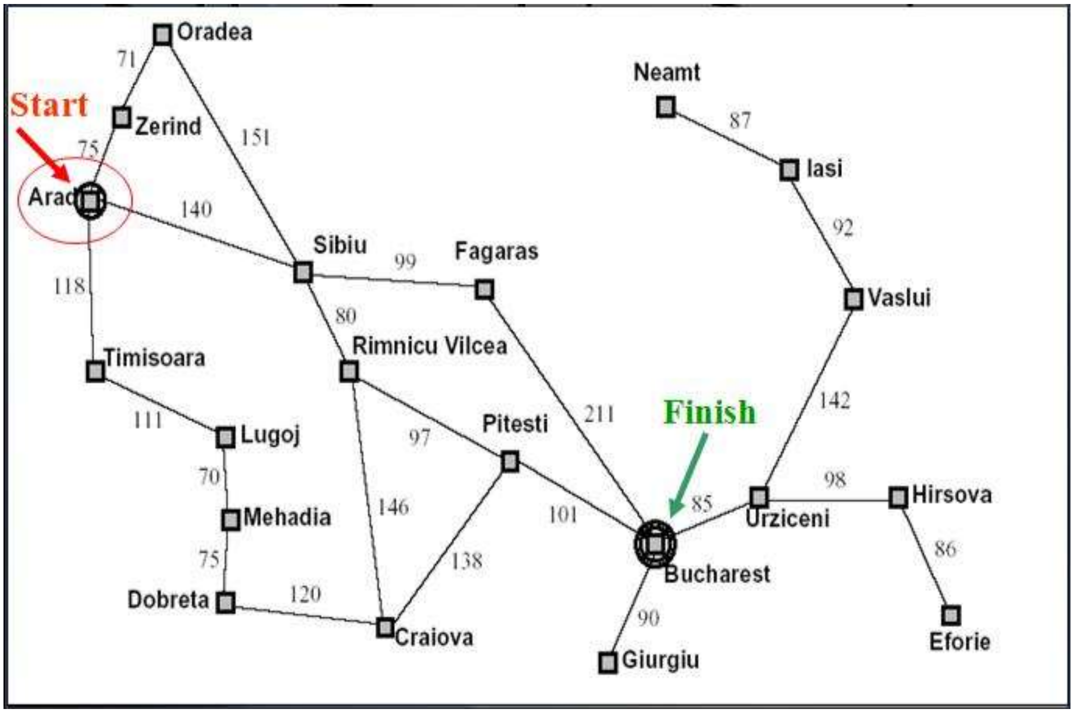


Figure 2.6: A simplified road map of part of Romania

Problem Formulation:

Initial State: Arad

Goal State: Bucharest

Operator: driving between cities

State space: consists of all 20 cities in the graph

Goal Test: is the current state Bucharest?

Path cost: is a function of time/distance/risk/petrol/…

A formal description of a problem, we must do the following:

1. **Define a state space**

   It contains all the possible configurations of the relevant objects and perhaps some impossible ones too. It is, of course, possible to define this space without explicitly enumerating all of the states it contains.

2. **Specify initial states**

   One or more states within that space that describe possible situations from which the problem-solving process may start.

3. **Specify Goal states.**

   Goal states are one or more states that would be acceptable as solutions to the problem.

4. Specify **a set of rules** that describe the **actions** (operators) available.

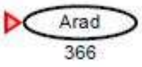
6. SEARCHING FOR SOLUTIONS

Search is one of the most powerful approaches to problem solving in AI. Search is a universal problem solving mechanism that 1) systematically explores the **alternatives** and finds the sequence of steps **towards a solution**.

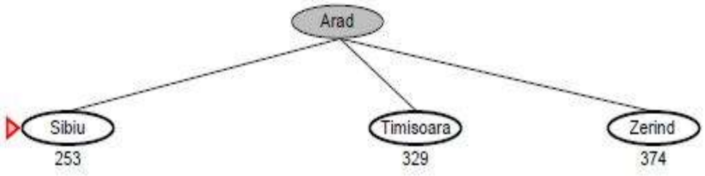
After formulating the problems, the next step is to solve the problem. This is done by a search through the state space. A **search tree** is a tree generated by the initial state and the successor function that together define the state space. It is a technique to search the nodes in a tree or **graph** (graph: - when the same state can be reached from multiple paths), which represent the state-space. The sequence of states formed by possible moves is called a search tree. This search tree is used by the different search techniques to find the solution of the problem. A search trees is an effective way to represent the search process.

The root of a tree is a **search node.** To expand the search tree to find a path from initial state to reach to the goal state, the first step is to test whether the search node is a goal state. If it is the goal state, then return the goal state and stop search. If not, expand the current state; that is, applying the successor function to the current state, thereby generating a new set of states.

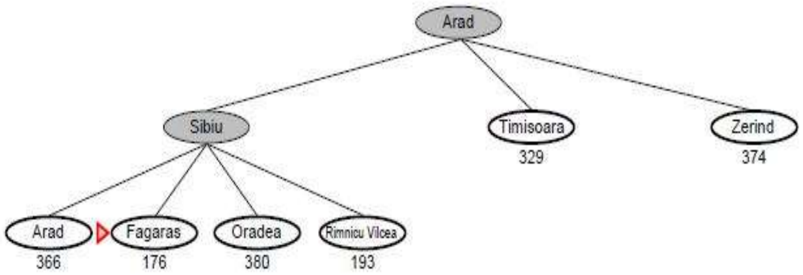
A **search strategy** is a technique that determines the choice of which state to expand next in the search tree. A **fringe** is a collection of nodes that have been generated but not yet expanded.



(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu

Figure 2.6: Partial search tree for finding a route from **Arad** to **Bucharest**. **The** nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold.

Measuring problem-solving performance

A problem solving algorithm either returns a **solution** or a **failure** as its output. It is not always the case that it either returns a solution or failure, but there is a possibility that some algorithms get stuck in an infinite loop and never return an output. The performance of a search algorithm is evaluated by the following measures:

Completeness: Does it always find a solution if one exists?

Optimality: Does it always find a least-cost solution?

Time complexity: Number of nodes generated. That is, it determines the time taken to find a solution.

Space complexity: Maximum number of nodes in memory. That is, it determines how much memory is needed for the search to store the nodes.

Time and space complexity are measured in terms of maximum **branching factor *b*** (maximum number of successors of any node)of the search tree, depth of the least-cost solution, ***d*** and maximum depth ***m*** of the state space (may be ∞).

Search cost = time complexity or space complexity

Total cost = search cost + path cost of the solution found

7. PRODUCTION SYSTEMS

Definition

Production systems provide search structures that form the core of many intelligent processes. Hence it is useful to structure AI programs in a way that facilitates describing and performing the search process. Do not be confused by other uses of the word *production,* such as to describe what is done in factories.

Components of production system

A production system consists of:

A Set of Rules: each consisting of a left side and a right hand side. Left hand side or pattern determines the applicability of the rule and a right side describes the operation to be performed if the rule is applied.

One or More Knowledge/Databases: contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.

A Control Strategy: specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.

A Rule Applier: execute the sequence of rules to get the solution.

In order to solve a problem, firstly we must reduce it to one for which a precise statement can be given. This is done by defining the problem's state space, which includes the start and goal states and a set of operators for moving around in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modelled as a production system.

8. CONSTRAINT SATISFACTION PROBLEM

Search can be made easier in cases where the solution instead of corresponding to an optimal path, is only required to satisfy local consistency conditions. We call such problems *Constraint Satisfaction (CS) Problems*. In other words, CSPs are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations.

For example, in a crossword puzzle it is only required that words that cross each other have the same letter in the location where they cross.

Examples of simple problems that can be modeled as a constraint satisfaction problem:

- The n-Queen Problem
- A Crossword Problem
- A Cryptography Problem
- A Map Coloring Problem
- Sudoku

A **Constraint Satisfaction Problem** is characterized by:

- ***A set of variables*** $\{x_1, x_2, .., x_n\}$,
- **For each variable x_i, a *domain* D_i** 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.

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. That is, a **solution** to a CSP is an assignment of values, one to each variable in such a way that no constraint is broken.

A constraint satisfaction problem on such domain contains a set of variables whose values can only be taken from the domain, and a set of constraints, each constraint specifying the allowed values for a group of variables. A solution to this problem is an evaluation of the variables that satisfies all constraints. In other words, a solution is a way for assigning a value to each variable in such a way that all constraints are satisfied by these values.

Resolution of CSPs

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search.

Backtracking is a recursive algorithm. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. Several variants of backtracking exist. Backmarking improves the efficiency of checking consistency. Backjumping allows saving part of the search by backtracking "more than one variable" in some cases. Constraint learning infers and saves new constraints that can be later used to avoid part of the search. Look-ahead is also often used in backtracking to attempt to foresee the effects of choosing a variable or a value, thus sometimes determining in advance when a sub-problem is satisfiable or unsatisfiable.

Constraint propagation techniques are methods used to modify a constraint satisfaction problem. More precisely, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraints propagation has various uses. First, they turn a problem into one that is equivalent but is usually simpler to solve. Second, they may prove satisfiability or unsatisfiability of problems. This is not guaranteed to happen in general; however, it always happens for some forms of constraint propagation and/or for some certain kinds of problems.

Local search methods are incomplete satisfiability algorithms. They may find a solution of a problem, but they may fail even if the problem is satisfiable. They work by iteratively improving a complete assignment over the variables. At each step, a small number of variables are changed value, with the overall aim of increasing the number of constraints satisfied by this assignment.

Example: A Cryptography Problem

Problem description:

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

Initial State:

- No two letters have the same value
- The sums of the digits must be as shown in the problem

STEP ONE

If you add two, four-digit numbers, such as those represented by the words "SEND" and "MORE," and you wind up with a five-digit number, like "MONEY," then the "M" in the word "MONEY" must be the result of the number 1 being carried over from the addition of S+M.

We know that M can't be greater than 1, because you can't carry over a 2 from the addition of "S+M."

And we know M can't be zero, otherwise it wouldn't be out front in "MORE" and "MONEY." (How many numbers do you know that start with 0? We rest our case!)

So, we know that M = 1.

STEP TWO

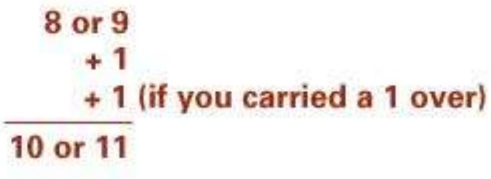
Remember that the sum of S+M must be 10, or greater. If it's not, there's nothing to carry over to give us the M in "MONEY."

Since we already know that M=1, there are only two choices for S. It can be 9, or it can be 8, if there is a "1" carried over from the sum of "E" and "O"

So, S = 8 or 9

STEP THREE

You can either carry a 1 over from the E and O column, or not. That means that the addition in the next column, S+M looks like this:



```
  8 or 9
     + 1
     + 1 (if you carried a 1 over)
 --------
 10 or 11
```

That means what? O is either 0 or 1. And we know it can't be 1, because M is one.

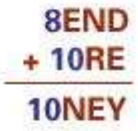
So, O = zero.

We're on our way to solving this puzzler, really!

STEP FOUR

Here's where it gets really interesting. Let's try to use S = 8.

Here's what you get:



```
   8END
 + 10RE
 ------
  10NEY
```

For this to be true, E must be equal to 9, so you can carry a 1 over to the next column. In that case, the N in MONEY has to be zero.

But, wait... we already have a zero, remember? So, this example doesn't work. S can't be equal to 8.

Therefore, we know that S = 9.

STEP FIVE

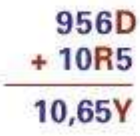
We know that E is one less than N, because E+0 =N in the column above, which means that for that column to be true, there must be a carry over of 1 from the "N+R" column. Otherwise, E would be equal to N, and that can't happen.

STEP SIX

Now, try substituting consecutive numbers for E and N.

Try 2 and 3 for E and N. It doesn't work. Try 3 and 4 — those don't work, either. Neither do 4 and 5.

But, try 5 and 6... and you get lucky! It works, and you get....



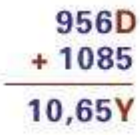
```
  956D
+ 10R5
------
10,65Y
```

So, R must be equal to 9 or, if 1 is carried over from D+5, then 8. But, we know that S = 9...

R must be equal to 8.

STEP SEVEN

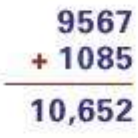
So, here's what we have now:



```
  956D
+ 1085
------
10,65Y
```

D must be at least 5, since one need to be carried over. But, we already have assigned 5, 6, 8 and 9.

So, D must be 7.

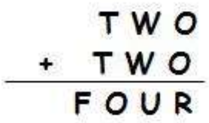


```
  9567
+ 1085
------
10,652
```

Thus, s = 9, e = 5, n = 6, d = 7, m = 1, o = 0, r = 8 and y = 2

Assignments

1) State the 8-queen problem and formulate this problem.
2) Solve the following crypt arithmetic problem:



$$\begin{array}{r} \mathrm{T\,W\,O} \\ +\ \mathrm{T\,W\,O} \\ \hline \mathrm{F\,O\,U\,R} \end{array}$$

3) Solve all the question of Exercise 2.

Exercise 2

1) What are the aspects of a formal definition of a problem? Explain with example. [2008, 2011]
2) "Constraint satisfaction is a two step process". Illustrate this with example. [2007]
3) What do you mean by well-defined problem? Explain the steps of problem solving. [2007]
4) What do you mean by a production system in AI? What are the major components of AI production system? [2004]
5) Model the water-jug problem as an AI production system. [2004]
6) Discuss the role of control strategy in AI production system. [2004]
7) What do you mean by constraint satisfaction problem? Explain with sample example. [2006]

Table of Contents

1. PLANNING

Planning refers to the process of choosing or computing the correct sequence of steps to solve a given problem before executing the sequence of actions. Planning starts with general facts about the world, facts about the particular situation and a statement of a goal. From these, they generate a strategy for achieving the goal. In the most common cases, the strategy is just a sequence of actions. A plan can then be seen as a sequence of operations that transform the initial state into the goal state. Typically we will use some kind of search algorithm to find a good plan. To do this we need some convenient representation of the problem domain. We can define states in some formal language, such as a subset of predicate logic, or a series of rules.

To the point, planning g**enerate** a set of **actions (a plan)** that can transform an **initial state** of the problem to a **goal state. Execution then s**tarts at the initial state, and **perform** each action of a generated plan. Planning is a very simple concept: a planner starts in an initial state and has a particular goal it needs to achieve. To reach the goal state, the planner develops a plan and then **executes** that plan. **Planning is a problem solving technique**.

1.2 Components of a Planning System

To perform its task of transforming an initial state into a goal state, a planning system has the following five components:

1. Choosing Rules to Apply
2. Applying Rules
3. Detecting a Solution
4. Detecting Dead Ends
5. Repairing an Almost Correct Solution

Choosing Rules to Apply

- Choose the best rule to apply next based on the best available heuristic information.

Applying Rules

- Apply the chosen rule to compute the new problem state that arises from its application

Detecting a Solution

- Detect when a solution has been found

Detecting Dead Ends

- Detect dead ends so that they can be abandoned and the system's effort directed in more fruitful directions.

Repairing an Almost Correct Solution

- Detect when an almost correct solution has been found and employ special techniques to make it totally correct.

1.3 Types of Planning

There are a number of planning approaches. Some of the important planning techniques include following:

a) Linear planning (Goal stack planning)
b) Nonlinear planning
c) Hierarchical Planning
d) Case-based Planning

2.1.1 Linear Planning

The linear planning is also called ***goal stack planning*** since this technique uses a ***goal stack*** to perform its task. The basic idea of linear planning is: ***Work on one goal until completely solved before moving on to the next goal.***

In this planning, the problem solver makes use of single stack that contains both goals and operators that have been proposed to satisfy those goals. The problem solver also relies[1] on a database that describes the current situation and a set of operators described as PRECONDITION, ADD and DELETE lists.

The **goal stack planning** attacks problems involving conjoined[2] goals by solving the goals one at a time. A plan generated by this method contains a sequence of operators for attaining the first

[1] Trust
[2] Come together

goal, followed by a complete sequence for the second goal etc. STRIPS (STanford Research Institute Problem Solver) is an example of linear planner.

Advantages of linear planning

- Reduced search space, since goals are solved one at a time and not all possible goal orderings are considered
- Advantageous if goals are (mainly) independent
- Linear planning is **sound**

Disadvantages of linear planning

- Linear planning may produce **suboptimal** solutions (based on the number of operators in the plan)
- Planner's efficiency is sensitive to goal orderings
  - Control knowledge for the —right‖ ordering
  - Random restarts
  - deepening
- Iterative Linear planning is **incomplete**

2.1.2 How the linear planning works

To illustrate how the linear planning works, let us consider a simple Block World problem as shown in below figure.

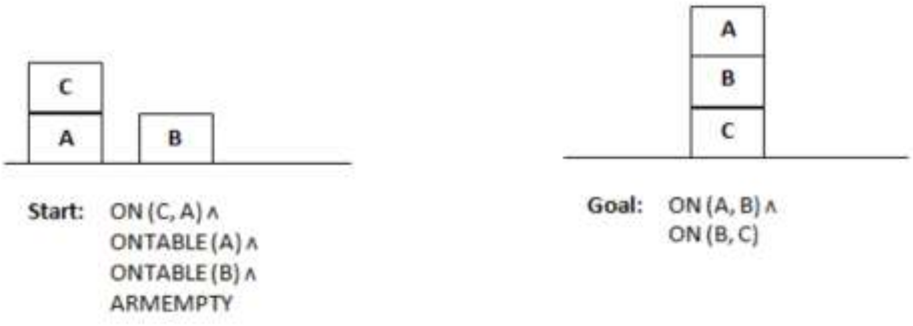


Figure: A Blocks World Problem (The Sussman Anomaly)

Description of the problem: there is a flat surface on which blocks can be placed. There are a number of square blocks, all the same size. They can be stacked one upon another. There is a robot arm that can manipulate the blocks. The actions it can perform include:

- **UNBLOCK (A, B)**:- Pick up block A from its current position on block B. The arm must be empty and the block A must have no blocks on top of it.
- **STACK (A, B):-** Place block A on Block B. The arm must already be holding and the surface of B must be clear.
- **PICKUP (A)**:- Pick up block A from the table and hold it. The arm must be empty and there must be nothing on the top of the block A
- **PUTDOWN (A):**– Put block A down in table. The arm must have been holding block A.

The robot arm can hold only one block at a time. Each block can have at most one other block directly on top of it. To specify both the conditions under which an operation may be performed and the results of performing it, we have the following predicates:

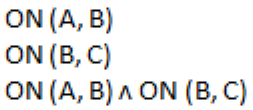
- **ON (A, B): -** Block A is on block B.
- **ONTABLE (A): -** Block A is on table.
- **CLEAR (A): -** There is nothing on top of block A.
- **HOLDING (A): -** The arm is holding block A.
- **ARMEMPTY: -** The arm is holding nothing.

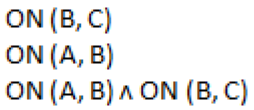
Solution:

1) At the beginning, the goal stack is simply:

   ON (A, B) ∧ ON (B, C)

2) Divide this problem into two sub-problems, one for each component of the original goal. Depending upon the order in which we want to tackle the sub-problems, there are two goal stacks that could be created where each line represents one goal on the stack.


ON (A, B)
ON (B, C)
ON (A, B) ∧ ON (B, C)


ON (B, C)
ON (A, B)
ON (A, B) ∧ ON (B, C)

[1] [2]

Figure: Two Goal Stacks

3) Suppose that we choose alternative [1] and begin trying to get A on B. we will eventually produce the goal stack as shown in below figure:

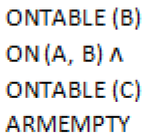
ON (C, A)
CLEAR (C)
ARMEMPTY
ON (C, A) ∧ CLEAR (C) ∧ ARMEMPTY
UNSTACK (C, A)
PUTDOWN (C)
ARMEMPTY
CLEAR (A) ∧ ARMEMPTY
PICKUP (A)
CLEAR (B) ∧ HOLDING (A)
STACK (A, B)
ON (B, C)
ON (A, B) ∧ ON (B, C)

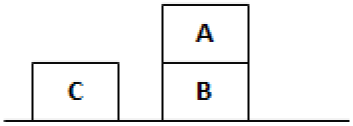
Figure: A Goal Stack

4) Pop off the stack goals (that have already been satisfied) until the goal stack is

ON (B, C)
ON (A, B) ∧ ON (B, C)

Then the current state is:


ONTABLE (B) ∧
ON (A, B) ∧
ONTABLE (C) ∧
ARMEMPTY



The sequence of operators applied so far is

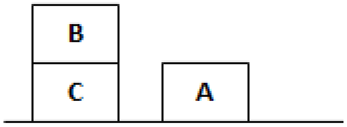
1. **UNSTACK (C, A)**
2. **PUTDOWN (C)**
3. **PICKUP (A)**
4. **STACK (A, B)**

5) Now we begin to work on satisfying ON (B, C). To do this, the sequence of operators to be applied will be

1. **UNSTACK (A, B)**
2. **PUTDOWN (A)**
3. **PICKUP (B)**
4. **STACK (B, C)**

Then the current state is:

ONTABLE (A) ∧
ON (B, C) ∧
ONTABLE (C) ∧
ARMEMPTY



6) If we check remaining goal on the stack,

ON (A, B) ∧ ON (B, C)

We found that it is not satisfied. We have undone ON (A, B) in the process of achieving ON (B, C). The difference between the goal and the current state is ON (A, B), which is now added to the stack so that it can be achieved again.

The sequence of operators applied now is:

1. **PICK (A)**
2. **STACK (A, B)**

Now goal is again checked, and this time it is satisfied. The complete plan that has been generated is

1. **UNSTACK (C, A)**
2. **PUTDOWN (C)**
3. **PICKUP (A)**
4. **STACK (A, B)**
5. **UNSTACK (A, B)**
6. **PUTDOWN (A)**
7. **PICKUP (B)**
8. **STACK (B, C)**
9. **PICK (A)**
10. **STACK (A, B)**

Problem with thus generated plan:

Although this plan will achieve the desired goal, it does not do so very efficiently. A similar situation would have occurred if we had examined the two major sub-goals in the opposite order. Thus this method (linear planning) is not capable of finding an efficient way of solving the problem.

In the above method, we perform an operation and then immediately undo it. If we find any such places, we can eliminate both the doing and undoing steps from the plan. Applying this rule to the previous plan, we eliminate steps 4 and 5 and 3 and 6. The resulting plan will be

1. **UNSTACK (C, A)**
2. **PUTDOWN (C)**
3. **PICKUP (B)**
4. **STACK (B, C)**
5. **PICKUP(A)**
6. **STACK (A, B)**

This plan now contains the minimum number of operators needed to solve the problem. This can be done using the non-linear planning.

STRIPS

A STRIP (STanford Research Institute Problem Solver) is an **operator-based** planning approach that was developed by Fikes and Nilsson in the 1970s. STRIPS uses a means–ends analysis strategy. Means–ends analysis simply involves identifying the differences between the current state and the goal state, and selecting actions that reduce those differences. STRIPS uses well-formed formulae (WFFs) in first-order predicate calculus to describe the world. STRIPS was designed to provide planning for robotic agents to enable them to navigate through a world of blocks, but the approach can also be used in other planning problems. It solved the frame problem. It deals with plan execution and learning.

STRIPS Algorithm

1. STRIPS (initial-state, goals)
    - state = initial-state; plan = []; stack = []
    - Push goals on stack
    - Repeat until stack is empty
2. If top of stack is **goal** that matches state, then pop stack
3. Else if top of stack is a **conjunctive goal** g, then
    - **Select** an ordering for the sub-goals of g, and push them on stack
4. Else if top of stack is a **simple goal** sg, then
    - **Choose** an operator o whose add-list matches goal sg
    - Replace goal sg with operator o
    - Push the preconditions of o on the stack

5. Else if top of stack is an **operator** o, then
   - state = apply(o, state)
   - plan = [plan; o]

1.4 Nonlinear Planning

From previous Sussman anomaly, we saw that difficult problems cause goal interactions. The operators used to solve one sub-problem may interfere with the solution to a previous sub-problem. Most problems require an intertwined[3] plan in which multiple sub-problems are worked on simultaneously. Such a plan is called non-linear plan because it is not composed of a linear sequence of complete sub-plans.

The basic idea of nonlinear planning is that it uses goal **set** instead of goal stack and includes all possible sub-goal orderings in the search space. It handles goal interactions by **interleaving. Some examples of nonlinear planners** include NOAH, NONLIN, DEVISER, SIPE, TWEAK etc.

As an example of the need for a nonlinear plan, let us consider the previous Sussman anomaly. A good plan for the solution of this problem is the following:

1. Begin work on the goal ON (A, B) by clearing A, thus putting C on the table.
2. Achieve the goal ON (B, C) by stacking B on C.
3. Compute the goal ON (A, B) by stacking A on B.

The planning system MOLGEN and TWEAK used constraint posting as a central technique. The idea of constraint posting is to build up a plan by incrementally hypothesizing operators, partial orderings between operators and bindings of variables within operators. At any given time in the problem-solving process, we may have a set of useful operators but perhaps no clear idea how those operators should be ordered with respect to each other. A solution is a partially ordered, partially instantiated set of operators; to generate an actual plan. We convert the partial order into any of a number of total orders.

[3] Make a loop in.

Heuristic for planning using constraint posting (TWEAK):

1. **Step Addition**: - Creating new steps for plan.
2. **Promotion:** - Constraining one step to come before another in a final plan.
3. **De-clobbering**[4]: - Placing one (possibly new) step s_2 between two old steps s_1 and s_3 such that s_2 reasserts some preconditions of s_3 that was negated (or ―clobbered‖) by s_1.
4. **Simple establishment:** - Assigning a value to a variable in order to ensure the preconditions of some step.
5. **Separation:** - Preventing the assignment of certain values to a variable.

(**Assignment:** Generate a nonlinear plan to solve the Sussman anomaly. *Refer "Artificial Intelligence, by Rich and Knight", Page no: 349 to 352*)

Advantages

- Non-linear planning is **sound**
- Non-linear planning is **complete**
- Non-linear planning may be **optimal** with respect to plan length (depending on search strategy employed)

Disadvantages

- Larger search space, since all possible goal orderings may have to be considered
- Somewhat more complex algorithm

[4] Clobber: - Beat thoroughly in a competition or fight.

Properties of Planning Algorithms

Soundness: - A planning algorithm is **sound** if all solutions found are legal plans, All preconditions and goals are satisfied and no constraints are violated (temporal, variable binding)

Completeness: - A planning algorithm is **complete** if a solution can be found whenever one actually exists. A planning algorithm is **strictly complete** if all solutions are included in the search space

Optimality: - A planning algorithm is **optimal** if the order in which solutions are found is consistent with some measure of plan quality

1.5 Non-Linear Planning Algorithm

```
1. NLP (initial-state, goals)
       – state = initial-state; plan = []; goalset = goals; opstack = []
       – Repeat until goalset is empty
2. Choose a goal g from the goalset
3. If g does not match state, then
       – Choose an operator o whose add-list matches goal g
       – Push o on the opstack
       – Add the preconditions of o to the goalset
4. While all preconditions of operator on top of opstack are met in state
       – Pop operator o from top of opstack
       – state = apply(o, state)
       – plan = [plan; o]
```

Linear vs. Non-Linear Planning

	Linear Planning	Nonlinear Planning
Idea	• Solve one goal at a time • Search with a **stack** of unachieved	• Interleave attending to subgoals • Search with a **set** of unachieved

	goals	goals
Advantages	+ Simple search strategy + Efficient **if** goals are indeed independent	+ Complete + Can produce shorter plans
Disadvantages	- May produce suboptimal plans - **Incomplete**	- Larger search space

Total vs. Partial Order Planning

	Total Order Planning	Partial Order Planning
Idea	• Plan is always a strict sequence of actions	• Plan steps may be unordered • Plan may be **linearized** prior to execution
Advantages	+ Simpler planning algorithm	+ Least commitment + Easily handles concurrent plans
Disadvantages	- No concurrent plans - May be forced to make unnecessary decisions	- Hard to determine which goals are achieved at any given time - More complex set of plan operators

2. PLANNING AS SEARCH

One approach to planning is to use search techniques. For example, a robotic planning agent might have a state that is described by the following variables:

Room the robot is in

Room the cheese is in

Is the robot holding the cheese?

Let us further suppose that there are just three rooms—room 1, room 2, and room 3—and that these rooms are arranged such that there is a door from each room to each other room. The robot starts out in room 1, and the cheese starts in room 3. The robot's goal is to find the cheese.

The actions the robot can take are as follows:

Move from room 1 to room 2

Move from room 1 to room 3

Move from room 2 to room 1

Move from room 2 to room 3

Move from room 3 to room 1

Move from room 3 to room 2

Pick up cheese

Note that each of these rules has a set of dependencies: to move from room 1 to room 2, the robot must currently be in room 1, in order to pick up the cheese, and the robot and the cheese must be in the same room, and so on.

We will use a three-value vector to represent the current state:

(room robot is in, room cheese is in, is robot holding cheese?)

So the initial state can be described as (1, 3, no), and the goal state can be described as (x, x, yes) where x is a variable that indicates the room in which both the cheese and the robot are located.

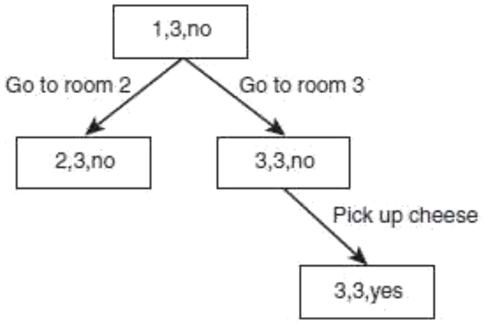


Figure: A simplistic search

Note that there are 18 possible states that can be described by this vector, but that the following 6 states are not possible because they involve the robot holding the cheese, but with the cheese in a different room from the robot: (1, 2, yes), (1, 3, yes), (2, 1, yes), (2, 3, yes), (3, 1, yes), (3, 2, yes).

Hence, there are actually only 12 valid states. In each state where the robot and the cheese are in different rooms, there are two possible actions that can be taken. For example, in the state (1, 2, no), the robot can either move from room 1 to room 2, or can move from room 1 to room 3.

To develop a plan, the robot can simply produce a search tree, such as the one shown in Fig 13.1, which starts with the initial state and shows every state that can be achieved by applying each action from that state. A suitable plan is found when the goal state is reached. Note that in this simple problem, the search tree is very small, and finding a suitable plan is thus very easy. In fact, in most real-world problems, and, indeed, in most Artificial Intelligence problems, there are many more possible states, many more actions that can be taken, and many more variables to consider. For a robotic agent to be of any use in the real world, it would need to have hundreds or even thousands of possible actions it could take to be able to deal with the enormous complexities it would surely face.

