DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SEMESTER -VI

CS1351 – ARTIFICIAL INTELLIGENCE

PREPARED BY

H.Prabha

Lecturer/CSe

CS1351 – ARTIFICIAL INTELLIGENCE

UNIT I FUNDAMENTALS 8

Intelligent agents − Agents and environments − Good behavior − The nature of environments − Structure of agents − Problem solving − Problem solving agents −Example problems − Searching for solutions − Uniformed search strategies −Avoiding repeated states − Searching with partial information.

UNIT II SEARCHING TECHNIQUES 10

Informed search and exploration − Informed search strategies − Heuristic function − Local search algorithms and optimistic problems − Local search in continuous spaces − Online search agents and unknown environments − Constraint Satisfaction Problems(CSP) − Backtracking Search and Local Search for CSP − Structure of problems − Adversarial search − Games − Optimal decisions in games − Alpha-Beta pruning − Imperfect real-time decision − Games that include an element of chance.

UNIT III KNOWLEDGE REPRESENTATION 10

First order logic − Representation revisited − Syntax and semantics for first order logic − Using first order logic − Knowledge engineering in first order logic − Inference in first order logic − Propositional versus first order logic − Unification and lifting − Forward chaining − Backward chaining − Resolution – Knowledge representation − Ontological engineering − Categories and objects − Actions − Simulation and events − Mental events and mental objects.

UNIT IV LEARNING 9

Learning from observations − Forms of learning − Inductive learning – Learning decision trees − Ensemble learning − Knowledge in learning − Logical formulation of learning − Explanation based learning − Learning using relevant information − Inductive logic programming − Statistical Learning Methods − Learning with Complete Data − Learning with Hidden Variable − EM Algorithm − Instance Based Learning − Neural Networks − Reinforcement Learning − Passive Reinforcement Learning − Active reinforcement learning − Generalization in reinforcement learning.

UNIT V APPLICATIONS 8

Communication − Communication as action − Formal grammar for a fragment of english − Syntactic analysis − Augmented grammars − Semantic interpretation − Ambiguity and disambiguation − Discourse understanding − Grammar Induction − Probabilistic language processing − Probabilistic language models – Information Retrieval − Information extraction − Machine translation.

L:45 T:15 Total : 60

TEXT BOOKS

1. Stuart Russell and Peter Norvig, "Artificial Intelligence−A Modern Approach", 2nd Edition, Pearson Education / Prentice Hall of India, 2004.
2. Nilsson, N.J., "Artificial Intelligence: A new Synthesis", Elsevier, 2003.

REFERENCES

1. Elaine Rich and Kevin Knight,"Artificial Intelligence",2nd Edition,TataMcGraw-Hill 2003.
2. Luger, G.F., "Artificial Intelligence-Structures and Strategies for Complex Problem Solving", Pearson Education / PHI, 200

UNIT – I

FUNDAMENTALS

Intelligent agents − Agents and environments − Good behavior − The nature of environments − Structure of agents − Problem solving − Problem solving agents − Example problems − Searching for solutions − Uniformed search strategies − Avoiding repeated states − Searching with partial information.

1.1 Introduction to AI

What is an AI?

Artificial Intelligence is the study of how to make computers do things which, at the moment, people do better.

It leads **FOUR** important categories.

i) Systems that think like humans
ii) Systems that act like humans
iii) Systems that think rationally
iv) Systems that act rationally

Acting humanly: The Turing Test approach

To conduct this test, we need two people and the machine to be evaluated. One person plays the role of the interrogator, who is in a separate room from the compute and the other person. The interrogator can ask questions of either the person or the computer by typing questions and receiving typed responses. However, the interrogator knows them only as A and B and aims to determine which the person is and which are the machine. The goal of the machine is to fool the interrogator into believing that is the person. If the machine succeeds at this, then we will conclude that the machine is acting humanly. But programming a computer to pass the test provides plenty to work on, to posses the following capabilities.

- **Natural Language Processing** – to enable it to communicate successfully in English.
- **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 adopt to new circumstances and to learn from experience , example etc.,

Total Turing Test: The test which includes a video signal so that the interrogator can test the perceptual abilities of the machine. To undergo the total Turing test, the computer will need:

- **Computer Vision** - to perceive objects
- **Robotics** - to move them.

Thinking Humanly: The Cognitive modeling approach

To construct a machine program to think like a human, first it requires the knowledge about the actual workings of human mind. After completing the study about human mind it is possible to express the theory as a computer program. It the program’s input/output and timing behavior matches with the human behavior then we can say that the program’s mechanism is working like a human mind.

Example: General Problem Solver (GPS)

Thinking rationally:The laws of thought approach

The right thinking introduced the concept of logic.

Example: Ram is a student of III year CSE.

All students are good in III year in CSE.

Ram is a good student.

Acting rationally: The rational agent approach

Acting rationally means, to achieve one's goal given one's beliefs. In the previous topic laws of thought approach, correct inference is selected, conclusion is derived, but the agent acts on the conclusion defined the task of acting rationally. The study of rational agent has **TWO** advantages:

i. Correct inference is selected and applied.
ii. It concentrates on scientific development rather than other methods.

The History of AI

1943	McCulloch & Pitts: Boolean circuit of model of brain
1950	Turing's "Computing Machinery and Intelligence"
1950s	Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
1956	Dartmouth meeting: " Artificial Intelligence" adopted
1965	Robinson's complete algorithm for logical reasoning
1960 – 74	AI discovers computational complexity Neural network research almost disappears
1969 – 79	Early development of knowledge based systems
1980 – 88	Expert systems industry booms
1988 – 93	Expert systems industry busts: "AI winter"

1985 – 95	Neural networks return to popularity
1988	Resurgence of probability; general increase in technical path " Nouvelle AI";ALife, GA's,soft computing
1995	Agents
2003	Human-level AI back on the agenda

1.2 Intelligent Agents

1.3 Agents and Environments

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators** shown in Figure.

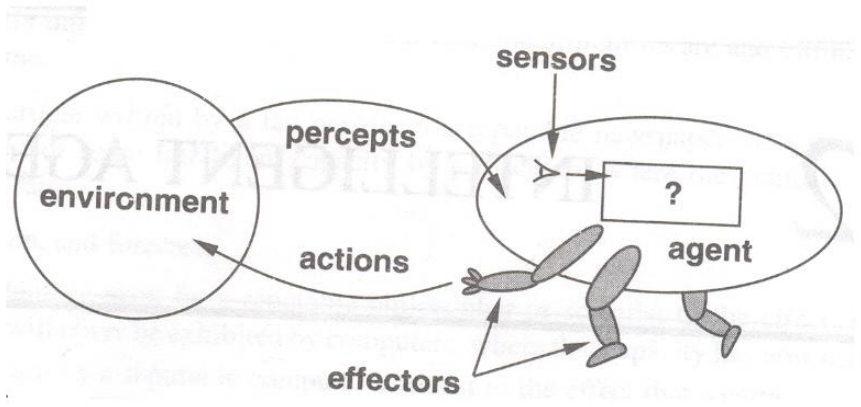


Figure: Agents interact with environment through sensors and effectors

The different types of agent are:

- **Human Agent**: A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth and other body parts for actuators.

- **Robotic Agent**: A robotic agent has cameras and infrared range finders for the sensors and various motors for the actuators.

- **Software Agent**: A software agent has encoded bit strings as its percepts and actions.

- **Generic Agent**: A general structure of an agent who interacts with the environment.

The **agent function** for an agent specifies the action taken by the agent in response to any percept sequence. It is an abstract mathematical description

Internally, the agent function for an artificial agent will be implemented by an agent program. An **agent program** is a function that implements the agent mapping from percepts to actions. It is a concrete implementation, running on the agent architecture.

1.4 Good Behavior: The Concept of Rationality

An agent should act as a Rational Agent. A **rational agent** is one that does the right thing that is the right actions will cause the agent to be most successful in the environment.

Performance measures

A **performance measures** embodies the criterion for success of an agent's behavior. As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a **definition of a rational agent** (ideal rational agent)

“*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, that is the task of rational agent is to improve the performance measure depends on percept sequence*”

Omniscience, learning, and autonomy

An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.

A rational agent not only to gather information, but also to **learn** as much as possible from what it perceives. The agent’s initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented.

Successful agents **split the task** of computing the agent function into **three different periods:** when the agent is being designed, some of the computation is done by its designers; when it is deliberating on its next action, the agent does more computation; and as it learns from experience, it does even more computation to decide how to modify its behavior.

A rational agent should be **autonomous** – it should learn what it can to compensate for partial or incorrect prior knowledge.

1.5 The Nature of Environments

Specifying the task environment

A **task environment** specification includes the performance measure, the external environment, the actuators, and the sensors. In designing an agent, the first step must always be to specify the task environment as fully as possible. Task environments are specified as a **PAGE** (**P**ercept, **A**ction, **G**oal, **E**nvironment) (or) **PEAS** (**P**erformance, **E**nvironment, **A**ctuators, **S**ensors) description, both means the same.

The following table describes some of the agent types and the basic **PEAS** description.

Agent Type	**Performance Measure**	**Environment**	**Actuators**	**Sensors**
Taxi Driver	Safe, fast, legal, comfortable trip, maximize profits	Roads and other traffic, pedestrians, customers	Steering, accelerator, brake, signal horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive	Maximize student's score	Set of students,	Display exercises,	Keyboard entry

English tutor	on test	testing agency	suggestions, corrections	

Table: Examples of agent types and their PEAS descriptions

Some **Software Agents** (or) **Software Robots** (or) **Soft bots** exist in rich, unlimited domains. So,

- Actions are done in real time.
- Action is performed in the complex environment.
- It is designed to scan online news sources and shows the actions with natural language processing.

Properties of task environments

1. Accessible vs. inaccessible (or) **fully observable vs. partially observable**

If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is **fully observable**. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world.

An environment might be **partially observable** because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

2. Deterministic vs. non-deterministic (or) **stochastic**

If the next state of the environment is completely determined by the current state and the action executed by the agent, then the environment is **deterministic**; otherwise, it is **stochastic**. In principle, an agent need not worry about uncertainty in a **fully observable**, **deterministic** environment. If the environment is **partially observable,** however, then it could appear to be stochastic.

3. Episodic vs. non-episodic (or) **sequential**

In an **episodic environment**, the agent's experience is divided into atomic episodes. Each episode consists of its own percepts and actions and it does not depend on the previous episode.

In **sequential environments**, the current decision could affect all future of decisions. Eg. Chess and taxi driving.

Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

4. Static vs. dynamic

If the environment is changing for agent's action then the environment is **dynamic** for that agent otherwise it is **static**.

If the environment does not change for some time, then it changes due to agent's performance is called **semi dynamic environment**.

E.g.

Taxi driving is **dynamic**.

Chess is **semi dynamic**.

Crossword puzzles are **static**.

5. Discrete vs. continuous

If the environment has limited number of distinct, clearly defined percepts and actions then the environment is **discrete**. E.g. **Chess**

If the environment changes continuously with range of value the environment is **continuous**. E.g. **Taxi driving**.

6. Single agent vs. multiagent

The following Table lists the properties of a number of familiar environments

Task Environment	**Observable**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Agents**
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

1.6 The Structure of Agents

An intelligent agent is a combination of Agent Program and Architecture.

Intelligent Agent = Agent Program + Architecture

Agent Program is a function that implements the agent mapping from percepts to actions. There exists a variety of basic agent program designs, reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.

Architecture is a computing device used to run the agent program.

To perform the mapping task **four types of agent programs** are there. They are:

1. Simple reflex agents
2. Model-based reflex agents
3. Goal-based agents
4. Utility-based agents

We then explain in general terms **how to convert** all these into **learning agents**.

1. <u>Simple reflex agents</u>

The simplest kind of agent is the simple reflex agent. It responds directly to percepts i.e. these agent select actions on the basis of the current percept, ignoring the rest of the percept history.

An agent describes about how the **condition – action rules** allow the agent to make the connection from percept to action.

Condition – action rule: if condition then action

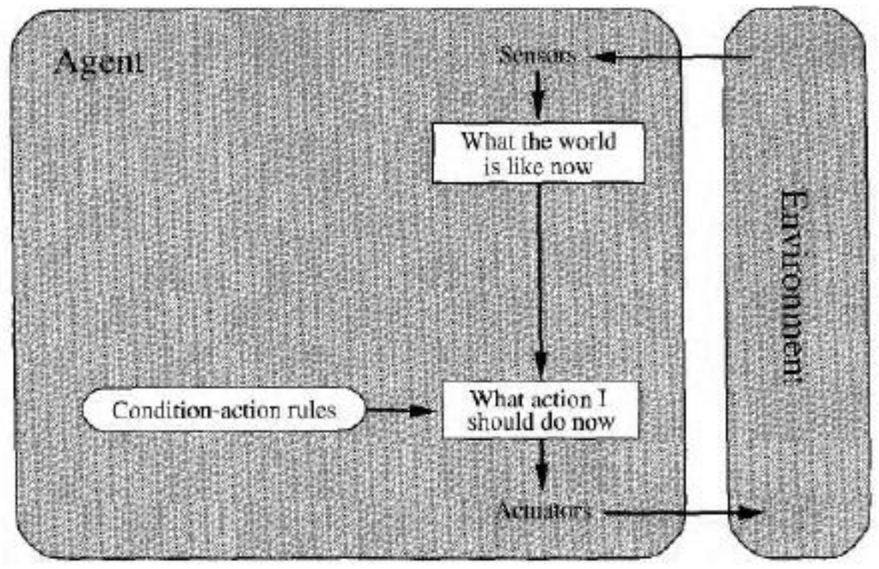


Figure: **Schematic diagram of a simple reflex agent**

Figure: gives the structure of this general program in schematic form, showing how the condition – action rules allow the agent to make the connection from percept to action. In the above schematic diagram, the shapes are described as:

Rectangle – to denote the current internal state of the agent's decision process.

Oval – to represent the background information in the process.

The agent program, which is also very simple, is shown in the following figure.

```
function SIMPLE-REFLEX-AGENT (percept) returns an action
    static: rules, a set of condition-action rules
    state ← INTERPRET – INPUT(percept)
    rule ← RULE – MATCH(state, rules)
    action ← RULE – ACTION[rule]
    return action
```

Figure: A simple reflex agent

INTERRUPT-INPUT **-** function generates an abstracted description of the current state from the percept.

RULE-MATCH - function returns the first rule in the set of rules that matches the given state description.

RULE-ACTION - the selected rule is executed as action of the given percept.

The agent in figure will work only "*if the correct decision can be made on the basis of only the current percept – that is, only if the environment is fully observable*".

Example: Medical diagnosis system

if the patient has reddish brown spots **then** start the treatment for measles.

2. <u>Model-based reflex agents</u> (Agents that keep track of the world)

The most effective way to handle partial observability is for the agent "*to keep track of the part of the world it can't see now"*. That is, the agent which combines the current percept with the old internal state to generate updated description of the current state.

The current percept is combined with the old internal state and it derives a new current state is updated in the state description is also. This updation requires **two kinds of knowledge** in the agent program. **First**, we need some information about how the world evolves independently of the agent. **Second**, we need some information about how the agent's own actions affect the world.

The above two knowledge implemented in simple Boolean circuits or in complete scientific theories – is called a **model** of the world. An agent that uses such a model is called a **model- based agent**.

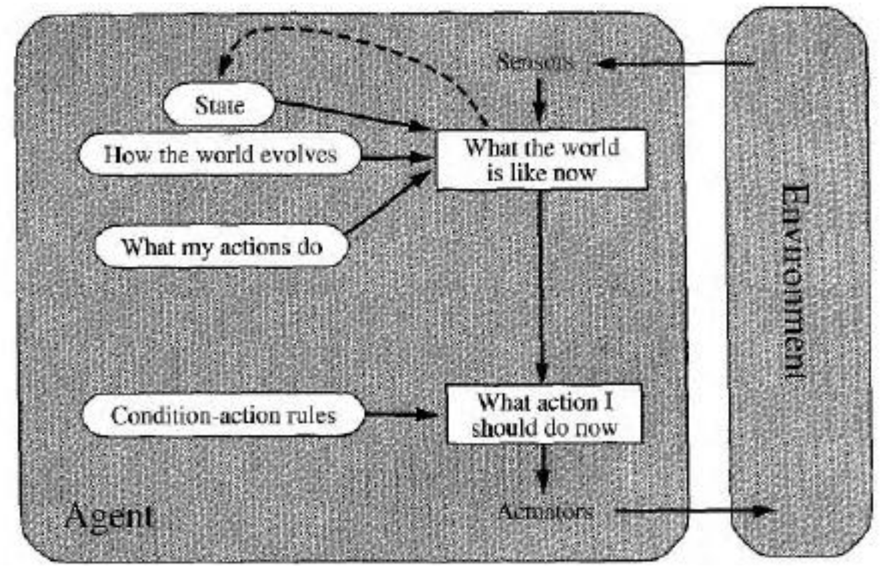


Figure: A model – based reflex agent.

The above figure shows the structure of the reflex agent with internal state, showing how the current percept id combined with the old internal state to generate the updated description of the current state. The agent program is shown in figure.

```
function REFLEX-AGENT-WITH-STATE (percept) returns an action
    static: state, a description of the current world state
            rules, a set of condition-action rules
            action, the most recent action, initially none
    state ← UPDATE-STATE(state, action, percept)
    rule ← RULE-MATCH(state, rules)
    action ← RULE-ACTION[rule]
return action
```

Figure: A model-based reflex agent. It keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

UPDATE-STATE - This is responsible for creating the new internal state description by combining percept and current state description.

3. Goal-based agents

An agent knows the description of current state and also needs some sort of **goal** information that describes situations that are desirable. The action matches with the current state is selected depends on the **goal** state.

The goal based agent is more flexible for more than one destination also. After identifying one destination, the new destination is specified, goal based agent is activated to come up with a new behavior. **Search** and **Planning** are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

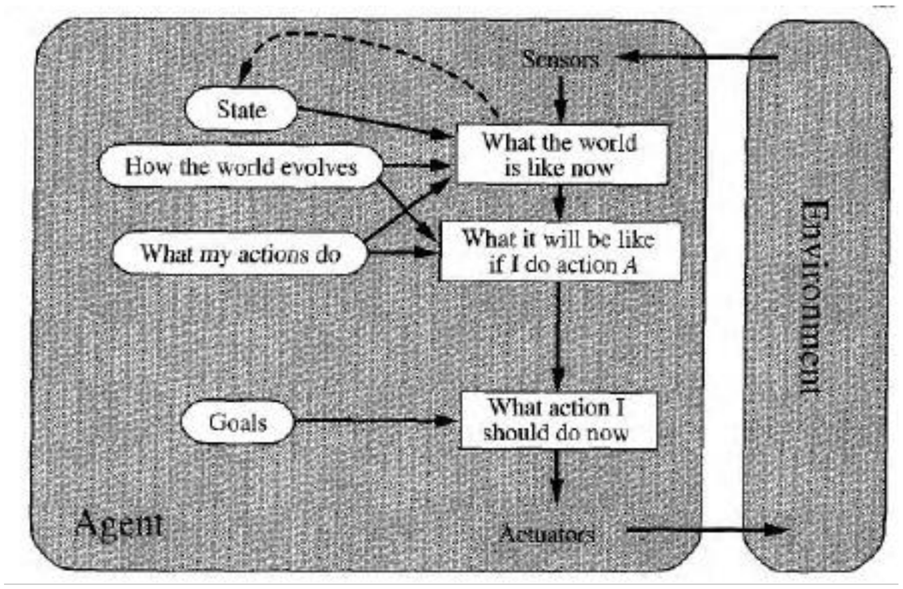

Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Figure: A model-based, goal-based agents

The goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. The goal-based agent's behavior can easily be changed to go to a different location.

4. <u>Utility-based agents</u> (Utility – refers to " the quality of being useful")

An agent generates a goal state with high – quality behavior (utility) that is, if more than one sequence exists to reach the goal state then the sequence with more reliable, safer, quicker and cheaper than others to be selected.

A **utility function** maps a state (or sequence of states) onto a real number, which describes the associated degree of happiness. The utility function can be used for **two different cases**: **First**, when there are conflicting goals, only some of which can be achieved (for e.g., speed and safety), the utility function specifies the appropriate tradeoff. **Second**, when the agent aims for several goals, none of which can be achieved with certainty, then the success can be weighted up against the importance of the goals.

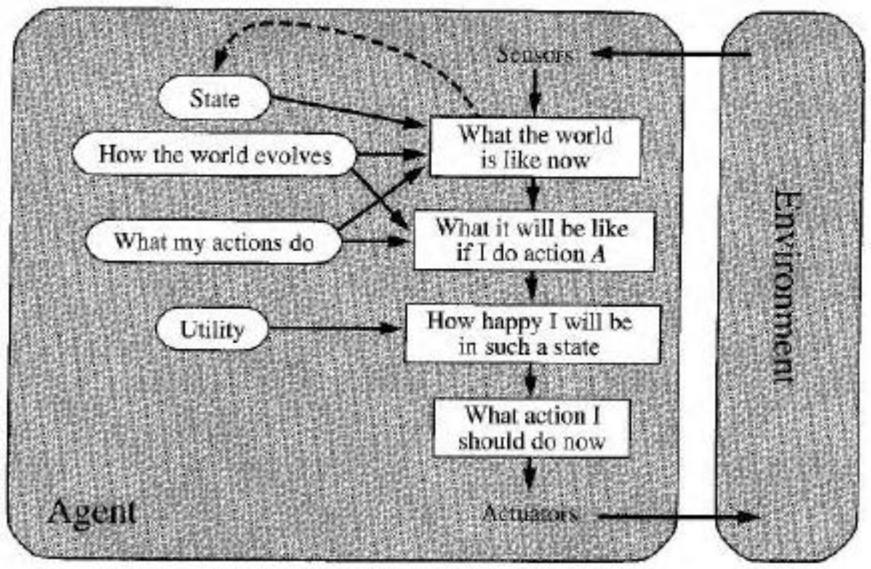

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure: A model-based, utility-based agent

Learning agents

The **learning** task allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge.

A learning agent can be divided into **four conceptual components**, a shown in Figure. They are:

Learning element - This is responsible for making improvements. It uses the feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future.

Performance element - which is responsible for selecting external actions and it is equivalent to agent: it takes in percepts and decides on actions.

Critic - It tells the learning element how well the agent is doing with respect to a fixed performance standard.

Problem generator - It is responsible for suggesting actions that will lead to new and informative experiences.

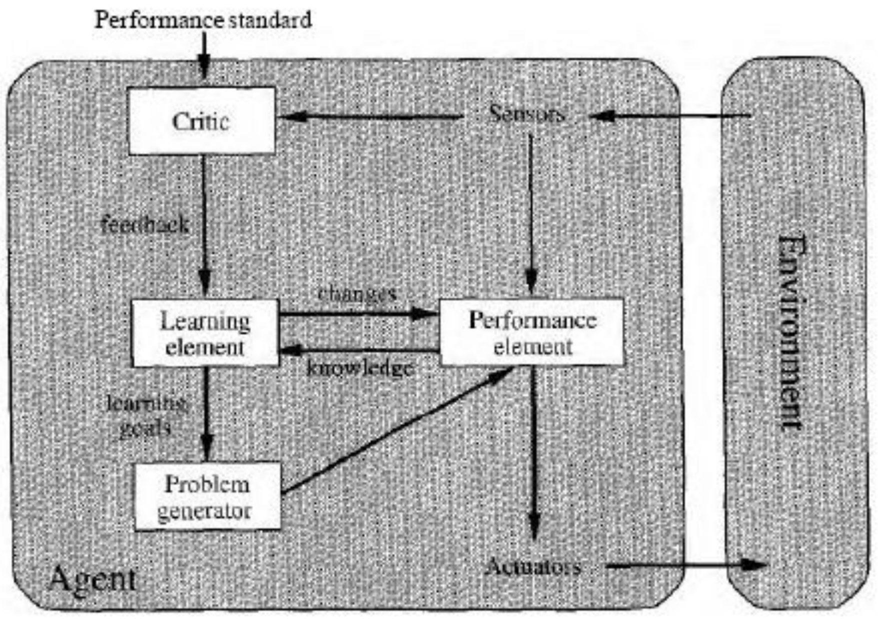


Figure: A general model of learning agents

In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. **Learning** in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent (All agents can improve their performance through **learning**).

1.7 Solving Problems by Searching

Problem –Solving Agents

Problem solving agent is one kind of goal-based agent, where the agent decide what do by finding sequences of actions that lead to desirable states. If the agent understood the definition of problem, it is relatively straight forward to construct a search process for finding solutions, which implies that problem solving agent should be an intelligent agent to maximize the performance measure.

The sequence of steps done by the intelligent agent to maximize the performance measure:

1. Goal formulation - based on the current situation and the agent's performance measure, is the first step of problem solving.

2. Problem formulation - is the process of deciding what actions and states to consider, given a goal.

3. Search - An agent with several immediate options of unknown value can decide what to do by first examining different possible sequence of actions that lead to states of known value, and then choosing in best sequence. This process of looking for such a sequence is called search.

4. Solution - a search algorithm takes a problem as input and returns a solution in the form of an action sequence.

5. Execution phase - Once a solution is found, the actions it recommends can be carried out.

The figure shown as a simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actins that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

```
function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

 state ← UPDATE-STATE (state, percept)
 if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
 action ← FIRST(seq)
 seq ← REST(seq)
 return action
```

Figure: A simple problem-solving agent.

Note: First - first action in the sequence

Rest - returns the remainder

Search- choosing the best one from the sequence of actions

Formulate problem - sequence of actions and states that lead to goal state

Update state - initial state is forced to next state to reach the goal state

Well-defined problems and solutions

A Problem can be defined formally by four components:

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent. The most common formulation uses a **successor function**. Given a particular state **x**, **SUCCESSOR-FN(x)** returns a set of **< action, successor >** ordered pairs, reachable from x by any single action.
  Together, the initial state and successor function implicitly define the **state space** of the problem- the set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes and actions. A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal state. If more than one goal state exists, then we can check whether any on of the goal state is reached or not.
- A **path cost** function that assigns a numeric cost to each path. The cost of a path can be described as the sum of the costs of the individual actions along the path. The **step cost** of taking action *a* to go from *x* to state *y* is denoted by *c(x, a, y).*

The preceding elements define a problem can be gathered together into a single data structure that is given as input to a problem-solving algorithm. A **solution** to a problem is a path from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

Formulating problems

We derive a formulation of the problem in terms of the **initial state**, **successor function**, **goal test**, and **path cost**.

The process of removing detail from a representation is called **abstraction**. In addition to abstracting the state description, we must abstract the actions themselves.

data type PROBLEM

Components: initial state, successor function, goal-test, path-cost.

1.8 Example Problems

Toy problems

i. The 8 – puzzle problem

The 8 – puzzle consists of a 3 × 3 board with eight numbered tiles and a blank space a shown in figure 3.2. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state. The standard formulation is as follows:

- ✓ **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- ✓ **Initial state**: Any state can be designated as the initial state.
- ✓ **Successor function**: This generates the legal states that result from trying the four actions (blank moves Left, Right, Up and Down).
- ✓ **Goal test**: This checks whether the state matches the goal configuration.
- ✓ **Path cost**: Each step costs 1, so the path cost is the number of steps in the path (length of the path).

