ARTIFICIAL INTELLIGENCE

(R18A1205)

DIGITAL NOTES

B.TECH

(III YEAR – II SEM)

(2021-2022)

DEPARTMENT OF AERONAUTICAL ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Affiliated to JNTU, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Kompally), Secunderabad – 500100, Telangana State, India.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

III Year B. Tech, ANE-II Sem	**L**	**T/P/D**	**C**
	2	**1/-/-**	**3**

(R18A1205)
ARTIFICIALINTELLIGENCE

Course Objectives: To train the students to understand different types of AI agents, various AI search algorithms, fundamentals of knowledge representation, building of simple knowledge- based systems and to apply knowledge representation, reasoning. Study of Markov Models enable the student ready to step into applied AI.

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)

UNIT - II

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

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

UNIT - III

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

Reasoning Under Uncertainty: Basic probability, Acting Under Uncertainty, Bayes' Rule,

Representing Knowledge in an Uncertain Domain, Bayesian Networks

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.

UNIT - V

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

TEXT BOOK:

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

REFERENCE BOOKS:

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

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.

History of AI:

Important research that laid the groundwork for AI:

- In 1931, Goedel layed the foundation of Theoretical Computer Science **1920-30s**: He published the first universal formal language and showed that math itself is either flawed or allows for unprovable but true statements.
- In 1936, Turing reformulated Goedel's result and church's extension thereof.
- In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the **Dartmouth Conference**, the first conference devoted to the subject.

- In 1957, The **General Problem Solver (GPS)** demonstrated by Newell, Shaw & Simon
- In 1958, John McCarthy (MIT) invented the Lisp language.
- In 1959, Arthur Samuel (IBM) wrote the first game-playing program, for checkers, to achieve sufficient skill to challenge a world champion.
- In 1963, Ivan Sutherland's MIT dissertation on Sketchpad introduced the idea of interactive graphics into computing.
- In 1966, Ross Quillian (PhD dissertation, Carnegie Inst. of Technology; now CMU) demonstrated semantic nets
- In 1967, Dendral program (Edward Feigenbaum, Joshua Lederberg, Bruce Buchanan, Georgia Sutherland at Stanford) demonstrated to interpret mass spectra on organic chemical compounds. First successful knowledge-based program for scientific reasoning.
- In 1967, Doug Engelbart invented the mouse at SRI
- In 1968, Marvin Minsky & Seymour Papert publish Perceptrons, demonstrating limits of simple neural nets.
- In 1972, Prolog developed by Alain Colmerauer.
- In Mid 80's, Neural Networks become widely used with the Backpropagation algorithm (first described by Werbos in 1974).
- 1990, Major advances in all areas of AI, with significant demonstrations in machine learning, intelligent tutoring, case-based reasoning, multi-agent planning, scheduling, uncertain reasoning, data mining, natural language understanding and translation, vision, virtual reality, games, and other topics.
- In 1997, Deep Blue beats the World Chess Champion Kasparov
- In 2002, iRobot, founded by researchers at the MIT Artificial Intelligence Lab, introduced **Roomba**, a vacuum cleaning robot. By 2006, two million had been sold.

Application 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:

a) "The exciting new effort to make computers think . . . *machines with minds,* in the full and literal sense" (Haugeland, 1985) "The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning..."(Bellman, 1978)	b) "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
c) "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1 99 1)	d) "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1 990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

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	**Rationally**
Think:	**Cognitive Science Approach** ***"Machines that think like humans"***	**Laws of thought Approach** ***" Machines that think Rationally"***
Act:	**Turing Test Approach** ***"Machines that behave like humans"***	**Rational Agent Approach** ***"Machines that behave Rationally"***

Scientific Goal: To determine which ideas about knowledge representation, learning, rule systems search, and so on, explain various sorts of real intelligence.

Engineering Goal: To solve real world problems using AI techniques such as Knowledge representation, learning, rule systems, search, and so on.

Traditionally, computer scientists and engineers have been more interested in the engineering goal, while psychologists, philosophers and cognitive scientists have been more interested in the scientific goal.

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 it 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

The difference between strong AI and weak AI:

Strong AI makes the bold claim that computers can be made to think on a level (at least) equal to humans.

Weak AI simply states that some "thinking-like" features can be added to computers to make them more useful tools... and this has already started to happen (witness expert systems, drive-by-wire cars and speech recognition software).

AI Problems:

AI problems (speech recognition, NLP, vision, automatic programming, knowledge representation, etc.) can be paired with techniques (NN, search, Bayesian nets, production systems, etc.).AI problems can be classified in two types:

1. Common-place tasks(Mundane Tasks)
2. Expert tasks

Common-Place Tasks:

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

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

Expert tasks:

1. Medical diagnosis.

2. Mathematical problem solving
3. Playing games like chess

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

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

1. *Intelligent Agent's:*

2.1 Agents and environments:

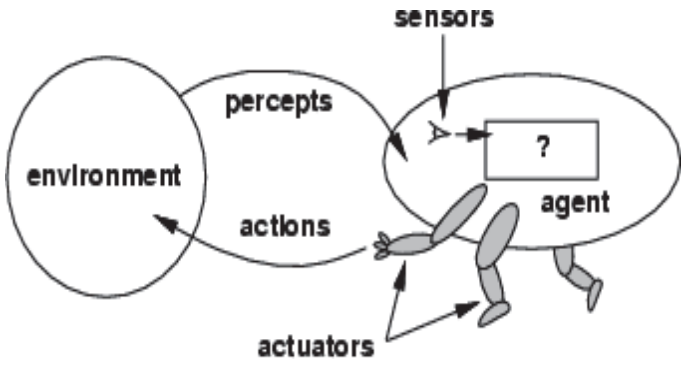


Fig 2.1: Agents and Environments

2.1.1 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.

2.1.2 Percept:

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

2.1.3 Percept Sequence:

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

2.1.4 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**.

2.1.5 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**.

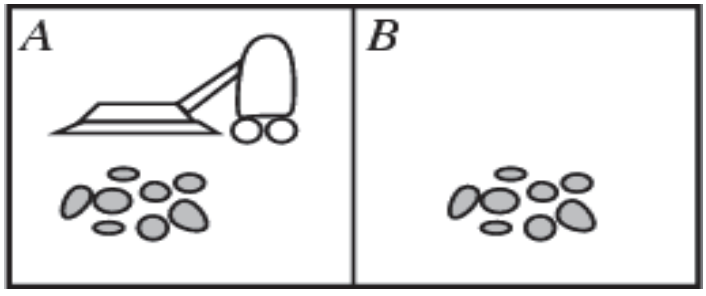


Fig 2.1.5: A vacuum-cleaner world with just two locations.

2.1.6 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

Strategies of Solving Tic-Tac-Toe Game Playing

Tic-Tac-Toe Game Playing:

Tic-Tac-Toe is a simple and yet an interesting board game. Researchers have used various approaches to study the Tic-Tac-Toe game. For example, Fok and Ong and Grim et al. have used artificial neural network based strategies to play it. Citrenbaum and Yakowitz discuss games like Go-Moku, Hex and Bridg-It which share some similarities with Tic-Tac-Toe.

.

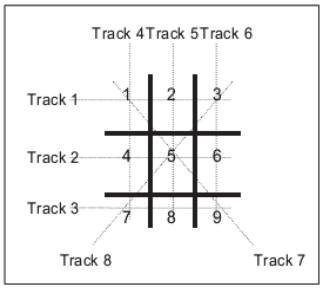


Fig 1.

A Formal Definition of the Game:

The board used to play the Tic-Tac-Toe game consists of 9 cells laid out in the form of a 3x3 matrix (Fig. 1).

The game is played by 2 players and either of them can start. Each of the two players is assigned a unique symbol (generally 0 and X). Each player alternately gets a turn to make a move. Making a move is compulsory and cannot be deferred. In each move a player places the symbol assigned to him/her in a hitherto blank cell.

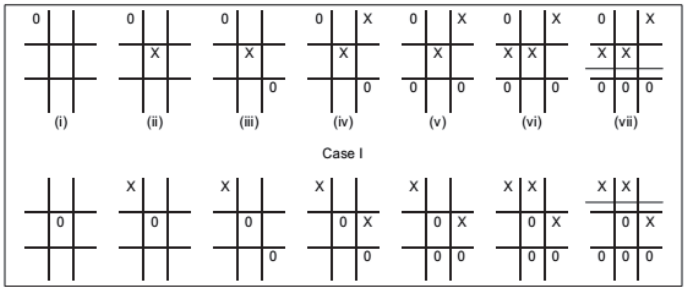
Let a track be defined as any row, column or diagonal on the board. Since the board is a

square matrix with 9 cells, all rows, columns and diagonals have exactly 3 cells. It can be easily

observed that there are 3 rows, 3 columns and 2 diagonals, and hence a total of 8 tracks on the board (Fig. 1). The goal of the game is to fill all the three cells of any track on the board with the symbol assigned to one before the opponent does the same with the symbol assigned to him/her. At any point of the game, if there exists a track whose all three cells have been marked by the same symbol, then the player to whom that symbol have been assigned wins and the game terminates.

If there exist no track whose cells have been marked by the same symbol when there is no more blank cell on the board then the game is drawn.

Let the priority of a cell be defined as the number of tracks passing through it. The priorities of the nine cells on the board according to this definition are tabulated in Table 1. Alternatively, let the priority of a track be defined as the sum of the priorities of its three cells. The priorities of the eight tracks on the board according to this definition are tabulated in Table 2. The prioritization of the cells and the tracks lays the foundation of the heuristics to be used in this study. These heuristics are somewhat similar to those proposed by Rich and Knight.



Strategy 1:

Algorithm:

1. View the vector as a ternary number. Convert it to a decimal number.
2. Use the computed number as an index into Move-Table and access the vector stored there.
3. Set the new board to that vector.

Procedure:

1) Elements of vector:

0: Empty

1: X

2: O

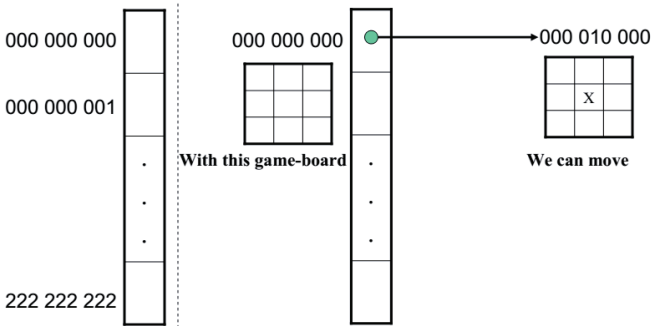
→ the vector is a ternary number

2) Store inside the program a move-table (lookuptable):

a) Elements in the table: 19683 (3^9)

b) Element = A vector which describes the most suitable move from the

❖ Data Structure:



Comments:

1. A lot of space to store the Move-Table.
2. A lot of work to specify all the entries in the Move-Table.
3. Difficult to extend

Explanation of Strategy 2 of solving Tic-tac-toe problem

Stratergy 2:

Data Structure:

1) Use vector, called board, as Solution 1
2) However, elements of the vector:

2: Empty

3: X

5: O

3) Turn of move: indexed by integer

1,2,3, etc

Function Library:

1. Make2:

a) Return a location on a game-board.

IF (board[5] = 2)

RETURN 5; //the center cell.

ELSE

RETURN any cell that is not at the board's corner;

// (cell: 2,4,6,8)

b) Let P represent for X or O

c) can_win(P) :

P has filled already at least two cells on a straight line (horizontal, vertical, or diagonal)

d) cannot_win(P) = NOT(can_win(P))

2. Posswin(P):

IF (cannot_win(P))

RETURN 0;

ELSE

RETURN index to the empty cell on the line of

can_win(P)

Let odd numbers are turns of X

Let even numbers are turns of O

3. Go(n): make a move

Algorithm:

1. Turn = 1: (X moves)

Go(1) //make a move at the left-top cell

2. Turn = 2: (O moves)

IF board[5] is empty THEN

Go(5)

ELSE

Go(1)

3. Turn = 3: (X moves)

IF board[9] is empty THEN

Go(9)

ELSE

Go(3).

4. Turn = 4: (O moves)

IF Posswin (X) <> 0 THEN

Go (Posswin (X))

//Prevent the opponent to win

ELSE Go (Make2)

5. Turn = 5: (X moves)

IF Posswin(X) <> 0 THEN

Go(Posswin(X))

//Win for X.

ELSE IF Posswin(O) <> THEN

Go(Posswin(O))

//Prevent the opponent to win

ELSE IF board[7] is empty THEN

Go(7)

ELSE Go(3).

Comments:

1. Not efficient in time, as it has to check several conditions before making each move.
2. Easier to understand the program's strategy.
3. Hard to generalize.

Introduction to Problem Solving, General problem solving

Problem solving is a process of generating solutions from observed data.

−a problem is characterized by a set of goals,

−a set of objects, and

−a set of operations.

These could be ill-defined and may evolve during problem solving.

Searching Solutions:

To build a system to solve a problem:

1. Define the problem precisely
2. Analyze the problem
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best problem-solving techniques and apply it to the particular problem.

Defining the problem as State 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)**

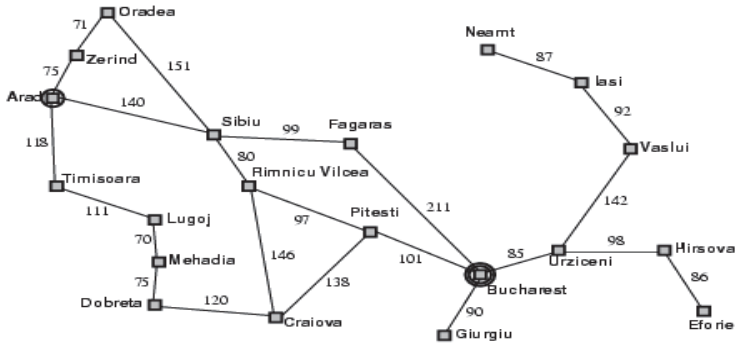
Specify a set of rules that describe the actions **(operations)** available

State-Space Problem Formulation:

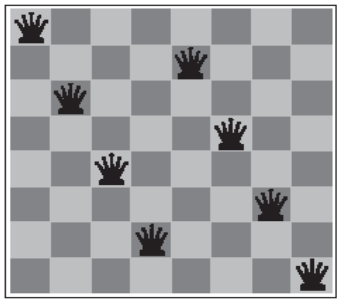
Example: A problem is defined by four items:

1. **initial state** e.g., "at Arad“
2. **actions or successor function** : *S(x)* = set of action–state pairs
   e.g., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }
3. **goal test (or set of goal states)**
   e.g., *x* = "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).

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

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.

General problem solving, Water-jug problem, 8-puzzle problem

General Problem Solver*:*

The General Problem Solver (GPS) was the first useful AI program, written by Simon, Shaw, and Newell in 1959. As the name implies, it was intended to solve nearly any problem.

Newell and Simon defined each problem as a space. At one end of the space is the starting point; on the other side is the goal. The problem-solving procedure itself is conceived as a set of operations to cross that space, to get from the starting point to the goal state, one step at a time.

The General Problem Solver, the program tests various actions (which Newell and Simon called operators) to see which will take it closer to the goal state. An operator is any activity that changes the state of the system. The General Problem Solver always chooses the operation that appears to bring it closer to its goal.

Example: Water Jug Problem

Consider the following problem:

> A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State :

> We will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$. Our initial state: (0, 0)

Goal Predicate - state = (2, y) where $0 \leq y \leq 3$.

Operators -we must defi ne a set of operators that will take us from one state to another:

1. Fill 4-gal jug	(x,y) $x < 4$	→ (4,y)
2. Fill 3-gal jug	(x,y) $y < 3$	→ (x,3)
3. Empty 4-gal jug on ground	(x,y) $x > 0$	→ (0,y)
4. Empty 3-gal jug on ground	(x,y)	→ (x,0)

	y > 0	
5. Pour water from 3-gal jug to ll 4-gal jug	(x,y) 0 < x+y 4 and y > 0	→! (4, y - (4 - x))
6. Pour water from 4-gal jug to ll 3-gal-jug	(x,y) 0 < x+y 3 and x > 0	→ (x - (3-y), 3)
7. Pour all of water from 3-gal jug into 4-gal jug	(x,y) 0 < x+y 4 and y 0	→ (x+y, 0)
8. Pour all of water from 4-gal jug into 3-gal jug	(x,y) 0 < x+y 3 and x 0	→ (0, x+y)

Through Graph Search, the following solution is found :

Gals in 4-gal jug	Gals in 3-gal jug	Rule Applied
0	0	
		1. Fill 4
4	0	
		6. Pour 4 into 3 to ll
1	3	
		4. Empty 3
1	0	
		8. Pour all of 4 into 3
0	1	
		1. Fill 4
4	1	
		6. Pour into 3
2	3	

Second Solution:

Number of Steps	*Rules applied*	*4-g jug*	*3-g jug*
1	**Initial State**	0	0
2	R2 {Fill 3-g jug}	0	3
3	R7 {Pour all water from 3 to 4-g jug }	3	0
4	R2 {Fill 3-g jug}	3	3
5	R5 {Pour from 3 to 4-g jug until it is full}	4	2
6	R3 {Empty 4-gallon jug}	0	2
7	R7 {Pour all water from 3 to 4-g jug}	2	0
			Goal State

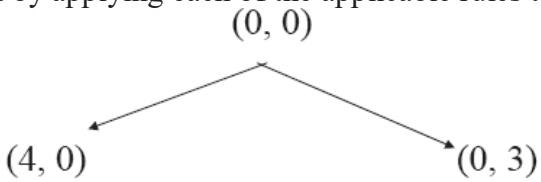
Systematic Control Strategies (Blind searches):

Breadth First Search:

Let us discuss these strategies using water jug problem. These may be applied to any search problem.

Construct a tree with the initial state as its root.

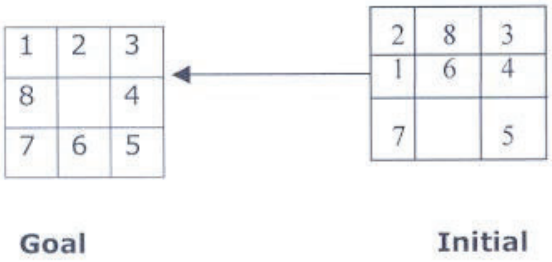
Generate all the offspring of the root by applying each of the applicable rules to the initial state**.**



Now for each leaf node, generate all its successors by applying all the rules that are appropriate.

8 Puzzle Problem.

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.



The program is to change the initial configuration into the goal configuration. A solution to the problem is an appropriate sequence of moves, such as “move tiles 5 to the right, move tile 7 to the left, move tile 6 to the down, etc”.

Solution:

To solve a problem using a production system, we must specify the global database the rules, and the control strategy. For the 8 puzzle problem that correspond to these three components. These elements are the problem states, moves and goal. In this problem each tile configuration is a state. The set of all configuration in the space of problem states or the problem space, there are only 3, 62,880 different configurations o the 8 tiles and blank space. Once the problem states have been conceptually identified, we must construct a computer representation or description of them . this description is then used as the database of a production system. For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. The initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.

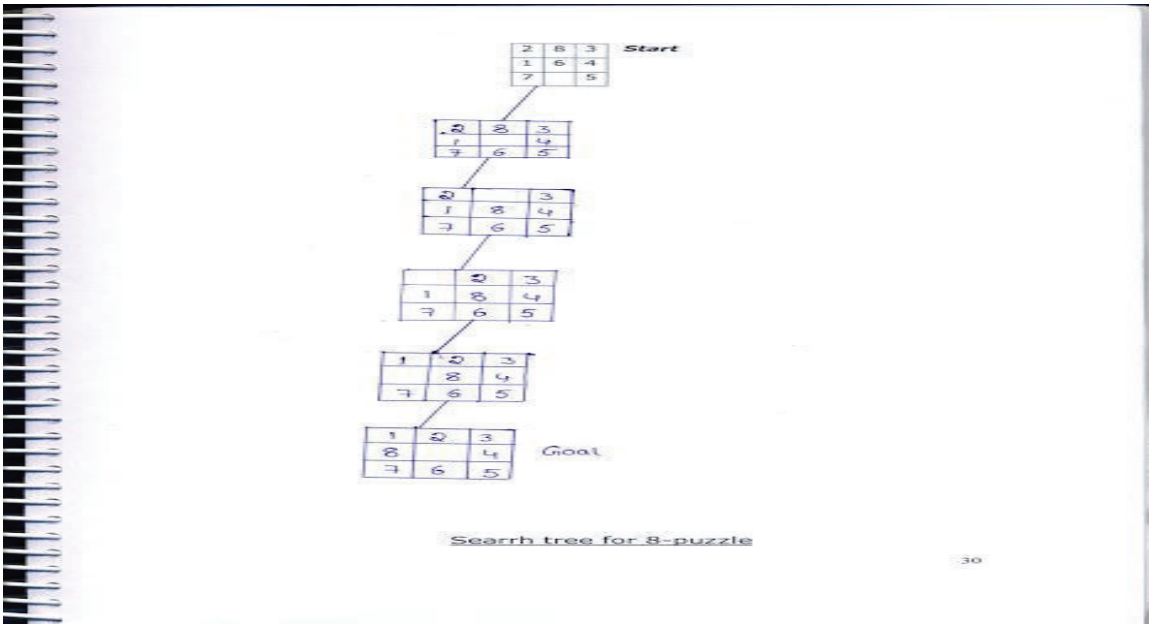
A move transforms one problem state into another state. The 8-puzzle is conveniently interpreted as having the following for moves. Move empty space (blank) to the left, move blank up, move blank to the right and move blank down,. These moves are modeled by production rules that operate on the state descriptions in the appropriate manner.

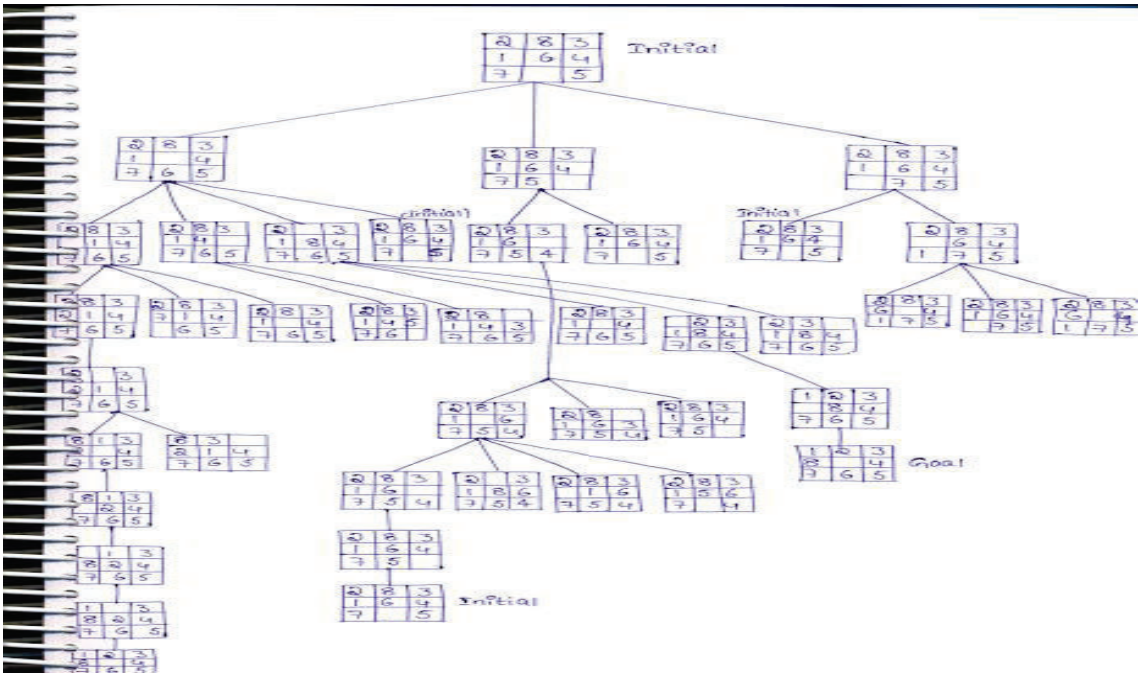
The rules each have preconditions that must be satisfied by a state description in order for them to be applicable to that state description. Thus the precondition for the rule associated with “move blank up” is derived from the requirement that the blank space must not already be in the top row.

The problem goal condition forms the basis for the termination condition of the production system. The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced. It also keeps track of rules that have been applied so that it can compose them into sequence representing the problem solution. A solution to the 8-puzzle problem is given in the following figure.

Example:- Depth – First – Search traversal and Breadth - First - Search traversal

for 8 – puzzle problem is shown in following diagrams.





Exhaustive Searches, BFS and DFS

Search is the systematic examination of states to find path from the start/root state to the goal state.

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 aremore 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 Methods:

Breadth- First -Search:

Consider the state space of a problem that takes the form of a tree. Now, if we search the goal along each breadth of the tree, starting from the root and continuing up to the largest depth, we call it *breadth first search*.

