

UNIT- I

Introduction: AI problems, Agents and Environments, Structure of Agents, Problem Solving Agents **Basic Search Strategies:** Problem Spaces, Uninformed Search (Breadth First, Depth-First Search, Depth-first with Iterative Deepening), Heuristic Search (Hill Climbing, Generic Best-First, A*), Constraint Satisfaction (Backtracking, Local Search)

Introduction:

- Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by John McCarthy in 1956.
- Intelligence is the ability to acquire, understand and apply the knowledge to achieve goals in the world.
- AI is the study of the mental faculties through the use of computational models
- AI is the study of intellectual/mental processes as computational processes.
- AI program will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task.
- AI is unique, sharing borders with Mathematics, Computer Science, Philosophy, Psychology, Biology, Cognitive Science and many others.
- Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, able to learn and use knowledge to solve problems on their own.

Sub Areas of AI:

1) Game Playing

Deep Blue Chess program beat world champion Gary Kasparov

2) Speech Recognition

PEGASUS spoken language interface to American Airlines' EAASY SABRE reservation system, which allows users to obtain flight information and make reservations over the

telephone. The 1990s has seen significant advances in speech recognition so that limited systems are now successful.

3) Computer Vision

Face recognition programs in use by banks, government, etc. The ALVINN system from CMU autonomously drove a van from Washington, D.C. to San Diego (all but 52 of 2,849 miles), averaging 63 mph day and night, and in all weather conditions. Handwriting recognition, electronics and manufacturing inspection, photo interpretation, baggage inspection, reverse engineering to automatically construct a 3D geometric model.

4) Expert Systems

Application-specific systems that rely on obtaining the knowledge of human experts in an area and programming that knowledge into a system.

- a. **Diagnostic Systems:** MYCIN system for diagnosing bacterial infections of the blood and suggesting treatments. Intellipath pathology diagnosis system (AMA approved). Pathfinder medical diagnosis system, which suggests tests and makes diagnoses. Whirlpool customer assistance center.
- b. **System Configuration**
DEC's XCON system for custom hardware configuration. Radiotherapy treatment planning.
- c. **Financial Decision Making**
Credit card companies, mortgage companies, banks, and the U.S. government employ AI systems to detect fraud and expedite financial transactions. For example, AMEX credit check.
- d. **Classification Systems**
Put information into one of a fixed set of categories using several sources of information. E.g., financial decision making systems. NASA developed a system for classifying very faint areas in astronomical images into either stars or galaxies with very high accuracy by learning from human experts' classifications.

5) Mathematical Theorem Proving

Use inference methods to prove new theorems.

6) Natural Language Understanding

AltaVista's translation of web pages. Translation of Caterpillar Truck manuals into 20 languages.

7) Scheduling and Planning

Automatic scheduling for manufacturing. DARPA's DART system used in Desert Storm and Desert Shield operations to plan logistics of people and supplies. American Airlines rerouting contingency planner. European space agency planning and scheduling of spacecraft assembly, integration and verification.

8) Artificial Neural Networks:

9) Machine Learning

Applications of AI:

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless, for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches

1. Business; financial strategies
2. Engineering: check design, offer suggestions to create new product, expert systems for all engineering problems
3. Manufacturing: assembly, inspection and maintenance
4. Medicine: monitoring, diagnosing
5. Education: in teaching
6. Fraud detection
7. Object identification
8. Information retrieval
9. Space shuttle scheduling

Building AI Systems:

1) Perception

Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an autonomous vehicle, input might be images from a camera and range information from a rangefinder. For a medical diagnosis system, perception is the set of symptoms and test results that have been obtained and input to the system manually.

2) Reasoning

Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc.

Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.

3) Action

Biological systems interact within their environment by actuation, speech, etc. All behavior is centered around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

The definitions of AI:

<p>a) "The exciting new effort to make computers think . . . <i>machines with minds</i>, in the full and literal sense" (Haugeland, 1985)</p> <p>"The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning..."(Bellman, 1978)</p>	<p>b) "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)</p> <p>"The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)</p>
<p>c) "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)</p> <p>"The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)</p>	<p>d) "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990)</p> <p>"The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)</p>

The definitions on the top, (a) and (b) are concerned with **reasoning**, whereas those on the bottom, (c) and (d) address **behavior**. The definitions on the left, (a) and (c) measure success in terms of human performance, and those on the right, (b) and (d) measure the ideal concept of intelligence called rationality

Intelligent Systems:

In order to design intelligent systems, it is important to categorize them into four categories (Luger and Stubberfield 1993), (Russell and Norvig, 2003)

1. Systems that think like humans
2. Systems that think rationally
3. Systems that behave like humans
4. Systems that behave rationally

	Human-Like	Rationality
Think:	Cognitive Science Approach <i>“Machines that think like humans”</i>	Laws of thought Approach <i>“Machines that think Rationally”</i>
Act:	Turing Test Approach <i>“Machines that behave like humans”</i>	Rational Agent Approach <i>“Machines that behave Rationally”</i>

Cognitive Science: Think Human-Like

- a. Requires a model for human cognition. Precise enough models allow simulation by computers.
- b. Focus is not just on behavior and I/O, but looks like reasoning process.
- c. Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

Laws of thought: Think Rationally

- a. The study of mental faculties through the use of computational models; that is, the study of computations that make it possible to perceive reason and act.

- b. Focus is on inference mechanisms that are probably correct and guarantee an optimal solution.
- c. Goal is to formalize the reasoning process as a system of logical rules and procedures of inference.
- d. Develop systems of representation to allow inferences to be like

“Socrates is a man. All men are mortal. Therefore Socrates is mortal”

Turing Test: Act Human-Like

- a. The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which, at the moment, people do better.
- b. Focus is on action, and not intelligent behavior centered around the representation of the world
- c. Example: Turing Test
 - 3 rooms contain: a person, a computer and an interrogator.
 - The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance of voice of the person)
 - The interrogator tries to determine which the person is and which the machine is.
 - The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
 - If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

Rational agent: Act Rationally

- a. Tries to explain and emulate intelligent behavior in terms of computational process; that it is concerned with the automation of the intelligence.
- b. Focus is on systems that act sufficiently if not optimally in all situations.
- c. Goal is to develop systems that are rational and sufficient

Agents and Environments:

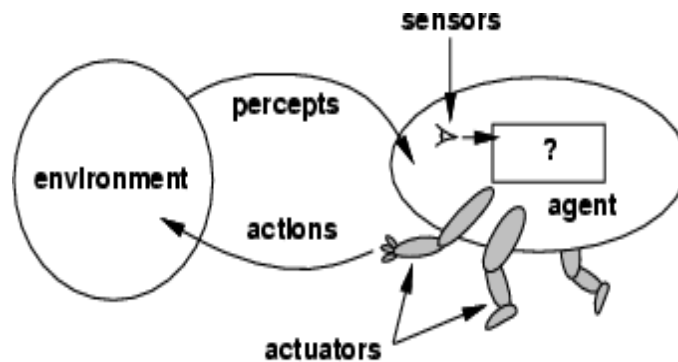


Fig 2.1: Agents and Environments

Agent:

An *Agent* is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- ✓ A *human agent* has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- ✓ A *robotic agent* might have cameras and infrared range finders for sensors and various motors for actuators.
- ✓ A *software agent* receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Percept:

We use the term percept to refer to the agent's perceptual inputs at any given instant.

Percept Sequence:

An agent's percept sequence is the complete history of everything the agent has ever perceived.

Agent function:

Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any **given percept sequence to an action**.

Agent program

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract

mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in **Fig 2.1.5**. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in **Fig 2.1.6**.

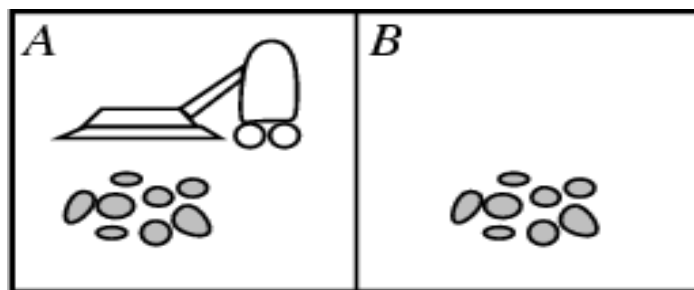


Fig 2.1.5: A vacuum-cleaner world with just two locations.

Agent function

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

Fig 2.1.6: Partial tabulation of a simple agent function for the example: vacuum-cleaner world shown in the Fig 2.1.5

```
Function REFLEX-VACCUM-AGENT ([location, status]) returns an  
action  
If status=Dirty then return Suck  
  
else if location = A then return Right  
  
else if location = B then return Left
```

Fig 2.1.6(i): The REFLEX-VACCUM-AGENT program is invoked for each new percept (location, status) and returns **an** action each time

- A **Rational agent** is one that does the right thing. we say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding how and when to evaluate the agent's success.
We use the term performance measure for the how—the criteria that determine how successful an agent is.
 - ✓ Ex-Agent cleaning the dirty floor
 - ✓ Performance Measure-Amount of dirt collected
 - ✓ When to measure-Weekly for better results

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

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

Omniscience ,Learning and Autonomy:

- We need to distinguish between rationality and omniscience. An **Omniscient agent** knows the actual outcome of its actions and can act accordingly but omniscience is impossible in reality.
- Rational agent not only gathers information but also **learns** as much as possible from what it perceives.
- If an agent just relies on the prior knowledge of its designer rather than its own percepts then the agent lacks **autonomy**.
- A system is autonomous to the extent that its behavior is determined its own experience.
- A rational agent should be autonomous.

E.g., a clock(lacks autonomy)

- No input (percepts)
- Run only but its own algorithm (prior knowledge)
- No learning, no experience, etc.

ENVIRONMENTS:

The Performance measure, the environment and the agents actuators and sensors comes under the heading task environment. We also call this as PEAS(Performance,Environment,Actuators,Sensors)

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

Other PEAS Examples

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	healthy patient, costs, lawsuits	patient, hospital, stuff	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	purity, yield, safety	refinery, operators	valves pumps, heaters displays	temperature, pressure, chemical sensors
Interactive English tutor	student's score on test	set of students, testing agency	display exercises, suggestions, corrections	keyboard entry

Environment-Types:

1. Accessible vs. inaccessible or Fully observable vs Partially Observable:

If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.

2. Deterministic vs. Stochastic:

If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic

3. Episodic vs. nonepisodic:


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

4. Static vs. dynamic.

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static.

5. Discrete vs. continuous:

If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Otherwise, it is continuous.



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

Figure 2.6 Examples of task environments and their characteristics.

STRUCTURE OF INTELLIGENT AGENTS

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

$$\text{agent} = \text{architecture} + \text{program}$$

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

Figure 2.3 Examples of agent types and their PAGE descriptions.

Agent programs:

- Intelligent agents accept percepts from an environment and generates actions. The early versions of agent programs will have a very simple form (Figure 2.4)

- Each will use some internal data structures that will be updated as new percepts arrive.
- These data structures are operated on by the agent's decision-making procedures to generate an action choice, which is then passed to the architecture to be executed

```

function TABLE-DRIVEN-AGENT(percept) returns action
  static: percepts, a sequence, initially empty
           table, a table, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action

```

Figure 2.5 An agent based on a prespecified lookup table. It keeps track of the percept sequence and just looks up the best action.

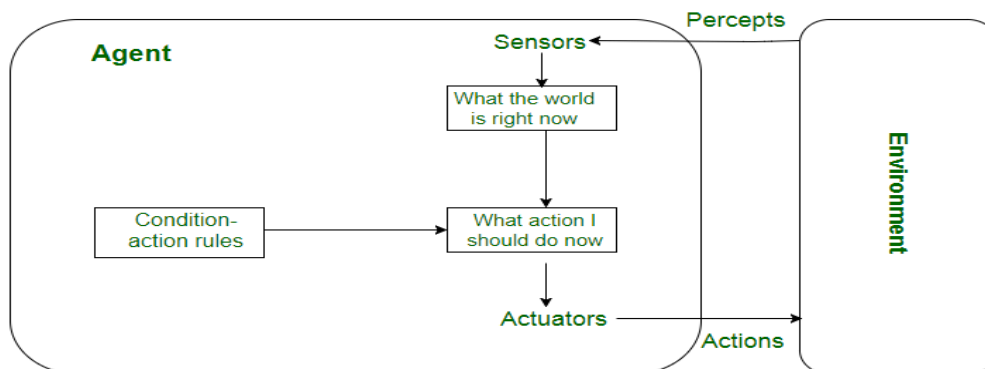
Types of agents:

Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents

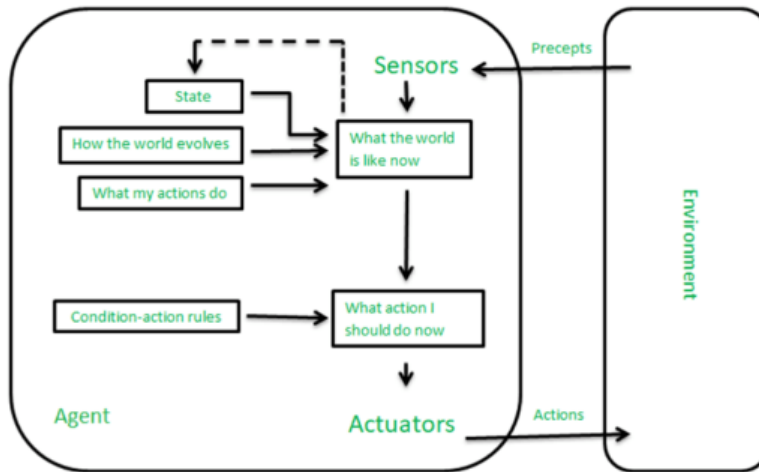
Simple reflex agents:

- Simple reflex agents ignore the rest of the percept history and act only on the basis of the current percept.
- The agent function is based on the condition-action rule.
- If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable.



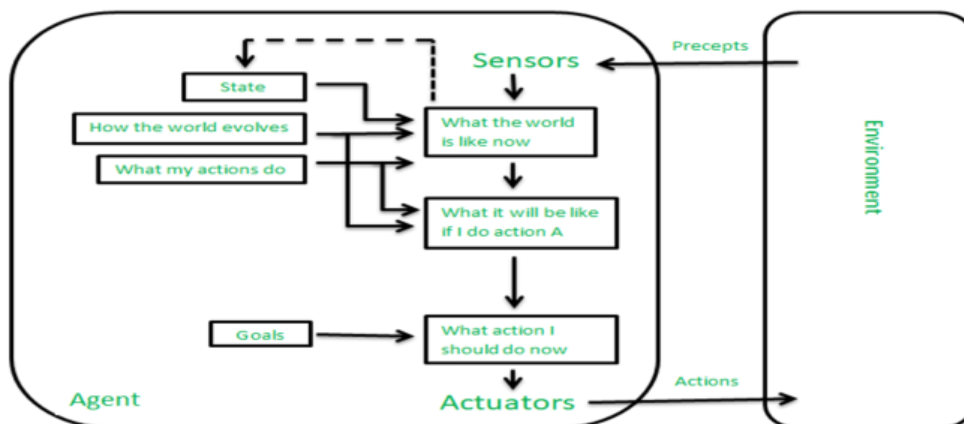
Model-based reflex agents:

- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
- Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
- Internal State: It is a representation of the current state based on percept history.



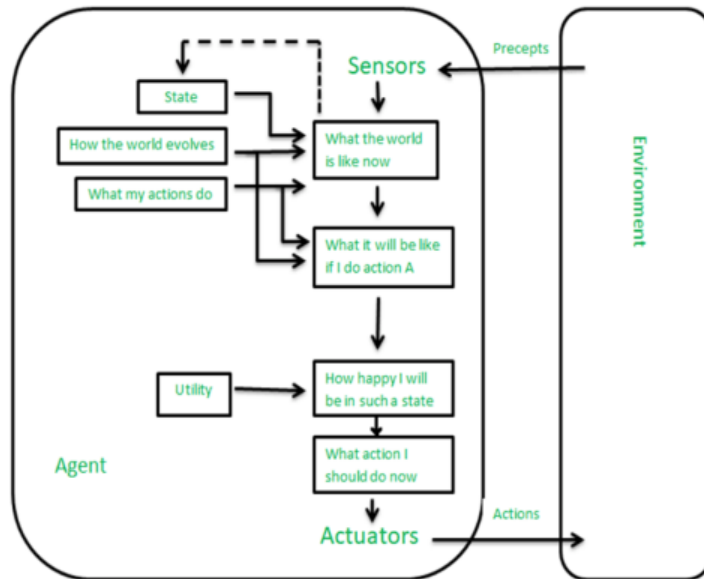
Goal-based agents:

- A goal-based agent has an agenda.
- It operates based on a goal in front of it and makes decisions based on how best to reach that goal.
- A goal-based agent operates as a search and planning function, meaning it targets the goal ahead and finds the right action in order to reach it.
- Expansion of model-based agent.



Utility-based agents:

- A utility-based agent is an agent that acts based not only on what the goal is, but the best way to reach that goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The term utility can be used to describe how "happy" the agent is.



Problem Solving Agents:

- Problem solving agent is a goal-based agent.
- Problem solving agents decide what to do by finding sequence of actions that lead to desirable states.

Goal Formulation:

It organizes the steps required to formulate/ prepare one goal out of multiple goals available.

Problem Formulation:

It is a process of deciding what actions and states to consider to follow goal formulation.

The process of looking for a best sequence to achieve a goal is called **Search**.

A search algorithm takes a problem as input and returns a solution in the form of action sequences.

Once the solution is found the action it recommends can be carried out. This is called **Execution phase**.

Well Defined problems and solutions:

A problem can be defined formally by 4 components:

- The **initial state** of the agent is the state where the agent starts in. In this case, the initial state can be

described as In: Arad

- The possible **actions** available to the agent, corresponding to each of the state the agent resides in.

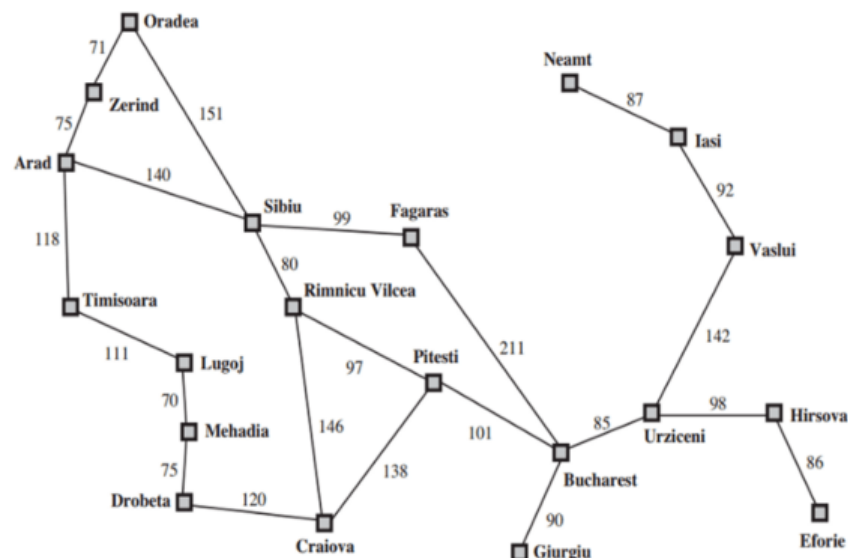
For example, $ACTIONS(In: Arad) = \{Go: Sibiu, Go: Timisoara, Go: Zerind\}$.

Actions are also known as operations.

- A description of what each action does. the formal name for this is **Transition model**, Specified by the function $Result(s,a)$ that returns the state that results from the action a in state s .

We also use the term Successor to refer to any state reachable from a given state by a single action.

For EX: $Result(In(Arad),GO(Zerind))=In(Zerind)$



Together the initial state, actions and transition model implicitly defines the **state space** of the problem

State space: set of all states reachable from the initial state by any sequence of actions

- **The goal test**, determining whether the current state is a goal state. Here, the goal state is {In: Bucharest}
- **The path cost function**, which determine the cost of each path, which is reflecting in the performance measure.

we define the cost function as $c(s, a, s')$, where s is the current state and a is the action performed by the agent to reach state s' .


```

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

  state ← UPDATE-STATE(state, p)
  if s is empty then
    g ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, g)
    s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action

```

Figure 3.1 A simple problem-solving agent.

Example –

8 puzzle problem

Initial State

2	8	3
1	6	4
7		5

Goal State

1	2	3
8		4
7	6	5

- **States:** a state description specifies the location of each of the eight tiles in one of the nine squares. For efficiency, it is useful to include the location of the blank.
- **Actions:** blank moves left, right, up, or down.
- **Transition Model:** Given a state and action, this returns the resulting state. For example if we apply left to the start state the resulting state has the 5 and the blank switched.
- **Goal test:** state matches the goal configuration shown in fig.
- **Path cost:** each step costs 1, so the path cost is just the length of the path.

State Space Search/Problem Space Search:

The state space representation forms the basis of most of the AI methods.

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

Formal Description of the problem:

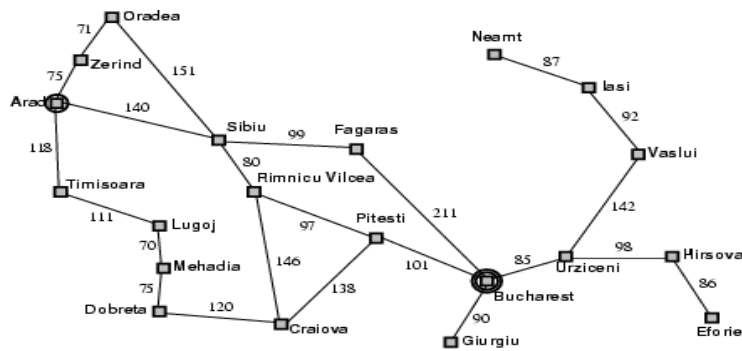
1. Define a state space that contains all the possible configurations of the relevant objects.
 2. Specify one or more states within that space that describe possible situations from which the problem solving process may start (**initial state**)
 3. Specify one or more states that would be acceptable as solutions to the problem. (**goal states**)
- Specify a set of rules that describe the actions (**operations**) available

State-Space Problem Formulation:

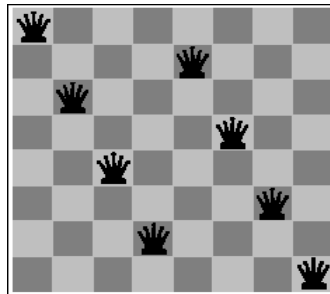
Example: A problem is defined by four items:

1. **initial state** e.g., "at Arad"
2. **actions or successor function** : $S(x)$ = set of action–state pairs
e.g., $S(Arad) = \{ \langle Arad \longrightarrow Zerind, Zerind \rangle, \dots \}$
3. **goal test (or set of goal states)**
e.g., $x = \text{"at Bucharest"}$, $Checkmate(x)$
4. **path cost (additive)**
e.g., sum of distances, number of actions executed, etc.
 $c(x,a,y)$ is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state



Example: 8-queens problem



1. **Initial State:** Any arrangement of 0 to 8 queens on board.
2. **Operators:** add a queen to any square.
3. **Goal Test:** 8 queens on board, none attacked.
4. **Path cost:** not applicable or Zero (because only the final state counts, search cost might be of interest).

Search strategies:

Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

Search Space: Search space represents a set of possible solutions, which a system may have.

Start State: It is a state from where agent begins the search.

Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.

Properties of Search Algorithms

Which search algorithm one should use will generally depend on the problem domain. There are four important factors to consider:

1. **Completeness** – Is a solution guaranteed to be found if at least one solution exists?
2. **Optimality** – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?
3. **Time Complexity** – The upper bound on the time required to find a solution, as a function of the complexity of the problem.
4. **Space Complexity** – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

State Spaces versus Search Trees:

- State Space
 - Set of valid states for a problem
 - Linked by operators
 - e.g., 20 valid states (cities) in the Romanian travel problem
- Search Tree
 - Root node = initial state
 - Child nodes = states that can be visited from parent
 - Note that the depth of the tree can be infinite
 - E.g., via repeated states
 - Partial search tree
 - Portion of tree that has been expanded so far
 - Fringe
 - Leaves of partial search tree, candidates for

expansion Search trees = data structure to search state-space

Searching

Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed; using heuristic functions. The algorithms that use heuristic functions are called heuristic algorithms. Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.

Heuristic algorithms are more efficient because they take advantage of feedback from the data to direct the search path.

Uninformed search

Also called blind, exhaustive or brute-force search, uses no information about the problem to guide the search and therefore may not be very efficient.

Informed Search:

Also called heuristic or intelligent search, uses information about the problem to guide the search, usually guesses the distance to a goal state and therefore efficient, but the search may not be always possible.