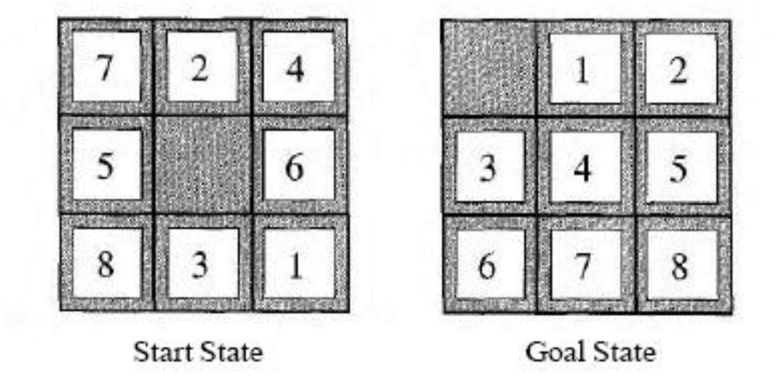


Figure: A typical instance of the 8 – puzzle

ii. The 8 – queen's problem

The goal of the 8 – queen's problem is to place eight queens on a chessboard such that no queen attacks any other in the same row, column or diagonal shown in figure. There are **two main kinds of formulation**. An **incremental formulation** involves operators that augment the state description, starting with an empty state; for the 8 – queen's problem, this means that each action adds a queen to the state.

The incremental formulation is as follows:

- ✓ **States**: Any arrangement of 0 to 8 queens on the board is a state.
- ✓ **Initial state**: No queens on the board.
- ✓ **Successor function**: Add a queen to any empty square.
- ✓ **Goal test**: 8 queens are on the board, none attacked.

In this formulation, we have 3 × 1014 possible sequences to investigate.

A **complete state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts.

A better formulation would prohibit placing a queen in any square that is already attacked:

- ✓ **States**: Arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another are states.

- ✓ **Successor function**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8 – queens' state space from 3 × 1014 to just 2,057, and solutions are easy to find.

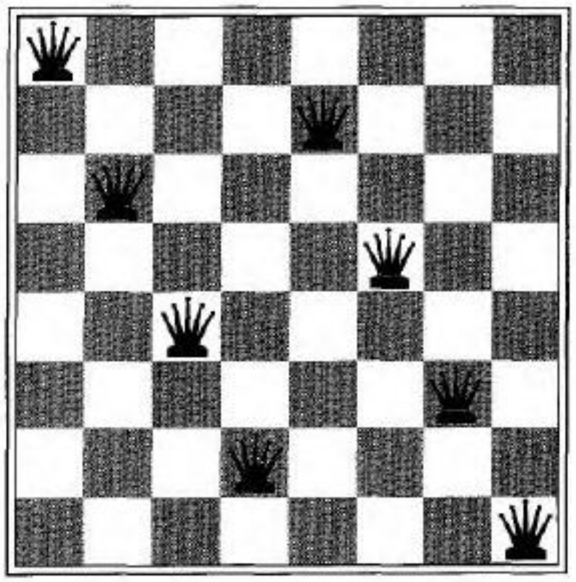


Figure: The 8 – queen's problem

Real-world problems

Name of the problems	**Applications**
i) Route finding	Routing in computer networks, military operations planning and air line travel planning systems.
ii) Touring and traveling salesperson problem	Shortest path tour.
iii) VLSI layout	Cell layout, channel routing

iv) Robot navigation	Route finding problem in continuous space.
v) Automatic assembly sequencing	Assembly of complex objects by Robot. E.g. Electric motors.

1.9 Searching for solutions

We need to solve the formulated problems are done by a search techniques through the state space that use an **search tree** that is generated by the initial state and the successor function that together define the state space. The root of the search tree is a **search node** corresponding to the initial state.

We continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the **search strategy**. The general tree-search algorithm is described in figure as follows:

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Figure: An informal description of the general tree-search algorithm

The nodes in the search tree are defined using **five components** in data structure. They are

1. **STATE**: the state in the state space to which the node corresponds;
2. **PARENT-NODE**: the node in the search tree that generated this node;
3. **ACTION**: the action that was applied to the parent to generate the node;
4. **PATH-COST**: the cost, traditionally denoted by g(n), of the path fro the initial state to the node, as indicated by the parent pointers.
5. **DEPTH**: the number of steps along the path from the initial state.

The **difference between** nodes and states, a **node** is bookkeeping data structure used to represent the search tree. A **state** corresponds to a configuration of the world.

To represent the collection of nodes that have been generated but not yet expanded – this collection is called **fringe**. Each element of the fringe is a **leaf node**, that is, a node with no successors in the tree. The representation of the fringe would be a set of nodes. The search strategy then would be a function that selects the next node to be expanded from this set. It could be computationally expensive, because the strategy function might have to look at every element of the set to choose the best one. Alternatively, the collection of nodes is implemented as a **queue** representation. The **queue operations** as follows:

- ✓ **MAKE-QUEUE** (element…) – creates a queue with the given elements.
- ✓ **EMPTY?** (queue) – returns true only if there are no more elements in the queue.
- ✓ **FIRST** (queue) – returns the first element of the queue.
- ✓ **REMOVE-FIRST** (queue) returns **FIRST** (queue) and removes it from the queue.
- ✓ **INSERT** (element, queue) – inserts an element into the queue and the resulting queue.
- ✓ **INSERT-ALL** (elements, queue) – inserts a set of elements into the queue and returns the resulting queue.

The formal version of the general tree-search algorithm shown in figure.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]),fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem] applied to STATE[node] succeeds
            then return SOLUTION(node)
        fringe ← INSERT-ALL(EXPAND(node, problem),fringe)

function EXPAND(node, problem) returns a set of nodes
    successors ← the empty set
    for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        STATE[s] ← result
        PARENT-NODE[s] ← node
        ACTION[s] ← action
        PATH-COST[s] ← PATH-COST [node] + STEP-COST (node, action, s)
        DEPTH[s] ← DEPTH [node] + 1
        add s to successors
    return successors
```

Figure: The General Tree-Search Algorithm

Measuring problem-solving performance

The output of a problem-solving is either failure or a solution. We will evaluate an algorithm's performance in **four** ways:

- ✓ **Completeness**: The strategy guaranteed to find a solution when there is one.
- ✓ **Optimality**: If more than one way exists to derive the solution then the best one is selected.
- ✓ **Time complexity**: Time taken to run a solution.
- ✓ **Space complexity**: Memory needed to perform the search.

In AI, where the graph is represented implicitly by the initial state and successor function and is frequently infinite, **complexity** is expressed in terms of three quantities: **b**, the branching factor or maximum number of successors of any node; **d**, the depth of the shallowest goal node; and **m**, the maximum length of any path in the state space.

Definition of branching factor (b): The number of nodes which is connected to each of the node in the search tree. It is used to find space and time complexity of the search strategy.

1.10 Search Strategies

In general the Search Strategies are classified into **uninformed search** (or) **Blind search** and **informed search** (or) **Heuristic search**.

Uninformed search vs. Informed search

S.No	**Uninformedsearch(Blind search)**	**Informed search(Heuristic search)**
1	No information about the number of steps (or) path cost from the current state to goal state	The path cost from the current state to goal state is calculated, to select the minimum path cost as the next state.

2	Less effective in search method	More effective
3	Problem to be solved with the given information	Additional information can be added as assumption to solve the problem
4	E.g. a)Breadth first search b)Uniform cost search c) Depth first search d) Depth limited search e) Interactive deepening search f) Bi-directional search	E.g. a) Best first search b) Greedy search c) A* search

Breadth first search

Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Simple calling TREE-SEARCH (problem, FIFO-QUEUE ( )) results in a breadth first search. Figure 3.6 shows the progress of the search on a simple binary tree.

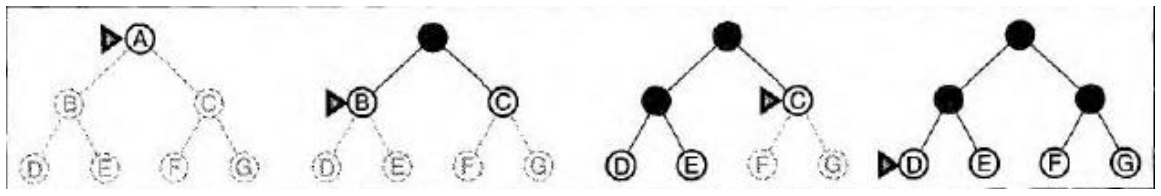


Figure: Breadth first search on a simple binary tree.

Time complexity, the total number of nodes generated is

B + b2 + b3 +…. +bd + bd+1 – b) = O (bd+1)

Every node that is generated must remain in memory. The space complexity is same as the time complexity.

Adv: Guaranteed to find the single solution at the shallowest depth level.

Disadv: **I)** The memory requirements are a bigger problem for breadth-first search than is the execution time.

II) Exponential-complexity search problems cannot be solved by uninformed methods for any but only suitable for smallest instances problem (i.e.) (number of levels to be minimum (or) branching factor to be minimum)

Uniform-cost search

The root node is expanded first, and then the next node to be expanded is selected as the lowest cost node on the fringe rather than the lowest depth node i.e.

g (n) = path cost. Breadth first search is equivalent to Uniform cost search when g(n) = DEPTH(n).

Adv: Guaranteed to find the single solution at minimum path cost.

Disadv: Only suitable for smallest instances problem.

Depth-first search

It always expands the deepest node in the current fringe of the search tree. If dead end occurs, backtracking is done to the next immediate previous node for nodes to be expanded.

Adv: If more than one solution exists (or) number of levels is high then DFS is best because exploration is done only in a small portion of the whole space.

Disadv: Not guaranteed to find a solution.

Depth-limited search

The problem of unbounded trees can be alleviated by supplying depth-first search with predetermined depth limit **L**. i.e., nodes at depth **L** are treated as if they have no successors. This approach is called **depth-limited search**. The maximum level of depth depends on the number of states.

Interactive deepening search

It is a general strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits. It combines the benefits of depth-first and breadth-first search.

Bidirectional search

It is a strategy that simultaneously searches both the directions i.e. forward from the initial state and backward from the goal, and stops when the two searches meet in the middle. It is implemented by having one or both of the searches check each node before it is expanded to see if tit is in the fringe of the order search tree; if so, a solution has been found.

1.11 Avoiding Repeated States

The repeated states can be avoided using **three** different ways. They are:

1. Do not return to the state you just came from i.e. avoid any successor that is the same state as the node's parent.
2. Do not create path with cycles i.e. avoid any successor of a node that is the same as any of the node's ancestors.
3. Do not generate any state that was ever generated before.

1.12 Searching with Partial Information

When the knowledge of the states or actions is incomplete about the environment, then only partial information is known to the agent. This incompleteness leads to three distinct problem types:

1. **Sensor less problems** (**conformant problems**): If the agent has no sensors at all, then it could be one of several possible initial states, and each action might therefore lead to one of several possible successor states.
   E.g. the Vacuum world problem

2. **Contingency Problems**: If the environment is partially observable or ir actions are uncertain, then the agent's percepts provide new information after each action. Each possible percept defines a contingency that must be planned for. A problem is called adversarial if the uncertainty is caused by the actions of another agent.
3. **Exploration problems**: When the states and actions of the environment are unknown, the agent must act to discover them. It can be viewed as an extreme case on contingency problems.

Question Bank

UNIT - I

Part – A

1. Define AI.
2. What is meant by Turing Test?
3. Define Agent and Agent Function.
4. Give the Structure of agent in an environment.
5. How to measure the performance of an agent?
6. Define ideal Rational Agent.
7. Define Mapping. Give Example.
8. Define Ideal Mapping with example.
9. Write Short notes on Autonomy.
10. Define Agent Program.
11. Give two examples for agent type and write the corresponding PAGE description.
12. Define Environment Program.
13. List the environment of environment.
14. Define Problem Solving Agent.
15. List the steps involved in simple problem solving technique.
16. List the four different types of problem with an example.
17. Define the term : Problem
18. What is meant by operators?
19. Define State Space.
20. Define Path.
21. Define Path Cost.
22. Give example for AI problems.
23. Give example for real world problems in AI.
24. Define the following terms for 8 Puzzle Problem: States, Operator, Goal Test, and Path Cost.
25. Write short notes on Search Tree.
26. Define Fringe or Frontier.
27. List the steps to measure the performance of search strategies.
28. Differentiate blind search with heuristic search.
29. Define branching factor.
30. Differentiate Best First Search, Depth First Search with Breadth First Search.
31. What is the use of Heuristic Functions?
32. Why problem formulation must follow the goal formulation?
33. Give the PEAS description of an “Interactive English Tutor” System.
34. Write an informal description for the general tree search algorithm.

PART – B

1. Explain briefly about different types of Agent Program with an example.
1. Explain briefly about the Environment Programs with suitable example.
2. Define and solve the given crypt arithmetic problem using the following descriptions: a) State b) Operator c) Initial State d) Goal Test e) Solution Process
3. Define and Solve the following Problems : a) Water Jug Problem
   b) Missionaries and Cannibals Problem.

5. Define the Vacuum World Problem and draw the corresponding state set space representations
6. Explain briefly about blind search techniques with an example for each search.
7. Explain the following uninformed search strategies with examples.
   a) Breadth First Search b) Uniform Cost Search c) depth first Search d) Depth Limited Search e) Bidirectional Search

UNIT – II

SEARCHING TECHNIQUES

Informed search and exploration − Informed search strategies − Heuristic function − Local search algorithms and optimistic problems − Local search in continuous spaces − Online search agents and unknown environments − Constraint Satisfaction Problems (CSP) − Backtracking search and local search for CSP − Structure of problems − Adversarial search − Games − Optimal decisions in games − Alpha-Beta pruning − Imperfect real-time decision − Games that include an element of chance.

2.1 Informed (Heuristic) search strategies

2.2 Informed search (Heuristic search):

The path cost from the current state to goal state is calculated, to select the minimum path cost as the next state.

Additional information can be added as assumption to solve the problem.

E.g. a) Best first search

b) Greedy search

c) A* search

Best first search

Best first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, f(n).

A Key component of these algorithms is a heuristic function, h (n):

h (n) = estimated cost of the cheapest path from node n to a goal node.

The two types of evaluation functions are:

Expand the node closest to the goal state using estimated cost as the evaluation is called **Greedy best-first search**.

Expand the node on the least cost solution path using estimated cost and actual cost as the evaluation function is called **A* search**.

Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the **heuristic function:** f (n) = h (n).

It resembles depth-first search in a way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end. It is **not optimal**, and it is **incomplete**. The worst case **time** and **space complexity** is O (bm), where m is the maximum depth of the search space. It can be reduced with good heuristic function.

A* search: Minimizing the total estimated solution cost

Expand the node on the least cost solution path using estimated cost and actual cost as the evaluation function is called **A* search.** It evaluates nodes by combining g (n) , the cost to reach the node, and h (n), the cost to get from the node to the goal:

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

since g (n) gives the path cost from the start node to node n, and h (n) is the estimated cost of the cheapest path from n to the goal, we have

f (n) = estimated cost of the cheapest solution through n.

A* search is both complete and optimal.

Monotonicity (consistency): In search tree any path from the root, the f-cost never decreases. This condition is true for almost all admissible heuristics. A heuristic which satisfies this property is called monotonicity.

Optimality: It is derived with two approaches. They are a) A* used with Tree-search b) A* used with Graph-search.

Memory bounded heuristic search

The memory requirements of A* is reduced by combining the heuristic function with iterative deepening resulting an IDA* algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost(g+h) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. The main disadvantage is, it will require more storage space in complex domains.

The two recent memory bounded algorithms are:

1. Recursive best-first search(RBFS)
2. Memory bounded A* search (MA*)

Recursive best-first search (RBFS)

RBFS is simple recursive algorithm uses only linear space. The algorithm is as follows:

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS (problem, MAKE-NODE (INITIAL-STATE [problem]), ∞)
function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](state) then return node
    successors ← EXPAND (node, problem)
    if successors is empty then return failure, ∞
```

```
for each s in successors do
        f[s] ← max(g(s) +  h(s).f[node])
repeat
        best ← the lowest f-value node in successors
        if f[best] > f_limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

Figure 2.1 The algorithm for recursive best-first search

Its **time complexity** depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. Its **space complexity** is O (bd), even though more memory is available.

Search techniques which use all available memory are:

1) MA* (Memory-bounded A*)

2) SMA* (Simplified MA*)

SMA* (Simplified MA*)

It can make use of all available memory to carry out the search.

Properties: i) It will utilize whatever memory is made available to it.

ii) It avoids repeated states as far as its memory allows.

It is **complete** if the available memory is sufficient to store the deepest solution path.

It is **optimal** if enough memory is available to store the deepest solution path. Otherwise, it returns the best solution that can be reached with the available memory.

Advantage: SMA* uses only the available memory.

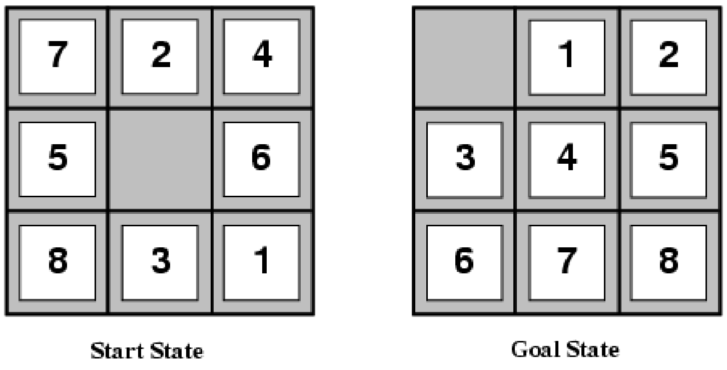
Disadvantage: If enough memory is not available it leads to suboptimal solution.

Space and Time complexity: depends on the available number of nodes.

2.3 Heuristic Functions

The 8-puzzle problem

Given:



Task: Find the shortest solution using heuristic function that never over estimates the number of steps to the goal.

Solution: To perform the given task two candidates are required, which are named as h1 and h2

h1 = the number of misplaced tiles. From figure, all of the eight tiles are out of position, so the start state would have h1 = 8. h1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

h2 = the sum of the distances of the tiles from their goal positions. The sum of the distances of the tiles from their goal positions is called as city block distance or Manhattan distance of:

h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

The effect of heuristic accuracy on performance

Effective branching factor (b*)

In the search tree, if the total number of nodes expanded by A* for a particular problem is N, and the solution depth is d, then b* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus

$$N + 1 = 1 + b^* + (b^*)\ 2 + \dots + (b^*)\ d.$$

For example: depth = 5; N = 52;

Effective branching factor = 1.91

Relaxed problem

A Problem with fewer restrictions on the actions is called **relaxed problem**. *The cost of an optimal solution to be relaxed problem is an admissible heuristic for the original problem*. If the given problem is a relaxed problem then it is possible to produce good heuristic function.

2.4 Local search algorithms and optimization problems

Local search algorithms operate using a single current state and generally move only to neighbors of that state. Local search algorithms are not systematic, they have **two key advantages**: **1)** they use very little memory – usually a constant amount **2)** they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

To understand local search, we will find it very useful to consider the state space landscape as shown in figure.

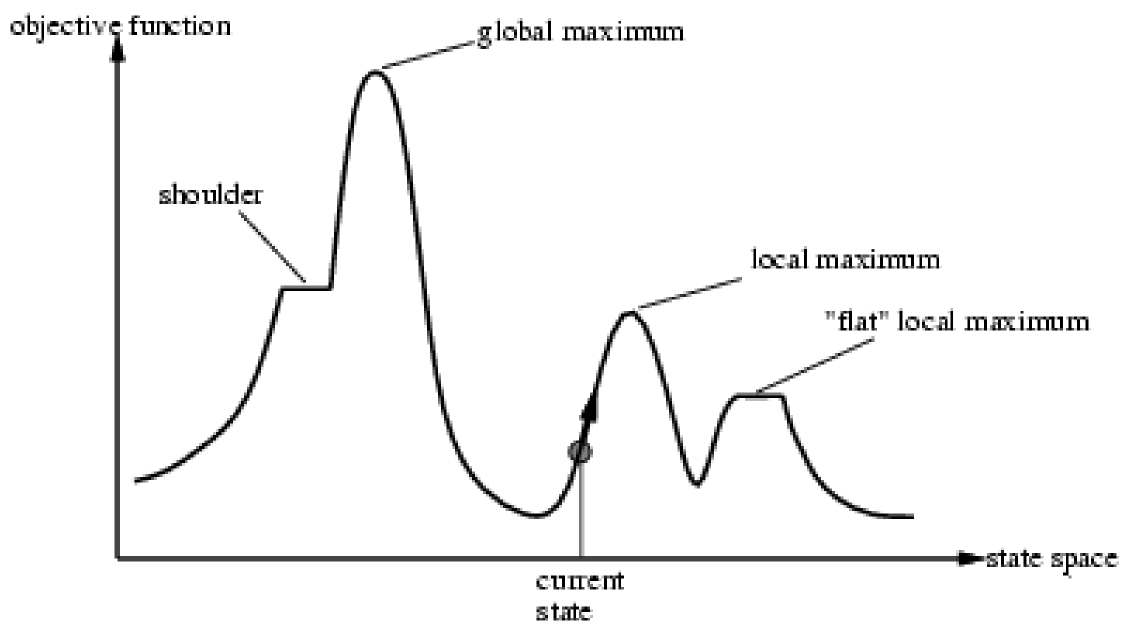


Figure: A one-dimensional state space landscape

Applications:

- Integrated – circuit design
- Factory – floor layout
- Job-shop scheduling
- Automatic programming
- Vehicle routing
- Telecommunications network optimization

Advantages:

- Constant search space. It is suitable for online as well as offline search.
- The search cost is less when comparing to informed search methods.

Some of the local search algorithms are:

i. Hill climbing search
ii. Simulated annealing
iii. Local beam search
iv. Genetic algorithm(GA)

Hill-climbing search

A search technique that move in the direction of increasing value to reach a peak state. It terminates when it reaches a peak where no neighbor has a higher value. The Hill-climbing search algorithm as shown in figure.

```
function HILL-CLIMBING(problem) returns a state that is local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node
    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Figure : The hill-climbing search algorithm

Drawbacks:

- **Local maxima** (Foot hills): a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.

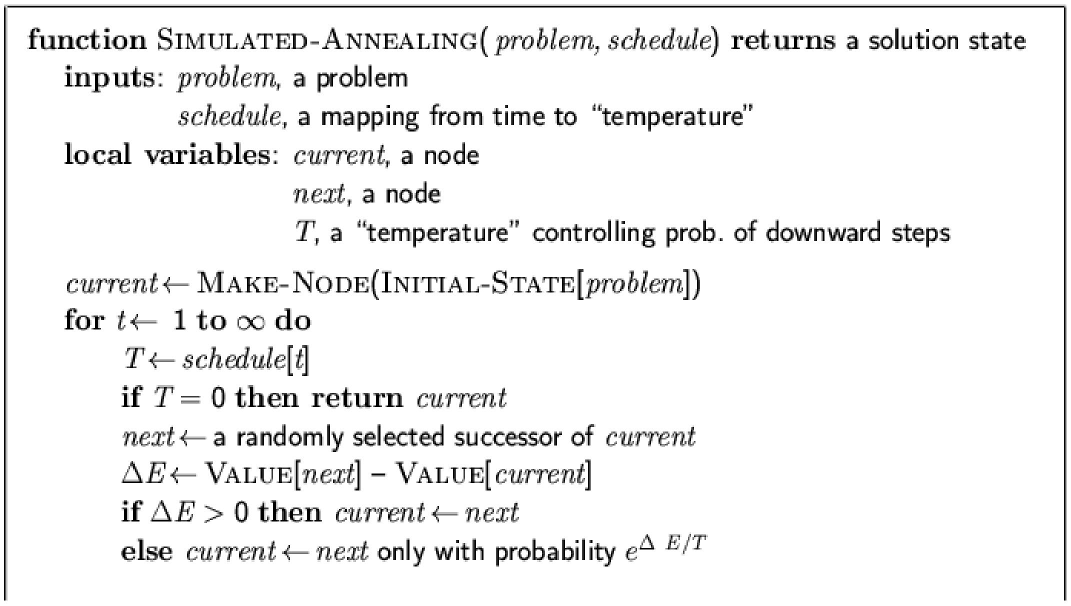
- **Ridges**: a sequence of local maxima, which had a slope that gently moves to a peak.
- **Plateaux** (shoulder): is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum.

Some of the variants of hill-climbing are:

- **Stochastic hill climbing**: chooses at random from among the uphill moves.
- **First choice hill climbing**: implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- **Random-restart hill climbing**: overcomes local maxima-trivially complete.

Simulated annealing Search

An algorithm which combines hill climbing with random walk to yield both efficiency and completeness. This algorithm was developed using annealing process the process of gradually cooling a liquid until it freezes.



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability $e^{\Delta E/T}$
```

Figure 2.4 The simulated annealing search algorithm

Applications:

- VLSI layout
- Airline scheduling

Local beam search

The Local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats. *In a local beam search, useful information is passed among the k parallel search threads*.

This search will suffer from lack of diversity among k states. Therefore a variant named as stochastic beam search selects k successors at random, with the probability of choosing a given successor being an increasing function of its value.

Genetic Algorithm (GA)

A GA is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state. Figure describes a GA algorithm.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
    inputs: population, a set of individuals
            FITNESS-FN, a function that measures the fitness of an individual
    repeat
        new_population ← empty set
        loop for i from 1 to SIZE(population) do
            x ← RANDOM-SELECTION(population, FITNESS-FN)
            y ← RANDOM-SELECTION(population, FITNESS-FN)
            child ← REPRODUCE(x,y)
            if ( small random probability) then child ← MUTATE(child)
            add child to new_population
        population ← new_population
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to FITNESS-FN

function REPRODUCE(x,y) returns an individual
    inputs: x,y, parent individuals
    n ← LENGTH(x)
    c ← random number from 1 to n
    return APPEND(SUBSTRING(x,1,c),SUBSTRING(y,c+1,n))
```

Figure A Genetic algorithm

2.5 Local search in continuous spaces

Local search in continuous space is the one that deals with the real world problems.

- One way to solve continuous problems is to discretize the neighborhood of each state.
- Stochastic hill climbing and simulated annealing are applied directly in the continuous space.

- Empirical gradient, line search, Newton-raphson method can be applied in this domain t find the successor state.
- It may also lead to local maxima, ridges and plateau. This situation is avoided by using random restart method.

2.6 Online search agents and unknown environments

Offline search: They compute a complete solution before setting foot in the real world, and then execute the solution without recourse to their percepts.

Online search: Agents operate by interleaving computation and action: first it takes an action, and then it observes the environment and computes the next action.

Online search is a good idea in **dynamic** or **semi dynamic** domains and **stochastic** domains.

Online search is a necessary idea for an **exploration problem**, where the states and actions are unknown to the agent.

Online search problems

An online search problem can be solved only by an agent executing actions, rather than by a purely computational process. We will assume that the agent knows just the following:

- ACTIONS(s), which returns a list of actions allowed in state in s;
- The step-cost function c(s, a, s’) known to the agent when it reaches s’
- GOAL-TEST(s).

Here the agent cannot access the successors of a state, except by trying all possible actions in that state. This drawback of an agent can be avoided by:

- Visited states are known to the agent.
- Actions are deterministic.
- Possible to find Manhattan distance heuristic.

E.g. A simple maze problem

Competitive ratio: This defines the comparison between the total path costs with the computationally shortest complete exploration path cost. This ratio, should be as small as possible for better results.