- **Algorithm:**
    1. Create a variable called NODE-LIST and set it to initial state
    2. Until a goal state is found or NODE-LIST is empty do
        a. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit
        b. For each way that each rule can match the state described in E do:
            i. Apply the rule to generate a new state
            ii. If the new state is a goal state, quit and return this state
            iii. Otherwise, add the new state to the end of NODE-LIST

BFS illustrated:

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

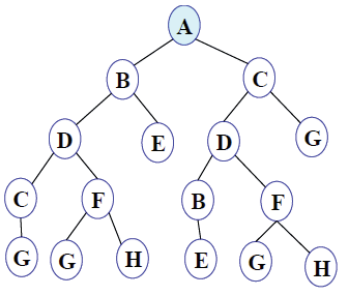


Figure 1

FRINGE: A

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

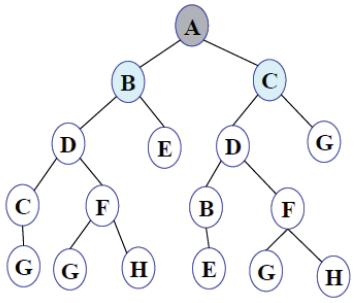


Figure 2

FRINGE: B C

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

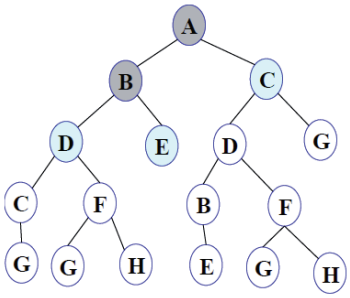


Figure 3

FRINGE: C D E

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

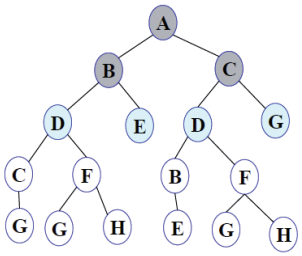


Figure 4

FRINGE: D E D G

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

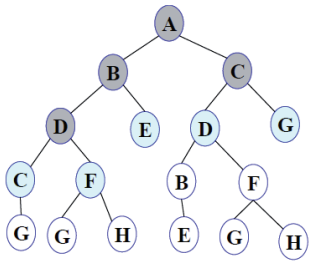


Figure 5

FRINGE: E D G C F

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

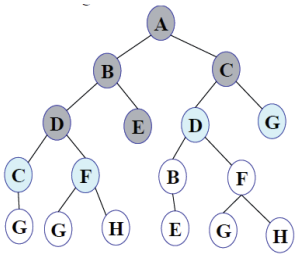


Figure 6

FRINGE: D G C F

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

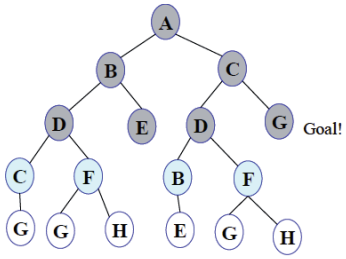


Figure 7

FRINGE: G C F B F

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

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: Finds the path of minimal length to the goal.

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.

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*.

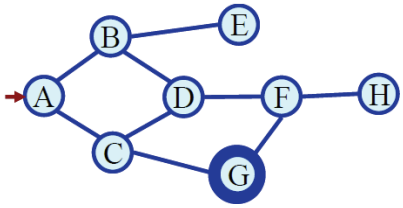
- **Algorithm:**
  1. Create a variable called NODE-LIST and set it to initial state
  2. Until a goal state is found or NODE-LIST is empty do
     a. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit
     b. For each way that each rule can match the state described in E do:

i. Apply the rule to generate a new state

ii. If the new state is a goal state, quit and return this state

iii. Otherwise, add the new state in front of NODE-LIST

DFS illustrated:



A State Space Graph

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

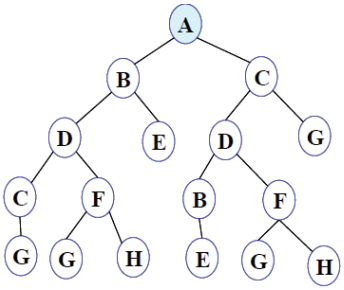


Figure 1

FRINGE: A

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

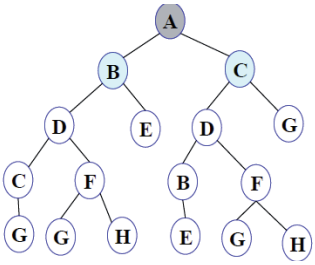


Figure 2

FRINGE: B C

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

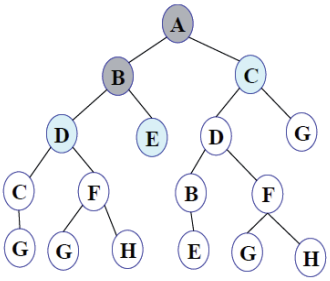


Figure 3

FRINGE: D E C

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

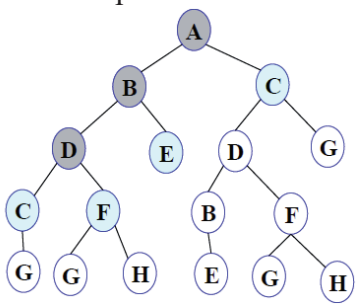


Figure 4

FRINGE: C F E C

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

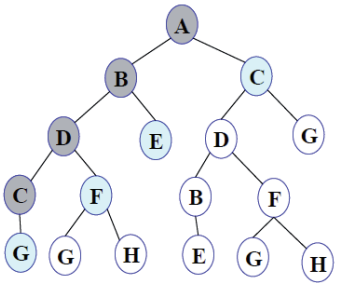


Figure 5

FRINGE: G F E C

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

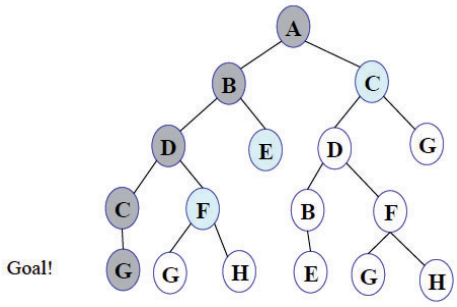


Figure 6

FRINGE: *G* F E C

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

Depth first search is:

1. The algorithm 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.**

Exhaustive searches- Iterative Deeping DFS

Description:

- It is a search strategy resulting when you combine BFS and DFS, thus combining the advantages of each strategy, taking the completeness and optimality of BFS and the modest memory requirements of DFS.
- IDS works by looking for the best search depth d, thus starting with depth limit 0 and make a BFS and if the search failed it increase the depth limit by 1 and try a BFS again with depth 1 and so on – first d = 0, then 1 then 2 and so on – until a depth d is reached where a goal is found.

Algorithm:

```
procedure IDDFS(root)
   for depth from 0 to ∞
      found ← DLS(root, depth)
      if found ≠ null
         return found


procedure DLS(node, depth)
   if depth = 0 and node is a goal
      return node
   else if depth > 0
      foreach child of node
         found ← DLS(child, depth−1)
         if found ≠ null
            return found
   return null
```

Performance Measure:

- Completeness: IDS is like BFS, is complete when the branching factor b is finite.
- Optimality: IDS is also like BFS optimal when the steps are of the same cost.

- Time Complexity:

- One may find that it is wasteful to generate nodes multiple times, but actually it is not that costly compared to BFS, that is because most of the generated nodes are always in the deepest level reached, consider that we are searching a binary tree and our depth limit reached 4, the nodes generated in last level = $2^4 = 16$, the nodes generated in all nodes before last level = $2^0 + 2^1 + 2^2 + 2^3 = 15$
- Imagine this scenario, we are performing IDS and the depth limit reached depth d, now if you remember the way IDS expands nodes, you can see that nodes at depth d are generated once, nodes at depth d-1 are generated 2 times, nodes at depth d-2 are generated 3 times and so on, until you reach depth 1 which is generated d times, we can view the total number of generated nodes in the worst case as:
  - $N(IDS) = (b)d + (d-1)b^2 + (d-2)b^3 + \ldots. + (2)b^{d-1} + (1)b^d = O(b^d)$
- If this search were to be done with BFS, the total number of generated nodes in the worst case will be like:
  - $N(BFS) = b + b^2 + b^3 + b^4 + \ldots. b^d + (b^{d+1} - b) = O(b^{d+1})$
- If we consider a realistic numbers, and use b = 10 and d = 5, then number of generated nodes in BFS and IDS will be like
  - N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450
  - N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100
  - BFS generates like 9 time nodes to those generated with IDS.

- Space Complexity:
  - IDS is like DFS in its space complexity, taking $O(bd)$ of memory.

    Weblinks:

    i. *https://www.youtube.com/watch?v=7QcoJjSVT38*

    ii. https://mhesham.wordpress.com/tag/iterative-deepening-depth-first-search

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 uniformed search method when there is a large search space and the depth of the solution is not known".

Heuristic Searches:

A Heuristic technique helps in solving problems, even though there is no guarantee that it will never lead in the wrong direction. There are heuristics of every general applicability as well as domain specific. The strategies are general purpose heuristics. In order to use them in a specific domain they are coupler with some domain specific heuristics. There are two major ways in which domain - specific, heuristic information can be incorporated into rule-based search procedure.

A heuristic function is a function that maps from problem state description to measures desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution.

Greedy Best First Search

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

$$\mathbf{f}\ (n) = h\ (n).$$

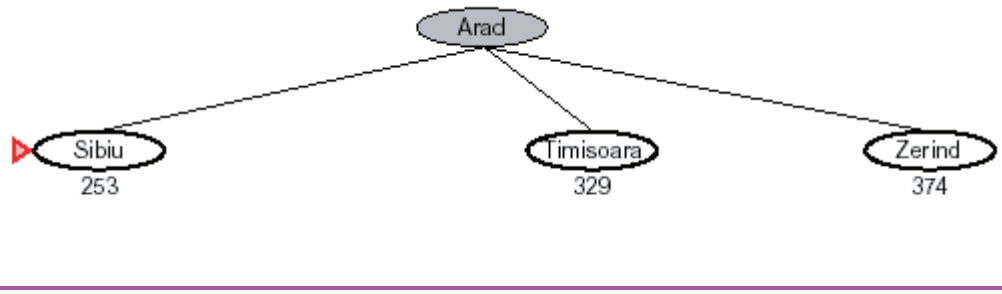
Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in **Figure 8.1**. For example, the initial state is In (Arad), and the straight line distance heuristic h_{SLD} (In (Arad)) is found to be 366. Using the **straight-line distance** heuristic $\mathbf{h_{SLD}}$, the goal state can be reached faster.

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

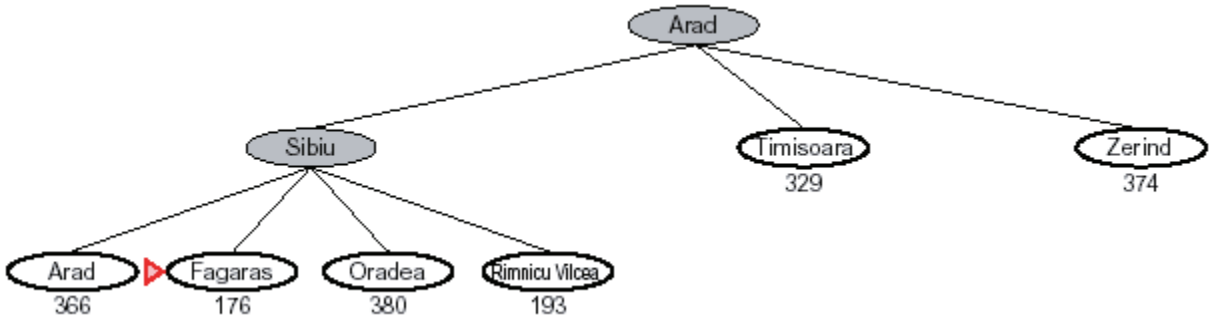
Figure 8.1: Values of h_{SLD}-straight-line distances to B u c h a r e s t.

The Initial State

After Expanding Arad



After Expanding Sibiu



After Expanding Fagaras

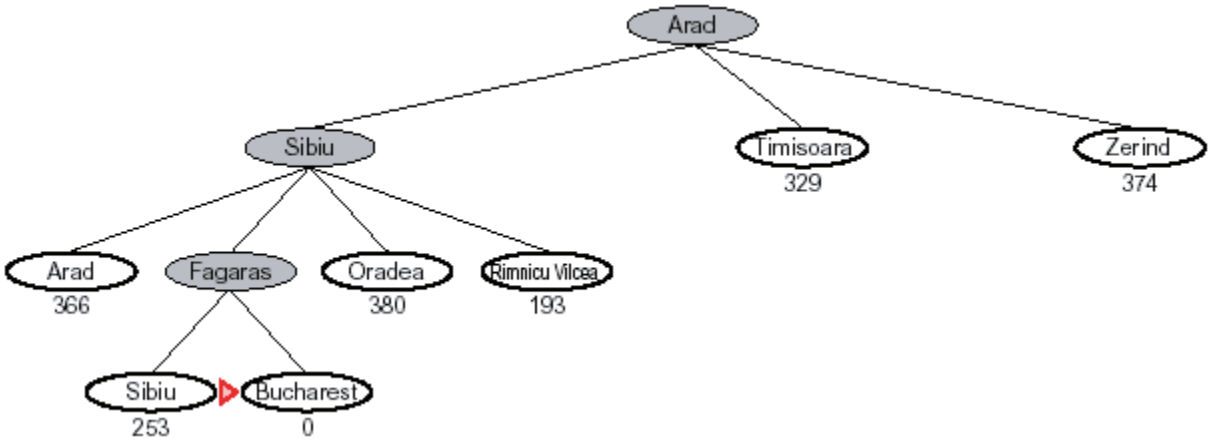


Figure 8.2: Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 8.2 shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest.

Fagaras in turn generates Bucharest, which is the goal.

Evaluation Criterion of Greedy Search

- **Complete: NO [can get stuck in loops, e.g., Complete in finite space with repeated-state checking]**
- **Time Complexity: O (bm) [but a good heuristic can give dramatic improvement]**
- **Space Complexity: O (bm) [keeps all nodes in memory]**
- **Optimal: NO**

Greedy best-first search is not optimal, and it is incomplete. The worst-case time and space complexity is **O (b^m),** where m is the maximum depth of the search space.

HILL CLIMBING PROCEDURE:

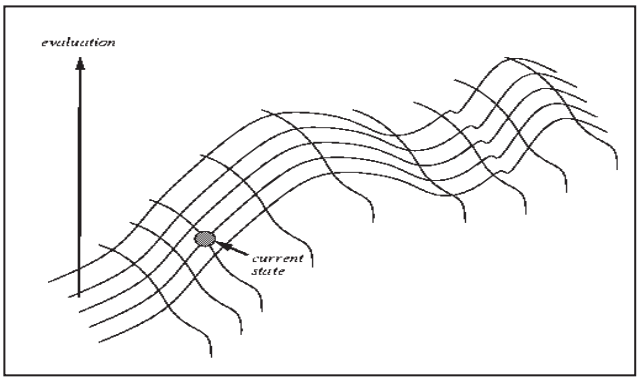
Hill Climbing Algorithm

We will assume we are trying to maximize a function. That is, we are trying to find a point in the search space that is better than all the others. And by "better" we mean that the evaluation is higher. We might also say that the solution is of better quality than all the others.

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.

We can also present this algorithm as follows (it is taken from the AIMA book (Russell, 1995) and follows the conventions we have been using on this course when looking at blind and heuristic searches).



Algorithm:

Function HILL-CLIMBING(*Problem*) **returns** a solution state

Inputs: *Problem*, problem

Local variables: *Current*, a node

Next, a node

Current = MAKE-NODE(INITIAL-STATE[*Problem*])

Loop do

Next = a highest-valued successor of *Current*

If VALUE[Next] < VALUE[Current] **then return** *Current*

Current = *Next*

End

You should note that this algorithm does not maintain a search tree. It only returns a final solution. Also, if two neighbors have the same evaluation and they are both the best quality, then the algorithm will choose between them at random.

Problems with Hill Climbing

The main problem with hill climbing (which is also sometimes called ***gradient descent***) is that we are not guaranteed to find the best solution. In fact, we are not offered any guarantees about the solution. It could be abysmally bad.

You can see that we will eventually reach a state that has no better neighbours but there are better solutions elsewhere in the search space. The problem we have just described is called a ***local maxima***.

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.

Figure 10.7 shows simulated annealing algorithm. It 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.

function ***S I M U L A T E D - A N NEALING****(problem, schedule)* **returns** a **solution state**
inputs: *problem,* a problem
schedule, a mapping from time to "temperature"
local variables: *current,* a node
next, a node
T, a "temperature" controlling the probability of downward steps
current ← **MAKE-NODE(INITIAL-STATE[***problem***])**
for t ← 1 **to** ∞ **do**
T ←*schedule[t]*
if *T* = 0 **then return** *current*
next ← a randomly selected successor of *current*
ΔE ← ***VALUE****[next] –* ***VALUE****[current]*
if ΔE > 0 **then** *current* ← *next*
else *current* ← *next* only with probability $e^{\Delta E / T}$

LOCAL SEARCH IN CONTINUOUS SPACES

- We have considered algorithms that work only in discrete environments, but real-world environment are continuous.
- Local search amounts to maximizing a continuous objective function in a multi-dimensional vector space.
- This is hard to do in general.
- Can immediately retreat
  - ✓ Discretize the space near each state
  - ✓ Apply a discrete local search strategy (e.g., stochastic hill climbing, simulated annealing)
- Often resists a closed-form solution

- ✓ Fake up an empirical gradient
- ✓ Amounts to greedy hill climbing in discretized state space

➢ Can employ Newton-Raphson Method to find maxima.

➢ Continuous problems have similar problems: plateaus, ridges, local maxima, etc.

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 priorityqueue 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.

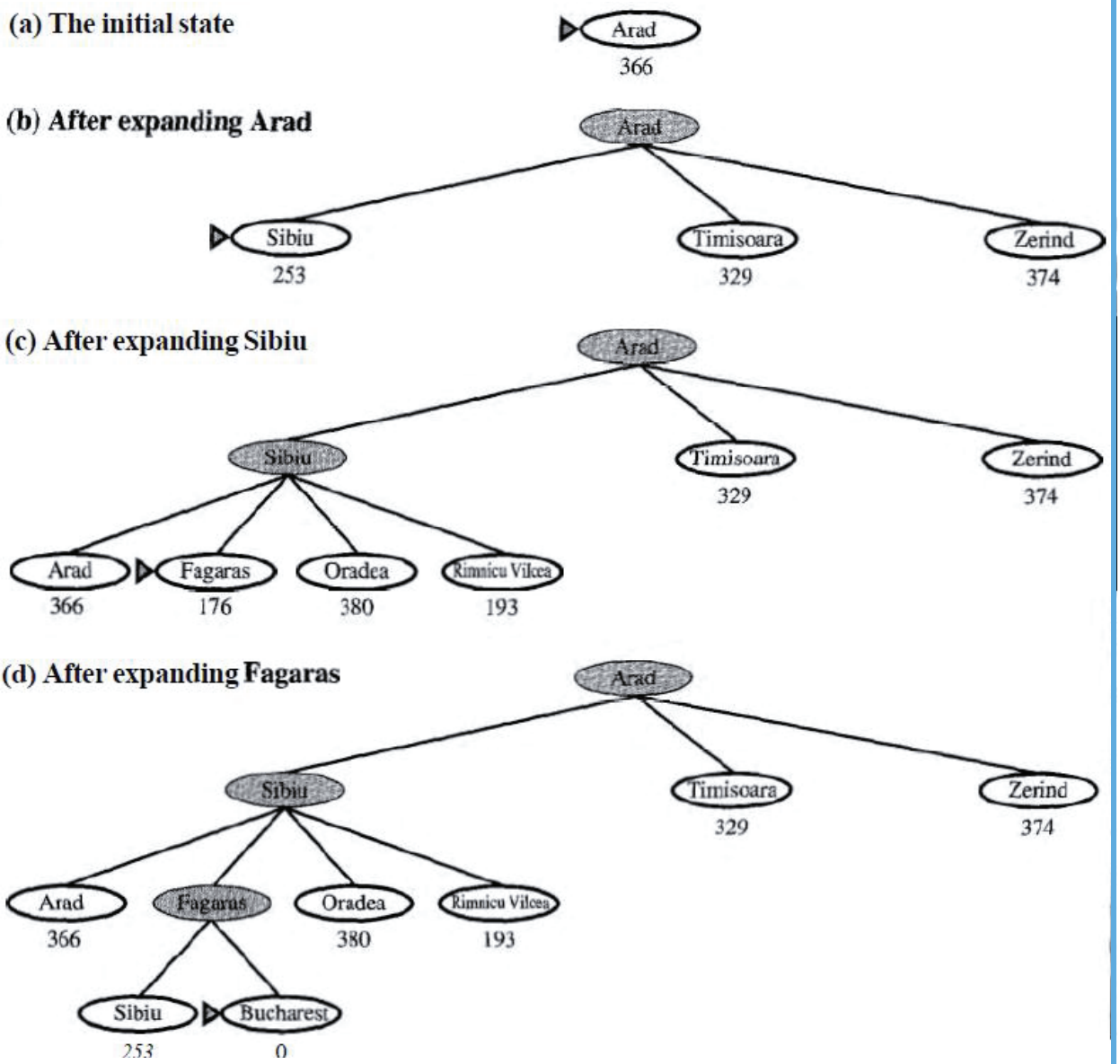
CLOSED are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

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:

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 € [OPEN U CLOSED]**
Set g(m) = g(n) + c(n , m)
Set f(m) = g(m) + h(m)
Insert m in OPEN

b) **If m € [OPEN U 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 € CLOSED

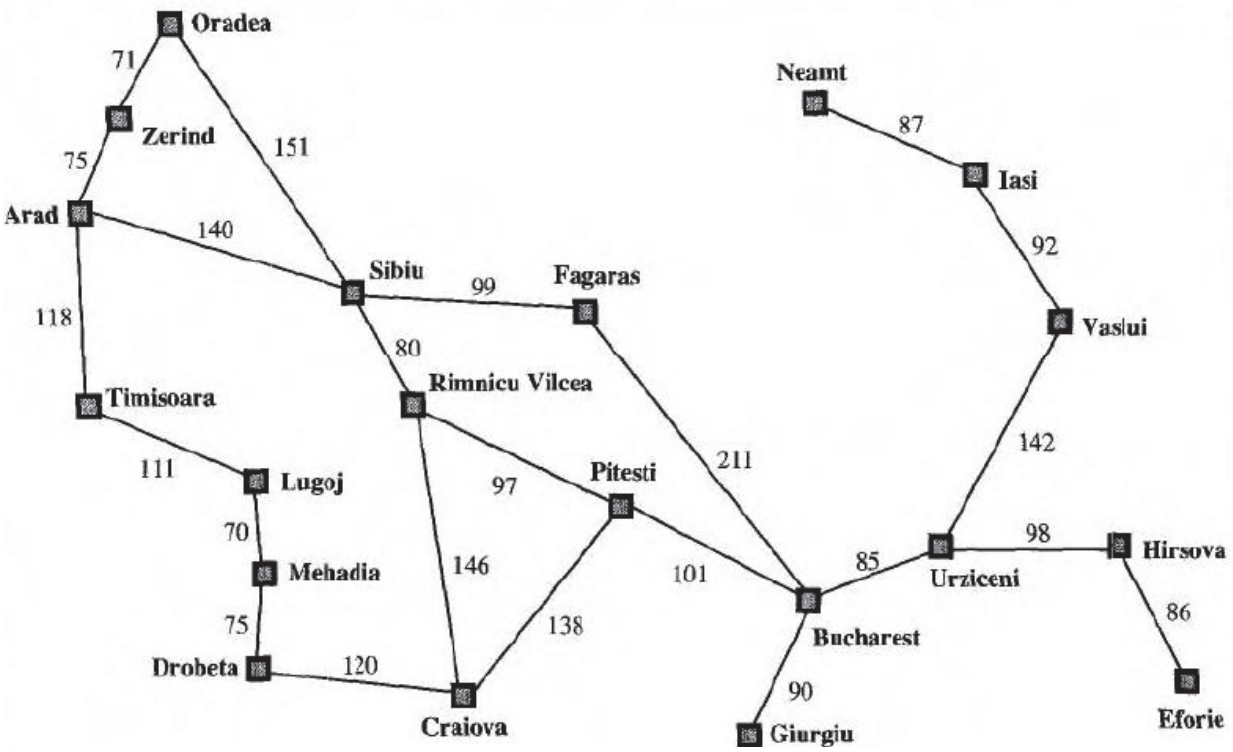
Move m to OPEN.

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:

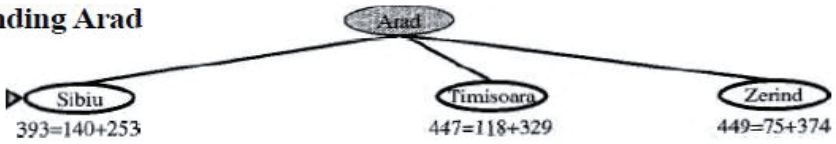
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



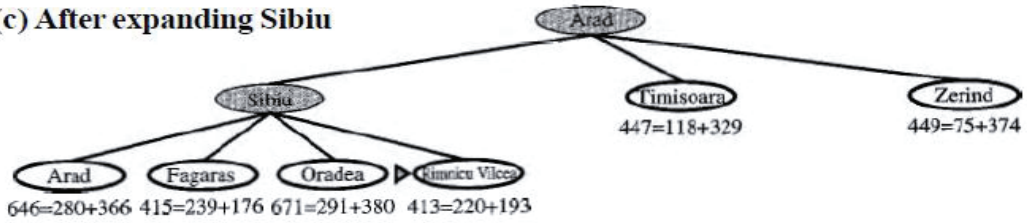
(a) The initial state

366=0+366

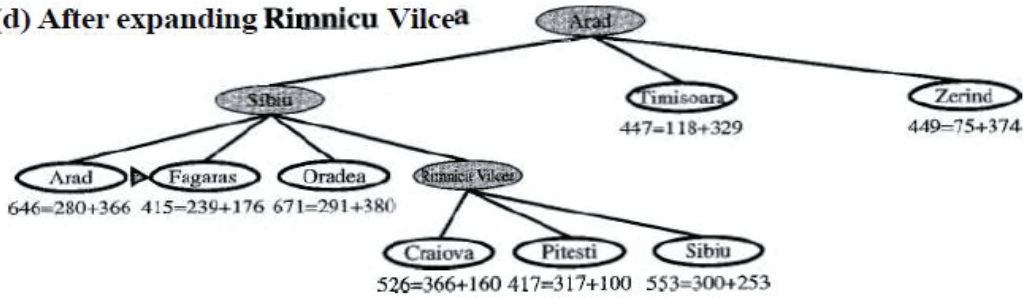
(b) After expanding Arad



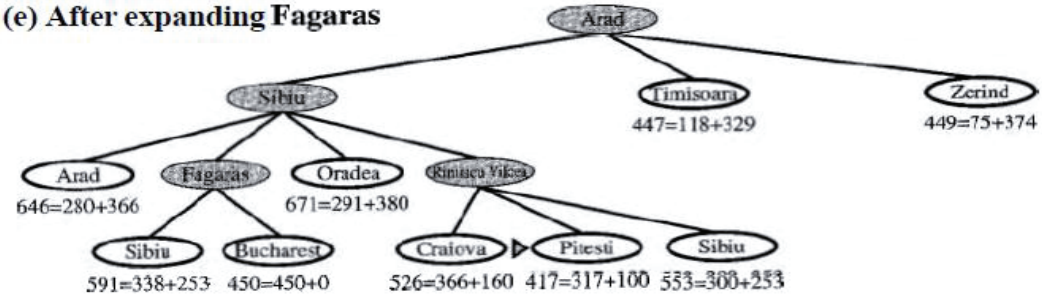
(c) After expanding Sibiu



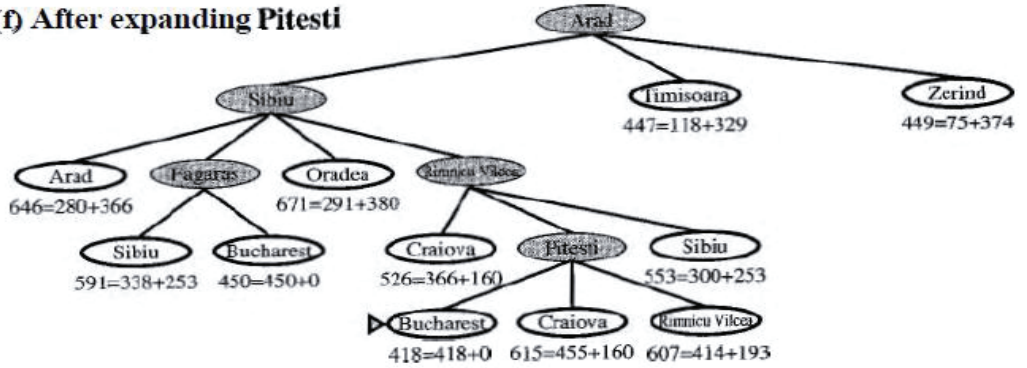
(d) After expanding Rimnicu Vilcea



(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 that uninformed search.