3. FORWARD CHAINING

Forward chaining is one of the two main methods of reasoning when using inference rules (in artificial intelligence) and can be described logically as repeated application of modus ponens. Forward chaining is a popular implementation strategy for expert systems, business and production rule systems. The opposite of forward chaining is backward chaining.

Forward chaining starts with the available data and uses inference rules to extract more data (from an end user for example) until a goal is reached. An inference engine using forward chaining searches the inference rules until it finds one where the antecedent[5] (**If** clause) is known to be true. When found it can conclude, or infer, the consequent (**Then** clause), resulting in the addition of new information to its data. **Inference engines will iterate through this process until a goal is reached.**

For example, suppose that the goal is to conclude the color of a pet named Fritz, given that he croaks and eats flies, and that the rule base contains the following four rules:

1. **If** X croaks and eats flies **- Then** X is a frog
2. **If** X chirps and sings **- Then** X is a canary
3. **If** X is a frog **- Then** X is green
4. **If** X is a canary **- Then** X is yellow

This rule base would be searched and the first rule would be selected, because its antecedent (**If** Fritz croaks and eats flies) matches our data. Now the consequents (**Then** X is a frog) is added to the data. The rule base is again searched and this time the third rule is selected, because its antecedent (**If** Fritz is a frog) matches our data that was just confirmed. Now the new consequent (**Then** Fritz is green) is added to our data. Nothing more can be inferred from this information, but we have now accomplished our goal of determining the color of Fritz.

Because the data determines which rules are selected and used, this method is called **data-driven**, in contrast to **goal-driven** backward chaining inference. The forward chaining approach is often employed by **expert systems**, such as **CLIPS.**

One of the advantages of forward-chaining over backward-chaining is that **the reception of new data can trigger new inferences,** which makes the engine better suited to dynamic situations in which conditions are likely to change.

[5] A preceding occurrence or cause or event.

4. BACKWARD CHAINING

Backward chaining (or **backward reasoning**) is an **inference** method that can be described as working backward from the goal(s). It is used in automated theorem provers, proof assistants and other artificial intelligence applications, but it has also been observed in primates.

In **game theory**, its application to (simpler) **sub-games** in order to find a solution to the game is called **backward induction**. In chess, it is called **retrograde analysis,** and it is used to generate table bases for **chess endgames** for **computer chess.**

Backward chaining is implemented in **logic programming** by **SLD resolution.** Both rules are based on the **modus ponens** inference rule. It is one of the two most commonly used methods of **reasoning** with **inference rules** and logical implications – the other is **forward chaining.** Backward chaining systems usually employ a **depth-first search** strategy, e.g. **Prolog**.

How it works

Backward chaining starts with a list of **goals** (or a **hypothesis**) and works backwards from the **consequent**[6] to the **antecedent** to see if there is data available that will support any of these consequents. An **inference engine** using backward chaining would search the **inference** rules until it finds one which has a consequent (**Then** clause) that matches a desired goal. If the antecedent (**If** clause) of that rule is not known to be true, then it is added to the list of goals (in order for one's goal to be confirmed one must also provide data that confirms this new rule).

For example, suppose that the goal is to conclude the color of my pet Fritz, given that he croaks and eats flies, and that the **rule base** contains the following four rules:

An Example of Backward Chaining

1. **If** X croaks and eats flies – **Then** X is a frog
2. **If** X chirps and sings – **Then** X is a canary
3. **If** X is a frog – **Then** X is green

[6] Following as an effect or result.

4. **If** X is a canary – **Then** X is yellow

This rule base would be searched and the third and fourth rules would be selected, because their consequents (**Then** Fritz is green, **Then** Fritz is yellow) match the goal (to determine Fritz's color). It is not yet known that Fritz is a frog, so both the antecedents (**If** Fritz is a frog, **If** Fritz is a canary) are added to the goal list. The rule base is again searched and this time the first two rules are selected, because their consequents (**Then** X is a frog, **Then** X is a canary) match the new goals that were just added to the list. The antecedent (**If** Fritz croaks and eats flies) is known to be true and therefore it can be concluded that Fritz is a frog, and not a canary. The goal of determining Fritz's color is now achieved (Fritz is green if he is a frog, and yellow if he is a canary, but he is a frog since he croaks and eats flies; therefore, Fritz is green).

Note that the goals always match the affirmed versions of the consequents of implications (and not the negated versions as in **modus tollens**) and even then, their antecedents are then considered as the new goals (and not the conclusions as in **affirming the consequent)** which ultimately must match known facts (usually defined as consequents whose antecedents are always true); thus, the inference rule which is used is **modus ponens**.

Because the list of goals determines which rules are selected and used, this method is called **goal-driven**, in contrast to **data-driven forward-chaining** inference. The backward chaining approach is often employed by expert systems.

Programming languages such as **Prolog, Knowledge Machine and ECLiPSe** support backward chaining within their inference engines.

Summary of Forward-Chaining and Backward-Chaining

In some problems, information is provided with the rules and the AI follows them to see where they lead. An example of this is a medical diagnosis in which the problem is to diagnose the underlying disease based on a set of symptoms (the working memory). A problem of this nature is solved using a forward-chaining, data-driven, system that compares data in the working memory against the conditions (**IF** parts) of the rules and determines which rules to fire.

In other problems, a goal is specified and the AI must find a way to achieve that specified goal. For example, if there is an epidemic of a certain disease, this AI could presume a given individual had the disease and attempt to determine if its diagnosis is correct based on available information. A backward-chaining, goal-driven, system accomplishes this. To do this, the system looks for the action in the **THEN** clause of the rules that matches the specified goal. In other words, it looks for the rules that can produce this goal. If a rule is found and fired, it takes each of that rule's conditions as goals and continues until either the available data satisfies all of the goals or there are no more rules that match.

Progression vs. Regression

	Progression (forward-chaining)	**Regression (backward-chaining)**
Idea	• (Non-deterministically) choose action whose preconditions are satisfied • Continue until goal state is reached	• (Non-deterministically) choose action that has an effect that matches an unachieved sub-goal • Add unachieved preconditions to set of sub-goals • Continue until set of unachieved sub-goals is empty
Advantages	+ Simple algorithm (―forward simulation‖)	+ Focused on achieving goals + Often more efficient
Disadvantages	- Often large branching factor - Unfocused search	- Need to reason about actions - Regression is incomplete, in general

5. Means-End Analysis

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

The means end analysis process centers around the detection of difference between the current state and the goal state. Once such a difference is isolated, an operator that can reduce the difference must be found. But perhaps that operator cannot be applied to the current state. So we set up a subproblem of getting to a state in which it can be applied. The kind of backward chaining in which operators are selected and then subgoals are set up to establish the preconditions of the operators is called operator subgoaling. But may be the operator does not produce exactly the goal state we want. Then we have a second subproblem of getting from the state it does produceto the goal. But if the difference was chosen correctly and if the operator is really effective at reducing the difference, then the two subproblems should be easier to solve than the original problem. The means-ends analysis process can then be applied recursively.

Means-ends analysis identifies and reduces, as soon as possible, *differences* between state and goals. It needs to find what *means* (operators) are available to achieve the desired *ends* (goal). To do this it finds *difference* between goal and current state, find *operator* to reduce difference and perform means-ends analysis on new subgoals. To perform means-ends analysis,

1) Until the goal is reached or no more procedures are available,
    - Describe the current state, the goal state, and the difference between the two.
    - Use the difference between the current state and goal state, possibly with the description of the current state or goal state, to select a promising procedure.
    - Use the promising procedure and update the current state.
2) If the goal is reached, announce success; otherwise, announce failure.

Algorithm: Means-End Analysis (CURRENT, GOAL)

1. Compare CURRENT to GOAL states. If there is no differences between them then return.
2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:
    a. Select an as yet untried operator O that is applicable to the current difference. If there are no such operators, then signal failure.

b. Attempt to apply O to CURRENT. Generate descriptions of two states: O-START, a state in which O's preconditions are satisfied and O-RESULT, the state that would result if O were applied in O-START.

c. If(FIRST-PART ← MEA(CURRENT, O-START)) and (LAST-PART ← MEA(O-RESULT, GOAL) are successful, then signal success and return the result of corresponding FIRST-PART, O and LAST-PART.

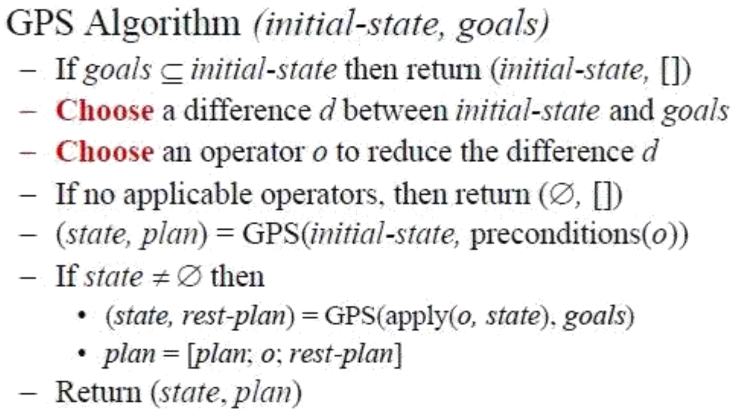
MEA (Means-Ends Analysis) is a problem solving strategy first introduced in GPS (General Problem Solver). The search process over the problem space combines aspects of both forward and backward reasoning. MEA improves over other search heuristics (again in the average case) by focusing the problem solving on the actual differences between the current state and that of the goal.

GPS

The General Problem Solver (GPS) was designed to solve problems requiring human intelligence, as well as to develop a theory of how humans solve problems. Although it was meant to be general it solved problems only in highly-structured domains (propositional calculus proofs, puzzles, symbolic integration, etc.).

GPS was the first planner to distinguish between general problem-solving knowledge and domain knowledge (hence the name). It introduced *means-end analysis* in its search. This approach tried to find a difference between the current object and the goal, and then used a lookup table to invoke an action that would minimize that difference. If the action could not be applied to the current object, it recursively tried to change the object into one that was appropriate.

General Problem Solver (GPS) is essentially linear planning using recursive procedure calls as the goal-stack mechanism.



```
GPS Algorithm (initial-state, goals)
  – If $goals \subseteq initial\text{-}state$ then return (initial-state, [])
  – Choose a difference d between initial-state and goals
  – Choose an operator o to reduce the difference d
  – If no applicable operators, then return ($\varnothing$, [])
  – (state, plan) = GPS(initial-state, preconditions(o))
  – If $state \neq \varnothing$ then
      • (state, rest-plan) = GPS(apply(o, state), goals)
      • plan = [plan; o; rest-plan]
  – Return (state, plan)
```

6. MYCIN

In artificial intelligence, **MYCIN** was an early expert system designed to identify bacteria causing severe infections, such as bacteremia and meningitis, and to recommend antibiotics, with the dosage adjusted for patient's body weight — the name derived from the antibiotics themselves, as many antibiotics have the suffix "-mycin". The Mycin system was also used for the diagnosis of blood clotting diseases.

MYCIN was developed over five or six years in the early 1970s at Stanford University. It was written in Lisp as the doctoral dissertation of Edward Shortliffe under the direction of Bruce Buchanan, Stanley N. Cohen and others. It arose in the laboratory that had created the earlier Dendral expert system.

MYCIN was never actually used in practice but research indicated that it proposed an acceptable therapy in about 69% of cases, which was better than the performance of infectious disease experts who were judged using the same criteria.

Method

MYCIN operated using a fairly simple inference engine, and a knowledge base of ~600 rules. It would query the physician running the program via a long series of simple yes/no or textual questions. At the end, it provided a list of possible culprit bacteria ranked from high to low based on the probability of each diagnosis, its confidence in each diagnosis' probability, the reasoning behind each diagnosis (that is, MYCIN would also list the questions and rules which led it to rank a diagnosis a particular way), and its recommended course of drug treatment.

Despite MYCIN's success, it sparked debate about the use of its ad hoc, but principled, uncertainty framework known as "certainty factors". The developers performed studies showing that MYCIN's performance was minimally affected by perturbations in the uncertainty metrics associated with individual rules, suggesting that the power in the system was related more to its knowledge representation and reasoning scheme than to the details of its numerical uncertainty model. Some observers felt that it should have been possible to use classical Bayesian statistics. MYCIN's developers argued that this would require either unrealistic assumptions of probabilistic independence, or require the experts to provide estimates for an unfeasibly large number of conditional probabilities.[1][2]

Subsequent studies later showed that the certainty factor model could indeed be interpreted in a probabilistic sense, and highlighted problems with the implied assumptions of such a model. However the modular structure of the system would prove very successful, leading to the development of graphical models such as Bayesian networks.[3]

Results

Research conducted at the Stanford Medical School found MYCIN to propose an acceptable therapy in about 69% of cases, which was better than the performance of infectious disease experts who were judged using the same criteria. This study is often cited as showing the potential for disagreement about thereapeutic decisions, even among experts, when there is no "gold standard" for correct treatment.[4]

Application of MYCIN

MYCIN was never actually used in practice. This wasn't because of any weakness in its performance. As mentioned, in tests it outperformed members of the Stanford medical school faculty. Some observers raised ethical and legal issues related to the use of computers in medicine — if a program gives the wrong diagnosis or recommends the wrong therapy, who should be held responsible? However, the greatest problem, and the reason that MYCIN was not used in routine practice, was the state of technologies for system integration, especially at the time it was developed. MYCIN was a stand-alone system that required a user to enter all relevant information about a patient by typing in response to questions that MYCIN would pose. The program ran on a large time-shared system, available over the early Internet (ARPANet), before personal computers were developed. In the modern era, such a system would be integrated with medical record systems, would extract answers to questions from patient databases, and would be much less dependent on physician entry of information. In the 1970s, a session with MYCIN could easily consume 30 minutes or more—an unrealistic time commitment for a busy clinician.

MYCIN's greatest influence was accordingly its demonstration of the power of its representation and reasoning approach. Rule-based systems in many non-medical domains were developed in the years that followed MYCIN's introduction of the approach. In the 1980s, expert system "shells" were introduced (including one based on MYCIN, known as E-MYCIN (followed by KEE)) and supported the development of expert systems in a wide variety of application areas.

A difficulty that rose to prominence during the development of MYCIN and subsequent complex expert systems has been the extraction of the necessary knowledge for the inference engine to use from the human expert in the relevant fields into the rule base (the so-called knowledge engineering).

Inference

Inference is the act or process of deriving logical conclusions from premises[1] known or assumed to be true. The laws of valid inference are studied in the field of logic. The process by which a conclusion is inferred from multiple observations is called **inductive reasoning**. The conclusion may be correct or incorrect, or correct to within a certain degree of accuracy, or correct in certain situations. Conclusions inferred from multiple observations may be tested by additional observations.

Example 1

All men are mortal

Socrates is a man

Therefore, Socrates is mortal.

Example 2

All fruits are sweet.

A banana is a fruit.

Therefore, a banana is sweet.

Reasoning

Reason is a term that refers to the capacity human beings have to make sense of things, to establish and verify facts, and to change or justify practices, institutions, and beliefs. Reasoning within an argument gives the rationale behind why one choice, for example should be selected over another. Types of reasoning include:

- **Abduction:** the process of creating explanatory hypotheses.
- **Analogical Reasoning:** relating things to novel other situations.
- **Cause-and-Effect Reasoning:** showing causes and resulting effect.
- **Cause-to-Effects Reasoning:** starting from the cause and going forward.
- **Effects-to-Cause Reasoning:** starting from the effect and working backward.
- **The Bradford Hill Criteria:** for cause and effect in medical diagnosis.
- **Comparative Reasoning:** comparing one thing against another.
- **Conditional Reasoning:** using if...then...
- **Criteria Reasoning:** comparing against established criteria.
- **Decompositional Reasoning**: **understand** the parts to understand the whole.
- **Deductive Reasoning:** starting from the general rule and moving to specifics.
- **Exemplar Reasoning:** using an example.

[1] A statement that is assumed to be true and from which a conclusion can be drawn.

- **Inductive Reasoning:** starting from specifics and deriving a general rule.
- **Modal Logic:** arguing about necessity and possibility.
- **Pros-vs-cons Reasoning:** using arguments both for and against a case.
- **Residue Reasoning:** Removing first what is not logical.
- **Set-based Reasoning:** based on categories and membership relationships.
- **Systemic Reasoning:** the whole is greater than the sum of its parts.
- **Syllogistic Reasoning:** drawing conclusions from premises.
- **Traditional Logic:** assuming premises are correct.

Deduction & Induction

In logic, we often refer to the two broad methods of reasoning as the ***deductive*** and ***inductive*** approaches.

Deductive Reasoning

Logic is the study of reasoning. **Deduction is a form of reasoning in which a conclusion follows necessarily from the stated premises**. Deduction is generally an inference by reasoning from the **general to the specific**. A deduction is also the conclusion reached by a deductive reasoning process. One classic example of deductive reasoning is that found in syllogisms like the following:

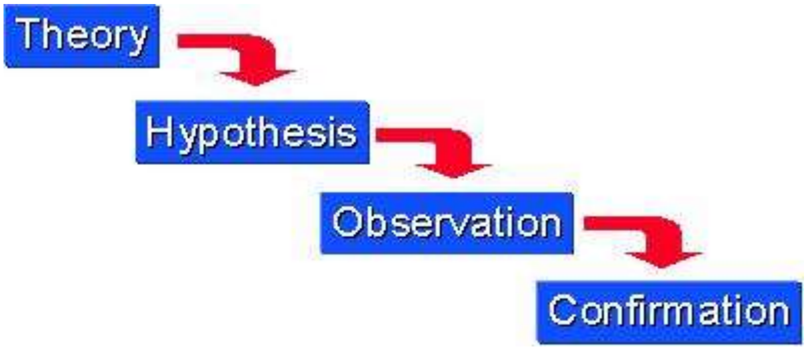
Premise 1: All humans are mortal.

Premise 2: Socrates is a human.

Conclusion: Socrates is mortal.

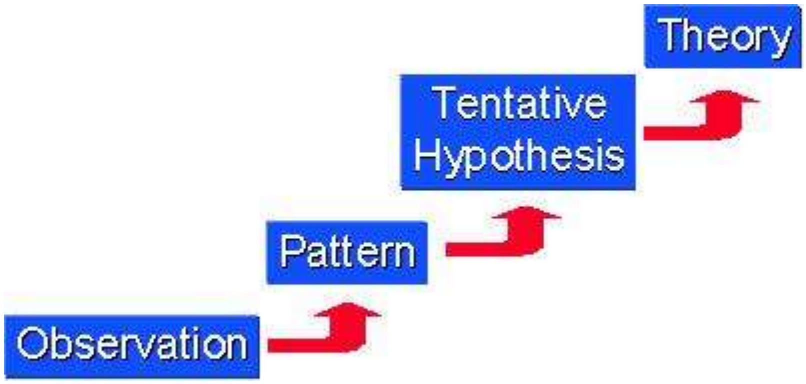
The reasoning in this argument is valid, because there is no way in which the premises, 1 and 2, could be true and the conclusion, 3, be false.

Deductive reasoning works from the more general to the more specific**. Sometimes this is informally called a "top-down" approach.** We might begin with thinking up a *theory* about our topic of interest. We then narrow that down into more specific *hypotheses* that we can test. We narrow down even further when we collect *observations* to address the hypotheses. This ultimately leads us to be able to test the hypotheses with specific data -- a *confirmation* (or not) of our original theories.



Inductive Reasoning

Inductive reasoning works the other way, moving from specific observations to broader generalizations and theories. Informally, we sometimes call this a "**bottom up"** approach (please note that it's "bottom up" and *not* "bottom**s** up" which is the kind of thing the bartender says to customers when he's trying to close for the night!). In inductive reasoning, we begin with specific observations and measures, begin to detect patterns and regularities, formulate some tentative hypotheses that we can explore, and finally end up developing some general conclusions or theories.



In summary, Deductive reasoning is one of the two basic forms of valid reasoning. It begins with a general hypothesis or known fact and creates a specific conclusion from that generalization. This is the opposite of inductive reasoning, which involves creating broad generalizations from specific observations. The basic idea of deductive reasoning is that if something is true of a class of things in general, this truth applies to all members of that class. One of the keys for sound deductive reasoning, then, is to be able to properly identify members of the class, because incorrect categorizations will result in unsound conclusions.

Truth Maintenance System (TMS)

A **truth maintenance system**, or **TMS** (also called Reason Maintenance Systems), is a knowledge representation method for representing both beliefs and their dependencies. The name *truth maintenance* is due to the ability of these systems to restore consistency[2].

It is also termed as a belief revision system; a truth maintenance system maintains consistency between old believed knowledge and current believed knowledge in the knowledge base (KB) through revision. If the current believed statements contradict the knowledge in KB, then the KB is updated with the new knowledge. It may happen that the same data will again come into existence, and the previous knowledge will be required in KB. If the previous data is not present, it is required for new inference. But if the previous knowledge was in the KB, then no retracing[3] of the same knowledge was needed. Hence the use of TMS to avoid such retracing; it keeps track of the contradictory data with the help of a dependency

[2] (logic) An attribute of a logical system that is so constituted that none of the propositions deducible from the axioms contradict one another.

[3] To go back over again.

record. This record reflects the retractions and additions which makes the inference engine (IE) aware of its current belief set.

Each statement having at least one valid justification is made a part of the current belief set. When a contradiction is found, the statement(s) responsible for the contradiction are identified and an appropriate is retraced. This results the addition of new statements to the KB. This process is called **dependency-directed backtracking.**

The TMS maintain the records in the form of a **dependency network**. The nodes in the network are one of the entries in the KB (a premise, antecedent, or inference rule etc.) Each arc of the network represents the inference steps from which the node was derived. A premise is a fundamental belief which is assumed to be always true. They do not need justifications.

Many kinds of truth maintenance systems exist. Two major types are

1. Single-context TMS and
2. Multi-context TMS

In single context systems, consistency is maintained among all facts in memory (database). Multi-context systems allow consistency to be relevant to a subset of facts in memory (a context) according to the history of logical inference. Multi-agent truth maintenance systems perform truth maintenance across multiple memories, often located on different machines.

A TMS is intended to satisfy a number of goals:

Provide justifications for conclusions

When a problem solving system gives an answer to a user's query, an explanation of the answer is usually required. If the advice to a stockbroker is to invest millions of dollars, an explanation of the reasons for that advice can help the broker reach a reasonable decision. An explanation can be constructed by the IE by tracing the justification of the assertion (statement).

Recognize inconsistencies

The IE may tell the TMS that some sentences are contradictory. Then, if on the basis of other IE commands and of inferences we find that all those sentences are believed true, then the TMS reports to the IE that a contradiction has arisen. The IE can eliminate an inconsistency by determining the assumptions used and changing them appropriately, or by presenting the contradictory set of sentences to the users and asking them to choose which sentence(s) to retract.

Support default reasoning

In many situations we want, in the absence of firmer knowledge, to reason from default assumptions. If Tweety is a bird, until told otherwise, we will assume that Tweety flies and use as justification the fact that Tweety is a bird and the assumption that birds fly.

Remember derivations computed previously

Support dependency driven backtracking

TMS Types

Truth Maintenance Systems can be categorizes, depending upon their characteristics, as:

Justification-Based Truth Maintenance System (JTMS)

It is a simple TMS where one can examine the consequences of the current set of assumptions. The meaning of sentences is not known.

Assumption-Based Truth Maintenance System (ATMS)

It allows maintaining and reasoning with a number of simultaneous, possibly incompatible, current sets of assumption. Otherwise it is similar to JTMS, i.e. it does not recognize the meaning of sentences.

Logical-Based Truth Maintenance System (LTMS)

Like JTMS in that it reasons with only one set of current assumptions at a time. More powerful than JTMS in that it recognizes the propositional semantics of sentences, i.e. understands the relations between p and ~p, p and q and p&q, and so on.

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.

Heuristic Search

Heuristic search is an AI search technique that employs heuristic for its moves. *Heuristic* is a rule of thumb that probably leads to a solution. Heuristics play a major role in search strategies because of exponential nature of the most problems. Heuristics help to reduce the number of alternatives from an exponential number to a polynomial number. In Artificial Intelligence, **heuristic search** has a general meaning, and a more specialized technical meaning. In a general sense, the term heuristic is used for any advice that is often effective, but is not guaranteed to work in every case. Within the heuristic search architecture, however, the term heuristic usually refers to the special case of a heuristic evaluation function.

Heuristic Information

In order to solve larger problems, domain-specific knowledge must be added to improve search efficiency. Information about the problem includes the nature of states, cost of transforming from one state to another, and characteristics of the goals. This information can often be expressed in the form of heuristic evaluation function, say f(n,g), a function of the nodes n and/or the goals g.

Following is a list of heuristic search techniques.

- Pure Heuristic Search
- A* algorithm
- Iterative-Deepening A*
- Depth-First Branch-And-Bound
- Heuristic Path Algorithm
- Recursive Best-First Search

Complexity of Finding Optimal Solutions

The time complexity of a heuristic search algorithm depends on the accuracy of the heuristic function. For example, if the heuristic evaluation function is an exact estimator, then A* search algorithm runs in linear time, expanding only those nodes on an optimal solution path.

Conversely, with a heuristic that returns zero everywhere, A* algorithm becomes uniform-cost search, which has exponential complexity.

Reasoning about Uncertainty:

Uncertainty: The lack of certainty, a state of having limited knowledge where it is impossible to exactly describe the existing state, a future outcome, or more than one possible outcome.

When an agent knows enough facts about its environment, the logical approach enables it to derive plans that are guaranteed to work. Unfortunately, agents almost never have access to the whole truth about their environment. Agent must therefore act under uncertainty.

If a logical agent cannot conclude that any particular course of action achieves its goal, then it will be able to act. Conditional planning can overcome uncertainty to some extent, but only if the agent's sensing actions can obtain the required information and only if there are not too many different contingencies. Another possible solution would be to endow the agent with a simple but incorrect theory of world that does enable to derive a plan presumably, such plans will work most of the time, but problems arise when events contradict the agent's theory. Moreover the tradeoff between the accuracy and usefulness of the agent's theory seems itself to require reasoning under uncertainty.

Uncertain knowledge

Let us take an example of a simple diagnosis. Diagnosis-whether for medicine, automobile or whatever-is a task that almost always involves uncertainty. Let us try to write rules for dental diagnosis using FOPL so that we can see how the logical approach breaks down. Consider the following rule:

ɏ p Symptom(p, Toothache) → Disease(p, Cavity)

The problem is that this rule is wrong. Not all patients with toothaches have cavities; some of them have gum disease, an abscess, or one of several other problems:

ɏ p Symptom(p, Toothache) → Disease(p, Cavity) ∧ Disease(p, GumDisease) ∧ Disease(p, Abscess) …

Unfortunately, in order to make rule true, we have to add an almost unlimited list of possible causes.
Thus, trying to use cope with a domain like medical diagnosis fails for the three reasons:

- **Theoretical ignorance**. There is no complete theory which is known about the problem domain. E.g., medical diagnosis.
- **Laziness**. The space of relevant factors is very large, and would require too much work to list the complete set of antecedents and consequents. Furthermore, it would be too hard to use the enormous rules that resulted.
- **Practical ignorance**. Uncertain about a particular individual in the domain because all of the information necessary for that individual has not been collected.

This is typical of medical domain, as well as most other judgmental domains include law business, design, automobile repair, gardening, dating and so on. The agent's knowledge can, at best, provide only the **degree of belief** in the relevant sentences. The main tool for dealing with the degree of belief will be the **probability theory** which assigns to each sentence a numerical degree of belief between 0 and 1. The degree of truth as opposed to the degree of belief is the subject of fuzzy logic. **Probability** provides a way of summarizing the uncertainty that comes from our laziness and ignorance.

Why Reason Probabilistically?

In many problem domains it isn't possible to create complete, consistent models of the world. Therefore agents (and people) must act in uncertain worlds (which the real world is). We want an agent to make rational decisions even when there is not enough information to prove that an action will work. **Some of the reasons for reasoning under uncertainty** include:

- **True uncertainty**. For example, flipping a coin.
- **Theoretical ignorance**. There is no complete theory which is known about the problem domain. E.g., medical diagnosis.
- **Laziness**. The space of relevant factors is very large, and would require too much work to list the complete set of antecedents and consequents. Furthermore, it would be too hard to use the enormous rules that resulted.
- **Practical ignorance**. Uncertain about a particular individual in the domain because all of the information necessary for that individual has not been collected.

Probability theory will serve as the formal language for representing and reasoning with uncertain knowledge.

Representing Belief about Propositions

Rather than reasoning about the truth or falsity of a proposition, reason about the belief that a proposition or event is true or false

For each primitive proposition or event, attach a **degree of belief** to the sentence

Use **probability theory** as a formal means of manipulating degrees of belief

Given a proposition, A, assign a probability, P(A), such that 0 <= P(A) <= 1, where if A is true, P(A)=1, and if A is false, P(A)=0. Proposition A must be either true or false, but P(A) summarizes our degree of belief in A being true/false.

Examples

P(Weather=Sunny) = 0.7 means that we believe that the weather will be Sunny with 70% certainty. In this case Weather is a random variable that can take on values in a domain such as {Sunny, Rainy, Snowy, Cloudy}.

P(Cavity=True) = 0.05 means that we believe there is a 5% chance that a person has a cavity. Cavity is a Boolean random variable since it can take on possible values *True* and *False*.

Example: P(A=a ^ B=b) = P(A=a, B=b) = 0.2, where A=My_Mood, a=happy, B=Weather, and b=rainy, means that there is a 20% chance that when it's raining my mood is happy.

Axioms of Probability Theory

Probability theory provides us with the formal mechanisms and rules for manipulating propositions represented probabilistically. The following are the three axioms of probability theory:

0 <= P(A=a) <= 1 for all a in sample space of A

P(True)=1, P(False)=0

P(A v B) = P(A) + P(B) - P(A ^ B)

From these axioms we can show the following properties also hold:

P(~A) = 1 - P(A)

P(A) = P(A ^ B) + P(A ^ ~B)

Sum{P(A=a)} = 1, where the sum is over all possible values *a* in the sample space of A

Unconditional Probability

The **unconditional** or **prior probability** associated with a proposition a is the degree of belief accorded to it in the absence of any other information; it is written as P (a). For example, if the **prior probability** that I have a cavity is 0.1, then we would write

P (Cavity=true) = 0.1 or P(Cavity) = 0.1.

It is important to remember that P (a) can be used only when there is no other information. **As soon as some new information is known, we must reason with the conditional probability** of a given that new information.