Online search agent

An online algorithm, expand only a node that it physically occupies. After each action, an online agent receives a percept to know the state it has reached and it can augment its map of the environment, to decide where to go next.

An online depth-first search agent is shown in figure.

```
function ONLINE-DFS_AGENT(s') returns an action
    inputs: s', a percept that identifies the current state
    static: result, a table, indexed by action and state, initially empty
            unexplored, a table that lists, for each visited state, the actions not yet tried
            unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
            s,a, the previous state and action, initially null
    if GOAL-TEST(s') then return stop
    if s' is a new state then unexplored[s'] ← ACTION(s')
    if s is not null then do
        result[a, s] ← s'
        add s to the front of unbacktracked[s']
    if unexplored[s'] is empty then
        if unbacktracked[s'] is empty then return stop
        else a ← an action b such that result[b, s'] = POP (unbacktracked[s'])
    else a ← POP(unexplored[s'])
    s ← s'
    return a
```

Figure An online search agent that uses depth-first exploration

Online local search

Hill climbing search technique can be used to perform online local search because it keeps just one current state in memory. To avoid the drawback of local maxima, random walk is chosen to explore the environment instead of random restart method. The concept of hill climbing with memory stores a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited is implemented as Learning Real-Time A* (LRTA*) algorithm.

An LRTA* agent algorithm is shown in figure.

```
function LRTA*-AGENT(s') returns an action
    inputs: s', a percept that identifies the current state
    static: result, a table, indexed by action and state, initially empty
        H, a table of cost estimates indexed by state, initially empty
        s , a, the previous state and action, initially null
    if GOAL-TEST(s') then return stop
    if s' is a new state ( not in H) then H[s'] ← h(s')
    unless s is null
        result[a, s] ← s'
        a ← an action b in ACTIONS(s') that minimizes LRTA*-COSTS(s', b, result[b.s'],H)
        s ← s'
        return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
```

```
if s' is undefined then return h(s)
else return c(s, a, s') + H[s']
```

Figure: LRTA*-Agent algorithm

2.7 Constraint satisfaction problems (CSP)

Constraint satisfaction problems (CSP)

Constraint satisfaction problems **(CSP)** is defined by a set of **variables**, X1, X2, …Xn, and a set of **constraints**, C1, C2,…Cm. Each variable Xi has a nonempty **domain** Di of all possible **values**. A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an **objective function**.

Some examples for CSP's are:

- The n-queens problem
- A crossword problem
- A map coloring problem

Constraint graph: A CSP is usually represented as an undirected graph, called constraint graph where the nodes are the variables and the edges are the binary constraints.

CSP can be viewed as an incremental formulation as a standard search problem as follows:

- **Initial state**: the empty assignment { }, in which all variables are unassigned.
- **Successor function**: assign a value to an unassigned variable, provided that it does not conflict with previously assigned variables.
- **Goal test**: the current assignment is complete.
- **Path cost**: a constant cost for every step.

Discrete variables:

1) Finite domains: For n variables with a finite domain size d, the complexity is O (dn). Complete assignment is possible.**E.g**. Map-coloring problems.

2) Infinite domains: For n variables with infinite domain size such as strings, integers etc. **E.g.** set of strings and set of integers.

Continuous variables: Linear constraints solvable in polynomial time by linear programming. **E.g**. start / end times for Hubble space telescope observations.

Types of Constraints:

1. **Unary constraints**, which restricts the value of a single variable.
2. **Binary constraints**, involve pair of variables.
3. **Higher order constraints**, involve three or more variables.

2.8 Backtracking Search for CSP's

Backtracking search, a form of depth first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING( { }, csp)
function RECURSIVE-BACKTRACKING(assignmen,csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLE[csp],assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment according to CONSTRAINT[csp] then
            add { var=value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove { var=value} from assignment
    return failure
```

Figure A simple backtracking algorithm for CSP

Propagating information through constraints

Forward checking: The key steps of forward checking process are:

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

This method propagates information from assigned to unassigned variables, but does not provide early detection for all failures.

Constraint propagation:

Constraint propagation repeatedly enforces constraints locally to detect inconsistencies. This propagation can be done with different types of consistency techniques. They are:

1. **Node consistency** (one consistency): The node representing a variable V in constraint graph is node consistent if for every value X in the current domain of V, each unary constraint on V is satisfied. The node inconsistency can be eliminated by simply removing those values from the domain D of each variable that do not satisfy unary constraint on V.
2. **Arc consistency** (two consistency): Arc refers to a directed arc in the constraint graph. The different versions of Arc consistency algorithms are exist such as AC-1, upto AC-7, but frequently used are AC-3 or AC-4.
3. **Path consistency** (K-consistency): An algorithm for making a constraint graph strongly three consistent that is usually referred as path consistency, ensures that problem can be solved without backtracking.

Handling special constraints

1. **Alldiff constraint**: All the variables involved must have distinct values.
2. **Resource constraint**: Higher order or atmost constraint, in which consistency is achieved by deleting the maximum value of any domain if it is not consistent with minimum values of the other domains.

Intelligent backtracking

Chronological backtracking: when a branch of the search f ails, back up to the preceding variable and try a different value for it. Here the most recent decision point is revisited.

Conflict directed backjumping: It backtracks directly to the source of the problem.

2.9 Local search for CSP

Local search using the min-conflicts heuristic has been applied to constraint satisfaction problems with great success. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by changing the value of one variable at a time.

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic.

Min-conflicts is surprisingly effective for many CSPs, particularly when given a reasonable initial state. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the runtime of minconflicts is roughly *independent of problem size*.

```
function MIN-CONFLICTS(csp, max steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
```

```
        max steps, the number of steps allowed before giving up
current  an initial complete assignment for csp
for i = 1 to max steps do
        if current is a solution for csp then return current
        var  a randomly chosen, conflicted variable from VARIABLES[csp]
        value the value v for var that minimizes CONFLICTS(var, v, current, csp)
        set var =value in current
return failure
```

Figure: The MIN-CONFLICTS algorithm for solving CSP

2.10 The Structure of problems

The complexity of solving CSP is strongly related to the structure of its constraint graph. If the CSP can be divided into independent sub problems, then each sub problem is solved independently then the solutions are combined. When n variables are divided as n/c sub problems, each will take dc work to solve. Hence the total work is O (dc n/c).

Any tree structured CSP can be solved in time linear in the number of variables.. The algorithm has the following steps:

1. Choose any variable as the root of the tree and order the variables from the root to the leaves in such a way that every node's parent in the tree precedes it in the ordering label the variables X1, …Xn in order, every variable except the root has exactly one parent variable.
2. For j from n down to 2, apply arc consistency to the arc(Xi, Xj), where Xi is the parent of Xj, removing values from DOMAIN[Xi] as necessary.
3. For j from 1 to n, assign any value for Xj consistent with the value assigned for Xi, where Xi is the parent of Xj.

The complete algorithm runs in time O (nd2).

General constraint graphs can be reduced to trees on two ways. They are:

1. Removing nodes – Cutest conditioning
2. Collapsing nodes together – Tree decomposition

2.11 Adversarial Search

2.11 Games

A game can be defined by the initial state, the legal actions in each state, a terminal test and a utility function that applies to terminal states.

In game playing to select the next state, search technique is required. The **pruning technique** allows us to ignore positions of the search tree that make no difference to the final choice, and **heuristic evaluation function** allow us to find the utility of a state without doing a complete search.

2.13 Optimal decisions in games

A game can be formally defined as a kind of search problem wit the following components:

- **Initial state**: This includes the board position and identifies the player to move.
- A **successor function** (operators), which returns a list of (move, state) pairs, each indicting a legal move and the resulting state.
- A **terminal test**, which determines when the game is over.
- A **utility function** (payoff function or objective function), which gives a numeric values for the terminal states.

The Minimax algorithm

The Minimax algorithm computes the minimax decision from the current state. It performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is O (bm).

Minimax is for deterministic games with perfect information. The **minimax** algorithm generates the whole game tree and applies the utility function to each terminal state. Then it propagates the utility value up one level and continues to do so until reaching the start node.

The minimax algorithm is as follows:

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    V ← -∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
```

```
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Figure: An algorithm for calculating minimax decisions

2.14 Alpha-Beta pruning

Pruning: The process of eliminating a branch of the search tree from consideration without examining is called pruning. The two parameters of pruning technique are:

1. **Alpha (α):** Best choice for the value of MAX along the path or lower bound on the value that on maximizing node may be ultimately assigned.
2. **Beta (β)**: Best choice for the value of MIN along the path or upper bound on the value that a minimizing node may be ultimately assigned.

Alpha-Beta Pruning: The alpha and beta values are applied to a minimax tree, it returns the same move as minimax, but prunes away branches that cannot possibly influence the final decision is called **Alpha-Beta pruning** or **Cutoff**.

Consider the two ply game tree from figure.

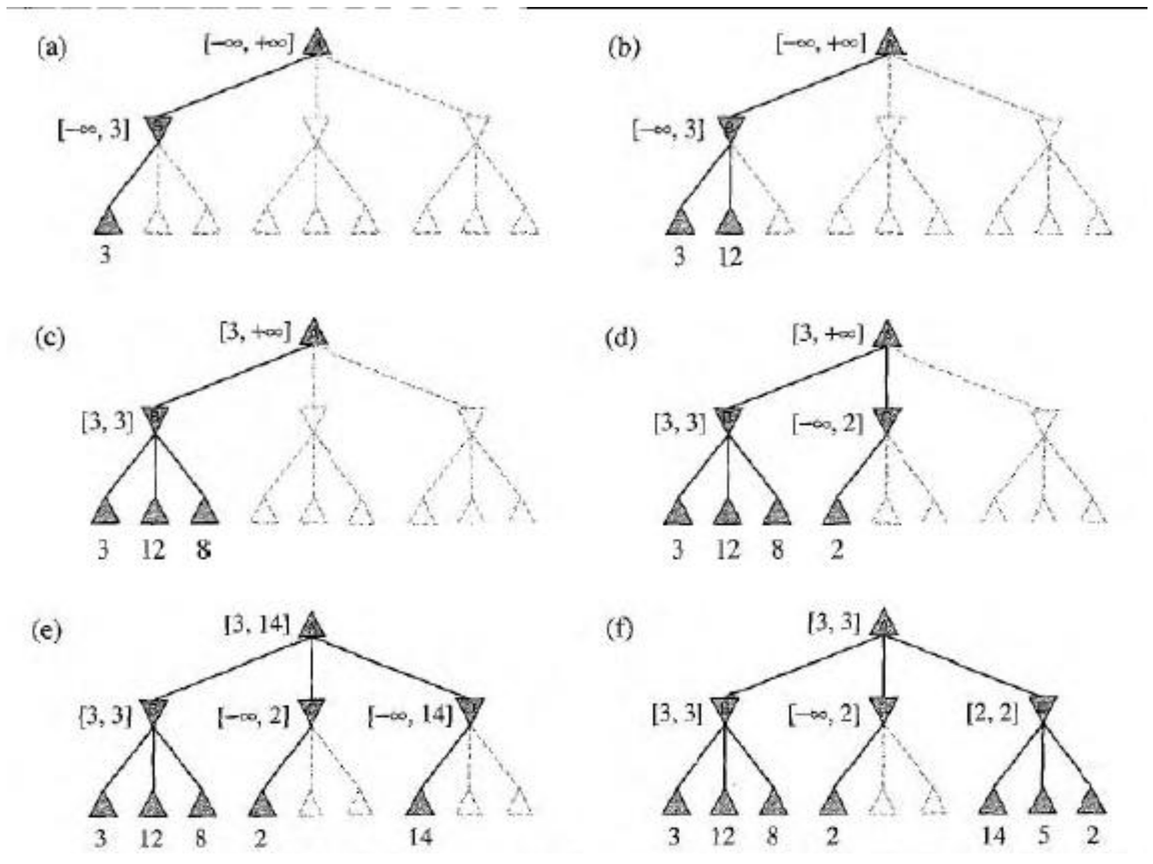


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of *B* is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, *C*, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX'S best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth *5*, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha –beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub trees rather than just leaves.

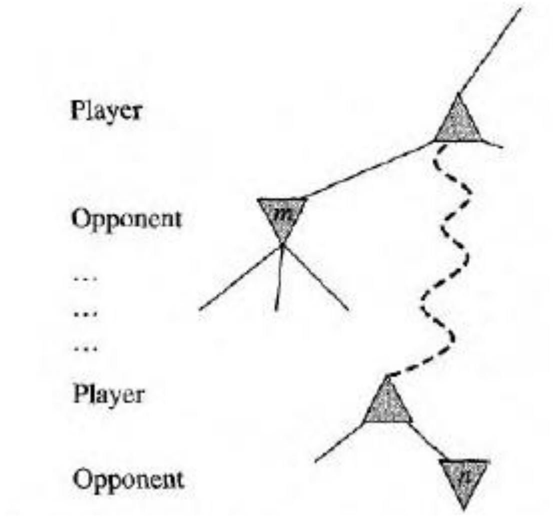


Figure Alpha-beta pruning: the general case. If m is better than n for player, we will never get to n in play.

```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game

   $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
   return the action in SUCCESSORS(state) with value $v$

function MAX-VALUE(state, $a$, $\beta$) returns a utility value
   inputs: state, current state in game
           $a$, the value of the best alternative for MAX along the path to state
           $\beta$, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$
   for $a$, $s$ in SUCCESSORS(state) do
      $v \leftarrow$ MAX($v$, MIN-VALUE($s$, $a$, $\beta$))
      if $v \geq \beta$ then return $v$
      $a \leftarrow$ MAX($\alpha$, $v$)
   return $v$

function MIN-VALUE(state, $a$, $\beta$) returns a utility value
   inputs: state, current state in game
           $a$, the value of the best alternative for MAX along the path to state
           $\beta$, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$
   for $a$, $s$ in SUCCESSORS(state) do
      $v \leftarrow$ MIN($v$, MAX-VALUE(%$a$, $\beta$))
      if $v \leq \alpha$ then return $v$
      $\beta \leftarrow$ MIN($\beta$, $v$)
   return $v$
```

Figure: The alpha-beta search algorithm

2.14 Imperfect, real time decisions

Effectiveness of Alpha – Beta pruning

It needs to examine only $O(b^{m/2})$ nodes to pick the best move.

Futility cut-off

Terminating the exploration of a sub tree that offers little possibility for improvement over other known path is called a futility cut off.

2.16 Games that include an element of chance

Backgammon problem is a typical game that combines luck and skill. Dice are rolled at the beginning of a player turn to determine the set of legal moves that is available to the player.

In figure 6.5, white has rolled a 6-5 , and has four possible moves.

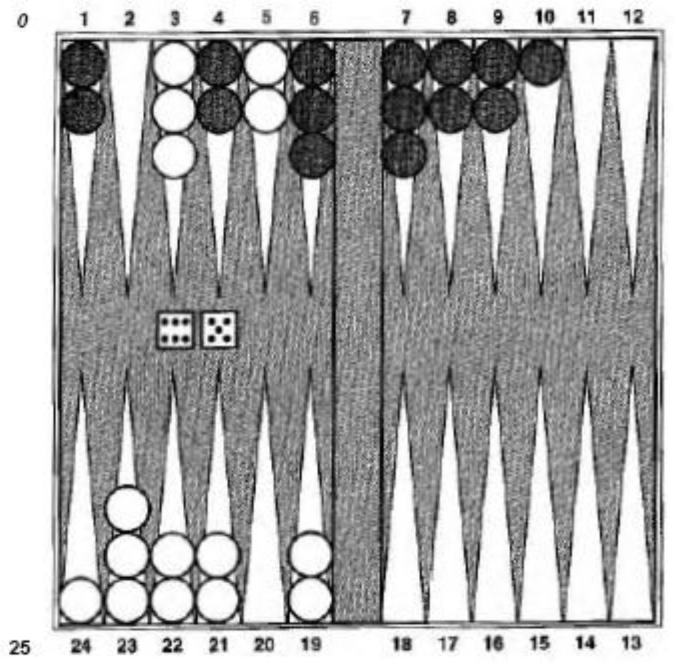


Figure 6.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counterclockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16).

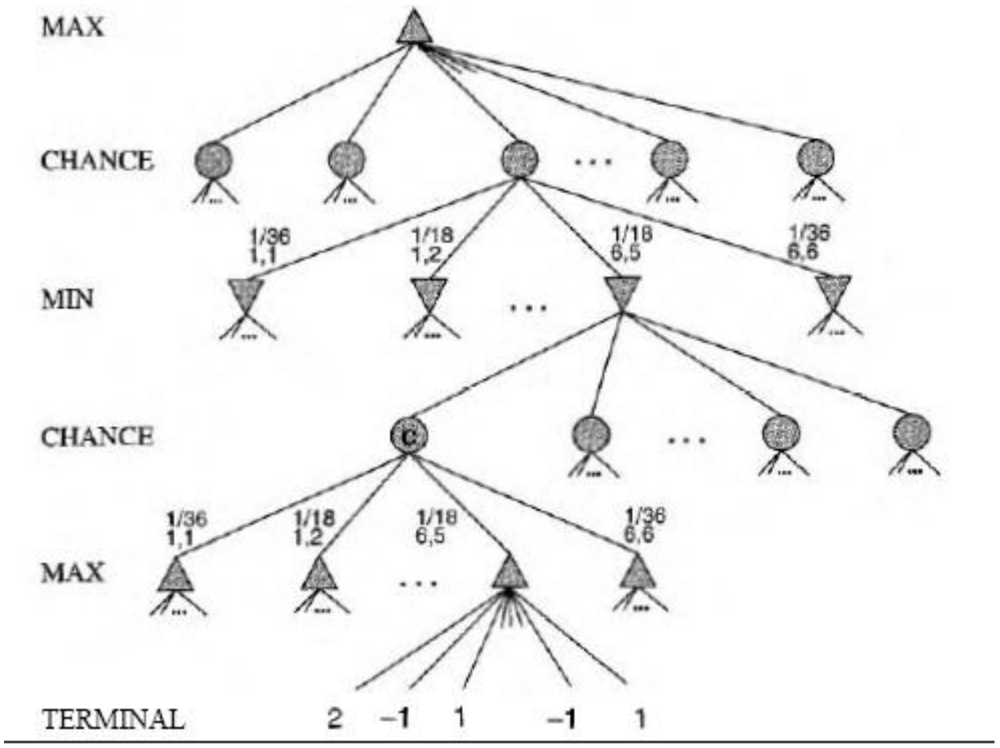


Figure: Schematic game trees for a backgammon position

Question Bank

UNIT II

Part – A

1. What is a local minima problem?
2. How does alpha-beta pruning technique works?
3. What is the use of online search agents in unknown environments?
4. Specify the complexity of expectiminimax.
5. How to improve the effectiveness of a search-based problem-solving technique?
6. What is a constraint satisfaction problem?
7. Differentiate greedy search with A* search.
8. Write short notes on monotonocity and optimality of A* search.
9. Give example for effective branching factor.
10. Define relaxed problem.
11. Give two examples for memory bounded search.
12. Give example for game playing problems.
13. Explain the three advantages of hill climbing.
14. Write the steps of minimax algorithm.
15. Define horizon problem with example.
16. Define pruning.
17. State the reasons to avoid the disadvantage of minimax algorithm.
18. Define alpha, beta cutoff with an example.
19. List the different types and applications of local search algorithm.
20. What is the need of memory bounded heuristic search?
21. Differentiate offline search with online search.
22. How the search technique is made efficient in continuous space.
23. Define CSP.
24. Define constraint graph.
25. List the different types of constraints.
26. Define backtracking search.
27. Define constraint propagation
28. What is the need of arc consistency?
29. Define conflict direct back jumping.
30. Define cycle cutset.
31. How free decomposition is achieved?
32. Write short notes on chance nodes.
33. List the different types of consistency techniques.
34. What are constraint satisfaction problems? How can you formulate them as search problems?

Part – B

1. Explain briefly about heuristic search techniques with an example for each search.
2. Explain the behavior of A* search algorithm with example.
3. Explain briefly about memory bounded search algorithms with an example for each search.
4. Explain minimax algorithm with its procedure with example.
5. Explain alpha beta pruning with its procedure with example.
6. Hoe does hill climbing ensure greedy local search?
7. Explain genetic algorithm with an example.
8. What is the need of online search agent? Explain with an algorithm.
9. Explain the structure of problem using CSP? Is the problem structure influence on the solving technique?
10. Explain back tracking search of CSP with algorithm.
11. Discuss the various issues associated with the backtracking search fro CSPs. How they are addressed?
12. Describe alpha-beta pruning and give the order modifications to the minimax procedure to improve its performance.

UNIT – III

KNOWLEDGE REPRESENTATION

First order logic − Representation revisited − Syntax and semantics for first order logic − Using first order logic − Knowledge engineering in first order logic − Inference in first order logic − Prepositional versus first order logic − Unification and lifting − Forward chaining − Backward chaining − Resolution − Knowledge representation − Ontological engineering − Categories and objects − Actions − Simulation and events − Mental events and mental objects.

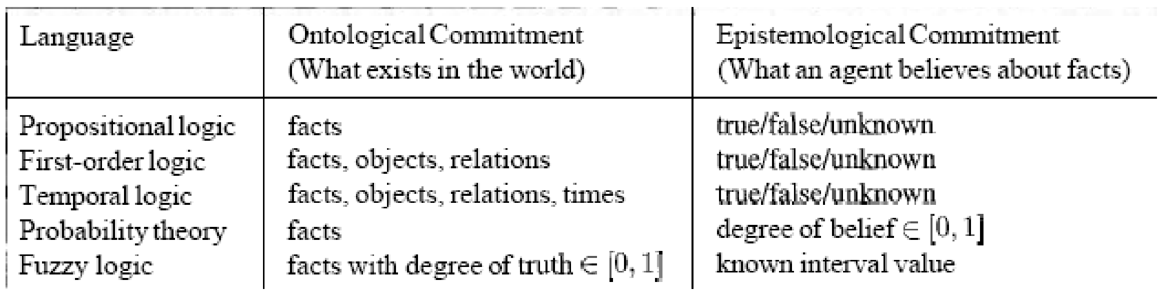
3.1 First order logic (FOL)

3.2 Representation revisited

FOL or **First Order Predicate Calculus**(FOPC), which makes a stronger set of ontological commitments i.e. objects, properties, relations and functions of the world belongings to be represented.

- **Objects** : Things with individual identities. (E.g. home, people, college, colors etc.)
- **Properties** : used to distinguish one object from the other. (E.g. big, small, round etc.)
- **Relations** : relation exists between objects. (E.g. owns, bigger than etc.)
- **Functions** : One kind of relation which has only one value fro a given input. (E.g. father of, best friend etc.)

The primary difference between propositional and FOL logic lies in the ontological commitment made by each language – that is, what it assumes about the nature of reality.



Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure: Formal languages and their ontological and epistemological commitments

3.3 Syntax and semantics for FOL

FOL has sentences, but it also has terms, which represent objects. Constant symbols, variables and functions are used to build terms and quantifiers and predicate symbols are used to build sentences.

Models for FOL

The domain of the model is a set of objects it contains; these objects are sometimes called domain elements. Figure 3.2 shows a model with five objects.

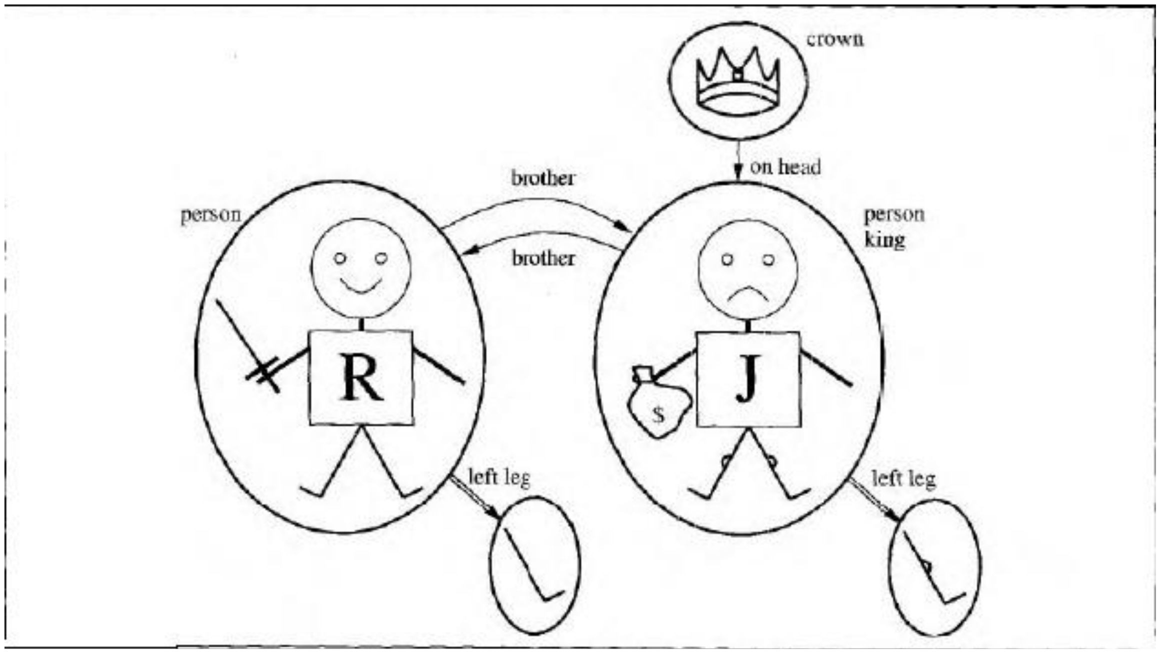


Figure: A model containing five objects, two binary relations, three unary relations and one unary function.

Symbols and interpretations

Figure shows the formal grammar of FOL**.**

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \\
 & | & (\ \mathit{Sentence}\ \mathit{Connective}\ \mathit{Sentence}\ ) \\
 & | & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
 & | & \neg\ \mathit{Sentence} \\
 & & \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate}(\mathit{Term}, \ldots)\ |\ \mathit{Term} = \mathit{Term} \\
 & & \\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
 & | & \mathit{Constant} \\
 & | & \mathit{Variable} \\
 & & \\
\mathit{Connective} & \rightarrow & \Rightarrow\ |\ \wedge\ |\ \vee\ |\ \Leftrightarrow \\
\mathit{Quantifier} & \rightarrow & \forall\ |\ \exists \\
\mathit{Constant} & \rightarrow & A\ |\ X_1\ |\ \mathit{John}\ |\ \ldots \\
\mathit{Variable} & \rightarrow & a\ |\ x\ |\ s\ |\ \ldots \\
\mathit{Predicate} & \rightarrow & \mathit{Before}\ |\ \mathit{HasColor}\ |\ \mathit{Raining}\ |\ \ldots \\
\mathit{Function} & \rightarrow & \mathit{Mother}\ |\ \mathit{LeftLeg}\ |\ \ldots
\end{array}
$$

Figure: The syntax of FOL with equality, specified in BNF.

The basic syntactic elements of the FOL are the symbols that stand for objects, relations and functions. The symbols come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions.

Terms are a logical expression that refers to an object.

An **atomic sentence** is formed from a predicate symbol followed by a parenthesized list of terms.

An **atomic sentence** is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

Quantifiers: Quantifiers are used to express properties of entire collection of objects, rather than representing the objects by name.

FOL contain two quantifiers:

1. Universal quantifiers
2. Existential quantifiers

Equality

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the **equality symbol** to make statements to the effect that two terms refer to the same object.

3.4 Knowledge engineering in FOL

Knowledge engineering: The general process of knowledge base constructional process called knowledge engineering.

The knowledge engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. Identify the task.

2. Assemble the relevant knowledge.

3**.** Decide on a vocabulary of predicates, functions, and (constants.

4. Encode general /knowledge about the domain.

5. Encode a description of the specific problem instance.

6. Pose queries to the inference procedure and get answers.

7. Debug the knowledge base.

The electronic circuits' domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure. We follow the seven-step process for knowledge engineering.

- **Identify the task**
- **Assemble the relevant knowledge**
- **Decide on a vocabulary**
- **Encode general knowledge of the domain**
- **Encode the specific problem instance**
- **Pose queries to the inference procedure**
- **Debug the knowledge base**

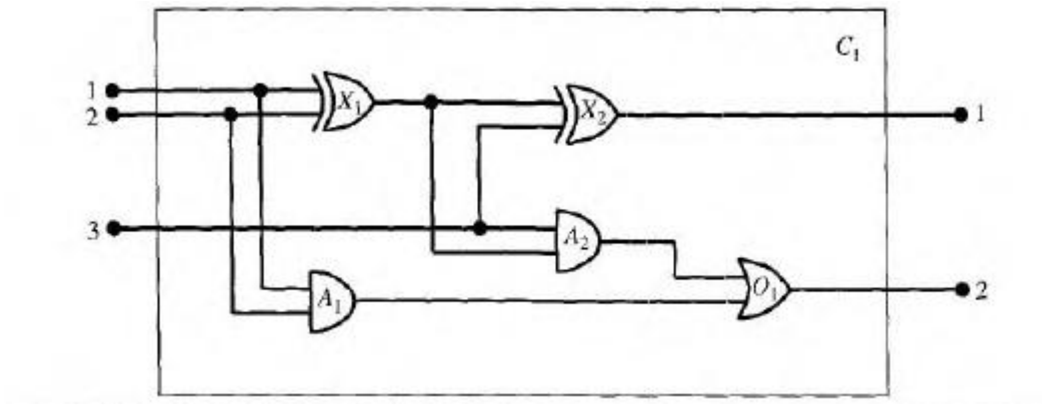


Figure: A digital circuit C1, purporting to be a one-bit. Full adder. The first two inputs 1are the two bits to be added and the third input is a carry bit. The first

output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates and one OR gate.

3.5 Inference in First-Order Logic

Propositional vs. FOL

Inference rules for quantifiers

The rule of **Universal Instantiation** (UI for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.

Let SUBST (*@a,)* denote the result of applying the substitution *8* to the sentence *a* . Then the rule is written

Vv *a*

SUBST ({V/~4),

for any variable v and ground term *g*.

The corresponding **Existential Instantiation** rule: for the existential quantifier is slightly more complicated. For any sentence *a*, variable v, and constant symbol *k* that does not appear elsewhere in the knowledge base.

3.6 Unification and Lifting

A first-order inference rule

This inference process can be captured as a single inference rule that we call **General-**

GENERALIZEPONEDN S **sized Modus Ponens:** For atomic sentences *p,, p,',* and *q,* where there is a substitution *8* such that *S u e s r (Q , p i l)= Suesr(B,p,),* for all *i,*

$$\frac{p_1{}',\ \ p_2{}',\ \ \ldots,\ \ p_n{}',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}$$

Unification

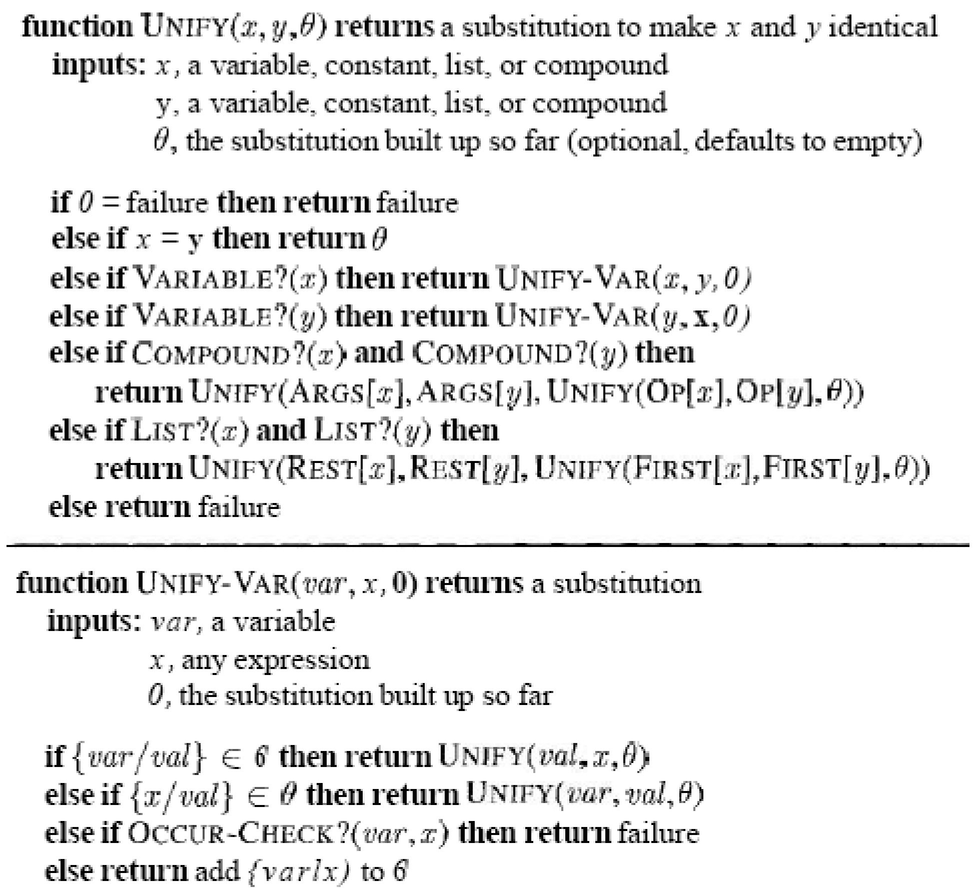
Lifted inference rules require finding substitutions that make different logical expressions

UNIFICATION look identical. This process is called **unification** and is a key component of all first-order UNIFIER inference algorithms.

The ***UNIFY* algorithm** takes two sentences and returns a **unifier** for them if one exists:

UNIFY (^, q) = **0** where *S U B S T (O, =~) S UBST (@q,).*

The problem arises only because the two sentences happen to use the same variable name, **1:** The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its APART variables to avoid name clashes.



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
   inputs: $x$, a variable, constant, list, or compound
           y, a variable, constant, list, or compound
           $\theta$, the substitution built up so far (optional, defaults to empty)

   if $0$ = failure then return failure
   else if $x$ = y then return $\theta$
   else if VARIABLE?($x$) then return UNIFY-VAR($x, y, 0$)
   else if VARIABLE?($y$) then return UNIFY-VAR($y$, x, $0$)
   else if COMPOUND?($x$) and COMPOUND?($y$) then
       return UNIFY(ARGS[$x$], ARGS[$y$], UNIFY(OP[$x$], OP[$y$], $\theta$))
   else if LIST?($x$) and LIST?($y$) then
       return UNIFY(REST[$x$], REST[$y$], UNIFY(FIRST[$x$], FIRST[$y$], $\theta$))
   else return failure
```

```
function UNIFY-VAR($var, x$, 0) returns a substitution
   inputs: $var$, a variable
           $x$, any expression
           $0$, the substitution built up so far

   if $\{var/val\} \in 6$ then return UNIFY($val, x, \theta$)
   else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
   else if OCCUR-CHECK?($var, x$) then return failure
   else return add $\{varlx)$ to $6$
```