Constraint Satisfaction Problems

Refer:

file:///D:/AI%20NOTES/chapter05.pdf

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, X1;X2; : : : ;Xn, and a set of constraints, C1;C2; : : : ;Cm. Each variable Xi has a nonempty domain Di of possible values. Each constraint Ci involves some subset of t 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, {Xi = vi;Xj = vj ; : : :} 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 objectivefunction.

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

1. **Initial state**: the empty assignment fg, 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.
2. **Crypt arithmetic** puzzles.

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

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.

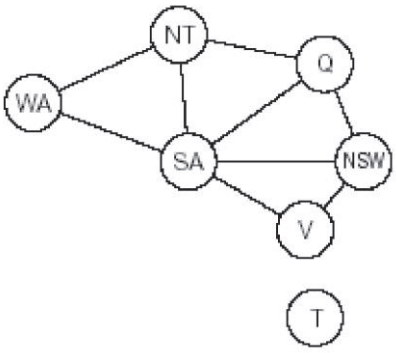
We are given the task of coloring each region red, green, or blue in such a way that the neighboring regions must not have the same color.

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 ≠ NT). There are many possible solutions, 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

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
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

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

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**.**

MiniMax 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.

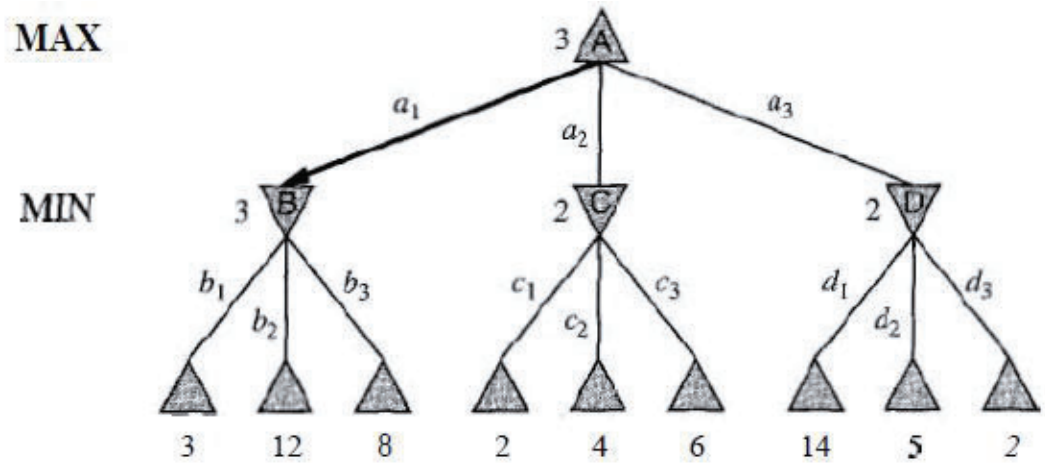
Algorithm: **MINIMAX** (Depth-First Version)

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 $J_1, J_2, \ldots, J_b$.
3. Evaluate $V(J_1), V(J_2), \ldots, V(J_b)$ from left to right.
4. If J is a MAX node, return $V(J) = \max[V(J_1), \ldots, V(J_b)]$.
5. If J is a MIN node, return $V(J) = \min[V(J_1), \ldots, V(J_b)]$.

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

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

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

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 ≈ 35, m ≈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
- ***α-β* 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 α.
- The MIN player is always trying to minimize the score. Call this β .
- **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) >= beta(n)
  (alpha increases and passes beta 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) <= alpha(n)
  (beta decreases and passes alpha from above)
- Carry alpha and beta values down during search Pruning occurs whenever alpha >= beta

Algorithm:

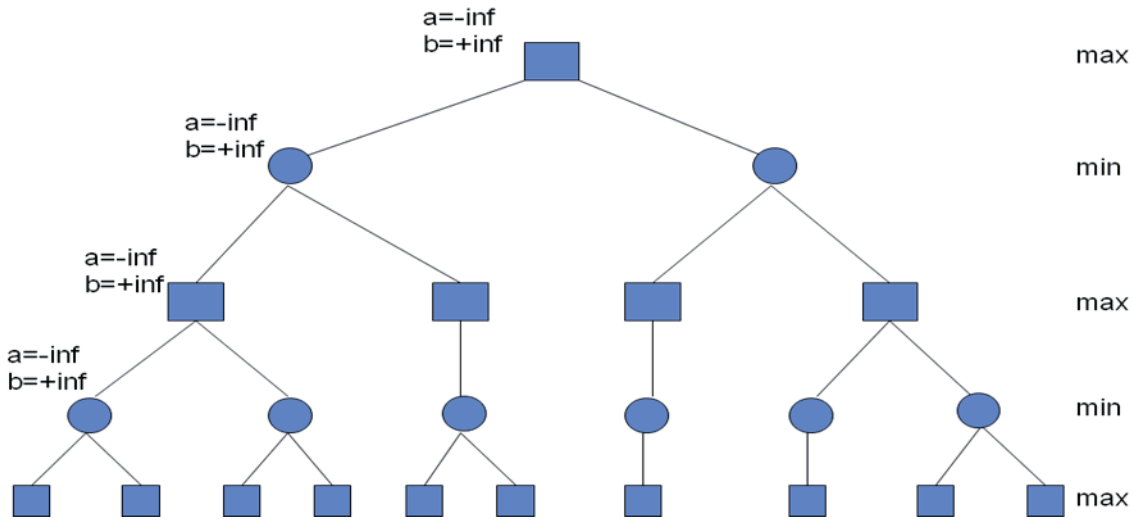
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v
```

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

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

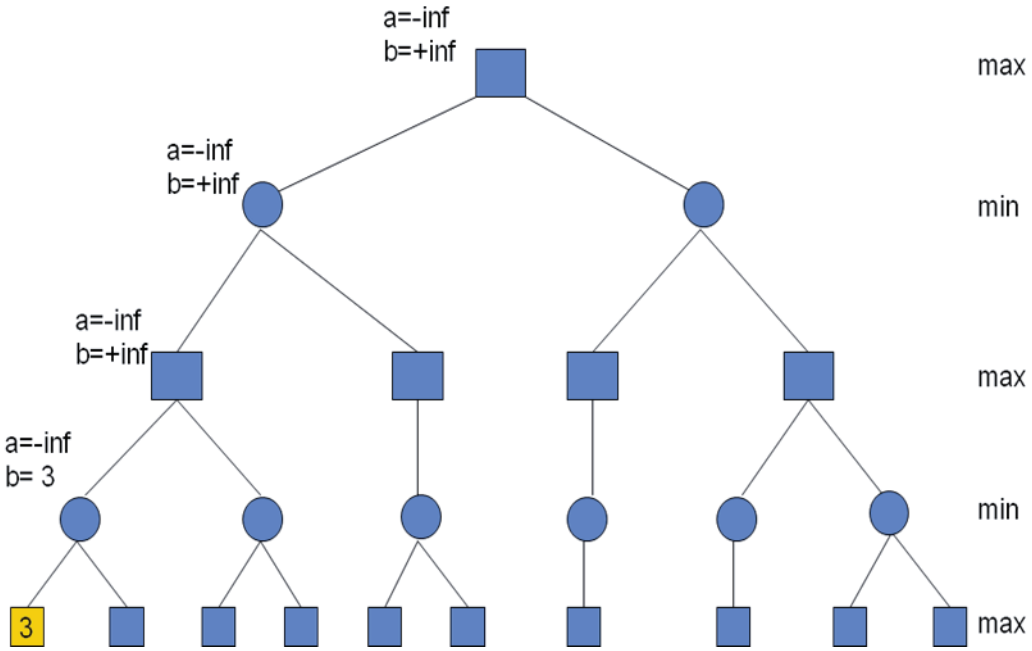
```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(%α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Example:

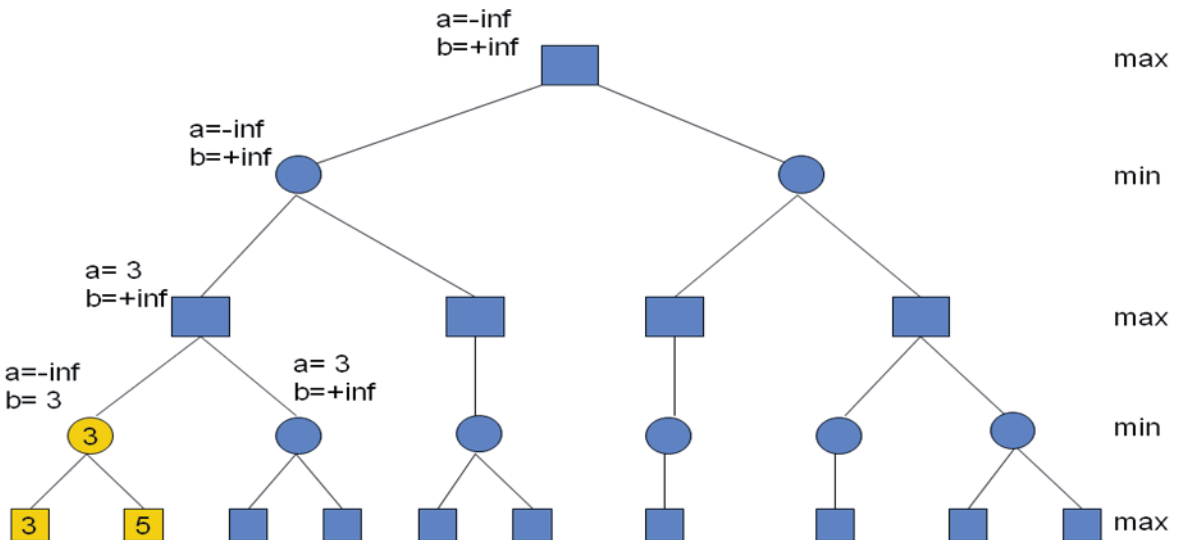
1) Setup phase: Assign to each left-most (or right-most) internal node of the tree, variables: alpha = -infinity, beta = +infinity



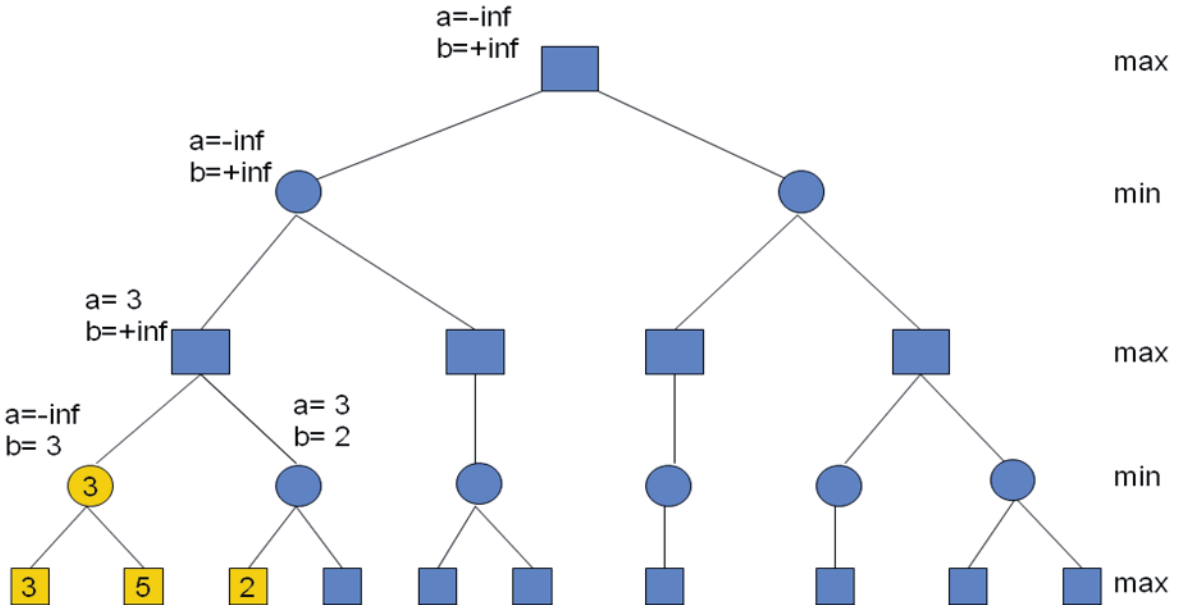
2) Look at first computed final configuration value. It's a 3. Parent is a min node, so set the beta (min) value to 3.



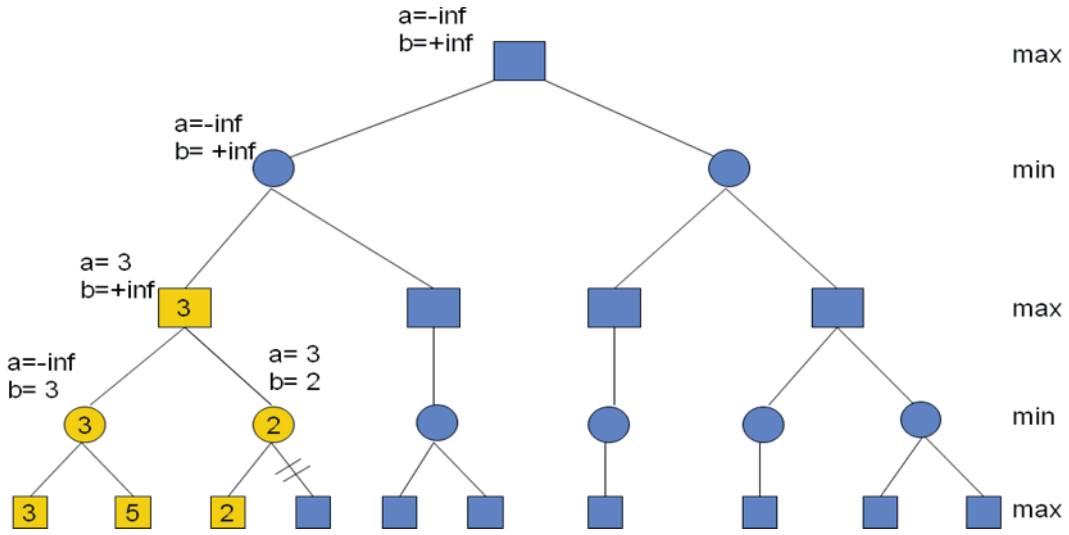
3) Look at next value, 5. Since parent is a min node, we want the minimum of 3 and 5 which is 3. Parent min node is done – fill alpha (max) value of its parent max node. Always set alpha for max nodes and beta for min nodes. Copy the state of the max parent node into the second unevaluated min child.



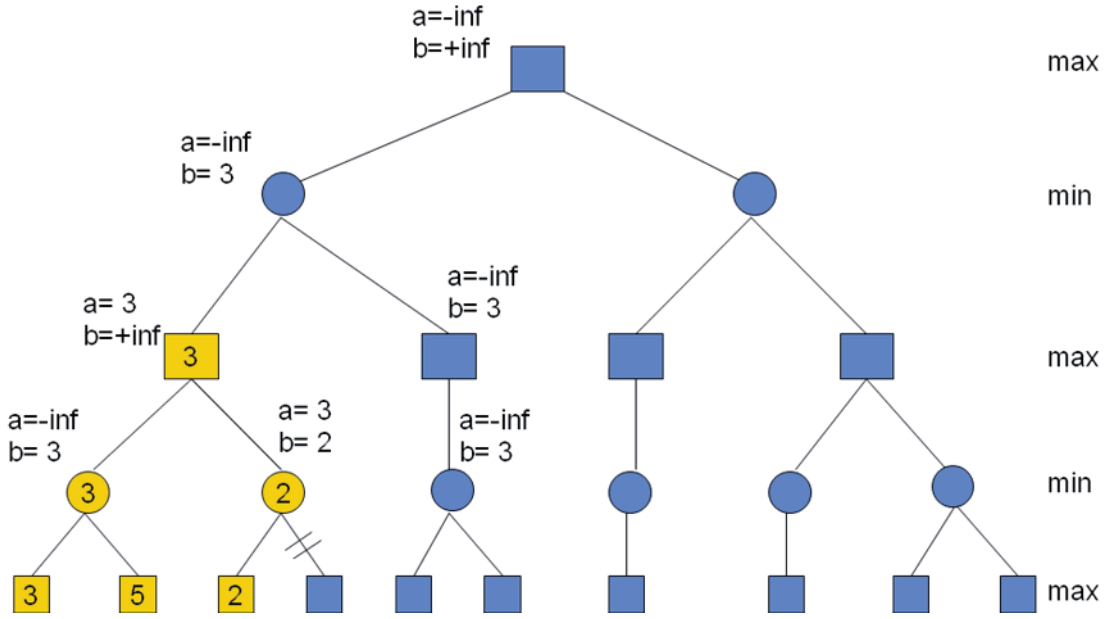
4) Look at next value, 2. Since parent node is min with b=+inf, 2 is smaller, change b.



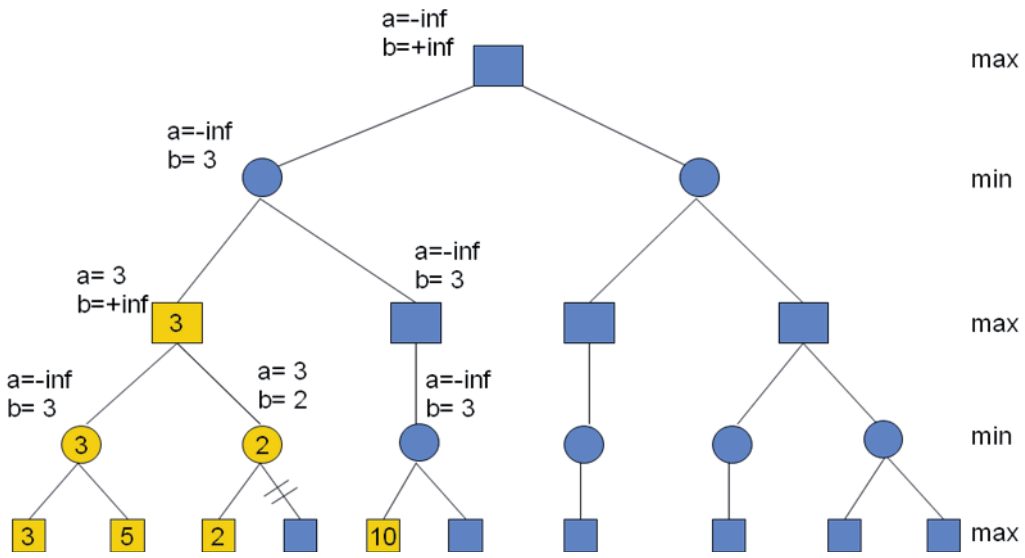
5) Now, the min parent node has a max value of 3 and min value of 2. The value of the 2nd child does not matter. If it is >2, 2 will be selected for min node. If it is <2, it will be selected for min node, but since it is <3 it will not get selected for the parent max node. Thus, we prune the right subtree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



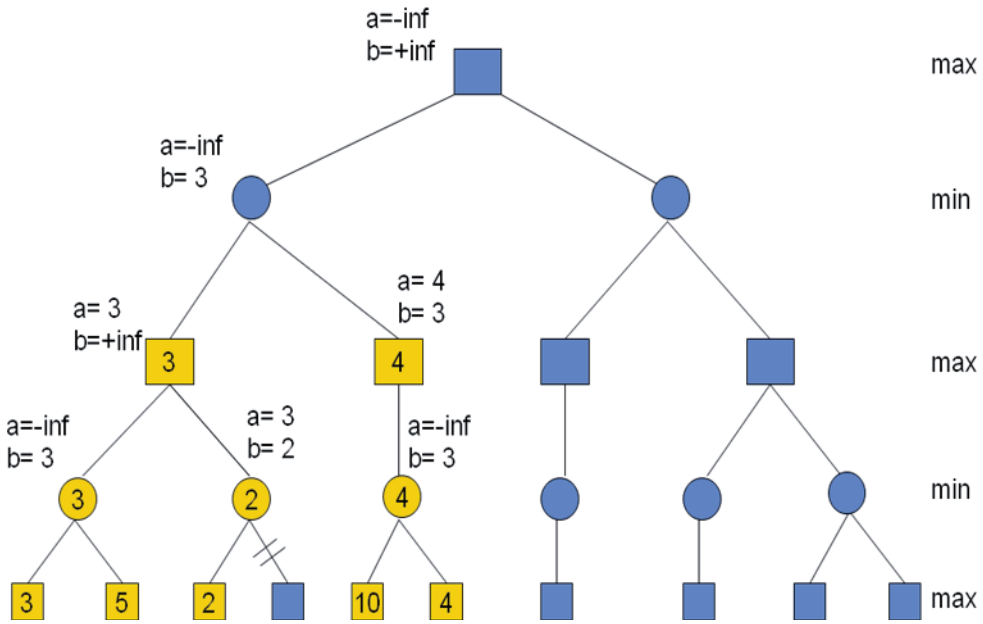
6) Max node is now done and we can set the beta value of its parent and propagate node state to sibling subtree's left-most path.



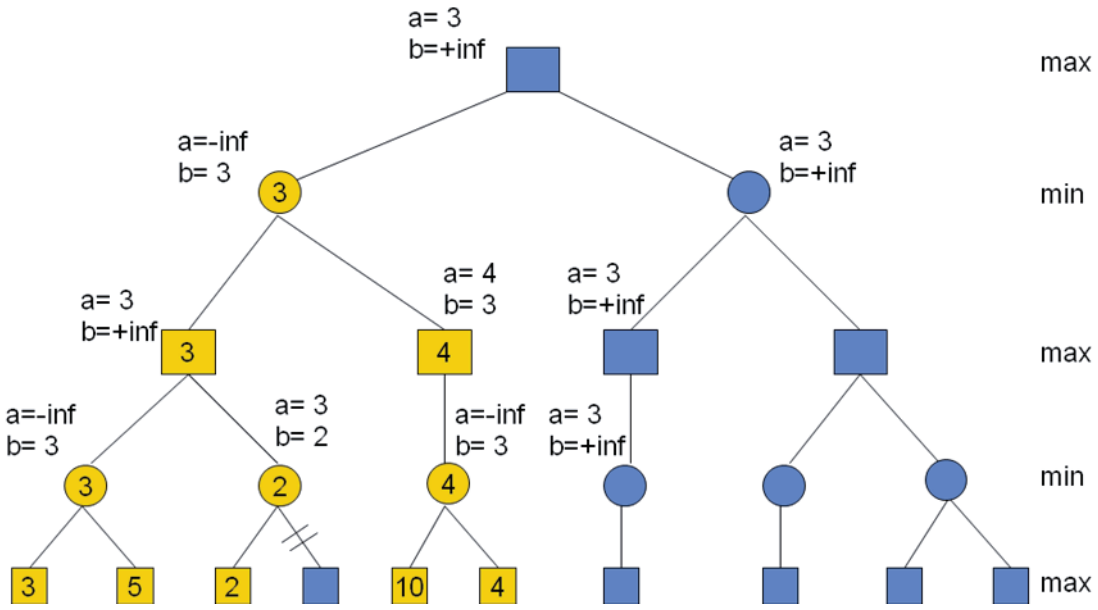
7) The next node is 10. 10 is not smaller than 3, so state of parent does not change. We still have to look at the 2nd child since alpha is still –inf.