Uninformed Search (Blind searches):

1. Breadth First Search:

- One simple search strategy is a breadth-first search. In this strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on.
- In general, all the nodes at depth d in the search tree are expanded before the nodes at depth $d + 1$.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

BFS illustrated:

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

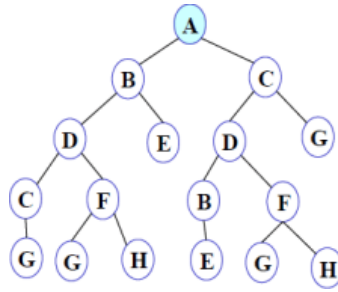


Figure 1

Frontier: A

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

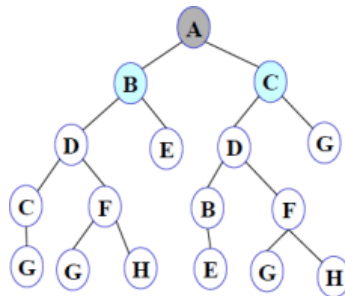


Figure 2

Frontier: B C

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

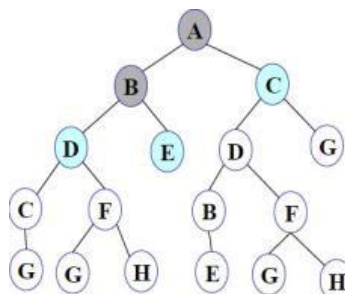


Figure 3

Frontier: C D E

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

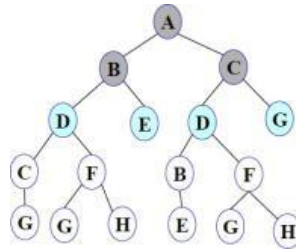


Figure 4

Frontier: D E D G

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

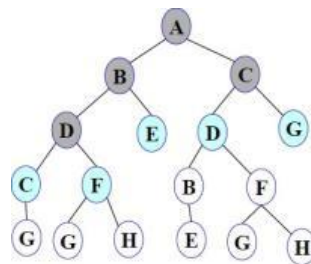


Figure 5

Frontier: E D G C F

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

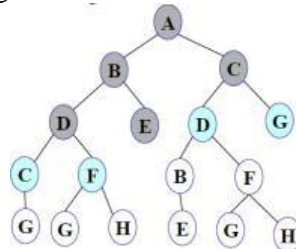


Figure 6

Frontier: D G C F

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

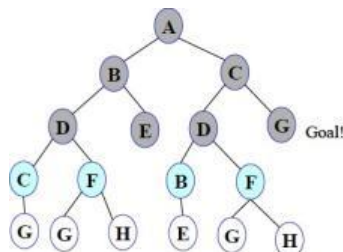


Figure 7

Frontier: G C F B F

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

Breadth first search is:

- One of the simplest search strategies
- Complete. If there is a solution, BFS is guaranteed to find it.
- If there are multiple solutions, then a minimal solution will be found
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- **Time complexity** : $O(b^d)$
- **Space complexity** : $O(b^d)$
- **Optimality** : Yes

b - branching factor(maximum no of successors of any node), d – Depth of the shallowest goal node

Maximum length of any path (m) in search space

Advantages:

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

Disadvantages:

- Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node.
- The breadth first search algorithm cannot be effectively used unless the search space is quite small.

Applications Of Breadth-First Search Algorithm

GPS Navigation systems: Breadth-First Search is one of the best algorithms used to find neighboring locations by using the GPS system.

Broadcasting: Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal

methods is Breadth-First Search. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

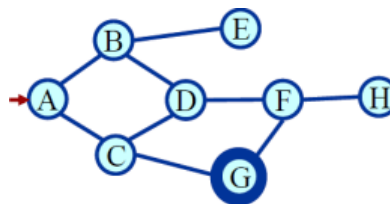
Depth- First- Search.

We may sometimes search the goal along the largest depth of the tree, and move up only when further traversal along the depth is not possible. We then attempt to find alternative offspring of the parent of the node (state) last visited. If we visit the nodes of a tree using the above principles to search the goal, the traversal made is called depth first traversal and consequently the search strategy is called *depth first search*.

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
else if *limit* = 0 **then return** cutoff
else
 cutoff_occurred? \leftarrow false
 for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
 child \leftarrow CHILD-NODE(*problem*, *node*, *action*)
 result \leftarrow RECURSIVE-DLS(*child*, *problem*, *limit* - 1)
 if *result* = cutoff **then** *cutoff_occurred?* \leftarrow true
 else if *result* \neq failure **then return** *result*
 if *cutoff_occurred?* **then return** cutoff **else return** failure

DFS illustrated:



A State Space Graph

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

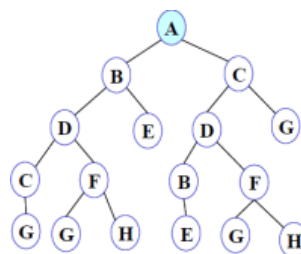


Figure 1

FRINGE: A

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

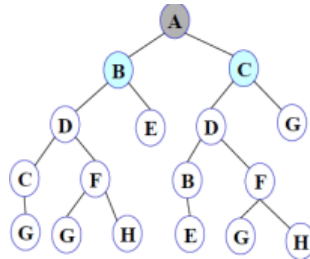


Figure 2

FRINGE: B C

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

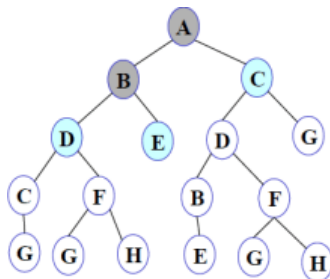


Figure 3

FRINGE: D E C

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

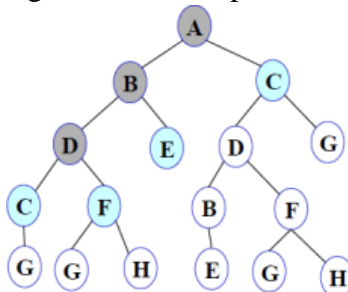


Figure 4

FRINGE: C F E C

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

Figure 5

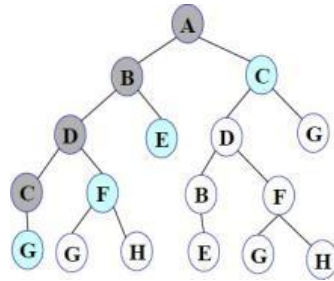


Figure 5

FRINGE: G F E C

Step 6: Node G is expanded and found to be a goal node.

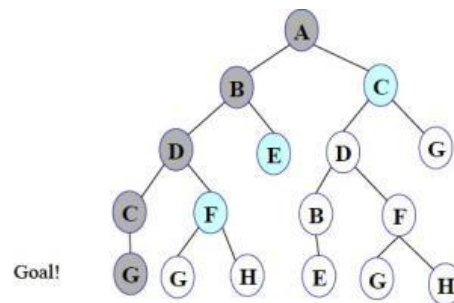


Figure 6

FRINGE: G F E C

The solution path A-B-D-C-G is returned and the algorithm terminates.

Depth first search

1. takes exponential time.
2. If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time $O(b^d)$.
3. The space taken is linear in the depth of the search tree, $O(bN)$.

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

Iterative Deeping DFS

- The iterative deepening algorithm is a combination of DFS and BFS algorithms.
- This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

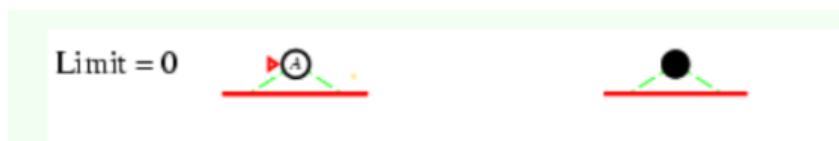
Advantages:

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

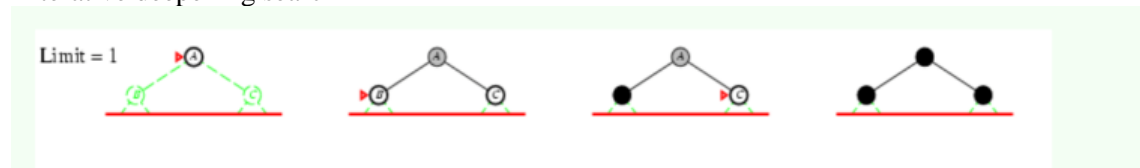
Disadvantages:

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

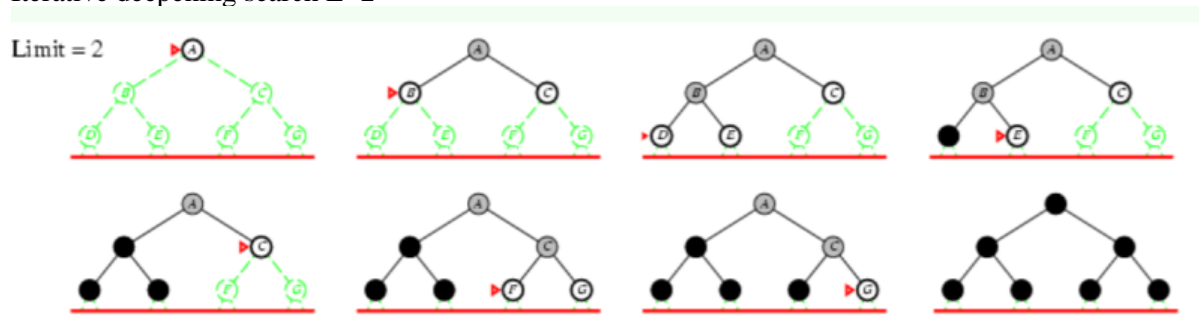
Iterative deepening search L=0



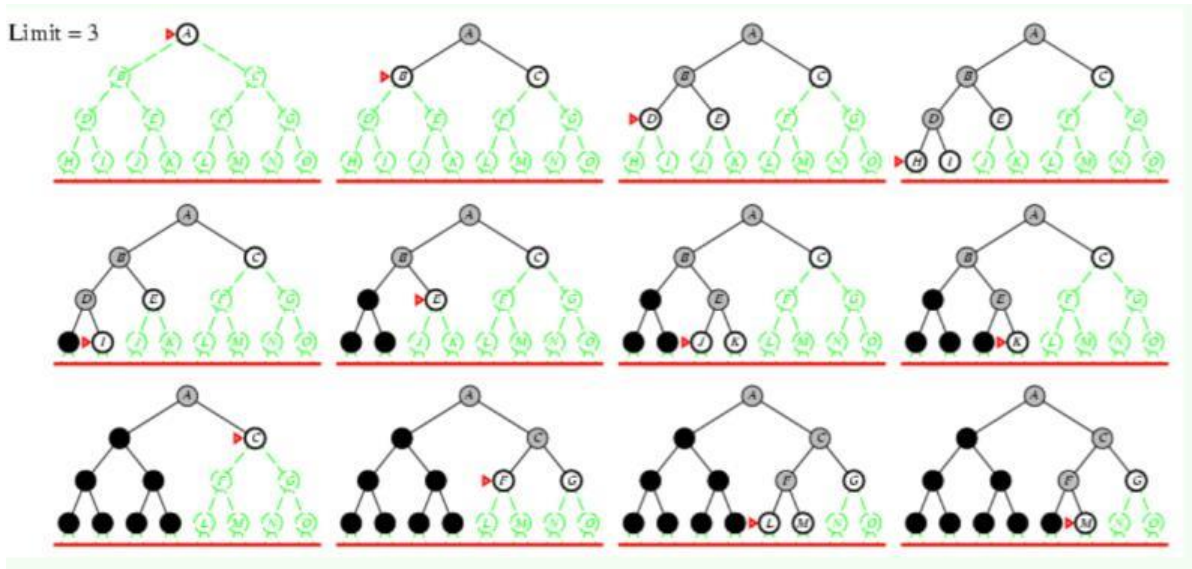
Iterative deepening search L=1



Iterative deepening search L=2



Iterative Deepening Search L=3



M is the goal node. So we stop there.

Complete: Yes

Time: $O(bd)$

Space: $O(bd)$

Optimal: Yes, if step cost = 1 or increasing function of depth.

Conclusion:

- ☐ We can conclude that IDS is a hybrid search strategy between BFS and DFS inheriting their advantages.
- ☐ IDS is faster than BFS and DFS.
- ☐ It is said that "IDS is the preferred uniform search method when there is a large search space and the depth of the solution is not known"

A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	$O(d)$	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	$O(bd)$	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

Informed search/Heuristic search

A *heuristic* is a method that

- might not always find the best solution **but** is guaranteed to find a good solution in reasonable time. By sacrificing completeness it increases efficiency.
- Useful in solving tough problems which
 - could not be solved any other way.
 - solutions take an infinite time or very long time to compute.

Calculating Heuristic Value:

- 1. Euclidian distance- used to calculate straight line distance.
- 2. Manhattan distance- If we want to calculate vertical or horizontal distance

For ex: 8 puzzle problem

Source state

1	3	2
6	5	4
	8	7

destination state

1	2	3
4	5	6
7	8	

Then the Manhattan distance would be sum of the no of moves required to move each number from source state to destination state.

Number in 8 puzzle	1	2	3	4	5	6	7	8
No. of moves to reach destination	0	2	1	2	0	2	2	0

3. No. of misplaced tiles for 8 puzzle problem

Source state

1	3	2
6	5	4
	8	7

Destination state

1	2	3
4	5	6
7	8	

Here just calculate the number of tiles that have to be changed to reach goal state

Here 1,5,8 need not be changed

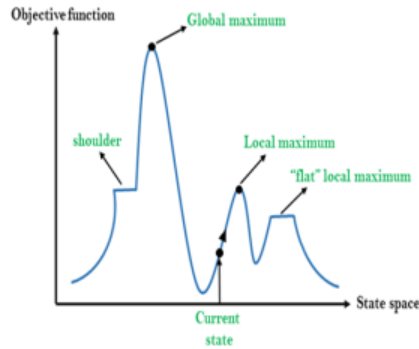
2,3,4,6,7 should be changed, so the heuristic value will be 5(because 5 tiles have to be changed)

Hill Climbing Algorithm

- ✓ Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- ✓ It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- ✓ Hill Climbing is mostly used when a good heuristic is available.
- ✓ In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

The idea behind hill climbing is as follows.

1. Pick a random point in the search space.
2. Consider all the neighbors of the current state.
3. Choose the neighbor with the best quality and move to that state.
4. Repeat 2 thru 4 until all the neighboring states are of lower quality.
5. Return the current state as the solution state.



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Algorithm for Hill Climbing

```

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

Problems in Hill Climbing Algorithm:

: Hill-climbing

This simple policy has three well-known drawbacks:

1. **Local Maxima:** a local maximum as opposed to global maximum.
2. **Plateaus:** An area of the search space where evaluation function is flat, thus requiring random walk.
3. **Ridge:** Where there are steep slopes and the search direction is not towards the top but towards the side.

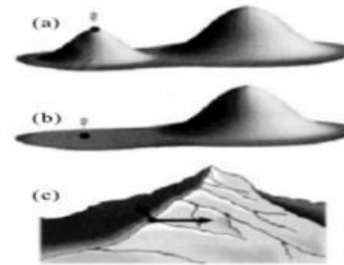
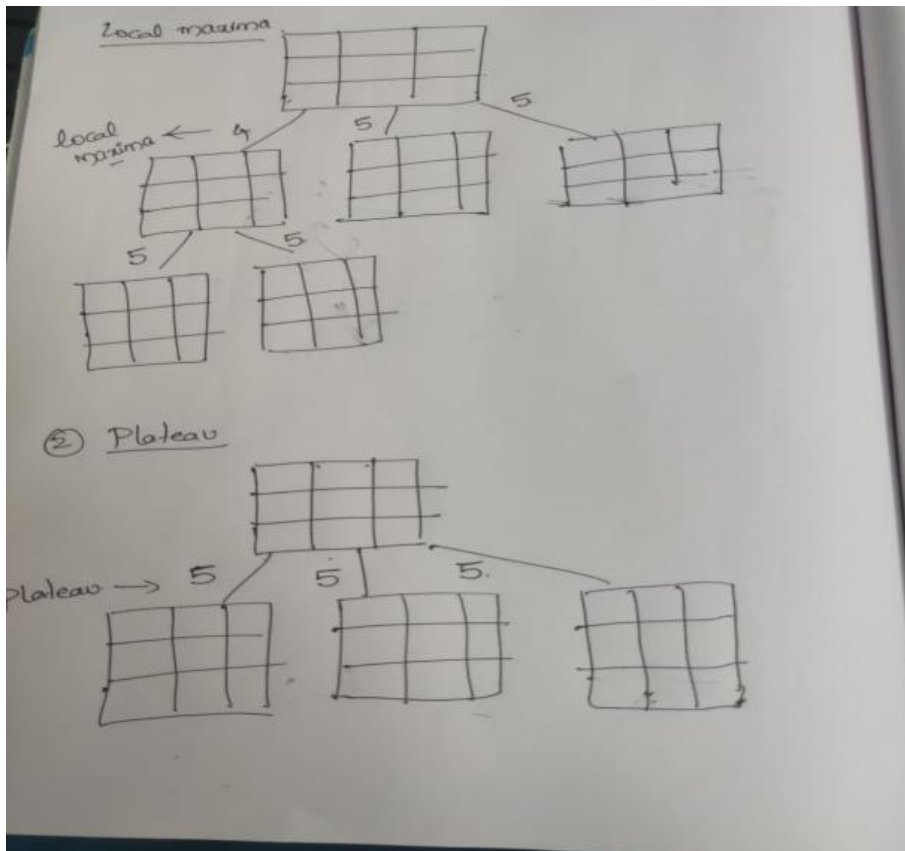


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing



Simulated annealing search

A hill-climbing algorithm that never makes “downhill” moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient. Simulated annealing is an

algorithm that combines hill-climbing with a random walk in some way that yields both efficiency and completeness.

simulated annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move – the amount ΔE by which the evaluation is worsened. The probability also decreases as the “temperature” T goes down: “bad moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems. It has been applied widely to factory scheduling and other large-scale optimization tasks.

Best First Search:

- A combination of depth first and breadth first searches.
- Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends.
- The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better than this previously unexpanded node and branch is not forgotten and the search method reverts to the

OPEN is a priority queue of nodes that have been evaluated by the heuristic function but which have not yet been expanded into successors. The most promising nodes are at the front.

CLOSED are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

Algorithm:

1. Start with OPEN holding the initial state

2. Until a goal is found or there are no nodes left on open do.

- Pick the best node on OPEN
- Generate its successors
- For each successor Do
 - If it has not been generated before ,evaluate it ,add it to OPEN and record its parent
 - If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes.

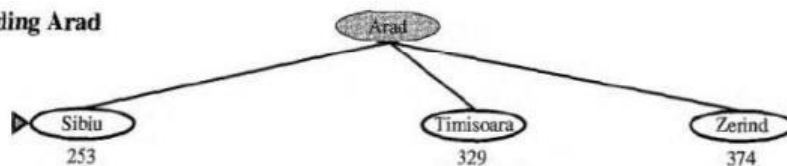
3. If a goal is found or no more nodes left in OPEN, quit, else return to 2.

Example:

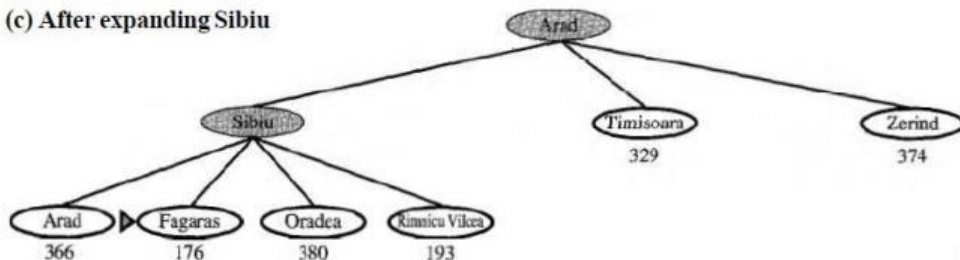
(a) The initial state



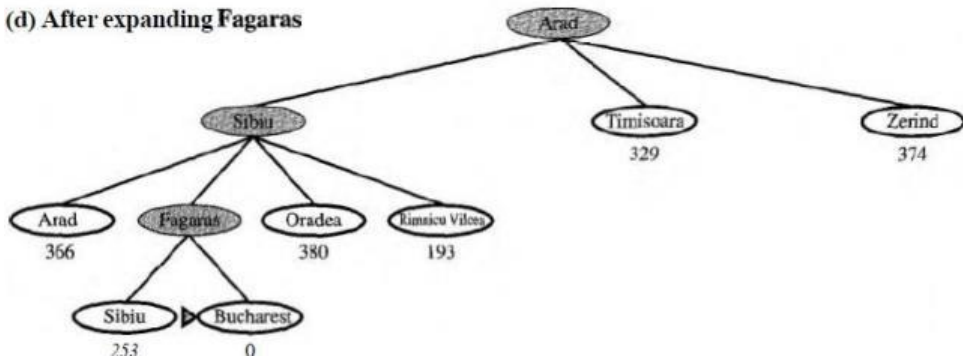
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



1. It is not optimal.

2. It is incomplete because it can start down an infinite path and never return to try other possibilities.
3. The worst-case time complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space.
4. Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity

A* Algorithm

The Best First algorithm is a simplified form of the A* algorithm.

The **A* search algorithm** (pronounced "Ay-star") is a tree search algorithm that finds a path from a given initial node to a given goal node (or one passing a given goal test). It employs a "heuristic estimate" which ranks each node by an estimate of the best route that goes through that node. It visits the nodes in order of this heuristic estimate.

Similar to greedy best-first search but is more accurate because A* takes into account the nodes that have already been traversed.

From A* we note that $f = g + h$ where

g is a measure of the distance/cost to go from the initial node to the current node

h is an estimate of the distance/cost to solution from the current node.

Thus f is an estimate of how long it takes to go from the initial node to the solution

Algorithm:

1. **Initialize** : Set OPEN = (S); CLOSED = ()
g(s) = 0, f(s) = h(s)
 2. **Fail** : If OPEN = (), Terminate and fail.
 3. **Select** : select the minimum cost state, n, from OPEN,
save n in CLOSED
 4. **Terminate** : If n ∈ G, Terminate with success and return f(n)
 5. **Expand** : for each successor, m, of n
-

- a) If $m \in [\text{OPEN} \cup \text{CLOSED}]$ Set $g(m) = g(n) + c(n, m)$ Set $f(m) = g(m) + h(m)$ Insert m in OPEN
- b) If $m \in [\text{OPEN} \cup \text{CLOSED}]$
- Set $g(m) = \min \{ g(m), g(n) + c(n, m) \}$ Set $f(m) = g(m) + h(m)$
 If $f(m)$ has decreased and $m \in \text{CLOSED}$
- Move m to OPEN.

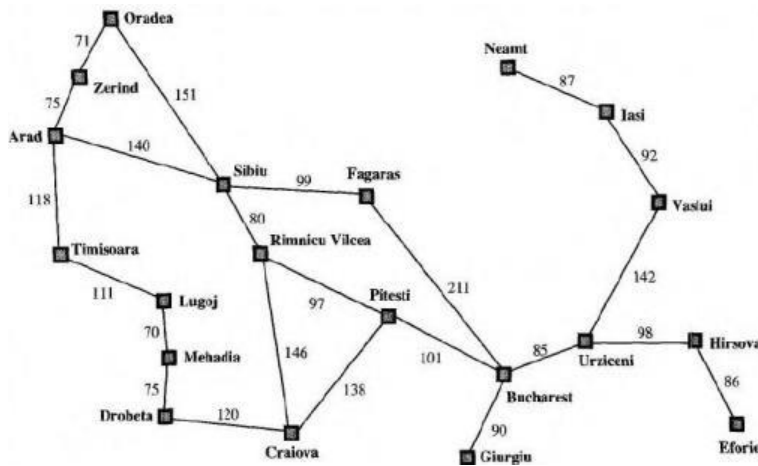
Description:

- A* begins at a selected node. Applied to this node is the "cost" of entering this node (usually zero for the initial node). A* then estimates the distance to the goal node from the current node. This estimate and the cost added together are the heuristic which is assigned to the path leading to this node. The node is then added to a priority queue, often called "open".
- The algorithm then removes the next node from the priority queue (because of the way a priority queue works, the node removed will have the lowest heuristic). If the queue is empty, there is no path from the initial node to the goal node and the algorithm stops. If the node is the goal node, A* constructs and outputs the successful path and stops.
- If the node is not the goal node, new nodes are created for all admissible adjoining nodes; the exact way of doing this depends on the problem at hand. For each successive node, A* calculates the "cost" of entering the node and saves it with the node. This cost is calculated from the cumulative sum of costs stored with its ancestors, plus the cost of the operation which reached this new node.
- The algorithm also maintains a 'closed' list of nodes whose adjoining nodes have been checked. If a newly generated node is already in this list with an equal or lower cost, no further processing is done on that node or with the path associated with it. If a node in the closed list matches the new one, but has been stored with a *higher* cost, it is removed from the closed list, and processing continues on the new node.

- Next, an estimate of the new node's distance to the goal is added to the cost to form the heuristic for that node. This is then added to the 'open' priority queue, unless an identical node is found there.
- Once the above three steps have been repeated for each new adjoining node, the original node taken from the priority queue is added to the 'closed' list. The next node is then popped from the priority queue and the process is repeated

The heuristic costs from each city to Bucharest:

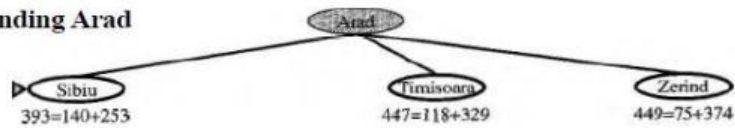
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



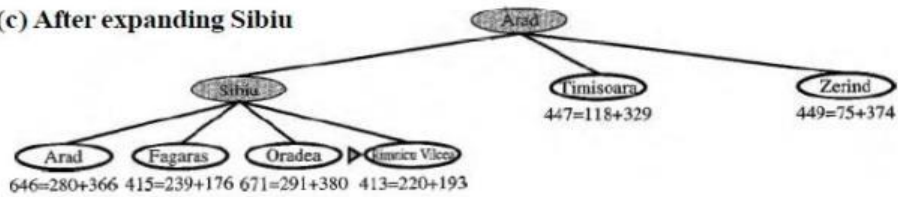
(a) The initial state

$$366=0+366$$

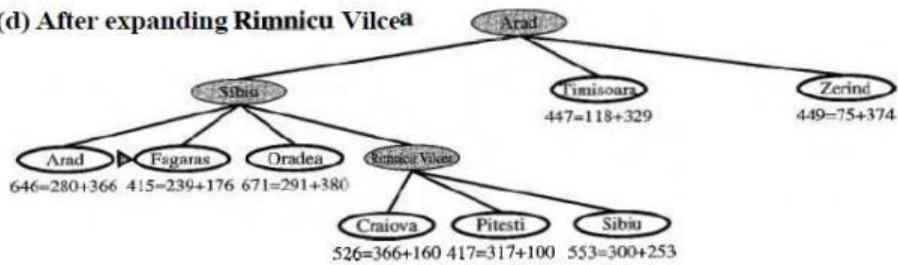
(b) After expanding Arad



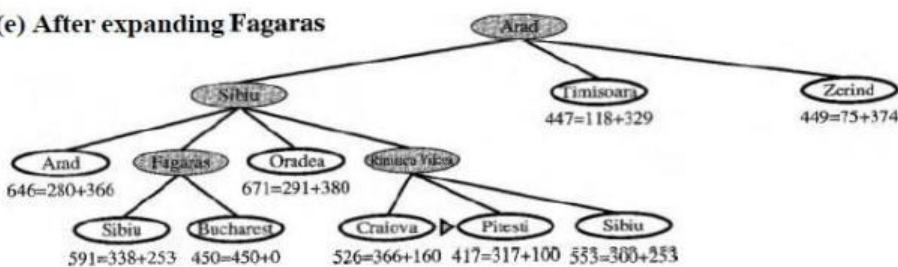
(c) After expanding Sibiu



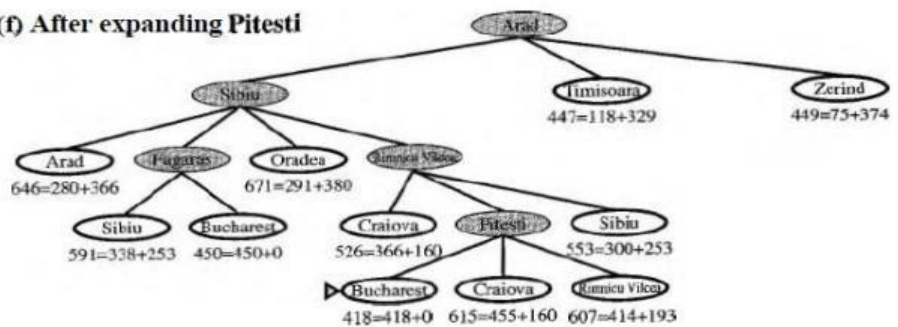
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



A* search properties:

- The algorithm A* is admissible. This means that provided a solution exists, the first solution found by A* is an optimal solution. A* is admissible under the following conditions:
 - Heuristic function: for every node n , $h(n) \leq h^*(n)$.
 - A* is also complete.
- A* is optimally efficient for a given heuristic.
- A* is much more efficient than uninformed search.

Constraint Satisfaction Problems

<https://www.cnblogs.com/RDaneelOlivaw/p/8072603.html>

Sometimes a problem is not embedded in a long set of action sequences but requires picking the best option from available choices. A good general-purpose problem solving technique is to list the constraints of a situation (either negative constraints, like limitations, or positive elements that you want in the final solution). Then pick the choice that satisfies most of the constraints.

Formally speaking, a **constraint satisfaction problem (or CSP)** is defined by a set of variables, $X_1; X_2; \dots$

$; X_n$, and a set of constraints, $C_1; C_2; \dots; C_m$. Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, $\{X_i = v_i; X_j = v_j; \dots\}$. An assignment that does not violate any constraints is called a consistent or

legal assignment. 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.

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

1. **Initial state:** the empty assignment ϕ , in which all variables are unassigned.
2. **Successor function:** a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
3. **Goal test:** the current assignment is complete.

4. **Path cost:** a constant cost for every step

Examples:

1. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a *convex* region.

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

2. **Crypt arithmetic** puzzles.

Example: The map coloring problem.

The task of coloring each region red, green or blue in such a way that no neighboring regions have the same color.

We are given the task of coloring each region red, green, or blue in such a way that the neighboring regions must not have the same color.

To formulate this as CSP, we define the variable to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set {red, green, blue}. The constraints require neighboring regions to have distinct colors: for example, the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}. (The constraint can also be represented as the inequality $WA \neq NT$). There are many possible solutions,

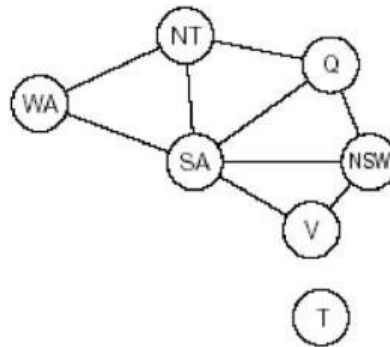


Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
 $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

such as {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}. Map of Australia showing each of its states and territories

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.

Constraint graph: nodes are variables, arcs show constraints



The map-coloring problem represented as a constraint

graph. CSP can be viewed as a standard search problem

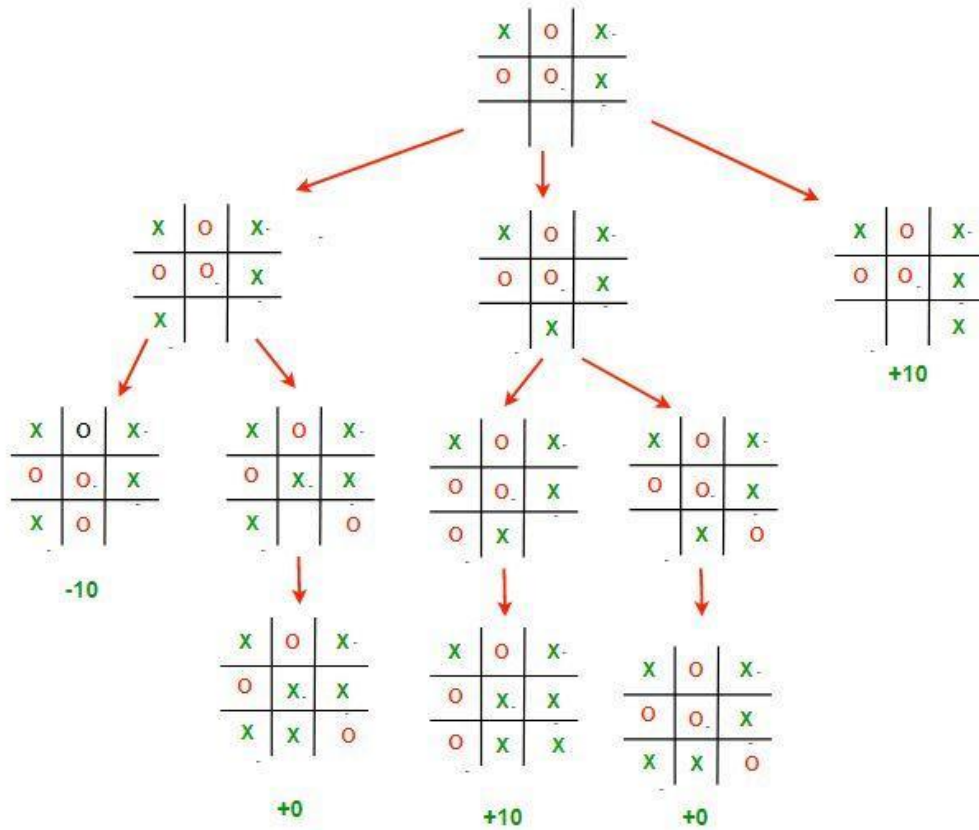
as follows:

- > **Initial state** : the empty assignment {}, in which all variables are unassigned.
- > **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- > **Goal test**: the current assignment is complete.
- > **Path cost**: a constant cost (E.g., 1) for every step.

UNIT II

Advanced Search: Constructing Search Trees, Stochastic Search, AO* Search Implementation, Minimax Search, Alpha-Beta Pruning Basic Knowledge Representation and Reasoning: Propositional Logic, First-Order Logic, Forward Chaining and Backward Chaining, Introduction to Probabilistic Reasoning, Bayes Theorem

Constructing Search Trees:



Game Playing

Adversarial search, or game-tree search, is a technique for analyzing an adversarial game in order to try to determine who can win the game and what moves the players should make in order to win. Adversarial search is one of the oldest topics in Artificial Intelligence. The original ideas for adversarial search were developed by Shannon in 1950 and independently by Turing in 1951, in the context of the game of chess—and their ideas still form the basis for the techniques used today.

2-Person Games:

- Players: We call them Max and Min.
- Initial State: Includes board position and whose turn it is.

- Operators: These correspond to legal moves.
- Terminal Test: A test applied to a board position which determines whether the game is over. In chess, for example, this would be a checkmate or stalemate situation.
- Utility Function: A function which assigns a numeric value to a terminal state. For example, in chess the outcome is win (+1), lose (-1) or draw (0). Note that by convention, we always measure utility relative to Max.

Mini Max Algorithm:

1. Generate the whole game tree.
2. Apply the utility function to leaf nodes to get their values.
3. Use the utility of nodes at level n to derive the utility of nodes at level $n-1$.
4. Continue backing up values towards the root (one layer at a time).
5. Eventually the backed up values reach the top of the tree, at which point Max chooses the move that yields the highest value. This is called the minimax decision because it maximises the utility for Max on the assumption that Min will play perfectly to minimise it.

Algorithm: MINIMAX (Depth-First Version)

To determine the minimax value $V(J)$, do the following:

1. If J is terminal, return $V(J) = e(J)$; otherwise
2. Generate J 's successors J_1, J_2, \dots, J_b .
3. Evaluate $V(J_1), V(J_2), \dots, V(J_b)$ from left to right.
4. If J is a MAX node, return $V(J) = \max[V(J_1), \dots, V(J_b)]$.
5. If J is a MIN node, return $V(J) = \min[V(J_1), \dots, V(J_b)]$.