Figure: The *UNIFY* algorithm

Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE (Ss) t ores a sentence *s* into the knowledge base and FETCH (^) returns all unifiers such that the query q unifies with some sentence in the knowledge base.

3.7 Forward chaining

A forward-chaining algorithm for propositional definite clauses was already given. The idea is simple: start with the atomic sentences in the knowledge base and apply ModusPonens in the forward direction, adding new atomic sentences, until no further inferences can be made.

First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses they are disjunctions of literals of which *exactly one is positive. A* definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal.

This knowledge base contains no function symbols and is therefore an instance of' the class DATALOG of **Data log** knowledge bases-that is, sets of first-order definite clauses with no function symbols.

A simple forward-chaining algorithm

The first forward chaining algorithm we will consider is a very simple one, as shown in

Figure

```
function FOL-FC-ASK($KB$, $a$) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          $a$, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each sentence $r$ in $KB$ do
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$ ← STANDARDIZE-APART($r$)
      for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                for some $p'_1, \ldots, p'_n$ in $KB$
        $q'$ ← SUBST($\theta, q$)
        if $q'$ is not a renaming of some sentence already in $KB$ or new then do
          add $q'$ to new
          $\phi$ ← UNIFY($q', a$)
          if $\phi$ is not fail then return $\phi$
    add new to KB
  return false
```

Figure: Efficient forward chaining

There are three possible sources of complexity. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal.

Matching rules against Unknown facts

We can express every finite-domain CSP as a single definite clause together with some associated ground facts.

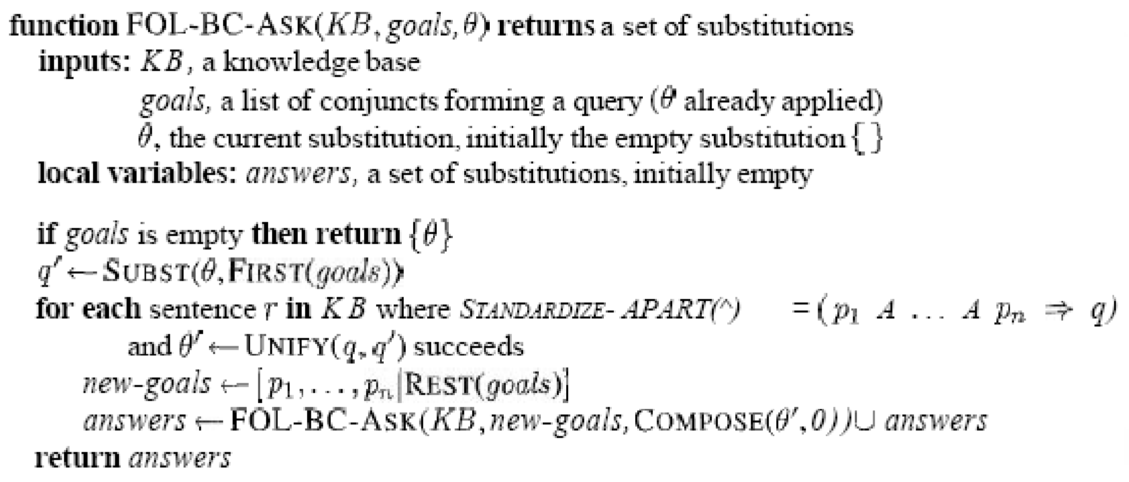
Incremental forward chaining

Redundant rule matching can be avoided if we make the following observation: ***Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t - 1.***

3.8 Backward chaining

A backward chaining algorithm

Figure shows a simple backward-chaining algorithm,



```
function FOL-BC-ASK($KB$, $goals$, $\theta$) returns a set of substitutions
   inputs: $KB$, a knowledge base
           $goals$, a list of conjuncts forming a query ($\theta$ already applied)
           $\theta$, the current substitution, initially the empty substitution $\{\ \}$
   local variables: $answers$, a set of substitutions, initially empty

   if $goals$ is empty then return $\{\theta\}$
   $q' \leftarrow$ SUBST($\theta$, FIRST($goals$))
   for each sentence $r$ in $KB$ where STANDARDIZE- APART(^)  $= (p_1\ A\ \ldots\ A\ p_n \Rightarrow q)$
           and $\theta' \leftarrow$ UNIFY($q$, $q'$) succeeds
      $new\text{-}goals \leftarrow [p_1, \ldots, p_n | \text{REST}(goals)]$
      $answers \leftarrow$ FOL-BC-ASK($KB$, $new\text{-}goals$, COMPOSE($\theta'$, 0)) $\cup$ $answers$
   return $answers$
```

Figure A simple backward-chaining algorithm.

Logic programming

***Algorithm = Logic + Control*.**

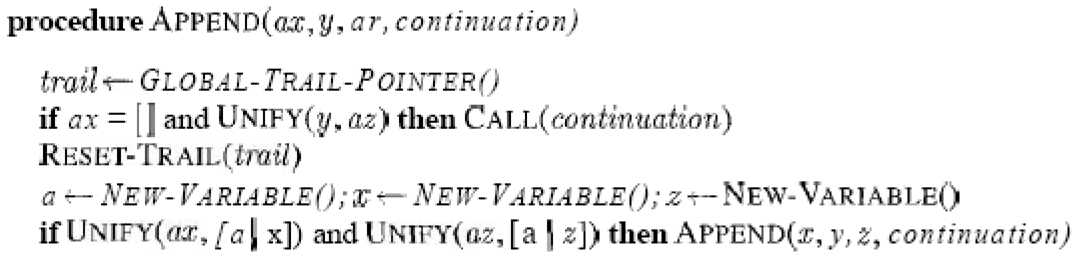
Prolog is by far the most widely used logic programming language. Its users number in the hundreds of thousands. It is used primarily as a rapid-prototyping language and for symbol manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language. Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order. Logic Prolog uses uppercase letters for variables and lowercase for constants. Clauses are written with the head preceding the body; “: -" is used for left implication, commas separate literals in the body, and a period marks the end of a sentence:

The execution of Prolog programs is done via depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some: aspects of Prolog fall outside standard logical inference:

Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled.

Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure



```
procedure APPEND(ax, y, ar, continuation)
    trail ← GLOBAL-TRAIL-POINTER()
    if ax = [ ] and UNIFY(y, az) then CALL(continuation)
    RESET-TRAIL(trail)
    a ← NEW-VARIABLE(); x ← NEW-VARIABLE(); z ← NEW-VARIABLE()
    if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)
```

Figure: The APPEND Algorithm

3.9 Resolution

Conjunctive normal form for first-order logic

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF)-that is, a conjunction of clauses, where each clause is a disjunction of literal. Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

V *x American(x) A Weapon(y) A Sells(z, y , z) A Hostile(z) +- Criminal (x)*

becomes, in CNF,

l A m e r i c a n (x) V -1 *Weapon(y)* V *l S e l l s (x , y , z)* V 1 *Hostile(z)* V *Criminal (x)* .

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.

We will illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or *'dx* ['d *y Animal(y)* J *Loves(x, y)] + [3 y Loves(y, x)]*.

The steps are as follows:

1. Eliminate implications: *ti x [i ' d y* **1** *Animal (y)* V *Loves(x, y j] v* [3 *y Loves (y, xj]*

2. Move *for* inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have *l ' d x p* becomes *3 x 1p* 73 *x p* becomes *'dx 1p* .

Our sentence goes through the following transformations:

'd *x [3 y l(lAnima1 (Y)* V *Loves(x, y))]* V *[3 y Loves(y, x)]* .

'dx [3 *y iiAnirnal(y)* A *lLoves(x, y)]* V [3 *y Loves(y, x)]* .

t i x [3 *y Animal(y)* A *lLoves(x, y)] v [3 y Loves(y, x)]* .

Standardize variables:

Solemnize

Drop universal quantifiers:

Distribute V over A:

Completeness of resolution

Resolution is **refutation-complete,** which means that is a set of sentences is unclassifiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences.

Our goal therefore is to prove the following: if S ***is an unsatisjable set of clauses, then the application of finite number of resolution steps to S will yield a contradiction.***

The basic structure of the proof is shown in Figure.

It proceeds as follows:

1. First, we observe that if S is unclassifiable, then there exists a particular set of ***ground instances*** of the clauses of S such that this set is also unclassifiable (Her brand’s theorem).

2. We then appeal to the **ground resolution theorem**, which states that propositional resolution is complete for ground sentences.

3**.** We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

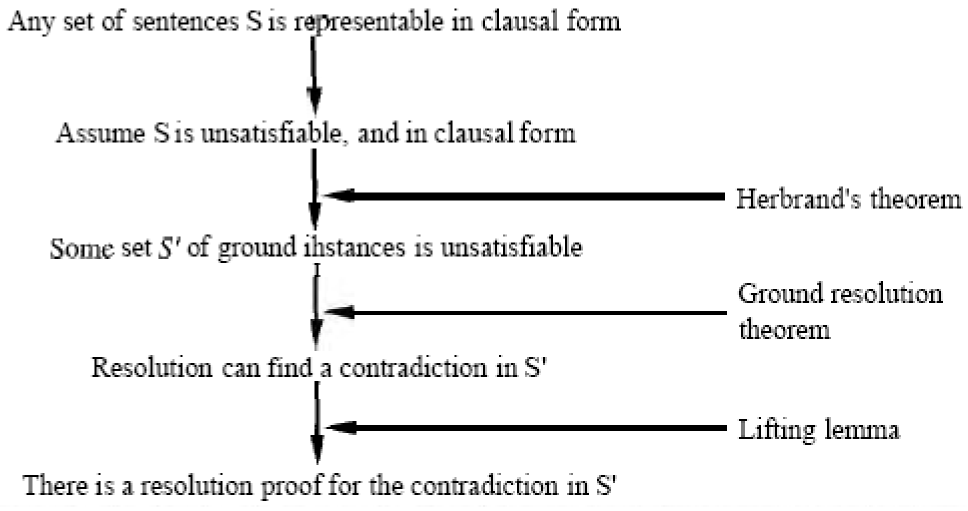

Any set of sentences S is representable in clausal form
Assume S is unsatisfiable, and in clausal form
Herbrand's theorem
Some set S' of ground instances is unsatisfiable
Ground resolution theorem
Resolution can find a contradiction in S'
Lifting lemma
There is a resolution proof for the contradiction in S'

Figure: Structure of a completeness proof for resolution.

```
procedure OTTER(sos, usable)
  inputs: sos, a set of support—clauses defining the problem (a global variable)
          usable, background knowledge potentially relevant to the problem

  repeat
     clause ← the lightest member of sos
     move clause from sos to usable
     PROCESS(INFER(clause, usable), sos)
  until sos = [ ] or a refutation has been found
```

```
function INFER(clause, usable) returns clauses

  resolve clause with each member of usable
  return the resulting clauses after applying FILTER
```

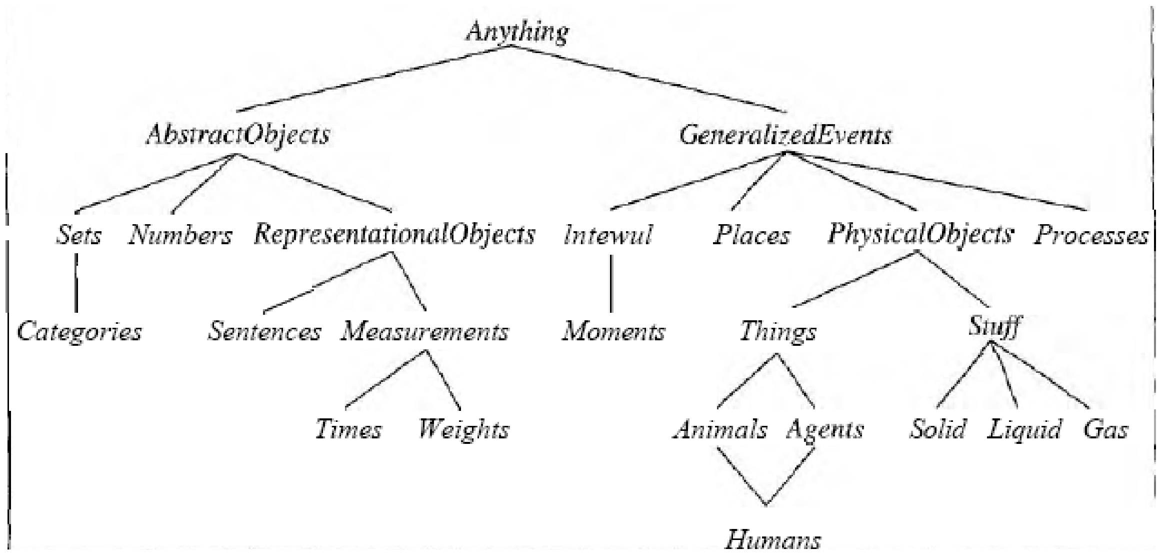
```
procedure PROCESS(clauses, sos)

  for each clause in clauses do
     clause ← SIMPLIFY(clause)
     merge identical literals
     discard clause if it is a tautology
     sos ← [clause | sos]
     if clause has no literals then a refutation has been found
     if clause has one literal then look for unit refutation
```

Figure : Sketch of the OTTER theorem prover. Heuristic control is applied in the selection of the "lightest" clause and in the FILTER function that eliminates uninteresting clauses from consideration.

3.10 Knowledge Representation

3.11 Ontological Engineering

This chapter shows how to create these representations, concentrating on general concepts-such as *Actions, Time, Physical Objects,* and Beliefs-that occur in many different domains. Representing these abstract concepts is sometimes called ontological engineering-it is related to the knowledge engineering process, but operates on a grander scale. The prospect of representing *everything* in the world is daunting.



For example, we will define what it means to be a physical object, and the details of different types of objects-robots, televisions, books, or whatever-can be filled in later.

The general framework of concepts is called an **upper ontology,** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts.

3.12 CATEGORIES AND OBJECTS

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories.*

There are two chances for representing categories in first-order logic: **predicates** and **objects**.

Measurements

Kn both scientific and commonsense theories of the world, objects have height, mass, cost,and so on. The values that we assign for these properties are called **measures.** Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract "measure objects," such as the *length* that is the length of this line segment:

3.13 Actions, Situations, and Events

The ontology of situation calculus

One obvious way to avoid multiple copies of axioms is simply to quantify over time-to say, *" V t ,* such-and-such is the result at *t* + 1 of doing the action at *t."* Instead of dealing with explicit times like *t* 4 1, we will concentrate in this section on ***situations,*** which denote the states resulting from executing actions. This approach is called **situation calculus** and involves the following ontology:

Actions are logical terms such as ***Forward*** and ***Turn*** *(Right).* For now, we will assume that the environment contains only one agent. (If there is more than one, an additional argument can be inserted to say which agent is doing the action.)

Situations are logical terms consisting of the initial situation (usually called *So)* and all situations that are generated by applying an action to a situation. The function *Result(a, s)* (sometimes called *Do)* names the situation that results when action *a* is executed in situation *s.* Figure 3.11 illustrates this idea.

Fluent are functions and predicates that vary from one situation to the next, such as the location of the agent or the aliveness of the wumpus. The dictionary says a fluent

is something that flows, like a liquid. In this use, it means flowing or changing across situations. By convention, the situation is always the last argument of a fluent. For example, *lHoldzng(G1, So)* says that the agent is not holding the gold GI in the initial situation *So. Age (Wumpus, So)* refers to the wumpus's age in *So.*

A temporal or **eternal** predicates and functions are also allowed. Examples include the predicate Gold (GI) and the function *Left Leg Of* (Wumpus*).*

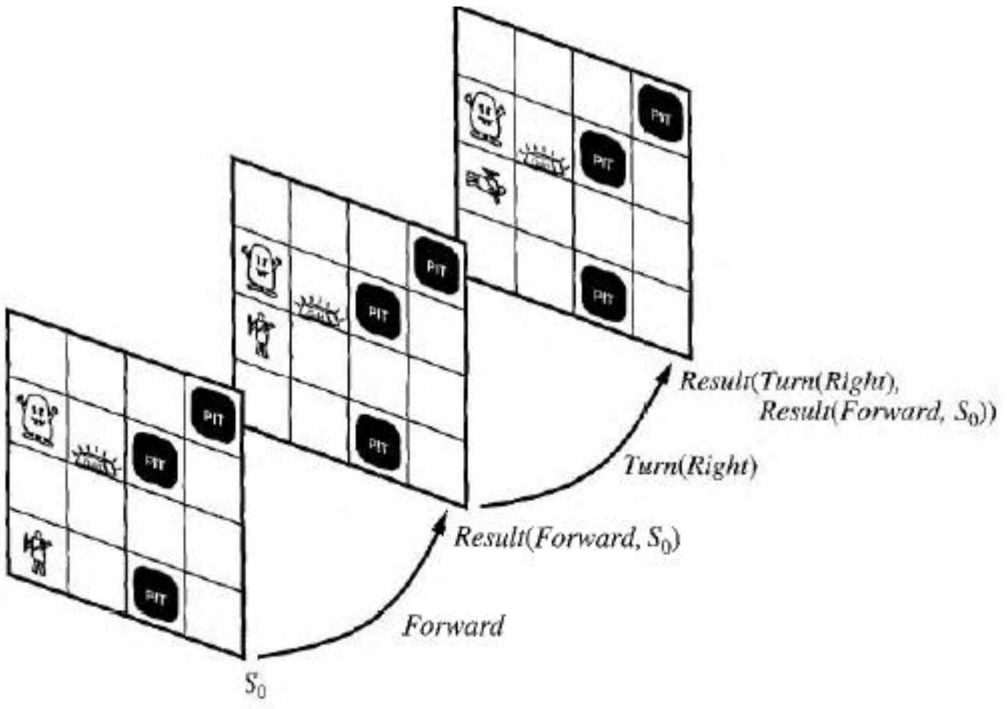


Figure: In situation calculus, each situation (except *So)* is the result of an action.

A situation calculus agent should be able to deduce the outcome of a given sequence of

PROJECTION actions; this is the **projection** task. With a suitable constructive inference algorithm, it should also be able to join a sequence that achieves a desired effect; this is the **planning** task.

Describing actions in situation calculus

In the simplest version of situation calculus, each action is described by two axioms: a **possibility axiom** that says when it is possible to execute the action, and an **effect axiom** that says EFFECT AXIOM what happens when a possible action is executed.

The axioms have the following form:

POSSIBILITY AXIOM: *Preconditions + Poss(a, s) .*

EFFECT AXIOM: *Poss(a, s) + Changes that result from taking action.*

The problem is that the effect axioms say what changes, but don't say what stays the same.

Representing all the things that stay the same is called the **frame problem.** We must find an efficient solution to the frame problem because, in the real world, almost everything stays the same almost all the time. Each action affects only a tiny fraction of all fluent.

One approach is to write explicit **frame axioms** that *do* say what stays the same.

Solving the representational frame problem

The solution to the representational frame problem involves just a slight change in viewpoint on how to write the axioms. Instead of writing out the effects of each action, we consider how each fluent predicate evolves over time.3 The axioms we use are called **successor-state axioms.** They have the following form:

AXIOM SUCCESSOR-STATE AXIOM:

Action is possible + (Fluent is true in result state # Action S *effect made it true*

V *It was true before and action left it alone)* .The *unique names axiom* states a disqualify for every pair of constants in the knowledge base.

Solving the inferential frame problem

To solve the inferential frame problem, we have two possibilities. First, we wo*uld* discard situation calculus and invent a new formalism for writing axioms. This has been done with formalisms such as *1* his *fluent calculus.* Second, we could alter the inference mechanism to handle frame axioms rnose efficiently.