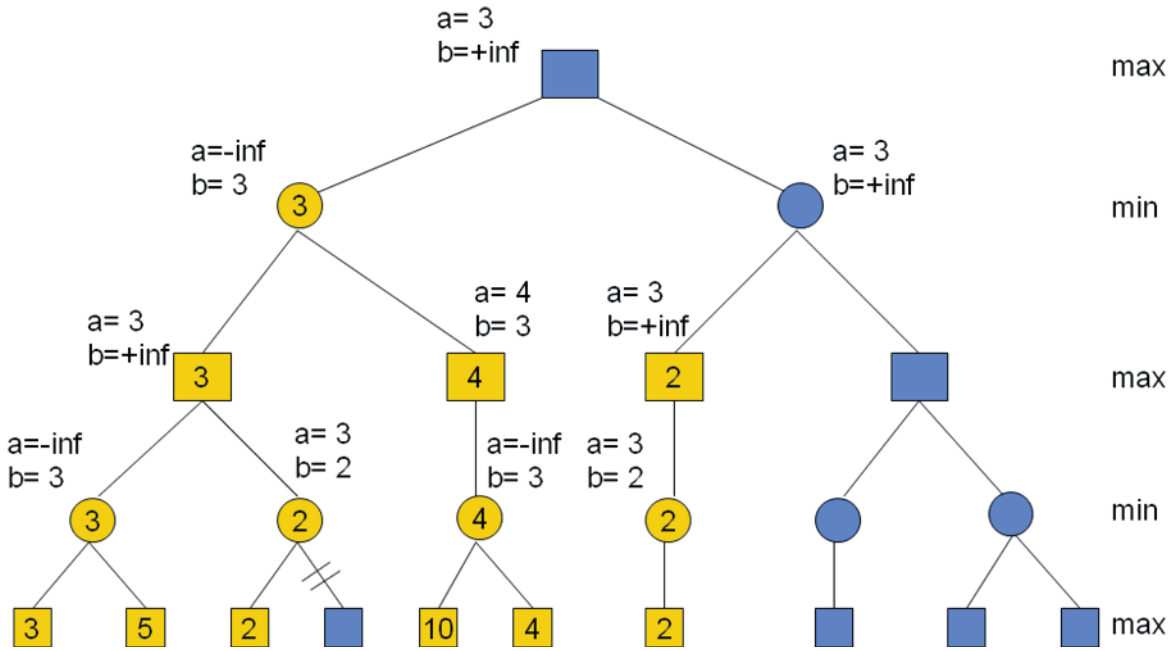
8) The next node is 4. Smallest value goes to the parent min node. Min subtree is done, so the parent max node gets the alpha (max) value from the child. Note that if the max node had a 2nd subtree, we can prune it since a>b.



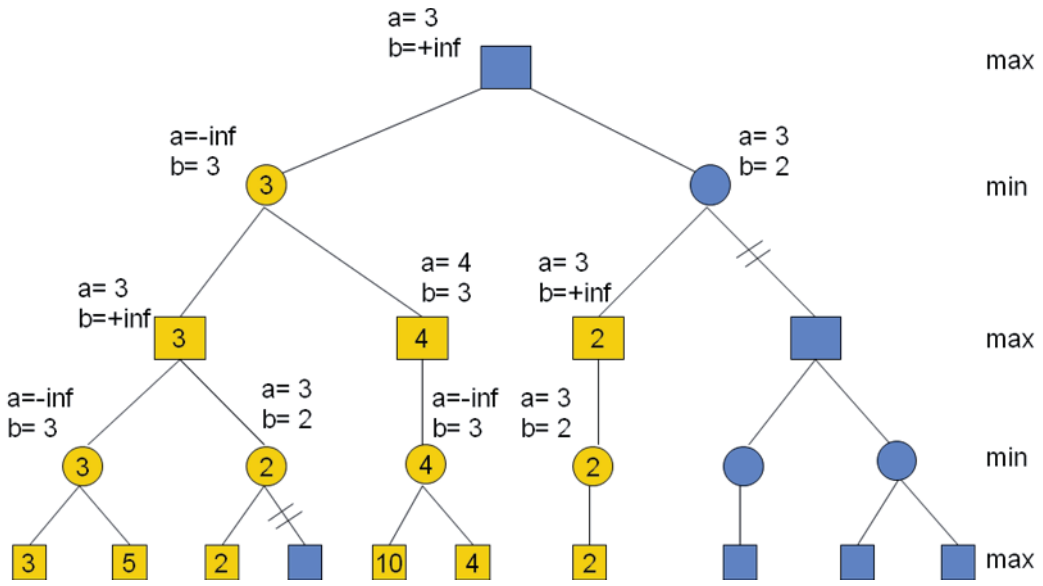
9) Continue propagating value up the tree, modifying the corresponding alpha/beta values. Also propagate the state of root node down the left-most path of the right subtree.



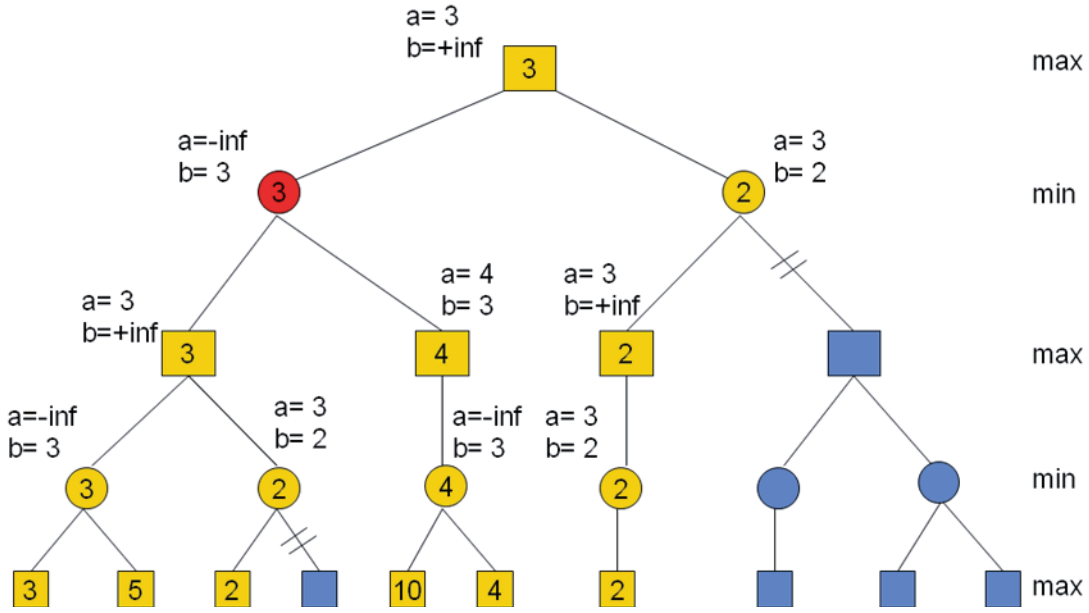
10) Next value is a 2. We set the beta (min) value of the min parent to 2. Since no other children exist, we propagate the value up the tree.



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.

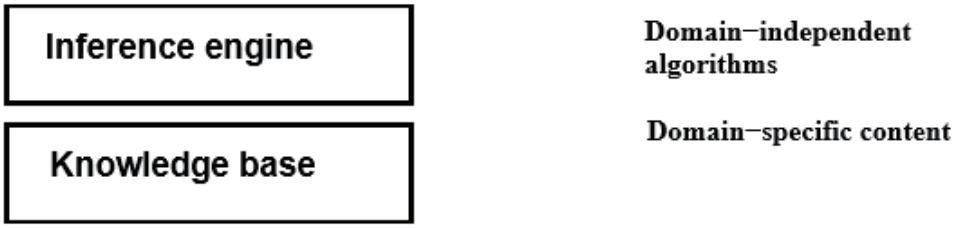


12) Finally, no more nodes remain, we propagate values up the tree. The root has a value of 3 that comes from the left-most child. Thus, the player should choose the left-most child's move in order to maximize his/her winnings. As you can see, the result is the same as with the mini-max example, but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



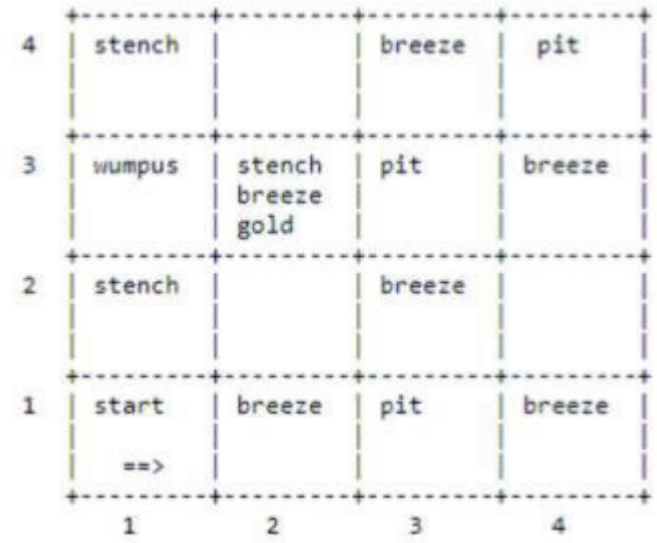
Knowledge Based Agents A knowledge-based agent needs a KB and an inference mechanism. It operates by storing sentences in its knowledge base, inferring new sentences with the inference mechanism, and using them to deduce which actions to take. ... The interpretation of a sentence is the fact to which it refers.

Knowledge Bases:



Knowledge base = set of sentences in a formal language Declarative approach to building an agent (or other system): Tell it what it needs toknow - Thenitcan Askitselfwhattodo—answersshouldfollowfromtheKB Agents can be viewed at the knowledge leveli.e., what they know, regardless of howimplemented or at the implementation leveli.e.,data structuresinKBand algorithmsthatmanipulatethem. The Wumpus World:

A variety of "worlds" are being used as examples for Knowledge Representation, Reasoning, and Planning. Among them the Vacuum World, the Block World, and the Wumpus World. The Wumpus World was introduced by Genesereth, and is discussed in Russell-Norvig. The Wumpus World is a simple world (as is the Block World) for which to represent knowledge and to reason. It is a cave with a number of rooms, represented as a 4x4 square



Rules of the Wumpus World The neighborhood of a node consists of the four squares north, south, east, and west of the given square. In a square the agent gets a vector of percepts, with components Stench, Breeze, Glitter, Bump, Scream For example [Stench, None, Glitter, None, None] Stench is perceived at a square iff the wumpus is at this square or in its neighborhood. Breeze is perceived at a square iff a pit is in the neighborhood of this square. Glitter is perceived at a square iff gold is in this square Bump is perceived at a square iff the agent goes Forward into a wall Scream is perceived at a square iff the wumpus is killed anywhere in the cave An agent can do the following actions (one at a time): Turn (Right), Turn (Left), Forward, Shoot, Grab, Release, Climb The agent can go forward in the direction it is currently facing, or Turn Right, or Turn Left. Going forward into a wall will generate a Bump percept. The agent has a single arrow that it can shoot. It will go straight in the direction faced by the agent until it hits (and kills) the wumpus, or hits (and is absorbed by) a wall. The agent can grab a portable object at the current square or it can Release an object that it is holding. The agent can climb out of the cave if at the Start square.The Start square is (1,1) and initially the agent is facing east. The agent dies if it is in the same square asthe wumpus. The objective of the game is to kill the wumpus, to pick up the gold, and to climb out with it. Representing our Knowledge about the Wumpus World Percept(x, y) Where x must be a percept vector and y must be a situation. It means that at situation y theagentperceives x.For convenience we introduce the following definitions: Percept([Stench,y,z,w,v],t) = > Stench(t) Percept([x,Breeze,z,w,v],t) = > Breeze(t) Percept([x,y,Glitter,w,v],t) = > AtGold(t) Holding(x, y)

Where x is an object and y is a situation. It means that the agent is holding the object x in situation y. Action(x, y) Where x must be an action (i.e. Turn (Right), Turn (Left), Forward,) and y must be a situation. It means that at situation y the agent takes action x. At(x,y,z) Where x is an object, y is a Location, i.e. a pair [u,v] with u

and v in {1, 2, 3, 4}, and z is a situation. It means that the agent x in situation z is at location y. Present(x,s) Means that object x is in the current room in the situation s. Result(x, y) It means that the result of applying action x to the situation y is the situation Result(x,y).Notethat Result(x,y) is a term, not a statement. For example we can say □ Result(Forward, S0) = S1 □ Result(Turn(Right),S1) = S2 These definitions could be made more general. Since in the Wumpus World there is a single agent, there is no reason for us to make predicates and functions relative to a specific agent. In other"worlds" we should change things appropriately.

Validity And Satisfiability

A sentence is valid

if it is true in all models, e.g.,True,A∨¬A, A⇒A,(A∧(A⇒B)) ⇒B Validity is connected to inference via the Deduction Theorem: KB |= αif and onlyif(KB⇒α) isvalid Asentenceissatisfiableifitistrue insome model e.g., A∨B, C Asentence isunsatisfiableifitistrueinnomodels e.g., A ∧¬A Satisfiability is connected to inference via the following: KB|=α iff(KB∧¬α)isunsatisfiable i.e., prove α by reductionandabsurdum

Proof Methods

Proof methods divide into (roughly)two kinds:

Application of inference rules – Legitimate(sound)generationofnewsentencesfromold – Proof=asequenceofinferenceruleapplicationscanuseinferencerulesasoperatorsinastand ardsearch algorithm – Typicallyrequiretranslationofsentencesintoanormalform Model checking – Truthtableenumeration(alwaysexponentialinn) – Improvedbacktracking,e.g.,Davis–Putnam–Loge Mann–Loveland – Heuristic searchinmodelspace(soundbutincomplete) e.g.,min-conflicts-likehillclimbingalgorithms

Forward and Backward Chaining

Horn Form (restricted) KB = conjunction of Horn clauses Horn clause = – proposition symbol;or – (conjunctionofsymbols) ⇒ symbol Example KB: C∧(B ⇒ A) ∧ (C∧D ⇒ B) Modus Ponens (for Horn Form): complete for Horn KBs

α1,...,αn,α1∧···∧α⇒ β β

Canbeusedwithforwardchaining orbackwardchaining. These algorithms areverynaturalandruninlineartime.,

ForwardChaining

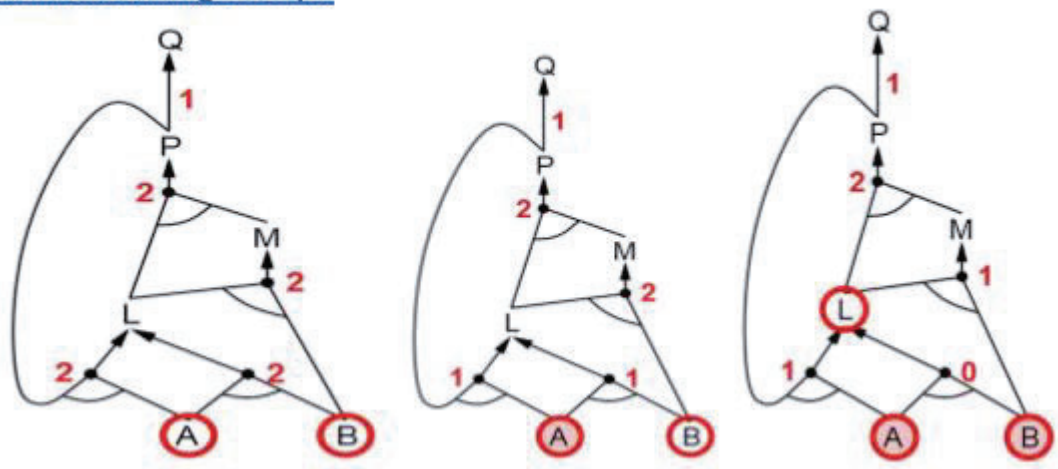
Idea: If anyrulewhosepremisesaresatisfiedintheKB, additsconclusiontotheKB,untilqueryisfound

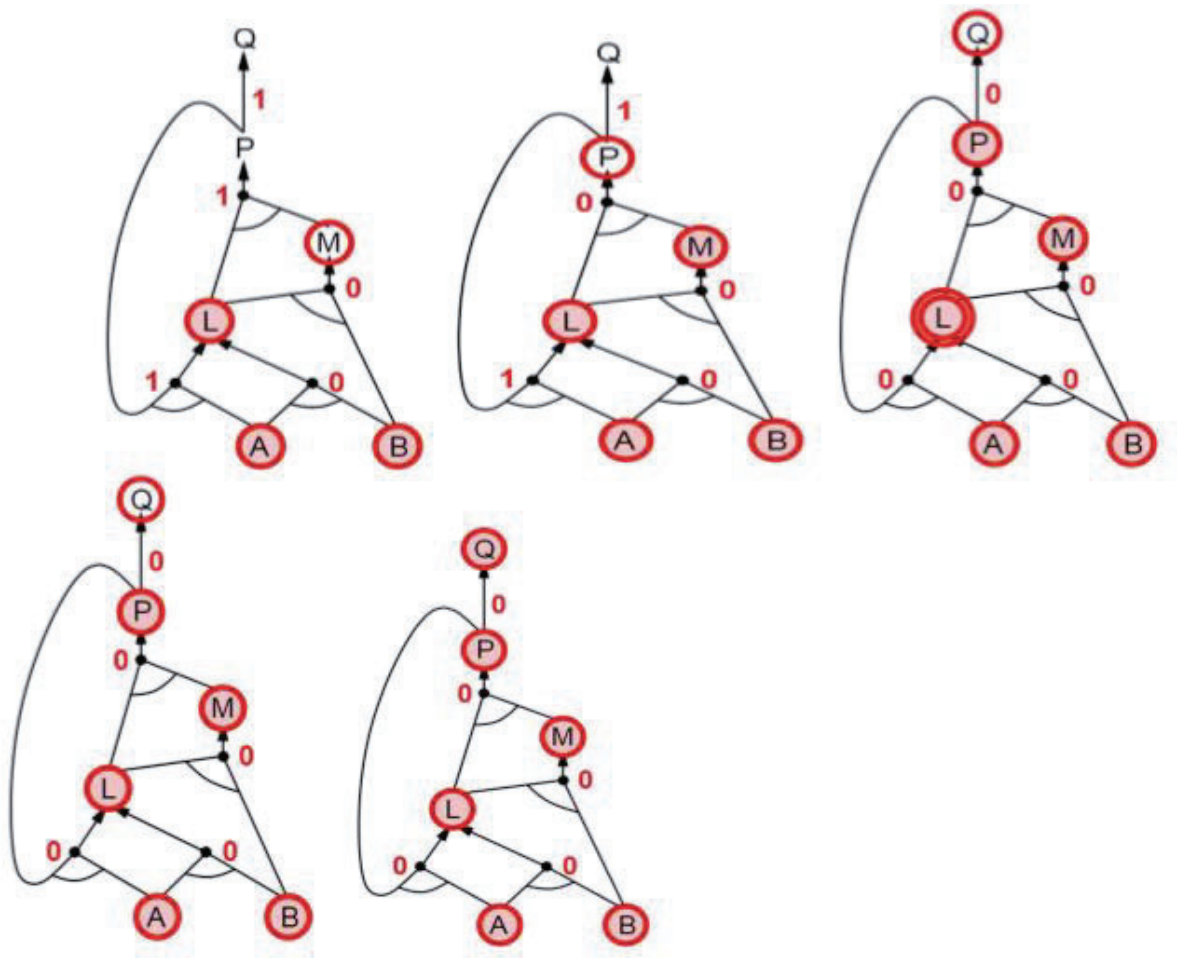
ForwardChaining Algorithm

ForwardChaining Algorithm

```
function PL-FC-Entails?(KB,q) returns true or false
  inputs: KB, the knowledge base, a set of propositional Horn clauses
          q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known in KB
  while agenda is not empty
    do p ← Pop(agenda)
    unless inferred[p] do
      inferred[p] ← true
      for each Horn clause c in whose premise p appears do
        decrement count[c]
```

ForwardChaining Example



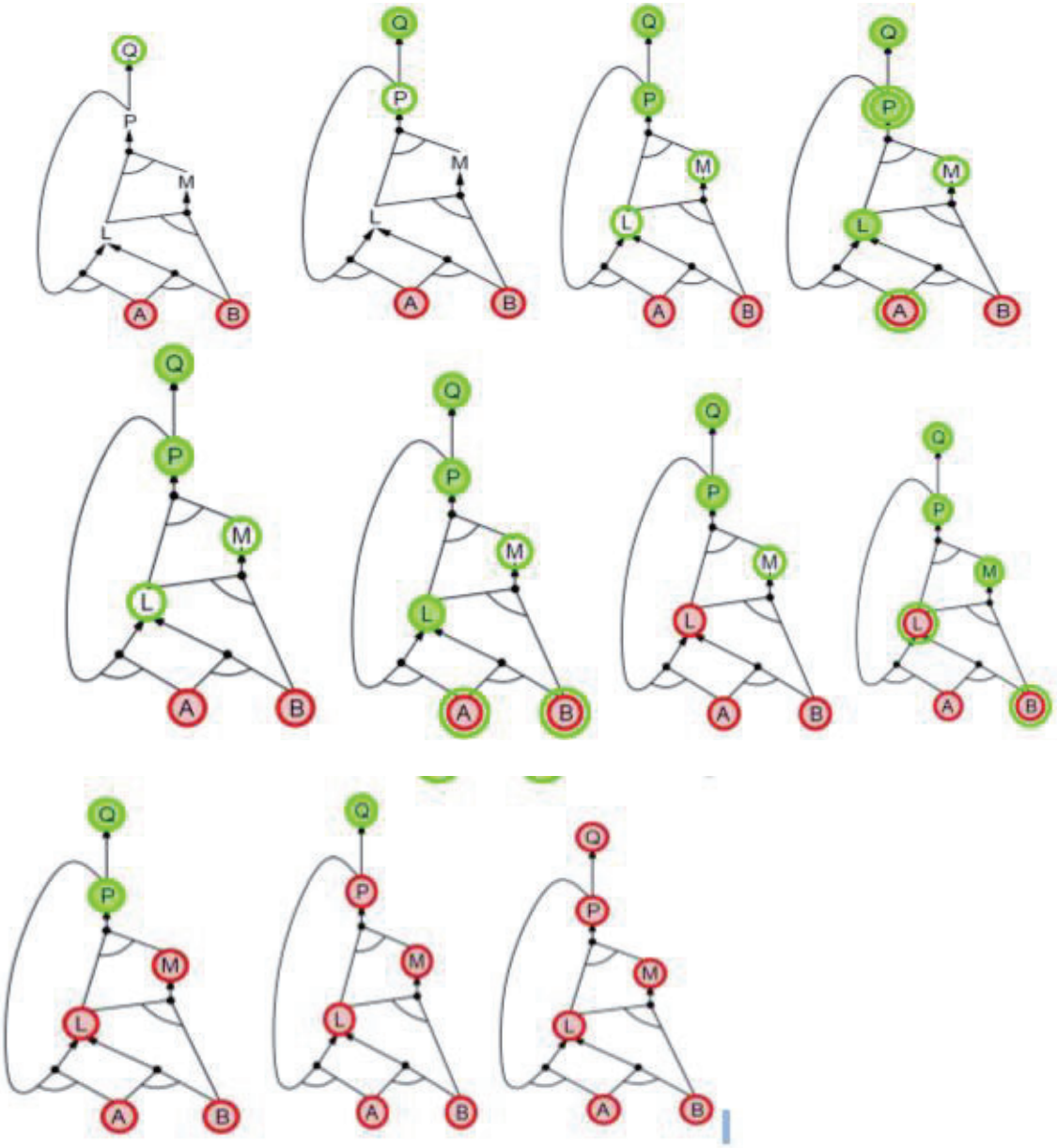


Proof of Completeness

FC derives every atomic sentence that is entailed by KB 1. FCreachesafixedpointwherenonewatomicsentencesarederived 2. Considerthefinalstateasamodelm,assigningtrue/falsetosymbols 3. Every clause in the original KB is true inm i. Proof:Supposeaclausea1∧...∧ak⇒bisfalsei nm Then a1∧. . . ∧akis true in m and b is false in m Thereforethealgorithmhasnotreachedafixedpoint ! 4. Hence m is a model ofKB 5. IfKB|=q,thenqistrueineverymodelofKB,includingm a. Generalidea: constructanymodelofKBby soundinference,checkα

Backward Chaining

Idea:workbackwardsfromthequeryq: to prove q byBC, check if q is known already, or prove by BC all premises of some rule concluding q Avoidloops: checkifnewsubgoalisalreadyonthegoalstack Avoid repeated work: check if new subgoal 1. has already been proved true, or 2. has alreadyfailed



Forward vs Backward Chaining

FC is data-driven, cf. automatic, unconscious processing, e.g.,objectrecognition,routinedecisions Maydolotsofworkthatisirrelevanttothegoal BC is goal-driven, appropriate forproblem-solving, e.g., Where are my keys? How do I get into a PhD program? Complexity of BC can be much less than linear in size of KB

These rules exhibit a trivial form of the reasoning process called perception.

Simple "reflex" behavior can also be implemented by quantified implication sentences.

For example, we have ∀tGlitter (t) ⇒BestAction(Grab, t) .

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion Best Action (Grab, 5)—that is, Grab is the right thing to do.

Environment Representation

Objects are squares, pits, and the wumpus. Each square could be named—Square1,2and so on—but then the fact that Square1,2and Square1,3 are adjacent would have to be an "extra" fact, and this needs one suchfact for each pair of squares. It is better to use a complex term in which the row and columnappear as integers;

For example, we can simply use the list term [1, 2].

Adjacency of any two squares can be defined as:

∀x, y, a, b Adjacent ([x, y], [a, b]) ⇔ (x = a ∧(y = b − 1 ∨y = b + 1)) ∨(y = b ∧(x = a − 1 ∨x = a + 1)).

Each pit need not be distinguished with each other. The unary predicate Pit is true of squares containing pits.

Since there is exactly one wumpus, a constant Wumpus is just as good as a unary predicate. The agent's location changes over time, so we write At (Agent, s, t) to mean that theagent is at square s at time t.

To specify the Wumpus location (for example) at [2, 2] we can write ∀t At (Wumpus, [2, 2], t).

Objects can only be at one location at a time: ∀x, s1, s2, t At(x, s1, t) ∧At(x, s2, t) ⇒s1 = s2 .

Given its current location, the agent can infer properties of the square from properties of its current percept

For example, if the agent is at a square and perceives a breeze, then that square is breezy:

∀s, t At(Agent, s, t) ∧Breeze(t) ⇒Breezy(s) .

It is useful to know that a square is breezy because we know that the pits cannot move about.

Breezy has no time argument.

Having discovered which places are breezy (or smelly) and, very importantly, not breezy (or not smelly), the agent can deduce where the pits =e (and where the wumpus is).