```

function MINIMAX-DECISION(state) returns an action
     $v \leftarrow \text{MAX-VALUE}(\textit{state})$ 
    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)
     $v \leftarrow -\infty$ 
    for  $a, s$  in SUCCESSORS(state) do
         $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$ 
    return  $v$ 

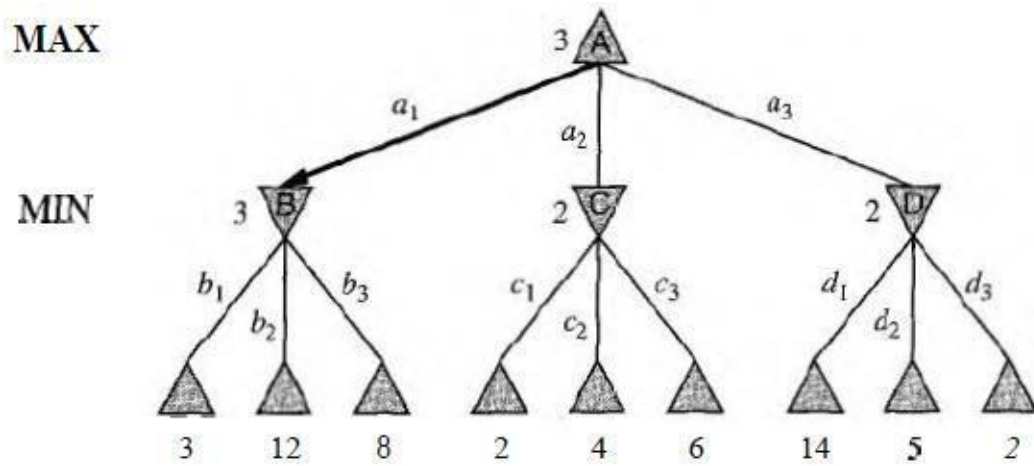


---

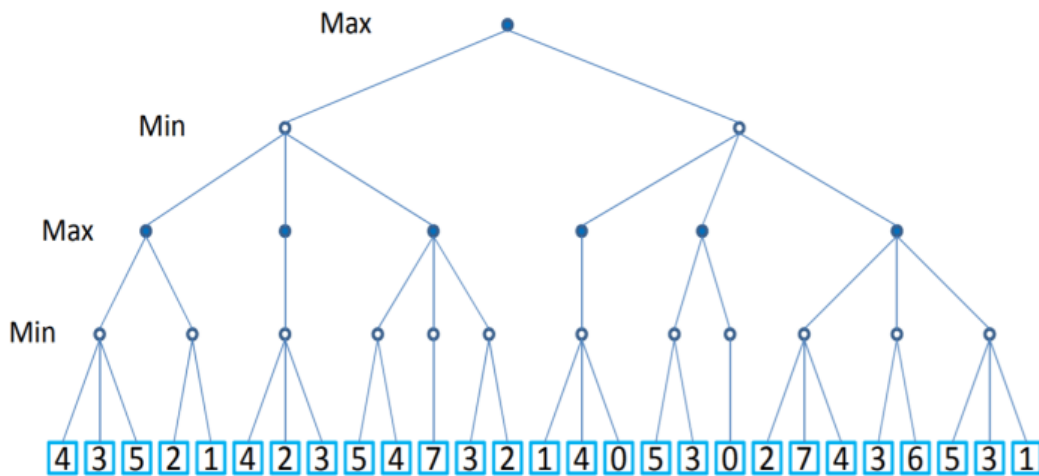

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow \infty$ 
    for  $a, s$  in SUCCESSORS(state) do
         $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$ 
    return  $v$ 

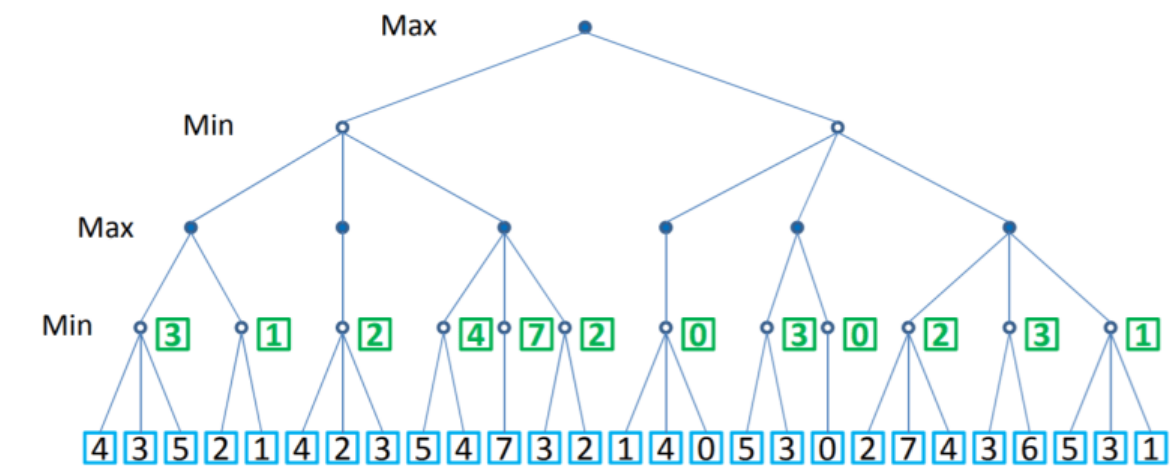
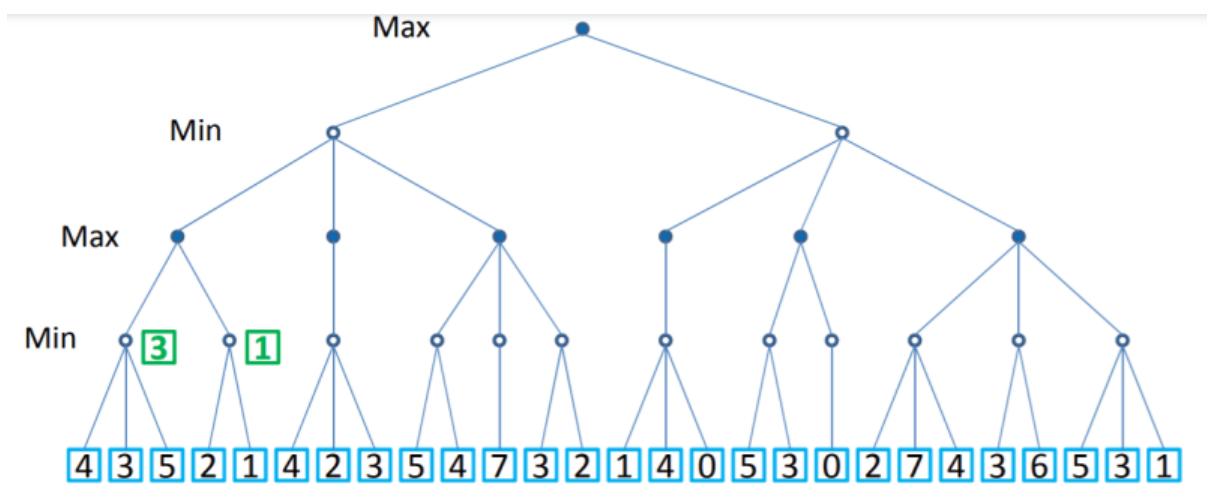
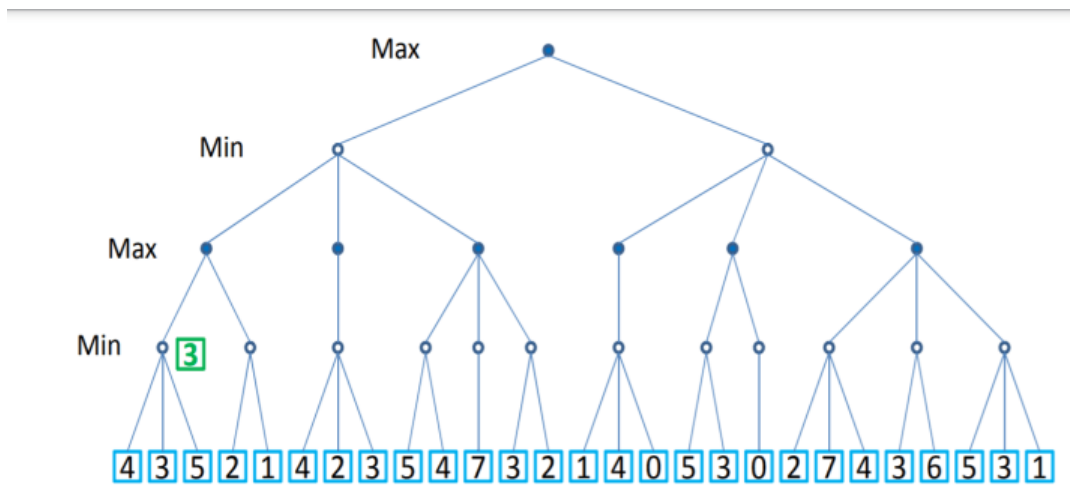
```

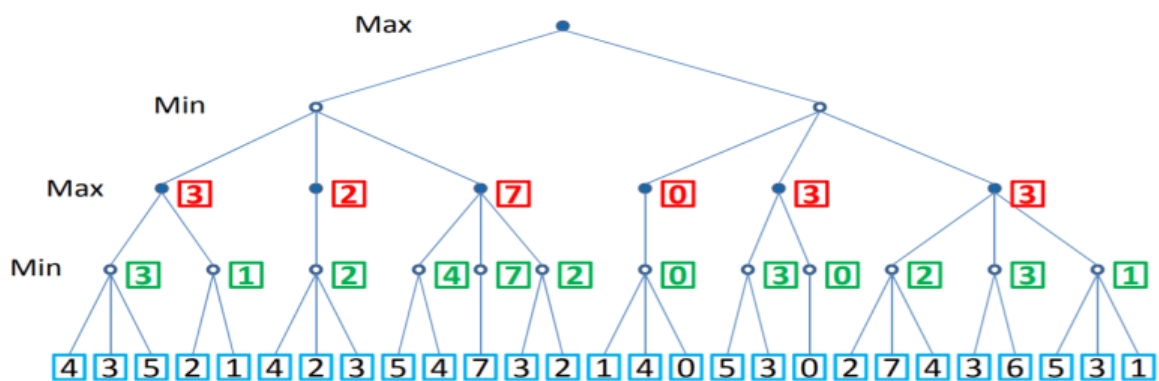
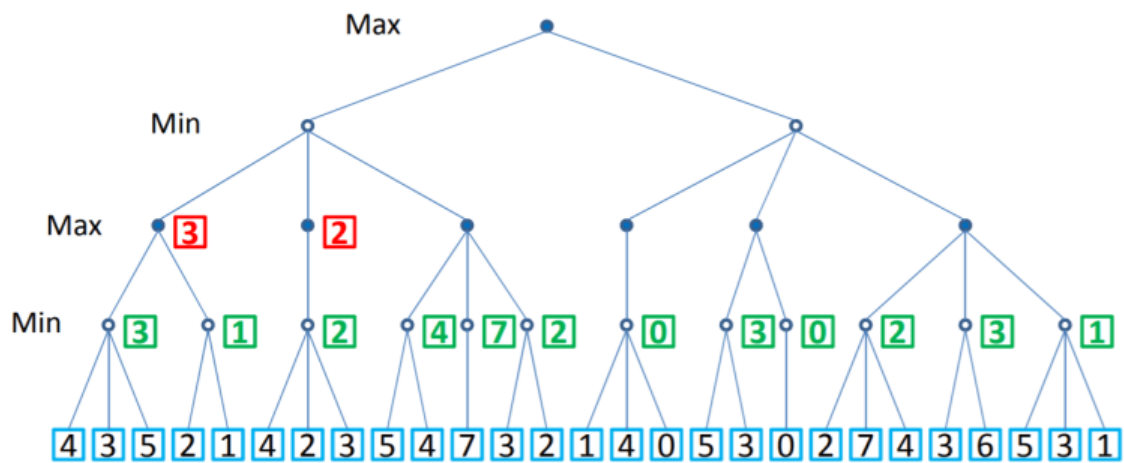
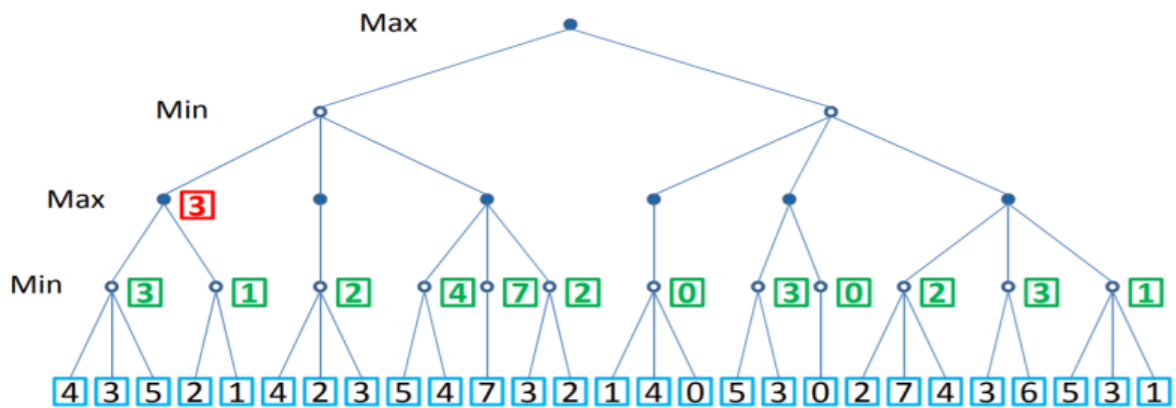
Example:

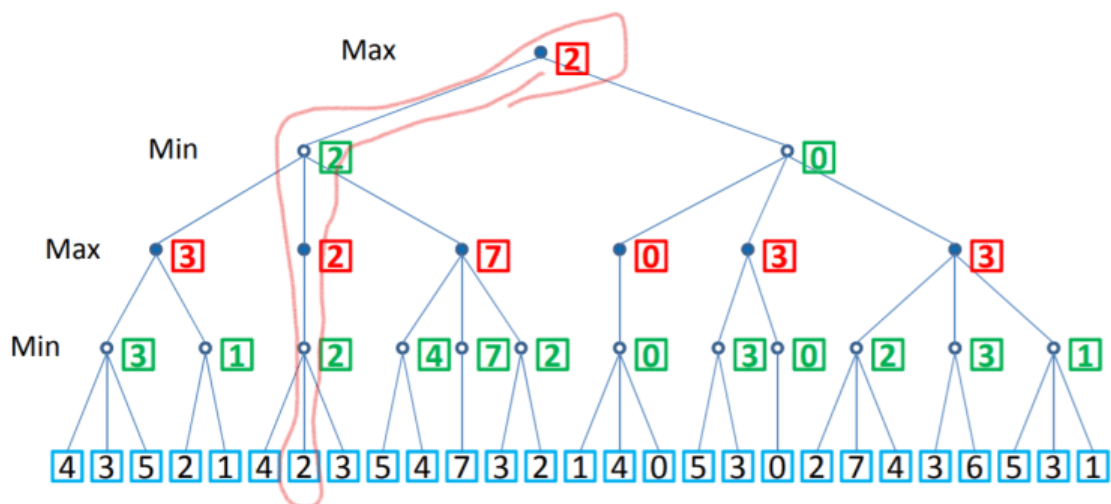
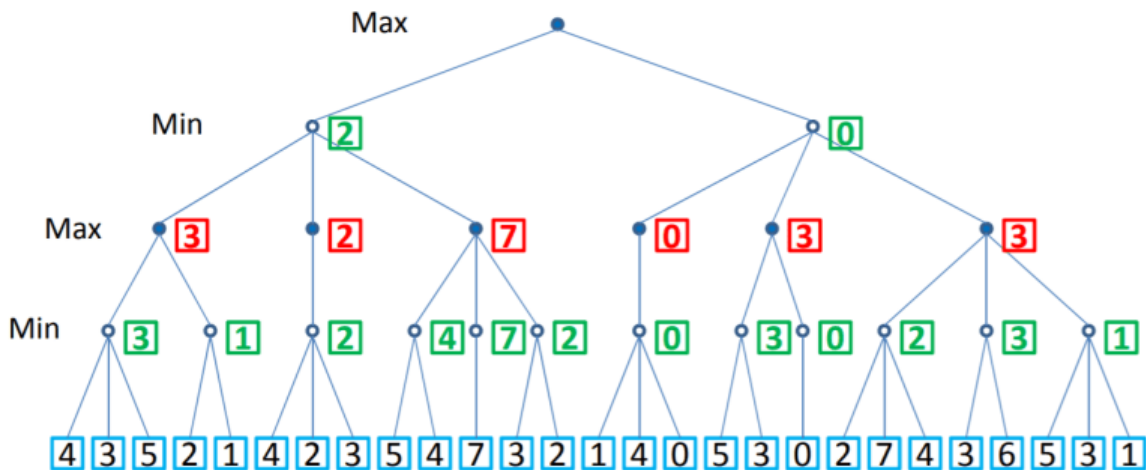


Example:









Properties of minimax:

- Complete : Yes (if tree is finite)
- Optimal : Yes (against an optimal opponent)
- Time complexity : $O(b^m)$
- Space complexity : $O(bm)$ (depth-first exploration)
- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
 —→ exact solution completely infeasible.

Limitations

- Not always feasible to traverse entire tree
- Time limitations

Alpha-Beta pruning algorithm:

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node
- **□-□ Pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.
- *Alpha-beta pruning* is used on top of minimax search to detect paths that do not need to be explored. The intuition is:
- The MAX player is always trying to maximize the score. Call this □.
- The MIN player is always trying to minimize the score. Call this □.
- **Alpha cutoff:** Given a Max node n , cutoff the search below n (i.e., don't generate or examine any more of n 's children) if $\alpha(n) \geq \beta(n)$
(α increases and passes β from below)
- **Beta cutoff.:** Given a Min node n , cutoff the search below n (i.e., don't generate or examine any more of n 's children) if $\beta(n) \leq \alpha(n)$
(β decreases and passes α from above)
- Carry α and β values down during search Pruning occurs whenever $\alpha \geq \beta$

Algorithm:

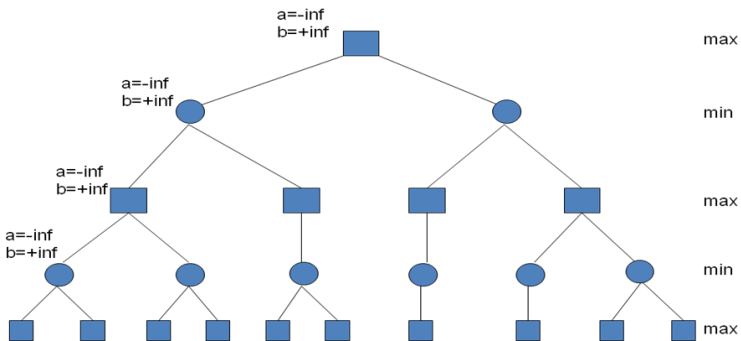
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
   $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$ 
  return the action in SUCCESSORS(state) with value  $v$ 

function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: state, current state in game
          $\alpha$ , 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
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, a, \beta))$ 
    if  $v \geq \beta$  then return  $v$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return  $v$ 

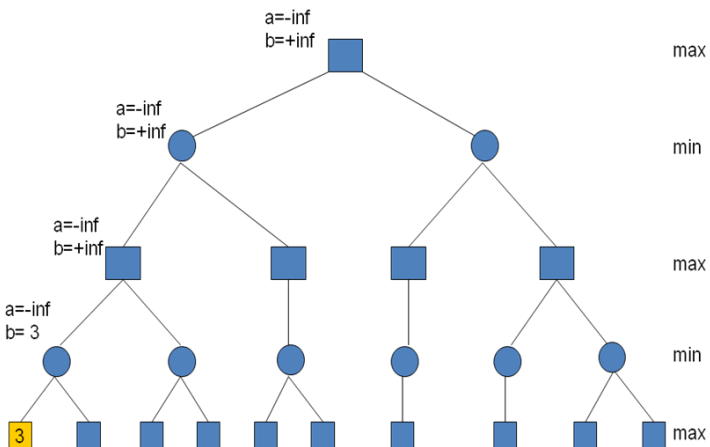
function MIN-VALUE(state,  $a, \beta$ ) returns a utility value
  inputs: state, current state in game
          $\alpha$ , the value of the best alternative for MAX along the path to state
          $\beta$ , the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, a, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 
```