Joint Probability Distribution

Given an application domain in which we have determined a sufficient set of random variables to encode all of the relevant information about that domain, we can completely specify all of the possible probabilistic information by constructing the **full joint probability distribution**, P (V_1=v_1, V_2=v_2, ..., V_n=v_n), which assigns probabilities to all possible combinations of values to all random variables.

For example, consider a domain described by three Boolean random variables, Bird, Flier, and Young. Then we can enumerate a table showing all possible interpretations and associated probabilities:

Bird	Flier	Young	Probability
T	T	T	0.0
T	T	F	0.2
T	F	T	0.04
T	F	F	0.01
F	T	T	0.01
F	T	F	0.01
F	F	T	0.23
F	F	F	0.5

Notice that there are 8 rows in the above table representing the fact that there are 2^3 ways to assign values to the three Boolean variables. More generally, with *n* Boolean variables the table will be of size 2^n. And if *n* variables each had *k* possible values, then the table would be size k^n.

Also notice that the sum of the probabilities in the right column must equal 1 since we know that the set of all possible values for each variable are known. This means that for *n* Boolean random variables, the table has 2^n-1 values that must be determined to completely fill in the table.

If all of the probabilities are known for a full joint probability distribution table, then we can compute *any* probabilistic statement about the domain. For example, using the table above, we can compute

P (Bird=T) = P (B) = 0.0 + 0.2 + 0.04 + 0.01 = 0.25

P (Bird=T, Flier=F) = P (B, ~F) = P(B, ~F, Y) + F(B, ~F, ~Y) = 0.04 + 0.01 = 0.05

Conditional Probabilities

Conditional probabilities are keys for reasoning because they formalize the process of accumulating evidence and updating probabilities based on new evidence. For example, if we know there is a 4% chance of a person having a cavity, we can represent this as the **prior** (aka unconditional) probability P(Cavity)=0.04. Say that person now has a symptom of a tooth-ache; we'd like to know what the **conditional** or **posterior** probability of a Cavity given this new evidence is. That is, compute P (Cavity | Toothache).

If P (A|B) = 1, this is equivalent to the sentence in Propositional Logic B => A. Similarly, if P (A|B) =0.9, then this is like saying B => A with 90% certainty. In other words, we've made implication fuzzy because it's not absolutely certain.

Given several measurements and other "evidence", E1, ..., Ek, we will formulate queries as P (Q | E1, E2, ..., Ek) meaning "what is the degree of belief that Q is true given that we know E1, ..., Ek *and nothing else*."

Conditional probability is defined as:

$P(A|B) = P(A \wedge B)/P(B) = P(A,B)/P(B)$, where $P(B) > 0$.

This equation can be written as

$P(A \wedge B)=P(A|B)P(B)$

This is called the product rule.

Bayes’ Rule and Its Use

We have product rules as:

$P(A \wedge B)=P(A|B)P(B)$

$P(A \wedge B)=P(B|A)P(B)$

Then equating these two equations we get,

$PP(B|A) = P(A|B)\, P(B)/P(A)$

This equation is called Bayes’ rule also Bayes’ law or theorem. This simple equation underlines all modern AI systems for probabilistic inference.

Using Bayes' Rule

Bayes' Rule is the basis for probabilistic reasoning. Bayes’ rule allows unknown probabilities to be computed from known conditional probabilities. It requires three terms—a conditional probability and two unconditional probabilities—just to compute one conditional probability. Bayes’ rule is useful in practice because there are many cases where we do have good probability estimates for these numbers and need to compute the fourth.

For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say 5o% of the time. The doctor also knows some unconditional facts: the prio probabitlity that a patient has meningitis is 1/50,000 and the prior probability that any patient has a stiff neck is 20. Letting s be the proposition that the patient has a stiff neck and m be the propositional that the patient has meningitis, we have

$P(s|m) = 0.5$

$P(m) = 1/50{,}000$

$P(s) = 1/20$

$P(m|s) = P(s|m)\, P(m)/P(s) = (0.5*1/50{,}000)/(1/20) = 0.0002$

That is, we expect only 1 in 5000 patient with a stiff neck to have meningitis.

Combining Multiple Evidence using Bayes' Rule

Generalizing Bayes' Rule for two pieces of evidence, B and C, we get:

P (A|B, C) = ((P (A) P (B, C | A))/ P (B, C)

= P (A) * [P (B|A)/P (B)] * [P (C | A, B)/P (C|B)]

Again, this shows how the conditional probability of A is updated given B and C. The problem is that it may be hard in general to obtain or compute P(C | A, B). But this difficulty is circumvented if we know evidence B and C are conditionally independent or unconditionally independent.

Bayesian Networks (aka Belief Networks)

Bayesian Networks, also known as Bayes Nets, Belief Nets, Causal Nets, and Probability Nets, are a space-efficient data structure for encoding all of the information in the **full joint probability distribution** for the set of random variables defining a domain. That is, from the Bayesian Net one can compute any value in the full joint probability distribution of the set of random variables.

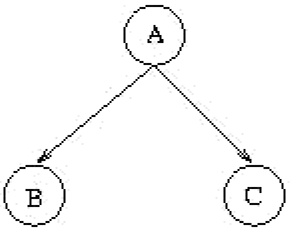
- Represents all of the direct causal relationships between variables
- Intuitively, to construct a Bayesian net for a given set of variables, draw arcs from cause variables to immediate effects.
- Space efficient because it exploits the fact that in many real-world problem domains the dependencies between variables are generally local, so there are a lot of conditionally independent variables
- Captures both qualitative and quantitative relationships between variables
- Can be used to reason
- Forward (top-down) from causes to effects -- **predictive reasoning** (aka **causal reasoning**)
- Backward (bottom-up) from effects to causes -- **diagnostic reasoning**

Formally, a Bayesian Net is a **directed, acyclic graph (DAG)**, where there is a node for each random variable and a directed arc from A to B whenever A is a direct causal influence on B. Thus the arcs represent direct causal relationships and the nodes represent states of affairs. The occurrence of A provides support for B, and vice versa. The backward influence is call "diagnostic" or "evidential" support for A due to the occurrence of B.

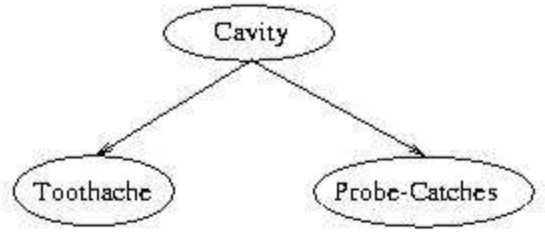
Each node A in a net is conditionally independent of any subset of nodes that are not descendants of A given the parents of A.

Net Topology Reflects Conditional Independence Assumptions

Conditional independence defines local net structure. For example, if B and C are conditionally independent given A, then by definition P(C|A,B) = P(C|A) and, symmetrically, P(B|A,C) = P(B|A). Intuitively, think of A as the direct cause of both B and C. In a Bayesian Net this will be represented by the local structure:



For example, in the dentist example in the textbook, having a Cavity causes both a Toothache and the dental probe to Catch, but these two events are conditionally independent given Cavity. That is, if we know nothing about whether or not someone has a Cavity, then Toothache and Catch are dependent. But as soon as we definitely know the person has a cavity or not, then knowing that the person has a Toothache as well has no effect on whether Catch is true. This conditional independence relationship will be reflected in the Bayesian Net topology as:



In general, we will construct the net so that given its parents, a node is conditionally independent of the rest of the net variables. That is,

P(X1=x1, ..., Xn=xn) = P(xi | Parents(Xi)) * ... * P(xn | Parents(Xn))

Hence, we don't need the full joint probability distribution, only conditionals relative to the parent variables.

Example

Consider the problem domain in which when I go home I want to know if someone in my family is home before I go in. Let's say I know the following information: (1) Why my wife leaves the house, she often (but not always) turns on the outside light. (She also sometimes turns the light on when she's expecting a guest.) (2) When nobody is home, the dog is often left outside. (3) If the dog has bowel-troubles, it is also often left outside. (4) If the dog is outside, I will probably hear it barking (though it might not bark, or I might hear a different dog barking and think it's my dog). Given this information, define the following five Boolean random variables:

O: Everyone is Out of the house

L: The Light is on

D: The Dog is outside

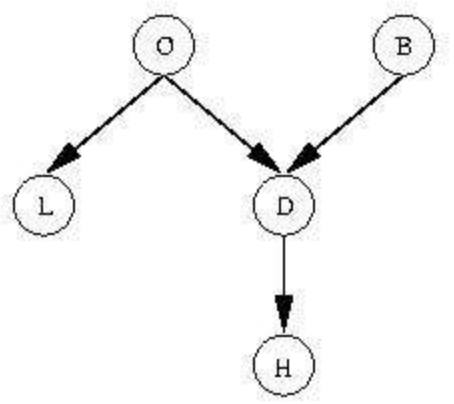
B: The dog has Bowel troubles

H: I can Hear the dog barking

From this information, the following direct causal influences seem appropriate:

- H is only directly influenced by D. Hence H is conditionally independent of L, O and B given D.
- D is only directly influenced by O and B. Hence D is conditionally independent of L given O and B.
- L is only directly influenced by O. Hence L is conditionally independent of D, H and B given O. and
- B is independent.

Based on the above, the following is a Bayesian Net that represents these direct causal relationships (though it is important to note that these causal connections are not absolute, i.e., they are not implications):

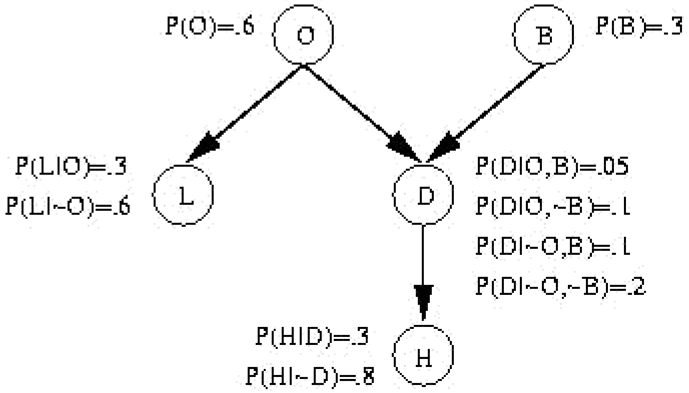


Next, the following quantitative information is added to the net; this information is usually given by an expert or determined empirically from training data.

For each root node (i.e., node without any parents), the prior probability of the random variable associated with the node is determined and stored there

For each non-root node, the conditional probabilities of the node's variable given all possible combinations of its immediate parent nodes are determined. This results in a **conditional probability table** (CPT) at each non-root node.

Doing this for the above example, we get the following Bayesian Net:

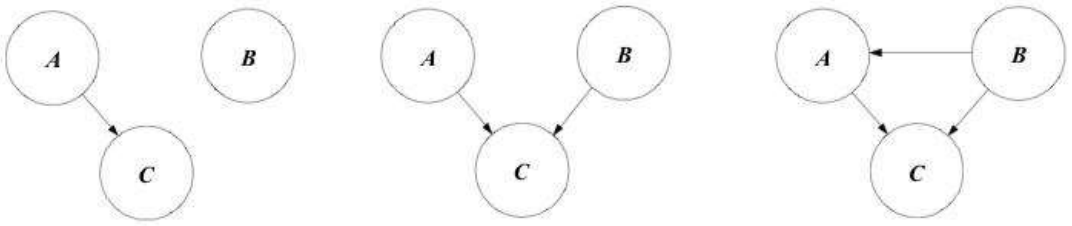


Notice that in this example, a total of 10 probabilities are computed and stored in the net, whereas the full joint probability distribution would require a table containing 2^5 = 32 probabilities. The reduction is due to the conditional independence of many variables.

Two variables that are not directly connected by an arc can still affect each other. For example, B and H are *not* (unconditionally) independent, but H does not directly depend on B.

Given a Bayesian Net, we can easily read off the conditional independence relations that are represented. Specifically, **each node, V, is conditionally independent of all nodes that are not descendants of V, given V's parents**. For example, in the above example H is conditionally independent of B, O, and L given D. So, P(H | B,D,O,L) = P(H | D).

The figure below gives three simple examples of qualitatively different probability relationships among three random variables.



$P(A,B,C) = P(C|A)P(A)$ $P(A,B,C) = P(C|A,B)P(A)P(B)$ $P(A,B,C) = P(C|A,B)P(A|B)P(B)$

Limitations of Probabilistic Reasoning with Bayesian Networks

Bayesian Networks have received praise for being a powerful tool for performing probabilistic inference, but they do have some limitations that impede their application to complex problems. As the technique grew in popularity, Bayesian Network's limitations became increasingly apparent. One of the most important limitations for it to be applied in the context of PR-OWL is the fact that, although a powerful tool, BNs are not expressive enough for many real-world applications. More specifically, Bayesian Networks assume a simple attribute-value representation – that is, each problem instance involves reasoning about the same fixed number of attributes, with only the evidence values changing from problem instance to problem instance.

This type of representation is inadequate for many problems of practical importance. Many domains require reasoning about varying numbers of related entities of different types, where the numbers, types and relationships among entities usually cannot be specified in advance and may have uncertainty in their own definitions. As will be demonstrated below, Bayesian networks are insufficiently expressive for such problems.

Game Playing

Game playing has been a major topic of AI since the very beginning. Beside the attraction of the topic to people, it is also because it’s close relation to "intelligence", and its well-defined states and rules. Games are well-defined problems that are generally interpreted as requiring intelligence to play well and introduce uncertainty since opponent’s moves cannot be determined in advance.

The most common used AI technique in game is search. In other problem-solving activities, state change is solely caused by the action of the agent. However, in Multi-agent games, it also depends on the actions of other agents who usually have different goals.

A special situation that has been studied most is "two-person zero-sum game", where the two players have exactly opposite goals. (Not all competitions are zero-sum)

There are *perfect information games* (such as Chess and Go) and *imperfect information games* (such as Bridge and games where a dice is used). Given sufficient time and space, usually an optimum solution can be obtained for the former by exhaustive search, though not for the latter. However, for most interesting games, such a solution is usually too inefficient to be practically used.

Search spaces for a game can be very large. For chess:

- Branching factor: 35
- Depth: 50 moves each player
- Search tree: 35_{100} nodes (~10_{40} legal positions)

The Mini-Max Procedure

For two-agent zero-sum perfect-information game, if the two players take turn to move, the *minimax procedure* can solve the problem given sufficient computational resources. This algorithm assumes that each player takes the best option in each step.

First, we distinguish two types of nodes, MAX and MIN, in the state graph, determined by the depth of the search tree.

Minimax procedure: starting from the leaves of the tree (with final scores with respect to one player, MAX), and go backwards towards the root.

At each step, one player (MAX) takes the action that leads to the highest score, while the other player (MIN) takes the action that leads to the lowest score.

All nodes in the tree will all be scored, and the path from root to the actual result is the one on which all nodes have the same score.

Example:

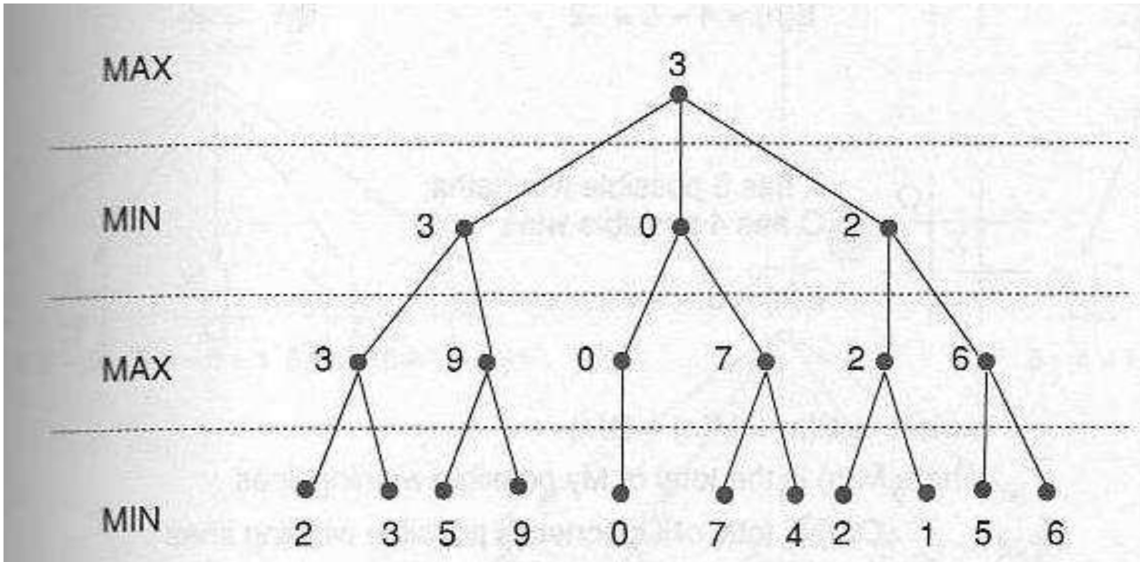
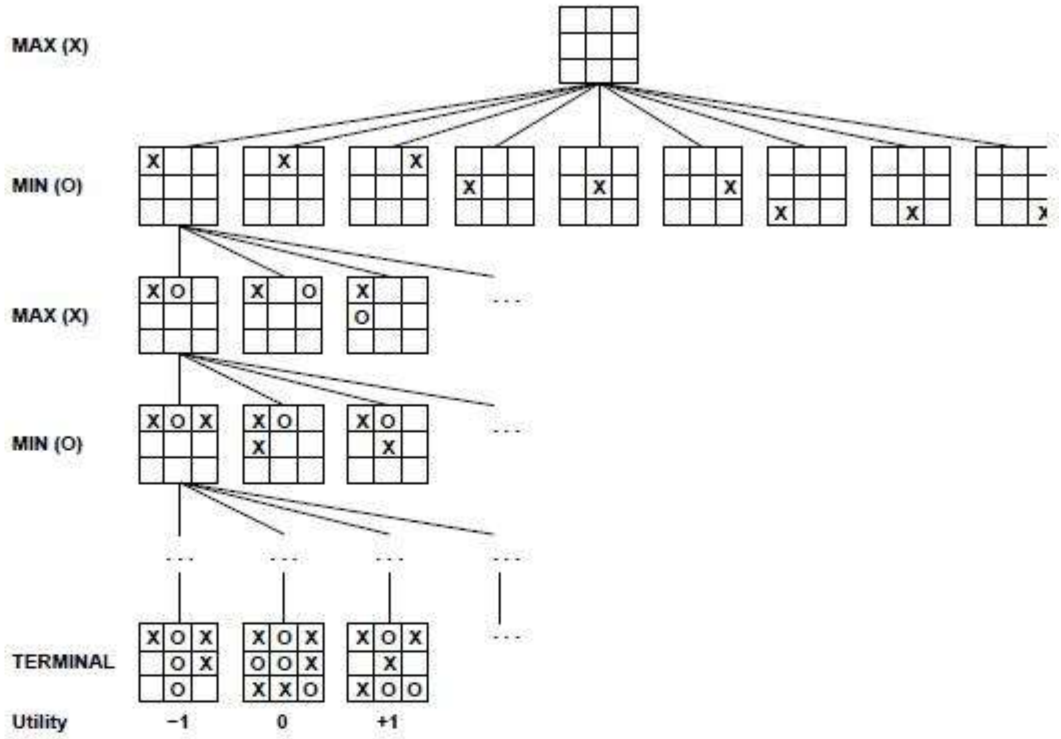


Figure 4.21 Minimax to a hypothetical state space. Leafstates show heuristic values; internal states show backed-up values.

Because of computational resources limitation, the search depth is usually restricted to a constant, and estimated scores (generated by a heuristic function) will replace the actual score in the above procedure.

Example: Tic-tac-toe, with the difference of possible win paths as the henristic function.



3. The Alpha-Beta Procedure

Very often, the game graph does not need to be fully explored using Minimax. Based on explored children's score, inequity can be set up for nodes whose children haven't been exhaustively explored. Under certain conditions, some branches of the tree can be ignored without changing the final score of the root.

In Alpha-Beta Procedure, each MAX node has an *alpha* value, which never decreases; each MIN node has a *beta* value, which never increases. These values are set and updated when the value of a child is obtained. Search is depth-first, and stops at any MIN node whose *beta* value is smaller than or equal to the *alpha* value of its parent, as well as at any MAX node whose *alpha* value is greater than or equal to the *beta* value of its parent.

Concept Learning

Concept learning also refers to a learning task in which a human or machine learner is trained to classify objects by being shown a set of example objects along with their class labels. The learner will simplify what has been observed in an example. This simplified version of what has been learned will then be applied to future examples.

It can be defined as the search for and listing of attributes that can be used to distinguish exemplars from non exemplars of various categories." More simply put, concepts are the mental categories that help us classify objects, events, or ideas and each object, event, or idea has a set of common relevant features. Thus, concept learning is a strategy which requires a learner to compare and contrast groups or categories that contain concept-relevant features with groups or categories that do not contain concept-relevant features.

More to the point, Concept learning is the classification of objects, events, or ideas based on similar characteristics. The basis for using concept learning is to simplify the learning process for the learner. The thought is that the learner will be able to digest several smaller steps easier than a single large step. The concepts can be built upon once understood, so the learner can more easily understand more complex material later in the instruction.

Concept learning ranges in simplicity and complexity because learning takes place over many areas. When a concept is more difficult, it will be less likely that the learner will be able to simplify, and therefore they will be less likely to learn.

The basis for using concept learning is to simplify the learning process for the learner. The thought is that the learner will be able to digest several smaller steps easier than a single large step (Joshi, 2010). The concepts can be built upon once understood, so the learner can more easily understand more complex material later in the instruction.

Winston Learning

Winston (1975) described a **Blocks World Learning** program. This program operated in a simple blocks domain. The goal is to construct representation of the definition of concepts in the block domain.

- An early structural concept learning program.
- This program operates in a simple blocks world domain.
- Its goal was to construct representations of the definitions of concepts in blocks domain.
- For example, it learned the concepts House, Tent and Arch.
- A near miss is an object that is not an instance of the concept in question but that is very similar to such instances.

Basic approach of Winston's program

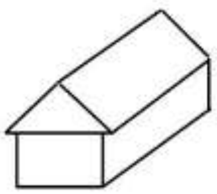
1. Begin with a structural description of one known instance of the concept. Call that description the concept definition.
2. Examine descriptions of other known instances of the concepts. Generalize th definition to include them.

3. Examine the descriptions of near misses of the concept. Restrict the definition to exclude these.

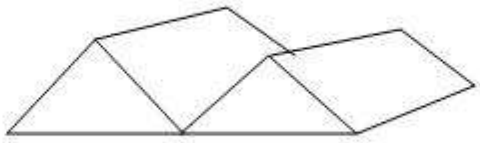
The both steps 2 and 3 rely heavily on comparison process by which similarities and differences between structures can be detected.

Example: The concepts House or Tent, Arch:

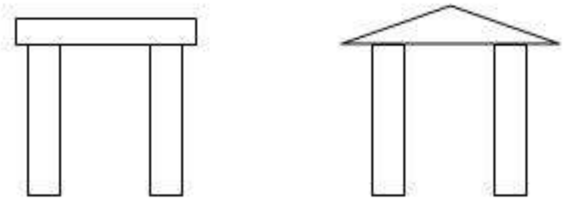
House: break (rectangular block) with a wedge (triangular block) suitably placed on top of it.



Tent: two wedges touching side by side.

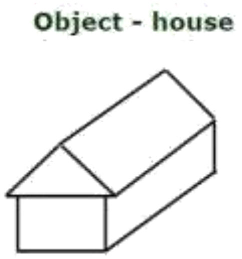


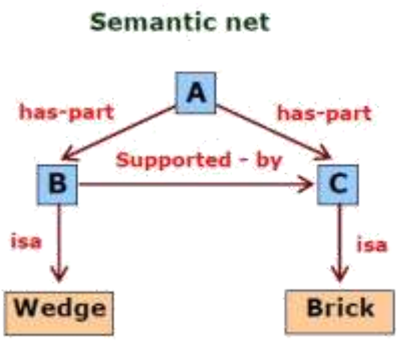
Arch: two non-touching bricks supporting a third wedge or brick.



The program for each concept is learnt through near miss. A near miss is an object that is not an instance of the concept but a very similar to such instances.

The program uses procedures to analyze the drawing and construct a semantic net representation. An example of such semantic net for the house is shown below:





Node A represents entire structure (i.e. house) which is composed of two parts: node B (a wedge) and node C (a brick). Links in the semantic network include **supported-by, has-part** and **is-a.**

Finally, Winston viewed the learning task as one that involved:

- *representing* the current example;
- *predicting* from the representation of the current hypothesis whether or not the current instance exemplifies the concept;
    - ○ if *correct* then *retain (save)* the current hypothesis; ○ if *incorrect* then
        - *identify the differences* between the current example and the current concept;
        - *choose from this set of differences* and use the chosen differences to
            - *Generalize* the concept hypothesis if the instance was a positive instance. OR
            - *Specialize* the concept hypothesis if the instance was a negative instance

Genetic Algorithm

A **genetic algorithm (GA)** is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. *Genetic Algorithms* are a way of solving problems by mimicking the same processes the nature uses. They use the same combination of selection, recombination and mutation to evolve a solution to a problem.

Concisely stated, a genetic algorithm is a programming technique that mimics biological evolution as a problem-solving strategy. Given a specific problem to solve, the input to the GA is a set of potential solutions to that problem, encoded in some fashion, and a metric called a *fitness function* that allows each candidate to be quantitatively evaluated. These candidates may be solutions already known to work, with the aim of the GA being to improve them, but more often they are generated at random.

The GA then evaluates each candidate according to the fitness function. In a pool of randomly generated candidates, of course, most will not work at all, and these will be deleted. However, purely by chance, a few may hold promise - they may show activity, even if only weak and imperfect activity, toward solving the problem.

These promising candidates are kept and allowed to reproduce. Multiple copies are made of them, but the copies are not perfect; random changes are introduced during the copying process. These digital offspring then go on to the next generation, forming a new pool of candidate solutions, and are subjected to a second round of fitness evaluation. Those candidate solutions which were worsened, or made no better, by the changes to their code are again deleted; but again, purely by chance, the random variations introduced into the population may have improved some individuals, making them into better, more complete or more efficient solutions to the problem at hand. Again these winning individuals are selected and copied over into the next generation with random changes, and the process repeats. The expectation is that the average fitness of the population will increase each round, and so by repeating this process for hundreds or thousands of rounds, very good solutions to the problem can be discovered.

As astonishing and counterintuitive as it may seem to some, genetic algorithms have proven to be an enormously powerful and successful problem-solving strategy, dramatically demonstrating the power of evolutionary principles. Genetic algorithms have been used in a wide variety of fields to evolve solutions to problems as difficult as or more difficult than those faced by human designers. Moreover, the solutions they come up with are often more efficient, more elegant, or more complex than anything comparable a human engineer would produce. In some cases, genetic algorithms have come up with solutions that baffle the programmers who wrote the algorithms in the first place!

How it works

GA was introduced as a computational analogy of adaptive systems. They are modeled loosely on the principles of the evolution via natural selection, employing a population of individuals that undergo selection in the presence of variation-inducing operators such as **mutation** and **recombination** (**crossover**). A **fitness function** is used to evaluate individuals, and reproductive success varies with fitness.

The Algorithms

1. Randomly generate an initial population M(0)
2. Compute and save the fitness u(m) for each individual m in the current population M(t)
3. Define selection probabilities p(m) for each individual m in M(t) so that p(m) is proportional to u(m)
4. Generate M(t+1) by probabilistically selecting individuals from M(t) to produce offspring via genetic operators
5. Repeat step 2 until satisfying solution is obtained.

The paradigm of GAs described above is usually the one applied to solving most of the problems presented to GAs. Though it might not find the best solution, more often than not, it would come up with a partially optimal solution.

Methods of representation

Before a genetic algorithm can be put to work on any problem, a method is needed to encode potential solutions to that problem in a form that a computer can process. One common approach is to encode solutions as binary strings: sequences of 1's and 0's, where the digit at each position represents the value of some aspect of the solution.

Another, similar approach is to encode solutions as arrays of integers or decimal numbers, with each position again representing some particular aspect of the solution.

A third approach is to represent individuals in a GA as strings of letters, where each letter again stands for a specific aspect of the solution.

The virtue of all three of these methods is that they make it easy to define operators that cause the random changes in the selected candidates: flip a 0 to a 1 or vice versa, add or subtract from the value of a number by a randomly chosen amount, or change one letter to another.

Another strategy, developed principally by John Koza of Stanford University and called *genetic programming*, represents programs as branching data structures called trees. In this approach, random changes can be brought about by changing the operator or altering the value at a given node in the tree, or replacing one sub-tree with another.

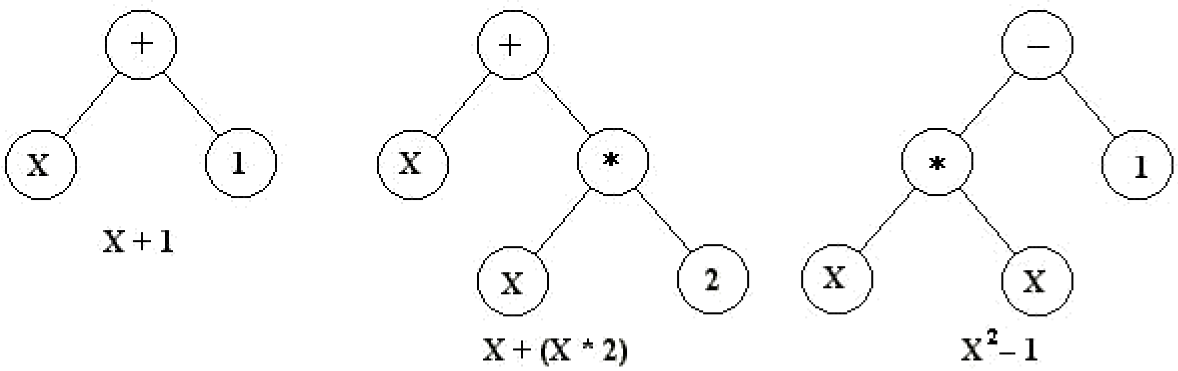


Figure 1: Three simple program trees of the kind normally used in genetic programming. The mathematical expression that each one represents is given underneath.

Methods of selection

There are many different techniques which a genetic algorithm can use to select the individuals to be copied over into the next generation, but listed below are some of the most common methods. Some of these methods are mutually exclusive, but others can be and often are used in combination.

Elitist selection: The most fit members of each generation are guaranteed to be selected. (Most GAs do not use pure elitism, but instead use a modified form where the single best, or a few of the best, individuals from each generation are copied into the next generation just in case nothing better turns up.)

Fitness-proportionate selection: More fit individuals are more likely, but not certain, to be selected.