Time and event calculus

The Initiates and Terminates relations play a role similar to the Result relation in situation calculus; Initiates(e, f , t) means that the occurrence of event e at time t causes fluent f to become true, while Terminates (w , f , t) means that f ceases to be true. We use Happens(e, t) to mean that event e happens at time t , and we use Clipped(f , t , t2) to mean that f is terminated by some event sometime between t and t2. Formally, the axiom is:

EVENT CALCULUS AXIOM:

$$
\begin{aligned}
T(f, t_2) \;&\Leftrightarrow\; \exists\, e, t \;\; Happens(e, t) \land Initiates(e, f, t) \land (t < t_2) \\
&\qquad\qquad \land \neg Clipped(f, t, t_2) \\
Clipped(f, t, t_2) \;&\Leftrightarrow\; \exists\, e, t_1 \;\; Happens(e, t_1) \land Terminates(e, f, t_1) \\
&\qquad\qquad \land (t < t_1) \land (t_1 < t_2)\,.
\end{aligned}
$$

Generalized events

A generalized event is composed from aspects of some "space-time chunk"--a piece of this multidimensional space-time universe. This extraction generalizes most of the concepts we have seen so far, including actions, locations, times, fluent, and physical objects.

3.14 Mental events and mental objects

A formal theory of beliefs, we begin with the relationships between agents and "mental objects"-relationships such as Believes, Knows, and Wants. Relations of this kind are called propositional attitudes, be- ATTITUDE because they describe an attitude that an agent can take toward a proposition. Turning a proposition into an object is called reification.

Technically, the property of being able to substitute a term freely for an equal term is called **referential transparency**.

There are two ways to achieve this.

The **first** is to use a different form of logic called **modal logic**, in which propositional attitudes such as Believes and Knows become modal operators that are referentially opaque. This approach is covered in the historical notes section.

The **second** approach, which we will pursue, is to achieve effective opacity within a referentially transparent language using a syntactic theory of mental objects. This means that mental objects are represented by strings.

Knowledge and belief

The relation between believing and knowing has been studied extensively in philosophy. It is commonly said that knowledge is justified true belief. Knowledge, time, and action

Actions can have knowledge preconditions and knowledge effects.

Plans to gather and use information are often represented using a shorthand notation called **runtime variables**,

Plans to gather and use information are often represented using a shorthand notation called **runtime variables**.

Question Bank

UNIT III

PART –A

1. Write short notes on knowledge based agent.

2. Write the two functions of KB agent.

3. Define inference.

4. Explain three levels of knowledge based agent with an example.

5. Define logic.

6. Define entailment.

7. Define truth preserving in logic.

8. Give example for syntax and semantic representation.

9. Define inference procedure.

10. Define logical inference or deduction.

11. Define validity with an example.

12. Define satisfiablity with an example.

13. List the names of five different types of logic.

14. Differentiate propositional logic with FOL.

15. Write the BNF grammer representation for propositional logic.

16. List the names of inference rules of propositional logic.

17. Write the BNF grammer representation for FOL.

18. Define predicate symbol with an example.

19. Define WFF with an example.

20. Differentiate function and relation in FOL with an example.

21. Define ground term.

22. Define horn clause.

23. Write short notes on higher order logic.

24. Write short notes on uniqueness quantifier.

25. Write short notes on uniqueness operator.
26. Define domain. Give example.
27. Define axiom.
28. Define independent axiom.
29. List the names of logical agents for wumpus world problem.
30. List the names of inference rules with quantifiers.
31. State the advantage of generalized modus ponen rule.
32. Write short notes on canonical form.
33. Write short notes on unification.
34. Differentiate forward and backward chaining.
35. Define refutation.
36. Define skolimization with an example.
37. Differentiate CNF and implicative normal form.
38. Write short notes on resolution strategies.
39. Covert the given sentence into propositional form.
40. Convert the given sentence into FOL form.
41. Convert the given sentence into CNF & INF form.
42. Define production system. Give example.
43. Define binding list with an example.
44. Write short notes on subsumption method.
45. How parallelization can be achieved in logic programming?
46. Define ontological engineering with an example.
47. Write short notes on situation calculus.
48. Differentiate suff nouns with count nouns with an example.
49. Define frame problem.
50. List the types of frame problem.
51. List the predicates of time intervals.

52. Define verification.

PART –B

1. Define the problem of wumpus world environment and derive the steps to reach a goal Test with corresponding structure representation.
2. Solve the wumpus world problem using propositional logic.
3. Discuss the logical agents of wumpus world problem.
4. Solve the given KB problem using FOL representation.
5. Explain the inference rules of FOL with an example for each.
6. a. Write the rules to convert a FOL sentence into Normal form
   b. Solve the given KB problem using resolution with refutation technique in CNF & INF form.
7. Solve the given KB problem using resolution with refutation technique in INF & CNF.
8. Explain knowledge engineering in FOL with an example.
9. Explain unification algorithm with an example.
10. Explain Forward chaining algorithm with an example.
11. Explain Backward algorithm with an example.
12. Explain with suitable example how the real world happening are represented in FOL.

UNIT – IV

LEARNING

Learning from observations − Forms of learning − Inductive learning − Learning decision trees − Ensemble learning − Knowledge in learning − Logical formulation of learning − Explanation based learning − Learning using relevant information − Inductive logic programming − Statistical learning methods − Learning with complete data − Learning with hidden variable − EM algorithm − Instance based learning − Neural networks − Reinforcement learning − Passive reinforcement learning − Active reinforcement learning − Generalization in reinforcement learning.

4.1 Learning from observations

Learning takes place as the agent observes its interactions with the world and its own decision-making processes. Learning takes many forms, depending on the nature of the performance element, the component to be improved, and the available feedback.

4.2 Forms of learning

Learning agent is a performance agent that decides what actions to take and a learning element that modifies the performance element so that better decisions can be taken in the future. The design of a learning element is affected by three major issues:

- Which *components* of the performance element are to be learned?
- What *feedback* is available to learn these components?
- What *representation* is used for the components?

The components of these agents include the following:

1. A direct mapping from conditions on the current state to actions.

2. A means to infer relevant properties of the world from the percept sequence.

3. Information about the way the world evolves and about the results of possible actions the agent can take.

4. Utility information indicating the desirability of world states.

5. Action-value information indicating the desirability of actions.

6. Goals that describe classes of states whose achievement maximizes the agent's utility.

The type of feedback available for learning is usually the most important factor in determining the nature of the learning problem that the agent faces. The field of machine learning usually distinguishes three cases: **supervised, unsupervised,** and **reinforcement** learning.

Supervised learning

1. A correct answer for each example or instance is available.
2. Learning is done from known sample input and output.

Unsupervised learning

It is a learning pattern is which correct answers are not given for the input. It is mainly used in probabilistic learning system.

Reinforcement learning

Here learning pattern is rather than being told by a teacher, it learns from reinforcement, i.e. by occasional rewards. The representation of the learned information also plays a very important role in determining how the learning algorithm must work. The last major factor in the design of learning systems is the availability of prior knowl*edge.*

4.3 Inductive learning

An **example** is a pair *(x, f* (z)), where x is the input and *f(x)* is the output of the function applied to *x*. The task of **pure inductive inference** (or **induction)** is this: Given a collection of examples of *f*, return a function *h* that approximates f. The function h is called a **hypothesis.**

For nondeterministic functions, there is an inevitable tradeoff between the complexity of the hypothesis and the degree of jit to the data. There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding simple, consistent hypotheses within that space.

4.4 Learning decision trees

Decision tree induction is one of the simplest, and yet most successful forms of learning algorithm. It serves as a good introduction to the area of inductive learning, and is easy to implement.

Decision trees as performance elements

A decision tree takes as input an object or situation described by a set of attributes and returns a decision the predicted output value for the input. The input attributes can be discrete or continuous. For now, we assume discrete inputs. The output value can also be discrete or continuous; learning a discrete-valued function is called **classification** learning; learning a continuous function is called **regression**.

A decision tree reaches its decision by performing a sequence of tests. Each internal

node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

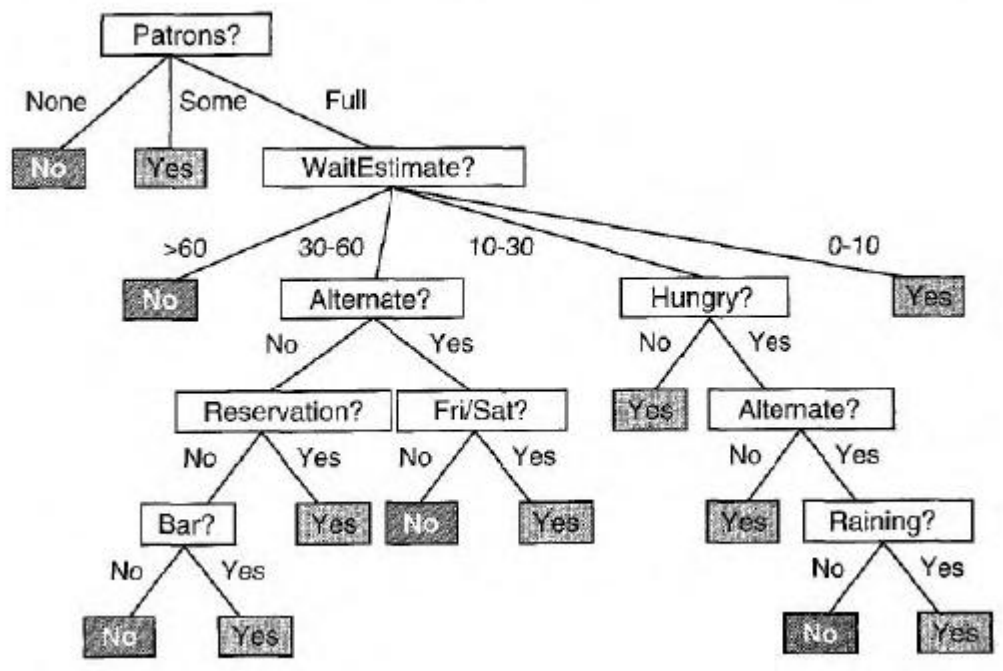


Figure: A decision tree for deciding whether to wait for a table.

Expressiveness of decision trees

The relation ship between the conclusion and the logical combination of attribute values the decision tree expressed as:

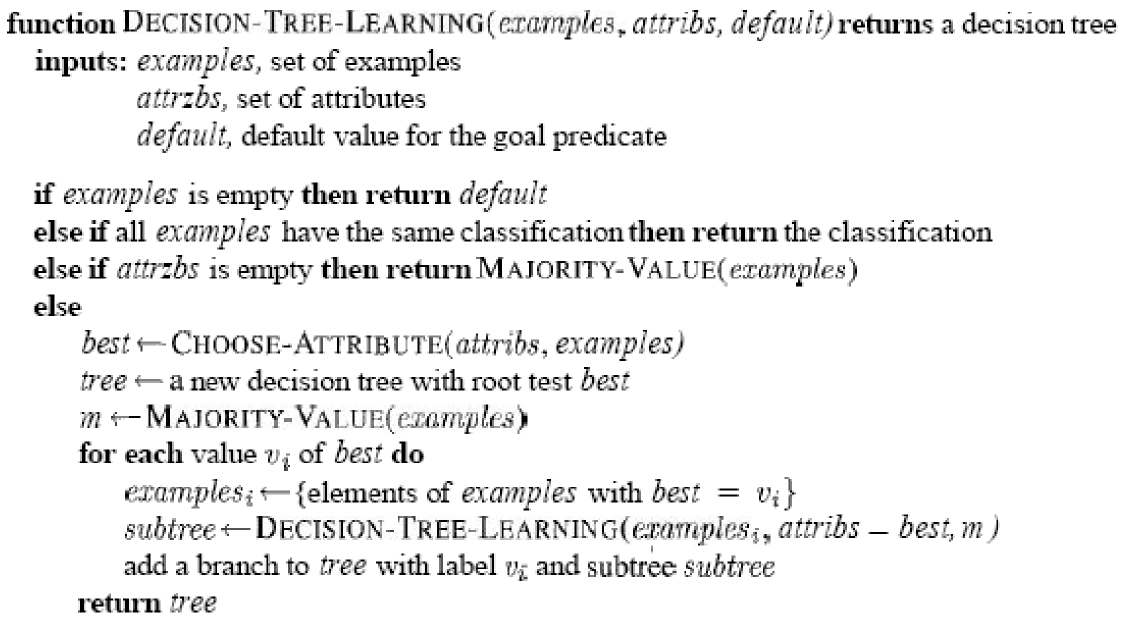
- **Propositional logic –** one variable and other predicates are unary**.**
- **Parity function -** which returns 1 if and only if an even number of inputs are 1.
- **Majority function -** which returns 1 if more than half of its inputs are 1.

The truth table has 2n rows, because each input case is described by n attributes. If it takes 2n bits to define the function, then there are 22 n different functions on n Attributes.

Inducing decision trees from examples

An example for a Boolean decision tree consists of a vector of' input attributes, X, and a single Boolean output value *y*.

The **DECISION-TREE-LEARNING** algorithm is shown in Figure



```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
   inputs: examples, set of examples
           attrzbs, set of attributes
           default, default value for the goal predicate

   if examples is empty then return default
   else if all examples have the same classification then return the classification
   else if attrzbs is empty then return MAJORITY-VALUE(examples)
   else
       best ← CHOOSE-ATTRIBUTE(attribs, examples)
       tree ← a new decision tree with root test best
       m ← MAJORITY-VALUE(examples)
       for each value $v_i$ of best do
           $examples_i$ ← {elements of examples with best $= v_i$}
           subtree ← DECISION-TREE-LEARNING($examples_i$, attribs − best, m)
           add a branch to tree with label $v_i$ and subtree subtree
       return tree
```

Figure: DECISION-TREE-LEARNING algorithms

The performance of learning algorithm depends on

- choosing attribute tests
- data set for training and testing without noise and over lifting

Choosing attribute tests

The scheme used in decision tree learning for selecting attributes is designed to minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets that are all positive or all negative.

CHOOSE-ATTRIBUTE function

The measure should have its maximum value when the attribute is perfect and its minimum value when the attribute is of no use at all.

In general, if the possible answers vi have probabilities P (v i), then the information content I of the actual answer is given by

$$I(P(v_1), \ldots, P(v_n)) = \sum_{i=1} -P(v_i) \log_2 P(v_i)$$

Assessing the performance of the learning algorithm

A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples. It is more convenient to adopt the following methodology:

1. Collect a large set of examples.
2. Divide it into two disjoint sets: the **training set** and the **test set.**
3. Apply the learning algorithm to the training set, generating a hypothesis h.
4. Measure the percentage of examples in the test set that is correctly classified by h.
5. Repeat steps 2 to 4 for different sizes of training sets and different randomly selected training sets of each size.

The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set. This function can be plotted on a graph, giving what is called the **learning curve** for the algorithm on the particular domain.

Noise and over fitting

Noise: Two or more examples with the same description in attributes but different classifications.

Over lifting: When there is a large set of possible hypothesis tree result with meaningless regularity in the data. To overcome the problem of over lifting decision tree pruning or cross-validation can be applied.

Broadening the applicability of decision trees

The decision tree induction in applied to variety of problems, to address the following issues:

1. Missing data.
2. Multivalued attributes.

3. Continuous and integer-valued input attributes.

4. Continuous valued output attributes.

4.5 Ensemble learning

The idea of **ensemble learning** methods is to select a whole collection, or **ensemble,** of hypotheses from the hypothesis space and combine their predictions.

The most popular ensemble learning methods are

1. Boosting
2. Bagging

Boosting: The most widely used ensemble method is called **boosting.** To understand how it works, WSEEIGTH TEDTRA'NING we need first to explain the idea of a **weighted training set.** In such a training set, each example has an associated weight wJ > 0. The higher the weight of an example, the higher is the importance attached to it during the learning of a hypothesis. It is straightforward to modify the learning algorithms we have seen so far to operate with weighted training sets; Boosting starts with w:, = 1 for all the examples (i.e., a normal training set). From this set, it generates the first hypothesis, *hl.* This hypothesis wall classify some of the training examples correctly and some incorrectly. We would like the next hypothesis to do better on the misclassified examples, so we increase their weights while decreasing the weights of the correctly classified examples. From this new weighted training set, we generate hypothesis *h2.* The process continues in this way until we have generated M hypotheses, where M is an input to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination of all the M hypotheses, each weighted according to how well it performed on the training set.

```
function ADABOOST(examples, L, M) returns a weighted-majority hypothesis
   inputs: examples, set of $N$ labelled examples (XI, $y_1$), . . . , ($x_N$, $y_N$)
           L, a learning algorithm
           M, the number of hypotheses in the ensemble
   local variables: w, a vector of $N$ example weights, initially $1/N$
                    h, a vector of $M$ hypotheses
                    z, a vector of M hypothesis weights

   for m = 1 to M do
      h[$m$] ← $L$(examples, w)
      error ← 0
      for $j$ = 1 to $N$ do
         if h[$m$]($x_j$) $\neq$ $y_j$ then error ← error + w[$j$]
      for $j$ = 1 to $N$ do
         if h[$m$]($x_j$) = $y_j$ then w[$j$] ← w[$j$] . error/(1 − error)
      w ← NORMALIZE(w)
      z[$m$] ← log (1 − error)/ error
   return WEIGHTED-MAJORITY(~z)
```

Figure: The ADABOOSTv ariant of the boosting method for ensemble learning

4.6 Knowledge in learning

This simple knowledge-free picture of inductive learning persisted until the early 1980s.

The modem approach is to design agents that already know something and are trying to learn some more this may not sound like a terrifically deep insight, but it makes quite a difference to the way we design agents. It might also have some relevance to our theories about how science itself works. The general idea is shown schematically in Figure .

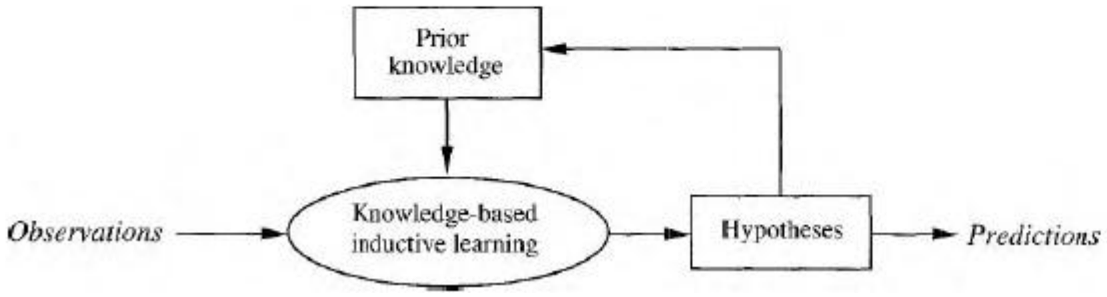


Figure: A cumulative learning process uses, and adds to, its stock of background knowledge over time.

4.7 Logical formulation of learning

Pure inductive learning as a process of finding a hypothesis that agrees with the observed examples. Here, we specialize this definition to the case where the hypothesis is represented by a set of logical sentences.

Examples and hypotheses

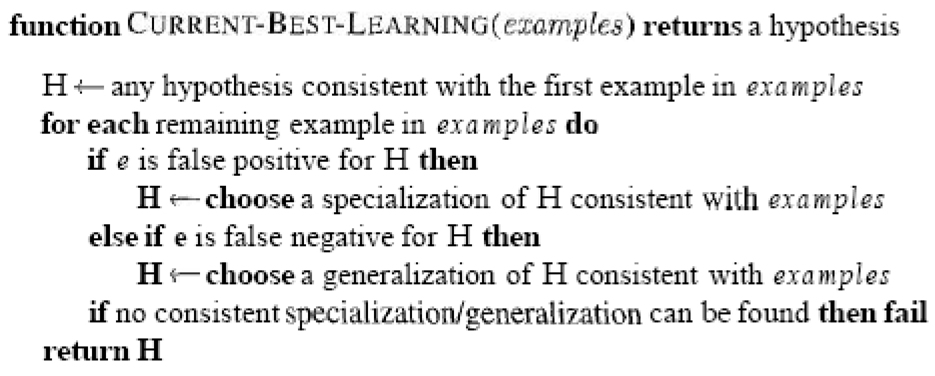
The restaurant learning problem: learning a rule for deciding whether to wait for a table. Examples were described by **attributes** such as Alternate, Bar, FrilSat, and so on.

For example, a decision tree asserts that the goal predicate is true of an object if only if one of the branches leading to *true* is satisfied. Each hypothesis predicts that certain set of examples--namely, those that satisfy its candidate definition-will be examples of the goal predicate. This set is called the **extension** of the predicate. Two hypotheses with different extensions are therefore logically inconsistent with each other, because they disagree on their predictions for at least one example. Logically, this is exactly analogous to the resolution rule of inference. We therefore can characterize inductive learning in a logical setting as a process of gradually eliminating hypotheses that are inconsistent with the examples, narrowing down the possibilities.

Current-best-hypothesis search

The idea behind **current-best-hypothesis** search is to maintain a single hypothesis, and to adjust it as new examples arrive in order to maintain consistency. The hypothesis says it should be negative but it is actually positive. The extension of the hypothesis must be increased to include it. This is called **generalization;** the extension of the hypothesis must be decreased to exclude the example. This is called **specialization.**

Now specify the **CURRENT-B EST-LEARNING** algorithm



```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis
  H ← any hypothesis consistent with the first example in examples
  for each remaining example in examples do
      if e is false positive for H then
          H ← choose a specialization of H consistent with examples
      else if e is false negative for H then
          H ← choose a generalization of H consistent with examples
      if no consistent specialization/generalization can be found then fail
  return H
```

Figure: The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis and backtracks when no consistent specialization/generalization can be found.

We have defined generalization and specialization as operations that change the **extension** of a hypothesis. Now we need to determine exactly how they can be implemented as syntactic operations that change the candidate definition associated with the hypothesis, so that a program can carry them out. This is done by first noting that generalization and. Specialization is also logical relationships between hypotheses. If hypothesis HI, with definition C1, is a generalization of hypothesis H2 with definition

C2, then we must have***'t*** x C2(x) =+ Cl (x). Therefore in order to construct a generalization of *Hz,* we simply need to find a definition Cl that is logically implied by C2. This is easily done. For example, if C2(x) is *Alternate(x) A Patrons(x, Some),* then one possible generalization is given by *Cl (x) Patrons(x, Some).* This is called **dropping conditions.**

In such cases, the program must backtrack to a previous choice point. The **CURRENT-BEST-LEARNING** algorithm and its variants have been used in many machine learning systems, starting with Patrick Winston's (1970) "arch-learning" program.

With a large number of instances and a large space, however, some difficulties arise:

1. Checking all the previous instances over again for each modification is very expensive.

2. The search process may involve a great deal of backtracking.

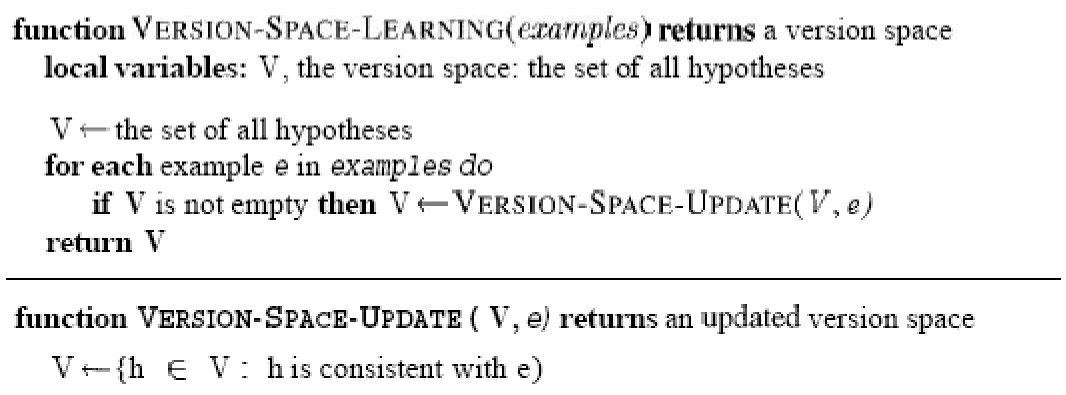
Hypothesis space can be a doubly exponentially large place.

Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to **choose** a particular hypothesis as its best guess even though it does not have enough data yet to be were of the choice.

H l V H 2 V H 3 . . . V H n.

The set of hypotheses remaining is called the **version space,** and the learning algorithm is called the version space learning algorithm (also the **candidate elimination** algorithm).



```
function VERSION-SPACE-LEARNING(examples) returns a version space
   local variables: V, the version space: the set of all hypotheses

   V ← the set of all hypotheses
   for each example e in examples do
      if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
   return V
```

```
function VERSION-SPACE-UPDATE ( V, e) returns an updated version space
   V ← {h ∈ V : h is consistent with e)
```

Figure: The version space learning algorithm.

4.8 Explanation based learning (EBS)

Explanation-based learning is a method for extracting general rules from individual observations. The technique of **memoization** has long been used in computer science to speed up programs by saving the results of computation. The basic idea of memo functions is to accumulate a database of input/output pairs; when the function is called, it first checks the database to see whether it can avoid solving the problem from scratch. Explanation-based learning takes this a good deal further, by creating general rules that cover an entire class of cases.

Extracting general rules from examples

The constraints will involve the Background knowledge, in addition to the Hypothesis and the observed Descriptions and Classifications. In the case of lizard toasting, the cavemen generalize by explaining the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called **explanation-based learning,** or **EBL.**

"the agent does not actually learn anything factually new from the instance".

We can generalize this example to come up with the entailment constraint:

Background A Hypothesis A *Descriptions* I= *c7lassijications.*

In ILP systems, prior knowledge plays two key roles in reducing the complexity of learning:

1. Because any hypothesis generated must be consistent with the prior knowledge as well as with the new observations, the effective hypothesis space size is reduced .to include only those theories that are consistent with what is already known.

2. For any given set of observations, the size of the hypothesis required to construct an explanation for the observations can be much reduced, because the prior knowledge will be available to help out the new rules in explaining the observations. The smaller the hypothesis, the easier it is to find.

The basic **EBL** process works as follows:

1. Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge.

2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.

3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof).

4. Drop any conditions that are true regardless of the values of the variables in the goal.

Improving efficiency

There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do riot yield a solution. In other words, it increases the **branching factor** in the search space.

2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.

3. Derived rules should be as general as possible, SCI that they apply to the largest possible set of cases.

By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect.