Example:

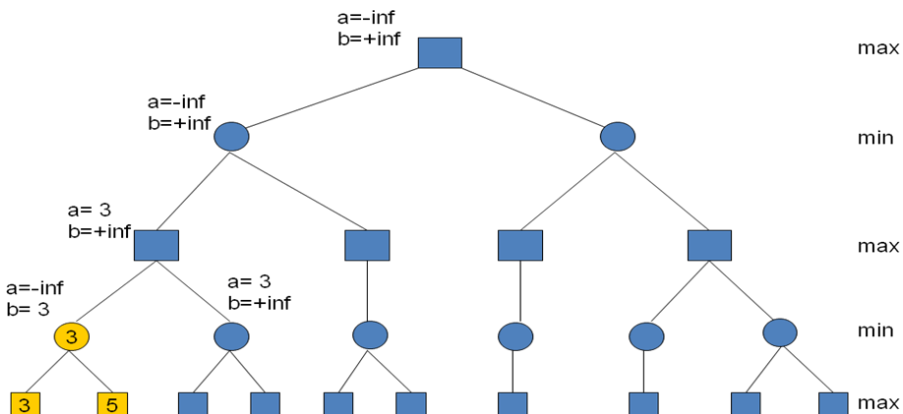
- 1) Setup phase: Assign to each left-most (or right-most) internal node of the tree,
variables: $\alpha = -\text{infinity}$, $\beta = +\text{infinity}$



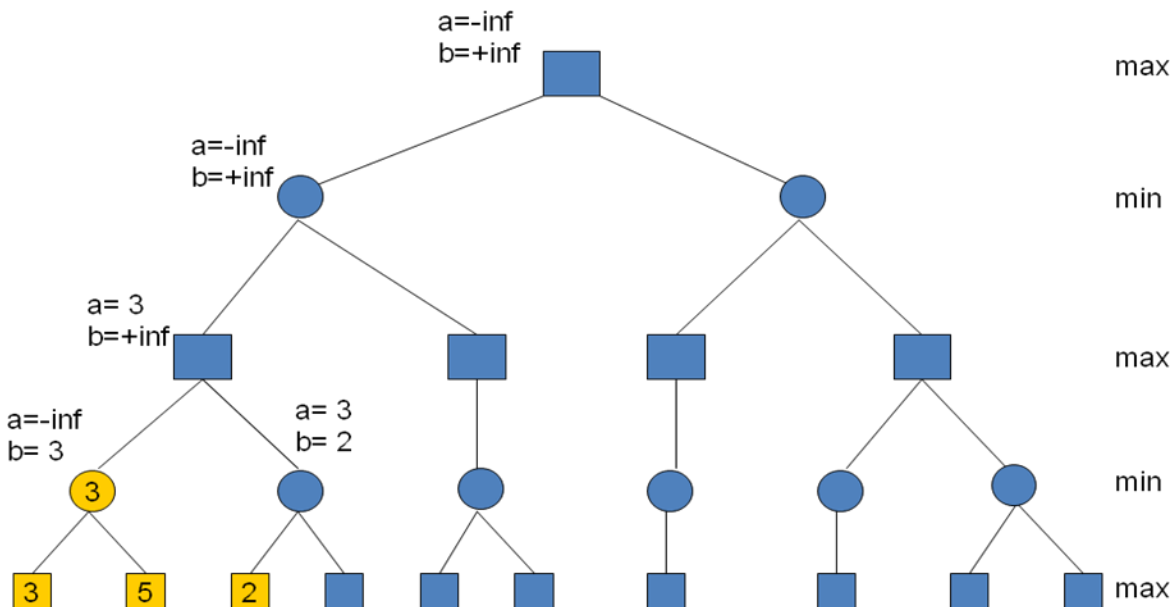
- 2) Look at first computed final configuration value. It's a 3. Parent is a min node, so set the beta (min) value to 3.



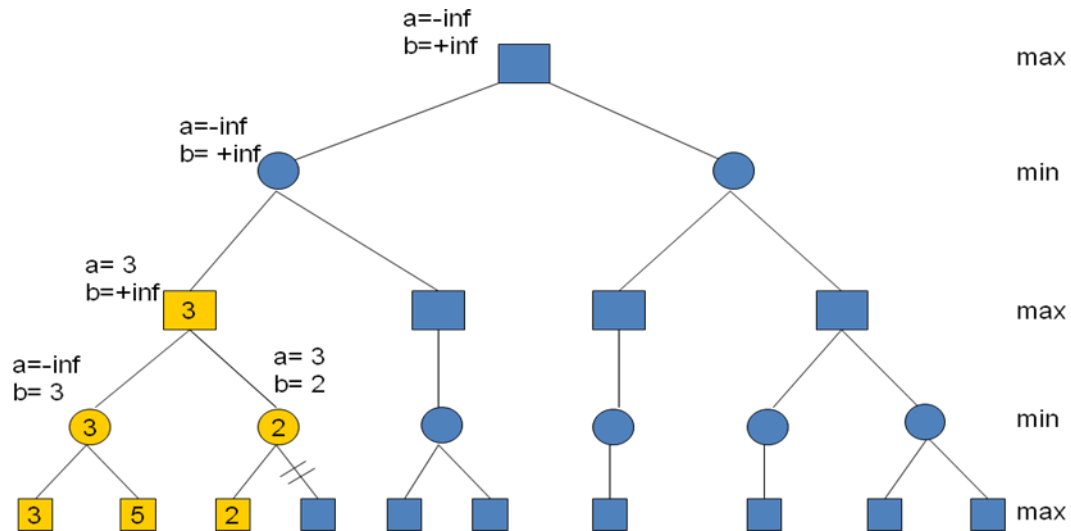
3) Look at next value, 5. Since parent is a min node, we want the minimum of 3 and 5 which is 3. Parent min node is done – fill alpha (max) value of its parent max node. Always set alpha for max nodes and beta for min nodes. Copy the state of the max parent node into the second unevaluated min child.



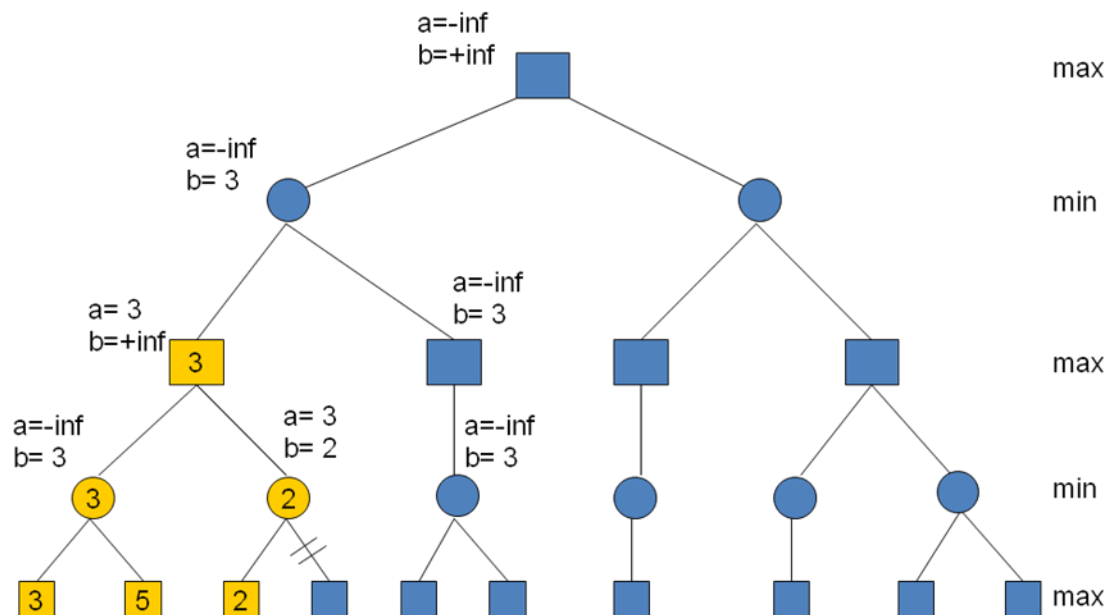
4) Look at next value, 2. Since parent node is min with $b = +\text{inf}$, 2 is smaller, change b.



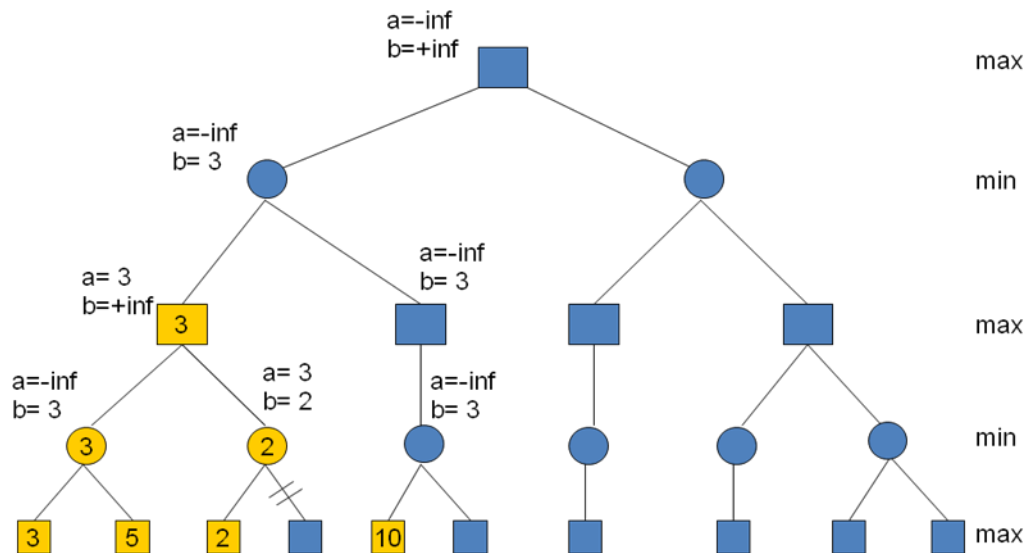
5) Now, the min parent node has a max value of 3 and min value of 2. The value of the 2nd child does not matter. If it is >2 , 2 will be selected for min node. If it is <2 , it will be selected for min node, but since it is <3 it will not get selected for the parent max node. Thus, we prune the right subtree of the min node. Propagate max value up the tree.



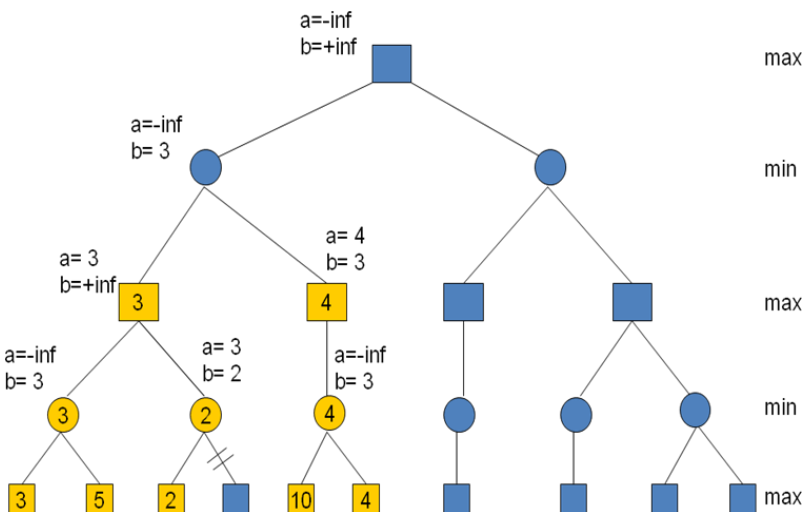
6) Max node is now done and we can set the beta value of its parent and propagate node state to sibling subtree's left-most path.



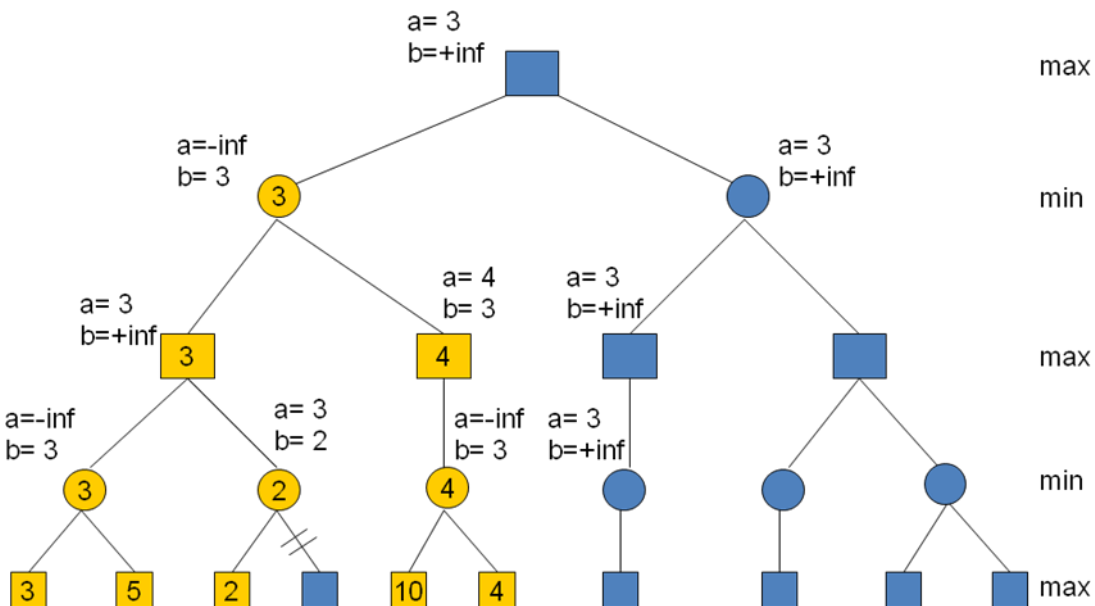
7) The next node is 10. 10 is not smaller than 3, so state of parent does not change. We still have to look at the 2nd child since alpha is still $-\infty$.



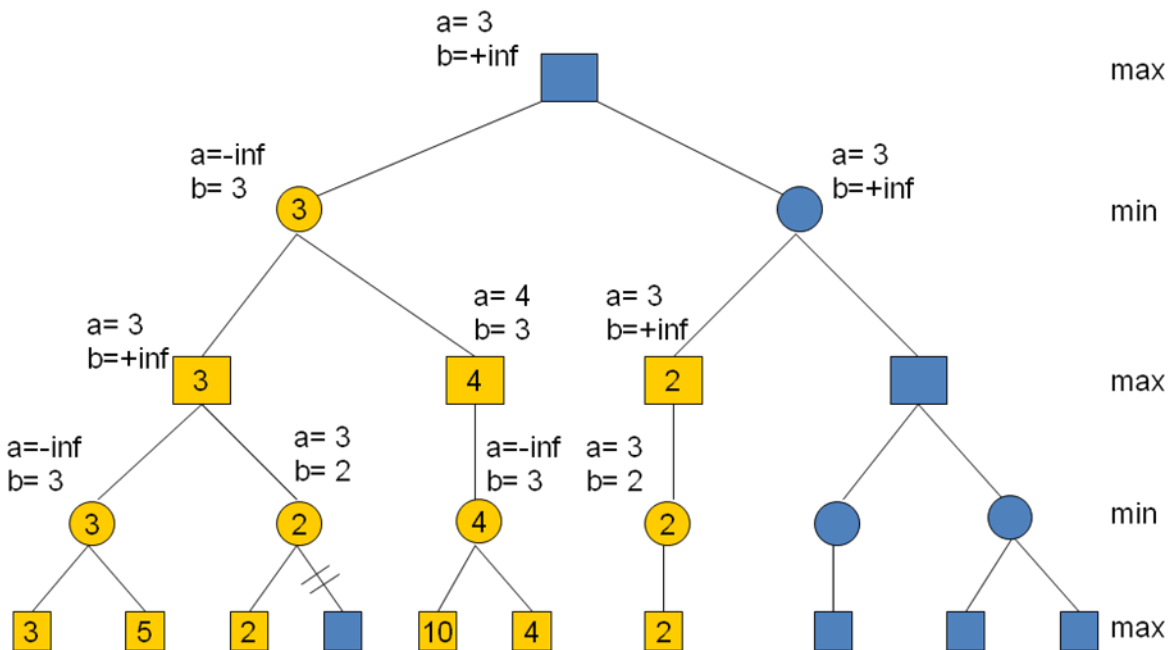
8) The next node is 4. Smallest value goes to the parent min node. Min subtree is done, so the parent max node gets the alpha (max) value from the child. Note that if the max node had a 2nd subtree, we can prune it since $a > b$.



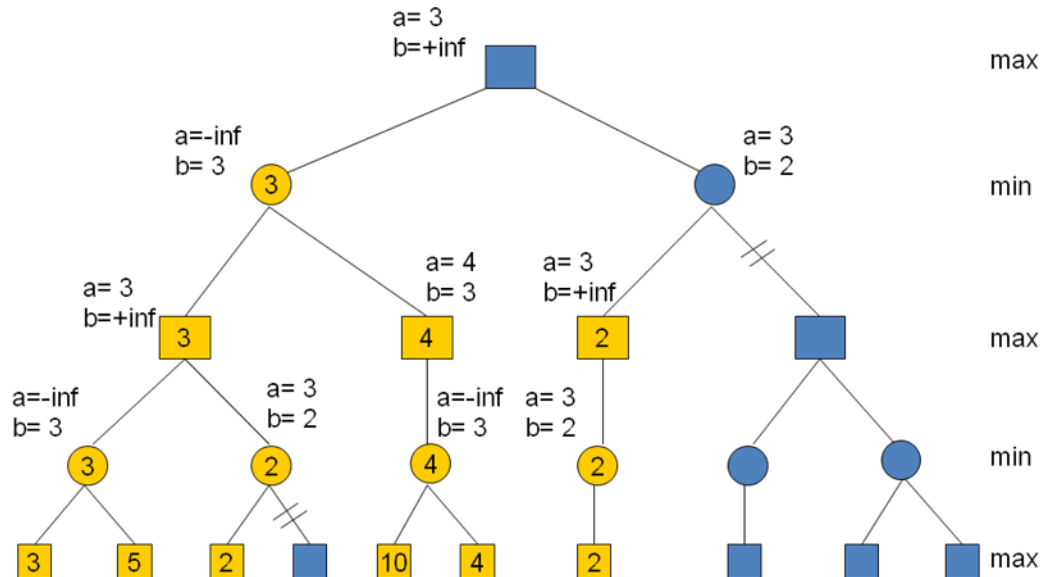
9) Continue propagating value up the tree, modifying the corresponding alpha/beta values. Also propagate the state of root node down the left-most path of the right subtree.