Roulette-wheel selection: A form of fitness-proportionate selection in which the chance of an individual's being selected is proportional to the amount by which its fitness is greater or less than its competitors' fitness. (Conceptually, this can be represented as a game of roulette - each individual gets a slice of the wheel, but more fit ones get larger slices than less fit ones. The wheel is then spun, and whichever individual "owns" the section on which it lands each time is chosen.)

Scaling selection: As the average fitness of the population increases, the strength of the selective pressure also increases and the fitness function becomes more discriminating. This method can be helpful in making the best selection later on when all individuals have relatively high fitness and only small differences in fitness distinguish one from another.

Tournament selection: Subgroups of individuals are chosen from the larger population, and members of each subgroup compete against each other. Only one individual from each subgroup is chosen to reproduce.

Rank selection: Each individual in the population is assigned a numerical rank based on fitness, and selection is based on these ranking rather than absolute differences in fitness. The advantage of this method is that it can prevent very fit individuals from gaining dominance early at the expense of less fit ones, which would reduce the population's genetic diversity and might hinder attempts to find an acceptable solution.

Generational selection: The offspring of the individuals selected from each generation become the entire next generation. No individuals are retained between generations.

Steady-state selection: The offspring of the individuals selected from each generation go back into the pre-existing gene pool, replacing some of the less fit members of the previous generation. Some individuals are retained between generations.

Hierarchical selection: Individuals go through multiple rounds of selection each generation. Lower-level evaluations are faster and less discriminating, while those that survive to higher levels are evaluated more rigorously. The advantage of this method is that it reduces overall computation time by using faster, less selective evaluation to weed out the majority of individuals that show little or no promise, and only subjecting those who survive this initial test to more rigorous and more computationally expensive fitness evaluation.

Methods of change

Once selection has chosen fit individuals, they must be randomly altered in hopes of improving their fitness for the next generation. There are two basic strategies to accomplish this.

The first and simplest is called ***mutation*.** Just as mutation in living things changes one gene to another, so mutation in a genetic algorithm causes small alterations at single points in an individual's code.

The second method is called ***crossover*,** and entails choosing two individuals to swap segments of their code, producing artificial "offspring" that are combinations of their parents. This process is intended to simulate the analogous process of ***recombination*** that occurs to chromosomes during sexual reproduction.

Common forms of crossover include *single-point crossover*, in which a point of exchange is set at a random location in the two individuals' genomes, and one individual contributes all its code from before that point and the other contributes all its code from after that point to produce an offspring, and *uniform crossover*, in which the value at any given location in the offspring's genome is either the value of one parent's genome at that location or the value of the other parent's genome at that location, chosen with 50/50 probability.

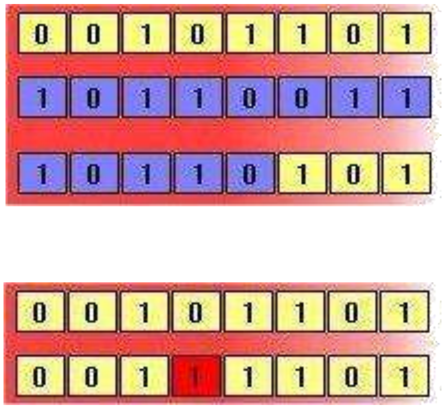


Figure 2: Crossover and mutation. The above diagrams illustrate the effect of each of these genetic operators on individuals in a population of 8-bit strings. The upper diagram shows two individuals undergoing single-point crossover; the point of exchange is set between the fifth and sixth positions in the genome, producing a new individual that is a hybrid of its progenitors. The second diagram shows an individual undergoing mutation at position 4, changing the 0 at that position in its genome to a 1.

Advantages

Nearly everyone can gain benefits from Genetic Algorithms, once he can encode solutions of a given problem to chromosomes in GA, and compare the relative performance (fitness) of solutions. An effective GA representation and meaningful fitness evaluation are the keys of the success in GA applications. The appeal of GAs comes from their simplicity and elegance as robust search algorithms as well as from their power to discover good solutions rapidly for difficult high-dimensional problems. GAs are useful and efficient when

- The search space is large, complex or poorly understood.
- Domain knowledge is scarce or expert knowledge is difficult to encode to narrow the search space.
- No mathematical analysis is available.
- Traditional search methods fail.

The advantage of the GA approach is the ease with which it can handle arbitrary kinds of constraints and objectives; all such things can be handled as weighted components of the fitness function, making it easy to adapt the GA scheduler to the particular requirements of a very wide range of possible overall objectives.

GAs have been used for problem-solving and for modeling. GAs are applied to many scientific, engineering problems, in business and entertainment, including:

- **Optimization:** GAs have been used in a wide variety of optimization tasks, including numerical optimization, and combinatorial optimization problems such as traveling salesman problem (TSP), circuit design, job shop scheduling and video & sound quality optimization.
- **Automatic Programming:** GAs have been used to evolve computer programs for specific tasks, and to design other computational structures, for example, cellular automata and sorting networks.
- **Machine and robot learning:** GAs have been used for many machine- learning applications, including classification and prediction, and protein structure prediction. GAs have also been used to design neural networks, to evolve rules for learning classifier systems or symbolic production systems, and to design and **control robots**.
- **Economic models:** GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.
- Immune system models: GAs have been used to model various aspects of the natural immune system, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
- **Ecological models**: GAs have been used to model ecological phenomena such as biological arms races, host-parasite co-evolutions, symbiosis and resource flow in ecologies.
- **Population genetics models**: GAs have been used to study questions in population genetics, such as "under what conditions will a gene for recombination be evolutionarily viable?"
- **Interactions between evolution and learning**: GAs have been used to study how individual learning and species evolution affect one another.
- **Models of social systems:** GAs have been used to study evolutionary aspects of social systems, such as the evolution of cooperation, the evolution of communication, and trail-following behavior in ants.

What are the strengths of GAs?

- **The first and most important point is that genetic algorithms are intrinsically parallel.**

  Most other algorithms are serial and can only explore the solution space to a problem in one direction at a time, and if the solution they discover turns out to be suboptimal, there is nothing to do but abandon all work previously completed and start over. However, since GAs have multiple offspring, they can explore the solution space in multiple directions at once. If one path turns out to be a dead end, they can easily eliminate it and continue work on more promising avenues, giving them a greater chance each run of finding the optimal solution.

- Due to the parallelism that allows them to implicitly evaluate many schema at once, genetic algorithms are particularly well-suited to solving problems where the space of all potential solutions is truly huge - too vast to search exhaustively in any reasonable amount of time.
- Another notable strength of genetic algorithms is that they perform well in problems for which the fitness landscape is complex - ones where the fitness function is discontinuous, noisy, changes over time, or has many local optima. Most practical problems have a vast solution space, impossible to search exhaustively; the challenge then becomes how to avoid the local optima - solutions that are better than all the others that are similar to them, but that are not as good as different ones elsewhere in the solution space. Many search algorithms can become trapped by local optima: if they reach the top of a hill on the fitness landscape, they will discover that no better solutions exist nearby and conclude that they have reached the best one, even though higher peaks exist elsewhere on the map.

  Evolutionary algorithms, on the other hand, have proven to be effective at escaping local optima and discovering the global optimum in even a very rugged and complex fitness landscape.
- Another area in which genetic algorithms excel is their ability to manipulate many parameters simultaneously
- Finally, one of the qualities of genetic algorithms which might at first appear to be a liability turns out to be one of their strengths: namely, GAs know nothing about the problems they are deployed to solve. Instead of using previously known domain-specific information to guide each step and making changes with a specific eye towards improvement, as human designers do, they are "blind watchmakers" (Dawkins 1996); they make *random* changes to their candidate solutions and then use the fitness function to determine whether those changes produce an improvement.

What are the limitations of GAs?

Although genetic algorithms have proven to be an efficient and powerful problem-solving strategy, they are not a panacea. GAs do have certain limitations; however, it will be shown that all of these can be overcome and none of them bear on the validity of biological evolution.

- The first, and most important, consideration in creating a genetic algorithm is defining a representation for the problem. The language used to specify candidate solutions must be robust; i.e., it must be able to tolerate random changes such that fatal errors or nonsense do not consistently result.
- The problem of how to write the fitness function must be carefully considered so that higher fitness is attainable and actually does equate to a better solution for the given problem. If the fitness function is chosen poorly or defined imprecisely, the genetic algorithm may be unable to find a solution to the problem, or may end up solving the wrong problem.
- In addition to making a good choice of fitness function, the other parameters of a GA - the size of the population, the rate of mutation and crossover, the type and strength of selection - must be also chosen with care. If the population size is too small, the genetic algorithm may not explore enough of the solution space to consistently find good solutions. If the rate of genetic change is too high or the selection scheme is chosen poorly, beneficial schema may be disrupted and the population may enter error catastrophe, changing too fast for selection to ever bring about convergence.
- One type of problem that genetic algorithms have difficulty dealing with are problems with "deceptive" fitness functions (Mitchell 1996, p.125), those where the locations of improved

points give misleading information about where the global optimum is likely to be found. For example, imagine a problem where the search space consisted of all eight-character binary strings, and the fitness of an individual was directly proportional to the number of 1s in it - i.e., 00000001 would be less fit than 00000011, which would be less fit than 00000111, and so on - with two exceptions: the string 11111111 turned out to have very low fitness, and the string 00000000 turned out to have very high fitness. In such a problem, a GA (as well as most other algorithms) would be no more likely to find the global optimum than random search.

- One well-known problem that can occur with a GA is known as *premature convergence*. If an individual that is more fit than most of its competitors emerges early on in the course of the run, it may reproduce so abundantly that it drives down the population's diversity too soon, leading the algorithm to converge on the local optimum that that individual represents rather than searching the fitness landscape thoroughly enough to find the global optimum (Forrest 1993, p. 876; Mitchell 1996, p. 167). This is an especially common problem in small populations, where even chance variations in reproduction rate may cause one genotype to become dominant over others.

Applications of GA

As the power of evolution gains increasingly widespread recognition, genetic algorithms have been used to tackle a broad variety of problems in an extremely diverse array of fields, clearly showing their power and their potential. Some of the field where the GA is used are:

- Acoustics
- Aerospace engineering
- Astronomy and astrophysics
- Chemistry
- Electrical engineering
- Financial markets
- Game playing
- Geophysics
- Materials engineering
- Mathematics and algorithms
- Military and law enforcement
- Molecular biology
- Pattern recognition and data mining
- Robotics
- Routing and scheduling
- Systems engineering

Inductive bias Learning

The inductive bias of a learning algorithm is the set of assumptions that the learner uses to predict outputs given inputs that it has not encountered.

In machine learning, one aims to construct algorithms that are able to *learn* to predict a certain target output. To achieve this, the learning algorithm is presented some training examples that demonstrate the intended relation of input and output values. ***Then the learner is supposed to approximate the correct output, even for examples that have not been shown during training. Without any additional assumptions, this task cannot be solved exactly since unseen situations might have an arbitrary output value.*** The kind of necessary assumptions about the nature of the target function are subsumed in the term *inductive bias*.

Inductive bias refers to your suspicion[1] that if the sun has risen for the last billion days in a row, then it may rise tomorrow as well. Since it is **logically possible** that the laws of physics will arbitrarily[2] cease to work and that the sun will *not* rise tomorrow, coming to this conclusion requires an inductively biased prior. This sort of bias is not a bad thing - without "inductive bias" you can't draw any conclusion at all from the data. It's just a different technical meaning attached to the same word.

A classical example of an inductive bias is **Occam's Razor**, assuming that the simplest consistent hypothesis about the target function is actually the best. Here *consistent* means that the hypothesis of the learner yields correct outputs for all of the examples that have been given to the algorithm.

Suppose that you see a swan for the first time, and it is white. It does not follow *logically* that the next swan you see must be white, but white seems like a better guess than any other color. A machine learning algorithm of the more rigid sort, if it sees a single white swan, may thereafter predict that any swan seen will be white. But this, of course, does not follow logically - though AIs of this sort are often misnamed "logical". For a purely logical reasoner to label the next swan white as a *deductive* conclusion, it would need an additional assumption: "All swans are the same color." This is a wonderful assumption to make if all swans are, *in reality,* the same color; otherwise, not so good. Tom Mitchell's *Machine Learning* defines the inductive bias of a machine learning algorithm as the assumptions that must be added to the observed data to transform the algorithm's outputs into logical deductions.

A **more general view of inductive bias** would identify it with a Bayesian's prior over sequences of observations. Consider the case of an urn filled with red and white balls, from which we are to sample without replacement. I might have prior information that the urn contains 5 red balls and 5 white balls. Or, I might have prior information that a random number was selected from a uniform distribution between 0 and 1, and this number was then used as a fixed probability to

[1] An impression that something might be the case

[2] In a random manner

independently generate a series of 10 balls. In either case, I will estimate a 50% probability that the first ball is red, a 50% probability that the second ball is red, etc., which you might foolishly think indicated the same prior belief. But, while the marginal probabilities on each round are equivalent, the probabilities over sequences are different. In the first case, if I see 3 red balls initially, I will estimate a probability of 2/7 that the next ball will be red. In the second case, if I see 3 red balls initially, I will estimate a 4/5 chance that the next ball will be red (by Laplace's Law of Succession, thus named because it was proved by Thomas Bayes). In both cases we refine our future guesses based on past data, but in opposite directions, which demonstrates the importance of prior information.

Suppose that your prior information about the urn is that a monkey tosses balls into the urn, selecting red balls with 1/4 probability and white balls with 3/4 probability, each ball selected independently. The urn contains 10 balls, and we sample without replacement. (E. T. Jaynes called this the "binomial monkey prior".) Now suppose that on the first three rounds, you see three red balls. What is the probability of seeing a red ball on the fourth round?

First, we calculate the prior probability that the monkey tossed 0 red balls and 10 white balls into the urn; then the prior probability that the monkey tossed 1 red ball and 9 white balls into the urn; and so on. Then we take our evidence (three red balls, sampled without replacement) and calculate the likelihood of seeing that evidence, conditioned on each of the possible urn contents. Then we update and normalize the posterior probability of the possible remaining urn contents. Then we average over the probability of drawing a red ball from each possible urn, weighted by that urn's posterior probability. And the answer is... *(scribbles frantically for quite some time)*... 1/4!

Of course it's 1/4. We specified that each ball was independently tossed into the urn, with a known 1/4 probability of being red. Imagine that the monkey is tossing the balls to you, one by one; if it tosses you a red ball on one round, that doesn't change the probability that it tosses you a red ball on the next round. When we withdraw one ball from the urn, it doesn't tell us anything about the other balls in the urn.

If you start out with a maximum-entropy prior, then you never learn anything, ever, no matter how much evidence you observe. You do not even learn anything wrong - you always remain as ignorant as you began.

The more inductive bias you have, the faster you learn to predict the future, but only if your inductive bias does in fact concentrate more probability into sequences of observations that actually occur. If your inductive bias concentrates probability into sequences that don't occur, this diverts probability mass from sequences that *do* occur, and you will learn more slowly, or not learn at all, or even - if you are unlucky enough - learn in the wrong direction.

Inductive biases can be probabilistically correct or probabilistically incorrect, and if they are correct, it is good to have as much of them as possible, and if they are incorrect, you are left worse off than if you had no inductive bias at all. Which is to say that inductive biases are like any other kind of belief; the true ones are good for you, the bad ones are worse than nothing. In contrast, statistical bias is always bad, period - you can trade it off against other ills, but it's never a good thing for itself. Statistical bias is a systematic direction in *errors*; inductive bias is a systematic direction in *belief revisions.*

As the example of maximum entropy demonstrates, without a direction to your belief revisions, you end up not revising your beliefs at all. No future prediction based on past experience follows as a matter of strict logical deduction. Which is to say: *All learning is induction, and all induction takes place through inductive bias.*

Learning paradigms

There are three major learning paradigms, each corresponding to a particular abstract learning task. These are

1. Supervised learning,
2. Unsupervised learning and
3. Reinforcement learning.

Supervised learning

In supervised learning, we are given a set of example pairs and the aim is to find a function in the allowed class of functions that matches the examples. In other words, we wish to *infer* the mapping implied by the data; the cost function is related to the mismatch between our mapping and the data and it implicitly contains prior knowledge about the problem domain.

A commonly used cost is the mean-squared error, which tries to minimize the average squared error between the network's output, f(x), and the target value y over all the example pairs. When one tries to minimize this cost using gradient descent for the class of neural networks called multilayer perceptrons, one obtains the common and well-known backpropagation algorithm for training neural networks.

Tasks that fall within the paradigm of supervised learning are pattern recognition (also known as classification) and regression (also known as function approximation). The supervised learning paradigm is also applicable to sequential data (e.g., for speech and gesture recognition). This can be thought of as learning with a "teacher," in the form of a function that provides continuous feedback on the quality of solutions obtained thus far.

Unsupervised learning

In unsupervised learning, some data is given and the cost function to be minimized, that can be any function of the data and the network's output, .

The cost function is dependent on the task (what we are trying to model) and our *a priori* assumptions (the implicit properties of our model, its parameters and the observed variables).

As a trivial example, consider the model , where is a constant and the cost . Minimizing this cost will give us a value of that is equal to the mean of the data. The cost function can be much more complicated. Its form depends on the application: for example, in compression it could be related to the mutual information between and , whereas in statistical modeling, it could be related to the posterior probability of the model given the data. (Note that in both of those examples those quantities would be maximized rather than minimized).

Tasks that fall within the paradigm of unsupervised learning are in general estimation problems; the applications include clustering, the estimation of statistical distributions, compression and filtering.

Reinforcement learning

In reinforcement learning, data are usually not given, but generated by an agent's interactions with the environment. At each point in time, the agent performs an action and the environment generates an observation and an instantaneous cost, according to some (usually unknown) dynamics. The aim is to discover a *policy* for selecting actions that minimizes some measure of a long-term cost; i.e., the expected cumulative cost. The environment's dynamics and the long-term cost for each policy are usually unknown, but can be estimated.

More formally, the environment is modeled as a Markov decision process (MDP) with states and actions with the following probability distributions: the instantaneous cost distribution, the observation distribution and the transition, while a policy is defined as conditional distribution over actions given the observations. Taken together, the two define a Markov chain (MC). The aim is to discover the policy that minimizes the cost; i.e., the MC for which the cost is minimal.

ANNs are frequently used in reinforcement learning as part of the overall algorithm. Dynamic programming has been coupled with ANNs (Neuro dynamic programming) by Bertsekas and Tsitsiklis[2] and applied to multi-dimensional nonlinear problems such as those involved in vehicle routing or natural resources management because of the ability of ANNs to mitigate losses of accuracy even when reducing the discretization grid density for numerically approximating the solution of the original control problems.

Tasks that fall within the paradigm of reinforcement learning are control problems, games and other sequential decision making tasks.

12.15 Supervised Learning

- Supervised learning networks learn by being presented with preclassified training data.
- The techniques we have discussed so far in this chapter use forms of supervised learning.
- Neural networks that use supervised learning learn by modifying the weights of the connections within their networks to more accurately classify the training data.
- In this way, neural networks are able to generalize extremely accurately in many situations from a set of training data to the full set of possible inputs.
- One of the most commonly used methods for supervised learning is backpropagation.

12.16 Unsupervised Learning

- Unsupervised learning methods learn without any human intervention.
- A good example of an unsupervised learning network is a **Kohonen map**.
- A Kohonen map is a neural network that is able to learn to classify a set of input data without being told what the classifications are and without being given any training data.
- This method is particularly useful in situations where data need to be classified, or clustered, into a set of classifications but where the classifications are not known in advance.
- For example, given a set of documents retrieved from the Internet (perhaps by an intelligent information agent), a Kohonen map could cluster similar documents together and automatically provide an indication of the distinct subjects that are covered by the documents.
- Another method for unsupervised learning in neural networks was proposed by Donald Hebb in 1949 and is known as Hebbian learning.
- Hebbian learning is based on the idea that if two neurons in a neural network are connected together, and they fire at the same time when a particular input is given to the network, then the connection between those two neurons should be strengthened.
- It seems likely that something not dissimilar from Hebbian learning takes place in the human brain when learning occurs.

12.17 Reinforcement Learning

- Classifier systems use a form of **reinforcement learning**.
- A system that uses reinforcement learning is given a positive reinforcement when it performs correctly and a negative reinforcement when it performs incorrectly.
- For example, a robotic agent might learn by reinforcement learning how to pick up an object. When it successfully picks up the object, it will receive a positive reinforcement.
- The information that is provided to the learning system when it performs its task correctly does not tell it *why* or *how* it performed it correctly, simply that it did.
- Some neural networks learn by reinforcement. The main difficulty with such methods is the problem of **credit assignment**. The classifier systems use a **bucket brigade algorithm** for deciding how to assign credit to the individual components of the system.
- Similar methods are used with neural networks to determine to which neurons to give credit when the network performs correctly and which to blame when it does not.

Introduction

An **expert system** is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, like an expert, and not by following the procedure of a developer as is the case in conventional programming.

An expert system has a unique structure, different from traditional **programs**. It is divided into two parts, one fixed, independent of the expert system: **the inference engine**, and one variable: the **knowledge base**. To run an expert system, the engine reasons about the knowledge base like a human.

The rule base or knowledge base

In expert system technology, the knowledge base is expressed with **natural language** rules IF ... THEN ... For example,

- "IF it is living THEN it is mortal"
- "IF his age = known THEN his year of birth = date of today - his age in years"
- "*IF the identity of the germ is not known with certainty AND the germ is gram-positive AND the morphology of the organism is "rod" AND the germ is aerobic THEN there is a strong probability (0.8) that the germ is of type enterobacteriacae.*"

Rules express the knowledge to be exploited by the expert system. There exist other formulations of rules, which are not in everyday language, understandable only to computer scientists. Each rule style is adapted to an engine style. The whole problem of expert systems is to collect this knowledge, usually unconscious, from the experts.

The inference engine

The inference engine is a computer program designed to produce reasoning on rules. In order to produce reasoning, it is based on **logic.** There are several kinds of logic: **propositional logic, first order predicates logic** or more, **epistemic logic, modal logic, temporal logic, fuzzy logic,** etc. With logic, the engine is able to generate new information from the knowledge contained in the rule base and data to be processed.

Thus, an expert system is a computer program, with a set of ***rules*** encapsulating knowledge about a particular problem domain (i.e., medicine, chemistry, finance, flight, et cetera). These rules prescribe actions to take when certain conditions hold, and define the effect of the action on deductions or data. The expert system, seemingly, uses reasoning capabilities to reach conclusions or to perform analytical tasks. Expert systems that record the knowledge needed to solve a problem as a collection of ***rules*** stored in a ***knowledge-base*** are called ***rule-based systems***.

One of the early applications, **MYCIN,** was created to help physicians diagnose and treat bacterial infections. Expert systems have been used to analyze geophysical data in our search for petroleum and metal deposits (e.g., PROSPECTOR). They are used by the investments, banking, and telecommunications industries. They are essential in robotics, natural language processing, theorem proving, and the intelligent retrieval of information from databases. They are used in many other human endeavors which might be considered more practical. Rule-based systems have been used to monitor and control traffic, to aid in the development of flight systems, and by the federal government to prepare budgets.

Expert System Architecture

Figure 1 shows the most important modules that make up a rule-based expert system. The figure shows the different components of a rule based expert system. The components in a expert system includes:

1. User Interface
2. Explanation System
3. Inference Engine
4. Knowledge Base-Editor
5. Case-Specific Data
6. Knowledge Base

The user interacts with the system through a ***user interface*** which may use menus, natural language or any other style of interaction.

Then an ***inference engine*** is used to reason with both the *expert knowledge* and data specific to the particular problem being solved.

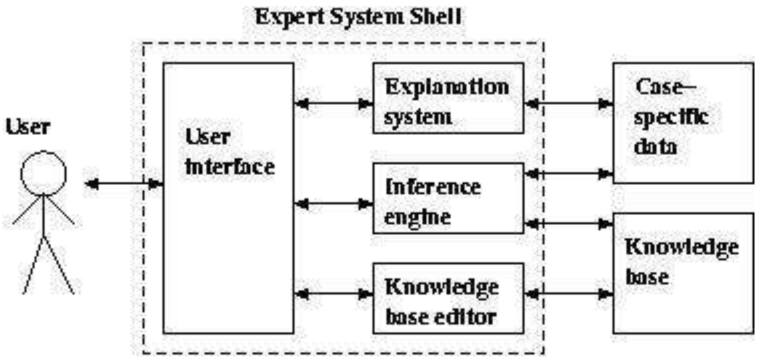


Figure 1: Expert System Architecture

The expert knowledge will typically be in the form of a set of IF-THEN rules. The ***case (problem) specific data*** includes both data provided by the user and partial conclusions based on this data. In a simple **forward chaining rule-based system** the ***case specific data*** will be the elements in *working memory*.

An ***explanation subsystem*** allows the program to explain its reasoning to the user.

A ***knowledge base editor*** helps the expert or knowledge engineer to easily update and check the knowledge base.

One important feature of expert systems is the way they (usually) separate domain specific knowledge from more general purpose reasoning and representation techniques. The general purpose bit (in the dotted box in the figure) is referred to as an ***expert system shell*.** As we see in the figure, the shell will provide the inference engine (and knowledge representation scheme), a **user interface**, an **explanation system** and sometimes **a knowledge base editor**. Given a new kind of problem to solve (say, car design), we can usually find a shell that provides the right sort of support for that problem, so all we need to do is provide the expert knowledge. There are numerous commercial expert system shells, each one appropriate for a slightly different range of problems. (Expert systems work in industry includes both writing expert system shells and writing expert systems using shells.) Using shells to write expert systems generally greatly reduces the cost and time of development (compared with writing the expert system from scratch).

Advantages

(Assignment)

Disadvantages

(Assignment)

Applications

(Assignment)

Knowledge Acquisition

Knowledge acquisition is the process of extracting, structuring and organizing knowledge from one source, usually human experts, so it can be used in software such as an ES. Knowledge acquisition includes the elicitation, collection, analysis, modeling and validation of knowledge for knowledge engineering and knowledge management projects.

Knowledge acquisition is the general term used for the process of developing a computational problem-solving model, specifically a program to be used in some consultative or advisory role. Such programs are generally called "expert systems". The knowledge acquisition process actually has several steps. The most important are:

- selecting a problem to be solved by the program,
- interviewing an expert,
- codifying the knowledge in some representation language, and
- Refining the knowledge base by testing it and extending its capability.

There are three main topic areas central to knowledge acquisition that requires consideration in all ES (Expert System) projects.

- First, the domain must be evaluated to determine if the type of knowledge in the domain is suitable for an ES.
- Second, the source of expertise must be identified and evaluated to ensure that the specific level of knowledge required by the project is provided.
- Third, if the major source of expertise is a person, the specific knowledge acquisition techniques and participants need to be identified.

Issues in Knowledge Acquisition

Some of the most important issues in knowledge acquisition are as follows:

- Most knowledge is in the heads of experts
- Experts have vast amounts of knowledge
- Experts have a lot of tacit knowledge
    - They don't know all that they know and use
    - Tacit knowledge is hard (impossible) to describe
- Experts are very busy and valuable people
- Each expert doesn't know everything
- Knowledge has a "shelf life"

Requirements for KA Techniques

Because of these issues, techniques are required which:

- Take experts off the job for short time periods
- Allow non-experts to understand the knowledge

- Focus on the essential knowledge
- Can capture tacit knowledge
- Allow knowledge to be collated from different experts
- Allow knowledge to be validated and maintained

Knowledge Acquisition Technique

At the heart of the process is the interview. The heuristic model of the domain is usually extracted through a series of intense, systematic interviews, usually extending over a period of many months. Note that this assumes the expert and the knowledge engineer are not the same person. It is generally best that the expert and the knowledge engineer not be the same person since the deeper the experts' knowledge, the less able they are in describing their logic. Furthermore, in their efforts to describe their procedures, experts tend to rationalize their knowledge and this can be misleading.

General suggestions about the knowledge acquisition process are summarized in rough sequential order below:

1. Observe the person solving real problems.
2. Through discussions, identify the kinds of data, knowledge and procedures required to solve different types of problems.
3. Build scenarios with the expert that can be associated with different problem types.
4. Have the expert solve a series of problems verbally and ask the rationale behind each step.
5. Develop rules based on the interviews and solve the problems with them.
6. Have the expert review the rules and the general problem solving procedure.
7. Compare the responses of outside experts to a set of scenarios obtained from the project's expert and the ES.

Note that most of these procedures require a close working relationship between the knowledge engineer and the expert.

Development of Expert System

The remainder of this section will describe a range of knowledge acquisition techniques that have been successfully used in the development of ES.

Operational Goals

After an evaluation of the problem domain shows that an ES solution is appropriate and feasible, then realistic goals for the project can be formulated. An ES's operational goals should define exactly what level of expertise its final product should be able to deliver, who the expected user is and how the product is to be delivered. If participants do not have a shared concept of the project's operational goals, knowledge acquisition is hampered.

Pre-training

Pre-training the knowledge engineer about the domain can be important. In the past, knowledge engineers have often been unfamiliar with the domain. As a result, the development process was greatly hindered. If a knowledge engineer has limited knowledge of the problem domain, then pre-training in the domain is very important and can significantly boost the early development of the ES.

Knowledge Document

Once development begins on the knowledge base, the process should be well documented. In addition to tutorial a document, a knowledge document that succinctly states the project's current knowledge base should be kept. Conventions should be established for the document such as keeping the rules in quasi-English format, using standard domain jargon, giving descriptive names to the rules and including supplementary, explanatory clauses with each rule. The rules should be grouped into natural subdivisions and the entire document should be kept current.

Scenarios

An early goal of knowledge acquisition should be the development of a series of well developed scenarios that fully describe the kinds of procedures that the expert goes through in arriving at different solutions. If reasonably complete case studies do not exist, then one goal of pre-training should be to become so familiar with the domain that the interviewer can compose realistic scenarios. Anecdotal stories that can be developed into scenarios are especially useful because they are often examples of unusual interactions at the edges of the domain. Familiarity with several realistic scenarios can be essential to understanding the expert in early interviews and the key to structuring later interviews. Finally, they are ultimately necessary for validation of the system.

Interviews

Experts are usually busy people and interviews held in the expert's work environment are likely to be interrupted. To maximize access to the expert and minimize interruptions it can be helpful to hold meetings away from the expert's workplace. Another possibility is to hold meetings after work hours and on weekends. At least initially, audiotape recordings ought to be made of the interviews because often times notes taken during an interview can be incomplete or suggest inconsistencies that can be clarified by listening to the tape. The knowledge engineer should also be alert to fatigue and limit interviews accordingly.