There are two kinds of synchronic rules that could allow such deductions:

Diagnostic rules:

Diagnostic rules lead from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit, or

$\forall s$ Breezy(s) $\Rightarrow \exists r$ Adjacent (r, s)$\wedge$Pit(r) ,

and that if a square is not breezy, no adjacent square contains a pit: $\forall s \neg$Breezy (s) $\Rightarrow \neg \exists r$ Adjacent (r, s) $\wedge$ Pit (,r) .Combining these two, we obtain the biconditional sentence $\forall s$ Breezy (s) $\Leftrightarrow \exists r$ Adjacent(r, s) $\wedge$ Pit (r) .

Causal rules:

Causal rules reflect the assumed direction of causality in the world: some hidden property of the world causes certain percepts to be generated. For example, a pit causes all adjacent squares to be breezy:

and if all squares adjacent to a given square are pitless, the square will not be breezy: $\forall s[\forall r$ Adjacent (r, s) $\Rightarrow \neg$Pit (r)] $\Rightarrow \neg$Breezy (s) .

It is possible to show that these two sentences together are logically equivalent to the biconditional sentence " $\forall s$ Breezy (s) $\Leftrightarrow \exists r$ Adjacent(r, s) $\wedge$ Pit (r)" .

The biconditional itself can also be thought of as causal, because it states how the truth value of Breezy is generated from the world state.

Systems that reason with causal rules are called model-based reasoning systems, because the causal rules form a model of how the environment operates.

Whichever kind of representation the agent uses, ifthe axioms correctly and completely describe the way the world works and the way that percepts are produced, then any complete logical inference procedure will infer the strongest possible description of the world state, given the available percepts. Thus, the agent designer can concentrate on getting the knowledgeright, without worrying too much about the processes of deduction.

FIRST ORDER LOGIC:

PROCEDURAL LANGUAGES AND PROPOSITIONAL LOGIC:

Drawbacks of Procedural Languages

 Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent only computational processes. Data structures within programs can represent facts.

For example, a program could use a 4 × 4 array to represent the contents of the wumpus world. Thus, the programming language statement World[2,2]← Pit is a fairly natural way to assert that there is a pit in square [2,2].

What programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain.

 A second drawback of is the lack the expressiveness required to handle partial information . For example data structures in programs lack the easy way to say, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].”

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 “S1,4∧ S1,2” is related to the meanings of “S1,4” and “S1,2.

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

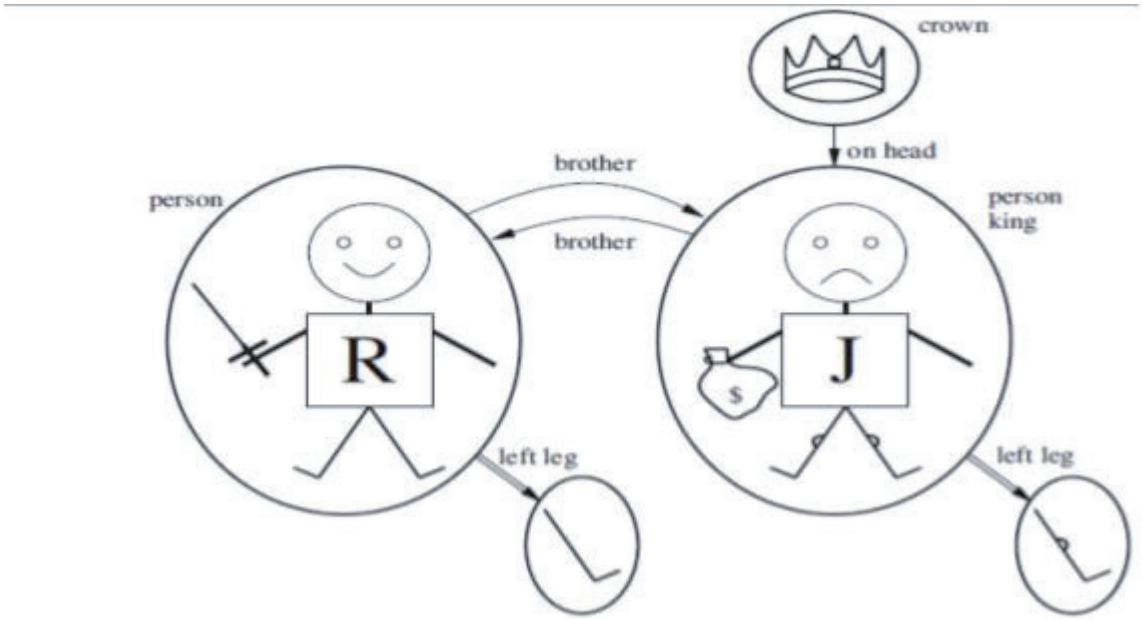
Models for first-order logic :

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 ofjohn 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 possiblemodels 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 anbounded, 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 (tl,. . . , t,). The function symbol frefers to some function in the model (F); the argument terms refer to objects in the domain (call them d1….dn); and the term as a whole refers to the object that is the value of the function Fapplied to dl, . . . , d,. For example,: the LeftLeg function symbol refers to the function " (King John) -+ 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:

- ✓ ¬Brother (LeftLeg(Richard), John)
- ✓ Brother (Richard , John) ∧ Brother (John, Richard)
- ✓ King(Richard) ∨ King(John)
- ✓ ¬King(Richard) ⇒ King(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 ∀x P, where P is any logical expression, says that P is true for every object x."
More precisely, ∀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
∀xKing(x) ⇒Person(x) .
∀ 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→ Richard the Lionheart, x→ King John, x→ Richard's left leg, x→ John's left leg, x→ the crown

The universally quantified sentence ∀x King(x) ⇒Person(x) is true in the original model if the sentence King(x) ⇒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 ⇒Richard the Lionheart is a person. King John is a king ⇒King John is a person. Richard's left leg is a king ⇒Richard's left leg is a person. John's left leg is a king ⇒John's left leg is a person. The crown is a king ⇒the crown is a person.

Existential quantification (∃)

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 ∃x P says that P is true for at least one object x. More precisely, ∃x P is true in a given model if P is true in at least one extended interpretationthat assigns x to a domain element." ∃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 ∃xCrown(x) ∧OnHead(x, John)

Given assertions:

Richard the Lionheart is a crown ∧Richard the Lionheart is on John's head; King John is a crown ∧King John is on John's head; Richard's left leg is a crown ∧Richard's left leg is on John's head; John's left leg is a crown ∧John's left leg is on John's head; The crown is a crown ∧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 ⇒appears to be the natural connective to use with ∀, ∧is the natural connective to use with ∃.

Nested quantifiers

One can express more complex sentences using multiple quantifiers.

For example, "Brothers are siblings" can be written as ∀x∀y Brother (x, y) ⇒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∀x, y Sibling(x, y) ⇔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: ∀x∃y Loves(x, y) . 2. On the other hand, to say "There is someone who is loved by everyone," we write ∃y∀x Loves(x, y) .

Connections between ∀and ∃

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: "∀x ¬Likes(x, medicine)" is equivalent to "¬∃x Likes(x, medicine)".

2. “Everyone likes ice cream” means that “ there is no one who does not like ice cream” : ∀xLikes(x, IceCream) is equivalent to ¬∃x ¬Likes(x, IceCream) .

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

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

$$\forall x\ \neg P \equiv \neg\exists x\ P \qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$$
$$\neg\forall x\ P \equiv \exists x\ \neg P \qquad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\forall x\ P \equiv \neg\exists x\ \neg P \qquad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$$
$$\exists x\ P \equiv \neg\forall x\ \neg P \qquad P \vee Q \equiv \neg(\neg P \wedge \neg Q)\ ..$$

Thus, Quantifiers are important in terms of readability.

Equality

First-order logic includes one more way to make atomic sentences, other than using a predicateand 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, ∃x, y Brother (x,Richard) ∧Brother (y,Richard) ∧¬(x=y) .

The sentence

∃x, y Brother (x,Richard) ∧Brother (y,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 ¬(x=y) rules out such models.

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\textit{Sentence}) \mid [\textit{Sentence}] \\
& \mid & \neg \textit{Sentence} \\
& \mid & \textit{Sentence} \wedge \textit{Sentence} \\
& \mid & \textit{Sentence} \vee \textit{Sentence} \\
& \mid & \textit{Sentence} \Rightarrow \textit{Sentence} \\
& \mid & \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
& \mid & \textit{Quantifier Variable}, \ldots \textit{Sentence} \\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term}, \ldots) \\
& \mid & \textit{Constant} \\
& \mid & \textit{Variable} \\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

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, ∀x King(x) ⇒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.

Anyquery that is logically entailed by the knowledge base should be answered affirmatively.

Forexample, 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 ASK (KB, ∃x 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 ASKVARS (KB, Person(x)) and which yields a stream of answers.

In this case there will be two answers: {x/John} and {x/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 termsof the other symbols.

For example:- 1. one's mother is one's female parent: ∀m, c Mother (c)=m ⇔Female(m) ∧Parent(m, c) .

2. One's husband is one's male spouse: ∀w, h Husband(h,w) ⇔Male(h) ∧Spouse(h,w) .

3. Male and female are disjoint categories: ∀xMale(x) ⇔¬Female(x) .

4. Parent and child are inverse relations: ∀p, c Parent(p, c) ⇔Child (c, p) .

5. A grandparent is a parent of one's parent: $\forall g, c$ Grandparent $(g, c) \Leftrightarrow \exists p$ Parent$(g, p) \wedge$Parent(p, c) .

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

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: $\forall x, y$ Sibling$(x, y) \Leftrightarrow$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

∀xPerson(x) ⇔. . .

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:

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

Natural numbers are defined recursively: NatNum(0) . ∀n NatNum(n) ⇒ 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: ∀n 0 != S(n) . ∀m, n m != n ⇒ S(m) != S(n) .

Now we can define addition in terms of the successor function: ∀m NatNum(m) ⇒ + (0, m) = m . ∀m, n NatNum(m) ∧ NatNum(n) ⇒ + (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 :

∀m, n NatNum (m) ∧ NatNum(n) ⇒ (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 individualsets, 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∈ s (x is a member of set s) s1 ⊆ s2 (set s1 is a subset, not necessarily proper, of set s2).

The binary functions are

 s1 ∩ s2 (the intersection of two sets), s1 ∪ s2 (the union of two sets), and {x|s} (the set resulting from adjoining element x to set s).

One possible set of axioms is as follows:

 The only sets are the empty set and those made by adjoining something to a set: ∀sSet(s) ⇔(s={}) ∨(∃x, s2 Set(s2) ∧s={x|s2}) . The empty set has no elements adjoined into it. In other words, there is no way to decompose {} into a smaller set and an element: ¬∃x, s {x|s}={} . Adjoining an element already in the set has no effect: ∀x, s x∈s ⇔s={x|s} . The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s2 adjoined with some element y, where either y is the same as x or x is a member of s2: ∀x, s x∈s ⇔∃y, s2 (s={y|s2} ∧(x=y ∨x∈s2)) A set is a subset of another set if and only if all of the first set's members are members of the

second set: ∀s1, s2 s1 ⊆s2 ⇔(∀x x∈s1 ⇒x∈s2) Two sets are equal if and only if each is a subset of the other: ∀s1, s2 (s1 =s2) ⇔(s1 ⊆s2 ∧s2 ⊆s1)

 An object is in the intersection of two sets if and only if it is a member of both sets: ∀x, s1, s2 x∈(s1 ∩ s2) ⇔(x∈s1 ∧x∈s2) An object is in the union of two sets if and only if it is a member of either set: ∀x, s1, s2 x∈(s1 ∪s2) ⇔(x∈s1 ∨x∈s2)

Lists : are similar to sets. The differences are that lists are ordered and the same element canappear more than once in a list. We can use the vocabulary of Lisp for lists:

 Nil is the constant list with no elements; Cons, Append, First, and Rest are functions; Find is the predicate that does for lists what Member does for sets. List? is a predicate that is true only of lists. The empty list is [ ]. The term Cons(x, y), where y is a nonempty list, is written [x|y]. The term Cons(x, Nil) (i.e., the list containing the element x) is written as [x]. A list of several elements, such as [A,B,C], corresponds to the nested term Cons(A, Cons(B, Cons(C, Nil))).

The wumpus world

Agents Percepts and Actions

The wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what.We use integers for time steps. A typical percept sentence would be

Percept ([Stench, Breeze, Glitter,None, None], 5).

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

Turn (Right), Turn (Left), Forward,Shoot,Grab, Climb.

To determine which is best, the agent program executes the query:

ASKVARS (∃a BestAction (a, 5)), which returns a binding list such as {a/Grab}.

The agent program can then return Grab as the action to take.

The raw percept data implies certain facts about the current state.

For example: ∀t, s, g, m, c Percept ([s, Breeze, g,m, c], t) ⇒Breeze(t) , ∀t, s, b, m, c Percept ([s, b, Glitter,m, c], t) ⇒Glitter (t) ,

Propositional Vs First Order Inference

Earlier inference in first order logic is performed with *Propositionalization* which is a process of converting the Knowledgebase present in First Order logic into Propositional logic and on that using any inference mechanisms of propositional logic are used to check inference.

Inference rules for quantifiers:

There are some Inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers**.** These rules will lead us to make the conversion.

Universal Instantiation (UI):

The rule says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. Let SUBST (θ*)* denote the result of applying the substitution θ to the sentence *a.* Then the rule is written

$$\frac{\forall v \ \ a}{\text{SUBST}(\{v/g\}, \alpha)}$$

For any variable v and ground term *g.*

For example, there is a sentence in knowledge base stating that all greedy kings are Evils

$$\forall x \ \ King(x) \ A \ Greedy(x) \Rightarrow \ Evil(x) \ .$$

For the variable x, with the substitutions like {x/John},{x/Richard}the following sentences can be inferred.

$$King(John) \ A \ Greedy(John) \ \Rightarrow \ Evil(John).$$
$$King(Richard) \ A \ Greedy(Richard) \ \Rightarrow \ Evil(Richard) \ .$$

Thus a universally quantified sentence can be replaced by the set of *all* possible instantiations.

Existential Instantiation (EI)*:*

The existential sentence says there is some object satisfying a condition, and the instantiation process is just giving a name to that object, that name must not already belong to another object. This new name is called a **Skolem constant.** Existential Instantiation is a special case of a more general process called *"skolemization"*.

For any sentence *a,* variable v, and constant symbol *k* that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

For example, from the sentence

$$\exists x \ \ Crown(x) \wedge OnHead(x, John)$$

So, we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

As long as C_1 does not appear elsewhere in the knowledge base. Thus an existentially quantified sentence can be replaced by one instantiation

Elimination of Universal and Existential quantifiers should give new knowledge base which can be shown to be *inferentially equivalent* to old in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

Reduction to propositional inference:

Once we have rules for inferring non quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. For example, suppose our knowledge base contains just the sentences

$$\forall x \ \ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John).$$

Then we apply UI to the first sentence using all possible ground term substitutions from the vocabulary of the knowledge base-in this case, *{xl John)* and *{x/Richard).* We obtain

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John),$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$

We discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences-King *(John), Greedy (John),* and Brother (Richard, *John*) as proposition symbols. Therefore, we can apply any of the complete propositional algorithms to obtain conclusions such as *Evil (John).*

Disadvantage:

If the knowledge base includes a function symbol, the set of possible ground term substitutions is infinite. Propositional algorithms will have difficulty with an infinitely large set of sentences.

NOTE:

Entailment for first-order logic is *semi decidable* which means algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non entailed sentence

2. Unification and Lifting

Consider the above discussed example, if we add Siblings (Peter, Sharon) to the knowledge base then it will be

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John)$$
$$\mathbf{Siblings(Peter, Sharon)}$$

Removing Universal Quantifier will add new sentences to the knowledge base which are not necessary for the query *Evil (John)?*

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Peter) \wedge Greedy(Peter) \Rightarrow Evil(Peter)$$
$$King(Sharon) \wedge Greedy(Sharon) \Rightarrow Evil(Sharon)$$

Hence we need to teach the computer to make better inferences. For this purpose Inference rules were used.

First Order Inference Rule:

The key advantage of lifted inference rules over *propositionalization* is that they make only those substitutions which are required to allow particular inferences to proceed.

Generalized Modus Ponens:

If there is some substitution **θ** that makes the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying **θ**. This inference process can be captured as a single inference rule called General ized Modus Ponens which is a ***lifted*** version of Modus Ponens-it raises Modus Ponens from propositional to first-order logic

For atomic sentences pi, pi ', and q, where there is a substitution θ such that SUBST(θ , pi) = SUBST(θ , pi '), for all i,

$$\mathbf{p1\ ',\ p2\ ',\ \ldots,\ pn\ ',\ (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}$$

SUBST (θ, q)

There are N + 1 premises to this rule, N atomic sentences + one implication.

Applying SUBST (θ, q) yields the conclusion we seek. It is a sound inference rule.

Suppose that instead of knowing Greedy (John) in our example we know that everyone is greedy:

∀y Greedy(y)

We would conclude that Evil(John).

Applying the substitution {x/John, y / John) to the implication premises King (x) and Greedy (x) and the knowledge base sentences King(John) and Greedy(y) will make them identical. Thus, we can infer the conclusion of the implication.

For our example,

p_1' is $King(John)$ $\quad$ p_1 is $King(x)$
p_2' is $Greedy(y)$ $\quad$ p_2 is $Greedy(x)$
θ is $\{x/John, y/John)$ $\quad$ q is $Evil(x)$
SUBST(θ, q) is $Evil(John)$.

Unification:

It is the process used to find substitutions that make different logical expressions look identical. **Unification** is a key component of all first-order Inference algorithms.

UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q) θ is our unifier value (if one exists).

Ex: "Who does John know?"

UNIFY (Knows (John, x), Knows (John, Jane)) = {x/ Jane}.

UNIFY (Knows (John, x), Knows (y, Bill)) = {x/Bill, y/ John}.

UNIFY (Knows (John, x), Knows (y, Mother(y))) = {x/Bill, y/ John}

UNIFY (Knows (John, x), Knows (x, Elizabeth)) = FAIL

- The last unification fails because both use the same variable, X. X can't equal both John and Elizabeth. To avoid this change the variable X to Y (or any other value) in Knows(X, Elizabeth)

Knows(X, Elizabeth) → Knows(Y, Elizabeth)

Still means the same. This is called **standardizing apart.**

- sometimes it is possible for more than one unifier returned:

UNIFY (Knows (John, x), Knows(y, z)) =???

This can return two possible unifications: {y/ John, x/ z} which means Knows (John, z) OR {y/ John, x/ John, z/ John}. For each unifiable pair of expressions there is a single **most general unifier (MGU)**, In this case it is *{y/ John, x/z)*.

An algorithm for computing most general unifiers is shown below.

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

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

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

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

Figure 2.1 *The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution 0 that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression, such as F (A, B), the function OP picks out the function symbol F and the function*

ARCS picks out the argument list (A, B).

The process is very simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. **Occur check** step makes sure same variable isn't used twice.

Storage and retrieval

- STORE(s) stores a sentence *s* into the knowledge base
- FETCH(s) returns all unifiers such that the query q unifies with some sentence in the knowledge base.

Easy way to implement these functions is Store all sentences in a long list, browse list one sentence at a time with UNIFY on an ASK query. But this is inefficient.

To make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. (i.e. Knows(John, x) vs. Brother(Richard, John) are not compatible for unification)

- To avoid this, a simple scheme called ***predicate indexing*** puts all the *Knows* facts in one bucket and all the *Brother* facts in another.
- The buckets can be stored in a hash table for efficient access. Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol.

But if we have many clauses for a given predicate symbol, facts can be stored under multiple index keys.

For the fact *Employs (AIMA.org, Richard),* the queries are
Employs (A IMA. org, Richard) Does AIMA.org employ Richard?
Employs (x, Richard) who employs Richard?
Employs (AIMA.org, y) whom does AIMA.org employ?
Employs Y(x), who employs whom?

We can arrange this into a **subsumption lattice,** as shown below**.**

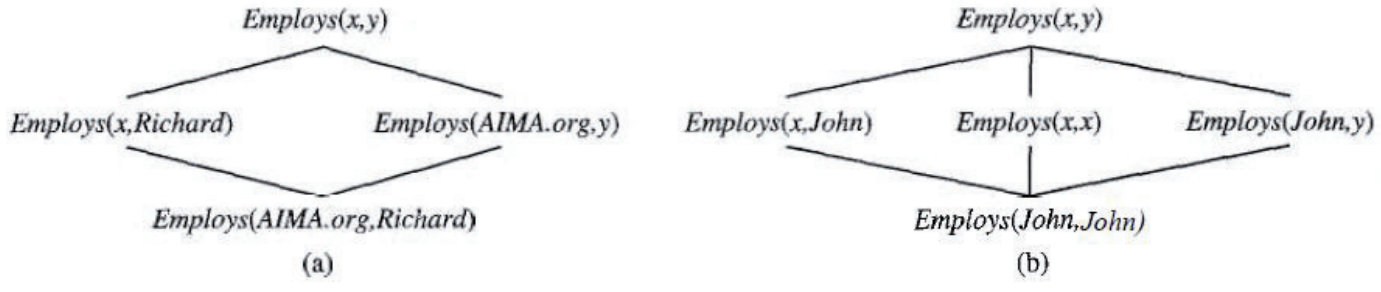


Figure 2.2 *(a) The subsumption lattice whose lowest node is the sentence Employs (AIMA.org, Richard). (b) The subsumption lattice for the sentence Employs (John, John).*

A subsumption lattice has the following properties:

- ✓ child of any node obtained from its parents by one substitution
- ✓ the "highest" common descendant of any two nodes is the result of applying their most general unifier
- ✓ predicate with n arguments contains O(2n) nodes (in our example, we have two arguments, so our lattice has four nodes)
- ✓ Repeated constants = slightly different lattice.

3. Forward Chaining

First-Order Definite Clauses:

A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.

Consider the following problem;

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

We will represent the facts as first-order definite clauses

". . . It is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$ --------- (1)

"Nono . . . has some missiles." The sentence 3 x *Owns (Nono,* .rc) *A Missile* (x) is transformed into two definite clauses by Existential Elimination, introducing a new constant *M1:*

Owns (Nono, M1) ---------- (2)

Missile (Ml) ---------- (3*)*

"All of its missiles were sold to it by Colonel West":

Missile (x) A Owns (Nono, x) =>Sells (West, z, Nono) ---------- (4)

We will also need to know that missiles are weapons:

Missile (x) => Weapon (x) ---------- (5)

We must know that an enemy of America counts as "hostile":

Enemy (x, America) =>Hostile(x) ---------- (6)

"West, who is American":

American (West) **---------- (7)**

"The country Nono, an enemy of America ":

Enemy (Nono, America) ---------- (8)

A simple forward-chaining algorithm:

- Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions lo the known facts
- The process repeats until the query is answered or no new facts are added. Notice that a fact is not "new" if it is just *renaming* of a known fact.

We will use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (1), (4), (5), and (6). Two iterations are required:

- ✓ On the first iteration, rule (1) has unsatisfied premises.

Rule (4) is satisfied with *{x/Ml),* and *Sells (West,* M1, *Nono)* is added.

Rule (5) is satisfied with *{x/M1) and Weapon (M1)* is added.

Rule (6) is satisfied with *{x/Nono},* and *Hostile (Nono)* is added.

- ✓ On the second iteration, rule (1) is satisfied with *{x/West, Y/MI, z /Nono),* and *Criminal* (*West)* is added.

It is ***sound*****,** because every inference is just an application of Generalized Modus Ponens, it is ***complete*** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses

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

  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of some sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', a)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figure 3.1 *A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB.*

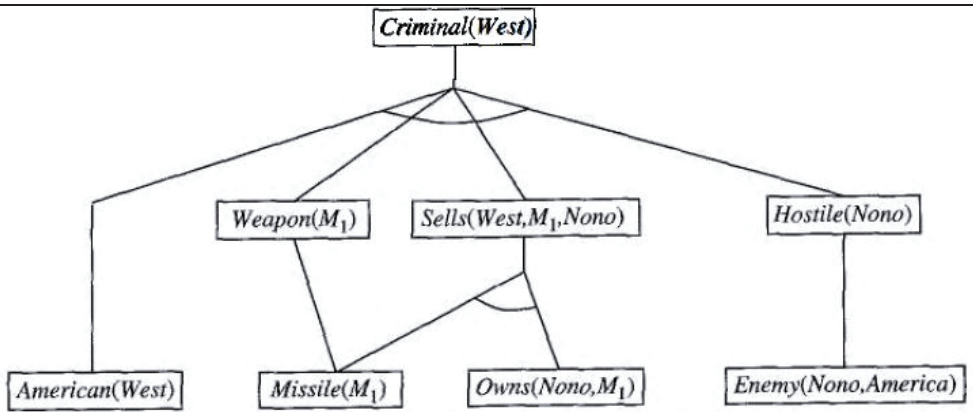


Figure 3.2 *The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.*

Efficient forward chaining:

The above given forward chaining algorithm was lack with efficiency due to the the three sources of complexities:

- ✓ Pattern Matching
- ✓ Rechecking of every rule on every iteration even a few additions are made to rules
- ✓ Irrelevant facts

1. ***Matching rules against known facts:***

For example, consider this rule,

Missile(x) A Owns (Nono, x) => Sells (West, x, Nono).

The algorithm will check all the objects owned by Nono in and then for each object, it could check whether it is a missile. This is the ***conjunct ordering problem:***

"Find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized". The **most constrained variable** heuristic used for CSPs would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains-for example, Missile(x) is a unary constraint on x. Extending this idea, we can express every finite-domain CSP as a single definite clause together with some associated ground facts. Matching a definite clause against a set of facts is NP-hard

2. Incremental forward chaining:

On the second iteration, the rule ***Missile (x) => Weapon (x)***

Matches against Missile (M1) (again), and of course the conclusion Weapon(x/M1) is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation:

"Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t – 1".

This observation leads naturally to an incremental forward chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p, that unifies with a fact p: newly inferred at iteration t - 1. The rule matching step then fixes p, to match with p', but allows the other conjuncts of the rule to match with facts from any previous iteration.