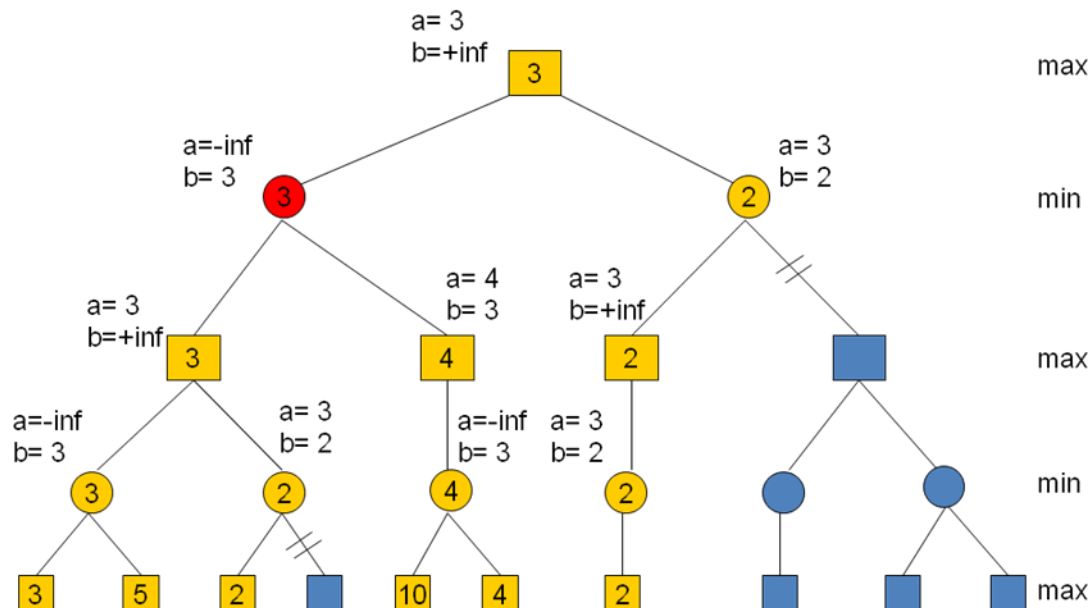
10) Next value is a 2. We set the beta (min) value of the min parent to 2. Since no other children exist, we propagate the value up the tree.



11) We have a value for the 3rd level max node, now we can modify the beta (min) value of the min parent to 2. Now, we have a situation that $a > b$ and thus the value of the rightmost subtree of the min node does not matter, so we prune the whole subtree.



12) Finally, no more nodes remain, we propagate values up the tree. The root has a value of 3 that comes from the left-most child. Thus, the player should choose the left-most child's move in order to maximize his/her winnings. As you can see, the result is the same as with the mini-max example, but we did not visit all nodes of the tree.



AO* Search: (And-Or) Graph:

The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

OPEN:

It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.

CLOSE:

It contains the nodes that have already been processed.

6 7: The distance from current node to goal node.

Algorithm:

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T0.

Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in

CLOSE

Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.

Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.

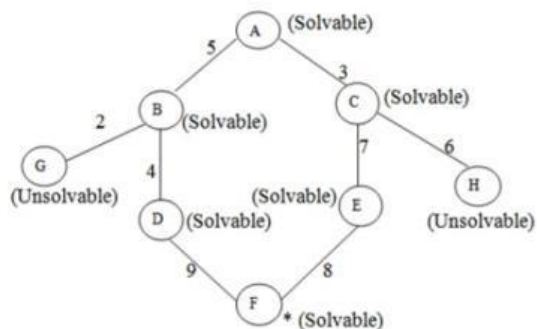
Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

Step 7: Return to Step 2.

Step 8: Exit.

Implementation:

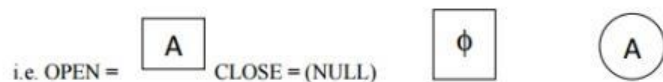
Let us take the following example to implement the AO* algorithm.



Figure

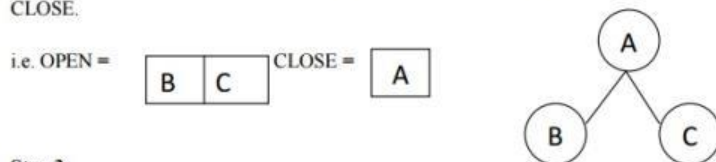
Step 1:

In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H. Take A as the starting node. So place A into OPEN.



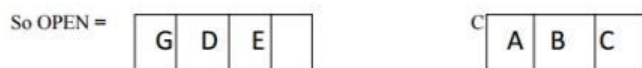
Step 2:

The children of A are B and C which are solvable. So place them into OPEN and place A into the CLOSE.



Step 3:

Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.



(O)

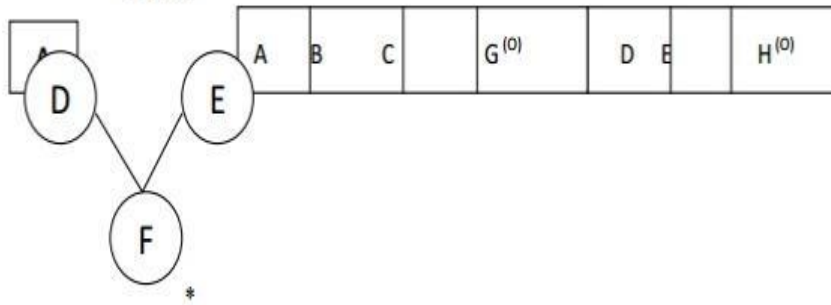
'O' indicated that the nodes G and H are unsolvable.

Step 4:

As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN =

CLOSE =



Step 5:

Now we have been reached at our goal state. So place F into CLOSE.

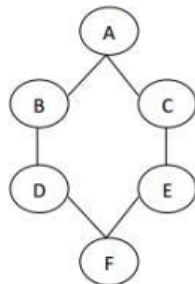


i.e. CLOSE =

Step 6:

Success and Exit

AO* Graph:



Figure

Advantages:

It is an optimal algorithm.

If traverse according to the ordering of nodes. It can be used for both OR and AND graph.

Disadvantages:

Sometimes for unsolvable nodes, it can't find the optimal path. Its complexity is than other algorithms.

BASIC KNOWLEDGE REPRESENTATION AND REASONING:

- Humans are best at understanding, reasoning, and interpreting knowledge. Human knows things, which is knowledge and as per their knowledge they perform various actions in the real world.
- But how machines do all these things comes under knowledge representation
 - **There are three factors which are put into the machine, which makes it valuable:**
 - **Knowledge:** The information related to the environment is stored in the machine.
 - **Reasoning:** The ability of the machine to understand the stored knowledge.
 - **Intelligence:** The ability of the machine to make decisions on the basis of the stored information.
 - A knowledge representation language is defined by two aspects:
 - The syntax of a language describes the possible configurations that can constitute sentences.
 - The semantics determines the facts in the world to which the sentences refer.
 - For example, the syntax of the language of arithmetic expressions says that if x and y are expressions denoting numbers, then $x > y$ is a sentence about numbers. The semantics of the language says that $x > y$ is false when y is a bigger number than x , and true otherwise. From the syntax and semantics, we can derive an inference mechanism for an agent that uses the language.
 - Recall that the semantics of the language determine the fact to which a given sentence refers. Facts are part of the world,
- whereas their representations must be encoded in some way that can be physically stored within an agent. We cannot put the world inside a computer (nor can we put it inside a human), so all reasoning mechanisms must operate on representations of facts, rather than on the facts themselves. Because sentences are physical configurations of parts of the agent,

Reasoning must be a process of constructing new physical configurations from old ones. Proper reasoning should ensure that the new configurations represent facts that actually follow from the facts that the old configurations represent.

- We want to generate new sentences that are necessarily true, given that the old sentences are true.

This relation between sentences is called entailment.

- In mathematical notation, the relation of entailment between a knowledge base KB and a sentence α is pronounced "KB entails α " and written as $KB \models \alpha$.
- An inference procedure can do one of two things:
 - given a knowledge base KB, it can generate new sentences α that purport to be entailed by KB.
 - E.g., $x + y = 4$ entails $4 = x + y$
- Entailment is a relationship between sentences (i.e., syntax) that is based on semantics

PROPOSITIONAL LOGIC:

- Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions.
- A proposition is a declarative statement which is either true or false.

It is a technique of knowledge representation in logical and mathematical form

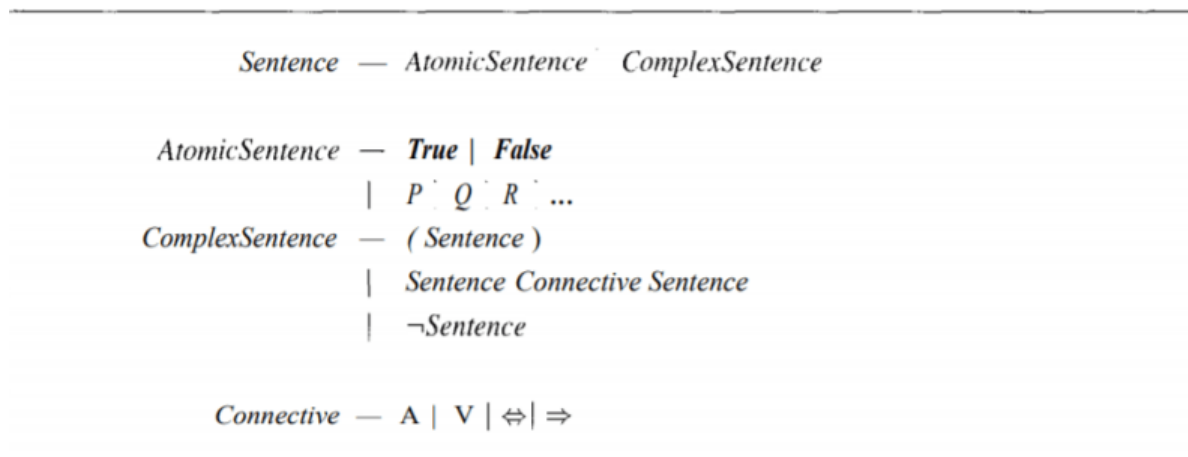


Figure 6.8 A BNF (Backus-Naur Form) grammar of sentences in propositional logic.

Syntax of propositional logic:

- The symbols of propositional logic are the logical constants True and False, proposition symbols such as P and Q, the logical connectives A, V, \Leftrightarrow , \Rightarrow and \neg and parentheses,
- All sentences are made by putting these symbols together using the following rules:
 - The logical constants True and False are sentences by themselves.
 - A propositional symbol such as P or Q is a sentence by itself.
 - Wrapping parentheses around a sentence yields a sentence, for example, (P A Q).

A sentence can be formed by combining simpler sentences with one of the five logical connectives::

1. Negation: A sentence such as $\neg P$ is called negation of P. A literal can be either Positive literal or negative literal.

Example: P=Today is not Sunday $\rightarrow \neg p$

2. Conjunction: A sentence which has \wedge connective such as, $P \wedge Q$ is called a conjunction.

Example: Rohan is intelligent and hardworking. It can be written as,

P= Rohan is intelligent,

Q= Rohan is hardworking. $\rightarrow P \wedge Q$.

3. Disjunction: A sentence which has \vee connective, such as $P \vee Q$. is called disjunction, where P and Q are the propositions.

4. Example: "Ritika is a doctor or Engineer",

Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as $P \vee Q$.

5. Implication: A sentence such as $P \rightarrow Q$, is called an implication. Implications are also known as if-then rules. It can be represented as

If it is raining, then the street is wet.

Let P= It is raining, and Q= Street is wet, so it is represented as $P \rightarrow Q$

6. Biconditional: A sentence such as $P \Leftrightarrow Q$ is a Biconditional sentence, example If I am breathing, then I am alive

P= I am breathing, Q= I am alive, it can be represented as $P \Leftrightarrow Q$.

Precedence of connectives:

Precedence Operators

First Precedence Parenthesis

Second Precedence Negation

Third Precedence Conjunction(AND)

Fourth Precedence Disjunction(OR)

Fifth Precedence Implication

Six Precedence Biconditional

Precedence of connectives:

Semantics

- The semantics of propositional logic is also quite straightforward. We define it by specifying the interpretation of the proposition symbols and constants, and specifying the meanings of the logical connectives.

Validity

- Truth tables can be used not only to define the connectives, but also to test for valid sentences.
- Given a sentence, we make a truth table with one row for each of the possible combinations of truth values for the proposition symbols in the sentence.
- If the sentence is true in every row, then the sentence is valid. For example, the sentence $((P \vee H) \wedge \neg H) \Rightarrow P$

Translating English into logic:

- User defines semantics of each propositional symbol
- P: It is Hot
- Q: It is Humid
- R: It is raining

1. If it is humid then it is hot

$Q \rightarrow P$

.If it is hot and humid , then it is raining

$(P \wedge Q) \rightarrow R$

Limitations of Propositional logic:

- In propositional logic, we can only represent the facts, which are either true or false.
- PL is not sufficient to represent the complex sentences or natural language statements.
- The propositional logic has very limited expressive power.
- Consider the following sentence, which we cannot represent using PL logic.
- "Some humans are intelligent", or "Sachin likes cricket"

First-order logic:

Advantages of Propositional Logic

- The declarative nature of propositional logic, specify that knowledge and inference are separate, and inference is entirely domain-independent. □ Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds.
- It also has sufficient expressive power to deal with partial information, using disjunction and negation.
- Propositional logic has a third COMPOSITIONALITY property that is desirable in representation languages, namely, compositionality. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, the meaning of “ $S_{1,4} \wedge S_{1,2}$ ” is related to the meanings of “ $S_{1,4}$ ” and “ $S_{1,2}$ ”.

Drawbacks of Propositional Logic

Propositional logic lacks the expressive power to concisely describe an environment with many objects.

For example, we were forced to write a separate rule about breezes and pits for each square, such as

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.

In English, it seems easy enough to say, “Squares adjacent to pits are breezy.”

The syntax and semantics of English somehow make it possible to describe the environment concisely

SYNTAX AND SEMANTICS OF FIRST-ORDER LOGIC

Models for first-order logic :

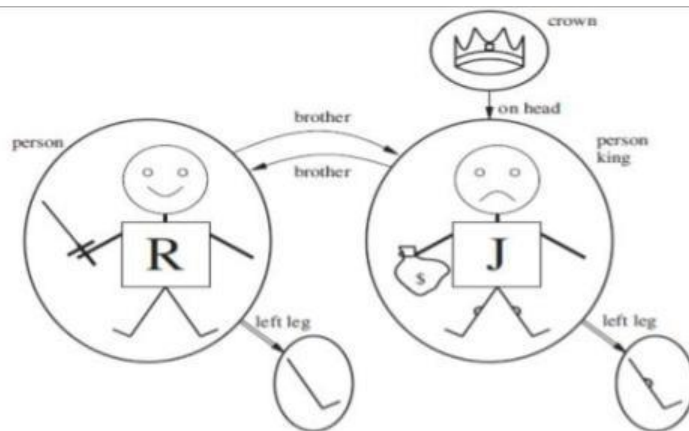
The models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic have objects. The domain of a model is the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object.

A relation is just the set of tuples of objects that are related.

Unary Relation: Relations relates to single Object Binary Relation: Relation Relates to multiple objects Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object.

For Example:

Richard the Lionheart, King of England from 1189 to 1199; His younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; crown



Unary Relation : John is a king **Binary Relation :** crown is on head of john , Richard is brother of john The unary "left leg" function includes the following mappings: (Richard the Lionheart) ->Richard's left leg (King John) ->Johns left Leg

Symbols and interpretations

Symbols are the basic syntactic elements of first-order logic. Symbols stand for objects, relations, and functions.

The symbols are of three kinds:
 ☐ Constant symbols which stand for objects; Example: John, Richard
 ☐ Predicate symbols, which stand for relations; Example: OnHead, Person, King, and Crown

☐ Function symbols, which stand for functions. Example: left leg Symbols will begin with uppercase letters.

Interpretation The semantics must relate sentences to models in order to determine truth. For this to happen, we need an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

For Example:

Richard refers to Richard the Lionheart and John refers to the evil king John. Brother refers to the brotherhood relation. OnHead refers to the "on head relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns. LeftLeg refers to the "left leg" function,

The truth of any sentence is determined by a model and an interpretation for the sentence's symbols. Therefore, entailment, validity, and so on are defined in terms of all possible models and all possible interpretations. The number of domain elements in each model may be unbounded—for example, the domain elements may be integers or real numbers. Hence, the number of possible models is unbounded, as is the number of interpretations.

Term

A term is a logical expression that refers to an object. Constant symbols are therefore terms. Complex Terms A complex term is just a complicated kind of name. A complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. For example: "King John's left leg". Instead of using a constant symbol, we use LeftLeg(John). The formal semantics of terms Consider a term $f(t_1, \dots, t_n)$. The function symbol f refers to some function in the model (F); the argument terms refer to objects in the domain (call them d_1, \dots, d_n); and the term as a whole refers to the object that is the value of the function F applied to d_1, \dots, d_n . For example, the LeftLeg function symbol refers to the function “(King John) \rightarrow John's left leg” and John refers to King John, then LeftLeg(John) refers to King John's left leg. In this way, the interpretation fixes the referent of every term.

Atomic sentences

An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms: For Example: Brother(Richard, John).

Atomic sentences can have complex terms as arguments. For Example: Married (Father(Richard), Mother(John)).

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

Complex sentences Complex sentences can be constructed using logical Connectives, just as in

propositional calculus. For Example:

- ✓ $\neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{John})$
- ✓ $\text{Brother}(\text{Richard}, \text{John}) \wedge \text{Brother}(\text{John}, \text{Richard})$
- ✓ $\text{King}(\text{Richard}) \vee \text{King}(\text{John})$
- ✓ $\neg \text{King}(\text{Richard}) \Rightarrow \text{King}(\text{John})$

Quantifiers

Quantifiers express properties of entire collections of objects, instead of enumerating the objects by name.

First-order logic contains two standard quantifiers:

1. Universal Quantifier
2. Existential Quantifier

Universal Quantifier

Universal quantifier is defined as follows:

"Given a sentence $\forall x P$, where P is any logical expression, says that P is true for every object x ."

More precisely, $\forall x P$ is true in a given model if P is true in all possible **extended interpretations** constructed from the interpretation given in the model, where each extended interpretation specifies a domain element to which x refers.

For Example: "All kings are persons," is written in first-order logic as

$\forall x \text{King}(x) \Rightarrow \text{Person}(x)$.

\forall is usually pronounced "For all"

Thus, the sentence says, "For all x , if x is a king, then x is a person." The symbol x is called a variable. Variables are lowercase letters. A variable is a term all by itself, and can also serve as the argument of a function. A term with no variables is called a ground term.

Assume we can extend the interpretation in different ways: $x \rightarrow$ Richard the Lionheart, $x \rightarrow$ King John, $x \rightarrow$ Richard's left leg, $x \rightarrow$ John's left leg, $x \rightarrow$ the crown

The universally quantified sentence $\forall x \text{King}(x) \Rightarrow \text{Person}(x)$ is true in the original model if the sentence $\text{King}(x) \Rightarrow \text{Person}(x)$ is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king \Rightarrow Richard the Lionheart is a person. King John is a king \Rightarrow King John is a person. Richard's left leg is a king \Rightarrow Richard's left leg is a person. John's left leg is a king \Rightarrow John's left leg is a person. The crown is a king \Rightarrow the crown is a person.

Existential quantification (\exists)

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier.

"The sentence $\exists x P$ says that P is true for at least one object x . More precisely, $\exists x P$ is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element." $\exists x$ is pronounced "There exists an x such that . . ." or "For some x . . .".

For example, that King John has a crown on his head, we write $\exists x \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$ Given assertions:

Richard the Lionheart is a crown \wedge Richard the Lionheart is on John's head; King John is a crown \wedge King John is on John's head; Richard's left leg is a crown \wedge Richard's left leg is on John's head; John's left leg is a crown \wedge John's left leg is on John's head; The crown is a crown \wedge the crown is on John's head. The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Just as \Rightarrow appears to be the natural connective to use with \forall , \wedge is the natural connective to use with \exists .

Nested quantifiers

One can express more complex sentences using multiple quantifiers.

For example, "Brothers are siblings" can be written as $\forall x \forall y \text{ Brother}(x, y) \Rightarrow \text{Sibling}(x, y)$. Consecutive quantifiers of the same type can be written as one quantifier with several variables.

For example, to say that siblinghood is a symmetric relationship, we can write $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x)$.

In other cases we will have mixtures.

For example: 1. "Everybody loves somebody" means that for every person, there is someone that person loves: $\forall x \exists y \text{ Loves}(x, y)$. 2. On the other hand, to say "There is someone who is loved by everyone," we write $\exists y \forall x \text{ Loves}(x, y)$.

Connections between \forall and \exists

Universal and Existential quantifiers are actually intimately connected with each other, through negation.

Example assertions:

1. "Everyone dislikes medicine" is the same as asserting "there does not exist someone who likes medicine", and vice versa: " $\forall x \neg \text{Likes}(x, \text{medicine})$ " is equivalent to " $\neg \exists x \text{ Likes}(x, \text{medicine})$ ".
2. "Everyone likes ice cream" means that "there is no one who does not like ice cream": $\forall x \text{ Likes}(x, \text{IceCream})$ is equivalent to $\neg \exists x \neg \text{Likes}(x, \text{IceCream})$.

Because \forall is really a conjunction over the universe of objects and \exists is a disjunction that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

Because \forall is really a conjunction over the universe of objects and \exists is a disjunction that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{ll}
 \forall x \neg P \equiv \neg \exists x P & \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\
 \neg \forall x P \equiv \exists x \neg P & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\
 \forall x P \equiv \neg \exists x \neg P & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\
 \exists x P \equiv \neg \forall x \neg P & P \vee Q \equiv \neg(\neg P \wedge \neg Q) ..
 \end{array}$$

Thus, Quantifiers are important in terms of readability.

Equality

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

For example,

“Father(John) = Henry” says that the object referred to by Father (John) and the object referred to by Henry are the same.

Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object. The equality symbol can be used to state facts about a given function. It can also be used with negation to insist that two terms are not the same object.

For example,

“Richard has at least two brothers” can be written as, $\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard}) \wedge \neg(x=y)$.

The sentence

$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard})$ does not have the intended meaning.

In particular, it is true only in the model where Richard has only one brother considering the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x=y)$ rules out such models.

<i>Sentence</i>	\rightarrow	<i>AtomicSentence</i> <i>ComplexSentence</i>
<i>AtomicSentence</i>	\rightarrow	<i>Predicate</i> <i>Predicate</i> (<i>Term</i> ,...) <i>Term</i> = <i>Term</i>
<i>ComplexSentence</i>	\rightarrow	(<i>Sentence</i>) ! <i>Sentence</i>
		\neg <i>Sentence</i>
		<i>Sentence</i> \wedge <i>Sentence</i>
		<i>Sentence</i> \vee <i>Sentence</i>
		<i>Sentence</i> \Rightarrow <i>Sentence</i>
		<i>Sentence</i> \Leftrightarrow <i>Sentence</i>
		<i>Quantifier</i> <i>Variable</i> ,... <i>Sentence</i>
<i>Term</i>	\rightarrow	<i>Function</i> (<i>Term</i> ,...)
		<i>Constant</i>
		<i>Variable</i>
<i>Quantifier</i>	\rightarrow	\forall \exists
<i>Constant</i>	\rightarrow	<i>A</i> <i>X₁</i> <i>John</i> ...
<i>Variable</i>	\rightarrow	<i>a</i> <i>x</i> <i>s</i> ...
<i>Predicate</i>	\rightarrow	<i>True</i> <i>False</i> <i>After</i> <i>Loves</i> <i>Raining</i> ...
<i>Function</i>	\rightarrow	<i>Mother</i> <i>LeftLeg</i> ...
OPERATOR PRECEDENCE	:	$\neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Backus Naur Form for First Order Logic

USING FIRST ORDER LOGIC Assertions and queries in first-order logic

Assertions:

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions.

For example,

John is a king, TELL (KB, King (John)). Richard is a person. TELL (KB, Person (Richard)). All kings are persons: TELL (KB, $\forall x$ King(x) \Rightarrow Person(x)).

Asking Queries:

We can ask questions of the knowledge base using ASK. Questions asked with ASK are called queries or goals.

For example,

ASK (KB, King (John)) returns true.

Any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two preceding assertions, the query:

“ASK (KB, Person (John))” should also return true.

Substitution or binding list

We can ask quantified queries, such as $\text{ASK}(\text{KB}, \exists x \text{ Person}(x))$.

The answer is true, but this is perhaps not as helpful as we would like. It is rather like answering “Can you tell me the time?” with “Yes.”

If we want to know what value of x makes the sentence true, we will need a different function, ASKVARS , which we call with $\text{ASKVARS}(\text{KB}, \text{Person}(x))$ and which yields a stream of answers.

In this case there will be two answers: $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. Such an answer is called a substitution or binding list.

ASKVARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values.

The kinship domain

The objects in Kinship domain are people.

We have two unary predicates, Male and Female.

Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle.

We use functions for Mother and Father, because every person has exactly one of each of these.

We can represent each function and predicate, writing down what we know in terms of the other symbols.

For example:-

1. one's mother is one's female parent: $\forall m, c \text{ Mother}(c)=m \Leftrightarrow \text{Female}(m) \wedge \text{Parent}(m,$
2. One's husband is one's male spouse: $\forall w, h \text{ Husband}(h,w) \Leftrightarrow \text{Male}(h) \wedge \text{Spouse}(h,w)$.
3. Male and female are disjoint categories: $\forall x \text{ Male}(x) \Leftrightarrow \neg \text{Female}(x)$.
4. Parent and child are inverse relations: $\forall p, c \text{ Parent}(p, c) \Leftrightarrow \text{Child}(c, p)$.
5. A grandparent is a parent of one's parent: $\forall g, c \text{ Grandparent}(g, c) \Leftrightarrow \exists p \text{ Parent}(g, p) \wedge \text{Parent}(p, c)$

6. A sibling is another child of one's parents: $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow x \neq y \wedge \exists p \text{ Parent}(p, x) \wedge \text{Parent}(p, y)$

Axioms:

Each of these sentences can be viewed as an axiom of the kinship domain. Axioms are commonly associated with purely mathematical domains. They provide the basic factual information from which useful conclusions can be derived.