In early interviews, the format should be unstructured in the sense that discussion can take its own course. The knowledge engineer should resist the temptation to impose personal biases on what the expert is saying. During early discussions, experts are often asked to describe the tasks encountered in the domain and to go through example tasks explaining each step. An alternative or supplemental approach is simply to observe the expert on the job solving problems without interruption or to have the expert talk aloud during performance of a task with or without interruption. These procedures are variations of protocol analysis and are useful only with experts that primarily use verbal thought processes to solve domain problems.

For shorter term projects, initial interviews can be formalized to simplify rapid prototyping. One such technique is a **structured interview** in which the expert is asked to list the variables considered when making a decision. Next the expert is asked to list possible outcomes (solutions) from decision making. Finally, the expert is asked to connect variables to one another, solutions to one another and variables to solutions through rules.

A second technique is called **twenty questions.** With this technique, the knowledge engineer develops several scenarios typical of the domain before the interview. At the beginning of the interview, the expert asks whatever questions are necessary to understand the scenario well enough to determine the solution. Once the expert begins the questions, the expert is asked to explain why each question is asked. When the interviewer perceives a rule, he interrupts and restates the rule to ensure that it is correct.

A third technique is **card sorting**. In this procedure, the knowledge engineer prepares a stack of cards with typical solutions to problems in the domain. The expert is asked to sort the cards according to some characteristic important to finding solutions to the problem. After each sort, the expert is asked to identify the sorting variable. After each sort, the expert is asked to repeat the process based on another variable. Note that this technique is usually not as effective as the 2 previous.

In large projects, later interviews cannot be expected to be as productive as early interviews. Typically, later interviews should become increasingly structured and follow a cyclical pattern where bits of knowledge are elicited, documented and tested. During this phase of knowledge acquisition, the interviewer must begin methodically to uncover the more subtle aspects of the knowledge. Typically, this process is based on scenarios. By modifying the scenarios in different ways, the interviewer can probe the expert's sensitivity.

During interviews, it may be helpful to work at a whiteboard to flexibly record and order the exact phraseology of rules or other representations. It may also be helpful to establish recording conventions for use such as color coding different aspects of a rule and using flags to note and defer consideration of significant but peripheral details.

Structured interviews should direct the course of a meeting to accomplish specific goals defined in advance. For instance, once a prototypic knowledge base is developed, the expert can be asked to evaluate it line by line. Other less obvious structures can be imposed on interviews, such as asking the expert to perform a task with limited information or during a limited period of time. Even these structured interviews can deviate from the session's intended goals. Sometimes such deviations show subtleties in the expert's procedures and at other times the interview simply becomes sidetracked, requiring the knowledge engineer to redirect the session.

Questionnaires

When specific information is needed, a questionnaire can sometimes be used effectively. Questionnaires are generally used in combination with other techniques such as interviews.

Decision Trees

Decision trees are widely recognized to be useful tools for the knowledge engineer in prototyping knowledge representations. In addition, they can be useful in knowledge acquisition on several different levels. Some knowledge engineers have found that experts can more readily relate to decision trees than rules.

Rule Development

Although complex representation techniques might eventually be used, rules are generally easier to use for characterizing knowledge during knowledge acquisition. Prototypic rules should be developed as soon as possible to serve as a focal point for directing the course of the knowledge acquisition process. The initial knowledge base can be developed from written materials or from example cases described by the expert during early unstructured interviews. Initial rules should be treated as approximations and their wording should be general to avoid pressuring the expert. As additional cases are described during interviews, the rule base can be expanded. Once a stable rule base begins to develop, it can provide feedback for structuring interviews. Initially the rules and procedures can be traced through by hand with the expert considering each step. The same pattern of tracing through rules should continue once a version of the knowledge base is developed on a computer and it frequent use should become part of the process.

Knowledge Elicitation: Process of Acquiring Knowledge

Knowledge Elicitation is the process of acquiring knowledge about a specific domain. A conceptual model of the domain knowledge is created at the end of the knowledge elicitation process. It is one of the most important and a crucial task of the development of an expert system since it directly has an impact on the overall quality of the system. Knowledge elicitation is also often viewed as the bottleneck in the development of expert systems or knowledge based systems. It is difficult and time consuming activity.

The knowledge is elicited chiefly from experts in the field and data/ information available from published literature. There are various known knowledge elicitation techniques available. The choice of technique to be used in the knowledge elicitation process depends on the nature of the situation within which the knowledge is elicited, the domain knowledge and availability of experts.

The knowledge elicitation process gets tricky as the vast amount of information is often kept inside the heads of domain experts. This makes the entire process complicated as the domain experts may not be willing to disclose the information, due to worries of being sidelined or becoming less important or getting redundant. In certain domains, the domain experts may not even be aware of the tacit knowledge and implicit conceptual models they come to use over many years of experience.

Some of the techniques used in the knowledge elicitation process are as follows:

- **Documentation Analysis**: It is used for orientation and preparation. Documentation is perhaps the most common source of information, as it is often readily available. It helps knowledge engineers to conceptualize unfamiliar content and identify critical concepts in the domain. Documentation should not be the solitary source of information, but it normally supplements other sources of information.
- **Interviews**: Interviews are the oldest and most common tool used for data collection. An interview can be structured or unstructured. Unstructured interviews normally carried out

at the early stages of the knowledge elicitation/ modelling process. The structured interviews help to refine the knowledge acquired

- **Observation**: There are two types of observation techniques, obtrusive and unobtrusive observation. In unobtrusive observation, the observer does not interact with the expert in action. The intention is to observe how a task is being performed usually, without disturbing or interfering in any way. The advantage with unobtrusive observation is that the person will carry out the tasks in a typical manner without any interference. Unobtrusive observation may not always be suitable as certain tasks require interaction to understand the reasoning behind certain steps in the process. In obtrusive observation, the observer gets the person to verbalize his thoughts as the task is being performed.
- **Questionnaires:** When specific information is needed, a questionnaire can sometimes be used effectively. Questionnaires are generally used in combination with other techniques such as interviews.
- **Protocol analysis:** are used with transcripts of interviews or other text-based information to identify various types of knowledge, such as goals, decisions, relationships and attributes. This acts as a bridge between the use of protocol-based techniques and knowledge modeling techniques.
- **Laddering:** are used to build taxonomies or other hierarchical structures such as goal trees and decision networks.
- **Card Sorting**
- **Twenty Questions Technique, etc**

KA Techniques

Many techniques have been developed to help elicit knowledge from an expert. These are referred to as knowledge elicitation or knowledge acquisition (KA) techniques. The term "KA techniques" is commonly used.

The following list gives a brief introduction to the types of techniques used for acquiring, analyzing and modeling knowledge:

- **Protocol-generation techniques** include various types of interviews (unstructured, semi-structured and structured), reporting techniques (such as self-report and shadowing) and observational techniques
- **Protocol analysis techniques** are used with transcripts of interviews or other text-based information to identify various types of knowledge, such as goals, decisions, relationships and attributes. This acts as a bridge between the use of protocol-based techniques and knowledge modeling techniques.
- **Hierarchy-generation techniques,** such as laddering, are used to build taxonomies or other hierarchical structures such as goal trees and decision networks.
- **Matrix-based techniques** involve the construction of grids indicating such things as problems encountered against possible solutions. **Important types include the use of frames for representing the properties of concepts** and the repertory grid technique used to elicit, rate, analyze and categorize the properties of concepts.

- **Sorting techniques** are used for capturing the way people compare and order concepts, and can lead to the revelation of knowledge about classes, properties and priorities.
- **Limited-information and constrained-processing tasks** are techniques that either limits the time and/or information available to the expert when performing tasks. For instance, the twenty questions technique provides an efficient way of accessing the key information in a domain in a prioritized order.
- **Diagram-based techniques** include the generation and use of concept maps, state transition networks, event diagrams and process maps. The use of these is particularly important in capturing the "what, how, when, who and why" of tasks and events.

Machine Vision

"Machine vision" is a field of study and technology whose goal is to provide machines with the ability to perceive selective aspects of the world using visual means.

"**Vision technology** is still a relatively young discipline, which had its breakthrough in the early 1980s. ***It deals with images or sequences of images with the objective of manipulating and analyzing them in order to***

a) Improve image quality (contrast, color, etc.)
b) Restore images (e.g. noise reduction)
c) Code pictures (data compression, for example)
d) Understand and interpret images (image analysis, pattern recognition).

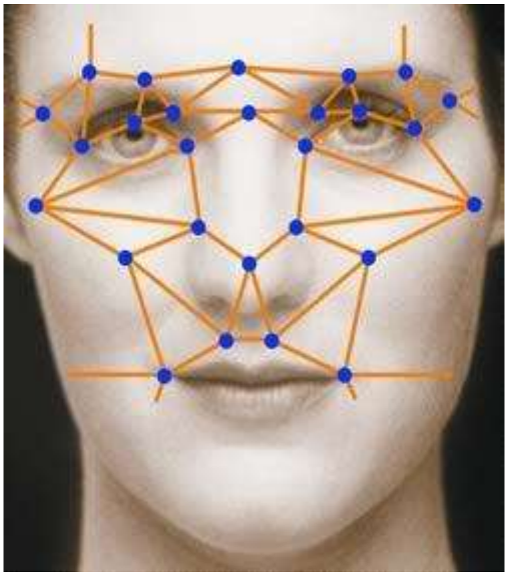


***Figure: Facial measures used in a biometrics vision system*.**

Thus **vision technology** can be applied wherever images are generated and need to be analyzed: in biology (counting cells), in medicine (interpreting CT scanning results), in the construction industry (thermo-graphic analysis of buildings) or in security (verification of biometric dimensions). **Vision technology** is an interdisciplinary technology that combines lighting, optics, electronics, information technology, software and automation technology.

Machine Vision is the study of methods and techniques whereby artificial vision systems can be constructed and usefully employed in practical applications. As such, it embraces both the science and engineering of vision. **Machine vision** (MV) is the process of applying a range of technologies and methods to provide imaging-based automatic inspection, process control and robot guidance in industrial applications. While the scope of MV is broad and a comprehensive definition is difficult to distil, a "generally accepted definition of machine vision is '... the analysis of images to extract data for controlling a process or activity.'"

Here are a just few examples for (machine) vision technology applications:

- Inspecting of the surfaces of bathtubs for scratches.
- Checking whether airbags have been properly installed into cars.
- Applying adhesives evenly and correctly.0. Verifying that welds are strong enough.
- Checking paper in the production process for flaws.
- Making sure that syringes are manufactured properly.
- Finding irregularities on flat glass.
- Guiding robots so that they can adapt to changes in their environment.
- Reading license plates of cars.
- Recognizing and identifying persons.

Top-Down Approach

A **top-down** approach (also known as stepwise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

In this an overview of the system is first formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements.

Bottom-Up Approach

A **bottom-up** approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of **information processing** based on incoming data from the environment to form a **perception**. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output).

In this approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

The bottom-up approach operated directly on the data without any knowledge about the structure of the data or what the data represents (i.e., higher-level knowledge). It does not necessarily exclude higher-level knowledge, but can operate *with or without* this knowledge. It is up to the implementer to decide what types of high-level knowledge the low-level techniques will use, and how and when this knowledge will be used. The top-down approach uses a model based approach.

Top-Down Approach vs Bottom-Up Approach

Top-down approach and Bottom-up approach are two approaches that are commonly employed when designing any project. Not many understand the differences between these two approaches and this article intends to highlight the features of both to make it easier for the reader to appreciate the two concepts in their entirety.

While top down design starts from abstract to finally achieve a solid design, bottom up approach is just the reverse as it begins with the concrete design to get to abstract entity. When it comes to designing brand new systems, it is top down approach that is most commonly employed. On the other hand, in the case of reverse engineering as when it is the goal to understand someone else's design, bottom up approach is utilized.

Bottom-up approach proceeds with the design of lowest level module or subsystem, to the highest module or subsystem. One needs a structure chart to know the steps involved in execution. Also needed are drivers to complete this type of designing.

Top-down approach starts with the top level module and progresses downward to the lowest level module. In reality however, no system is followed rigidly and designers tend to bounce back and forth between these two approaches as the need may be.

There are pros and cons of both the approaches. If we talk of advantages of a top down approach, it is easy to visualize, provides sense of completeness, and it is easy to assess the progress at any stage. On the downside, being a UI driven approach, there are chances of redundant business logics.

On the other hand, in a bottom-up approach, user has the advantages of solid business logic, ability to write good unit test and the ease with which changes can be managed and modified. Its disadvantages are that a lot of effort is required to write test cases and the progress cannot be verified easily at the mid stage.

Hypothesis Driven Approach

What is "Hypothesis Driven" problem solving?

Hypothesis driven problem solving tries to find the shortest path to the solution of a problem. It is based on the premise that the problem solver is an expert in the field. Hence, seeing the symptoms they have a fair idea of what the cause(s) of the problem are. This judgement or instinct about the possible causes is termed "hypotheses" (or hunch?). The expert then collects data to prove or disprove their hypotheses - quickly leading to the solution.

Of course the key here is expertise - in the abscence of which this could be a very difficult process for the client!

Hypothesis-Driven Development

A startup idea is not a plan of action. A startup idea is a series of unchecked hypotheses. In essence, it is a series of questions that you haven't completely answered yet. The process of progressing a startup from idea to functioning business is the process of answering these questions, of validating these hypotheses.

Let's consider a theoretical startup to illustrate this. The initial three questions for most web startups will be in the form:

- Can I actually build it?
- Can I get people to know about it?
- Can I make money from it?

Often, this is the order in which they will arise, if you have some experience of web startups but are fundamentally a builder type. Making money is the last concern. "If I can get lots of passionate users who are willing to pay something, then it will probably be alright."

To apply Hypothesis-driven development properly, you will want to order your questions by priority before proceeding. This is especially essential once you break down the questions into sub-questions and end up with dozens upon dozens of questions.

The best way to prioritise the questions is by uncertainty. An initial order for these three questions might then be:

1. Can I make money from it?
2. Can I get people to know about it?
3. Can I build it?

Your own prioritisation may vary, but if you're technical, "Can I build it?" will probably be last on the list. Of course you can build it. If you couldn't, you would probably have discarded the idea before even getting to this stage.

Before trying to answer the questions, you first break them down into sub-questions (please note this breakdown is nowhere near exhaustive enough, it's just an example):

1. Can I make money from it?
    1. How much will it cost me to serve the smallest users?
        1. Which cloud platform is best for this?
        2. How many instances will I need at a minimum to run the platform?
        3. How many users will I need just to break even?
    2. How much will I be able to charge per user?
    3. What proportion of paid vs free users will I have?
    4. How well will users convert from free to paid?
2. Can I get people to know about it?
    1. What channels are there to get the message out?
    2. How much will each of these channels cost me?
        1. How competitive are the Ad-words for this?
    3. Do I have enough contacts to get the initial, core users so the service will be useful to real users?

3. Can I build it?
    1. What are the hardest bits of technology I'll need to put together?
    2. Can the scope be cut down so that I have a chance of building a version 1 with extremely limited resources?
    3. Which features can be put off until later?

You should keep expanding this list until you can start to see what the burning uncertainties are. These will be unique to your startup idea and to your skills and available resources. Two people evaluating the same idea will probably come up with different key questions. Once you've got those key questions, shift those to the top.

Then, start working through the questions, one by one or even in parallel. Most of the time, the answer will not be found in code, but in good old-fashioned research, planning, and the dreaded Excel spreadsheets. You don't need to answer all these questions with 100% certainty, but you should be clearly aware of the limits of your answers, and when the answer is really critical, you should make an effort to answer it as fully as possible. You can't know everything, but you gotta know what you don't know and how much it can hurt you.

At some point, if the idea has answered enough questions, the next most important question will require you to build something - be it a paper prototype, a landing page, or something else. When it's the most important question, do it, and do just enough to answer the question. Later, if your idea is really good, you will probably, at some point, start to do the really expensive stuff: building a real application.

Neural Networks

Humans and other animals process information with *neural networks*. These are formed from *trillions* of **neurons** (nerve cells) exchanging brief electrical pulses called **action potentials**. ***Computer algorithms that mimic these biological structures are formally called artificial neural networks.***

Artificial neural network is an interconnected group of artificial neurons that uses a mathematical model for information processing based on a connectionist approach to computation.

Artificial neural network is an adaptive system that changes its structure based on external or internal information that flows through the network. Neural network mimic certain processing of the human brain as:

- Neural computing is an information processing paradigm, inspired by biological system, composed of a large number of highly interconnected processing elements(neurons) working in unison to solve specific problems.
- ANNs, like people, learn by example.
- An ANN is configured for a specific application such as pattern recognition or data classification through a learning process.
- Learning in biological system involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.

1. Network Structure

Many different neural network structures have been tried, some based on imitating what a biologist sees under the microscope, some based on a more mathematical analysis of the problem. The most commonly used structure is shown in Fig. 26-5. This neural network is formed in three layers,

1. **The input layer,**
2. **The hidden layer, and**
3. **Output layer**

Each layer consists of one or more **nodes**, represented in this diagram by the small circles. The lines between the nodes indicate the flow of information from one node to the next. In this particular type of neural network, the information flows only from the input to the output (that is, from left-to-right). Other types of neural networks have more complex connections, such as feedback paths.

The nodes of the input layer are **passive**, meaning they do not modify the data. They receive a single value on their input, and duplicate the value to their multiple outputs. In comparison, the nodes of the hidden and output layer are **active**. This means they modify the data as shown in Fig. 26-6.

The variables: $X1_1$, $X1_2$ … $X1_{15}$ hold the data to be evaluated (see Fig. 26-5). For example, they may be pixel values from an image, samples from an audio signal, stock market prices on successive days, etc. They may also be the output of some other algorithm, such as the classifiers in our cancer detection example: diameter, brightness, edge sharpness, etc.

Each value from the **input layer** is duplicated and sent to *all* of the **hidden nodes**. This is called a **fully interconnected** structure. As shown in Fig. 26-6, the values entering **a hidden node** are multiplied by **weights**, a set of predetermined numbers stored in the program. The weighted inputs are then added to produce a single number.

This is shown in the diagram by the symbol, ∑. Before leaving the node, this number is passed through a nonlinear mathematical function called **a *sigmoid***. This is an "s" shaped curve **that limits the node's output**. That is, the input to the sigmoid is a value between -∞ and +∞, while its output can only be between 0 and 1.

The outputs from the hidden layer are represented in the flow diagram (Fig 26-5) by the variables: $X2_1$, $X2_2$, $X2_3$ and $X2_4$. Just as before, **each of these values is duplicated and applied to the next layer**. The **active nodes** of the **output layer combine and modify the data to produce the two output values of this network, $X3_1$and $X3_2$**.

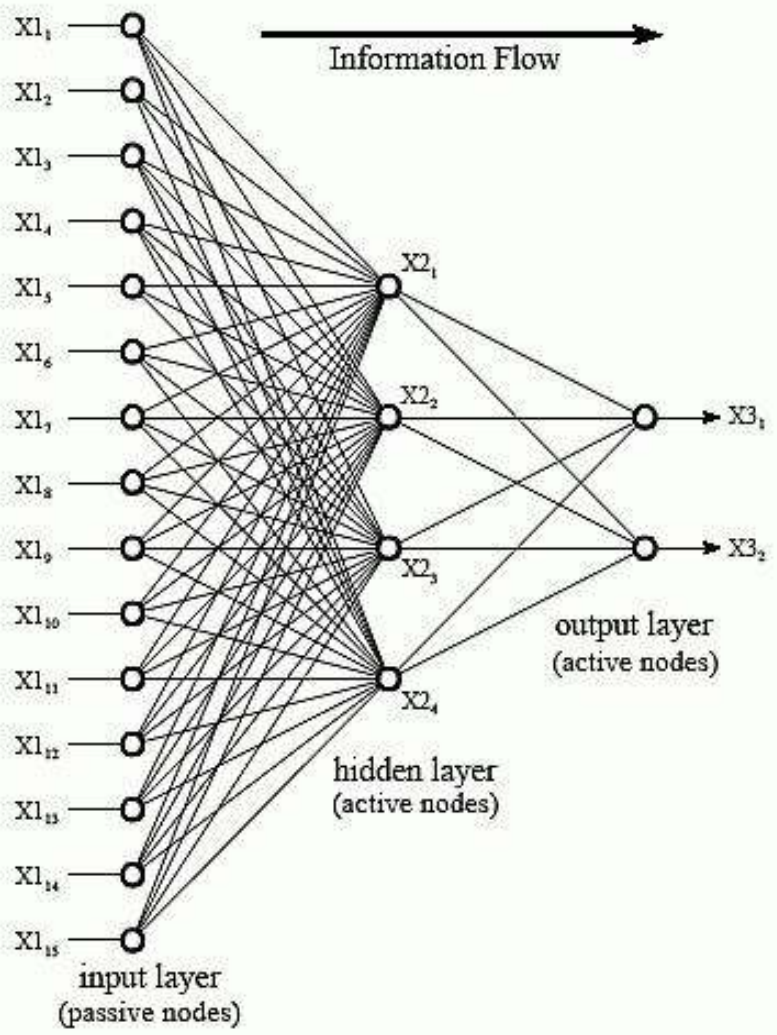


FIGURE 26-5
Neural network architecture. This is the most common structure for neural networks: three layers with full interconnection. The input layer nodes are passive, doing nothing but relaying the values from their single input to their multiple outputs. In comparison, the nodes of the hidden and output layers are active, modifying the signals in accordance with Fig. 26-6. The action of this neural network is determined by the weights applied in the hidden and output nodes.

Neural networks can have any number of layers, and any number of nodes per layer. **Most applications use the three layer structure with a maximum of a few hundred input nodes. The hidden layer is usually about 10% the size of the input layer.** In the case of target detection, the output layer only needs a single node. The output of this node is thresholded to provide a positive or negative indication of the target's presence or absence in the input data.

Example

As an example, imagine a neural network for recognizing objects in a sonar signal. Suppose that 1000 samples from the signal are stored in a computer. How does the computer determine if these data represent a submarine, whale, undersea mountain, or nothing at all? Conventional DSP would approach this problem with mathematics and algorithms, such as correlation and frequency spectrum analysis. With a neural network, the 1000 samples are simply fed into the input layer, resulting in values popping from the output layer. By selecting the proper weights, the output can be configured to report a wide range of information. For instance, there might be outputs for: submarine (yes/no), whale (yes/no), undersea mountain (yes/no), etc.

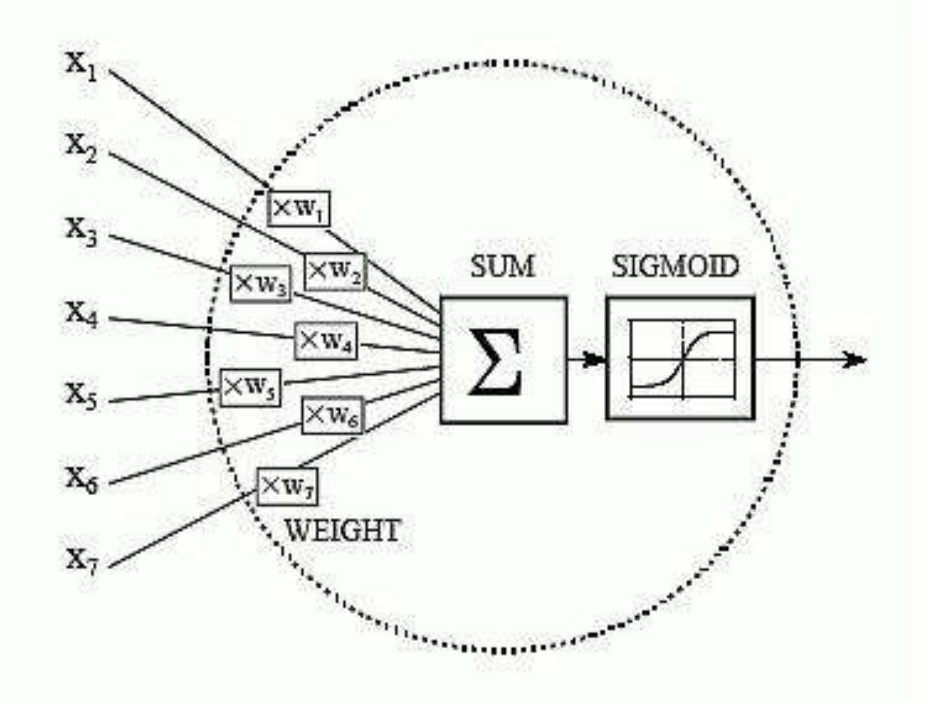


FIGURE 26-6: Neural network active node. This is a flow diagram of the active nodes used in the hidden and output layers of the neural network. Each input is multiplied by a weight (the Wn values) and then summed. This produces a single value that is passed through an "s" shaped non-linear function called a ***sigmoid***.

With other weights, the outputs might classify the objects as: metal or non-metal, biological or non-biological, enemy or ally, etc. No algorithms, no rules, no procedures; only a relationship between the input and output dictated by the values of the weights selected.

The exact shape of the sigmoid is not important, only that it is a **smooth threshold**. For comparison, a **simple threshold** produces a value of *one* when $x > 0$, and a value of *zero* when $x < 0$. The sigmoid performs this same basic threshold function, but is also *differentiable*.

Example:

The neuron shown consists of four inputs with the weights.

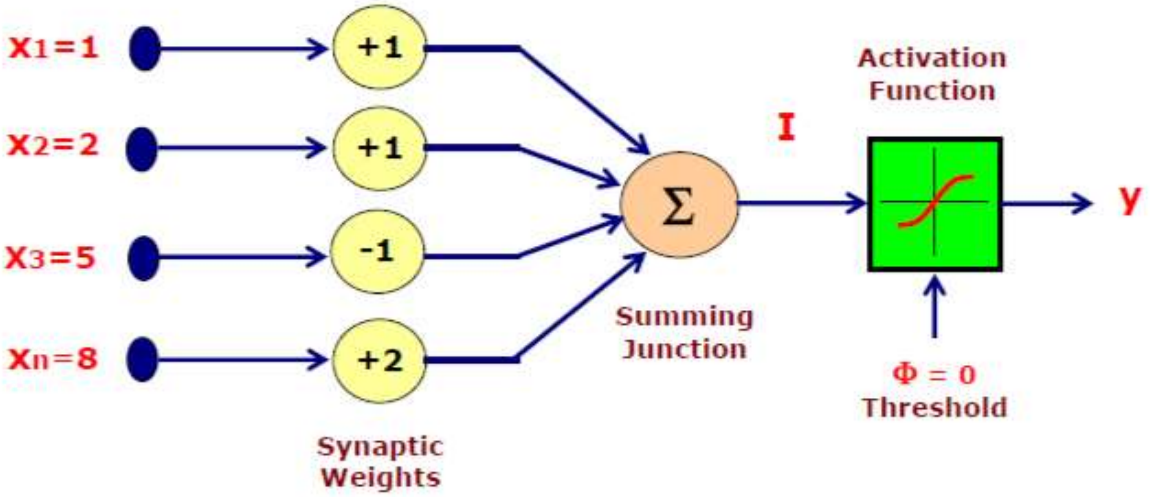


Fig Neuron Structure of Example

The output **I** of the network, prior to the activation function stage, is

$$\mathbf{I} = X^T . W = \begin{bmatrix} 1 & 2 & 5 & 8 \end{bmatrix} \bullet \begin{pmatrix} +1 \\ +1 \\ -1 \\ +2 \end{pmatrix} = 14$$

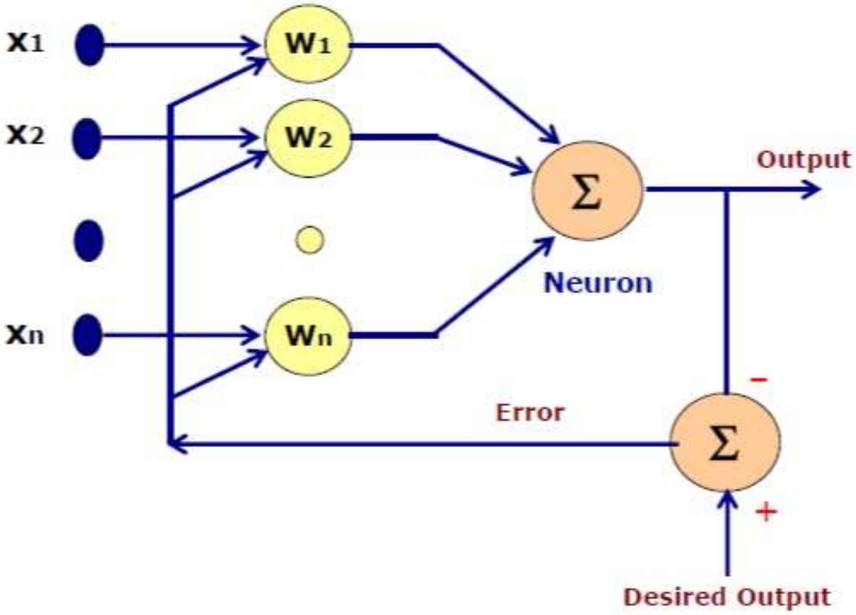
$$= (1 \times 1) + (2 \times 1) + (5 \times -1) + (8 \times 2) = 14$$

With a binary activation function the outputs of the neuron is:

y (threshold) = 1;

2. Adaline

ADALINE (**Adaptive Linear Neuron** or later **Adaptive Linear Element**) is a single layer neural network. It was developed by Professor Bernard Widrow and his graduate student Ted Hoff at Stanford University in 1960. It consists of a weight, a bias and a summation function.



Given the following variables:

- x is the input vector
- w is the weight vector
- n is the number of inputs
- θsome constant
- y is the output

Then we find that the output is $y = \sum_{j=1}^{n} x_j w_j + \theta$

The basic structure of an ADALINE is similar to a neuron with a linear activation and a feedback loop. During the training phase of ADALINE, the input vector as well as the desired output Teacher) are presented to the network.

Learning algorithm

Let us assume:

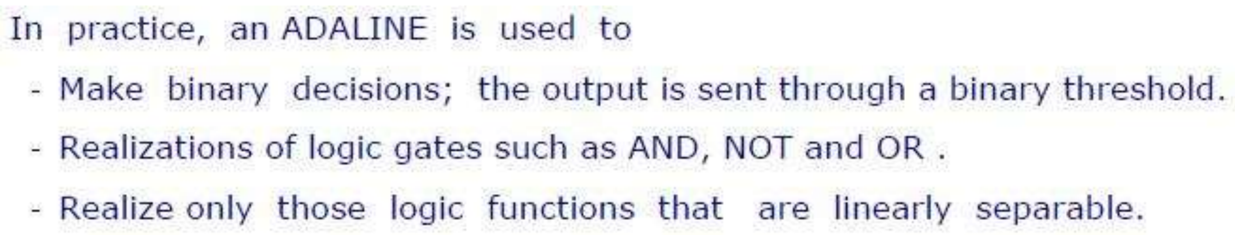
- $\eta$is the learning rate (some constant)
- d is the desired output
- o is the actual output

Then the weights are updated as follows

$$w \leftarrow w + \eta(d - o)x$$

The ADALINE converges to the least squares error which is $E = (d - o)^2$

Uses of ADALINE



In practice, an ADALINE is used to

- Make binary decisions; the output is sent through a binary threshold.
- Realizations of logic gates such as AND, NOT and OR .
- Realize only those logic functions that are linearly separable.