4.9 Learning using relevant information

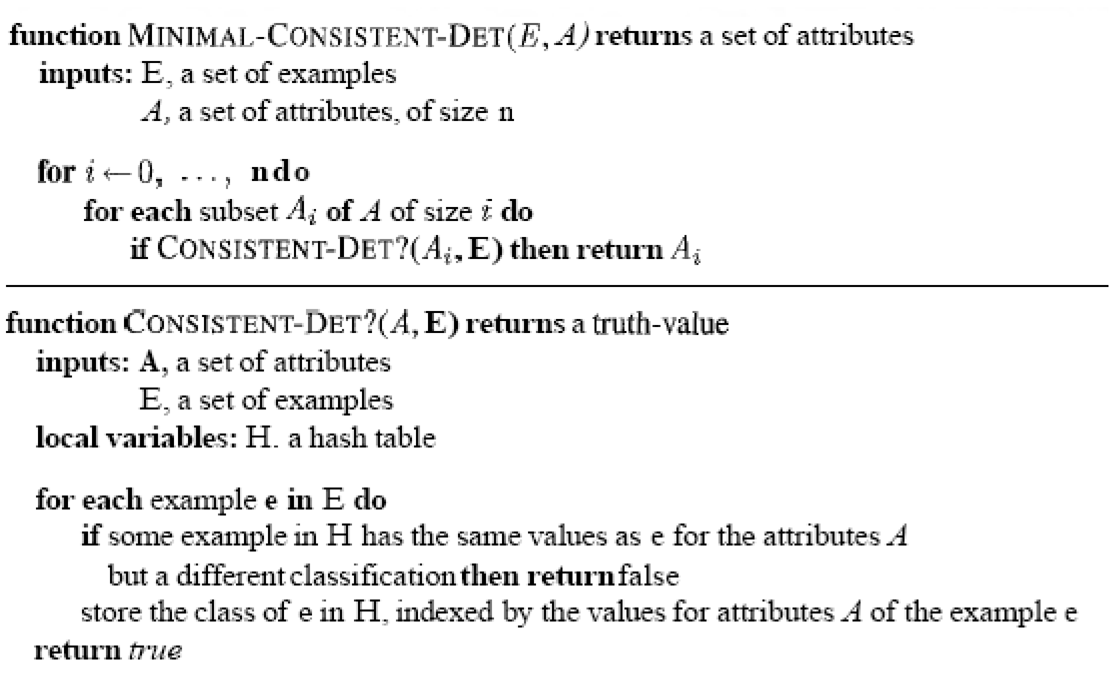
Sentences express a strict form of relevance: given nationality, language is fully determined. (Put another way: language is a function of nationality.) These sentences are called **functional dependencies** or **determinations.** They occur so commonly in certain kinds of applications (e.g., defining database designs) that a special syntax is used to write them. We adopt the notation of Davies (1985):

Nationality (x , n) + Language(x, 1)

Determining the hypothesis space

Determinations specify a sufficient basis vocabulary from which to construct hypotheses concerning the target predicate. This statement can be proven by showing that a given determination is logically equivalent to a statement that the correct definitions of the target predicate is one of the set of all definitions expressible using the predicates on the left-hand side of the determination.

Learning and using relevance information



```
function MINIMAL-CONSISTENT-DET($E, A$) returns a set of attributes
   inputs: E, a set of examples
           $A$, a set of attributes, of size n

   for $i \leftarrow 0, \ldots,$ n do
      for each subset $A_i$ of $A$ of size $i$ do
         if CONSISTENT-DET?($A_i$, E) then return $A_i$
```

```
function CONSISTENT-DET?($A$, E) returns a truth-value
   inputs: A, a set of attributes
           E, a set of examples
   local variables: H, a hash table

   for each example e in E do
      if some example in H has the same values as e for the attributes $A$
         but a different classification then return false
      store the class of e in H, indexed by the values for attributes $A$ of the example e
   return true
```

Figure: An algorithm for finding a minimal consistent determination.

4.10 Inductive logic programming

Inductive logic programming (ILP) combines inductive: methods with the power of first-order representations, concentrating in particular on the representation of theories as logic program .It has gained popularity for three reasons. First, ILP offers a rigorous approach to the general knowledge-based inductive learning problem. Second, it offers complete algorithms for inducing general, first-order theories from examples, which can therefore learn successfully in domains where attribute-based algorithms are hard to apply.

An example

The general knowledge-based induction problem is to "solve" the entailment constraint

Background A Hypothesis A Descriptions = (7lasszfications

For the unknown Hypothesis, given the Background knowledge and examples described by Descriptions and Classifications.

The corresponding descriptions are as follows:

Father (Philip, Charles) Father (Philip, Anne) . . .

Mother(Mum, Margaret) Mother(Mum, Elizabeth) . . .

Married(Diana, Charles) Married(Elixabeth, Philip) . . .

Male (Philip) Male(Charles) . . .

Fenaale (Beatrice) Female (Margaret:) . . .

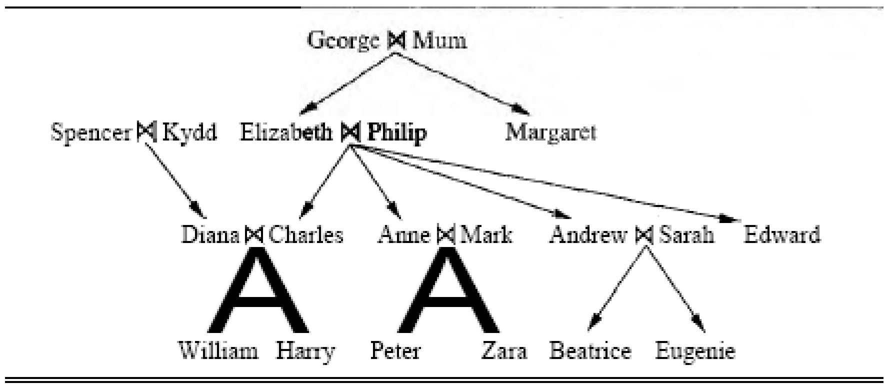


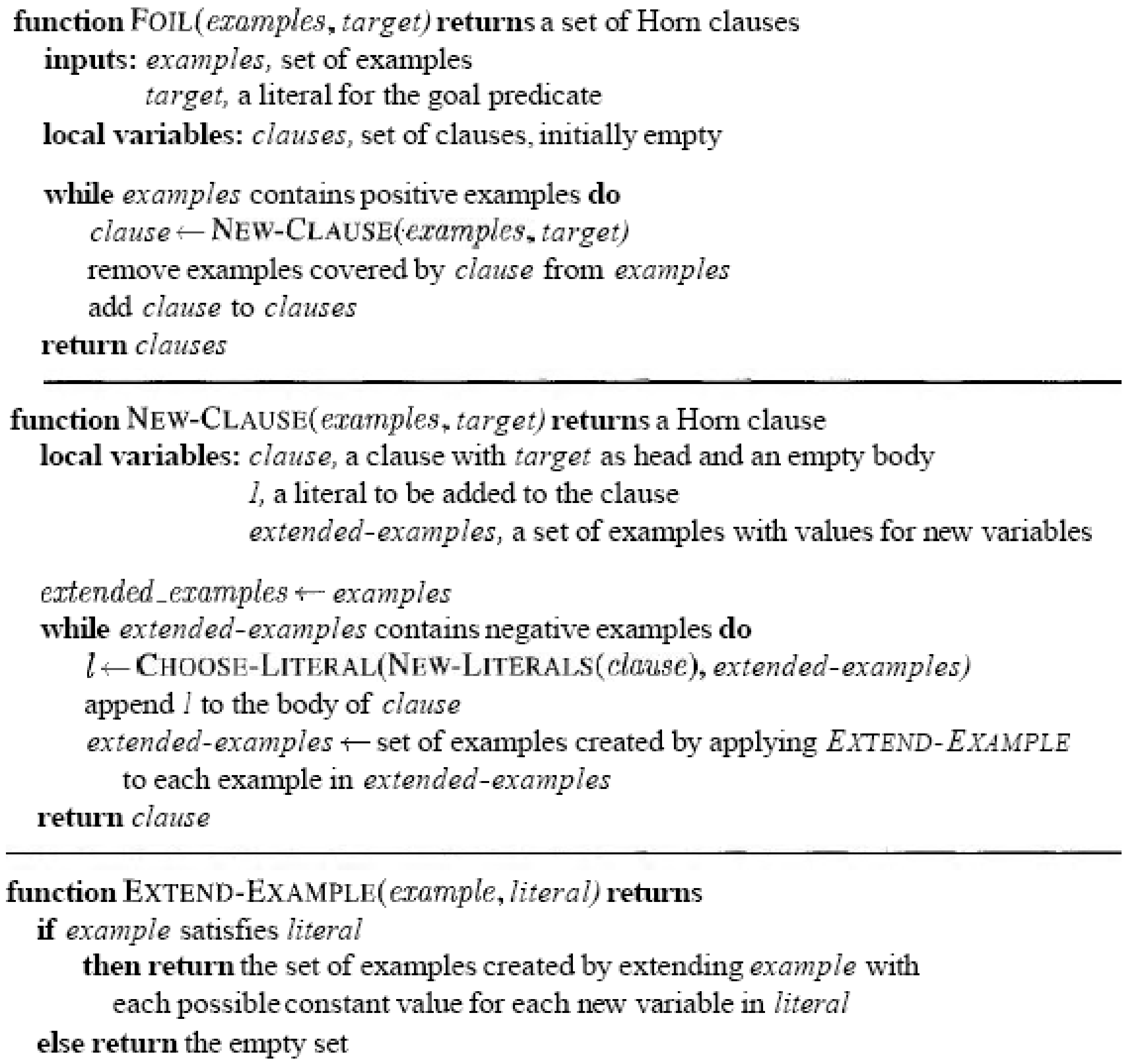
Figure: A typical family tree.

The object of an inductive learning program is to come up with a set of sentences for the Hypothesis such that the entailment constraint is satisfied. Suppose, for the moment, that the agent has no background knowledge: Background is empty

Attribute-based learning algorithms are incapable of learning relational predicates**.**

Top-down inductive learning methods

The first approach to ILP works by starting with a very general rule and gradually specializing it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. To do ILP we use first-order literals instead of attributes, and the hypothesis is a set of clauses instead of a decision tree.



```
function FOIL(examples, target) returns a set of Horn clauses
  inputs: examples, set of examples
          target, a literal for the goal predicate
  local variables: clauses, set of clauses, initially empty

  while examples contains positive examples do
     clause ← NEW-CLAUSE(examples, target)
     remove examples covered by clause from examples
     add clause to clauses
  return clauses

function NEW-CLAUSE(examples, target) returns a Horn clause
  local variables: clause, a clause with target as head and an empty body
                   l, a literal to be added to the clause
                   extended-examples, a set of examples with values for new variables

  extended_examples ← examples
  while extended-examples contains negative examples do
     l ← CHOOSE-LITERAL(NEW-LITERALS(clause), extended-examples)
     append l to the body of clause
     extended-examples ← set of examples created by applying EXTEND-EXAMPLE
        to each example in extended-examples
  return clause

function EXTEND-EXAMPLE(example, literal) returns
  if example satisfies literal
     then return the set of examples created by extending example with
        each possible constant value for each new variable in literal
  else return the empty set
```

Figure: Sketch of the *FOIL* algorithm for learning sets of first-order

Inductive learning with inverse deduction

The second major approach to ILP involves inverting the normal deductive proof process.

Inverse resolution is based on the observation that if the example Classifications follow

from *Background* A *Hypothesis A Descriptions,* then one must be able to prove this fact by resolution (because resolution is complete). ***12*** numbers of approaches to taming the search have been tried in implemented ILP systems:

1. Redundant choices can be eliminated
2. The proof strategy can be restricted.
3. The representation language can be restricted,
4. Inference can be done with model checking rather than theorem proving.
5. Inference can be done with ground propositional clauses rather than in first-order logic.

Making discoveries with inductive logic programming

An inverse resolution procedure that inverts a complete resolution strategy is, in principle, a complete algorithm for learning first-order theories.

4.11 Statistical learning methods

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis.

The key quantities in the Bayesian approach are the hypothesis prior, P(hi*),*a nd the likelihood of the data under each hypothesis, P*(*dJh i*).*

4.12 Learning with complete data

A statistical learning method begins with the simplest task: parameter learning with complete data. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed.

Maximum-likelihood parameter learning: Discrete models

In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

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

A significant problem with maximum-likelihood learning in general: “*when the data set is small enough that some events have not yet been observed-for instance, no cherry candies-the maximum Likelihood hypothesis assigns zero probability to those events”*.

The most important point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parametez3.* The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naïve Bayes** model. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables *Xi* are the leaves. The model is "naive7' because it assumes that the attributes are conditionally independent of each other, given the class.

Maximum-like likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model. The principles for maximum likelihood learning are identical to those of the discrete case. Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated also follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The parameters of this model are the mean, Y and the standard deviation *a.*

The quantity (yj - (B1xj + 02)) is the **error** for (zj, yj)-that is, the difference between the

actual value yj and the predicted value (01 x j + $2)-SO E is the well-known **sum of squared errors.** This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum likelihood straight-line model, *provided that the data are generated with Gaussian noise of* fi*xed variance.*

Bayesian parameter learning

The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra "principles of learning." Furthermore, *there is, in essence, just one learning algorithm,* i.e., the inference algorithm for Bayesian networks.

Learning Bayes net structures

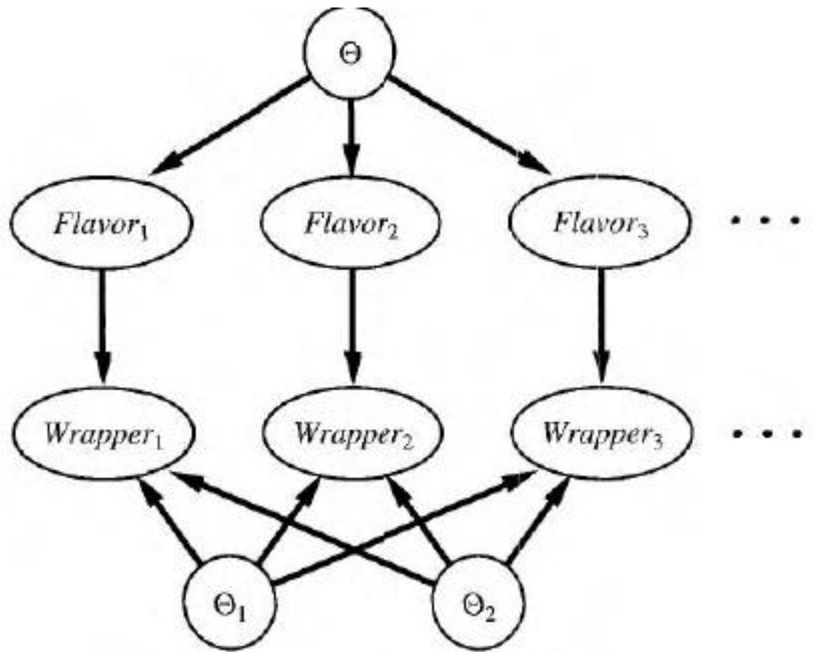


Figure: A Bayesian network that corresponds lo a Bayesian learning process. Posterior distributions for the parameter variables 0, el, and *e2* can be inferred from their prior distributions and the evidence in the *Flavor,* and *Wrapper* variables.

There are two alternative methods for deciding when a good structure has been found.

The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data

4.13 Learning with hidden variable

Many real-world problems have hidden variables (sometimes called latent variables) which are not observable in the data that are available for learning.

For example, medical records often include the observed symptoms, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself.

Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network.*

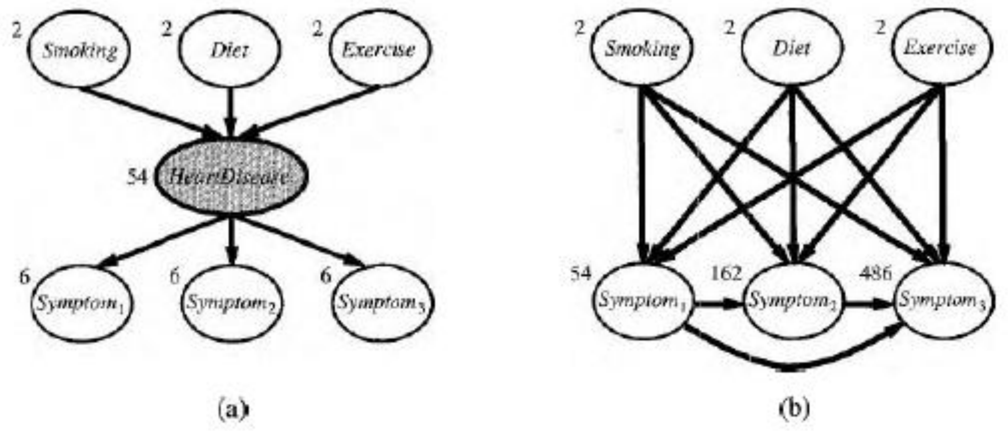


Figure*:* (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *Heart Disease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

Unsupervised clustering: Learning mixtures of Gaussians

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category 1abels is not given.

Clustering presumes that the data are generated from a **mixture distribution,** P. Such a distribution has *k* **components,** each of which is a distribution its own right.

A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable *C* denote the component, with values

1 . . . . *K;* then the mixture distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)\ P(\mathbf{x}|C=i)\ ,$$

where x refers to the values of the attributes for a data point.

For the mixture of Gaussians, we initialize the mixture model parameters arbitrarily and then iterate the following two steps:

1. **E-step**: Compute the probabilities *pij = P(C =* i *l x j)* , the probability that datum *x j* was generated by component i. By Bayes7 rule, we have *pij = a P (x j* IC = *i) P(C =* ***i).*** The term *P (x j* IC = i) is just the probability at *x j* of thle ith Gaussian, and the term *P(C* = i) is just the weight parameter for the ith Gaussian. Define *pi = C j pij*.

2. **M-step**: Compute the new mean, covariance, and component eights as follows:

$$\boldsymbol{\mu}_i \leftarrow \sum_j p_{ij}\mathbf{x}_j / p_i$$

xi + C pii (x j - pi) (xi - P i) / p i

W i + p i .

The E-step, or *expectation* step, can be viewed as computing; the expected values *p,:,* of the hidden **indicator variables** *Z,,,* where *Z,,* is 1 if daturn **x3** was generated by the ith component and 0 otherwise.

Learning Bayesian networks with hidden variables

The general lesson from this example is that *the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover only* local *posterior probabilities are needed for each parameter.*

Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs).Markov model can be represented by a dynamic Bayes net with a single discrete state variable. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences.

The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then ire computing the parameters, using the expected values as if they were observed values. Let **x** be all the observed values in all the examples, let **Z** denote all the hidden variables for all the examples, and let **8** be all the parameters for the probability model. Then the EM algorithm is

$$\boldsymbol{\theta}^{(i+1)} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{\mathbf{z}} P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{Z}=\mathbf{z}|\boldsymbol{\theta})$$

This equation is the EM algorithm in a nutshell.

Learning Bayes net structures with hidden variables

In the simplest case, the hidden variables are listed along with the observed variables; although their values are not observed, the learning algorithm is told that they exist and must find a place for them in the network structure. The latter approach can be implemented by including new modification choices in the structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity.

One possible improvement is the so-called **structural** EM algorithm, which operates in much the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters.

4.14 Instance based learning

Parametric learning methods are often simple and effective, but assuming a particular restricted family of models often oversimplifies what's happening in the real world, from where the data come. In contrast to parametric learning, **nonparametric learning** methods allow the hypothesis complexity to grow with the data.

Two very simple families of nonparametric **instance-based learning** (or **memory-based learning)** methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models

The key idea of **nearest-neighbor** models is that the properties of any particular input point x are likely to be similar to those of points in the neighborhood of x.

The k-nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well. It is a good thing to try first on a new learning problem. For large data sets, however, we require an efficient mechanism for finding the nearest neighbors of a query point x-simply calculating the distance to every point would take far too long. A variety of ingenious methods have been proposed to make this step efficient by preprocessing the training data. Unfortunately, most of these methods do not scale well with the dimension of the space (i.e., the number of features).

Kernel models

In a kernel model, we view each training instance as generating a little density function-a kernel function-of its own. The density estimate as a whole is just the normalized sum of the entire little kernel functions. A training instance at xi will generate a kernel function K(x, xi) that assigns a probability to each point x in the space. Thus, the density estimate is 1. p(x) = *N C* ~ (xxi,) .The kernel function normally depends only on the *distance* ***D(x,xi)*** from ***x*** to the instance ***xi.***The most popular kernel function is (of course) the Gaussian. For simplicity, we will assume spherical Gaussians with standard deviation w along each axis, i.e.

$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x},\mathbf{x}_i)^2}{2w^2}},$$

where d is the number of dimensions in x.

Supervised learning with kernels is done by taking a *weighted* combination of *all* the predictions from the training instances.

4.15 Neural networks

A neuron is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. The brain's information-processing capacity is thought to emerge primarily from *networks* of such neurons. For this reason, some of the earliest **A1** work aimed to create artificial **neural networks.** (Other names for the field include **connectionism, parallel distributed processing,** and **neural computation.)**

Units in neural networks

Neural networks are composed of nodes or **units** connected by directed **links. A** link from unit j to unit i serve to propagate the **activation** *aj* from **j** to *i*. Each link also has a numeric **weight** *Wj* associated with it, which determines strength and sign of WEIGHT the connection. Each unit i first computes a weighted sum of its inputs: *N* Then it applies an **activation function** g to this sum to derive the output: Notice that we have included a **bias weight** *Wo,i* connected to a fixed input *ao* = - *1*.

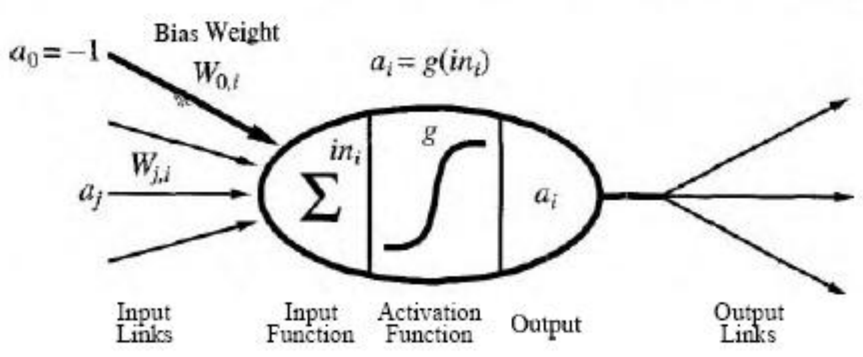


Figure: A simple mathematical model for a neuron.

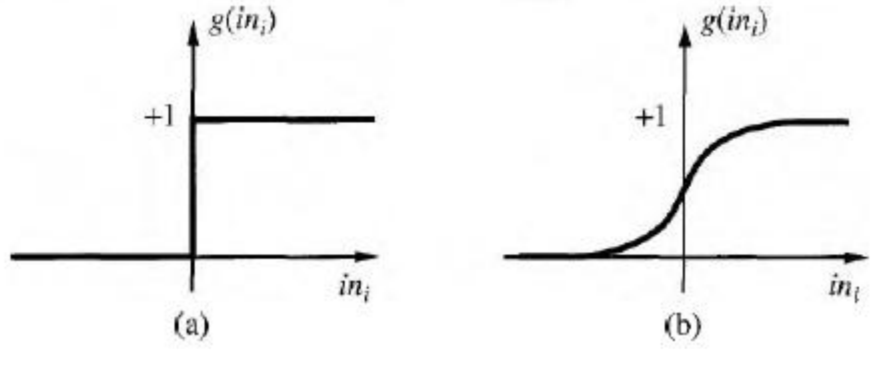


Figure 4.13 (a) The threshold activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the sign function is used instead, which

outputs are depending on the sign of the input.) (b) The sigmoid function 1/ (1 + ***e-"*****).**

Figure shows how the Boolean functions AND, OR, and NOT can be represented by threshold units with suitable weights. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

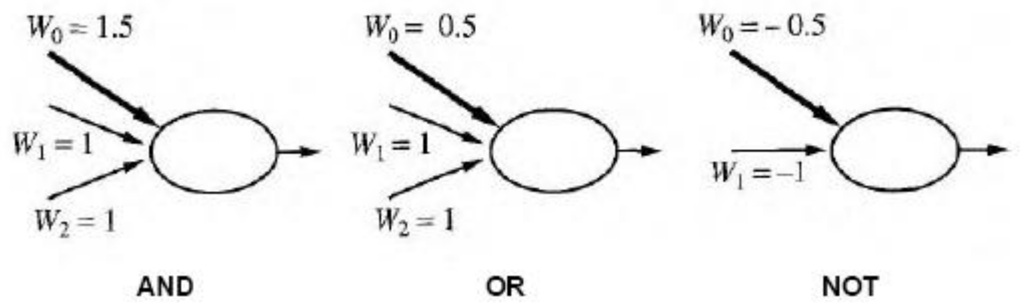


Figure: Units with a threshold activation function can act as logic gates, given appropriate input and bias weights.

Network structures

There are two main categories of neural network structures: acyclic or **feed-forward net-works** and cyclic or **recurrent networks.** A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent network, on the other hand, feeds its outputs back into1 its own inputs.

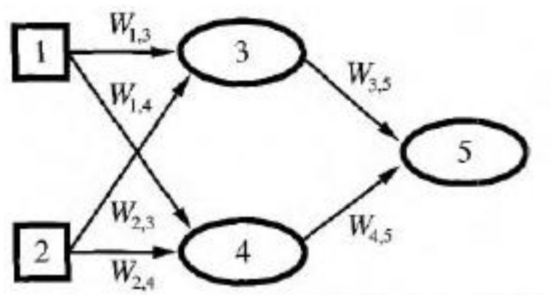


Figure: A very simple neural network with two inputs, one hidden layer of two units, and one output.

A neural network can be used for classification or regression.

Feed-forward networks are usually arranged in **layers,** such that each unit receives input only from units in the immediately preceding layer.

Single layer feed-forward neural networks (perceptrons)

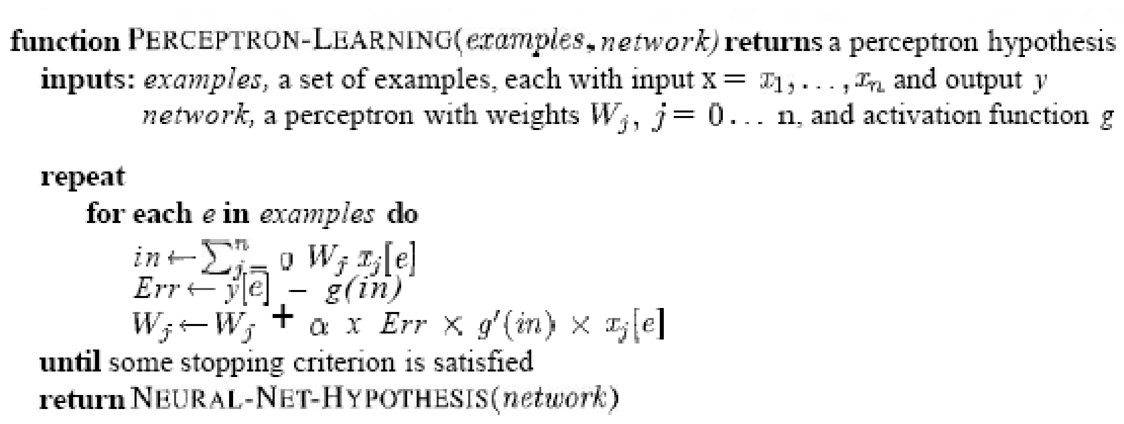
A network with all the inputs connected directly to the outputs is called a **single-layer neural** i O R **network,** or a **perceptron** network. Since each output unit is independent of the others-each weight affects only one of the outputs.

In general, *threshold perceptrons can, represent only linearly separable functions.*

Despite their limited expressive power, threshold perceptrons have some advantages.

In particular, *there is a simple learning algorithm that wills jii a threshold perceptron to any linearly separable training set.*

The complete algorithm is shown in Figure



```
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
   inputs: examples, a set of examples, each with input $x = x_1, \ldots, x_n$ and output $y$
           network, a perceptron with weights $W_j$, $j = 0 \ldots n$, and activation function $g$

   repeat
      for each $e$ in examples do
         $in \leftarrow \sum_{j=0}^{n} W_j \, x_j[e]$
         $Err \leftarrow y[e] - g(in)$
         $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
   until some stopping criterion is satisfied
   return NEURAL-NET-HYPOTHESIS(network)
```

Figure: The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g.