Kinship axioms are also definitions; they have the form $\forall x, y P(x, y) \Leftrightarrow \dots$

The axioms define the Mother function, Husband, Male, Parent, Grandparent, and Sibling predicates in terms of other predicates.

Our definitions “bottom out” at a basic set of predicates (Child, Spouse, and Female) in terms of which the others are ultimately defined. This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions.

Theorems:

Not all logical sentences about a domain are axioms. Some are theorems—that is, they are entailed by the axioms.

For example, consider the assertion that siblinghood is symmetric: $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x)$.

It is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return true. From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time.

Axioms :Axioms without Definition

Not all axioms are definitions. Some provide more general information about certain predicates without

constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully.

For example, there is no obvious definitive way to complete the sentence

$\forall x \text{Person}(x) \Leftrightarrow \dots$

Fortunately, first-order logic allows us to make use of the Person predicate without completely defining it.

Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$\forall x \text{Person}(x) \Rightarrow \dots \forall x \dots \Rightarrow \text{Person}(x) .$

Axioms can also be “just plain facts,” such as Male (Jim) and Spouse (Jim, Laura). Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms

Numbers, sets, and lists

Number theory

Numbers are perhaps the most vivid example of how a large theory can be built up from NATURAL NUMBERS a tiny kernel of axioms. We describe here the theory of natural numbers or non-negative integers. We need:

predicate NatNum that will be true of natural numbers;

PEANO AXIOMS constant symbol, 0; One function symbol, S (successor). The Peano axioms define natural numbers and addition.

Natural numbers are defined recursively: $\text{NatNum}(0) . \forall n \text{NatNum}(n) \Rightarrow \text{NatNum}(S(n)) .$

That is, 0 is a natural number, and for every object n, if n is a natural number, then S(n) is a natural number.

So the natural numbers are 0, S(0), S(S(0)), and so on. We also need axioms to constrain the successor function: $\forall n 0 \neq S(n) . \forall m, n m \neq n \Rightarrow S(m) \neq S(n) .$

Now we can define addition in terms of the successor function: $\forall m \text{NatNum}(m) \Rightarrow + (0, m) = m .$

$\forall m, n \text{NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow + (S(m), n) = S(+ (m, n))$

The first of these axioms says that adding 0 to any natural number m gives m itself. Addition is represented using the binary function symbol “+” in the term $+(m, 0)$;

To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so the second axiom becomes :

$$\forall m, n \text{ NatNum } (m) \wedge \text{NatNum}(n) \Rightarrow (m + 1) + n = (m + n) + 1 .$$

This axiom reduces addition to repeated application of the successor function. Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

Sets

The domain of sets is also fundamental to mathematics as well as to commonsense reasoning. Sets can be represented as individual sets, including empty sets.

Sets can be built up by:
 adding an element to a set or
 Taking the union or intersection of two sets.

Operations that can be performed on sets are:

To know whether an element is a member of a set Distinguish sets from objects that are not sets.

Vocabulary of set theory:

The empty set is a constant written as $\{ \}$. There is one unary predicate, Set , which is true of sets. The binary predicates are

$x \in s$ (x is a member of set s) $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2).

The binary functions are

$s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining element x to set s).

Forward Chaining and backward chaining in AI

Inference engine:

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

- a. **Forward chaining**
- b. **Backward chaining**

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: $(\neg p \vee \neg q \vee k)$. It has only one positive literal k .

It is equivalent to $p \wedge q \rightarrow k$.

A. Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available

data.

- Forward-chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Facts Conversion into FOL:

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)

American (p) \wedge weapon(q) \wedge sells (p, q, r) \wedge hostile(r) \rightarrow Criminal(p) ... (1)

- Country A has some missiles. **?p Owns(A, p) \wedge Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.

Owns(A, T1) (2)

Missile(T1) (3)

- All of the missiles were sold to country A by Robert.

?p Missiles(p) \wedge Owns (A, p) \rightarrow Sells (Robert, p, A) (4)

- Missiles are weapons.

Missile(p) \rightarrow Weapons (p) (5)

- Enemy of America is known as hostile.

Enemy(p, America) \rightarrow Hostile(p) (6)

- Country A is an enemy of America.

Enemy (A, America) (7)

- Robert is American

American(Robert). (8)

Forward chaining proof:

Step-1:

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**. All these facts will be represented as below.

American (Robert)	Missile (T1)	Owns (A,T1)	Enemy (A, America)
-------------------	--------------	-------------	--------------------

Step-2:

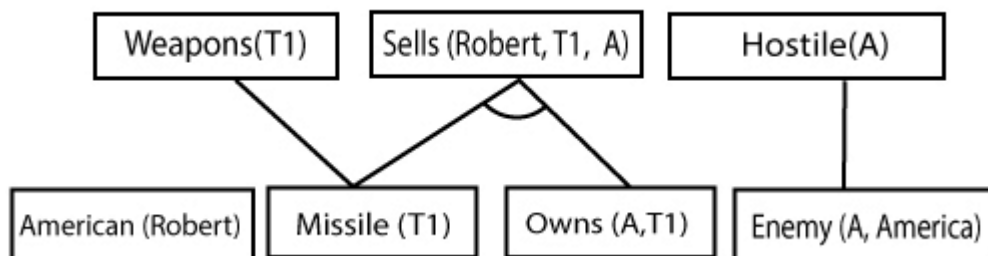
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

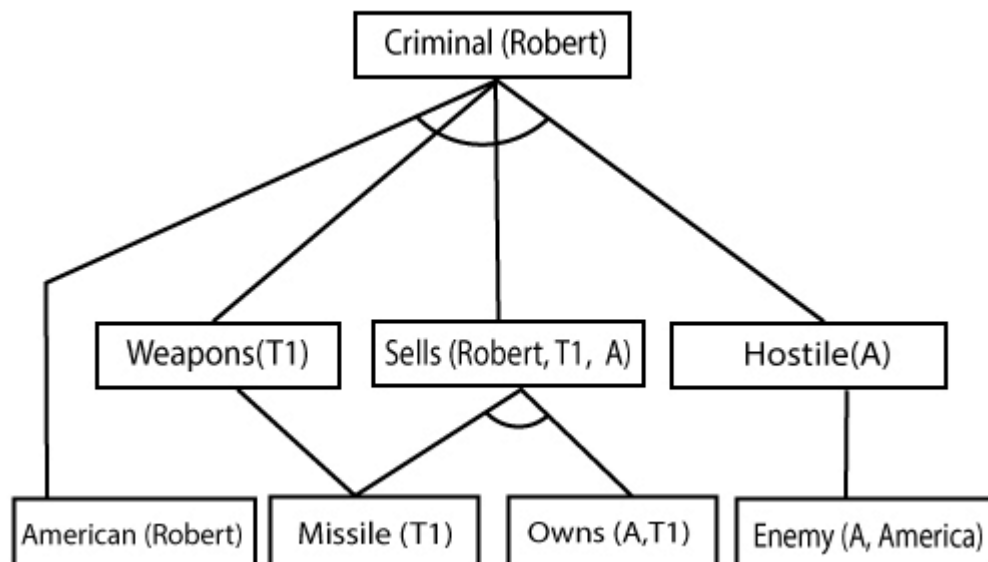
Rule-(4) satisfy with the substitution $\{p/T1\}$, so **Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution $\{p/A\}$, so **Hostile(A)** is added and which infers from Rule-(7).



Step-3:

At step-3, as we can check Rule-(1) is satisfied with the substitution $\{p/\text{Robert}, q/T1, r/A\}$, so we can **add Criminal(Robert)** which infers all the available facts. And hence we reached our goal statement.



Hence it is proved that Robert is Criminal using forward chaining approach.

Backward Chaining:

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

Example:

In backward-chaining, we will use the same above example, and will rewrite all the rules.

- **American (p) \wedge weapon(q) \wedge sells (p, q, r) \wedge hostile(r) \rightarrow Criminal(p) ... (1)**
- **Owens(A, T1) (2)**
- **Missile(T1)**
- **?p Missiles(p) \wedge Owens (A, p) \rightarrow Sells (Robert, p, A) (4)**
- **Missile(p) \rightarrow Weapons (p) (5)**
- **Enemy(p, America) \rightarrow Hostile(p) (6)**
- **Enemy (A, America) (7)**
- **American(Robert). (8)**

Backward-Chaining proof:

In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert)**, and then infer further rules.

Step-1:

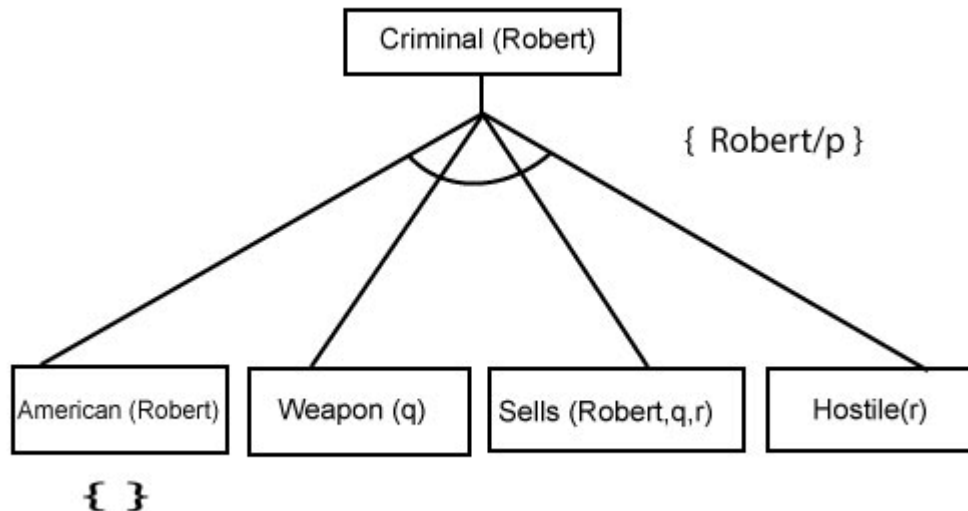
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

Criminal (Robert)

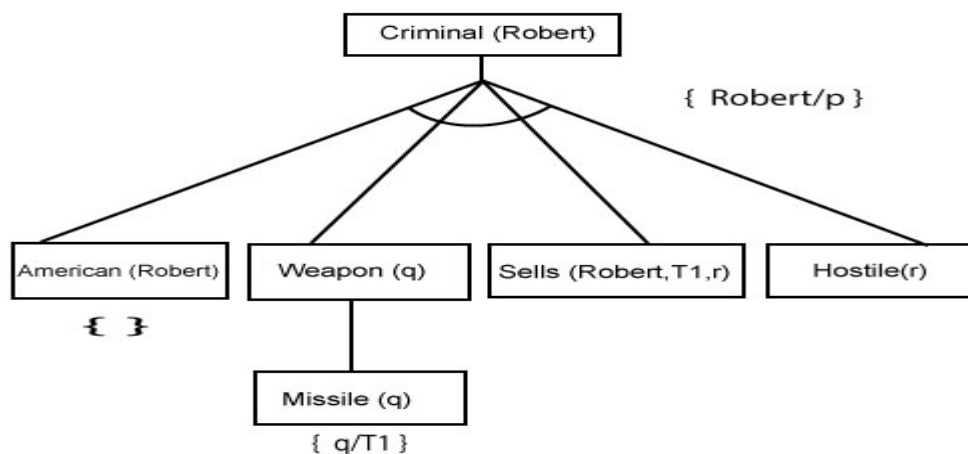
Step-2:

At the second step, we will infer other facts from goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution $\{ \text{Robert}/P \}$. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.



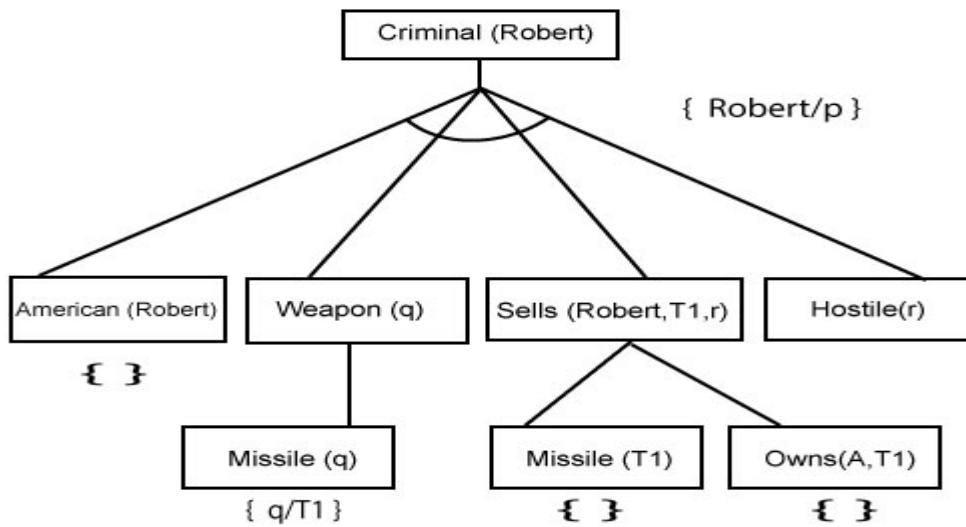
Step-3: At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



Step-4:

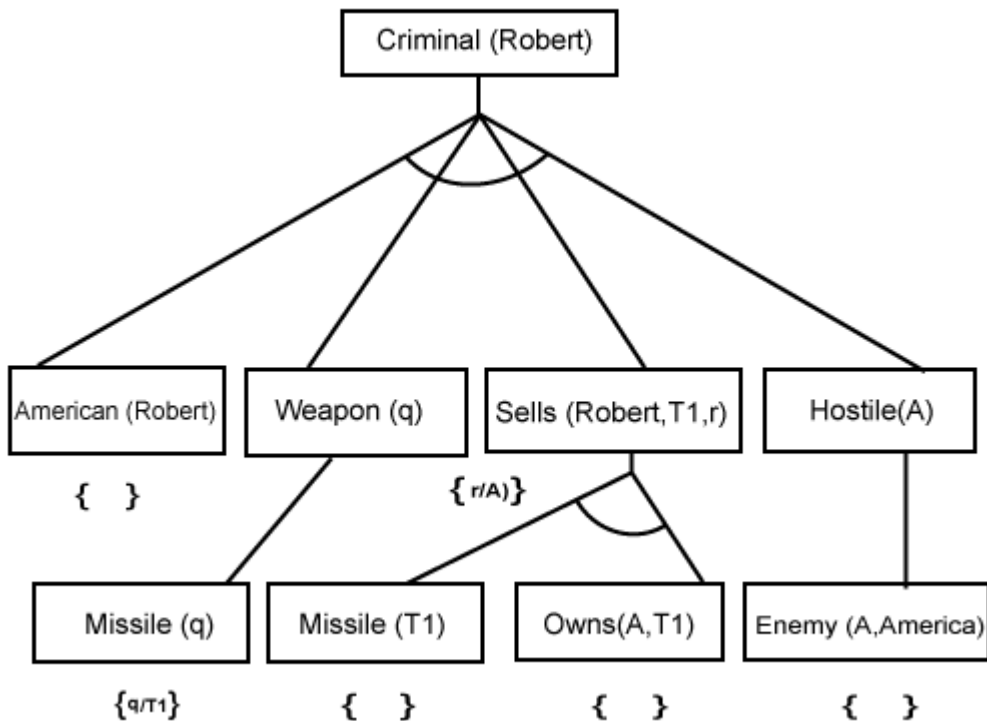
At step-4, we can infer facts Missile(T1) and Owns(A, T1) from Sells(Robert, T1, r) which satisfies

the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.



Step-5:

At step-5, we can infer the fact **Enemy(A, America)** from **Hostile(A)** which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.



Difference between backward chaining and forward chaining

No.	Forward Chaining	Backward Chaining
1	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2	It is a bottom-up approach	It is a top-down approach
3	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.
6	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8	It operates in the forward direction.	It operates in the backward direction.
9	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

Basic probability notation

- **Prior probability** :We will use the notation $P(A)$ for the unconditional or prior probability that the proposition A is true.

- For example, if Cavity denotes the proposition that a particular patient has a cavity, $P(\text{Cavity}) = 0.1$ means that in the absence of any other information, the agent will assign a probability of 0.1(a 10% chance)

- It is important to remember that $P(A)$ can only be used when there is no other information. As soon as some new information B is known, we have to reason with the **conditional probability** of A given B instead of $P(A)$ to the event of the patient's having a cavity.

- Propositions can also include equalities involving so-called random variables.

- For example, if we are concerned about the random variable Weather,

we might have $P(\text{Weather} = \text{Sunny}) = 0.7$

$P(\text{Weather} = \text{Rain}) = 0.2$

$P(\text{Weather} = \text{Cloudy}) = 0.08$

$P(\text{Weather} = \text{Snow}) = 0.02$

Each random variable X has a domain of possible values (x_1, \dots, x_n) that it can take on.

- We can view proposition symbols as random variables as well, if we assume that they have a domain $\{\text{true}, \text{false}\}$.

- Thus, the expression $P(\text{Cavity})$ can be viewed as shorthand for $P(\text{Cavity} = \text{true})$.

- Similarly, $P(\neg \text{Cavity})$ is shorthand for $P(\text{Cavity} = \text{false})$.

- Sometimes, we will want to talk about the probabilities of all the possible values of a random variable. In this case, we will use an expression such as $P(\text{Weather})$

- for example, we would write $P(\text{Weather}) = (0.7, 0.2, 0.08, 0.02)$

This statement defines a probability distribution

- We can also use logical connectives to make more complex sentences and assign probabilities to them.

For example, $P(\text{Cavity} \wedge \neg \text{Insured})$

Conditional probability:

- Once the agent has obtained some evidence concerning the previously unknown propositions making up the domain, prior probabilities are no longer applicable.

Instead, we use conditional or posterior probabilities, with the notation $P(A|B)$

- This is read as "the probability of A given that all we know is B."
- $P(B|A)$ means "Event B given Event A"
- In other words, event A has already happened, now what is the chance of event B?
- $P(B|A)$ is also called the "Conditional Probability" of B given A.

Ex: Drawing 2 Kings from a Deck

- **Event A is drawing a King first, and Event B is drawing a King second.**
- For the first card the chance of drawing a King is 4 out of 52 (there are 4 Kings in a deck of 52 cards):
 - $P(A) = 4/52$
 - But after removing a King from the deck the probability of the 2nd card drawn is less likely to be a King (only 3 of the 51 cards left are Kings):
 - $P(B|A) = 3/51$

And so: $P(A \text{ and } B) = P(A) \times P(B|A) = (4/52) \times (3/51) = 12/2652 = 1/221$

- So the chance of getting 2 Kings is 1 in 221, or about 0.5

BAYES Theorem:

- Bayes' Theorem is a way of finding a probability when we know certain other probabilities.

The formula is

$$P(A|B) = \frac{P(A) P(B|A)}{P(B)}$$

- Which tells us: how often A happens given that B happens, written $P(A|B)$,
- When we know: How often B happens given that A happens, written $P(B|A)$
- and how likely A is on its own, written $P(A)$
- and how likely B is on its own, written $P(B)$

Example:

- Dangerous fires are rare (1%)
- But smoke is fairly common (10%) due to barbecues, and 90% of dangerous fires make smoke

We can then discover the **probability of dangerous Fire when there is Smoke**:

$$P(\text{Fire}|\text{Smoke}) = P(\text{Fire}) P(\text{Smoke}|\text{Fire}) / P(\text{Smoke})$$

$$= 1\% \times 90 / 10\%$$

$$= 9\%$$

So it is still worth checking out any smoke to be sure.

Example 2:

You are planning a picnic today, but the morning is cloudy

Oh no! 50% of all rainy days start off cloudy!

But cloudy mornings are common (about 40% of days start cloudy)

And this is usually a dry month (only 3 of 30 days tend to be rainy, or 10%)

What is the chance of rain during the day?

We will use Rain to mean rain during the day, and Cloud to mean cloudy morning.

The chance of Rain given Cloud is written $P(\text{Rain}|\text{Cloud})$

So let's put that in the formula:

$$P(\text{Rain}|\text{Cloud}) = P(\text{Rain}) P(\text{Cloud}|\text{Rain}) / P(\text{Cloud})$$

$P(\text{Rain})$ is Probability of Rain = 10%

$P(\text{Cloud}|\text{Rain})$ is Probability of Cloud, given that Rain happens = 50%

$P(\text{Cloud})$ is Probability of Cloud = 40%

$$P(\text{Rain}|\text{Cloud}) = 0.1 \times 0.5 / 0.4 = .125$$

Or a 12.5% chance of rain. Not too bad, let's have a picnic!

UNIT-III

Advanced Knowledge Representation and Reasoning: Knowledge Representation Issues, Nonmonotonic Reasoning, Other Knowledge Representation Schemes Reasoning Under Uncertainty: Basic probability, Acting Under Uncertainty, Bayes' Rule, Representing Knowledge in an Uncertain Domain, Bayesian Networks

Artificial intelligence is a system that is concerned with the study of understanding, designing and implementing the ways, associated with knowledge representation to computers.

In any intelligent system, representing the knowledge is supposed to be an important technique to encode the knowledge.

The main objective of AI system is to design the programs that provide information to the computer, which can be helpful to interact with humans and solve problems in various fields which require human intelligence.

What is Knowledge?

Knowledge is an useful term to judge the understanding of an individual on a given subject.

In intelligent systems, domain is the main focused subject area. So, the system specifically focuses on acquiring the domain knowledge.

Issues in knowledge representation

The main objective of knowledge representation is to draw the conclusions from the knowledge, but there are many issues associated with the use of knowledge representation techniques.

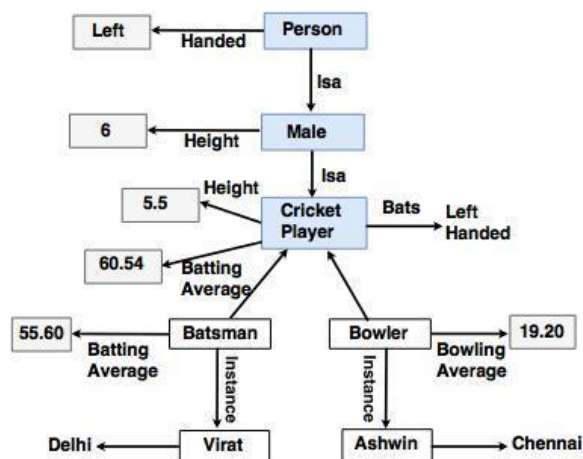


Fig: Inheritable Knowledge Representation

Refer to the above diagram to refer to the following issues.

1. Important attributes

There are two attributes shown in the diagram, instance and isa. Since these attributes support property of inheritance, they are of prime importance.

2. Relationships among attributes

Basically, the attributes used to describe objects are nothing but the entities. However, the attributes of an object do not depend on the encoded specific knowledge.

3. Choosing the granularity of representation

While deciding the granularity of representation, it is necessary to know the following:

- i. What are the primitives and at what level should the knowledge be represented?
- ii. What should be the number (small or large) of low-level primitives or high-level facts?

High-level facts may be insufficient to draw the conclusion while Low-level primitives may require a lot of storage.

For example: Suppose that we are interested in following facts:

John spotted Alex.

Now, this could be represented as "Spotted (agent(John), object (Alex))"

Such a representation can make it easy to answer questions such as: Who spotted Alex?

Suppose we want to know : "Did John see Sue?"

Given only one fact, user cannot discover that answer.

Hence, the user can add other facts, such as "Spotted (x, y) \rightarrow saw (x, y)"

4. Representing sets of objects.

There are some properties of objects which satisfy the condition of a set together but not as individual;

Example: Consider the assertion made in the sentences:

"There are more sheep than people in Australia", and "English speakers can be found all over the world."

These facts can be described by including an assertion to the sets representing people, sheep, and English.

5. Finding the right structure as needed

To describe a particular situation, it is always important to find the access of right structure. This can be done by selecting an initial structure and then revising the choice.

While selecting and reversing the right structure, it is necessary to solve following problem statements.

They include the process on how to:

- Select an initial appropriate structure.
- Fill the necessary details from the current situations.
- Determine a better structure if the initially selected structure is not appropriate to fulfill other conditions.
- Find the solution if none of the available structures is appropriate.
- Create and remember a new structure for the given condition.
- There is no specific way to solve these problems, but some of the effective knowledge representation techniques have the potential to solve them.

Non Monotonic reasoning:

- In Non-monotonic reasoning, some conclusions may be invalidated if we add some more information to our knowledge base.
- Logic will be said as non-monotonic if some conclusions can be invalidated by adding more knowledge into our knowledge base.
- Non-monotonic reasoning deals with incomplete and uncertain models.
- "Human perceptions for various things in daily life, "is a general example of non-monotonic reasoning.

Example: Let suppose the knowledge base contains the following knowledge:

- Birds can fly
- Penguins cannot fly
- Pitty is a bird

So from the above sentences, we can conclude that Pitty can fly.

However, if we add one another sentence into knowledge base "Pitty is a penguin", which concludes "Pitty cannot fly", so it invalidates the above conclusion.