3. Medaline

Madaline (Multiple Adaline) is a **two layer neural network** with a set of **ADALINEs** in parallel as its input layer and a single PE (processing element) in its output layer.

For problems with multiple input variables and one output, each input is applied to one Adaline.

For similar problems with multiple outputs, madalines in parallel can be used.

The madaline network is useful for problems which involve prediction based on multiple inputs, such as **weather forecasting** (Input variables: barometric pressure, difference in pressure. Output variables: rain, cloudy, sunny).

A MADALINE (many Adaline) network is created by combining a number of Adalines. The network of ADALINES can span many layers.

4. Perceptron

The **perceptron** is a type of artificial neural network invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt.

It can be seen as the simplest kind of feed forward neural network: a linear classifier.

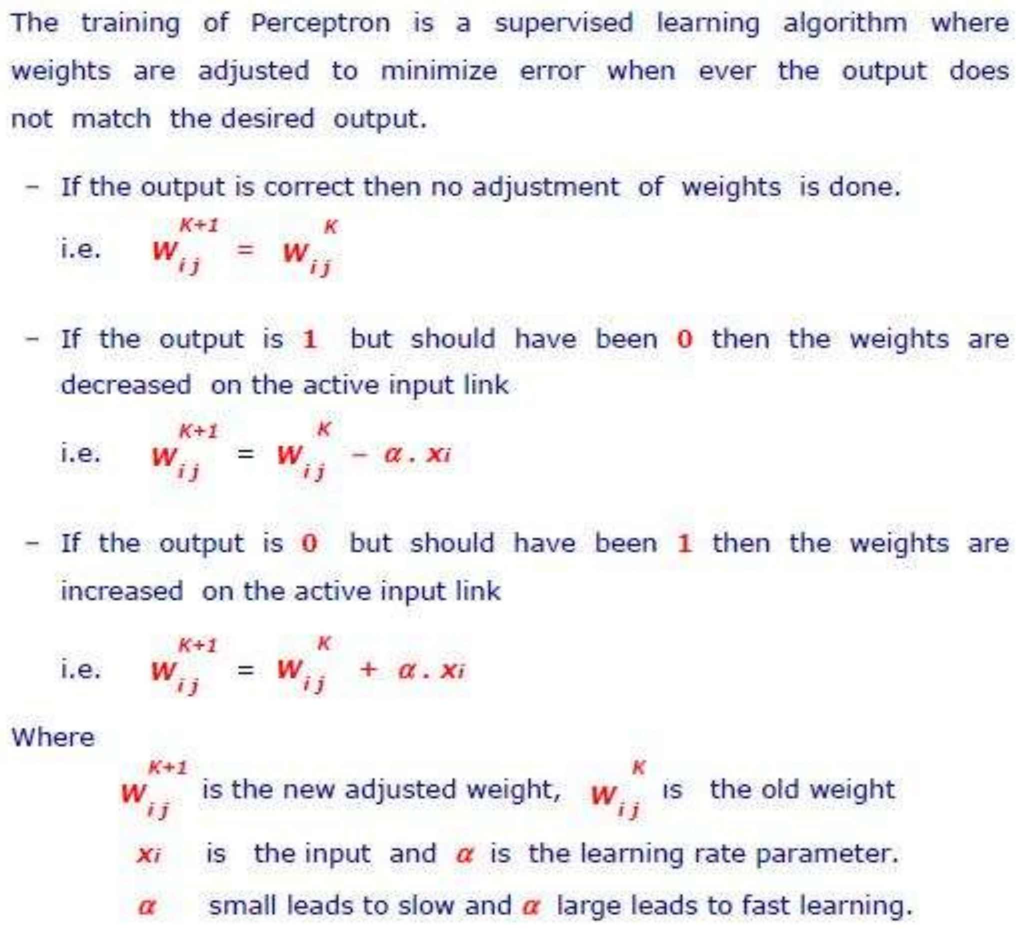
The perceptron is a binary classifier which maps its input x (a real-valued vector) to an output value $f(x)$ (a single binary value):

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Where ***w*** is **a vector of real-valued weights**, $w \cdot x$ is the dot product (which here computes a weighted sum), and b is the 'bias', a constant term that does not depend on any input value.

The value of $f(x)$ (0 or 1) is used to classify x as either a positive or a negative instance, in the case of a binary classification problem. If b is negative, then the weighted combination of inputs must produce a positive value greater than |b|. Spatially, the bias alters the position (though not the orientation) of the decision boundary. The perceptron learning algorithm does not terminate if the learning set is not linearly separable.

Single Layer Perceptron Learning Algorithm



The training of Perceptron is a supervised learning algorithm where weights are adjusted to minimize error when ever the output does not match the desired output.

- If the output is correct then no adjustment of weights is done.

  i.e. $W_{ij}^{K+1} = W_{ij}^{K}$

- If the output is **1** but should have been **0** then the weights are decreased on the active input link

  i.e. $W_{ij}^{K+1} = W_{ij}^{K} - \alpha . x_i$

- If the output is **0** but should have been **1** then the weights are increased on the active input link

  i.e. $W_{ij}^{K+1} = W_{ij}^{K} + \alpha . x_i$

Where

W_{ij}^{K+1} is the new adjusted weight, W_{ij}^{K} is the old weight

x_i is the input and α is the learning rate parameter.

α small leads to slow and α large leads to fast learning.

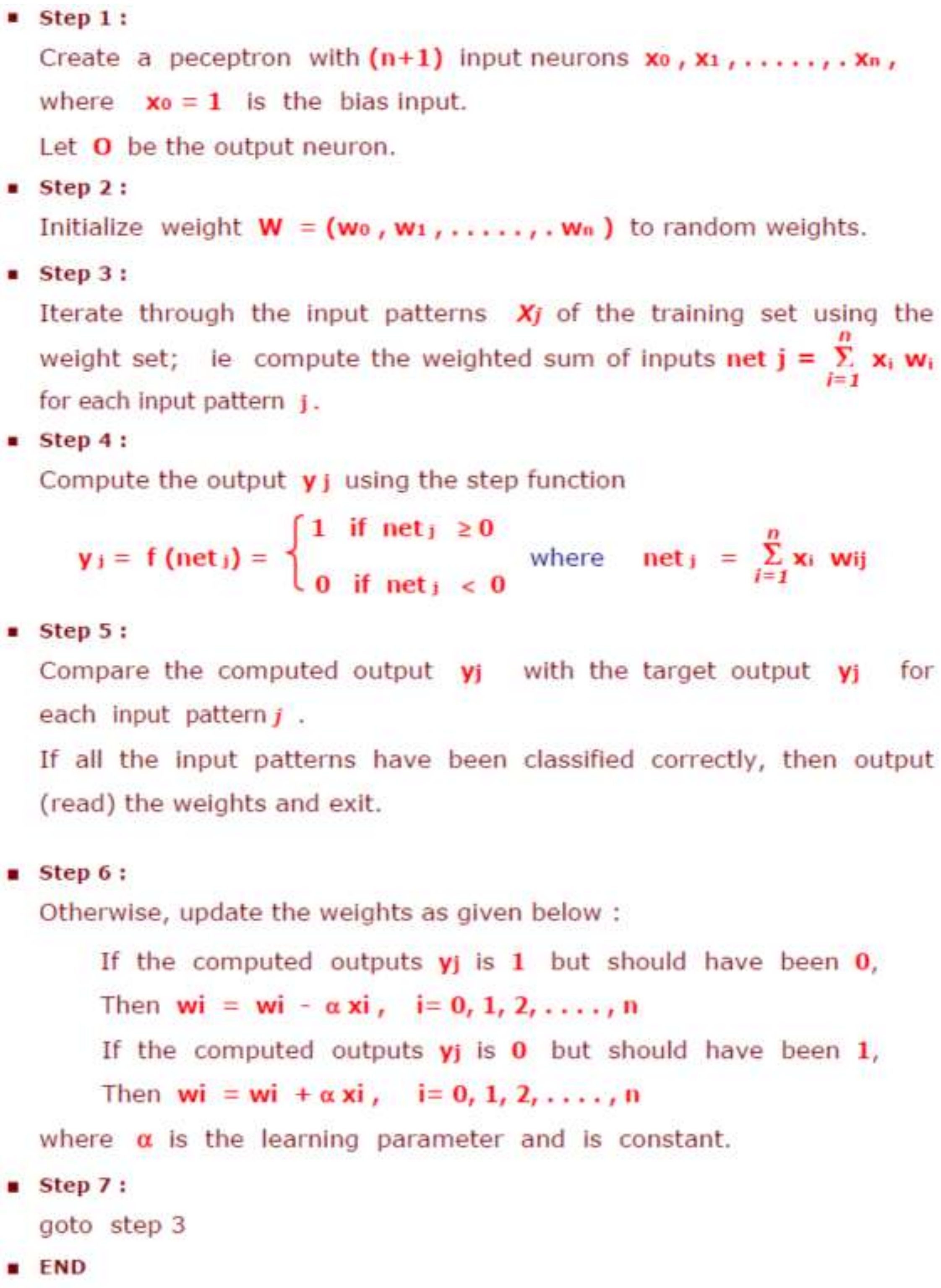
Here, bias is not considered in the learning algorithm of the perceptron.

Algorithm:

i. Initialize weights and threshold.
ii. Present input and desired output
iii. Calculate the actual output
iv. Adapts weights

Steps iii and iv are repeated until the iteration error is less than a user-specified error threshold or a predetermined number of iterations have been completed.

Algorithm (in detail)



- **Step 1 :**

  Create a peceptron with **(n+1)** input neurons $x_0, x_1, \dots\dots, x_n$, where $x_0 = 1$ is the bias input.

  Let **O** be the output neuron.

- **Step 2 :**

  Initialize weight $W = (w_0, w_1, \dots\dots, w_n)$ to random weights.

- **Step 3 :**

  Iterate through the input patterns X_j of the training set using the weight set; ie compute the weighted sum of inputs $\text{net } j = \sum_{i=1}^{n} x_i w_i$ for each input pattern j.

- **Step 4 :**

  Compute the output y_j using the step function

  $$y_j = f(\text{net}_j) = \begin{cases} 1 & \text{if } \text{net}_j \geq 0 \\ 0 & \text{if } \text{net}_j < 0 \end{cases} \quad \text{where} \quad \text{net}_j = \sum_{i=1}^{n} x_i \; w_{ij}$$

- **Step 5 :**

  Compare the computed output y_j with the target output y_j for each input pattern j .

  If all the input patterns have been classified correctly, then output (read) the weights and exit.

- **Step 6 :**

  Otherwise, update the weights as given below :

  If the computed outputs y_j is **1** but should have been **0**,

  Then $w_i = w_i - \alpha x_i$, $i = 0, 1, 2, \dots, n$

  If the computed outputs y_j is **0** but should have been **1**,

  Then $w_i = w_i + \alpha x_i$, $i = 0, 1, 2, \dots, n$

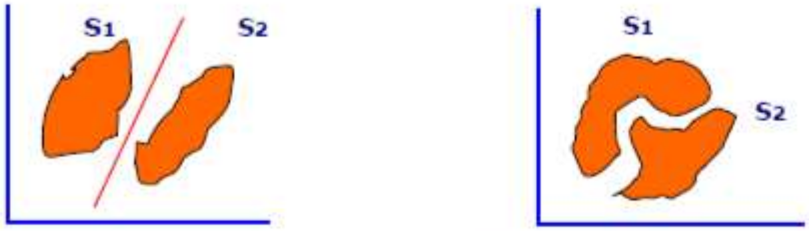
  where α is the learning parameter and is constant.

- **Step 7 :**

  goto step 3

- **END**

Note Perceptron cannot handle tasks that are not linearly separable.



(a) Linearly separable patterns (b) Not Linearly separable patterns

Note : Perceptron cannot find weights for classification problems that are not linearly separable.

5. Multilayer Perceptron

Single Layer Perceptron

Definition : An arrangement of one input layer of neurons feed forward to one output layer of neurons is known as Single Layer Perceptron.

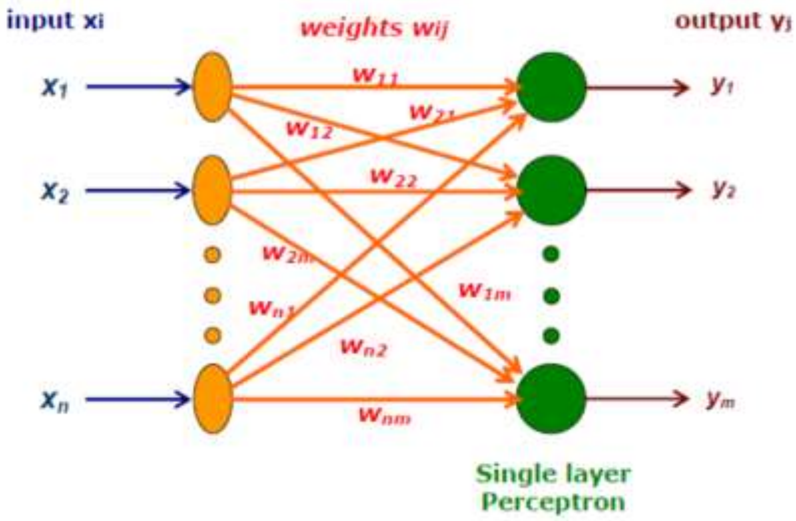


Fig. Simple Perceptron Model

$$y_j = f(net_j) = \begin{cases} 1 & \text{if } net_j \ge 0 \\ 0 & \text{if } net_j < 0 \end{cases} \quad \text{where } net_j = \sum_{i=1}^{n} x_i \, w_{ij}$$

Multilayer Perceptron

The *multilayer perceptron* (MLP) is a hierarchical structure of several perceptrons, and overcomes the shortcomings of these single-layer networks.

The *multilayer perceptron* is an artificial neural network that learns nonlinear function mappings. The multilayer perceptron is capable of learning a rich variety of nonlineardecision surfaces.

Nonlinear functions can be represented by multilayer perceptrons with units that use nonlinear activation functions. Multiple layers of cascaded linear units still produce only linear mappings.

A neural network with one or more layers of nodes between the input and the output nodes is called *multilayer network*.

The multilayer *network structure*, or *architecture*, or *topology*, consists of an input layer, two or more hidden layers, and one output layer. The input nodes pass values to the first hidden layer, its nodes to the second and so on till producing outputs.

A network with a layer of input units, a layer of hidden units and a layer of output units is a *two-layer network*. A network with two layers of hidden units is a *three-layer network*, and so on. A justification for this is that the layer of input units is used only as an input channel and can therefore be discounted.

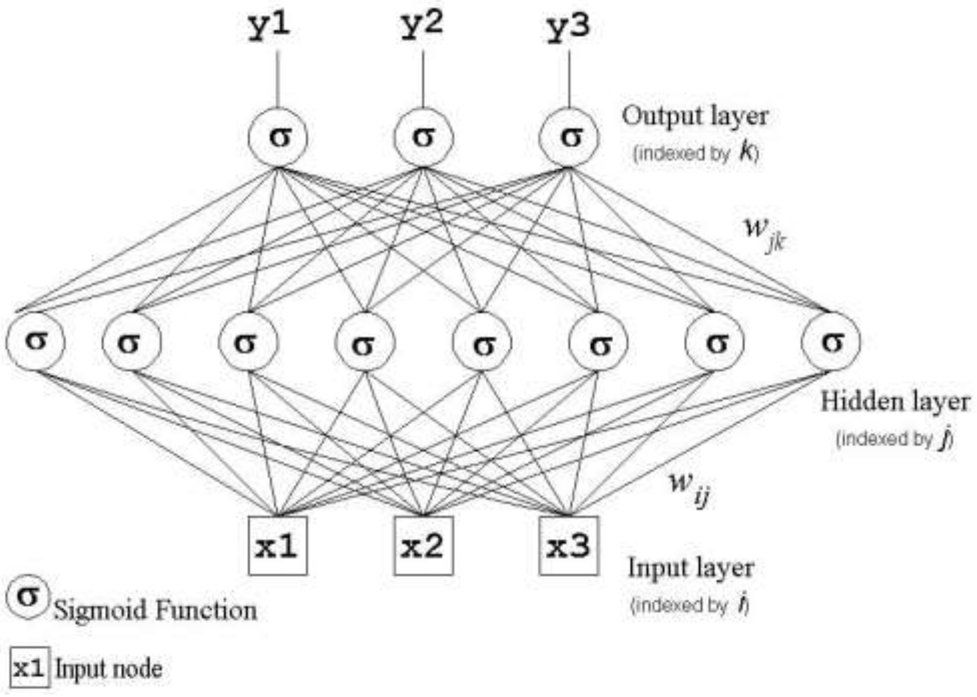


Figure 1. Multilayer Perceptron (MLP)

The algorithm for Perceptron Learning is based on the back-propagation rule discussed previously. The multilayer network MLP has a *highly connected topology* since every input is connected to all nodes in the first hidden layer, every unit in the hidden layers is connected to all nodes in the next layer, and so on.

The input signals, initially these are the input examples; propagate through the neural network in a forward direction on a layer-by-layer basis that is why they are often called *feed forward multilayer networks*.

Two kinds of signals pass through these networks:

- *Function signals*: the input examples propagated through the hidden units and processed by their activation functions emerge as outputs.

- *Error signals*: the errors at the output nodes are propagated backward layer-by-layer through the network so that each node returns its error back to the nodes in the previous hidden layer.

6. Radial Bias Function

A **radial basis function (RBF)** is a real-valued function whose value depends only on the distance from the origin, so that $\phi(\mathbf{x}) = \phi(\|\mathbf{x}\|)$; or alternatively on the distance from some other point *c*, called a *center*, so that $\phi(\mathbf{x}, \mathbf{c}) = \phi(\|\mathbf{x} - \mathbf{c}\|)$.
Any function ϕ that satisfies the property $\phi(\mathbf{x}) = \phi(\|\mathbf{x}\|)$ is a radial function.

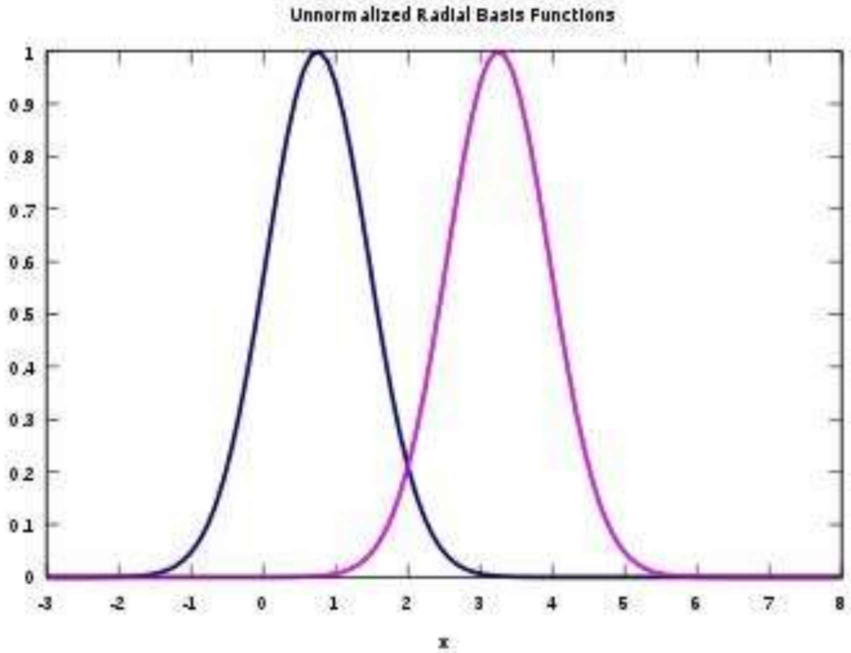


Figure: Two un-normalized Gaussian radial basis functions in one input dimension. The basis function centers are located at x_1=0.75 and x_2=3.25.

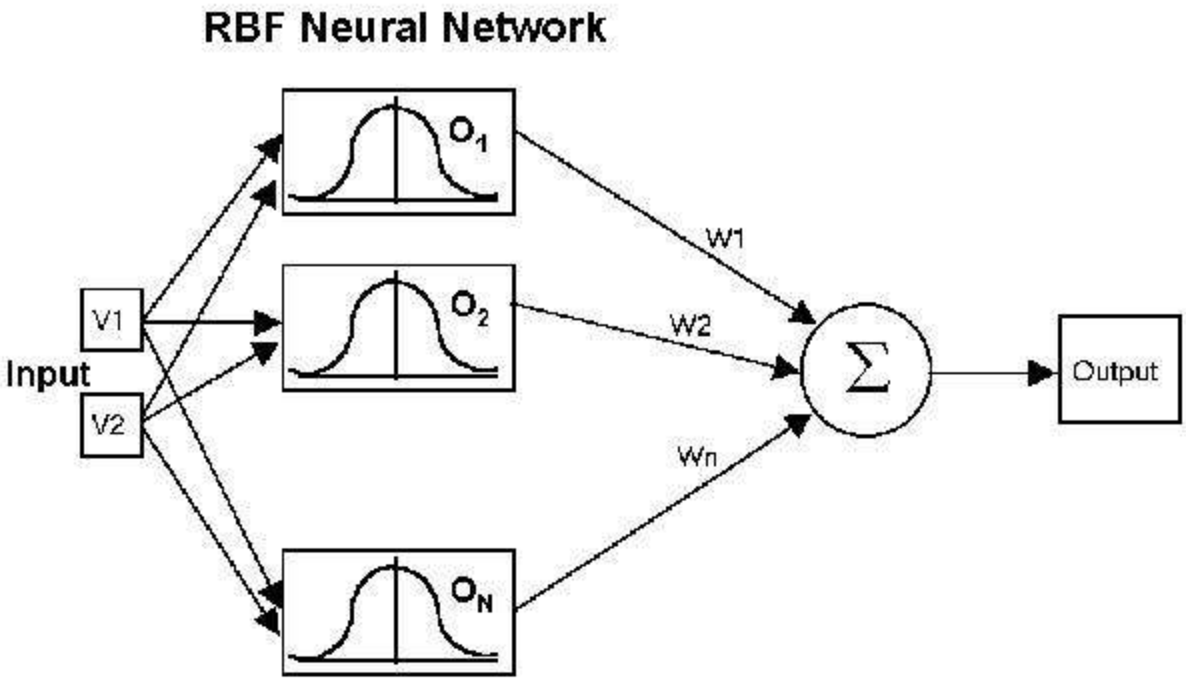
Radius Basis Function Network

The sum

$$y(\mathbf{x}) = \sum_{i=1}^{N} w_i \, \phi(\|\mathbf{x} - \mathbf{x}_i\|),$$

can also be interpreted as a rather simple single-layer type of **artificial neural network** called a **radial basis function network,** with the radial basis functions taking on the role of the activation functions of the network.

RBF Network Architecture



RBF networks have three layers:

1. **Input layer**

   There is one neuron in the input layer for each predictor variable. In the case of categorical variables, N-1 neurons are used where N is the number of categories. The input neurons (or processing before the input layer) standardize the range of the values by subtracting the median and dividing by the inter-quartile range. The input neurons then feed the values to each of the neurons in the hidden layer.

2. **Hidden layer**

   This layer has a variable number of neurons (the optimal number is determined by the training process). **Each neuron consists of a radial basis function centered on a point with as many dimensions as there are predictor variables**. The spread (radius) of the RBF function may be different for each dimension. The centers and spreads are determined by the training process. When presented with the x vector of input values from the input layer, a hidden neuron computes the Euclidean distance of the test case from the neuron's center point and then applies the RBF kernel function to this distance using the spread values. The resulting value is passed to the summation layer.

3. **Summation layer**

   The value coming out of a neuron in the hidden layer is multiplied by a weight associated with the neuron (W1, W2, ..., Wn in this figure) and passed to the summation which adds up the weighted values and presents this sum as the output of the network.

   Not shown in this figure is a bias value of 1.0 that is multiplied by a weight W0 and fed into the summation layer. For classification problems, there is one output (and a separate set of weights and summation unit) for each target category. The value output for a category is the probability that the case being evaluated has that category.

7. Hopfield Network

A **Hopfield network** is a form of **recurrent** artificial **neural network** invented by **John Hopfield**.

Hopfield network can be seen as **content-addressable memory** or **associative** and can be used for different pattern recognition problems.

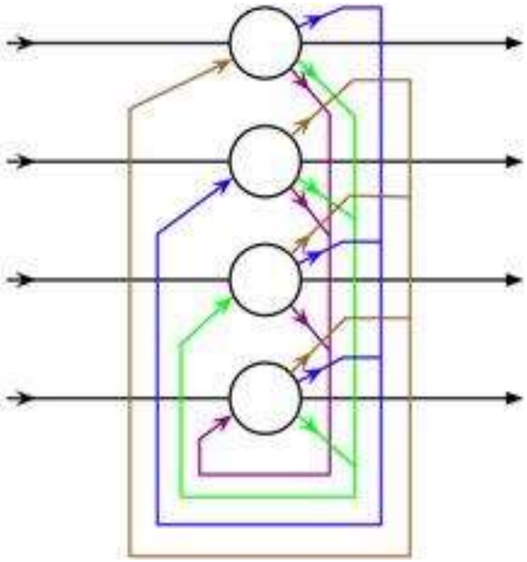
Hopfield networks provide a model for understanding human memory.

The Hopfield net is **a fully connected, symmetrically weighted network where each node functions both as input and output node.**
The idea is that, depending on the weights, some states are unstable and the net will iterate a number of times to settle in a stable state. The net is initialized to have a stable state with some known patterns. Then, the function of the network is to receive a noisy or unclassified pattern as input and produce the known, learnt pattern as output.

Properties of the Hopfield network

- A recurrent network with all nodes connected to all other nodes
- Nodes have binary outputs (either 0,1 or -1,1)
- Weights between the nodes are **symmetric** $W_{ij} = W_{ji}$
- No connection from a node to **itself** is allowed
- Nodes are updated **asynchronously** (i.e. nodes are selected at random)
- The network has no **"hidden" nodes or layer**

Structure



A Hopfield net with four nodes

The units in Hopfield nets are binary threshold units, i.e. the units only take on two different values for their states and the value is determined by whether or not the units' input exceeds their threshold.

Hopfield nets can either have units that take on values of 1 or -1, or units that take on values of 1 or 0. So, the two possible definitions for unit *i*'s activation, a_i, are:

(1) $$a_i \leftarrow \begin{cases} 1 & \text{if } \sum_j w_{ij} s_j > \theta_i, \\ -1 & \text{otherwise.} \end{cases}$$

(2) $$a_i \leftarrow \begin{cases} 1 & \text{if } \sum_j w_{ij} s_j > \theta_i, \\ 0 & \text{otherwise.} \end{cases}$$

Where, w_{ij} is the strength of the connection weight from unit j to unit i (the weight of the connection), s_j is the state of unit j and θ_i is the threshold of unit i.

The connections in a Hopfield net typically have the following restrictions:

- $w_{ii} = 0, \forall i$ (no unit has a connection with itself)
- $w_{ij} = w_{ji}, \forall i, j$ (connections are symmetric)

Activation algorithm

A node i is chosen at random for updating. Every node has a fixed threshold U_i and the nodes output or current state s_i is set by first calculating the netinput to the node which is the sum of all the input connection multiplied with their weights:

$$\text{netinput}_i = \sum_{j \neq i} W_{ij} s_j$$

And then the state is set to 0 or 1 according to this simple rule:

$$s_i = \begin{cases} 0 & \text{netinput}_i < U_i \\ 1 & \text{netinput}_i > U_i \end{cases}$$

Learning algorithm

The learning algorithm of the Hopfield network is unsupervised, meaning that there is no “teacher” telling the network what is the correct output for a certain input.

The algorithm is based on the principle of Hebbian learning which states that neurons that are active at the same time increase the weight between them, often simplified as “cells that fire together, wire together”.

Training a Hopfield net involves lowering the energy of states that the net should "remember". This allows the net to serve as a **content addressable memory system,** that is to say, the network will converge to a "remembered" state if it is given only part of the state.

The net can be used to recover from a distorted input the trained state that is most similar to that input. This is called **associative memory** because it recovers memories on the basis of similarity.

For example, if we train a Hopfield net with five units so that the state (1, 0, 1, 0, 1) is an energy minimum, and we give the network the state (1, 0, 0, 0, 1) it will converge to (1, 0, 1, 0, 1).

Thus, the network is properly trained when the energy of states which the network should remember are local minima.

Example: Pattern recognition

How can we use this network to recognize patterns, where the input patterns can be noisy compared to the actual stored patterns?

Let us say we want to recognize which symbol a MxN sized image resembles to a number of stored symbols/images. Also assume that the image is black and white only. One easy way to do this is to have one node for each pixel in the image, so that we have MxN nodes in total. A node is on, i.e. its state is equal to 1, if its pixel is black, and is off, or 0, if its pixel is white.

By first training the network with the symbols/images we want it to learn. And then setting the network in a state given by a input pattern/image which is noisy. The network will after a lot of updates in the nodes state values, according to the activation algorithm, converge to the stored symbol which the input symbol resembles the most.

Hopfield Network as Model of Associative memory

Let us say you hear a melody of a song and suddenly remember when you where on a concert hearing your favorite band playing just that song. That is associative memory. You get an input, which is a fragment of a memory you have stored in your brain, and get an output of the entire memory you have stored in your brain.

Our memories function in what is called an **associative** or **content-addressable** fashion. That is, a memory does not exist in some isolated fashion, located in a particular set of neurons. All memories are in some sense strings of memories - you remember someone in a variety of ways - by the color of their hair or eyes, the shape of their nose, their height, the sound of their voice, or perhaps by the smell of a favorite perfume. Thus memories are stored in *association* with one another. These different sensory units lie in completely separate parts of the brain, so it is clear that the memory of the person must be distributed throughout the brain in some fashion.