Notice that *the weight-update vector for maximum likelihood learning in sigmoid perceptrons is essentially identical to the update vector for squared error minimization.*

Multilayer feed-forward neural networks

The advantage of adding hidden layers is that it enlarges the space of hypotheses that the network can represent.

Learning algorithms for multilayer networks are similar to the perceptron learning algorithm. The back-propagation process can be summarized as follows:

Compute the A values for the output units, using the observed error.

Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:

- Propagate the A values back to the previous layer.
- Update the weights between the two layers.

The detailed algorithm is shown in Figure

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
   inputs: examples, a set of examples, each with input vector x and output vector y
           network, a multilayer network with L layers, weights $W_{j,i}$, activation function $g$

   repeat
      for each $e$ in examples do
         for each node $j$ in the input layer do $a_j \leftarrow x_j[e]$
         for $\ell = 2$ to L do
            $in_i \leftarrow \sum_j W_{j,i}\ a_j$
            $a_i \leftarrow g(in_i)$
         for each node $i$ in the output layer do
            $\Delta_i \leftarrow g'(in_i)\ x\ (y_i[e] - a_i)$
         for $\ell = L - 1$ to 1 do
            for each node $j$ in layer $\ell$ do
               $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i}\ \Delta_i$
               for each node $i$ in layer $\ell + 1$ do
                  $W_{j,i} \leftarrow W_{j,i} + \alpha\ x\ a_j\ x\ \Delta_i$
   until some stopping criterion is satisfied
   return NEURAL-NET-HYPOTHESIS(network)
```

Figure: The back-propagation algorithm for learning in multilayer networks.

Learning neural network structures

We want to consider networks that are not fully connected, and then we need to find some effective search method through the very large space of possible connection topologies. The optimal brain damage algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

4.16 Reinforcement learning

The problem is this: *without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make.* The agent needs to know that something good has happened when it wins and that something bad has happened when it loses. This kind of feedback is called a **reward,** or **reinforcement.**

The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment.

Three of the agent designs

a utility-based agent learns a utility function on states and uses it to select actions that maximize the expected outcome utility.

a Q-learning agent learns an **action-value** function, or Q-function, giving the expected utility of taking a given action in a given state.

a reflex agent learns a policy that maps directly from states to actions.

Types of reinforcement learning.

1. Passive reinforcement learning
2. Active reinforcement learning

4.17 Passive reinforcement learning

To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment. In passive learning, the agent's policy T is fixed: in state s, it always executes the action (s) It s goal is simply to learn how good the policy is-that is, to learn the utility function UT(s).

The main difference is that the passive learning agent does not know the **transition model** T(s, a, s'), which specifies the probability of reaching state s' from state s after doing action a; nor does it know the **reward function** R (s), which specifies the reward for each state.

The utility is defined to be the expected sum of (discounted) rewards obtained if policy .it is followed. This is written as

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s\right]$$

Direct utility estimation

A simple method for **direct utility estimation** was invented in the late 1950s in the area of **adaptive control theory.** The idea is that the utility of a THEORY state is the expected total reward from that state onward, and each trial provides a *sample* of this value for each state visited.

Adaptive dynamic programming

An **adaptive dynamic programming** (or **AIDP)** agent works by learning the transition model of the environment as it goes along and solving the corresponding Markov decision process using a dynamic programming method.

The full agent program for a passive ADP agent is shown in Figure

```
function PASSIVE-ADP-AGENT(percept) returns an action
    inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
    static: $\pi$, a fixed policy
            mdp, an MDP with model T, rewards R, discount $\gamma$
            $U$, a table of utilities, initially empty
            $N_{sa}$, a table of frequencies for state-action pairs, initially zero
            $N_{sas'}$, a table of frequencies for state-action-state triples, initially zero
            $s$, a, the previous state and action, initially null

    if $s$ is new then do $U[s] \leftarrow r$ ; $R[s] \leftarrow r'$
    if $s$ is not null then do
        increment $N_{sa}[s, a]$ and $N_{sas'}[s, a, s]$
        for each $t$ such that $N_{sas'}[s, a, t]$ is nonzero do
            $T[s, a, t] \leftarrow N_{sas'}[s, a, t] / N_{sa}[s, a]$
    $U \leftarrow$ POLICY- EVALUATION(^,  $U$, mdp)
    if TERMINAL?[$s'$] then $s, a \leftarrow$ null else $s$, a $\leftarrow s$ , $\pi[s']$
    return $a$
```

Figure: A passive reinforcement learning agent based on adaptive dynamic programming.

```
function PASSIVE-TD-AGENT(percept) returns an action
    inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
    static: n-, a fixed policy
            $U$, a table of utilities, initially empty
            $N_s$, a table of frequencies for states, initially zero
            $s$, a, $r$, the previous state, action, and reward, initially null

    if $s'$ is new then $U[s'] \leftarrow r'$
    if $s$ is not null then do
        increment $N_s[s]$
        $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
    if TERMINAL?[$s'$] then $s, a, r \leftarrow$ null else $s$, a, $r \leftarrow s$ , $\pi[s']$, $r'$
    return $a$
```

Figure: A passive reinforcement learning agent that learns utility estimates using temporal differences.

Temporal difference learning

It is possible to have (almost) the best of both worlds; that is, one can approximate the constraint equations shown earlier without solving them for all possible states. *The key is to use the observed transitions to adjust the values of the observed states so that they agree* *with the corzstraint equations.*

4.18 Active reinforcement learning

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take.

Exploration

EXPLORATION maximizes its reward-as reflected in its current utility estimates-and **exploration** to maximize its long-term well-being. Pure exploitation risks getting stuck in a rut. Pure exploration to improve one's knowledge is of no use if one never puts that knowledge into practice

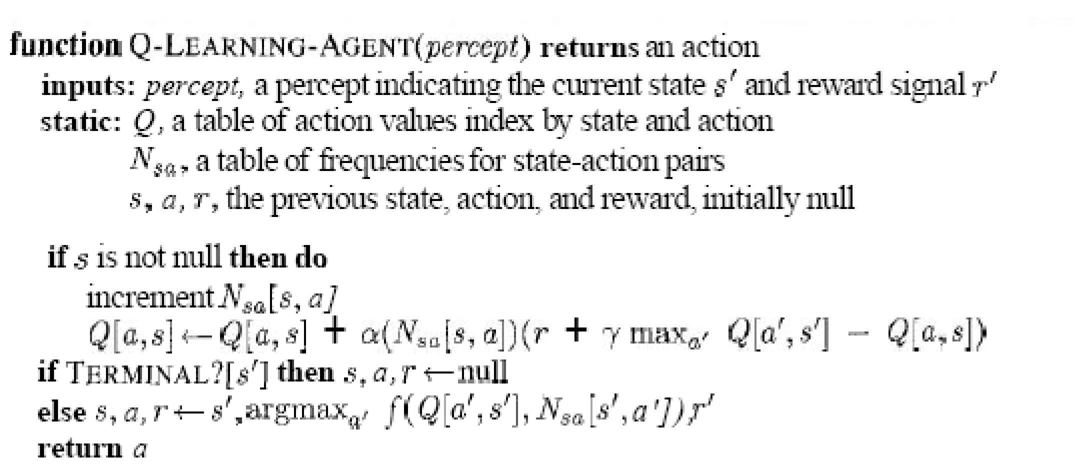
The following equation does this:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} T(s,a,s')U^+(s'),\ N(a,s)\right)$$

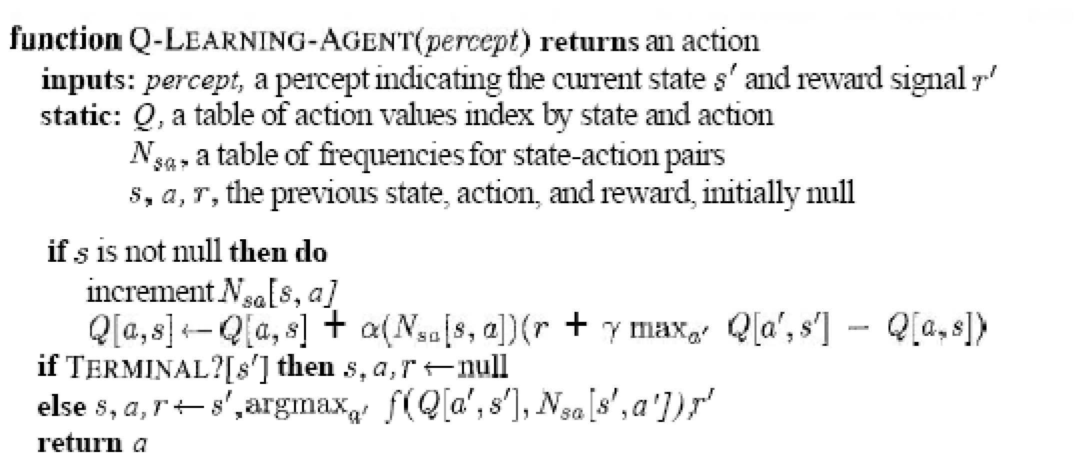
Here, f (u, n) is called the **exploration function.**

Learning an Action-Value Function

Q-learning at learns an action-value representation instead of learning utilities. a very important property: *a* TD *agent that learns a Q-function does not need a model* **for** *either learning or action selection.*



```
function Q-LEARNING-AGENT(percept) returns an action
   inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
   static: $Q$, a table of action values index by state and action
           $N_{sa}$, a table of frequencies for state-action pairs
           $s$, $a$, $r$, the previous state, action, and reward, initially null

    if $s$ is not null then do
       increment $N_{sa}[s, a]$
       $Q[a,s] \leftarrow Q[a,s] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[a',s'] - Q[a,s])$
   if TERMINAL?[$s'$] then $s, a, r \leftarrow$ null
   else $s, a, r \leftarrow s'$, argmax$_{a'}$ $f(Q[a',s'], N_{sa}[s',a'])$, $r'$
   return $a$
```



```
function Q-LEARNING-AGENT(percept) returns an action
   inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
   static: $Q$, a table of action values index by state and action
           $N_{sa}$, a table of frequencies for state-action pairs
           $s$, $a$, $r$, the previous state, action, and reward, initially null

    if $s$ is not null then do
       increment $N_{sa}[s, a]$
       $Q[a,s] \leftarrow Q[a,s] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[a',s'] - Q[a,s])$
   if TERMINAL?[$s'$] then $s, a, r \leftarrow$ null
   else $s, a, r \leftarrow s'$, argmax$_{a'}$ $f(Q[a',s'], N_{sa}[s',a'])$, $r'$
   return $a$
```

Figure : An exploratory Q-learning agent.

4.19 Generalization in reinforcement learning

Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit. *The compression achieved by a function approximated allows the learning agent to generalization it has visited to states it has not visited.*

Question Bank

UNIT –IV

PART-A

1.Differentiate supervised learning with unsupervised learning?
2.What is meant by reinforcement learning?
3. Define decision tree with an example.
4.Define regression.
5.Define over fitting.
6.What is meant by cross validation?
7. List the general uses of decision tree learning system?
8. what is meant by boosting?
9.Define the terms false negative and false positive for the hyposthesis.
10. What is meant by memorization?
11. What is meant by functional dependencies or determinations?
12. Define Baye's rule.
13. What is meant by Artificial Neural Network(ANN)?
14. What is need of activation functions in neural network?
15. Differentiate real neuron with simulated neuron.
16. Differentiate feed forward with recurrent network.
17. What is meant by learning rate?
18. Differentiate passive learning with active learning.
19. Write short notes on Q-functions.
20. Write short notes on perception.
21. Differentiate nearest-neighbor method with Kernal method.
22. What is meant by cumulative learning?
23. List some of the applications of learning.
24. Define learning.
25. List the different types of learning methods.

PART-B

1. Explain how the decision-tree-learning is done with an example and algorithm.
2. Explain the ensemble learning with boosting algorithm.
3. Explain the version space/candidate elimination algorithm with an example.
4. How to extract general rules from an example?Explain.
5. Explain how learning with complete data is achieved?
6. Explain in detail EM algorithm.
7. Explain the back-propagation algorithm of learning in a multilayer neural network.
8. Explain passive reinforcement learning agent with (i) Adaptive Dynamic Programming (ADP) (ii) Temporal Difference(TD)

UNIT – V

APPLICATIONS

Communication − Communication as action − Formal grammar for a fragment of English − Syntactic analysis − Augmented grammars − Semantic interpretation − Ambiguity and disambiguation − Discourse understanding − Grammar induction − Probabilistic language processing − Probabilistic language models − Information retrieval − Information extraction − Machine translation.

5.1 Introduction - Communication

Communication is the intentional exchange of information brought about by the production and perception of **signs** drawn from a shared system of conventional signs.

What sets humans apart from other animals is the complex system of structured messages known as **language** that enables us to communicate most of what we know about the world.

5.2 Communication as action

One of the actions available to an agent is to produce language. This is called **speech act**. The generic terms referring to any mode of communication is

Speaker → Utterance → Hearer

The different types of terms used in speech act are:

Inform
Query
Request
Acknowledge
Promise

Fundamentals of Language

A **formal language** is defined as a set of strings. Each string is a concatenation of terminal symbols, called **words**.

A **grammar** is a finite set of rules that specifies a language. The grammar is a set of rewrite rules.

Example: S – sentence

NP – Noun phrase

VP – Verb phrase

These are called **non terminal symbols**.

The Component steps of communication

A typical communication episode, in which speaker S wants to inform hearer H about proposition P using words W, is composed of **seven** steps.

1. Intention
2. Generation
3. Synthesis
4. Perception
5. Analysis
6. Disambiguation
7. Incorporation

The following figure shows the seven processes involved in communication using the example sentence “The wumpus is dead”.

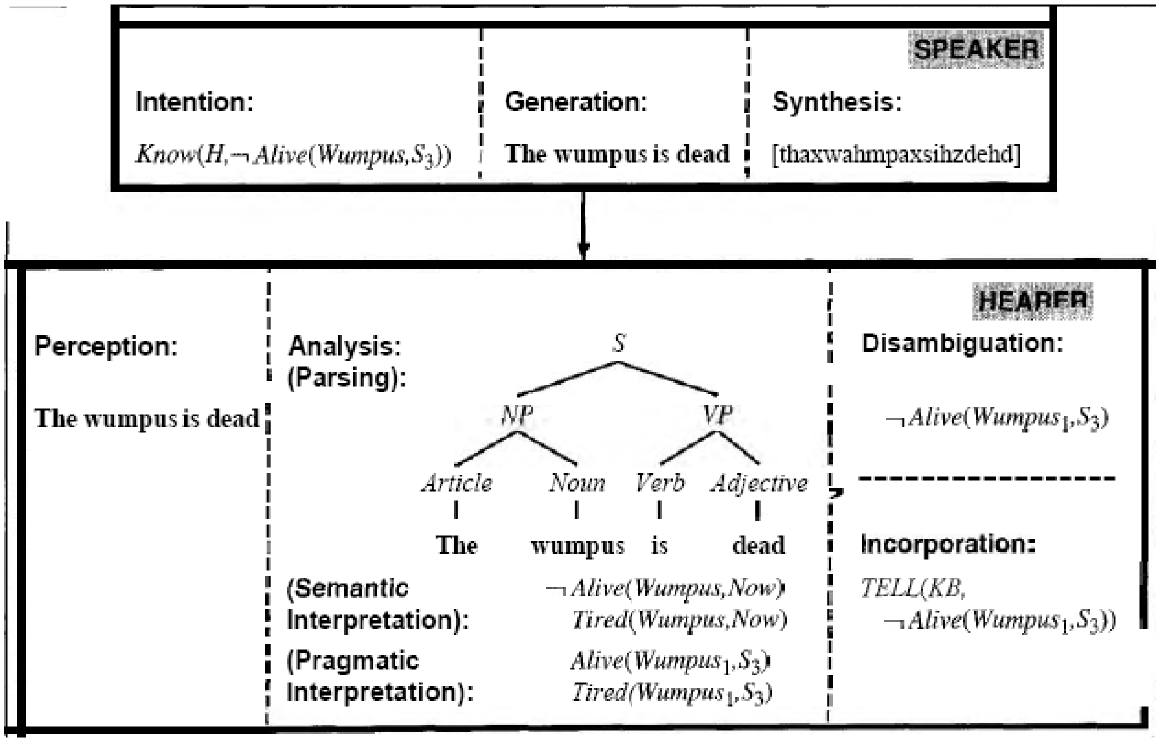


Figure: Seven processes involved in communication

5.3 Formal grammar for a fragment of English

A formal grammar for a small fragment of English to make statements about the wumpus world is constructed and this language$\mathcal{E}_0$.

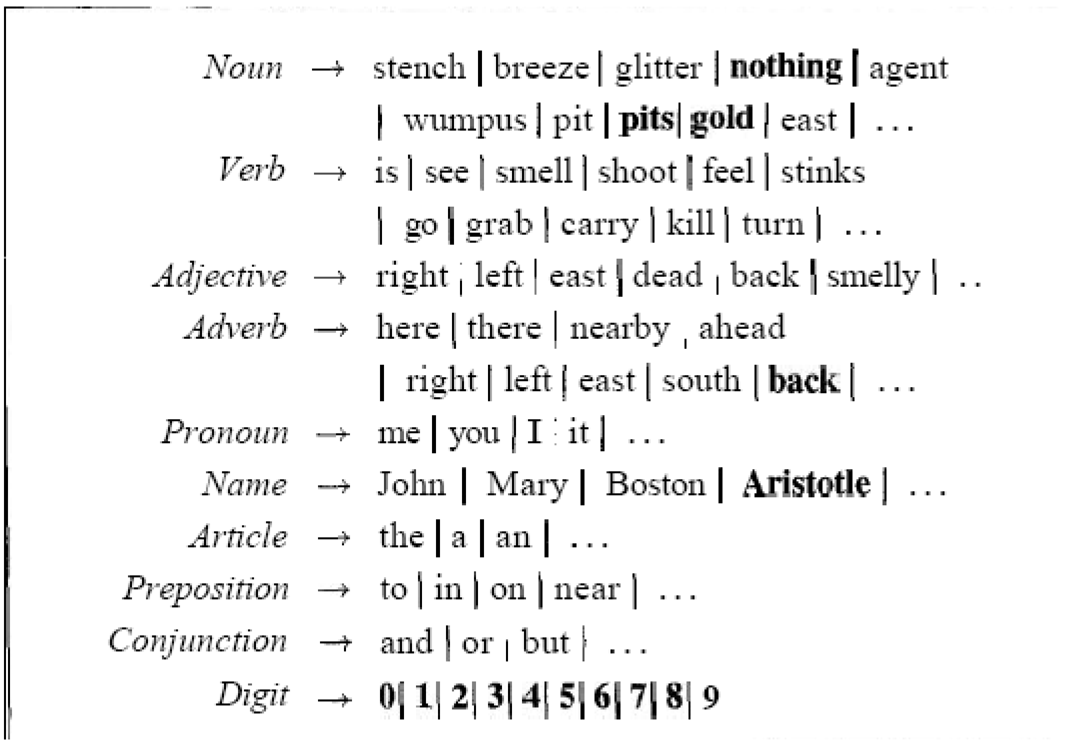
The lexicon of$\mathcal{E}_0$:

A list of allowable words, grouped into categories, such as

- Nouns
- Pronouns
- Names to things
- Verbs to denote events
- Adjectives to modify nouns

- Adverbs to modify verbs
- Articles
- Preposition
- Conjunction

The following figure shows a small lexicon.



Noun	→	stench | breeze | glitter | **nothing** | agent | wumpus | pit | **pits**| **gold** | east | . . .
Verb	→	is | see | smell | shoot | feel | stinks | go | grab | carry | kill | turn | . . .
Adjective	→	right | left | east | dead | back | smelly | . .
Adverb	→	here | there | nearby | ahead | right | left | east | south | **back** | . . .
Pronoun	→	me | you | I | it | . . .
Name	→	John | Mary | Boston | **Aristotle** | . . .
Article	→	the | a | an | . . .
Preposition	→	to | in | on | near | . . .
Conjunction	→	and | or | but | . . .
Digit	→	**0**| **1**| **2**| **3**| **4**| **5**| **6**| **7**| **8**| 9

The Grammar of $\mathcal{E}_0$.

It defines how to combine the word and phrases, using five nonterminal symbols. The different types of phrases are:

- Sentence
- Noun phrase(NP)
- Verb phrase(VB)
- Prepositional phrase(PP)
- Relative clause(Reclause)

The figure shows a grammar for $\mathcal{E}_0$.

S	→	*NP VP*	I + feel a breeze
	|	*S Conjunction S*	I feel a breeze + and + I smell a wumpus
NP	→	*Pronoun*	I
	|	*Name*	John
	|	*Noun*	pits
	|	*Article Noun*	the + wumpus
	|	*Digit Digit*	3 4
	|	*NP PP*	the wumpus + to the east
	|	*NP RelClause*	the wumpus + that is smelly
VP	→	*Verb*	stinks
	|	*VP NP*	feel + a breeze
	|	*VP Adjective*	is + smelly
	|	*VP PP*	turn + to the east
	|	*VP Adverb*	go + ahead
PP	→	*Preposition NP*	to + the east
RelClause	→	that *VP*	that + is smelly

Figure : The grammar for $\mathcal{E}_0$.

5.4 Syntactic analysis (Parsing)

Syntactic analysis is the step in which an input sentence is converted into a hierarchical structure that corresponds to the units of meaning in the sentence. This process is called **parsing**.

Parsing can be seen as a process of searching for a parse tree. There are two extreme ways of specifying the search space.

1. Top-down parsing
2. Bottom-up parsing

1. Top-down parsing:

Begin with the start symbol and apply the grammar rules forward until the symbols at the terminals of the tree correspond to the components of the sentence being parsed.

2. Bottom-up parsing

Begin with the sentence to be parsed and apply the grammar rules backward until a single tree whose terminals are the words of the sentence and whose top node is the start symbol has been produced.

5.5 Augmented grammars

The process of augmenting the existing rules of the grammar instead of introducing new rules. This formalism for augmentations is called **definite clause grammar** or **DCG**. The main advantage of DCG is that we can augment the category symbols with additional arguments.

We define DCG as follows:

- The notation $\mathrm{X} \rightarrow \mathrm{Y}\ Z\ \ldots$ translates as $\mathrm{Y}(s_1)\ \mathrm{A}\ Z(s_2)\ A \ldots \Rightarrow X(s_1 + s_2 + \ldots)$.
- The notation $\mathrm{X} \rightarrow \mathrm{Y} \mid Z \mid \ldots$ translates as $\mathrm{Y}(s) \vee Z(s) \vee \ldots \Rightarrow X(s)$.
- In either of the preceding rules, any nonterminal symbol Y can be augmented with one or more arguments. Each argument can be a variable, a constant, or a function of arguments. In the translation, these arguments precede the string argument (e.g., $NP(case)$ translates as $NP(case, s_1)$).
- The notation $\{P(\ldots)\}$ can appear on the right-hand side of a rule and translates verbatim into $\mathbf{P}(\ldots)$. This allows the grammar writer to insert a test for $\mathbf{P}(\ldots)$ without having the automatic string argument added.
- The notation $\mathrm{X} \rightarrow$ **word** translates as $X([word])$.

5.6 Semantic Interpretation

Semantics – the extraction of the meaning of utterances. Semantic interpretation is the process associating an FOL expression with a phrase.

$$
\begin{aligned}
&Exp(x) \rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2)\ \{x = Apply(op, x_1, x_2)\}\\
&Exp(x) \rightarrow (\ Exp(x)\ )\\
&Exp(x) \rightarrow Number(x)\\
&Number(x) \rightarrow Digit(x)\\
&Number(x) \rightarrow Number(x_1)\ Digit(x_2)\ \{x = 10 \times x_1 + x_2\}\\
&Digit(x) \rightarrow x\ \{0 \leq x \leq 9)\\
&Operator(x) \rightarrow x\ \{x \in \{+, -, \div, \times\}\}
\end{aligned}
$$

Figure: A grammar for arithmetic operations, augmented with semantics.

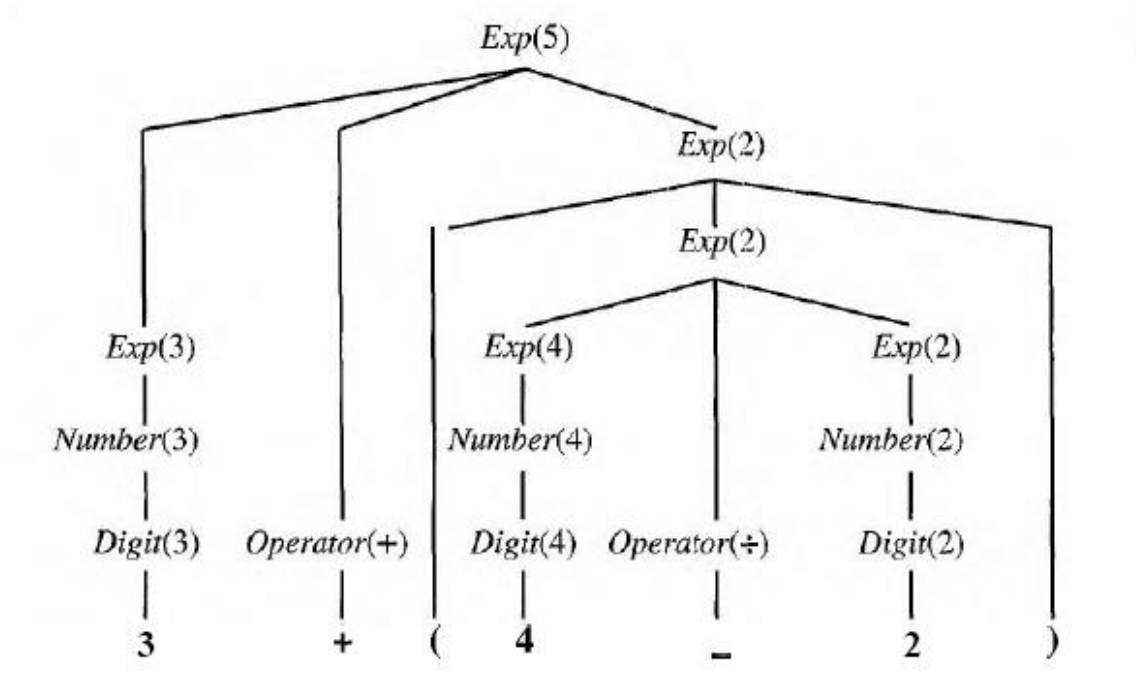


Figure: Parse tree with semantic interpretations for the string 3 + (4 / 2).

The semantics of an English fragment

$$S(rel(obj)) \rightarrow NP(obj)\ VP(rel)$$
$$VP(rel(obj)) \rightarrow Verb(rel)\ NP(obj)$$
$$NP(obj) \rightarrow Name(obj)$$

$$Name(John) \rightarrow \textbf{John}$$
$$Name(Mary) \rightarrow \textbf{Mary}$$
$$Verb(\lambda y\ \textbf{Ax}\ Loves(x,y)) \rightarrow \textbf{loves}$$

Figure: A grammar that can derive a parse tree and semantic interpretation for "John loves Mary".

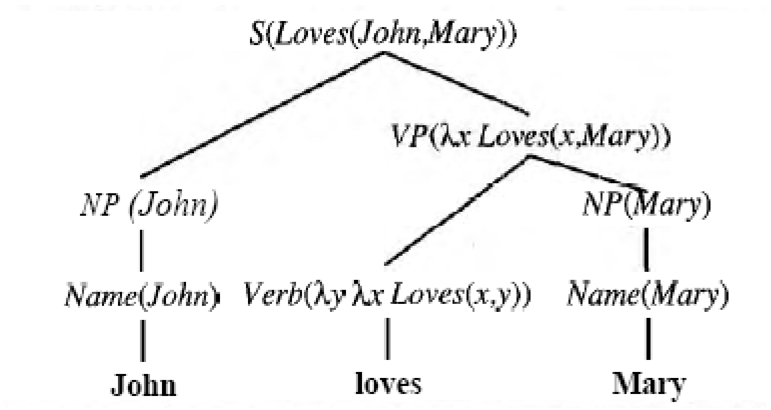


Figure: A parse tree with semantic interpretations for the string "John loves Mary".

5.7 Ambiguity and Disambiguation

A word, phrase or sentence is ambiguous if it has more than one meaning.

Types of ambiguity:

1. **Lexical ambiguity**: in which a word has more than one meaning. It is quite common;

2. **Syntactic ambiguity** (Structural ambiguity): can occur with or without lexical ambiguity.
3. **Semantic ambiguity**: The Syntactic ambiguity leads to a semantic ambiguity, because one parse means that the wumpus is in 2.2 and the other means that a stench is in 2.2. In this case, getting the wrong interpretation could be a deadly mistake. It can occur even in phrases with no lexical or syntactic ambiguity.

A **metonymy** is a figure of speech in which one object is used to stand fro another. A **metaphor** is a figure of speech in which a phrase with one literal meaning is used to suggest s different meaning ay way of an analogy.

Disambiguation

Disambiguation is a question of diagnosis. It is made possible by combining evidence, using all the techniques fro knowledge representation and uncertain reasoning.

We can break the knowledge down into **four** models

1. World model
2. Mental model
3. Language model
4. Acoustic model

5.8 Discourse understanding

A discourse is any string of language – usually one that is more than one sentence long. Textbooks, novels, weather reports and conversations are all discourses. We will look at two particular sub problems:

1. Reference resolution

Reference resolution is the interpretation of a pronoun or a definite noun phrase that refers to an object in the world. The resolution is based on knowledge of the world and of the previous parts of the discourse.

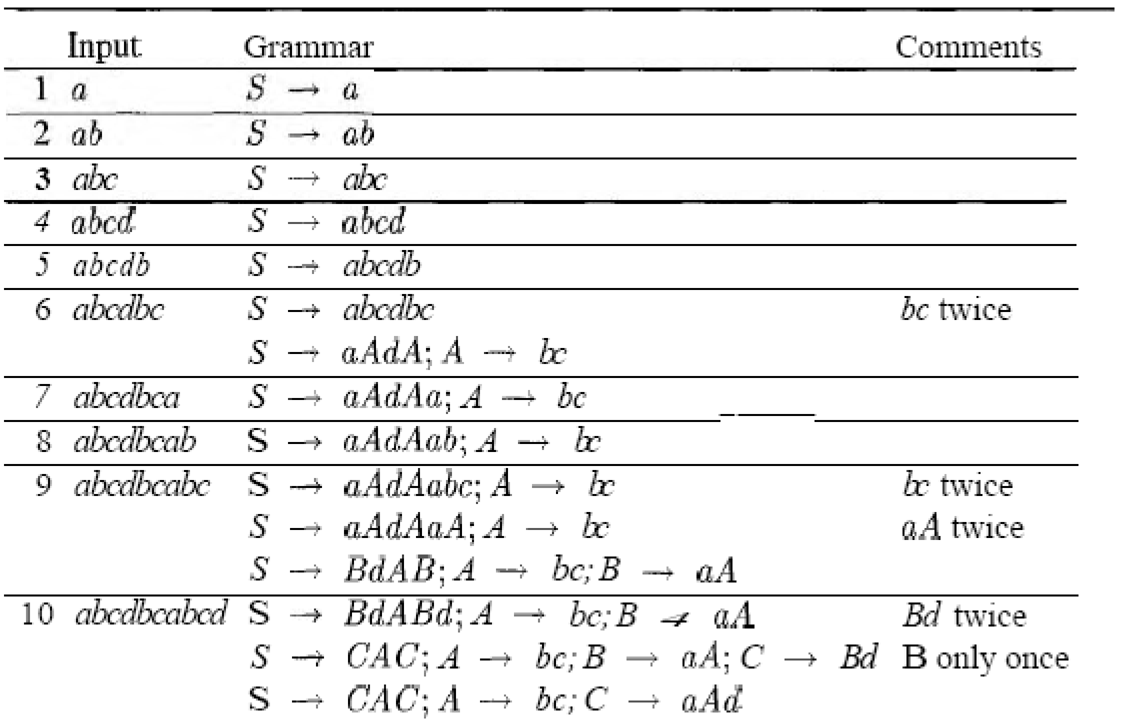
2. The structure of coherent discourse

A discourse has structure above the level of sentence. The grammar says that a discourse is composed of segments, which each segment is either a sentence or a group of sentences and where segments are joined by coherence relations.

Grammar Induction

Grammar induction is the task of learning a grammar from data. It produces a grammar of a very specialized from: a grammar that generates only a single string, namely, the original text.

Figure shows the algorithm in operation in the text "abcdbcabcd"



	Input	Grammar	Comments
1	a	$S \rightarrow a$	
2	ab	$S \rightarrow ab$	
3	abc	$S \rightarrow abc$	
4	$abcd$	$S \rightarrow abcd$	
5	$abcdb$	$S \rightarrow abcdb$	
6	$abcdbc$	$S \rightarrow abcdbc$	bc twice
		$S \rightarrow aAdA; A \rightarrow bc$	
7	$abcdbca$	$S \rightarrow aAdAa; A \rightarrow bc$	
8	$abcdbcab$	$S \rightarrow aAdAab; A \rightarrow bc$	
9	$abcdbcabc$	$S \rightarrow aAdAabc; A \rightarrow bc$	bc twice
		$S \rightarrow aAdAaA; A \rightarrow bc$	aA twice
		$S \rightarrow BdAB; A \rightarrow bc; B \rightarrow aA$	
10	$abcdbcabcd$	$S \rightarrow BdABd; A \rightarrow bc; B \rightarrow aA$	Bd twice
		$S \rightarrow CAC; A \rightarrow bc; B \rightarrow aA; C \rightarrow Bd$	B only once
		$S \rightarrow CAC; A \rightarrow bc; C \rightarrow aAd$	

Figure: The algorithm in operation in the text "abcdbcabcd"

5.9 Probabilistic language processing

Introduction

Probabilistic language models based on n-grams recover a surprising amount of information about a language.

In probabilistic language model, this can learn from data. It is simpler than **Define Clause Grammar** (**DCG**). It has three specific tasks.

1. Information retrieval
2. Information extraction
3. Machine translation

5.10 Probabilistic Language Models

A Probabilistic language model defines a probability distribution over a set of strings.

- It can be conveniently trained from data.
- Learning is just counting number of occurrences.
- It is robust
- It is used for disambiguation-probability can be used to choose the most likely interpretation.

E.g. Bigram and Trigram language model used in speech recognition.

A **unigram model** assigns a probability P (w) to each word in the lexicon. This model assumes that words are chosen in independently, so the probability of a string is just the product of the probability of words.

A **bigram model** assigns a probability $P(w_i|w_{i-1})$ to each word, given the previous word.

In general a n-gram model conditions on the previous n-1 words, assigns a probability for $P(w_i|w_{i-(n-1)} \cdots w_{i-1})$. We need someway of smoothing over the zero counts. The simplest way to do this is called add-one smoothing; we add one to the count of every possible bigram.

Another approach is **linear interpolation smoothing**, which combines trigram, bigram, and unigram models by linear interpolation.

Evaluating a language model

Split the corpus into a training corpus and a test corpus. Determine the parameters of the model from the training data. Then calculate the probability assigned to test corpus by the model. If the probability is higher, then it is better.

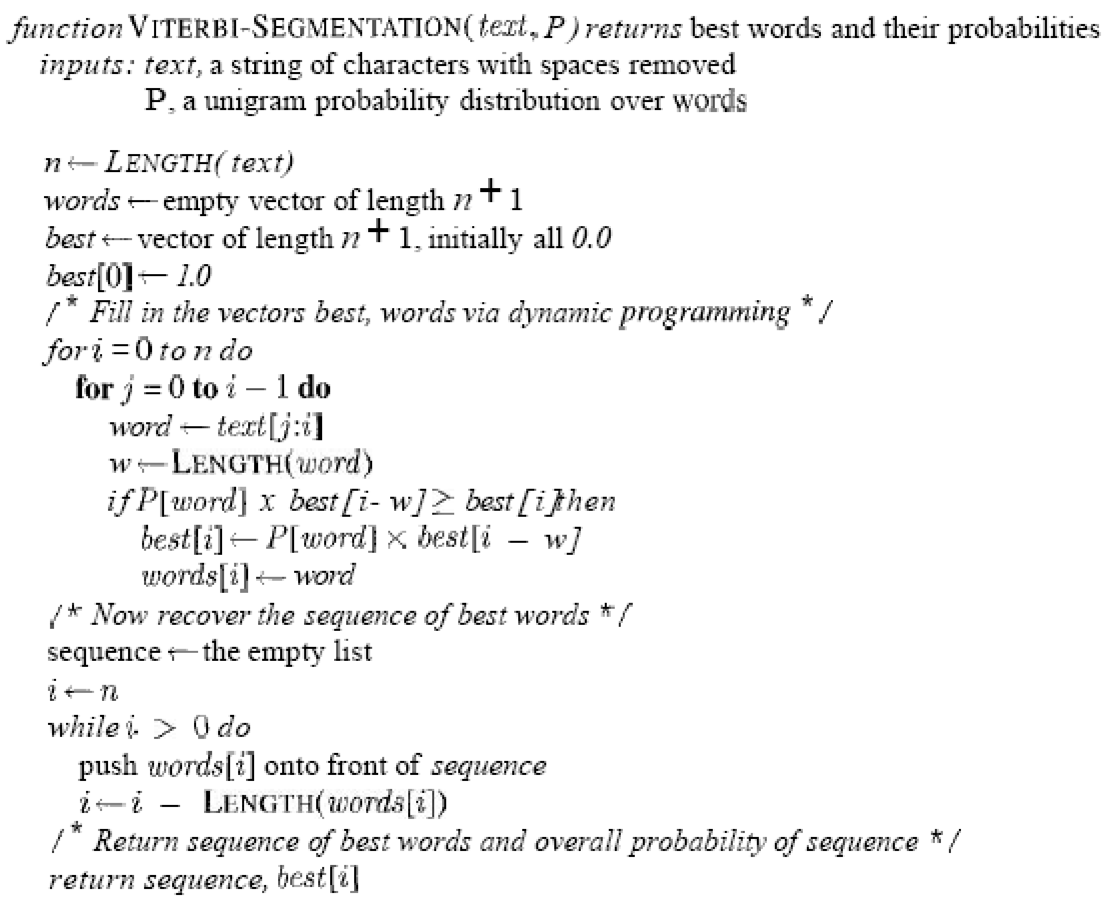
The second approach fro evaluating a model is by computing the perplexity of the model on test string of words.

Segmentation: It is used to find word boundaries in a text with no spaces.

It is used to read words without spaces.

It is easy to read by human because they have full knowledge of English syntax, semantics and pragmatics.

It is done by Viterbi algorithm specifically designed for segmentation problem.