ACTING UNDER UNCERTAINTY

Agents may need to handle uncertainty ,whether due to partial observability,non determinism or combination of two.

Summarizing Uncertainty:

Consider the following Simple rule:

Toothache=> Cavity

Not all the patients with toothaches have cavities ,some of them may have gum disease ,an abscess or

some other problems

Toothache= \Rightarrow cavity \vee Gum Problem \vee Abscess.....

Unfortunately in order to make the rule true we have to add an almost unlimited list of possible problems

Trying to use first-order logic to cope with a domain like medical diagnosis thus fails for three main reasons:

Laziness: It is too much work to list the complete set of antecedents or consequents needed to ensure an exceptionless rule, and too hard to use the enormous rules that result.

Theoretical ignorance: Medical science has no complete theory for the domain.

Practical ignorance: Even if we know all the rules, we may be uncertain about a particular patient because all the necessary tests have not or cannot be run.

The agent's knowledge can at best provide only a degree of belief in the relevant sentences. Our main tool for dealing with degrees of belief will be probability theory, which assigns a numerical degree of belief between 0 and 1 to sentences.

Probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance.

We may not know for sure what afflicts a particular patient, but we believe that there is, say, an 80% chance—that is, a probability of 0.8—that the patient has a cavity if he or she has a toothache

BASIC PROBABILITY NOTATION

Prior probability We will use the notation $P(A)$ for the unconditional or prior probability that the proposition A is true.

For example, if Cavity denotes the proposition that a particular patient has a cavity,

$$P(\text{Cavity}) = 0.1$$

means that in the absence of any other information, the agent will assign a probability of 0.1 (a 10% chance) to the event of the patient's having a cavity.

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

The proposition that is the subject of a probability statement can be represented by a proposition symbol, as in the $P(A)$ example. Propositions can also include equalities involving so-called random variables. For example, if we are concerned about the random variable Weather, we might have

$$P(\text{Weather} = \text{Sunny}) = 0.7$$

$$P(\text{Weather} = \text{Rain}) = 0.2$$

$$P(\text{Weather} = \text{Cloudy}) = 0.08$$

$$P(\text{Weather} = \text{Snow}) = 0.02$$

Each random variable X has a domain of possible values $\{x_1, \dots, x_n\}$ that it can take on

We can view proposition symbols as random variables as well, if we assume that they have a domain $\{\text{true}, \text{false}\}$. Thus, the expression $P(\text{Cavity})$ can be viewed as shorthand for $P(\text{Cavity} = \text{true})$. Similarly, $P(\neg \text{Cavity})$ is shorthand for $P(\text{Cavity} = \text{false})$. Usually, we will use the letters A , B , and so on for Boolean random variables, and the letters X , Y , and so on for multivalued variables.

Sometimes, we will want to talk about the probabilities of all the possible values of a random variable.

In this case, we will use an expression such as $P(\text{Weather})$, which denotes vector of values for the probabilities of each individual state of the weather.

Given the preceding values, for example, we would write $P(\text{Weather}) = (0.7, 0.2, 0.08, 0.02)$

This statement defines a probability distribution for the random variable Weather .

We will also use expressions such as $P(\text{Weather}, \text{Cavity})$ to denote the probabilities of all combinations of the values of a set of random variables.

In this case, $P(\text{Weather}, \text{Cavity})$ denotes a 4×2 table of probabilities. We will see that this notation simplifies many equations. We can also use logical connectives to make more complex sentences and assign probabilities to them. For example, $P(\text{Cavity} \wedge \neg \text{Insured}) = 0.06$ says there is a 6% chance that a patient has a cavity and has no insurance

Conditional probability:

- Once the agent has obtained some evidence concerning the previously unknown propositions making up the domain, prior probabilities are no longer applicable. Instead, we use conditional or posterior probabilities, with the notation $P(A|B)$.
- This is read as "the probability of A given that all we know is B ."

For example, indicates that if a patient is observed to have a toothache, and no other information is yet available,

then the probability of the patient having a cavity will be 0.8.

- It is important to remember that $P(A|B)$ can only be used when all we know is B . As soon as we

know C , then we must compute

➤ $P(A|B \wedge C)$ instead of $P(A|B)$. A prior probability $P(A)$ can be thought of as a special case of conditional probability $P(A|)$, where the probability is conditioned on no evidence.

➤ We can also use the P notation with conditional probabilities. $P(X|Y)$ is a two-dimensional table giving the values of $P(X=x_i|Y=y_j)$ for each possible i, j . Conditional probabilities can be defined in terms of unconditional probabilities. The equation

$$P(A|B) = \frac{P(A \wedge B)}{P(B)} \quad (14.1)$$

holds whenever $P(B) > 0$. This equation can also be written as

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

JCT RULE

which is called the **product rule**. The product rule is perhaps easier to remember: it comes from the fact that for A and B to be true, we need B to be true, and then A to be true given B . We can also have it the other way around:

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

In some cases, it is easier to reason in terms of prior probabilities of conjunctions, but for the most part, we will use conditional probabilities as our vehicle for probabilistic inference.

We can also extend our P notation to handle equations like these, providing a welcome degree of conciseness. For example, we might write

$$P(X, Y) = P(X|Y)P(Y)$$

Axioms of Probability:

➤ All probabilities are between 0 and 1.

$$0 < P(A) < 1$$

➤ Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0. $P(\text{True}) = 1$ $P(\text{False}) = 0$

➤ The probability of a disjunction is given by $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$

The joint probability distribution

The joint probability distribution (or "joint" for short), which completely specifies an agent's probability assignments to all propositions in the domain (both simple and complex).

A probabilistic model of a domain consists of a set of random variables that can take on particular values with certain probabilities. Let the variables be $X_1 \dots X_n$.

An atomic event is an assignment of particular values to all the variables—in other words, a complete specification of the state of the domain

The joint probability distribution $P(X_1, \dots, X_n)$ assigns probabilities to all possible atomic events. Recall that $P(X_i)$ is a one-dimensional vector of probabilities for the possible values of the variable X_i . Then

the joint is an w-dimensional table with a value in every cell giving the probability of that specific state occurring. Here is a joint probability distribution for the trivial medical domain consisting of the two Boolean variables Toothache and Cavity:

	<i>Toothache</i>	\neg <i>Toothache</i>
<i>Cavity</i>	0.04	0.06
\neg <i>Cavity</i>	0.01	0.89

Adding across a row or column gives the unconditional probability of a variable, for example, $P(\text{Cavity}) = 0.06 + 0.04 = 0.10$.

$$P(\text{Cavity} \vee \text{Toothache}) = 0.04 + 0.01 + 0.06 = 0.11$$

$$P(\text{Cavity}|\text{Toothache}) = \frac{P(\text{Cavity} \wedge \text{Toothache})}{P(\text{Toothache})} = \frac{0.04}{0.04 + 0.01} = 0.80$$

Bayes Rule:

Recall the two forms of the product rule:

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

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

Equating the two right-hand sides and dividing by $P(A)$, we get

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \quad (14.3)$$

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem).⁷ This simple equation underlies all modern AI systems for probabilistic inference. The more general case of multivalued variables can be written using the P notation as follows:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

where again this is to be taken as representing a set of equations relating corresponding elements of the tables. We will also have occasion to use a more general version conditionalized on some background evidence E :

$$P(Y|X, E) = \frac{P(X|Y, E)P(Y|E)}{P(X|E)} \quad (14.4)$$

Representing knowledge in uncertain domain

In the context of using Bayes' rule, conditional independence relationships among variables can simplify the computation of query results and greatly reduce the number of conditional probabilities that need to be specified. We use a data structure called a belief BELIEF NETWORK network' to represent the dependence between variables and to give a concise specification of the joint probability distribution.

A Bayesian Network Is a directed graph in which each node is annotated with quantitative probability information.

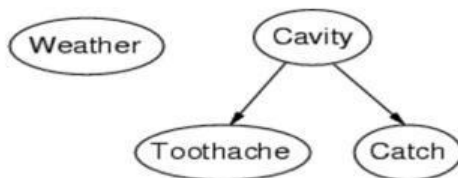
The full specification is as follows:

1. Each node corresponds to a random variable, which can be discrete or continuous.
2. If a set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y , X is said to be parent of Y . The graph has no directed cycles and hence it is called directed acyclic graph (DAG)
3. Each node X_i has a conditional probability distribution $P(X_i | \text{Parents}(X_i))$ that quantifies the effect of the parents on the node.

The intuitive meaning of an arrow from node X to node Y is that X has a direct influence on Y

Example

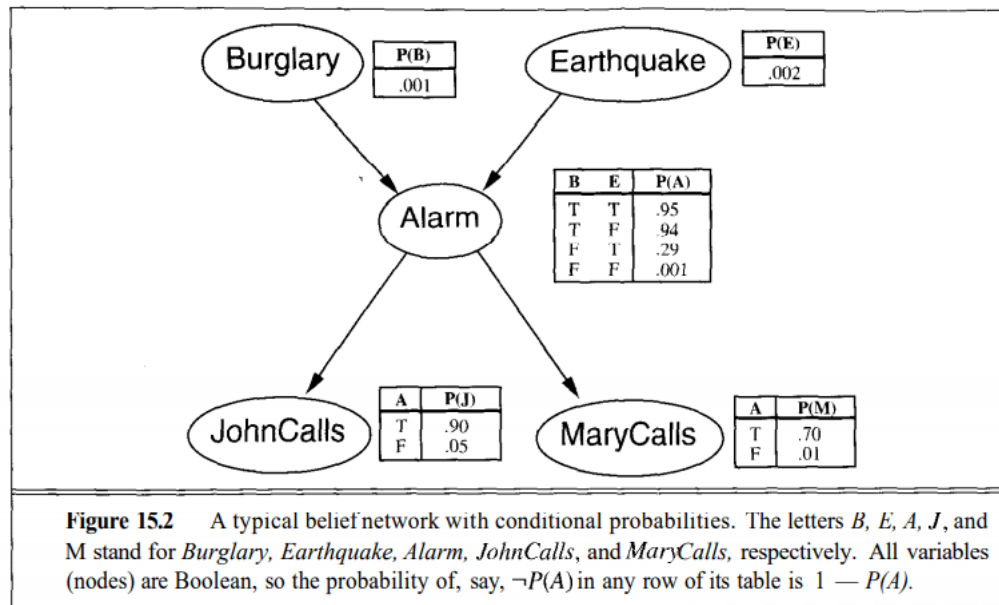
- Topology of network encodes conditional independence assertions:



- *Weather* is independent of the other variables
- *Toothache* and *Catch* are conditionally independent given *Cavity*

Consider the following situation. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. (This example is due to Judea Pearl, a resident of Los Angeles; hence the acute interest in earthquakes.) You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and sometimes misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

This simple domain is described by the belief network in Figure 15.2



Notice that the network does not have nodes corresponding to Mary currently listening to loud music, or to the telephone ringing and confusing John. These factors are summarized in the uncertainty associated with the links from Alarm to JohnCalls and MaryCalls.

This shows both laziness and ignorance in operation: it would be a lot of work to determine any reason why those factors would be more or less likely in any particular case, and we have no reasonable way to obtain the relevant information anyway.

The probabilities actually summarize a potentially infinite set of possible circumstances in which the alarm might fail to go off (high humidity, power failure, dead battery, cut wires, dead mouse stuck inside bell,...) or John or Mary might fail to call and report it (out to lunch, on vacation, temporarily deaf, passing helicopter, ...). In this way, a small agent can cope with a very large world, at least approximately. The degree of approximation can be improved if we introduce additional relevant information.

Bayesian belief network

Bayesian belief network is key computer technology for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their conditional dependencies using a directed acyclic graph."

It is also called a Bayes network, belief network, decision network, or Bayesian model.

Bayesian networks are probabilistic, because these networks are built from a probability distribution, and also use probability theory for prediction and anomaly detection

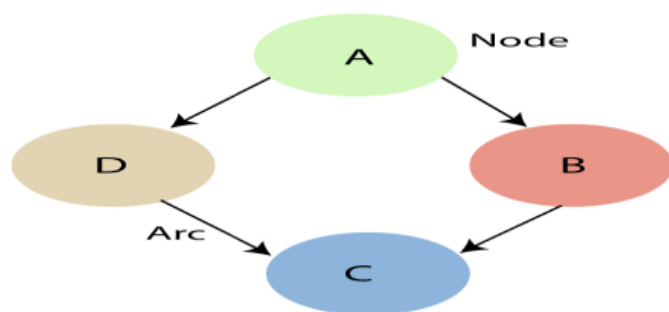
Bayesian Network can be used for building models from data and experts opinions, and it consists of two parts:

Directed Acyclic Graph

Table of conditional probabilities.

The generalized form of Bayesian network that represents and solve decision problems under uncertain knowledge is known as an Influence diagram.

A Bayesian network graph is made up of nodes and Arcs (directed links), where:



- Each node corresponds to the random variables, and a variable can be continuous or discrete.
- Arc or directed arrows represent the causal relationship or conditional probabilities between random variables. These directed links or arrows connect the pair of nodes in the graph. These links represent that one node directly influence the other node, and if there is no directed link that means that nodes are independent with each other
- In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.
- If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.
- Node C is independent of node A.

Each node in the Bayesian network has condition probability distribution $P(X_i | \text{Parent}(X_i))$, which determines the effect of the parent on that node.

Bayesian network is based on Joint probability distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables $x_1, x_2, x_3, \dots, x_n$, then the probabilities of a different combination of $x_1, x_2, x_3 \dots x_n$, are known as Joint probability distribution.

$P[x_1, x_2, x_3, \dots, x_n]$, it can be written as the following way in terms of the joint probability distribution.

$$= P[x_1 | x_2, x_3, \dots, x_n] P[x_2, x_3, \dots, x_n]$$

$$= P[x_1 | x_2, x_3, \dots, x_n] P[x_2 | x_3, \dots, x_n] \dots P[x_{n-1} | x_n] P[x_n].$$

In general for each variable X_i , we can write the equation as:

$$P(X_i | X_{i-1}, \dots, X_1) = P(X_i | \text{Parents}(X_i))$$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm. David always calls Harry when he hears the alarm, but sometimes he got confused with the phone ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

The Bayesian network for the above problem is given below. The network structure is showing that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.

The network is representing that our assumptions do not directly perceive the burglary and also do not notice the minor earthquake, and they also not confer before calling.

The conditional distributions for each node are given as conditional probabilities table or CPT.

Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.

In CPT, a boolean variable with k boolean parents contains 2^k probabilities. Hence, if there are two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

Burglary (B)

Earthquake(E)

Alarm(A)

David Calls(D)

Sophia calls(S)

We can write the events of problem statement in the form of probability: $P[D, S, A, B, E]$, can rewrite the above probability statement using joint probability distribution:

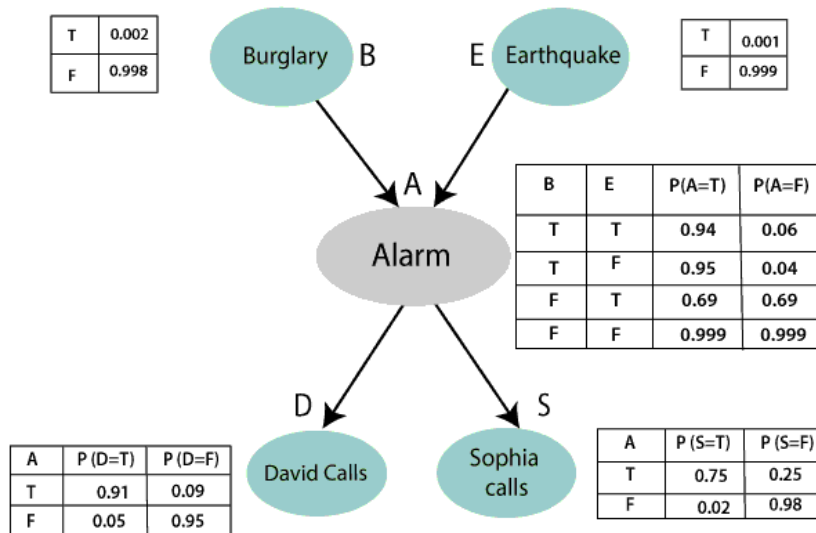
$$P[D, S, A, B, E] = P[D | S, A, B, E] \cdot P[S, A, B, E]$$

$$= P[D | S, A, B, E] \cdot P[S | A, B, E] \cdot P[A, B, E]$$

$$P[D | A] \cdot P[S | A, B, E] \cdot P[A, B, E]$$

$$= P[D | A] \cdot P[S | A] \cdot P[A | B, E] \cdot P[B, E]$$

$$= P[D | A] \cdot P[S | A] \cdot P[A | B, E] \cdot P[B | E] \cdot P[E]$$



Let's take the observed probability for the Burglary and earthquake component:

$P(B = \text{True}) = 0.002$, which is the probability of burglary.

$P(B = \text{False}) = 0.998$, which is the probability of no burglary.

$P(E = \text{True}) = 0.001$, which is the probability of a minor earthquake

$P(E = \text{False}) = 0.999$, Which is the probability that an earthquake not occurred.

Conditional probability table for Alarm A:

The Conditional probability of Alarm A depends on Burglar and earthquake:

B	E	P(A= True)	P(A= False)
True	True	0.94	0.06
True	False	0.95	0.04
False	True	0.31	0.69
False	False	0.001	0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A	P(D= True)	P(D= False)
True	0.91	0.09
False	0.05	0.95

Conditional probability table for Sophia Calls:

The Conditional probability of Sophia that she calls is depending on its Parent Node "Alarm."

A	P(S= True)	P(S= False)
True	0.75	0.25
False	0.02	0.98

From the formula of joint distribution, we can write the problem statement in the form of probability distribution:

$$P(S, D, A, \neg B, \neg E) = P(S|A) * P(D|A) * P(A|\neg B \wedge \neg E) * P(\neg B) * P(\neg E).$$

$$= 0.75 * 0.91 * 0.001 * 0.998 * 0.999$$

from the formula of joint distribution, we can write the problem statement in the form of probability distribution:

$$P(S, D, A, \neg B, \neg E) = P(S|A) * P(D|A) * P(A|\neg B \wedge \neg E) * P(\neg B) * P(\neg E).$$

$$= 0.75 * 0.91 * 0.001 * 0.998 * 0.999$$

.

UNIT-IV

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

What is learning?

Most often heard criticisms of AI is that machines cannot be called intelligent until they are able to learn to do new things and adapt to new situations, rather than simply doing as they are told to do.

Some critics of AI have been saying that computers cannot learn!

Definitions of Learning: changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

- Learning covers a wide range of phenomenon:
- Skill refinement: Practice makes skills improve. More you play tennis, better you get
- Knowledge acquisition: Knowledge is generally acquired through experience

Various learning mechanisms:

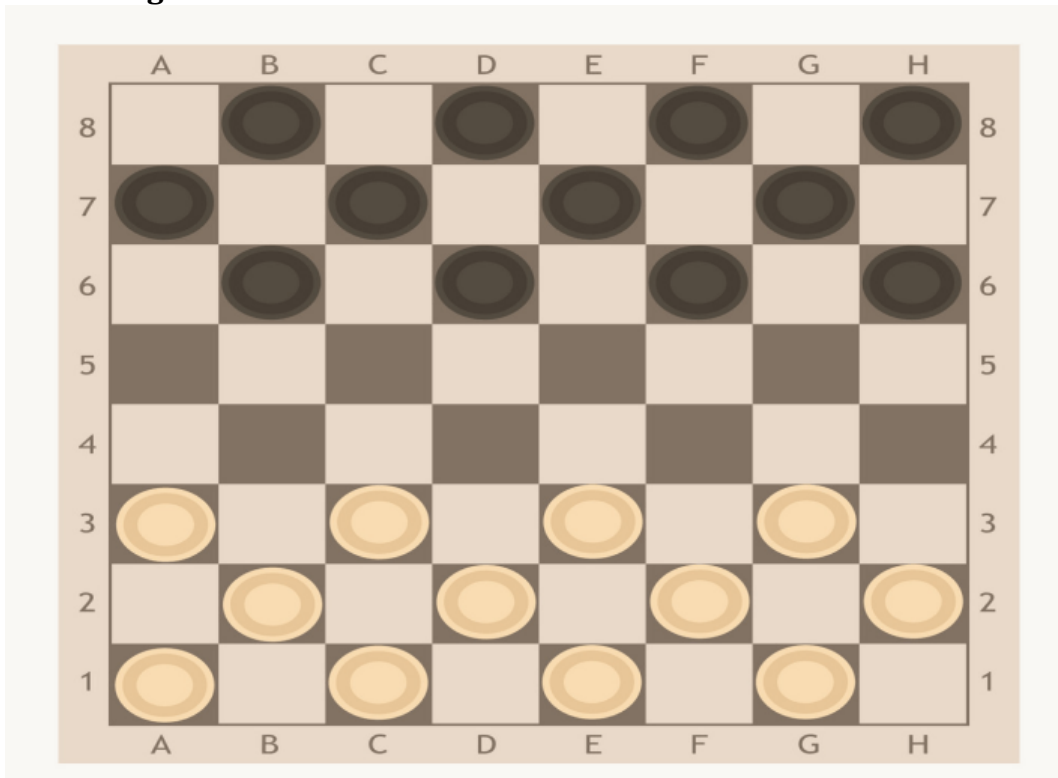
- Simple storing of computed information or rote learning, is the most basic learning activity.
- Many computer programs i.e., database systems can be said to learn in this sense although most people would not call such simple storage learning.
- Another way we learn is through taking advice from others. Advice taking is similar to rote learning, but high-level advice may not be in a form simple enough for a program to use directly in problem solving.
- People also learn through their own problem-solving experience.
- Learning from examples: we often learn to classify things in the world without being given explicit rules.
- Learning from examples usually involves a teacher who helps us classify things by correcting us when we are wrong.

Rote learning:

- Rote Learning is basically memorisation.
- Saving knowledge so it can be used again.
- Retrieval is the only problem.

- No repeated computation, inference or query is necessary.
- A simple example of rote learning is caching
- Store computed values (or large piece of data)
- Recall this information when required by computation.
- Significant time savings can be achieved.
- Many AI programs (as well as more general ones) have used caching very effectively.

Checkers game:



- Samuel's Checkers program employed rote learning (it also used parameter adjustment which will be discussed shortly).
- A minimax search was used to explore the game tree.
- Time constraints do not permit complete searches.
- It records board positions and scores at search ends.
- Now if the same board position arises later in the game the stored value can be recalled and the end effect is that deeper searched have occurred.

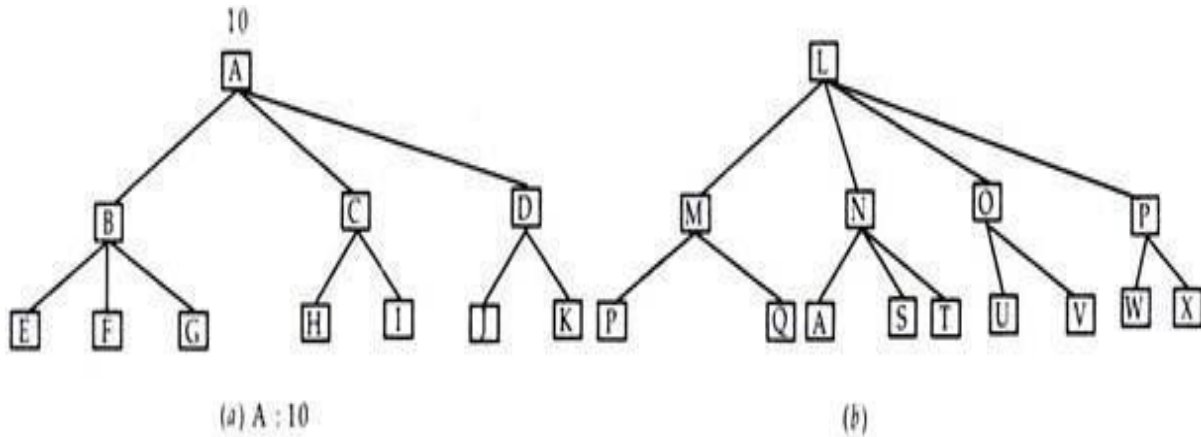


Fig. 9.2. Considering stored value of A explaining rote learning.

- Rote learning is basically a simple process. However it does illustrate some issues that are relevant to more complex learning issues.

Organisation

- -- access of the stored value must be faster than it would be to recompute it. Methods such as hashing, indexing and sorting can be employed to enable this.
- E.g Samuel's program indexed board positions by noting the number of pieces.

Generalisation

- -- The number of potentially stored objects can be very large. We may need to generalise some information to make the problem manageable.

E.g Samuel's program stored game positions only for white to move. Also rotations along diagonals are combined

Learning by taking advice:

This is a simple form of learning. Suppose a programmer writes a set of instructions to instruct the computer what to do, the programmer is a teacher and the computer is a student.

Once learned (i.e. programmed), the system will be in a position to do new things.

The advice may come from many sources: human experts, internet to name a few. This type of learning requires more inference than rote learning.

The knowledge must be transformed into an operational form before stored in the knowledge base.