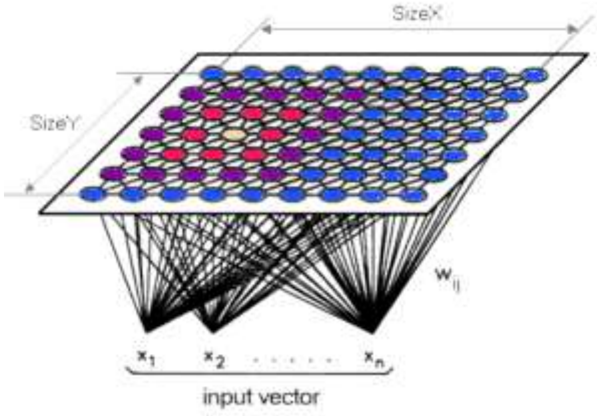
The **Hopfield neural network** is a simple artificial network which is able to store certain memories or patterns in a manner rather similar to the brain. That is; the Hopfield network explained here works in the same way. When the network is presented with an input, i.e. put in a state, the networks nodes will start to update and converge to a state which is a previously stored pattern. The learning algorithm "stores" a given pattern in the network by adjusting the weights. There is off course a limit to how many "memories" you can store correctly in the network, and empirical results show that for every pattern to be stored correctly, the number of patterns has to between 10-20% compared to the number of nodes.

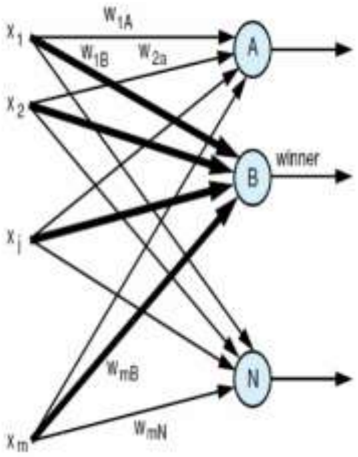
Q. The main idea of Hopfield Network is content addressable memory. Explain why it is not good for constraint satisfaction problem. Then what could be the alternative method in such case? [2010]

Q Explain the Hopfield network as a model of Content-Addressable Memory. [2007]

8. Kohonen Network

- A Kohonen map, or **self-organizing feature map** or **self-organizing map**, is a form of neural network invented by Kohonen in the 1980s
- The Kohonen map uses the winner-take-all-algorithm (a form of competitive learning)
  – Only one neuron provides the output of the network in response to a given input (the neuron that has the highest activation level)
  – Kohonen map is particularly useful for clustering data where the clusters are not known in advance
- Unsupervised Networks: do not require target outputs for each input vector in the training data
- Closely related to clustering
- Inputs are connected to a two-dimensional grid of neurons





- Multi-dimensional data can be mapped onto a two-dimensional surface
- A Kohonen map has two layers:
  – Input layer
  – C**luster layer** (Output layer)
- Each input node is connected to every node in the cluster layer

Learning in Kohonen Network

- All weights are set to small random values
- Learning rate is a small positive value
- Input vector is presented to the input layer of the map.
- Input layer feeds the input data to the cluster layer.
- The neuron in the cluster layer that most closely matches the input data is declared the winner.

  - This neuron provides the output classification of the map and also has its weights updated.
- For the winner node, the weight vector is rewarded
- To determine which neuron wins,
  - Weights of output node is treated as a vector
  - This weight vector is compared with the input vector
  - The neuron whose weight vector is closest to the input vector is the winner

Strength of Kohonen Network

- Unsupervised architecture
- Requires no target output vectors
- Simply organises itself into the best representation for the data used in training

Kohonen Neural Network (In Detail)

The Kohonen neural network works differently than the feed forward neural network. The Kohonen neural network contains only an input and output layer of neurons. There is no hidden layer in a Kohonen neural network. It is **unsupervised** learning network.

The input to a Kohonen neural network is given to the neural network using the input neurons. These input neurons are each given the floating point numbers that make up the input pattern to the network. A Kohonen neural network requires that these inputs be normalized to the range between -1 and 1. Presenting an input pattern to the network will cause a reaction from the output neurons.

The output of a Kohonen neural network is very different from the output of a feed forward neural network. If we had a neural network with five output neurons we would be given an output that consisted of five values. This is not the case with the Kohonen neural network. In a Kohonen neural network only one of the output neurons actually produces a value. Additionally, this single value is either true or false.

When the pattern is presented to the Kohonen neural network, one single output neuron is chosen as the output neuron. Therefore, the output from the Kohonen neural network is usually the index of the neuron (i.e. Neuron #5) that fired.

The structure of a typical Kohonen neural network is shown in below Figure.

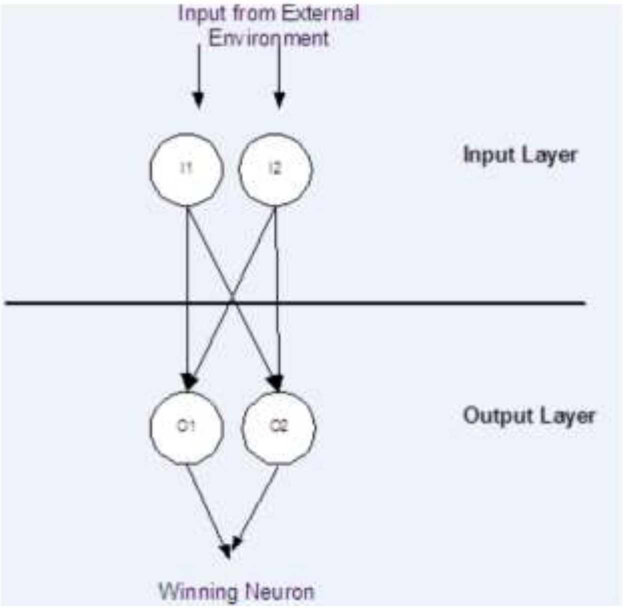


Figure: A Kohonen Neural Network

Now that you understand the structure of the Kohonen neural network we will examine how the network processes information. To examine this process we will step through the calculation process. For this example we will consider a very simple Kohonen neural network. This network will have only two input and two output neurons. The input given to the two input neurons is shown in Table 6.1.

Table 1: Sample Inputs to a Kohonen Neural Network

Input Neuron 1 (I1)	0.5
Input Neuron 2 (I2)	0.75

We must also know the connection weights between the neurons. These connection weights are given in Table 6.2.

Table 2: Connection weights in the sample Kohonen neural network

I1->O1	0.1
I2->O1	0.2
I1->O2	0.3
I2->O2	0.4

Using these values we will now examine which neuron would win and produce output. We will begin by normalizing the input.

Normalizing the Input

The Kohonen neural network requires that its input be normalized. Because of this some texts refer to the normalization as a third layer. For the purposes of this book the Kohonen neural network is considered a two layer network because there are only two actual neuron layers at work in the Kohonen neural network.

The requirements that the Kohonen neural network places on its input data are one of the most severe limitations of the Kohonen neural network. Input to the Kohonen neural network should be between the values -1 and 1. In addition, each of the inputs should fully use the range. If one, or more, of the input neurons were to use only the numbers between 0 and 1, the performance of the neural network would suffer.

To normalize the input we must first calculate the "vector length" of the input data, or vector.
This is done by summing the squares of the input vector. In this case it would be.

(0.5 * 0.5) + (0.75 * 0.75) = 0.8125

This would result in a "vector length" of 0.8125. If the length becomes too small, say less than the length is set to that same arbitrarily small value. In this case the "vector length" is a sufficiently large number. Using this length we can now determine the normalization factor. The normalization factor is the reciprocal of the square root of the length. For our value the normalization factor is calculated as follows.

$$\frac{1}{\sqrt{0.8125}}$$

This results in a normalization factor of 1.1094. This normalization process will be used in the next step where the output layer is calculated.

Calculating Each Neuron's Output

To calculate the output the input vector and neuron connection weights must both be considered. First the "dot product" of the input neurons and their connection weights must be calculated. The result of this is as follows.

$$|0.5 \quad 0.75| \bullet |0.1 \quad 0.2| = (0.5*0.75)+(0.1*0.2) = 0.395$$

This calculation will be performed for the first output neuron. This calculation will have to be done for each of the output neurons. Through this example we will only examine the calculations for the first output neuron. The calculations necessary for the second output neuron are calculated in the same way.

This output must now be normalized by multiplying it by the normalization factor that was determined in the previous step. You must now multiply the dot product of 0.395 by the normalization factor of 1.1094. This results in an output of 0.438213. Now that the output has been calculated and normalized it must be mapped to a bipolar number.

Mapping to Bipolar

In the bipolar system the binary zero maps to -1 and the binary remains a 1. Because the input to the neural network normalized to this range we must perform a similar normalization to the output of the neurons. To make this mapping we add one and divide the result in half. For the output of 0.438213 this would result in a final output of 0.7191065.

The value 0.7191065 is the output of the first neuron. This value will be compared with the outputs of the other neuron. By comparing these values we can determine a "winning" neuron.

Choosing the Winner

We have seen how to calculate the value for the first output neuron. If we are to determine a winning output neuron we must also calculate the value for the second output neuron. We will now quickly review the process to calculate the second neuron. For a more detailed description you should refer to the previous section.

The second output neuron will use exactly the same normalization factor as was used to calculate the first output neuron. As you recall from the previous section the normalization factor is 1.1094. If we apply the dot product for the weights of the second output neuron and the input vector we get a value of 0.45. This value is multiplied by the normalization factor of 1.1094 to give the value of 0.0465948. We can now calculate the final output for neuron 2 by converting the output of 0.0465948 to bipolar yields 0.49923.

As you can see we now have an output value for each of the neurons. **The first neuron has an output value of 0.7191065 and the second neuron has an output value of 0.49923. To choose the winning neuron we choose the output that has the largest output value.** In this case the winning neuron is the first output neuron with an output of 0.7191065, which beats neuron two's output of 0.49923.

9. Elastic net model

Elastic Net Algorithm

The elastic net algorithm is an iterative procedure where M points, with M larger than the number of cities N, are lying on a circular ring or "rubber band" originally located at the center of the cities. The rubber band is gradually elongated until it passes sufficiently near each city to define a tour. During that process two forces apply: one for minimizing the length of the ring, and the other one for minimizing the distance between the cities and the points on the ring. These forces are gradually adjusted as the procedure evolves.

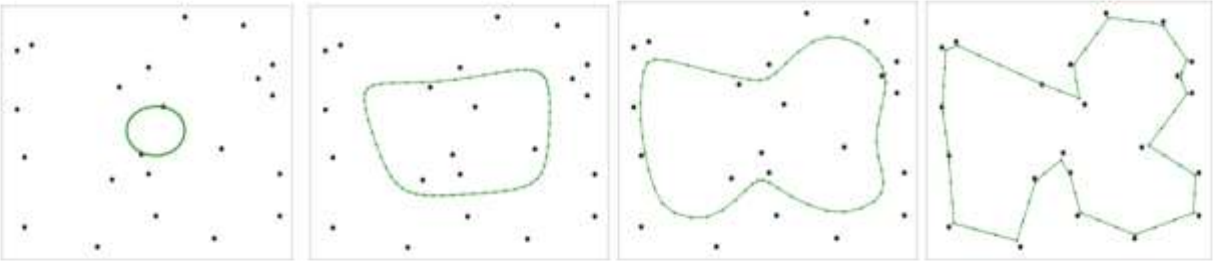


Figure: An Elastic Net Algorithm for the Travelling Salesman Problem.

10. Back –propagation

Back-propagation is a common method of training artificial neural networks so as to minimize the objective function. Arthur E. Bryson and Yu-Chi Ho described it as a multi-stage dynamic system optimization method in 1969.

It is a supervised learning method, and is a generalization of the delta rule. It requires a dataset of the desired output for many inputs, making up the training set.

It is most useful for feed-forward networks (networks that have no feedback, or simply, that have no connections that loop).

Back-propagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.

Understanding Back Propagation

When a learning pattern is clamped (fixed), the activation values are propagated to the output units, and the actual network output is compared with the desired output values, we usually end up with an error in each of the output units. Let's call this error e_o for a particular output unit o. We have to bring e_o to zero. The simplest method to do this is the greedy method: we strive to change the connections in the neural network in such a way that, next time around, the error eo will be zero for this particular pattern. We know from the delta rule that, in order to reduce an error, we have to adapt its incoming weights according to.

$$\Delta w_{ho} = (d_o - y_o) y_h.$$

That's step one. But it alone is not enough: when we only apply this rule, the weights from input to hidden units are never changed, and we do not have the full representational power of the feed-forward network as promised by the universal approximation theorem. In order to adapt the weights from input to hidden units, we again want to apply the delta rule. In this case, however, we do not have a value for δ for the hidden units. This is solved by the chain rule which does the following: distribute the error of an output unit o to all the hidden units that is it connected to, weighted by this connection. Differently put, a hidden unit h receives a delta from each output unit o equal to the delta of that output unit weighted with (= multiplied by) the weight of the connection between those units.

Back-propagation Algorithm

The application of the generalized delta rule thus involves two phases:

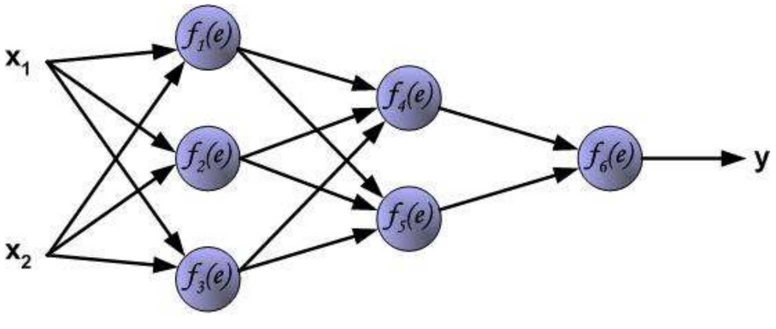
During the **first phase**, the input x is presented and propagated forward through the network to compute the output values y_{po} for each output unit. This output is compared with its desired value d_o, resulting in an error signal δ_{po} for each output unit.

The **second phase** involves a backward pass through the network during which the error signal is passed to each unit in the network and appropriate weight changes are calculated.

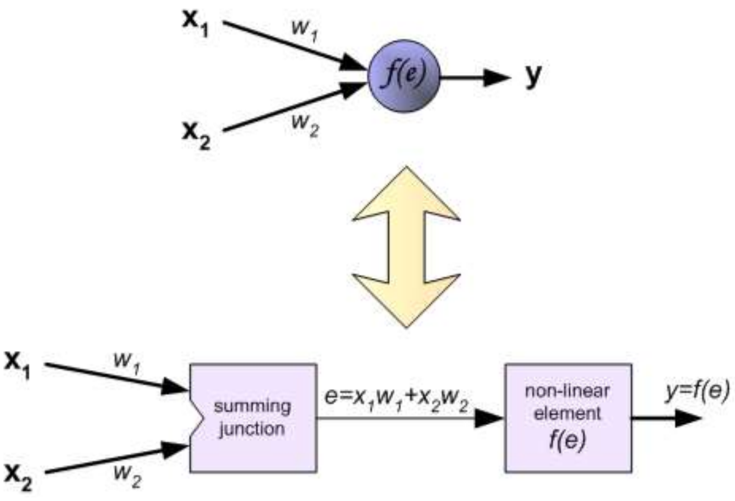
Note that back-propagation networks are necessarily multilayer perceptron (usually with one input, one hidden, and one output layer). In order for the hidden layer to serve any useful function, multilayer networks must have non-linear activation functions for the multiple layers: a multilayer network using only linear activation functions is equivalent to some single layer, linear network.

Example: Principles of training multi-layer neural network using backpropagation

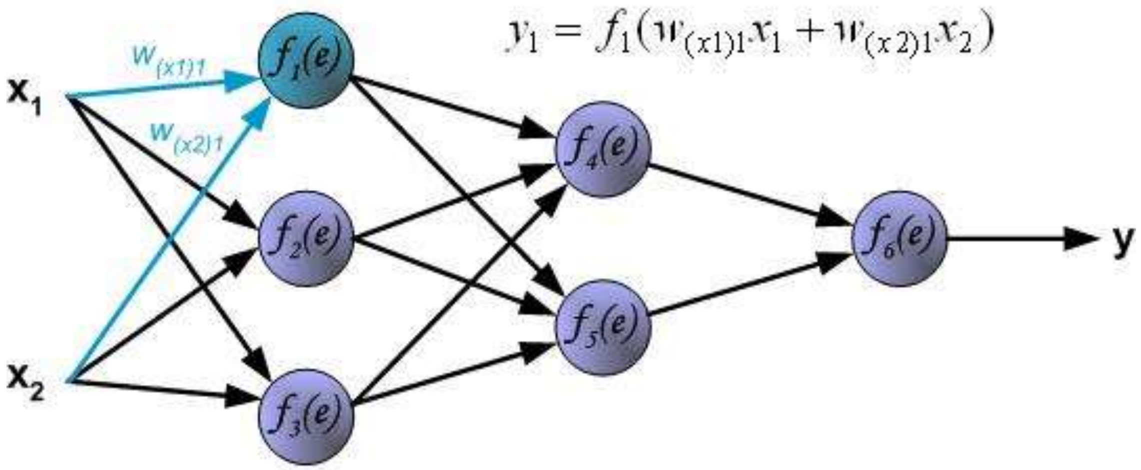
The project describes teaching process of multi-layer neural network employing *backpropagation* algorithm. To illustrate this process the three layer neural network with two inputs and one output,which is shown in the picture below, is used:

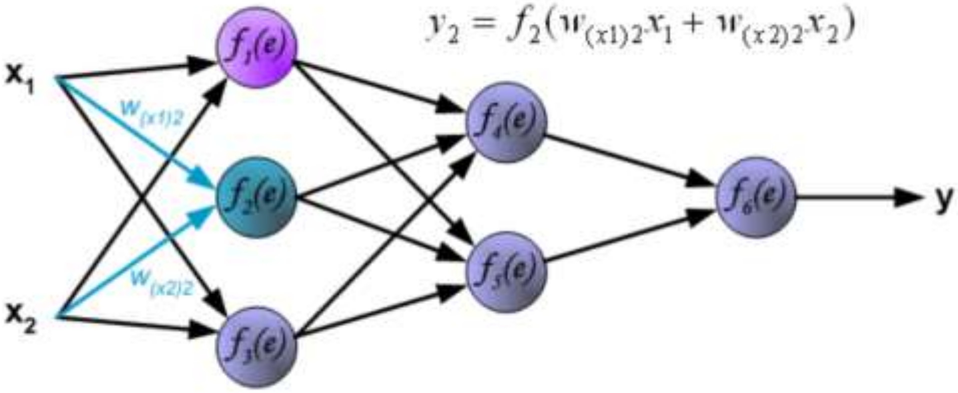


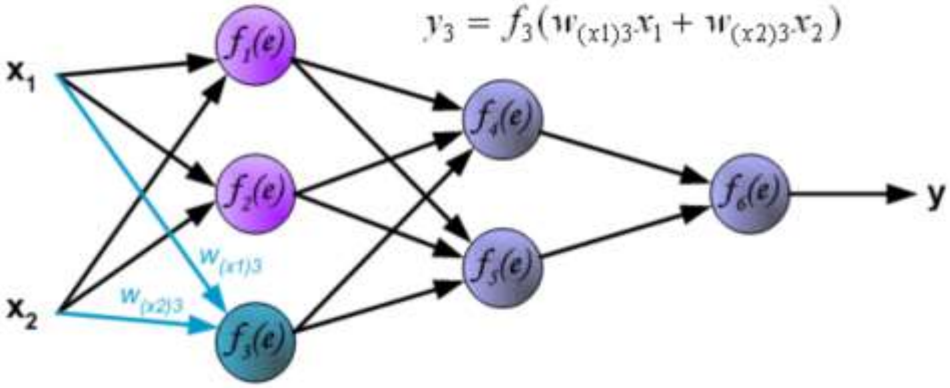
Each neuron is composed of two units. First unit adds products of weights coefficients and input signals. The second unit realise nonlinear function, called neuron activation function. Signal *e* is adder output signal, and *y = f(e)* is output signal of nonlinear element. Signal *y* is also output signal of neuron.



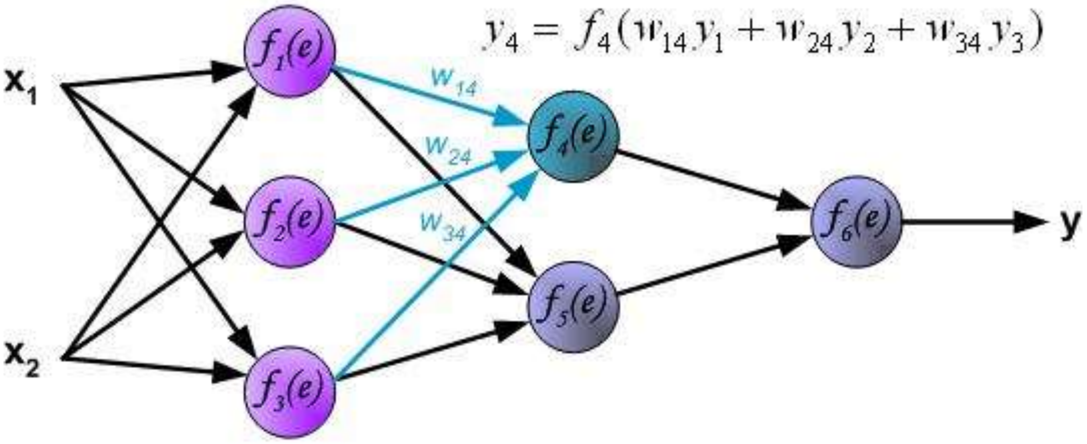
To teach the neural network we need training data set. The training data set consists of input signals (x_1 and x_2) assigned with corresponding target (desired output) *z*. The network training is an iterative process. In each iteration weights coefficients of nodes are modified using new data from training data set. Modification is calculated using algorithm described below: Each teaching step starts with forcing both input signals from training set. After this stage we can determine output signals values for each neuron in each network layer. Pictures below illustrate how signal is propagating through the network, Symbols $w_{(xm)n}$ represent weights of connections between network input x_m and neuron *n* in input layer. Symbols y_n represents output signal of neuron *n*.

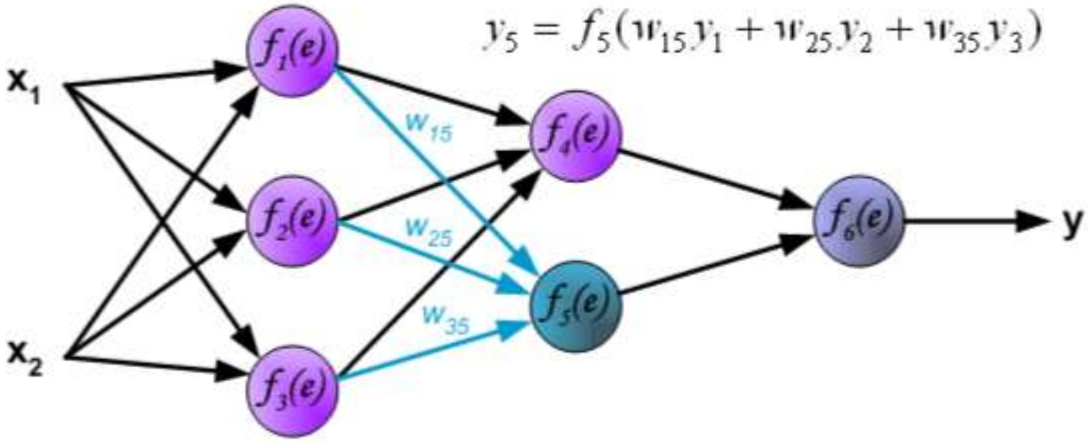




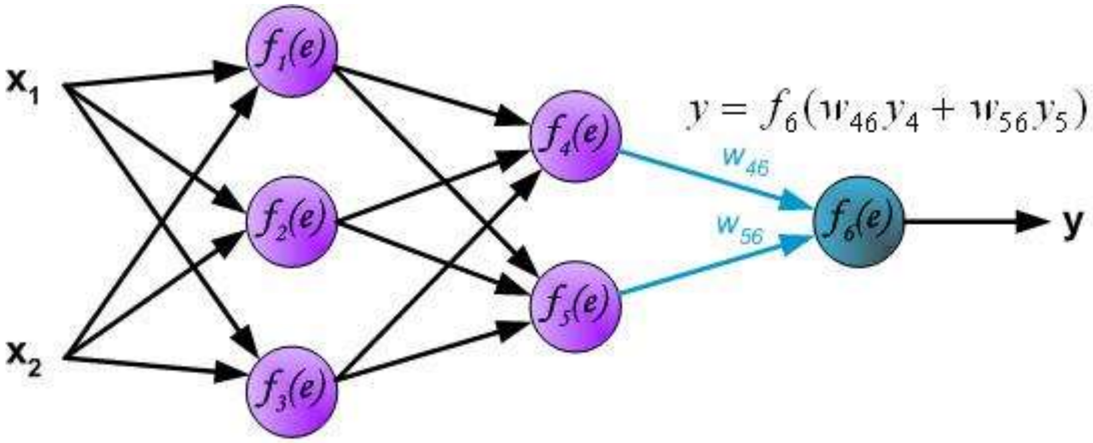


Propagation of signals through the hidden layer. Symbols w_{mn} represent weights of connections between output of neuron *m* and input of neuron *n* in the next layer.

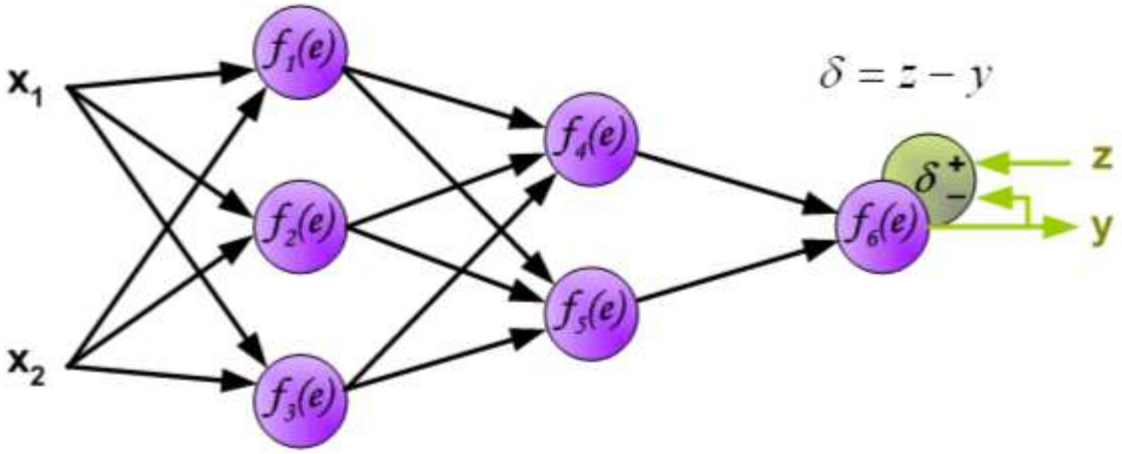




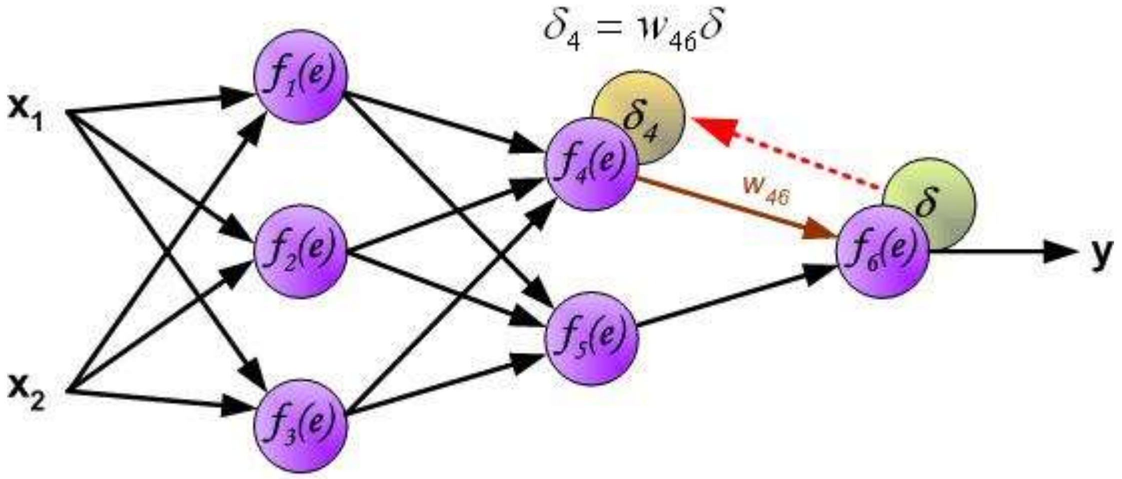
Propagation of signals through the output layer.

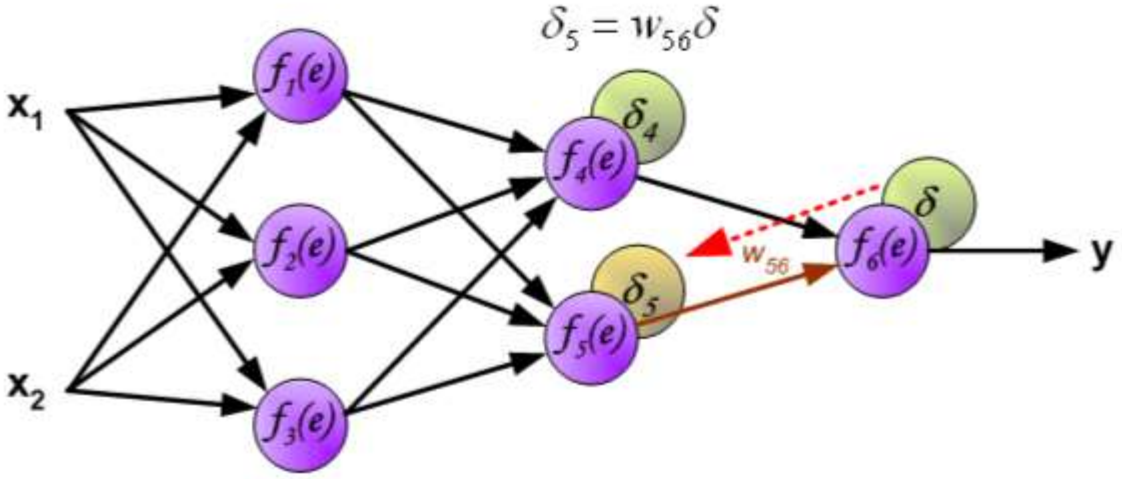


In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron.