3. Irrelevant facts:

- One way to avoid drawing irrelevant conclusions is to use backward chaining.
- Another solution is to restrict forward chaining to a selected subset of rules
- A third approach, is to rewrite the rule set, using information from the goal.so that only relevant variable bindings-those belonging to a so-called **magic** set-are considered during forward inference.

For example, if the goal is Criminal (West), the rule that concludes Criminal (x) will be rewritten to include an extra conjunct that constrains the value of x:

Magic(x) A American(z) A Weapon(y)A Sells(x, y, z) A Hostile(z) =>Criminal(x)

The fact *Magic (West)* is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process.

4. Backward Chaining

This algorithm work backward from the goal, chaining through rules to find known facts that support the proof. It is called with a list of goals containing the original query, and returns the set of all substitutions satisfying the query. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose **head,** unifies with the goal. Each such clause creates a new recursive call in which **body,** of the clause is added to the goal stack .Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new sub goals are added to the stack and the goal is solved. The algorithm for backward chaining and proof tree for finding criminal (West) using backward chaining are given below.

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

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

Figure 4.1 *A simple backward-chaining algorithm.*

.

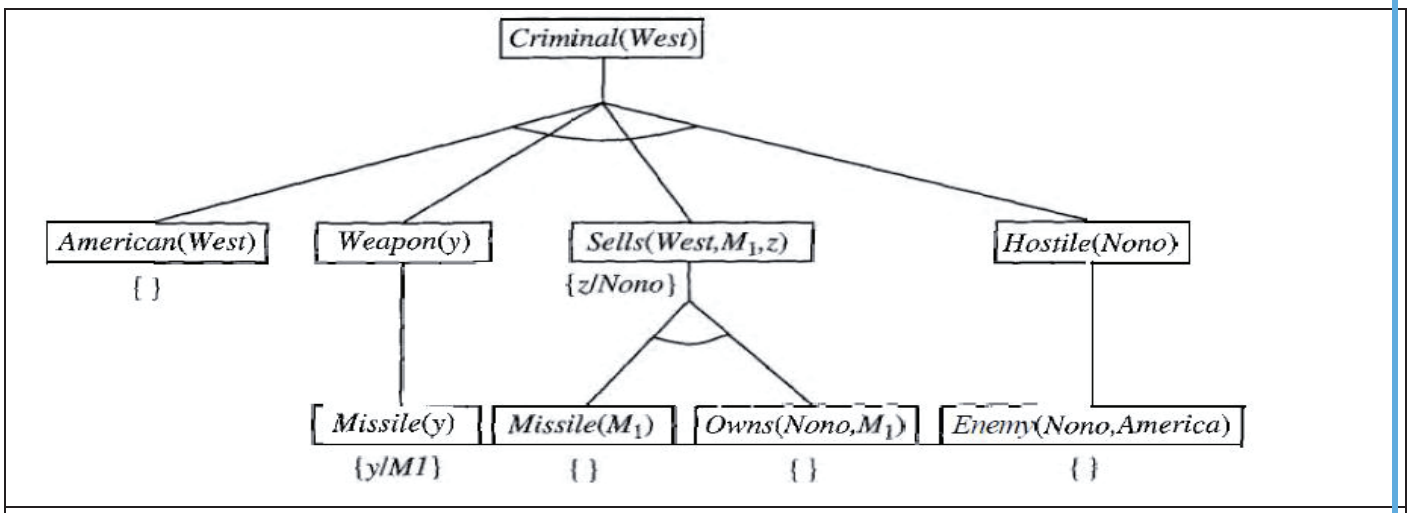


Figure 4.2 *Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding sub goal. Note that once one sub goal in a conjunction succeeds, its substitution is applied to subsequent sub goals.*

UNIT III

Knowledge Representation Issues

Background Knowledge and Observations

An **observation** is a piece of information received online from users, sensors, or other knowledge sources. For this chapter, we assume an observation is an atomic proposition. Observations are implicitly conjoined, so a set of observations is a conjunction of atoms. Neither users nor sensors provide rules directly from observing the world. The background knowledge allows the agent to do something useful with these observations.

In many reasoning frameworks, the observations are added to the background knowledge. But in other reasoning frameworks (e.g, in abduction, probabilistic reasoning, and learning), observations are treated separately from background knowledge. **Users** cannot be expected to tell us everything that is true. First, they do not know what is relevant, and second, they do not know what vocabulary to use. An **ontology** that specifies the meaning of the symbols, and a graphical user interface to allow the user to click on what is true, may help to solve the vocabulary problem. However, many problems are too big; what is relevant depends on other things that are true, and there are too many possibly relevant truths to expect the user to specify everything that is true, even with a sophisticated graphical user interface.

Similarly, passive sensors are able to provide direct observations of conjunctions of atomic propositions, but active sensors may have to be queried by the agent for the information that is necessary for a task.

Querying the User

At design time or offline, there is typically no information about particular cases. This information arrives online from users, sensors, and external knowledge sources. For example, a medical-diagnosis program may have knowledge represented as definite clauses about the possible diseases and symptoms but it would not have knowledge about the actual symptoms manifested by a particular patient. You would not expect that the user would want to, or even be able to, volunteer all of the information about a particular case because often the user does not know what information is relevant or know the syntax of the representation language. The user would prefer to answer explicit questions put to them in a more natural language. The idea of **querying the user** is that the system can treat the user as a source of information and ask the user specific questions about a particular case. The proof procedure can determine what information is relevant and will help to prove a query.

The simplest way to get information from a user is to incorporate an **ask-the-user** mechanism into the top-down proof procedure. In such a mechanism, an atom is **askable** if the user would know the truth value at run time. The top-down proof procedure, when it has selected an atom to prove, either can use a clause in the knowledge base to prove it, or, if the atom is askable, can ask the user whether or not the atom is true. The user is thus only asked about atoms that are relevant for the query. There are three classes of atoms that can be selected:

- atoms for which the user is not expected to know the answer, so the system never asks.
- askable atoms for which the user has not already provided an answer. In this case, the user should be asked for the answer, and the answer should be recorded.
- askable atoms for which the user has already provided an answer. In this case, that answer should be used, and the user should not be asked again about this atom.

A bottom-up proof procedure can also be adapted to ask a user, but it should avoid asking about all askable atoms; see Exercise 5.5.

It is important to note that a symmetry exists between roles of the user and roles of the system. They can both ask questions and give answers. At the top level, the user asks the system a question, and at each step the system asks a question, which is answered either by finding the relevant definite clauses or by asking the user. The whole system can be characterized by a protocol of questions and answers.

Example 5.10: In the electrical domain of Example 5.5, one would not expect the designer of the house to know the switch positions or expect the user to know which switches are connected to which wires. It is reasonable that all of the definite clauses of Example 5.5, except for the switch positions, should be given by the designer. The switch positions can then be made askable.

Here is a possible dialog, where the user asks a query and answers **yes** or **no**. The user interface here is minimal to show the basic idea; a real system would use a more sophisticated user-friendly interface.

ailog: **ask *lit_l_1*.**

Is *up_s_1* true? [yes,no,unknown,why,help]: **no.**

Is *down_s_1* true? [yes,no,unknown,why,help]: **yes.**

Is *down_s_2* true? [yes,no,unknown,why,help]: **yes.**

Answer: *lit_l_1*.

The system only asks the user questions that the user is able to answer and that are relevant to the task at hand.
Instead of answering questions, it is sometimes preferable for a user to be able to specify if there is something strange or unusual going on. For example, a patient may not be able to specify everything that is true about them but can specify what is unusual. For example, a patient may come in and say that their left knee hurts; it is unreasonable to expect them to volunteer that their left elbow doesn't hurt and, similarly, for every other part that does not hurt. It may be possible for a sensor to specify that something has changed in a scene, even though it may not be able to recognize what is in a scene.
Given that a user specified everything that is exceptional, an agent can often infer something from the lack of knowledge. The normality will be a **default** that can be overridden with exceptional information.

Knowledge-Level Explanation

The explicit use of semantics allows explanation and debugging at the **knowledge level**. To make a system usable by people, the system cannot just give an answer and expect the user to believe it. Consider the case of a system advising doctors who are legally responsible for the treatment that they carry out based on the diagnosis. The doctors must be convinced that the diagnosis is appropriate. The system must be able to justify that its answer is correct. The same features are used to explain how the system found a result and to debug the knowledge base.
Three complementary means of interrogation are used to explain the relevant knowledge: (1) a **how question** is used to explain how an answer was derived, (2) a **why question** is used to ask the system why it is asking the user a question, and (3) a **whynot question** is used to ask why an atom was not able to be proved.
To explain how an answer was derived, a "**how**" question can be asked by a user when the system has returned the answer. The system provides the definite clause used to deduce the answer. For any atom in the body of the definite clause, the user can ask how the system derived that atom.
The user can ask "**why**" in response to being asked a question. The system replies by giving the rule that produced the question. The user can then ask why the head of that rule was being proved. Together these rules allow the user to traverse a proof or a partial proof of the top-level query.
A "**whynot**" question can be used to ask why a particular atom was not able to be proved.

- 5.3.3.1 How Did the System Prove a Goal?
- 5.3.3.2 Why Did the System Ask a Question?

How Did the System Prove a Goal?

The first explanation procedure allows the user to ask "***how***" a goal was derived. If there is a proof for g, either g must be an atomic clause or there must be a rule

$g \leftarrow a_1 \wedge \ldots \wedge a_k$.

such that a proof exists for each a_i.

If the system has derived g, and the user asks ***how*** in response, the system can display the clause that was used to prove g. If this clause was a rule, the user can then ask

how i.

which will give the rule that was used to prove a_i. The user can continue using the ***how*** command to explore how g was derived.

Example 5.11: In the axiomatization of Example 5.5, the user can ask the query ***ask*** *lit_l2*. In response to the system proving this query, the user can ask ***how***. The system would reply:

lit_l2 ←
    light_l2 ∧
    live_l2 ∧
    ok_l2.

This is the top-level rule used to prove *lit_l2*. To find out how *live_l2* was proved, the user can ask

how 2.

The system can return the rule used to prove *live_l2*, namely,

live_l2←
    live_w4.

To find how *live_w4* was proved, the user can ask

how 1.

The system presents the rule

live_w4 ←
    live_w3 ∧
    up_s3.

To find how first atom in the body was proved, the user can ask

how 1.

The first atom, *live_w3*, was proved using the following rule:

live_w3 ←
    live_w5 ∧
    ok_cb1.

To find how the second atom in the body was proved, the user can ask

how 2.

The system will report that *ok_cb1* is explicitly given.

Notice that the explanation here was only in terms of the knowledge level, and it only gave the relevant definite clauses it has been told. The user does not have to know anything about the proof procedure or the actual computation.

Why Did the System Ask a Question?

Another useful explanation is for why a question was asked. This is useful because

- We want the system to appear intelligent. Knowing why a question was asked will increase a user's confidence that the system is working sensibly.
- One of the main measures of complexity of an interactive system is the number of questions asked of a user; you want to keep this to a minimum. Knowing why a question was asked will help the knowledge designer reduce this complexity.
- An irrelevant question is usually a symptom of a deeper problem.
- The user may learn something from the system by knowing why the system is doing something. This learning is much like an apprentice asking a master why the master is doing something.

When the system asks the user a question (q), there must be a rule used by the system that contains q in the body. The user can ask

why.

This is read as "Why did you ask me that question?" The answer can be simply the rule that contains q in the body. If the user asks ***why*** again, the system should explain why the goal at the head of the rule was asked, and so forth. Repeatedly asking ***why*** will eventually give the path of goals to the top-level query. If all of these rules are reasonable, this justifies why the system's question to the user is reasonable.

Example 5.12: Consider the dialog of Example 5.10. The following shows how repeated use of ***why*** can repeatedly find higher-level goals. The following dialog is for the query ***ask*** lit_l_1, with user asking the initial query, and responding with "**why**":

ailog: **ask** ***lit_l1*.**
Is up_s_1 true? **why**.
up_s_1 is used in the rule $live\_w_1 \leftarrow live\_w_3 \wedge up\_s_1$: **why**.
$live\_w_1$ is used in the rule $live\_w_0 \leftarrow live\_w_1 \wedge up\_s_2$: **why**.
$live\_w_0$ is used in the rule $live\_l_1 \leftarrow live\_w_0$: **why**.
$live\_l_1$ is used in the rule $lit\_l_1 \leftarrow light\_l_1 \wedge live\_l_1 \wedge ok\_l_1$: **why**.
Because that is what you asked me!

Typically, ***how*** and ***why*** are used together; ***how*** moves from higher-level to lower-level goals, and ***why*** moves from lower-level to higher-level goals. Together they let the user traverse a **proof tree**, where nodes are atoms, and a node together with its children corresponds to a clause in the knowledge base.

Example 5.13: As an example of the need to combine ***how*** and ***why***, consider the previous example where the user asked ***why*** up_s_1. The system gave the following rule:

$live\_w_1 \leftarrow live\_w_3 \wedge up\_s_1$.

This means that up_s_1 was asked because the system wants to know $live\_w_1$ and is using this rule to try to prove up_s_1. The user may think it is reasonable that the system wants to know $live\_w_1$ but may think it is inappropriate that up_s_1 be asked because the user may doubt that $live\_w_3$ should have succeeded. In this case it is useful for the user to ask ***how*** $live\_w_3$ was derived.

Knowledge-Level Debugging

Just as in other software, there can be errors and omissions in knowledge bases. Domain experts and knowledge engineers must be able to debug a knowledge base and add knowledge. In knowledge-based systems, debugging is

difficult because the domain experts and users who have the domain knowledge required to detect a bug do not necessarily know anything about the internal working of the system, nor do they want to. Standard debugging tools, such as providing traces of the execution, are useless because they require a knowledge of the mechanism by which the answer was produced. In this section, we show how the idea of semantics can be exploited to provide powerful debugging facilities for knowledge-based systems. Whoever is debugging the system is required only to know the meaning of the symbols and whether specific atoms are *true* or not. This is the kind of knowledge that a domain expert and a user may have.

Knowledge-level debugging is the act of finding errors in knowledge bases with reference only to what the symbols mean. One of the goals of building knowledge-based systems that are usable by a range of domain experts is that a discussion about the correctness of a knowledge base should be a discussion about the knowledge domain. For example, debugging a medical knowledge base should be a question of medicine that medical experts, who are not experts in AI, can answer. Similarly, debugging a knowledge base about house wiring should be with reference to the particular house, not about the internals of the system reasoning with the knowledge base.

Four types of non-syntactic errors can arise in rule-based systems:

- An incorrect answer is produced; that is, some atom that is *false* in the intended interpretation was derived.
- Some answer was not produced; that is, the proof failed when it should have succeeded (some particular *true* atom was not derived).
- The program gets into an infinite loop.
- The system asks irrelevant questions.

Ways to debug the first three types of error are examined below. Irrelevant questions can be investigated using the **why** questions as described earlier.

- 5.3.4.1 Incorrect Answers
- 5.3.4.2 Missing Answers
- 5.3.4.3 Infinite Loops

Incorrect Answers

An **incorrect answer** is an answer that has been proved yet is *false* in the intended interpretation. It is also called a **false-positive error**. An incorrect answer can only be produced by a sound proof procedure if an incorrect definite clause was used in the proof.

Assume that whoever is debugging the knowledge base, such as a domain expert or a user, knows the intended interpretation of the symbols of the language and can determine whether a particular proposition is *true* or *false* in the intended interpretation. The person does not have to know how the answer was derived. To debug an incorrect answer, a domain expert needs only to answer yes-or-no questions.

Suppose there is an atom g that was proved yet is *false* in the intended interpretation. Then there must be a rule $g \leftarrow a_1 \wedge \ldots \wedge a_k$ in the knowledge base that was used to prove g. Either

- one of the a_i is *false* in the intended interpretation, in which case it can be debugged in the same way, or
- all of the a_i are *true* in the intended interpretation. In this case, the definite clause $g \leftarrow a_1 \wedge \ldots \wedge a_k$ must be incorrect.

This leads to an algorithm, presented in Figure 5.6, to debug a knowledge base when an atom that is *false* in the intended interpretation is derived.

1: **Procedure** Debug(g,KB)
2: **Inputs**
3: KB a knowledge base
4: g an atom: $KB \vdash g$ and g is *false* in intended interpretation

Output clause in *KB* that is false
Find definite clause $g \leftarrow a_1 \wedge \ldots \wedge a_k \in KB$ used to prove g
for each a_i **do**
ask user whether a_i is true
if (user specifies a_i is *false*) **then**
return $Debug(a_i, KB)$

return $g \leftarrow a_1 \wedge \ldots \wedge a_k$

Figure 5.6: An algorithm to debug incorrect answers

This only requires the person debugging the knowledge base to be able to answer yes-or-no questions.

This procedure can also be carried out by the use of the ***how* command**. Given a proof for *g* that is false in the intended interpretation, a user can ask ***how*** that atom was proved. This will return the definite clause that was used in the proof. If the clause was a rule, the user could use ***how*** to ask about an atom in the body that was *false* in the intended interpretation. This will return the rule that was used to prove that atom. The user can repeat this until a definite clause is found where all of the elements of the body are *true* (or there are no elements in the body). This is the incorrect definite clause. The method of debugging assumes that the user can determine whether an atom is true or false in the intended interpretation. The user does not have to know the proof procedure used.

Example 5.14: Consider Example 5.5, involving the electrical domain, but assume there is a bug in the program. Suppose that the domain expert or user had inadvertently said that whether w_1 is connected to w_3 depends on the status of s_3 instead of s_1 (see Figure 1.8). Thus, the knowledge includes the following incorrect rule:

$live\_w_1 \leftarrow live\_w_3 \wedge up\_s_3.$

All of the other axioms are the same as in Example 5.5. Given this axiom set, the atom lit_l_1 can be derived, which is *false* in the intended interpretation. Consider how a user would go about finding this incorrect definite clause when they detected this incorrect answer.

Given that lit_l_1 is *false* in the intended interpretation, they ask how it was derived, which will give the following rule:

$lit\_l_1 \leftarrow light\_l_1 \wedge live\_l_1 \wedge ok\_l_1.$

They check the atoms in the body of this rule. $light\_l_1$ and ok_l_1 are *true* in the intended interpretation, but $live\_l_1$ is *false* in the intended interpretation. So they ask
how 2.

The system presents the rule
$live\_l_1 \leftarrow live\_w_0.$

$live\_w_0$ is *false* in the intended interpretation, so they ask
how 1.

The system presents the rule
$live\_w_0 \leftarrow live\_w_1 \wedge up\_s_2.$

$live\_w_1$ *is* false *in the intended interpretation, so they ask*

***how** 1.*

The system presents the rule
live_w_1 ←live_w_3 ∧up_s_3.

Both elements of the body are true in the intended interpretation, so this is the buggy rule.
The user or domain expert can find the buggy definite clause without having to know the internal workings of the system or how the proof was found. They only require knowledge about the intended interpretation and the disciplined use of ***how***.

Missing Answers
The second type of error occurs when an expected answer is not produced. This manifests itself by a failure when an answer is expected. A goal g that is true in the domain, but is not a consequence of the knowledge base, is called a **false-negative error**.
The preceding algorithm does not work in this case. There is no proof. We must look for why there is no proof for g.
An appropriate answer is not produced only if a definite clause or clauses are missing from the knowledge base. By knowing the intended interpretation of the symbols and by knowing what queries should succeed (i.e, what is true in the intended interpretation), a domain expert can debug a missing answer. Given an atom that failed when it should have succeeded, Figure 5.7 shows how to find an atom for which there is a missing definite clause.

1: **Procedure** DebugMissing(g,KB)
2: **Inputs**
3: KB a knowledge base
4: g an atom: $KB \nvdash g$ and g is *true* in the intended interpretation
5: **Output**
6: atom for which there is a clause missing
7: **if** (there is a definite clause $g \leftarrow a_1 \wedge \ldots \wedge a_k \in KB$ such that all a_i are true in the intended interpretation) **then**
8: **select** a_i that cannot be proved
9: *DebugMissing(a_i,KB)*
10: **else**
11: **return** g

Figure 5.7: An algorithm for debugging missing answers

Suppose g is an atom that should have a proof, but which fails. Because the proof for g fails, the bodies of all of the definite clauses with g in the head fail.

- Suppose one of these definite clauses for g should have resulted in a proof; this means all of the atoms in the body must be true in the intended interpretation. Because the body failed, there must be an atom in the body that fails. This atom is then true in the intended interpretation, but fails. So we can recursively debug it.
- Otherwise, there is no definite clause applicable to proving g, so the user must add a definite clause for g.

Example 5.15: Suppose that, for the axiomatization of the electrical domain in Example 5.5, the world of Figure 1.8 actually had s_2 down. Thus, it is missing the definite clause specifying that s_2 is down. The axiomatization of Example 5.5 fails to prove *lit_l_1* when it should succeed. Let's find the bug.

lit_l1 failed, so we find all of the rules with *lit_l1* in the head. There is one such rule:
lit_l1 ←*light_l1* ∧*live_l1* ∧*ok_l1*.

The user can then verify that all of the elements of the body are true. *light_l1* and *ok_l1* can both be derived, but *live_l1* fails, so we debug this atom. There is one rule with *live_l1* in the head:
live_l1 ←*live_w0*.

The atom *live_w0* cannot be proved, but the user verifies that it is true in the intended interpretation. So we find the rules for *live_w0*:
live_w0 ←*live_w1* ∧*up_s2*.
live_w_0 ←live_w_2 ∧down_s_2.

The user can say that the body of the second rule is true. A proof exists for *live_w2*, but there are no definite clauses for *down_s2*, so this atom is returned. The correction is to add the appropriate atomic clause or rule for it.

Infinite Loops

There is an infinite loop in the top-down derivation if there is an atom *a* that is being proved as a subgoal of *a*. (Here we assume that being a subgoal is transitive; a subgoal of a subgoal is a subgoal). Thus, there can be an infinite loop only if the knowledge base is cyclic. A knowledge base is **cyclic** if there is an atom *a* such that there is a sequence of definite clauses of the form

a ←...a_1 ...
a_1 ←...a_2 ...
...
a_n ←...*a* ...

(where if *n=0* there is a single definite clause with *a* in the head and body).

A knowledge base is **acyclic** if there is an assignment of natural numbers (non-negative integers) to the atoms so that the atoms in the body of a definite clause are assigned a lower number than the atom in the head. All of the knowledge bases given previously in this chapter are acyclic. There cannot be an infinite loop in an acyclic knowledge base.

To detect a cyclic knowledge base, the top-down proof procedure can be modified to maintain the set of all **ancestors** for each atom in the proof. In the procedure in Figure 5.4, the set *A* can contain pairs of an atom and its set of ancestors.

Initially the set of ancestors of each atom is empty. When the rule
$a \leftarrow a_1 \wedge \ldots \wedge a_k$

is used to prove *a*, the ancestors of a_i will be the ancestors of *a* together with *a*. That is,

$ancestors(a_i) = ancestors(a) \cup \{a\}$.

The proof can fail if an atom is in its set of ancestors. This failure only occurs if the knowledge base is cyclic. Note that this is a more refined version of cycle checking, where each atom has its own set of ancestors.

A cyclic knowledge base is often a sign of a bug. When writing a knowledge base, it is often useful to ensure an acyclic knowledge base by identifying a value that is being reduced at each iteration. For example, in the electrical domain, the number of steps away from the outside of the house is meant to be reduced by one each time through the loop. Disciplined and explicit use of a well-founded ordering can be used to prevent infinite loops in the same way that programs in traditional languages must be carefully programmed to prevent infinite looping.

Note that the bottom-up proof procedure does not get into an infinite loop, because it selects a rule only when the head has not been derived.

Other Knowledge Representation Schemes:

Over the past 40 years, numerous representational scheme have been proposed and implemented, each of them having its own strength and weakness.

According to Mylopoulos and Levesque (1984) they have been classified into four categories:

1. Logical Representation Scheme:

This class of representation uses expressions in formal logic to represent a knowledge base. Inference rules and proof procedures apply this knowledge to problem solving. First order predicate calculus is the most widely used logical representation scheme, and PROLOG is an ideal programming language for implementing logical representation schemes.

2. Procedural Representation Scheme:

Procedural scheme represents knowledge as a set of instructions for solving a problem. In a rule-based system, for example, an if then rule may be interpreted as a procedure for searching a goal in a problem domain: to arrive at the conclusion, solve the premises in order. Production systems are examples of a procedural representation scheme.

3. Network Representation Scheme:

Network representation captures knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs represent relations or associations between them. Examples of network representations include semantic network, conceptual dependencies and conceptual graphs.

4. Structured Representation Scheme:

Structured representation languages extend networks by allowing each node to be a complex data structure consisting of named slots with attached values. These values may be simple numeric or complex data, such as pointers to other frames, or even procedures.

Non-monotonic Reasoning

The definite clause logic is **monotonic** in the sense that anything that could be concluded before a clause is added can still be concluded after it is added; adding knowledge does not reduce the set of propositions that can be derived.

A logic is **non-monotonic** if some conclusions can be invalidated by adding more knowledge. The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. A **default** is a rule that can be used unless it overridden by an exception.

For example, to say that *b* is normally true if *c* is true, a knowledge base designer can write a rule of the form $b \leftarrow c \wedge \sim ab_a$.

where ab_a is an atom that means abnormal with respect to some aspect a. Given c, the agent can infer b unless it is told ab_a. Adding ab_a to the knowledge base can prevent the conclusion of b. Rules that imply ab_a can be used to prevent the default under the conditions of the body of the rule.

Example 5.27: Suppose the purchasing agent is investigating purchasing holidays. A resort may be adjacent to a beach or away from a beach. This is not symmetric; if the resort was adjacent to a beach, the knowledge provider would specify this. Thus, it is reasonable to have the clause

$away\_from\_beach \leftarrow \sim on\_beach.$

This clause enables an agent to infer that a resort is away from the beach if the agent is not told it is adjacent to a beach.