```
function VITERBI-SEGMENTATION(text, P) returns best words and their probabilities
  inputs: text, a string of characters with spaces removed
          P, a unigram probability distribution over words

  $n \leftarrow$ LENGTH(text)
  words $\leftarrow$ empty vector of length $n + 1$
  best $\leftarrow$ vector of length $n + 1$, initially all 0.0
  best[0] $\leftarrow$ 1.0
  /* Fill in the vectors best, words via dynamic programming */
  for $i = 0$ to $n$ do
    for $j = 0$ to $i - 1$ do
      word $\leftarrow$ text[j:i]
      w $\leftarrow$ LENGTH(word)
      if P[word] x best[i- w] $\geq$ best[i] then
        best[i] $\leftarrow$ P[word] $\times$ best[i - w]
        words[i] $\leftarrow$ word
  /* Now recover the sequence of best words */
  sequence $\leftarrow$ the empty list
  $i \leftarrow n$
  while $i > 0$ do
    push words[i] onto front of sequence
    $i \leftarrow i -$ LENGTH(words[i])
  /* Return sequence of best words and overall probability of sequence */
  return sequence, best[i]
```

Figure : A Viterbi-based word segmentation algorithms

Probabilistic context-free grammars (PCFG)

n-grams models take advantage of co-occurance statistics in the corpra, but they have no notion of grammar at distances greater than n. Probabilistic context grammar(PCFG) consists of a CFG where each rule has an associated probability.

The sum of probability of all rules is 1.

Learning probability fro PCFG

To create a PCFG, we have to combine probability fro each CFG rule.

This suggests that learning the grammar from data might be better than knowledge engineering approach.

Two types of data are given:

1. Parsed
2. Unparsed

The E step estimates the probabilities that each subsequence is generated by each rule. The M step then estimates the probability of each rule. It is done by Inside-Outside algorithm.

Inside-Outside Algorithm:

It induces grammar from unparsed tree.

It is slow.

Learning rule structure fro PCFG

If the structure of the grammar rules is not known, then make use of **Chomsky Normal Form (CNF)**.

$X \rightarrow Yz$

$X \rightarrow t$

Where X, Y, Z is non terminals and t is a terminal.

5.11 Information Retrieval (IR)

IR is a task of finding documents that are relevant to user’s need for information.

E.g. Google, Yahoo etc.

An IR is characterized as

1. A document collection

2. Query in a query language
3. A result set
4. A presentation of the result set.

The Earliest IR systems worked on a Boolean Keyword model. Each word in this document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not.

Evaluating IR model

There are two measures to evaluate whether the IR system is performing well or not. They are:

1. Recall
2. Precision

Recall: is the proportion of all the relevant documents in the collection that are in the result.

Precision: is the proportion of documents in the result set that are actually relevant.

Other measures to evaluate IR

The other measures are

1. Reciprocal rank.
2. Time to answer

IR refinements

The unigram model treats all words as completely independent, but we know that some words are correlated or co-related

E.g. the word “Couch” has two closely related words

Couches

Sofa

So IR systems do refinements fro these types of word by the following methods.

1. Case folding
2. Streaming
3. Recognize synonyms
4. Meta data

Presentation of result sets

There are three mechanisms to achieve performance improvement. They are

1. Relevance feedback
2. Document classification
3. Document clustering
    - Agglomerative clustering
    - K-means clustering

Implementing IR systems

IR systems are made efficient by two data structures. They are

1. Lexicon:
    - It lists all the words in the document collection.
    - It supports one operation
    - It is implemented by hash table.
2. Inverted index
    - It is similar to the index at back side of a book. It consists of a set of hit lists, which is the place where the word occurs.
    - In unigram model, it is a list of pairs.

5.12 Information extraction (IE)

IE is the process of creating database entries by a skimming text and looking for occurrences of a particular class of object or event and for relationships among those objects and events.

E.g. To extract addresses from web pages with database fields as street, city, state and pin code.

Types of IE system

1. **Attribute based system**
   This system assumes that the entire text as a single object and try to extract attributes of that object.

   If regular expression matches the text exactly once, then that portion of text is removed. It is the value of the attribute.

   If there is no match, nothing is done.

2. **Relational based system**
   It considers more than one object and the relations between them.

   This system is built by using cascaded finite state transducers, ie it consists of a series of Finite State Automata (FSA).

   An example of this system is FASTUS.

   It consists of five stages. They are

   i. Tokenization
   ii. Complex word handling
   iii. Basic groups
   iv. Complex phrases
   v. Merger structures

5.13 Machine translation

It is the automatic translation of text from one natural language (source) to another (target).

Types of translation

1. Rough translation
2. Restricted source translation
3. Pre-edited translation
4. Literary translation

Machine translation system

If translation is to be done fluently and perfectly then the translator (human or machine)

must read the original text, understood the situation to which it is referred and finds a good corresponding text in the target language describing same or similar situation.

These systems vary in the level to which they analyze the text.

Statistical machine translation

It is a new approach proposed during last decade. Here the whole translation process is based on finding the most probable translation of a sentence, using data gathered from s bilingual process.

The model for p (F/ E) has four set of parameters.

1. Language model
2. Translation model
3. Fertility model
4. Word choice model
5. Offset model

Learning probabilities for machine translation

- Segment into sentence
- Estimate the French language model
- Align sentences
- Estimate the initial fertility model
- Estimate the initial choice model
- Estimate the initial offset model
- Improve estimates

Question Bank

UNIT –V

PART –A

1.What is meant by communication?
2. Define utterance.
3. Define parsing.
4. List the types of grammar with its rules.
5. Give example for open classes and closed classes.
6. Differentiate bottom up parsing with top-down parsing.
7. What is meant by chart parsers?
8. Give example for chart parsing systems.
9. What is meant by define clause grammar.
10. Give example for augmented grammar.
11. List the types of ambiguity with example.
12. What is meant by probabilistic language model?
13. Define smoothing? What are the types of smoothing?
14. Differentiate information retrieval with information extraction.
15. List the stages of FASTOS.
16. What is meant by language model and translation model.
17. Write the steps for K means clustering.
18. Differentiate document classification with document clustering.
19. What is meant by stemming.
20. Define interlingua.

PART-B

1. Explain the seven processes involved in communication with an example.
2. Explain the two types of parsing with suitable example.
3. Explain the chart parsing algorithm with suitable example.
4. How semantic interpretation is done for a sentence? Explain with an example.
5. Write short notes on:
   (a) Discourse understanding.
   (b) Grammar induction
6. Explain the segmentation algorithm in detail.
7. Explain the steps to build an IR system.
8. List the different methods to do machine translation. Explain in detail about the statistical machine translation.
9. How IE is done using FASTOS system, explain the stages of it.

Previous Year Anna University Questions

B.E DEGREE EXAMINATION, APRIL /MAY 2008
SIXTH SEMESTER
CS 1351 –ARTIFICIAL INTELLIGENCE

Part –a

1. Define artificial intelligence
2. What is the use of heuristic functions
3. How to improve the effectiveness of a search based problem solving technique
4. What is constraint satisfaction problem
5. What is unification algorithm
6. How can you represent the resolution in predicate logic
7. List out the advantages of non monotonic reasoning
8. Differentiate between JTMS and LTMS
9. What are framesets and instances
10. List out the important components of a script

Part –b

11. A) i) give an example of a problem for which breadth first search would work better than depth first search.
ii) explain the algorithm for steepest hill climbing

Or

B) explain the following search strategies
i) best first search
ii) A* search

12. A) explain min-max search procedure.

Or

B) describe alpha-beta pruning and give the other modifications to the minmax procedure to improve its performance.

13. A) illustrate the use of predicate logic to represent the knowledge with suitable example.

Or

B) consider the following sentences
- John likes all kinds of food
- Apples are food
- Chicken is food
- Anything anyone eats and isn't killed by is food
- Bill eats peanuts and is still alive
- Sue eats everything bill eats

i) translate these sentences into formulas in predicate logic
ii) prove that john likes peanuts using backward chaining
iii) convert the formulas of a part into clause form

iv) prove that john likes peanuts using resolution

14. A) i) with an example the logics for non monotonic reasoning.

or

B) explain how Bayesian statistics provides reasoning under various kind s of uncertainty.

15)A) i) construct semantic net representations for the following .

- pomepeian (marcus), blacksmith(marcus)
- mary gave the green flowered vase to her favorite cousin.

ii)construct partitioned semantic net representations for the following :

- every batter hit a ball
- all the batters like the pitcher

Or

B) illustrate the learning from examples by induction with suitable example.

B.E./B.Tech, DEGREE EXAMINATION, NOVEMBER/DECEMBER 2003.

Fourth Semester

Computer Science and Engineering

CS 240 — ARTIFICIAL INTELLIGENCE

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A- (10 x 2 = 20 marks)

1. When is a class of problems said to be intractable?

2. Describe any two search strategies.

3. What is meant by syntax and semantics?

4. Give an example of multiple inheritance.

5. Define semantic networks.

6. When do you say two objects match?

7. State the various axioms of probability.

8. What is a Truth maintenance system?

9. Define conditional planning.

10. Give the general model of learning agents.

PART B — (5 x 16 = 80 marks)

11. (i) Explain in detail the History of Artificial Intelligence. (6)

(ii) Explain heuristic search with an example. (10)

12. (a) (i) Describe the different levels of knowledge used in language understanding.

(ii) Write the algorithm for depth first search and breadth first search techniques.

Or

(b) Write the minimax algorithm and how it works for the game of tic-tac-toe.

13. (a) Write the forward chaining and backward chaining algorithm and explain their use by taking simple examples.

Or

(b) Write the unification algorithm and explain its working by taking a suitable example.

14, (a) Represent the following sentences in predicate calculus and semantic networks.

John gave Mary a book.

John is a Programmer.

Mary is a Lawyer.

Mary's address is 37 Mylapore.

Or

(b) Explain the construction of frames taking a suitable example. Show the use of inheritance in frames.

15. (a) What is uncertainty? Explain the methods available for handling uncertain knowledge.

Or

(b) (i) Compare conditional planning and replanning.

(ii) Explain any two applications of Neural networks.

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2004.

Fourth Semester

Computer Science and Engineering

CS 240 - ARTIFICIAL INTELLIGENCE

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 x 2 = 20 marks)

1. What is AI?

2. What is the role of an agent program?

3. What is meant by epistemological commitment?

4. Compare forward chaining and backward chaining.

5. What does ‘description logics mean with reference to knowledge representation?

6. When does one use ‘event calculus"?

7. What is the syntax for default rules?

8. What is Markov’s assumption?

9. What are the steps of planning problems using state space search methodology?

10. What are the operations in Genetic algorithms?

PART B — (5 x 16 = 80 marks)

11. i) How is knowledge represented in Artificial neural networks and Genetic Algorithms?

(ii) How is learning made continuous in these two schemes?

(iii) When is learning complete in these architectures?

12. (a) (i) What are the various domains of Al? (8)

· (ii) What are the requirements of intelligent agents? (8)

Or

(b) (i) How does hill climbing ensure greedy local search? What are the problems of hill climbing? (10)

(ii) Describe in brief the depth first search. (6)

13. (a) (i) Explain unification with suitable illustration. (8)

(ii) Discuss a backward chaining algorithm and its strength. (8)

Or

(b) (i) Explain any two strategies used to resolve conflicts. (8) `

(ii) List out the conditions that are to be met tr. aptly forward chaining. What are the steps involved in forward chaining? (8)

14. (a) (i) Design a truth maintenance system. (10)

(ii) Do knowledge and belief vary over time? Is Action is time bound. Comment on these issues (6)

Or

(b) How are Baye's rule used to combine evidence in simple case? (16)

15. (a) How do you solve representational and inferential frame problem? (16)

Or

(b) (i) what are semantic networks and how do they perform inheritance? Give a detailed description on their usage. (10)

(ii) How do you evaluate a decision network? (6)

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2005.

Fourth Semester

Computer Science and Engineering

CS240 ARTIFICIAL INTELLIGENCE

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 x 2 = 20 marks)

1. Why do human beings want machines to perform very similar to them?

2. Name two areas (at least) where machines cannot excel human beings?

3. What are the requirements to develop knowledge base in first order logic?

4. What is meant by robotic agent?

5. What is a cognitive model J

6. What is meant by rule based training?

7. What is monatomic reasoning?

8, How is decision making possible in machines?

9. What are the learning methodologies suitable for machines?

10. What is aggregation?

PART B — (5 x 16 = 80 marks)

11. What is a Real World Problem? How to formulate a concise problem out of it for solving a RWP. How is the performance of an algorithm measured? Name the 5 search strategies of Blind search. (16)

12. (a) John flew to New York.

John flew a kite.

John flew down the street

John flew into the rage.

(i) Explain with reference to the above four sentences, how do you find the right structures as needed. (6)

(ii) What are issues to be addressed in knowledge representation? (5)

(iii) How do frame problems reason out? (5)

Or `

(b) (i) What is a semantic net? (2)

(ii) John gave a book on AI to Mary.

Represent the above sentence using a semantic net. (6)

(iii) What is a partitioned semantic net? (3)

(iv) Discuss on rule chaining. (5)

13. (a) (i) What are steps involved in representing knowledge using first order 4 logic. (8)

(ii) What do you understand by symbols, interpretations and quantifiers? (8)

Or

(b) Discuss unification and reduction to propositional inference.

14. (a) What is uncertainty? How to reason out in such situations? What are the various strategies under such cases? (16)

Or

(b) Discuss Truth Maintenance System based on Non monotonic reasoning. How do J TMS and LTMS differ? (16)

15. (a) (i) Is planning search totally ordered? If not, how is planning proceeded? How do Heuristic approach helpful in those situations?
(ii) How is conditional planning work? How does a replanning agent perform when encountering something unexpected?

Or

(b)Genetic learning and neural net learning are compared to other learning methodologies. Discuss how do genetic and Neural net algorithms adapt to learn and act.

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2006.

Fourth Semester

Computer Science and Engineering

CS 240 — ARTIFICIAL INTELLIGENCE

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 x 2 = 20 marks)

1. List the advantages of informed search strategies.

2. Define probabilistic language model.

3. Explain Modus Ponens and And-Elimination inference rules.

4. Write the semantics of the universal and existential quantifiers.

5. Define ontological engineering.

6. Describe default logic.

7. Define Baye`s rule.

8. What is me art by Decision networks?

9. Define Explanation-based learning.

10. List a few applications of Neural networks.

PART B — (5 x16 = 80 marks)

11. (i) Explain conditional planning. How is it implemented in fully observable and partially observable environments?

(ii) Describe least commitment search. Explain version—Space learning algorithm.

12. (a) (i) Explain the structure of agents with a neat diagram. (8)

(ii) Write any one of the word-segmentation algorithms. Explain the algorithm with an example sentence. (8)

Or

(b) (i) What is meant by Heuristic search? Explain. (8)

(ii) Write A* algorithm. Explain the processing of A* algorithm with an example search tree with Heuristic values. (8)

13. (a) (i) What is meant by knowledge Representation? (2)

(ii) With example sentences, explain how knowledge can be represented using Predicate logic. (8)

(iii) Explain Resolution procedure. (6)

Or

(b) (i) Explain forward-chaining and backward-chaining algorithms. (8)

(ii) Explain the steps in knowledge engineering process. (8)

14. (a) (i) How will you describe actions in ituation calculus? Explain. (8)

(ii) What is meant by weak slot and filler structures? Explain semantic nets in detail. (8)

Or

(b) (i) What is cognitive modeling? (4)

(ii) Write short notes on common sense ontologies. (4)

(iii) Write short notes on Frames. (B)

15. (a) (i) Explain the axioms of probability. (4)

(ii) Explain the desirable properties of rule-based methods for uncertain reasoning. (6)

(iii) Define Dempster-Shafer theory. (6)

Or

(b) (i) Explain the axioms of utility theory. (8)

(ii) Explain non-monotonic reasoning. (8)