- FOO (First Operational Operationaliser), for example, is a learning system which is used to learn the game of Hearts.
- It converts the advice which is in the form of principles, problems, and methods into effective executable (LISP) procedures (or knowledge). Now this knowledge is ready to use.
- A human user first translates the advice from English into a representation

That foo can understand

For eg: "Avoid taking points"

Avoid(take points me)(trick)

Achieve (not(during (trick)(take point-me))))

Learning in Problem Solving- learning by Parameter Adjustment

Many programs rely on an evaluation procedure to summarise the state of search etc. Game playing programs provide many examples of this.

However, many programs have a static evaluation function to get a score that achieves the desirable board position.

In learning a slight modification of the formulation of the evaluation of the problem is required.

Here the problem has an evaluation function that is represented as a polynomial of the form such as:

$$c_1t_1 + c_2t_2 + c_3t_3 + \dots$$

The 't' terms are the values that contribute to the evaluation. The 'C' terms are the coefficients (weights) that are attached to these values.

- But many moves must have contributed to that final outcome, Even if the program wins it may have made some wrong moves along the way
- Because of the limitations Samuel program did two things:
- When the program is in learning mode paly against the copy of itself, At the end of the game if the modified function won then the modified version is accepted otherwise the old one is retained.
- Periodically,one term in the scoring function was eliminated and replaced by another.

Learning in Problem Solving-Learning with macro operators:

- The basic idea here is similar to Rote Learning: Avoid expensive recomputation
- Macro-operators can be used to group a whole series of actions into one.
- For example: Making dinner can be described as lay the table, cook dinner, serve dinner. We could treat laying the table as one action even though it involves a sequence of actions.
- The STRIPS problem-solving employed macro-operators in its learning phase.
- Consider a blocks world example in which $ON(C,B)$ and $ON(A,TABLE)$ are true.
- STRIPS can achieve $ON(A,B)$ in four steps:

UNSTACK(C,B), PUTDOWN(C), PICKUP(A),
STACK(A,B)

STRIPS now builds a macro-operator MACROP with preconditions $ON(C,B)$, $ON(A,TABLE)$, postconditions $ON(A,B)$, $ON(C,TABLE)$ and the four steps as its body.

MACROP can now be used in future operation.

But it is not very general. The above can be easily generalised with variables used in place of the blocks.

However generalisation is not always that easy

Non Serializable subgoals:

- Non serializability means that working on one subgoal will necessarily interfere with previous solution to another subgoal
- Macro operators can be used for games like 8-Puzzle (for ex we have correctly placed 4 tiles and our job is to put fifth without disturbing the earlier tiles).
- A macro will not disturb 4 files externally (but within the macro tiles are disturbed).

Learning in Problem Solving-Learning from chunking:

- Chunking is similar to learning with macro-operators. Generally, it is used by problem solver systems that make use of production systems.
- A production system consists of a set of rules that are in if-then form. That is given a particular situation, what are the actions to be performed. For example, if it is raining then take umbrella
- To solve a problem, a system will compare the present situation with the left hand side of the rules. If there is a match then the system will perform the actions described in the right hand

side of the corresponding rule.

- Problem solvers solve problems by applying the rules. Some of these rules may be more useful than others and the results are stored as a chunk. Chunking can be used to learn general search control knowledge
- Several chunks may encode a single macro-operator and one chunk may participate in a number of macro sequences. Chunks learned in the beginning of problem solving, may be used in the later stage. The system keeps the chunk to use it in solving other problems.
- Soar is a general cognitive architecture for developing intelligent systems. Soar requires knowledge to solve various problems. It acquires knowledge using chunking mechanism
- An impasse arises when the system does not have sufficient knowledge. Consequently, Soar chooses a new problem space (set of states and the operators that manipulate the states) in a bid to resolve the impasse.
- While resolving the impasse, the individual steps of the task plan are grouped into larger steps known as chunks.
- The chunks decrease the problem space search and so increase the efficiency of performing the task.
- in Soar, the knowledge is stored in long-term memory. Soar uses the chunking mechanism to create productions that are stored in long-term memory.
- A chunk is nothing but a large production that does the work of an entire sequence of smaller ones.
- The productions have a set of conditions or patterns to be matched to working memory which consists of current goals, problem spaces, states and operators and a set of actions to perform when the production fires
- Chunks are generalized before storing. When the same impasse occurs again, the chunks so collected can be used to resolve it.

Learning from Examples-Induction:

- Classification is a process of assigning to a particular input, to the name of the class to which it belongs to. Classification is important component in many problem solving tasks.
- But often classification is embedded inside another operation.

For eg:

- If: the current goal is to get from place A to place B and there is a wall separating two places

Then look for a Doorway in the wall and through it.

- To use this rule successfully, the system's matching routine must be able to identify an object as a wall. Without this the rule can never be invoked.
- Then to apply the rule, the system must be able to recognize the a doorway.
- Before classification is done , the classes it will use must be defined . This can be done in variety of ways:
- Isolate a set of features that are relevent to task domain. Define each class by some values of these features.

Eg: for weather predictions the parameters can be of rainfall,sunny,cloudy

- Isolate a set of features that are relevant to the task domain. Define a class as a structure composed of those features.

For example if the task is to identify animals, the body of each type of animal can be stored as structure and various features like color, length of a neck can be represented. The task of constructing class definitions is called concept learning or Induction.

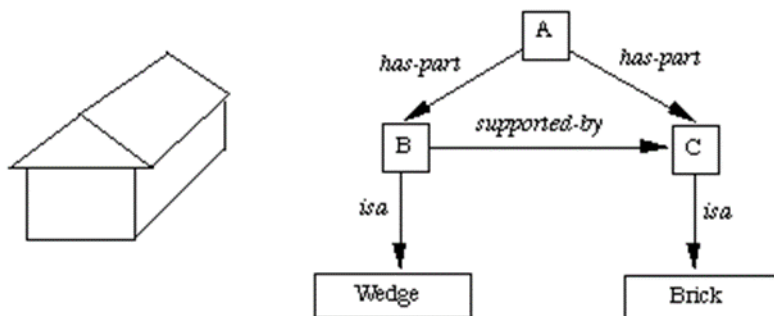
Let us the learn the techniques to define classes structurally.

Winston's Learning Program:

- Winston describes an early structural concept learning program.

Its goal is to construct representations of the definitions of concepts in the blocks domain.

For eg : it learned the concepts, House tent and Arch







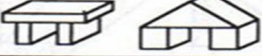

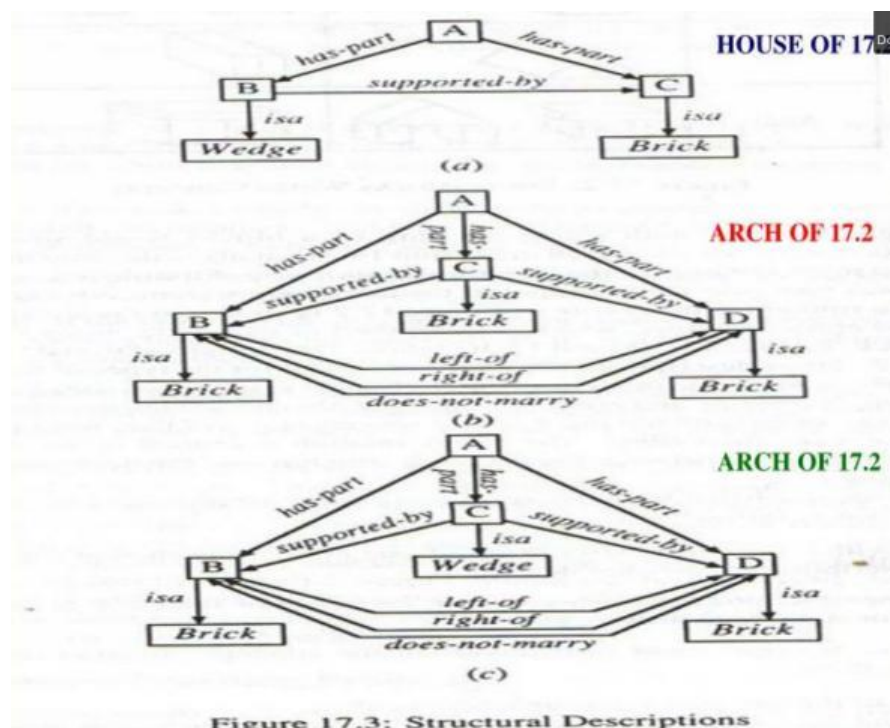
	Concept	Near Miss
House		
Tent		
Arch		

Figure 17.2: Some Blocks World Concepts



To objects marry if they have faces that touch each and they have a common edge.

The marry relation is critical in the definition of arch. It is the difference between the first arch and near miss arch structure. In fig 17.2

Winston Learning program

- Winston's program followed 3 basic steps in concept formulation:
 - Select one known instance of the concept.
Call this the **concept definition**.
 - Examine definitions of other known instance of the concept.
Generalize the definition to include them.
 - Examine descriptions of **near misses**.
Restrict the definition to **exclude** these.
- Both steps 2 and 3 of this procedure rely heavily on comparison process by which similarities and differences between structures can be detected.
- Winston's program can be similarly applied to learn other concepts such as "ARCH".

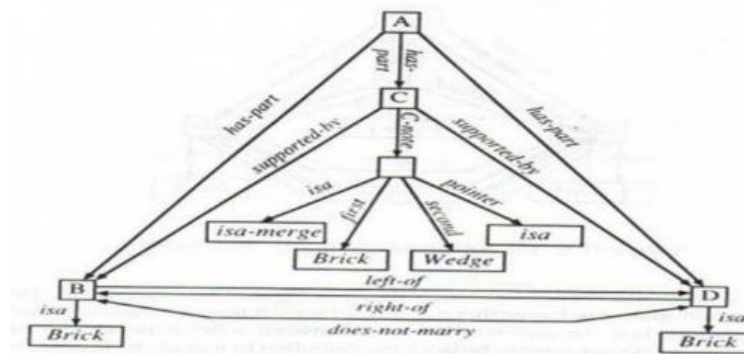


Figure 17.4: The Comparison of Two Arches

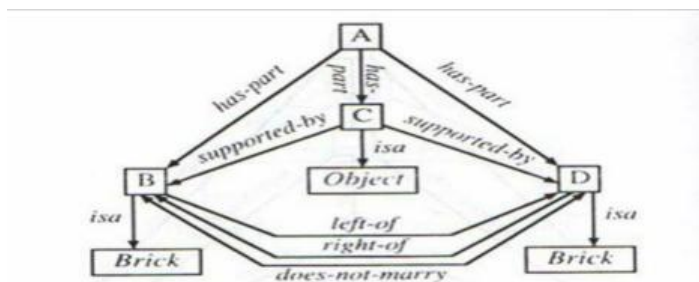


Figure 17.5: The Arch Description after Two Examples

Decision Trees:

A third approach to concept learning is the induction of *decision trees*, as exemplified by the ID3 program of Quinlan [1986]. ID3 uses a tree representation for concepts, such as the one shown in Fig. 17.13. To classify a particular input, we start at the top of the tree and answer questions until we reach a leaf, where the classification is stored. Fig. 17.13 represents the familiar concept "Japanese economy car." ID3 is a program that builds decision trees automatically, given positive and negative instances of a concept.⁴

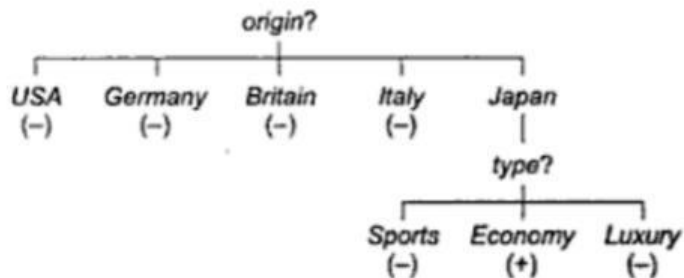


Fig. 17.13 A Decision Tree

ID3 uses an iterative method to build up decision trees, preferring simple trees over complex ones, on the theory that simple trees are more accurate classifiers of future inputs. It begins by choosing a random subset of the training examples. This subset is called the *window*. The algorithm builds a decision tree that correctly classifies all examples in the window. The tree is then tested on the training examples outside the window. If all the examples are classified correctly, the algorithm halts. Otherwise, it adds a number of training examples to the window and the process repeats. Empirical evidence indicates that the iterative strategy is more efficient than considering the whole training set at once.

So how does ID3 actually construct decision trees? Building a node means choosing some attribute to test. At a given point in the tree, some attributes will yield more information than others. For example, testing the attribute *color* is useless if the color of a car does not help us to classify it correctly. Ideally, an attribute will separate training instances into subsets whose members share a common label (e.g., positive or negative). In that case, branching is terminated, and the leaf nodes are labeled.

There are many variations on this basic algorithm. For example, when we add a test that has more than two branches, it is possible that one branch has no corresponding training instances. In that case, we can either leave the node unlabeled, or we can attempt to guess a label based on statistical properties of the set of instances being tested at that point in the tree. Noisy input is another issue. One way of handling noisy input is to avoid building new branches if the information gained is very slight. In other words, we do not want to overcomplicate the tree to account for isolated noisy instances. Another source of uncertainty is that attribute values may be unknown. For example a patient's medical record may be incomplete. One solution is to guess the correct branch to take; another solution is to build special "unknown" branches at each node during learning.

When the concept space is very large, decision tree learning algorithms run more quickly than their version space cousins. Also, disjunction is more straightforward. For example, we can easily modify Fig. 17.13 to represent the disjunctive concept "American car or Japanese economy car," simply by changing one of the negative (—) leaf labels to positive (+). One drawback to the ID3 approach is that large, complex decision trees can be difficult for humans to understand, and so a decision tree system may have a hard time explaining the reasons for its classifications.

UNIT-V

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

What is an Expert System?

An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert..

It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.

The system helps in decision making for complex problems using both facts and heuristics like a human expert.

It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain.

These systems are designed for a specific domain, such as medicine, science, etc.

The performance of an expert system is based on the expert's knowledge stored in its knowledge base.

The more knowledge stored in the KB, the more that system improves its performance.

One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.

Examples of the Expert System:

MYCIN: It was one of the earliest backward chaining expert systems that was designed to find the bacteria causing infections like bacteraemia and meningitis. It was also used for the recommendation of antibiotics and the diagnosis of blood clotting diseases.

PXDES: It is an expert system that is used to determine the type and level of lung cancer. To determine the disease, it takes a picture from the upper body, which looks like the shadow. This shadow identifies the type and degree of harm.

CaDeT: The CaDet expert system is a diagnostic support system that can detect cancer at early stages

Representing and using Domain knowledge:

The R1 program internally called XCON, for eXpert CONfigurer was a production-rule-based system to

assist in the ordering of DEC's VAX computer systems by automatically selecting the computer system components based on the customer's requirements.

It eventually had about 2500 rules.

Rule of Xcon that configures DEC VAX system

If: the most current active context is distributing mass bus devices and

There is a single-port disk drive that has not been assigned to a massbus and

There are no unassigned dual port disk drives and the number of devices that each mass bus should support is known and,

There is a mass bus that has been assigned at least one disk drive and that should support additional disk drives,

And the type of the cable needed to connect the disk drive to the previous device on the mass bus is known

Then: assign the disk drive to the massbus

- As RI is doing a design task (in contrast to the diagnosis task performed by MYCIN)it is not necessary to consider all the possible alternatives one good one is enough. As a result probabilistic information is not necessary in R1;

PROSPECTOR is a program that provides advice on mineral exploration. It's rule looks like this:

IF: magnetite and pyrite is disseminated or veinlet form is present

Then(2,-4) there is a favourable mineralization and texture for the propylitic stage

Here each rule contains two estimates.

The first indicates that the presence of evidence described in the condition part of the rule suggests the validity of the rules conclusion

The second measures the extent to which the evidence is necessary to the validity of the conclusion.

2 indicates the presence of the evidence is encouraging..

-4 indicates that the absence of the evidence is slightly discouraging

Reasoning with knowledge

- Expert systems exploit many of the representation and reasoning mechanisms that we have discussed.

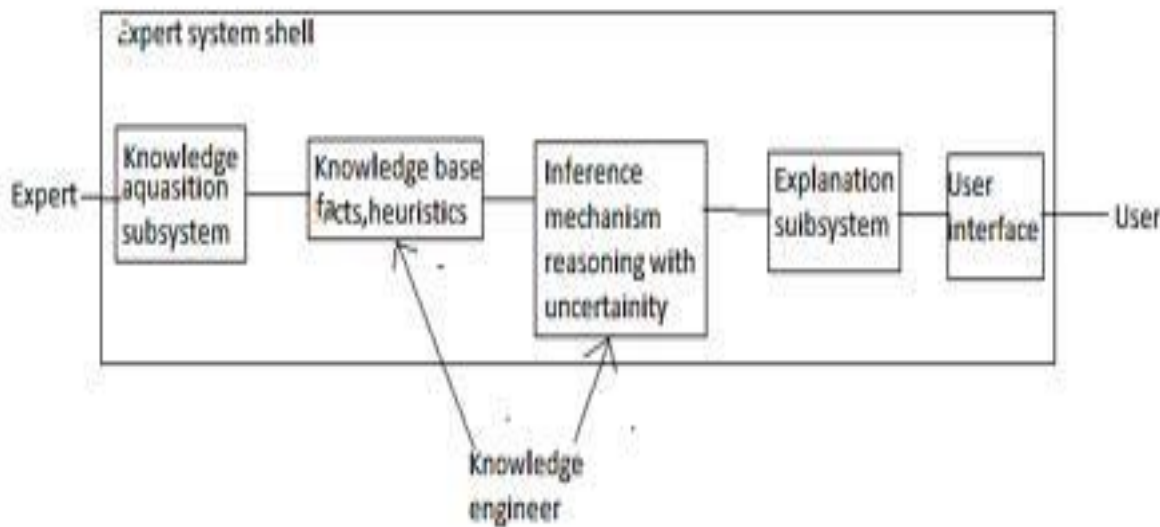
- Because these programs are usually written primarily as rule based systems, forward chaining and backward chaining are usually used .

■ For ex: MYCIN used backward chaining to discover what organisms are present. And then uses forward chaining to reason from the organisms to a treatment regime..

■ RI on the other hand uses Forward chaining.

Expert system Shells

A new expert system can be developed by adding domain knowledge to the shell. The figure depicts generic components of expert system.



- Knowledge acquisition system: It is the first and fundamental step. It helps to collect the experts knowledge required to solve the Problems and build the knowledge base.
- Knowledge Base: This component is the heart of expert systems. It stores all factual and heuristic knowledge about the application domain. It provides with the various representation techniques for all the data.
- Inference mechanism: Inference engine is the brain of the expert system. This component is mainly responsible for generating inference from the given knowledge from the knowledge base and produce line of reasoning in turn the result of the user's query.
- Explanation subsystem: This part of shell is responsible for explaining or justifying the final or intermediate result of user query. It is also responsible to justify need of additional knowledge
- User interface: It is the means of communication with the user. It decides the utility of expert system.

- Building expert systems by using shells has significant advantages. It is always advisable to use shell to develop expert system as it avoids building the system from scratch.
- To build an expert system using system shell, one needs to enter all necessary knowledge about a task domain into the shell.

Explanation:

An expert system is said to be effective when people can interact with it easily.

To facilitate the interaction ,the expert system must have the following two properties:

1. Explain its reasoning: In many of the domains in which experts system operate ,people will not accept results unless they have been convinced of the accuracy of the of the reasoning process that produced those results.

An expert system is said to be effective when people can interact with it easily.

2. Acquire new knowledge and modifications of old knowledge: since expert systems derive their power from the richness of the knowledge bases they is exploit it ,it is extremely important that those knowledge bases be complete and as accurate as possible

One way to get the knowledge into a program is through interaction with the human expert. Or to have a program that learns the expert behaviour from raw data.

- TEIRESIAS was the first program to support explanation and knowledge acquisition.

TEIRESIAS served as a front end for the MYCIN expert system.

The program asks for a piece information that it needs in order to continue its reasoning

The doctor wants to know why the program wants the information and later asks the how the program arrived at a conclusion that it claimed had reached

- Mycirm attempts to solve its goal of recommending a therapy for a particular patient by first finding the cause of the patient's illness.
 - It uses its production rules to reason backward from goals to clinical observations.
 - To solve the top level diagnostic goal, it looks for rules whose right side suggests diseases.
 - It then uses left sides of those rules(preconditions) to set up subgoals .
 - These subgoals are again matched against rules and their preconditions are used to set up additional goals.
 - Whenever a precondition specifies a specific piece of clinical evidence ,mycin uses that evidence,

otherwise it asks the user to provide the information.

- The actual goal that MYCIN set up are more general than the they need to specify the preconditions of a individual rule.

For ex:

- If a precondition satisfies that the identity of a organism X , MYCIN will set up the goal “infer identity”
- The first Question that the user asks is WHY? Why do you need to know that?
- Because the clinical tests are either expensive or dangerous..
- It is important for the doctor to be convinced that the information is really needed before ordering the test.

Because MYCIN is reasoning backward the question can be easily answered by examining the goal tree.

- The user can ask the question How did you know that?
- The question can be answered by looking at the goal tree and chaining backward from the stated fact to the evidence that allowed a rule that determined the fact to fire.

Knowledge Acquisition:

- How are experts system built?

Knowledge Engineer Interviews domain experts and to get the clear knowledge and the they are translated into rules- This process is expensive and time consuming.

- Look for Automatic ways of constructing expert knowledge bases, but no automatic knowledge acquisition systems exist yet.

- But there are programs that interact with domain experts to extract knowledge efficiently.

- These programs provides supports for the following activities:

1. Entering knowledge
2. maintaining Knowledge base consistency
3. Ensuring Knowledge base completeness.

- The most useful knowledge acquisition programs are those that are restricted to a particular problem solving paradigm eg: diagnosis or design.

- If the paradigm is diagnosis then the program can structure its knowledge base around symptoms, hypothesis and causes.

- It can identify symptoms for which the expert system has not yet provided causes.
- Since one system have many multiple causes the program can ask for knowledge about how to decide when one hypothesis is better than another.
- MOLE is a knowledge acquisition system for heuristic classification problems, such as diagnosing diseases.
- It used in conjunction with COVER AND DIFFERENTIATE problem solving method.
- An Expert system produced by MOLE accepts input data ,comes up with a set of candidate explanations or classifications that cover(explain) the data., the uses differentiating knowledge to determine which one is best.
- MOLE interacts with the human expert to produce a knowledge base that a system called MOLE-p(performance) uses to solve problems

The acquisition proceeds through several steps:

1. Initial knowledge base construction. MOLE asks the expert to list common symptoms or complaints that might require diagnosis.

For each symptom ,MOLE prompts for a list of possible explanations.

Whenever an event has multiple explanations, MOLE tries to determine the conditions under which the explanation is correct.

The expert provides COVERING knowledge ,that is the knowledge that a hypothesized event does occur, then the symptom will definitely appear.

2. Refinement of knowledge Base:

MOLE now tries to identify the weaknesses of knowledge base. One approach is to find holes and prompt the expert to fill them.

MoLE lets the expert watch MOLE-P solving sample problems.

When ever MOLE-p makes an incorrect diagnosis ,the expert adds new knowledge.

For Ex: suppose we have a patient with Symptoms A and B. Futher suppose that symptom A could be caused by the events X and Y, and that symptom B can be caused by Y and Z.

MOLE-p may conclude Y, since it explains both A and B.

If the expert indicates that this decision was incorrect, then MOLE will ask what evidence should be used tp prefer X and/or Z over Y

- Suppose if our task is to design an artifact for eg: an elevator system, then we must assign values to large number of parameters such as width of the platform, the type of door, the cable weight, cable strength.
- These parameters must be consistent with each other and they must result in the design that satisfies external constraints imposed by cost factors, the type of building involved and the expected payloads.
- One problem solving method useful for design tasks is called **propose and Revise**.
- Here the system first proposes an extension to the current design. Then it checks whether the extension violates any global or local constraints.
- Constraints violations are fixed and the process repeats.
- It turns out that domain experts are good at listing overall design constraints and providing local constraints on the individual parameters, but not so good at explaining how to arrive at global solutions.
- The SALT program provides mechanisms for elucidating this knowledge from the expert.
- SALT builds a dependency network as it converses with the expert.
- Each node stands for a value of a parameter that must be acquired or generated.
- There are three kinds of links:
 - *Contributes-to*, *constrains*, *suggests-revision-of*
 - *Contributes-to* link are procedures that allows SALT to generate a value for one parameter based on the value of another..
 - *Constrains* rules out certain parameter values.
 - *Suggests -revision- of* link points to ways in which a constrain violation can be fixed.

SALT uses the following heuristics to guide the acquisition process:

1. Every non-input node in the network needs atleast one link coming into it. If links are missing the expert is asked to fill it.
2. No contribute-to loops are allowed in the network. If a loop exists, SALT tries to transform one of the contributes to links into constrains links.
3. Constrains links should have Suggests-revision-of links associated with them.

- These includes constrain links that are created when dependency loops are broken.
- SALT compiles its dependency network into set of production rules..
- Consider a bank's problem in deciding whether to approve a loan a loan.
- One approach to automate this task is to interview loan officers in an attempt to extract domain knowledge.
- Another approach is to inspect the records of loans the bank has made in the past and try to generate rules automatically that will maximize the number of good loans and minimize the number of bad ones in the future.
- META DENDRAL was the first program to use learning techniques to construct rules for the expert system automatically