A **cooperative system** tries to not mislead. If we are told the resort is on the beach, we would expect that resort users would have access to the beach. If they have access to a beach, we would expect them to be able to swim at the beach. Thus, we would expect the following defaults:

$beach\_access \leftarrow on\_beach \wedge \sim ab_{beach\_access}.$

$swim\_at\_beach \leftarrow beach\_access \wedge \sim ab_{swim\_at\_beach}.$

A cooperative system would tell us if a resort on the beach has no beach access or if there is no swimming. We could also specify that, if there is an enclosed bay and a big city, then there is no swimming, by default:

$ab_{swim\_at\_beach} \leftarrow enclosed\_bay \wedge big\_city \wedge \sim ab_{no\_swimming\_near\_city}.$

We could say that British Columbia is abnormal with respect to swimming near cities:

$ab_{no\_swimming\_near\_city} \leftarrow in\_BC \wedge \sim ab_{BC\_beaches}.$

Given only the preceding rules, an agent infers *away_from_beach*. If it is then told *on_beach*, it can no longer infer *away_from_beach*, but it can now infer *beach_access* and *swim_at_beach*. If it is also told *enclosed_bay* and *big_city*, it can no longer infer *swim_at_beach*. However, if it is then told *in_BC*, it can then infer *swim_at_beach*.

By having defaults of what is normal, a user can interact with the system by telling it what is abnormal, which allows for economy in communication. The user does not have to state the obvious.

One way to think about non-monotonic reasoning is in terms of **arguments**. The rules can be used as components of arguments, in which the negated abnormality gives a way to undermine arguments. Note that, in the language presented, only positive arguments exist that can be undermined. In more general theories, there can be positive and negative arguments that attack each other.

1. ***Acting Under Uncertainty***

When an agent knows enough facts about its environment, the logical approach enables it to derive plans that are guaranteed to work. But unfortunately, *agents never have access to the whole truth about their environment.* This is called ***uncertainty*.**

- For example, an agent in the wumpus world consists of sensors that report only local information; most of the world is not immediately observable. A wumpus agent often will find itself unable to discover which of two squares contains a pit. If those squares are *en route* to the gold, then the agent might have to take a chance and enter one of the two squares.

- The real world is far more complex than the wumpus world. For a logical agent, it might be impossible to construct a complete and correct description of how its actions will work.
- Suppose, for example, that the agent wants to drive someone to the airport to catch a flight and is considering a plan, A90, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only about 15 miles away, the agent will not be able to conclude with certainty that "Plan Ago will get us to the airport in time." Instead, it reaches the weaker conclusion "Plan Ago will get us to the airport in time, as long as my car doesn't break down or run out of gas, and I don't get into an accident, and there are no accidents on the bridge, and the plane doesn't leave early and . . . ." None of these conditions can be deduced, so the plan's success cannot be inferred.
- The information that the agent has cannot guarantee any of these outcomes for A90, but it can provide some degree of belief that they will be achieved.
- Other plans, such as A120, might increase the agent's belief that it will get to the airport on time, but also increase the likelihood of a long wait. "***The right thing to do-the rational decision—therefore depends on both the relative importance of various goals and the likelihood that,*** *and* ***degree to which, they will be achieved".***

1.1 Handling uncertain knowledge

Here we consider the nature of uncertain knowledge; Let us see a simple-diagnosis example to illustrate the concepts involved. "Diagnosis for medicine". So write rules for dental diagnosis using first-order logic.

$$\forall p \;\; Symptom(p, Toothache) \Rightarrow Disease(p, Cavity)$$

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

$$\forall p \;\; Symptom(p, Toothache) \Rightarrow$$
$$Disease(p, Cavity) \vee Disease(p, GumDisease) \vee Disease(p, Abscess) \ldots$$

In order to make the rule true, we have to add an almost unlimited list of possible causes. We could try turning the rule into a causal rule:

$$\forall p \;\; Disease(p, Cavity) \Rightarrow Symptom(p, Toothache)$$

But this rule is not right either; not all cavities cause pain. The only way to fix the rule is to make it logically exhaustive: i.e., the left-hand side with all the qualifications required for a cavity to cause a toothache. 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 exception less rule and too hard to use such rules.

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

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

- The connection between toothaches and cavities is just not a logical consequence in either direction. This is typical of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, and so on.
- The agent's knowledge *DEGREE OF BELIEF* can at best provide only a ***degree of belief*** in the relevant sentences. Our main tool for dealing *PROBABILITY THEORY* with degrees of belief will be ***probability theory*****;** it assigns to each sentence a numerical degree of belief between 0 and 1.

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

- This belief could be derived from statistical data-80% of the toothache patients seen so far have had cavities. The 80% summarizes those cases in which all the factors needed for a cavity to cause a toothache are present and other cases in which the patient has both toothache and cavity but the two are unconnected.
- The missing 20% summarizes all the other possible causes of toothache that we are too lazy or ignorant to confirm or deny.
- The sentence is false, while assigning a probability of 1 corresponds to an unequivocal belief that the sentence is true. Probabilities between 0 and 1 correspond to intermediate degrees of belief in the truth of the sentence.
- Thus, probability theory makes the same ontological commitment as logic namely; the facts either do or do not hold in the world. Degree of truth, as opposed to degree of belief, is the subject of **fuzzy logic.**

Evidence: In logic, a sentence such as "The patient has a cavity" is true or false depending on the interpretation and the world; it is true just when the fact it refers to is the case. In probability theory, a sentence such as "The probability that the patient has a cavity is 0.8" is about the agent's beliefs, not directly about the world. These beliefs depend on the percepts that the agent has received to date. These percepts constitute the **evidence** on which probability assertions are based.

- Before the evidence is obtained, we talk about **prior** or **unconditional** probability; after the evidence is obtained, we talk about **posterior** or **conditional** probability.

1.2 Uncertainty and Rational decisions:

To make choices among alternatives, an agent must first have ***preferences*** between the different possible ***outcomes*** of the various plans. A particular outcome is a completely specified state, including such factors as whether the agent arrives on time and the length of the wait at the airport. We will be using ***utility theory*** to represent and reason with preferences. Utility theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states .with higher utility.

The utility of a state is relative to the agent whose preferences the utility function is supposed to represent. For example, the utility of a state in which White has won a game of chess is obviously high for the agent playing White, but low for the agent p1aying Black.

So, Preferences, as expressed by utilities, are combined with probabilities in the general theory of ra.tiona1 decisions called ***decision theory*:**

Decision theory = probability theory + utility theory

The fundamental idea of decision theory is that *an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action.* This is called the principle of **Maximum Expected Utility (MEU).**

1.3 Design for a decision-theoretic agent:

Figure 1.3.1 sketches the structure of an agent that uses decision theory to select actions. The agent is identica1, at an abstract level, to the logical agent. The primary difference is that the decision-theoretic agent's knowledge of the current state is uncertain; the agent's **belief state** is a representation of the probabilities of all possible actual states of the world.

As time passes, the agent accumulates more evidence and its belief state changes.

Given the belief state, the agent can make probabilistic predictions of action outcomes and hence select the action with highest expected utility.

```
function DT-AGENT(percept) returns an action
  static: belief-state, probabilistic beliefs about the current state of the world
          action, the agent's action

  update belief-state based on action and percept
  calculate outcome probabilities for actions,
      given action descriptions and current belief-state
  select action with highest expected utility
      given probabilities of outcomes and utility information
  return action
```

Figure 1.3.1 A decision-theoretic agent that selects rational actions.

2. ***Basic Probability Notation***

Any notation for describing degrees of belief must be able to deal with two main issues:

1) The nature of the sentences to which degrees of belief are assigned
2) The dependence of the degree of belief on the agent's experience.

Propositions:

Probability theory typically uses a language that is slightly *more expressive than propositional logic.* The basic element of the language is the *random variable*, which can be thought of as referring to a "part" of the world whose "status" is initially unknown.

- For example, *Cavity* might refer to whether my lower left wisdom tooth has a cavity. Random variables play a role similar to that of CSP variables in constraint satisfaction problems and that of proposition symbols in propositional logic. We will always capitalize the names of random variables. For example: P(a) = 1 - P(¬a)).
- Each random variable has a *domain* of values that it can take on. For example, the domain of *Cavity* might be *(true, fail)*.
- For example, *Cavity = true* might represent the proposition that I do in fact have a cavity in my lower left wisdom tooth.

As with CSP variables, random variables are typically divided into *three kinds*, depending on the type of the domain:

- Boolean random variables, such as *Cavity,* have the domain *(true, false).* We will often abbreviate a proposition such as *Cavity = true* simply by the lowercase name *cavity.* Similarly, *Cavity = false* would be abbreviated by 1 *cavity.*

- Discrete random variables, which include Boolean random variables as a special case, take on values from a *countable* domain. For example, the domain of *Weather* might be *(sunny, rainy, cloudy, snow).* The values in the domain must be mutually exclusive and exhaustive. Where no confusion arises, we: will use, for example, *snow* as an abbreviation for *Weather = snow.*
- Continuous random variables take on values from the: real numbers. The domain can be either the entire real line or some subset such as the interval [0, 1]. For example, the proposition X = 4.02 asserts that the random variable .*X* has the exact value 4.02.

Elementary propositions, such as *Cavity = true* and *Toothache =false,* can be combined to form complex propositions using all the standard logical connectives. For example, *Cavity = true A Toothache =false* is a proposition to which one may ascribe a degree of belief. As explained in the previous paragraph, this proposition may also be written as *cavity ∧ toothache.*

Atomic events:

The notion of an **atomic event** is useful in understanding the foundations of probability theory.

An atomic event is a complete specification of the state of the world about which the agent is uncertain. It can be thought of as an assignment of particular values to all the variables of which the world is composed. For example, if my world consists of only the Boolean variables *Cavity* and *Toothache,* then there are just four distinct atomic events; the proposition

Cavity =false ∧ Toothache = true is one such event.

Atomic events have some important properties:

- They are mutually exclusive-at most one can actually be the case. For example, *cavity* A *toothache* and *cavity ∧ -toothache* cannot both be the case.
- The set of all possible atomic events is exhaustive-at least one must be the case. That is, the disjunction of all atomic events is logically equivalent to *true.*
- Any particular atomic event entails the truth or falsehood of every proposition, whether simple or complex. This can be seen by using the standard semantics for logical connectives. For example, the atomic event *cavity ∧ ⌐ toothache* entails the truth of *cavity* and the falsehood of *cavity => toothache.*
- Any proposition is logically equivalent to the disjunction of all atomic events that entail the truth of the proposition. For example, the proposition *cavity* is equivalent to disjunction of the atomic events *cavity ∧ toothache* and *cavity ∧ ⌐toothache.*

Prior Probability:

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

***P (Cavity = true) = 0.1* or *P (cavity)* = 0.1**

It is important to remember that *P (a)* can be used only when there is no other information. As soon as some new information is known, we must reason with the conditional probability of *a* given that new information. Now if we talk about the probabilities of all the possible values of a random variable. In that case, we will use an expression such as **P (***Weather),* which denotes a ***vector*** of values for the probabilities of each individual state of the weather. So, Instead of writing the four equations

P (Weather = sunny) = 0.7

P (Weather = rain) = 0.2

P (Weather = cloudy) = 0.08

P (Weather = snow) = 0.02.

We may simply write

P (Weather) = (0.7, 0.2, 0.08, 0.02).

This statement defines a *prior probability distribution* for the random variable *Weather*

We will also use expressions such as *P(Weather, Cavity)* to denote the probabilities of all combinations of the values of a set of random variable^ In that case, *P(Weather, Cavity)* can be represented by a 4 x *2* table of probabilities. This is called the *joint probability distribution* of *Weather* and *Cavity.*

- A joint probability distribution that covers this complete set is called the *full joint probability distribution***.** For example, if the world consists of just the variables *Cavity, Toothache,* and *Weather,* then the full joint distribution is given by

  P (Cavity, *Toothache, Weather)*

This joint distribution can be represented as a *2* x *2* x *4* table with 16 entries. So, any probabilistic query can be answered from the full joint distribution. But, *for continuous variables*, it is not possible to write out the entire distribution as a table, because there are infinitely many values. Instead, one usually defines the probability that a random variable takes on some value *x* as a parameterized function of *x.* For example, let the random variable X denote tomorrow's maximum temperature in Berkeley. Then the sentence

P(X = x) = U [18, 26] (x)

X is distributed uniformly between 18 and 26 degrees Celsius. Probability distributions for continuous variables are called *probability density functions*. Density functions differ in meaning from discrete distributions. For example, using the temperature distribution given earlier, we find that

$$P(X = 20.5) = U[18, 26](2\ 0.5) == 0.125/C.$$

The technical meaning is that the probability that the temperature is in a small region around 20.5 degrees is equal, in the limit, to 0.125 divided by the width of the region in degrees Celsius:

$$\lim_{dx \to 0} P(20.5 \leq X \leq 20.5 + dx)/dx = 0.125/C.$$

Conditional probability:

Once the agent has obtained some evidence concerning the previously unknown random variables making up the domain, prior probabilities are no longer applicable. Instead, we use **conditional** or **posterior** probabilities. The notation used is *P (a / b),* where *a* and *b* are any proposition. This is read as "the probability of *a,* given that *all* we know is *b."*

For example,

P (cavity / toothache) = 0.8

Indicates that if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8. A prior probability, such as *P (cavity),* can be thought of as a special case of the conditional probability *P (cavity /)*, where the probability is conditioned on no evidence. Conditional probabilities can be defined in terms of unconditional probabilities. The defining equation is which holds whenever *P (b)* > 0. This equation can also be written as

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

Which holds whenever *P (b)* > 0. This equation can also be written as

$$\boldsymbol{P(a \wedge b) = P(a / b)\ P(b)}$$

Which is called the ***product rule.*** It comes from the fact that, for *a* and *b* to be true, we need *b* to be true, and we also need *a* to be true given *b.* We can also have it the other way;

$$\boldsymbol{P(a \wedge b) = P(b / a)\ P(a)}$$

We can also use the *P* notation for conditional distributions. *P(X / Y)* gives the values of *P(X = xi / Y = yj)* for each possible *i, j.* As an example consider applying the product rule to each case where the propositions *a* and *b* assert particular values of *X* and *Y* respectively. We obtain the following equations:

$$P(X = x_1 \wedge Y = y_1) = P(X = x_1 | Y = y_1) P(Y = y_1)$$
$$P(X = x_1 \wedge Y = y_2) = P(X = x_1 | Y = y_2) P(Y = y_2)$$

We can combine all these into the single equation

P(X, Y) = P(X / Y) P(Y)

It is wrong, because to view conditional probabilities as if they were logical implications with uncertainty added. For example, the sentence ***P (a / b) = 0.8*** *cannot* be interpreted to mean "whenever *b* holds, conclude that *P (a)* is 0.8." Such an interpretation would be wrong on two counts:

- first, *P(a)* always denotes the prior probability of *a,* not the posterior probability given some evidence;
- Second, the statement ***P (a / b) = 0.8*** is immediately relevant just when *b* is the *only* available evidence. When additional information *c* is available, the degree of belief in *a* is *P (a / b ^ c),* which might have little relation to *P (a / b).*
- For example, *c* might tell us directly whether *a* is true or false. If we examine a patient who complains of toothache, and discover a cavity, then we have additional evidence *cavity,* and we conclude (trivially) that ***P (cavity / toothache ^ cavity) = 1.0.***

3. ***The Axioms Of Probability***

So far, we have defined a syntax for propositions and for prior and conditional probability statements about those propositions. Now we must provide some sort of semantics for probability statements. We begin with the basic axioms that serve to define the probability scale and its endpoints:

1. All probabilities are between 0 and 1. For any proposition a,

   $0 \leq P(a) \leq 1$

2. Necessarily true (i.e., valid) propositions have probability I, and necessarily false (i.e., unsatisfiable) propositions have probability 0.

$P(true) = 1$ $\qquad$ $P(false) = 0$.

Next, we need an axiom that connects the probabilities of logically related propositions. The simplest way to do this is to define the probability of a disjunction as follows:

3. The probability of a disjunction is given by

   $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$.

This rule is easily remembered by noting that the cases where *a* holds, together with the cases where *b* holds, certainly cover all the cases where *a* V *b* holds; but summing the two sets of cases counts their intersection twice, so we need to subtract *Y (a* ∧ b).

These three axioms are often called **Kolmogorov's axioms.**

Using the axioms of probability:

We can derive a variety of useful facts from the basic, axioms. For example, the familiar rule for negation follows by substituting *~a for* b in axiom 3, giving us:

$$\begin{aligned} P(a \vee \neg a) &= P(a) + P(\neg a) - P(a \wedge \neg a) && \text{(by axiom 3 with b} = \neg a) \\ P(true) &= P(a) + P(\neg a) - P(false) && \text{(by logical equivalence)} \\ 1 &= P(a) + P(\neg a) && \text{(by axiom 2)} \\ P(\neg a) &= 1 - P(a) && \text{(by algebra).} \end{aligned}$$

The third line of this derivation is itself a useful fact and can be extended from the Boolean case to the general discrete case. Let the discrete variable D have the domain (dl, . . . , d,). Then it is easy to show that

$$\sum_{i=1}^{n} P(D = d_i) = 1 .$$

The probability of a proposition is equal to the sum of the probabilities of the atomic events in which it holds; that is,

$$P(a) = \sum_{e_i \in \mathbf{e}(a)} P(e_i) .$$

Why the axioms of probability are reasonable:

The axioms of probability can be seen as restricting the set of probabilistic beliefs that an agent can hold. Where a logical agent cannot simultaneously believe A, *B,* and *~ (A ∧ B)* for example. In the logical case, the semantic definition of conjunction means that at least one of the three beliefs just mentioned *must be false in the world,* so it is unreasonable for an agent to believe all three. With probabilities, on the other hand, statements refer not to the world directly, but to the agent's own state of knowledge. Why, then, can an agent not hold the following set of beliefs, which clearly violates axiom 3?

$$P(a) = 0.4 \qquad P(a \wedge b) = 0.0$$
$$P(b) = 0.3 \qquad P(a \vee b) = 0.8$$

Finetti proved something much stronger: I*f Agent* I *expresses a set of degrees of belief that violate the axioms of probability theory then there is a combination of bets by Agent 2 that* guarantees *that Agent I will lose money* every *time.*

We will not provide the proof of de Finetti's theorem, but we will show an example. Suppose that Agent 1 has the set of degrees of belief from Equation given above. Figure 3.1 shows that if Agent 2 chooses to bet $4 on a, $3 on *b,* and $2 on ~ *(a V b),* then Agent 1 always loses money, regardless of the outcomes for *a* and *b.*

Agent 1		Agent 2		Outcome for Agent 1			
Proposition	Belief	Bet	Stakes	*a* A *b*	*a* A ¬*b*	¬*a* A *b*	¬*a* ∧ ¬*b*
a	0.4	*a*	4 to 6	–6	–6	4	4
b	0.3	b	3 to 7	–7	3	–7	3
a ∨ *b*	*0.8*	¬(*a* ∨ *b*)	2 to 8	2	2	2	–8
				–11	–1	–1	–1

4. *Inference Using Full Joint Distributions*

Here we will use the full joint distribution as the "knowledge base" from which answers to all questions may be derived. Along the way we will also introduce several useful techniques for manipulating equations involving probabilities. We begin with a very simple example: a domain consisting of just the three Boolean variables *Toothache, Cavity,* and *Catch.* The full joint distribution is a 2 x 2 x 2 table as shown In Figure 4.1.

	toothache		¬*toothache*	
	catch	¬*catch*	*catch*	¬*catch*
cavity	0.108	0.012	0.072	0.008
¬*cavity*	0.016	0.064	0.144	0.576

Figure 4.1 *A full joint distribution for the Toothache, Cavity, and Catch world.*

Now identify those atomic events in which the proposition is true and add up their probabilities. For example, there are six atomic events in which *cavity V toothache* holds:

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$$

One common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the unconditional or marginal probability of *cavity:*

$$P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2$$

This process is called marginalization, or summing out-because the variables other than *Cavity* are summed out. We can write the following general marginalization rule for any sets of variables Y and Z:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{z})$$

That is, a distribution over Y can be obtained by summing out all the other variables from any joint distribution containing Y. A variant of this rule involves conditional probabilities instead of joint probabilities, using the product rule:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}|\mathbf{z})P(\mathbf{z})$$

This rule is called **conditioning.** Marginalization and conditioning will turn out to be useful rules for all kinds of derivations involving probability expressions.

For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

$$\begin{aligned} P(cavity|toothache) &= \frac{P(cavity\ A\ toothache)}{P(toothache)} \\ &= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6 \end{aligned}$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$\begin{aligned} P(\neg cavity|toothache) &= \frac{P(\neg cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 \end{aligned}$$

In these two calculations the term *1/P (toothache)* remains constant, no matter which value of *Cavity* we calculate. In fact, it can be viewed as a ***normalization*** constant for the distribution *P(Cavity / toothache),* ensuring that it adds up to 1.

We will use *a* to denote such constants. With this notation, we can write the two preceding equations in one:

$$\begin{aligned} \mathbf{P}(Cavity(toothache) &= a\mathbf{P}(Cavity, toothache) \\ &= a[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)] \\ &= \alpha[\langle 0.108, 0.016\rangle + \langle 0.012, 0.064\rangle] = \alpha\langle 0.12, 0.08\rangle = \langle 0.6, 0.4\rangle . \end{aligned}$$

Algorithm for probabilistic inference:

```
function ENUMERATE-JOINT-ASK(X, e, P) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          P, a joint distribution on variables {X} ∪ E ∪ Y   /* Y = hidden variables */

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
      Q(x_i) ← ENUMERATE-JOINT(x_i, e, Y, [], P)
  return NORMALIZE(Q(X))

function ENUMERATE-JOINT(x, e, vars, values, P) returns a real number
  if EMPTY?(vars) then return P(x, e, values)
  Y ← FIRST(vars)
  return Σ_y ENUMERATE-JOINT(x, e, REST(vars), [y|values], P)
```

Figure 4.2 An algorithm for probabilistic inference by enumeration of the entries in a full joint: distribution.

Given the full joint distribution to work with, ENUMERATE-JOINT-ASK is a complete algorithm for answering probabilistic queries for discrete variables. It does not scale well, however: For a domain described by n Boolean variables, it requires an input table of size 0(*2 pow n*) and takes

0(2 pow n) time to process the table. So, the full joint distribution in tabular form is not a practical tool for building reasoning systems.

5. ***Independence***

Let us expand the full joint distribution in Figure 13.3 by adding a fourth variable, *Weather* .The full joint distribution then becomes *P(Toothache, Catch, Cavity, Weather),* which has *32* entries (because *Weather* has four values). It contains four "editions" of the table shown in Figure 4.1, one for each kind of weather. Here we may ask what relationship these editions have to each other and to the original three-variable table. For example, how are *P(toothache, catch, cavity, Weather = cloudy)* and *P(toothache, catch, cavity)* related?

To answer this question is to use the product rule: *P(toothache, catch, cavity, Weather = cloudy)*

= P (Weather = cloudy / toothache, catch, cavity) P (toothache, catch, cavity).

One should not imagine that one's dental problems influence the weather. Therefore, the following assertion seems reasonable:

P(Weather = cloudy / toothache, catch, cavity) = P (Weather = cloudy)---------(1)

From this, we can deduce;

P(toothache, catch, cavity, weather = cloudy) = P(Weather = cloudy)P(toothache, catch, cavity).

Similar equation exists for *every entry* in *P(Toothache, Catch, Cavity, Weather).* In fact, we can write the general equation;

P(Toothache, Catch, Cavity, Weather) = P(Toothache, Catch, Cavity)P(Weather) .

Thus, the 32-element table for four variables can be constructed from one 8-element table and one four-element table. This decomposition is illustrated schematically in Figure 5.1(a). The property we used in writing Equation (1) is called "***independence".***

Independence between propositions *a* and *b* can be written as

P (a / b)= P (a) or *P(b / a)= P (b)* or *P (a A* b) = *P (a) P (b).*

Independence between variables X and Y can be written as follows (again, these are all equivalent):

P (X / Y)= P (X) or P (Y / X)= P(Y) or P (X ,Y)= P (X) P (Y).

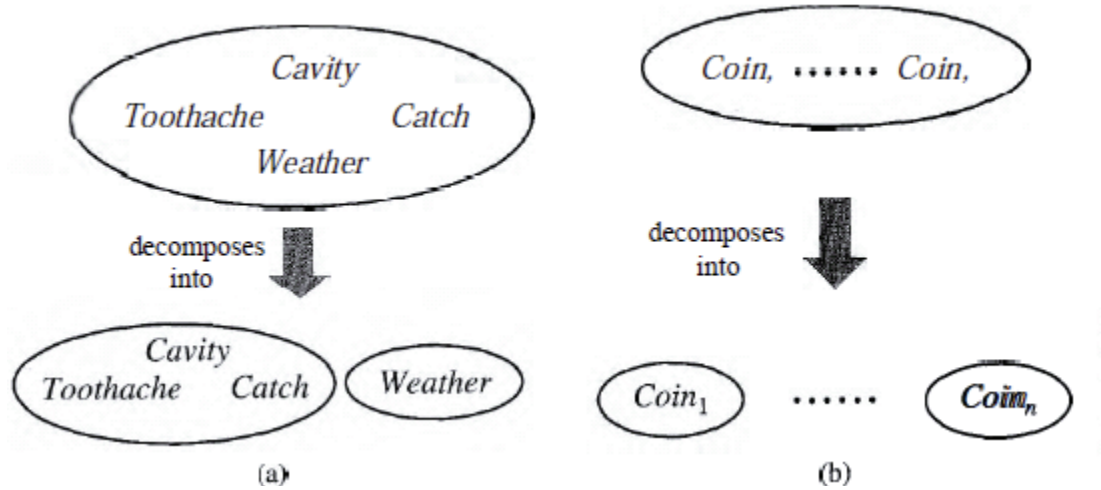


Figure 5.1 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

6. ***Baye's rule and its use***

We defined the **product rule** and pointed out that it can be written in two forms because of the commutativity of conjunction:

$$P(a \wedge b) = P(a|b)P(b)$$
$$P(a \wedge b) = P(b|a)P(a)$$

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

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)}$$

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

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

Where again this is to be taken as representing a set of equations, each dealing with specific values of the variables. We will also have occasion to use a more general version conditionalized on some background evidence **e:**

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

Applying Bayes' rule: The simple case:

It requires three terms-a conditional probability and two unconditional probabilities-just to compute one conditional probability. Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships and want to derive a diagnosis. A doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 50% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1150,000, and the prior probability that any patient has a stiff neck is 1120. Letting *s* be the proposition that the patient has a stiff neck and *m* be the proposition that the patient has meningitis, we have;

$$\begin{aligned} P(s|m) &= 0.5 \\ P(m) &= 1/50000 \\ P(s) &= 1/20 \\ P(m|s) &= \frac{P(s|m)P(m)}{P(s)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002 . \end{aligned}$$

That is, we expect only 1 in 5000 patients with a stiff neck to have meningitis. Notice that, even though a stiff neck is quite strongly indicated by meningitis (with probability 0.5), the probability of meningitis in the patient remains small. This is because the prior probability on stiff necks is much higher than that on meningitis.