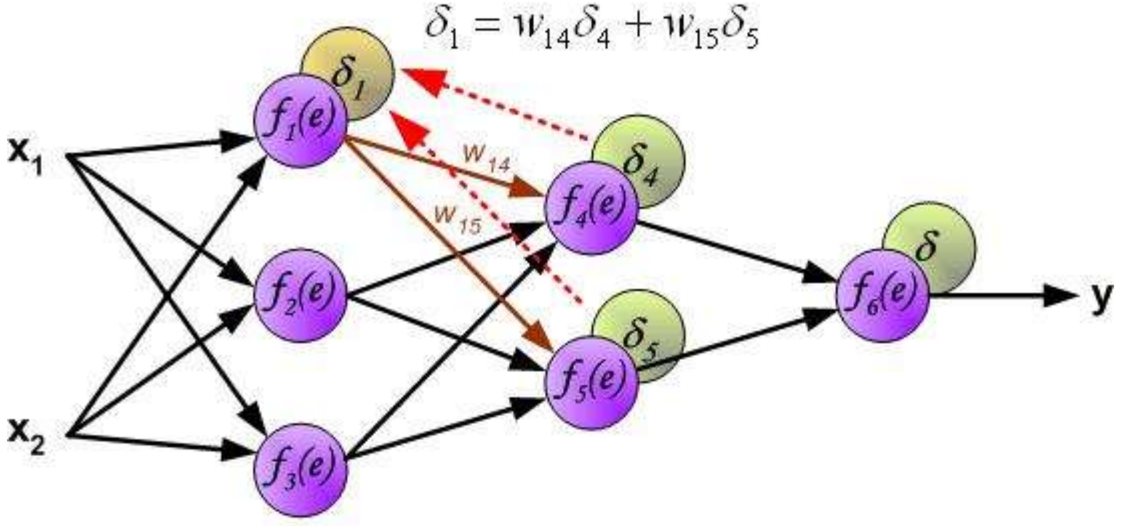


It is impossible to compute error signal for internal neurons directly, because output values of these neurons are unknown. For many years the effective method for training multiplayer networks has been unknown. Only in the middle eighties the back-propagation algorithm has been worked out. The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

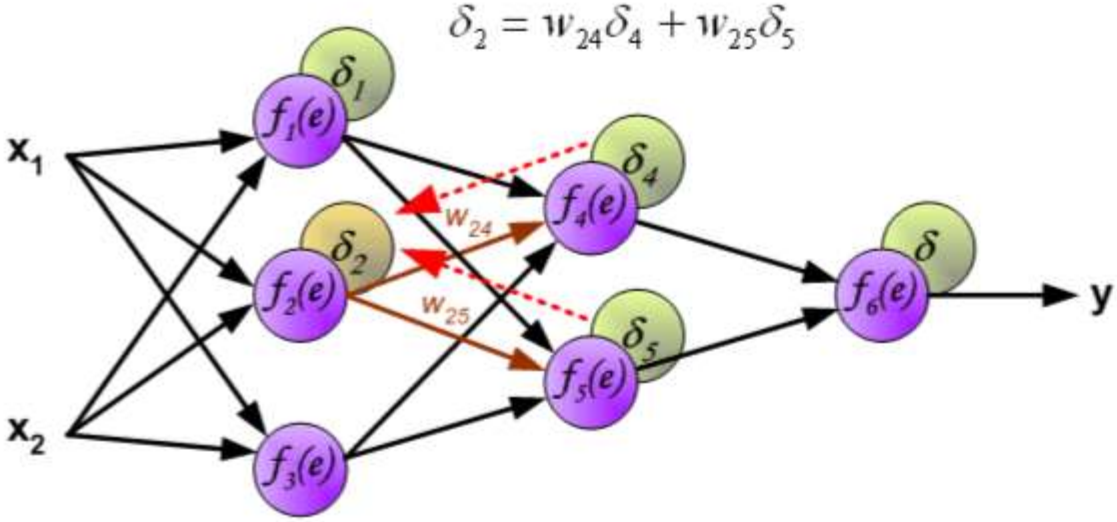




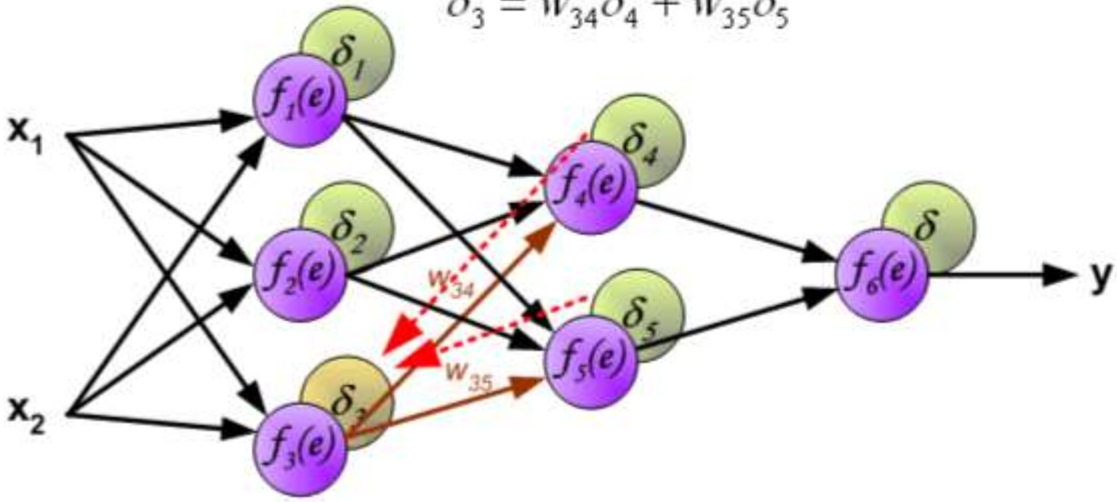
The weights' coefficients w_{mn} used to propagate errors back are equal to this used during computing output value. Only the direction of data flow is changed (signals are propagated from output to inputs one after the other). This technique is used for all network layers. If propagated errors came from few neurons they are added. The illustration is below:



$$\delta_2 = w_{24}\delta_4 + w_{25}\delta_5$$



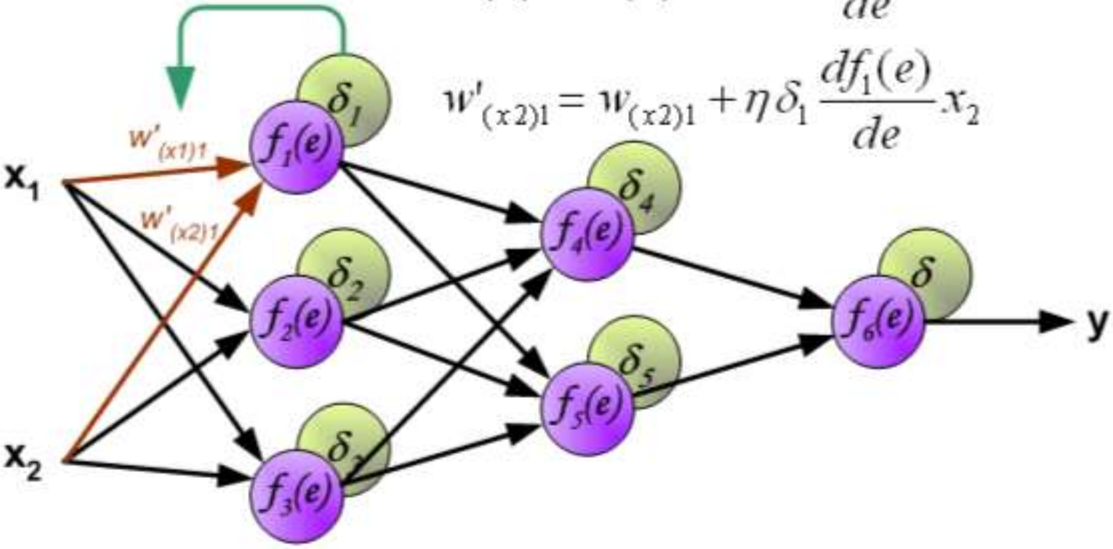
$$\delta_3 = w_{34}\delta_4 + w_{35}\delta_5$$



When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

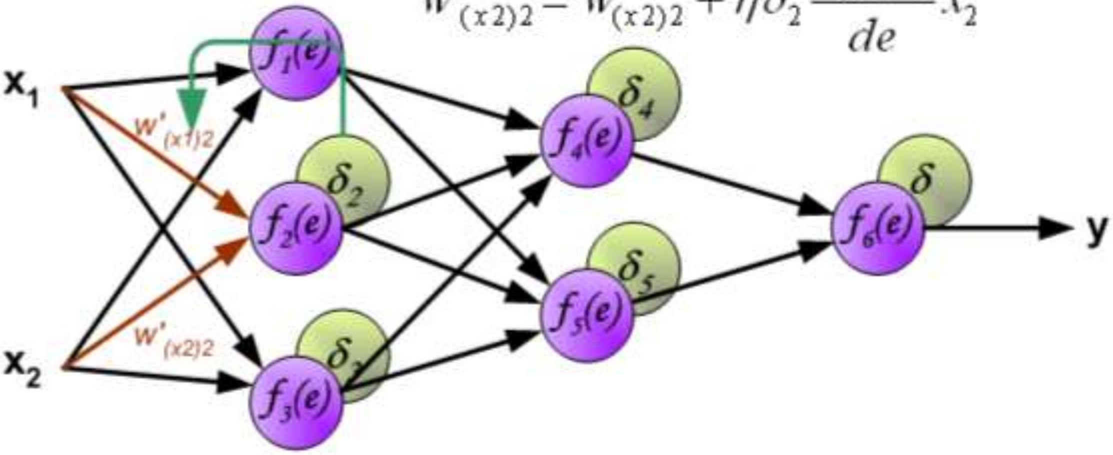
$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$



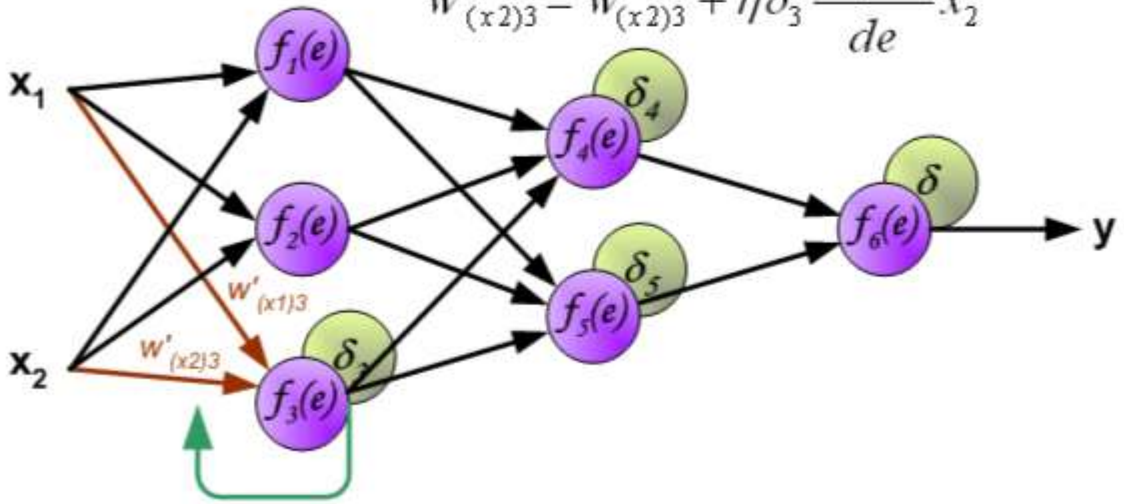
$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$



$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \frac{df_3(e)}{de} x_1$$

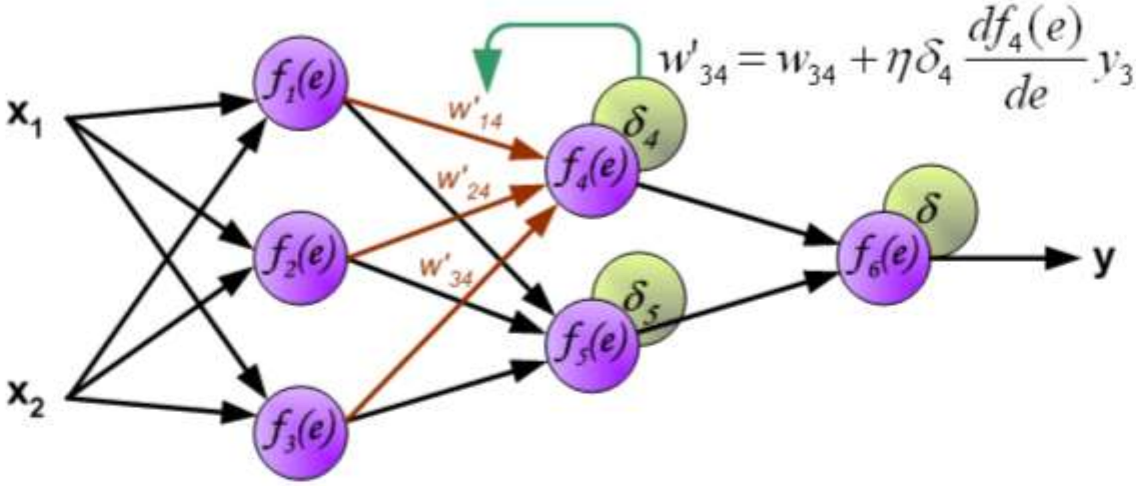
$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \frac{df_3(e)}{de} x_2$$

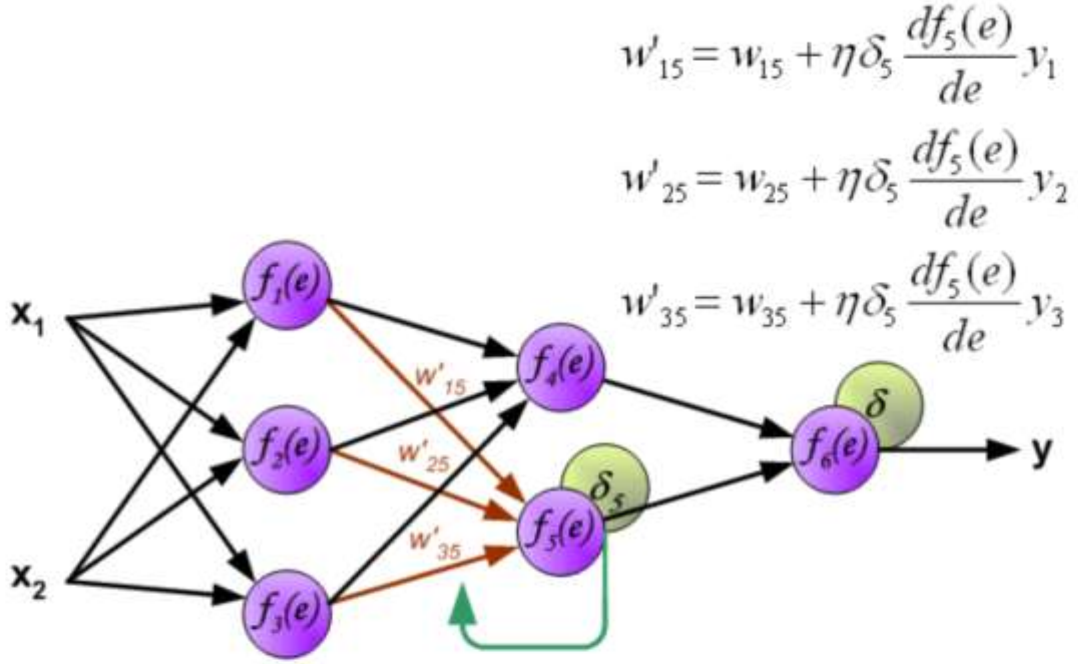


$$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$

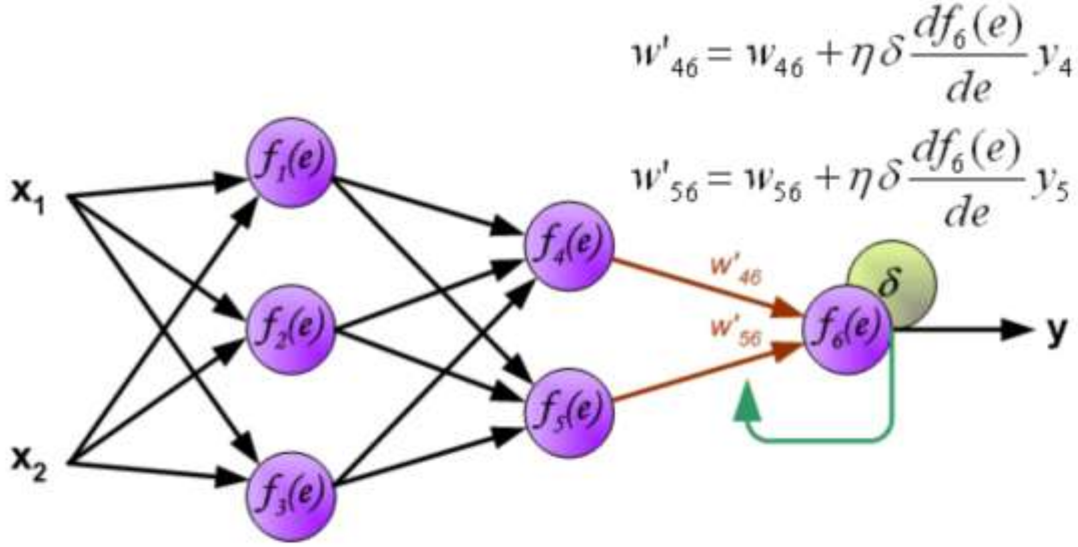




$$w'_{15} = w_{15} + \eta\delta_5 \frac{df_5(e)}{de} y_1$$

$$w'_{25} = w_{25} + \eta\delta_5 \frac{df_5(e)}{de} y_2$$

$$w'_{35} = w_{35} + \eta\delta_5 \frac{df_5(e)}{de} y_3$$



$$w'_{46} = w_{46} + \eta\delta \frac{df_6(e)}{de} y_4$$

$$w'_{56} = w_{56} + \eta\delta \frac{df_6(e)}{de} y_5$$

Coefficient *h* affects network teaching speed. There are a few techniques to select this parameter. The first method is to start teaching process with large value of the parameter. While weights coefficients are being established the parameter is being decreased gradually. The second, more complicated, method starts teaching with small parameter value. During the teaching process the parameter is being increased when the teaching is advanced and then decreased again in the final stage. Starting teaching process with low parameter value enables to determine weights coefficients signs.

11. Application of Artificial Neural Network

ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis of hepatitis; recovery of telecommunications from faulty software; interpretation of multi-meaning words; undersea mine detection; texture analysis; three-dimensional object recognition; hand-written word recognition; and facial recognition.

Detection of Medical Phenomena

A variety of health-related indices (e.g., a combination of heart rate, levels of various substances in the blood, respiration rate) can be monitored. The onset of a particular medical condition could be associated with a very complex (e.g., nonlinear and interactive) combination of changes on a subset of the variables being monitored. Neural networks have been used to recognize this predictive pattern so that the appropriate treatment can be prescribed. Neural networks are ideal in recognizing diseases using scans since there is no need to provide a specific algorithm on how to identify the disease. Neural networks learn by example so the details of how to recognize the disease are not needed. What is needed is a set of examples that are representative of all the variations of the disease.

Stock Market Prediction

Fluctuations of stock prices and stock indices are another example of a complex, multidimensional, but in some circumstances at least partially deterministic phenomenon. Neural networks are being used by many technical analysts to make predictions about stock prices based upon a large number of factors such as past performance of other stocks and various economic indicators.

Credit Assignment

A variety of pieces of information are usually known about an applicant for a loan. For instance, the applicant's age, education, occupation, and many other facts may be available. After training a neural network on historical data, neural network analysis can identify the most relevant characteristics and use those to classify applicants as good or bad credit risks.

Monitoring the Condition of Machinery

Neural networks can be instrumental in cutting costs by bringing additional expertise to scheduling the preventive maintenance of machines. A neural network can be trained to distinguish between the sounds a machine makes when it is running normally ("false alarms") versus when it is on the verge of a problem. After this training period, the expertise of the network can be used to warn a technician of an upcoming breakdown, before it occurs and causes costly unforeseen "downtime."

Engine Management

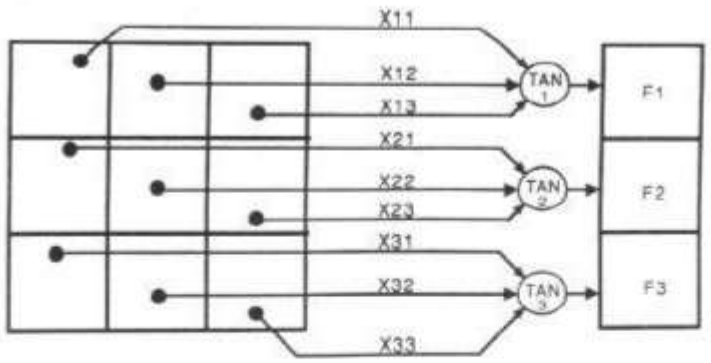
Neural networks have been used to analyze the input of sensors from an engine. The neural network controls the various parameters within which the engine functions, in order to achieve a particular goal, such as minimizing fuel consumption.

Language Processing

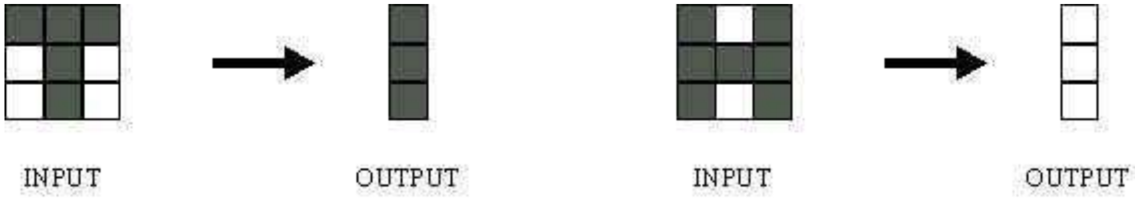
Language processing encompasses a wide variety of applications. These applications include text-to-speech conversion, auditory input for machines, automatic language translation, secure voice keyed locks, automatic transcription, aids for the deaf, aids for the physically disabled, which respond to voice commands, and natural language processing. Many companies and universities are researching how a computer, via ANNs, could be programmed to respond to spoken commands. The potential economic rewards are a proverbial gold mine. If this capability could be shrunk to a chip, that chip could become part of almost any electronic device sold today. Literally hundreds of millions of these chips could be sold. This magic-like capability needs to be able to understand the 50,000 most commonly spoken words. Currently, according to the academic journals, most of the hearing-capable neural networks are trained to only one talker. These one-talker, isolated-word recognizers can recognize a few hundred words. Within the context of speech, with pauses between each word, they can recognize up to 20,000 words. Some researchers are touting even greater capabilities, but due to the potential reward the true progress, and methods involved, are being closely held. The most highly touted, and demonstrated, speech-parsing system comes from the Apple Corporation. This network, according to an April 1992 Wall Street Journal article, can recognize most any person's speech through a limited vocabulary.

Pattern Recognition

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward neural network that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.



For example: The network of figure 1 is trained to recognize the patterns T and H. The associated patterns are all black and all white respectively as shown below.



If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurons after generalization are;

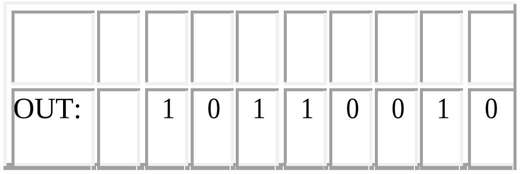
X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Middle neuron

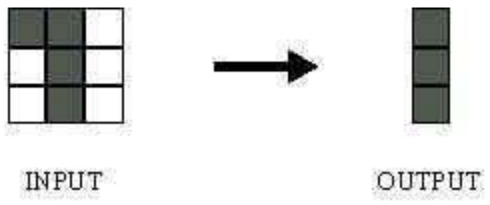
X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1



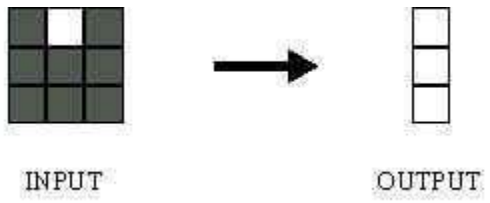
OUT:		1	0	1	1	0	0	1	0

Bottom neuron

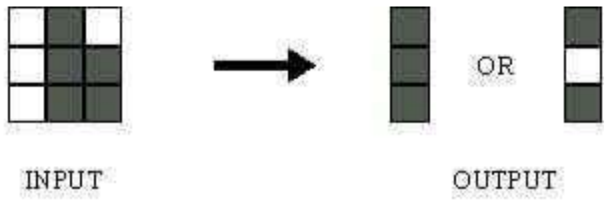
From the tables it can be seen the following associations can be extracted:



In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.



Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.



Here, the top row is 2 errors away from T and 3 from H. So the top output is black. The middle row is 1 error away from both T and H so the output is random. The bottom row is 1 error away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favor of the T shape.

A Brief Summary of Neural Network Types

Artificial neural network (ANN), usually called neural network (NN), is a system of mathematical model or computational model that is inspired by the structure and/or functional aspects of biological neural networks. There are many types of *artificial neural networks* (ANN). Each artificial neural network is a computational simulation of a biological neural network model. Artificial neural network models mimic the real life behavior of neurons and the electrical messages they produce between input (such as from the eyes or nerve endings in the hand), processing by the brain and the final output from the brain (such as reacting to light or from sensing touch or heat). There are other ANNs which are adaptive systems used to model things such as environments and population. Artificial neural network systems can be hardware and software based specifically built systems or purely software based and run in computer models. Some NN types are:

Feed Forward Neural Network - A simple neural network type where synapses are made from an input layer to zero or more hidden layers, and finally to an output layer. The feedforward neural network is one of the most common types of neural network in use. It is suitable for many types of problems. Feedforward neural networks are often trained with simulated annealing, genetic algorithms or one of the propagation techniques.

Self Organizing Map (SOM) - A neural network that contains two layers and implements a winner take all strategy in the output layer. Rather than taking the output of individual neurons, the neuron with the highest output is considered the winner. SOM's are typically used for classification, where the output neurons represent groups that the input neurons are to be classified into. SOM's are usually trained with a competitive learning strategy.

Hopfield Neural Network - A simple single layer recurrent neural network. The Hopfield neural network is trained with a special algorithm that teaches it to learn to recognize patterns. The Hopfield network will indicate that the pattern is recognized by echoing it back. Hopfield neural networks are typically used for pattern recognition.

Simple Recurrent Network (SRN) Elman Style - A recurrent neural network that has a context layer. The context layer holds the previous output from the hidden layer and then echos that value back to the hidden layer's input. The hidden layer then always receives input from its previous iteration's output. Elman neural networks are generally trained using genetic, simulated annealing, or one of the propagation techniques. Elman neural networks are typically used for prediction.

Simple Recurrent Network (SRN) Jordan Style - A recurrent neural network that has a context layer. The context layer holds the previous output from the output layer and then echos that value back to the hidden layer's input. The hidden layer then always receives input from the previous iteration's output layer. Jordan neural networks are generally trained using genetic, simulated annealing, or one of the propagation techniques. Jordan neural networks are typically used for prediction.

Simple Recurrent Network (SRN) Self Organizing Map - A recurrent self organizing map that has an input and output layer, just as a regular SOM. However, the RSOM has a context layer as well. This context layer echo's the previous iteration's output back to the input layer of the neural network. RSOM's are trained with a competitive learning algorithm, just as a non-recurrent SOM. RSOM's can be used to classify temporal data, or to predict.

Feedforward Radial Basis Function (RBF) Network - A feedforward network with an input layer, output layer and a hidden layer. The hidden layer is based on a radial basis function. The RBF generally used is the gaussian function. Several RBF's in the hidden layer allow the RBF network to approximate a more complex activation function than a typical feedforward neural network. RBF networks are used for pattern recognition. They can be trained using genetic, annealing or one of the propagation techniques. Other means must be employed to determine the structure of the RBF's used in the hidden layer.

Reprinted from Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77), August, 1977, p. 247.

Truth Maintenance Systems For Problem Solving

Jon Doyle
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

This summarizes the full report (Jon Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab TR-419) which describes progress that has been made in the ability of a computer system to understand and reason about its own reasoning faculties. A new method for representing knowledge about beliefs has been developed. This representation, called a non-monotonic dependency system, extends and clarifies several similar previous representation forms for such knowledge, and has been employed in developing new strategies for representing assumptions, backtracking, and controlling problem solving systems.

A truth maintenance system is a combination of a representation for recording justifications for program beliefs and procedures for effecting any updating of beliefs necessary upon the addition of new information. Such a system can easily be used by processes for reasoning about the recorded program reasoning. In particular, processes for non-chronological, dependency-directed backtracking and hypothetical reasoning are particularly straightforward in implementation given the representations of a truth maintenance system.

The basic operation of a truth maintenance system is to attach a justification to a fact. A fact can be linked with any component of program knowledge which is to be connected with other components of program information. Typically, a fact might be connected with each assertion and rule in a data base, or might be attached, with differing meanings, to various subsystem structures. The truth maintenance system decides, on the basis of the justifications attached to facts, which beliefs in the truth of facts are supported by the recorded justifications.

A belief may be justified on the basis of several other beliefs, by the conditional proof on one belief relative to other beliefs, or by the lack of belief in some fact. The latter form of justification allows the consistent representation and maintenance of hypothetical assumptions.

Truth maintenance processing is required when new justifications change previously existing beliefs. In such cases, the status of all beliefs depending on the changed beliefs must be redetermined. From the justifications used in this judgement of beliefs, a number of dependencies between be-liefs are determined, such as the set of beliefs depending on each particular belief or the beliefs upon which a particular belief depends.

Several useful processes are supported by the above func-tions and representations. It is a straightforward matter to interrogate the truth maintenance system representation for the basic material of explanations of beliefs. More sophisti-cated uses of the recorded justifications are in hypothetical reasoning, generalization, separation of levels of detail, and in dependency-directed backtracking.

Hypothetical reasoning is supported by the use of con-ditional proof justifications. These are justifications which support belief if a specified belief follows from a set of other beliefs. This capability is instrumental in summarizing dis-coveries in a manner independent of the hypotheses leading to their derivation.

The processes of generalization and separation of levels of detail are also supported by the mechanism of conditional proof. By using conditional proofs to remove dependence of beliefs on other beliefs, results can be justified independent of the particular quantities used in their computation, and results at one level of detail can be supported by reasons which are independent of results at lower levels of detail.

Dependency-directed backtracking is a powerful technique based on the representations of the truth maintenance system. This method employs the recorded dependencies to locate precisely those hypotheses relevant to the failure and uses the conditional proof mechanism to summarize the cause of the contradiction in terms of these hypotheses. Be-cause the failure is summarized independent of the hypothe-ses causing the failure, future occurrences of the failure are avoided.

This research is supported by a Fannie and John Hertz Foundation graduate fellowship. This research was con-ducted at the Artificial Intelligence Laboratory of the Mas-sachusetts Institute of Technology. Support for the labo-ratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Depart-ment of Defense under Office of Naval Research contract N00014-75-C-0643.