The same process can be applied when using Bayes' rule. We have

$$\mathbf{P}(M|s) = a\langle P(s|m)P(m), P(s|\neg m)P(\neg m)\rangle$$

Thus, in order to use this approach we need to estimate *P(s /~ m)* instead of *P(s)*

The general form of Bayes' rule with normalization is

P(Y / X) = a P(X /Y) P(Y), where *a* is the normalization constant needed to make the entries in

P(Y / X) sum to 1.

Using Bayes' rule: Combining evidence:

We have seen that Bayes' rule can be useful for answering probabilistic queries conditioned on one piece of evidence-for example, the stiff neck. In particular, we have argued that probabilistic information is often available in the form *P(effect / cause).* What happens when we have two or more pieces of evidence? For example, what can a dentist conclude if her nasty steel probe catches in the aching tooth of a patient? If we know the full joint distribution, one can read off the answer:

$$\mathbf{P}(Cavity\,(toothacheA\ catch) = {}_{a!}\langle 0.108, 0.016\rangle \approx \langle 0.871, 0.129\rangle$$

We know, however, that such an approach will not scale up to larger numbers of variables. We can try using Bayes' rule to reformulate the problem:

$$\mathbf{P}(Cavity|toothache\ A\ catch) = \alpha\mathbf{P}(toothache \wedge catch|Cavity)\mathbf{P}(Cavity)$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction *toothache* A *catch* for each value of *Cavity.* That might be feasible for just two evidence variables, but again it will not scale up. If there are n possible evidence variables (X rays, diet, oral hygiene, etc.), then there are *2n* possible combination so f observed values for which we would need to know conditional probabilities. We might as well go back to using the full joint distribution.

Rather than taking this route, we need to find some additional assertions about the domain that will enable us to simplify the expressions. The notion of **independence** provides a clue, but needs refining. It would be nice if *Toothache* and *Catch* were independent, but 'they are not: if the probe catches in the tooth, it probably has a cavity and that probably causes a toothache. These variables *are* independent.

Mathematically, this property is written as;

$$\mathbf{P}(toothache\ A\ catch|Cavity) = \mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)$$

This equation expresses the **conditional independence** of *toothache* and *catch* given *Cavity.* We can plug it into above equation to obtain the probability of a cavity:

$$\mathbf{P}(Cavity|toothache\ A\ catch) = {}_{a!}\mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)\mathbf{P}(Cavity).$$

The general definition of conditional independence of two variables X and *Y,* given a third variable *Z* is

$$\mathbf{P}(X, Y|Z) = \mathbf{P}(X|Z)\mathbf{P}(Y|Z)$$

In the dentist domain, for example, it seems reasonable to assert conditional independence of the variables *Toothache* and *Catch,* given *Cavity:*

$$\mathbf{P}(Toothache, Catch|Cavity) = \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity).$$

Which asserts independence only for specific values of *Toothache* and *Catch?* As with absolute independence in Equation

$$\mathbf{P}(X|Y,Z) = \mathbf{P}(X|Z) \quad \text{and} \quad \mathbf{P}(Y|X,Z) = \mathbf{P}(Y|Z)$$

It turns out that the same is true for conditional independence assertions. For example, given the assertion in Equation, We can derive decomposition as follows:

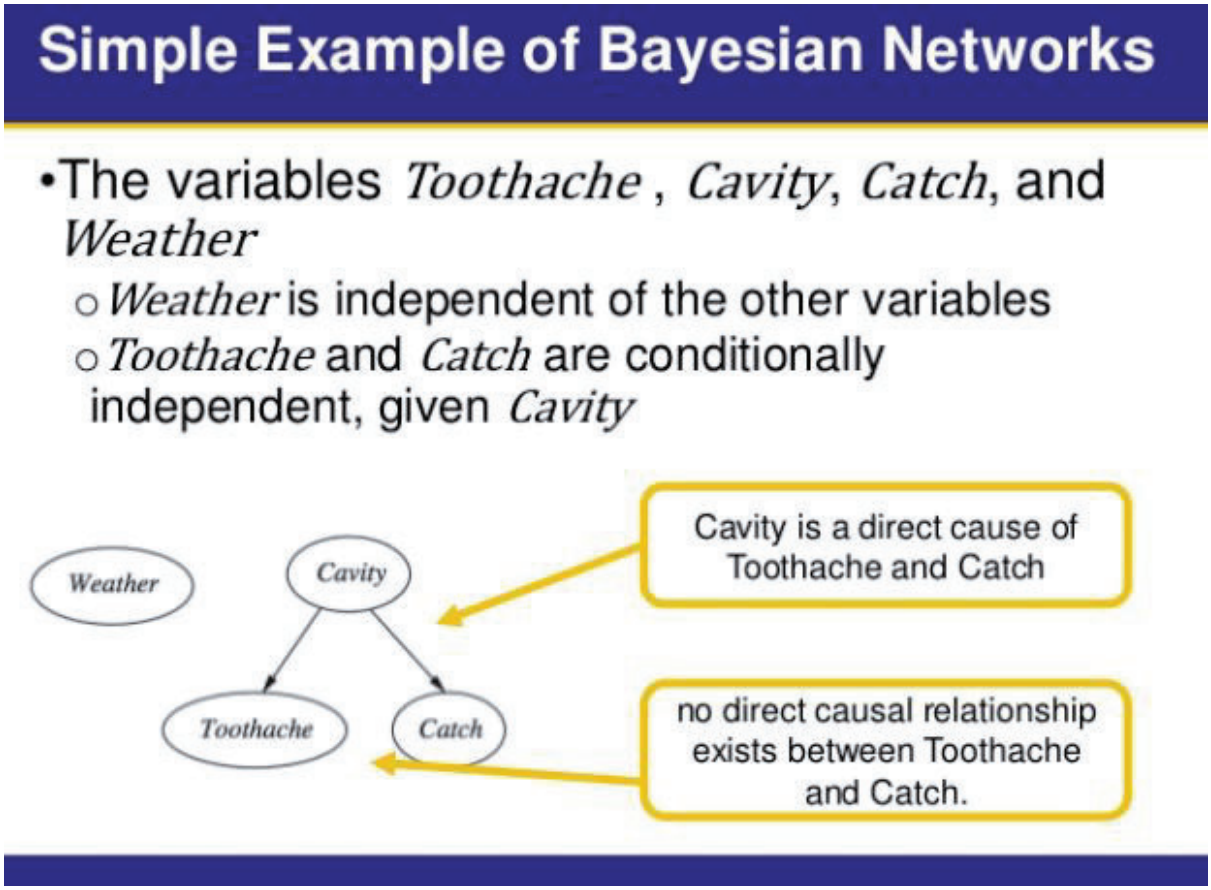
$$\mathbf{P}(Toothache, Catch, Cavity)$$
$$= \mathbf{P}(Toothache, Catch|Cavity)\mathbf{P}(Cavity) \quad \text{(product rule)}$$
$$= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)$$

In this way, the original large table is decomposed into three smaller tables.

Representing Knowledge in an Uncertain Domain •Bayesian Networks

A directed graph in which each node is annotated with quantitative probability information Definition

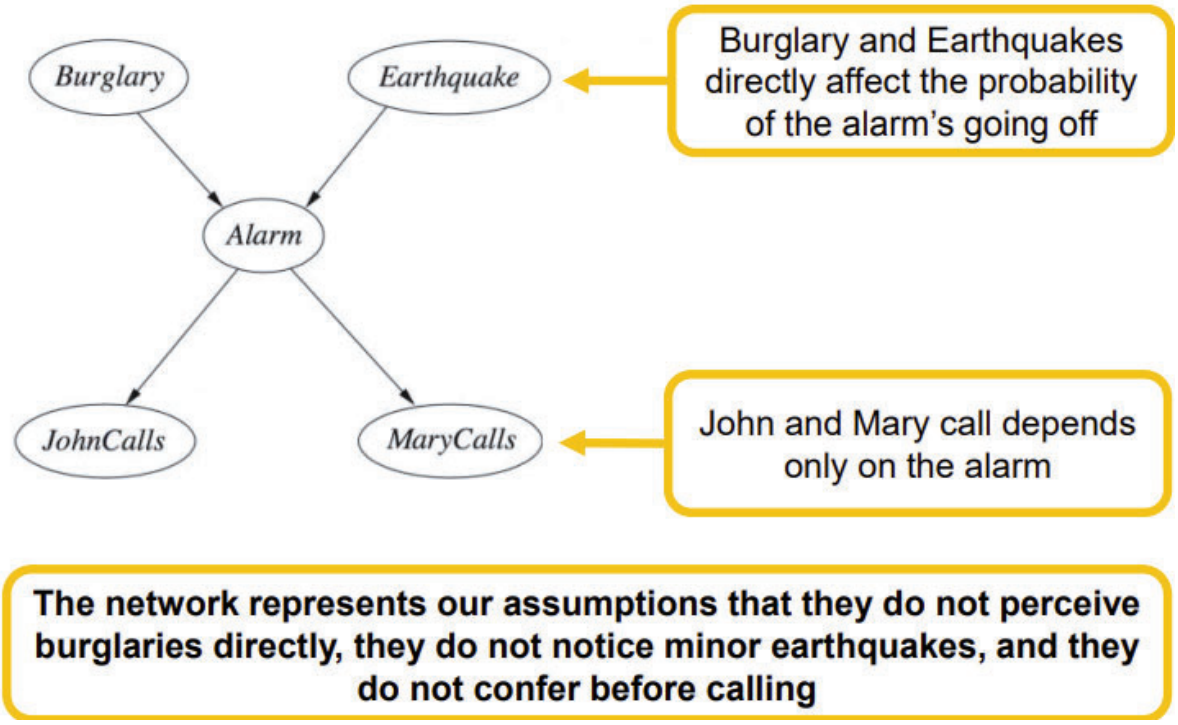
♣1. Each node corresponds to a random variable, which may be discrete or continuous ♣2. A set of directed links or arrows connects pairs of nodes. (If there is an arrow from node X to node Y , X is said to be a parent of Y.) ♣3. The graph has no directed cycle. ♣4. Each node Xi has a conditional probability distribution P(Xi|Parents(Xi)) that quantifies the effect of the parents on the node.



Complex Example of Bayesian Networks

The variables Burglary, Earthquake, Alarm, MaryCalls and JohnCalls oNew burglar alarm installed at home oFairly reliable at detecting a burglary oResponds on occasion to minor earthquakes oTwo neighbors, John and Mary oThey call you at work when they hear the alarm oJohn nearly always calls when he hears the alarm oBut sometimes confuses the telephone ringing oMary likes rather loud music and misses the alarm

Complex Example of Bayesian Networks(2/4)

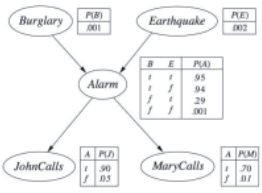


The Semantics of Bayesian Networks

The two ways to understand the meaning of Bayesian Networks oTo see the network as a representation of the joint probability distribution ♣To be helpful in understanding how to construct networks, oTo view it as an encoding of a collection of conditional independence statements ♣To be helpful in designing inference procedures

Representing the full joint distribution

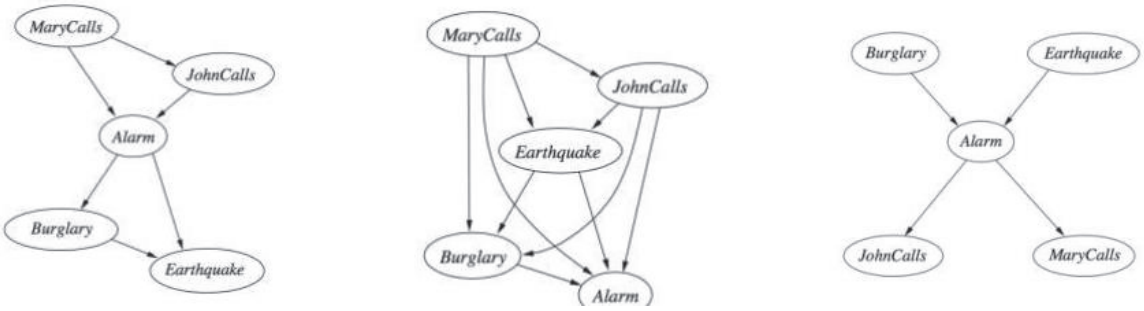
$$P(j,m,a,\neg b,\neg e)$$
$$= P(j|parents(j))P(m|parents(m)P(a|parents(a))$$
$$P(\neg b\,|parents(\neg b))P(\neg e\,|parents(\neg e))$$
$$= P(j|a)P(m|a)P(a|\neg b\wedge\neg e)P(\neg b)P(\neg e)$$
$$= 0.90\times0.70\times0.001\times0.999\times0.998=0.000628$$



1. A method for constructing Bayesian networks(1) •How to construct a GOOD Bayesian network •Full Joint Distribution $P\ x1\ ,\ \ldots\ldots,\ x\ n = i{=}1\ n\ P\ xi\ xi{-}1\ ,\ \ldots\ldots,\ x1\ )\ P\ x1\ ,\ \ldots\ldots,\ x\ n = i{=}1\ n\ P\ xi\ Parent(xi\ ))$ •Correct representation oonly if each node is conditionally independent of its other predecessors in the

node ordering, given its parents The parents of node Xi should contain all those nodes in X1,..,Xi−1 that directly influence Xi.

2. 13. A method for constructing Bayesian networks(2) •Ex oSuppose we have completed the network in Figure except for choices of parents for MaryCalls ♣MaryCalls is certainly influenced by whether there is a Burglary or an Earthquake, but not directly influenced ♣Also, given the state of the alarm, whether John calls has no influence on Mary's calling P(MaryCalls | JohnCalls, Alarm, Earthquake, Burglary) = P(MaryCalls | Alarm)
3. 14. Compactness and node ordering •Bayesian network can often be far more compact than the full joint distribution •It may not be worth the additional complexity in the network for the small gain in accuracy. •The correct procedure for adding a node is to first add the root cause first and then give the variables that they affect
4. 15. Compactness and node ordering •We will get a compact Bayesian network only if we choose the node ordering well •What happens if we happen to choose the wrong order? MaryCalls→JohnCalls →Alarm→Burglary →Earthquake MaryCalls→JohnCalls →Earthquake→Burgla ry→Alarm Burglary→Earthquake →Alarm→MaryCalls →JohnCalls



The Semantics of Bayesian Networks:

The two ways to understand the meaning of Bayesian Networks oTo see the network as a representation of the joint probability distribution ♣To be helpful in understanding how to construct networks, oTo view it as an encoding of a collection of conditional independence statements ♣To be helpful in designing inference procedures

Efficient Representation of Conditional Distributions:

•CPT cannot handle large number or continuous value varibles. •Relationships between parents and children are usually describable by some proper canonical distribution. •Use the deterministic nodes to demonstrate relationship. oValues are specified by some function. onondeterminism(no uncertainty) Ex. X = f(parents(X)) oCan be logical ♣NorthAmerica ↔ Canada ∨ US ∨ Mexico oOr numercial ♣Water level = inflow + precipitation – outflow – evaporation

•Uncertain relationships can be characterized by noisy logical relationships. •Ex. noisy-OR relation. oLogical OR with probability oEx. Cold ∨ Flu ∨ Malaria → Fever ♣In the real world, catching a cold sometimes does not induce fever. ♣There is some probability of catching a cold and having a fever.

Noisy-OR oAll possible causes are listed. (the missing can be covered by leak node) oCompute probability from the inhibition probability

$$P(x_i \mid parents(X_i)) = 1 - \prod_{\{j:X_j = true\}} q_j \ ,$$

Approximate Inference in Bayesian Networks:

Difficult to calculate multiply connected networks •It is essential to consider approximate inference methods •Monte Carlo algorithms oRandomized sampling algorithms otwo families of algorithms: direct sampling and Markov chain sampling oApply to the computation

Relational and First-order Probability models:

Bayesian networks are essentially propositional logic. •The set of random variables is fixed and finite. •However, if the number becomes large, intractable. •Need another method to represent the model

The set of first-order models is infinite. •Use database semantics instead called "Relational Probability models". •Make unique names assumption and assume domain closure. •Like first-order logic oConstant oFunction oPredicate symbols

Other Approaches to Uncertain Reasoning:

Rule-based methods for uncertain reasoning •Emerged from logical inference •Require 3 desirable properties oLocality : If A B, we can conclude B given evidence A without worrying about any other rules. ♣But in probabilistic systems, we need to consider all evidence. oDetachment : If we can derive B, we can use it without caring how it was derived. oTruth-functionality : truth value of complex sentences can be computed from the truth of the components. ♣Probability combination does not work this way

UNIT IV:

LEARNING

Learning is the improvement of performance with experience over time.

Learning element is the portion of a learning AI system that decides how to modify the performance element and implements those modifications.

We all learn new knowledge through different methods, depending on the type of material to be learned, the amount of relevant knowledge we already possess, and the environment in which the learning takes place. There are five methods of learning . They are,

1. Memorization (rote learning)
2. Direct instruction (by being told)
3. Analogy
4. Induction
5. deduction

Learning by memorizations is the simplest from of le4arning. It requires the least amount of inference and is accomplished by simply copying the knowledge in the same form that it will be used directly into the knowledge base.

Example:- Memorizing multiplication tables, formulate , etc.

Direct instruction is a complex form of learning. This type of learning requires more inference than role learning since the knowledge must be transformed into an operational form before learning when a teacher presents a number of facts directly to us in a well organized manner. Analogical learning is the process of learning a new concept or solution through the use of similar known concepts or solutions. We use this type of learning when solving problems on an exam where previously learned examples serve as a guide or when make frequent use of analogical learning. This form of learning requires still more inferring than either of the previous forms. Since difficult transformations must be made between the known and unknown situations. Learning by induction is also one that is used frequently by humans . it is a powerful form of learning like analogical learning which also require s more inferring than the first two methods. This learning re quires the use of inductive inference, a form of invalid but useful inference. We use inductive learning ofinstances of examples of the concept. For example we learn the concepts of color or sweet taste after experiencing the sensations associated with several examples of colored objects or sweet foods.

Deductive learning is accomplished through a sequence of deductive inference steps using known facts. From the known facts, new facts or relationships are logically derived. Deductive learning usually requires more inference than the other methods.

Review Questions:-

1. what is perception ?

2. How do we overcome the Perceptual Problems?
3. Explain in detail the constraint satisfaction waltz algorithm?
4. What is learning ?
5. What is Learning element ?
6. List and explain the methods of learning?

Types of learning:- Classification or taxonomy of learning types serves as a guide in studying or comparing a differences among them. One can develop learning taxonomies based on the type of knowledge representation used (predicate calculus , rules, frames), the type of knowledge learned (concepts, game playing, problem solving), or by the area of application(medical diagnosis , scheduling , prediction and so on).

The classification is intuitively more appealing and is one which has become popular among machine learning researchers . it is independent of the knowledge domain and the representation scheme is used. It is based on the type of inference strategy employed or the methods used in the learning process. The five different learning methods under this taxonomy are:

Memorization (rote learning)
Direct instruction(by being told)
Analogy

Induction
Deduction

Learning by memorization is the simplest form of learning. It requires the least5 amount of inference and is accomplished by simply copying the knowledge in the same form that it will be

used directly into the knowledge base. We use this type of learning when we memorize multiplication tables ,

for example.

A slightly more complex form of learning is by direct instruction. This type of learning requires more understanding and inference than role learning since the knowledge must be transformed into an operational form before being integrated into the knowledge base. We use this type of learning when a teacher presents a number of facts directly to us in a well organized manner.

The third type listed, analogical learning, is the process of learning an ew concept or solution through the use of similar known concepts or solutions. We use this type of learning when solving problems on an examination where previously learned examples serve as a guide or when we learn to drive a truck using our knowledge of car driving. We make frewuence use of analogical learning. This form of learning requires still more inferring than either of the previous forms, since difficult transformations must be made between the known and unknown situations. This is a kind of application of knowledge in a new situation.

The fourth type of learning is also one that is used frequency by humans. It is a powerful form of learning which, like analogical learning, also requires more inferring than the first two methods. This form of learning requires the use of inductive inference, a form of invalid but useful inference. We use inductive learning when wed formulate a general concept after seeing a number of instance or examples of the concept. For example, we learn the concepts of color sweet taste after experiencing the sensation associated with several examples of colored objects or sweet foods.

The final type of acquisition is deductive learning. It is accomplished through a sequence of deductive inference steps using known facts. From the known facts, new facts or relationships are logically derived. Deductive learning usually requires more inference than the other methods. The inference method used is, of course , a deductive type, which is a valid from of inference.

In addition to the above classification, we will sometimes refer to learning methods as wither methods or knowledge-rich methods. Weak methods are general purpose methods in which little or no initial knowledge is available. These methods are more mechanical than the classical AI knowledge – rich methods. They often rely on a form of heuristics search in the learning process.

Rote Learning

Rote learning is the basic learning activity. **Rote learning** is a memorization technique based

on repetition. It is also called memorization because the knowledge, without any modification is, simply copied into the knowledge base. As computed values are stored, this technique can save a significant amount of time.

Rote learning technique can also be used in complex learning systems provided sophisticated techniques are employed to use the stored values faster and there is a generalization to keep the number of stored information down to a manageable level. Checkers-playing program, for ex

The idea is that one will be able to quickly recall the meaning of the material the more one repeats it. Some of the alternatives to rote learning include meaningful learning, associative learning, and active learning. ample, uses this technique to learn the board positions it evaluates in its look-ahead search.

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. Moreover the reliability of the source of knowledge should be considered.

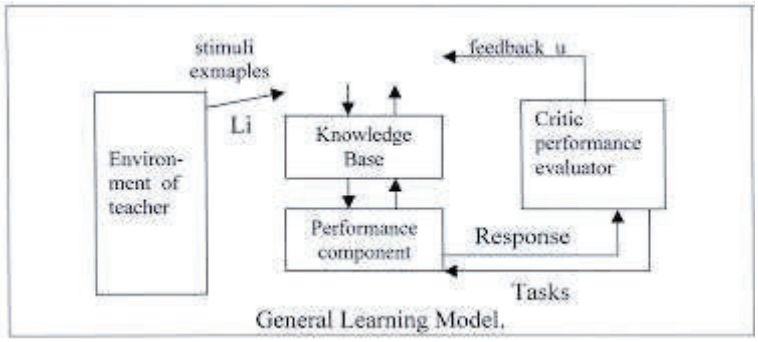
The system should ensure that the new knowledge is conflicting with the existing knowledge. FOO (First Operational Operationaliser), for example, is a learning system which is used to learn the game of Hearts. It converts the advice which is in the form of principles, problems, and methods into effective executable (LISP) procedures (or knowledge). Now this knowledge is ready to use.

General Learning Model.

General Learning Model: - AS noted earlier, learning can be accomplished using a number of different methods, such as by memorization facts, by being told, or by studying examples like problem solution. Learning requires that new knowledge structures be created from some form of input stimulus. This new knowledge must then be assimilated into a knowledge base and be tested in some way for its utility. Testing

means that the knowledge should be used in performance of some task from which meaningful feedback can be obtained, where the feedback provides some measure of the accuracy and usefulness of the newly acquired knowledge.

General Learning Model



General Learning Model.

general learning model is depicted in figure 4.1 where the environment has been included as a part of the overall learner system. The environment may be regarded as either a form of nature which produces random stimuli or as a more organized training source such as a teacher which provides carefully selected training examples for the learner component. The actual form of environment used will depend on the particular learning paradigm. In any case, some representation language must be assumed for communication between the environment and the learner. The language may be the same representation scheme as that used in the knowledge base (such as a form of predicate calculus). When they are hosen to be the same, we say the single representation trick is being used. This usually results in a simpler implementation since it is not necessary to transform between two or more different representations.

For some systems the environment may be a user working at a keyboard . Other systems will use program modules to simulate a particular environment. In even more realistic cases the system will have real physical sensors which interface with some world environment.

Inputs to the learner component may be physical stimuli of some type or descriptive , symbolic training examples. The information conveyed to the learner component is used to create and modify knowledge structures in the knowledge base. This same knowledge is used by the performance component to carry out some tasks, such as solving a problem playing a game, or classifying instances of some concept.given a task, the performance component produces a response describing its action in performing the task. The critic module then evaluates this response relative to an optimal response.

Feedback , indicating whether or not the performance was acceptable , is then sent by the critic module to the learner component for its subsequent use in modifying the

structures in the knowledge base. If proper learning was accomplished, the system's performance will have improved with the changes made to the knowledge base.

The cycle described above may be repeated a number of times until the performance of the system has reached some acceptable level, until a known learning goal has been reached, or until changes ceases to occur in the knowledge base after some chosen number of training examples have been observed.

There are several important factors which influence a system's ability to learn in addition to the form of representation used. They include the types of training provided, the form and extent of any initial background knowledge , the type of feedback provided, and the learning algorithms used.

The type of training used in a system can have a strong effect on performance, much the same as it does for humans. Training may consist of randomly selected instance or examples that have been carefully selected and ordered for presentation. The instances may be positive examples of some concept or task a being learned, they may be negative, or they may be mixture of both positive and negative. The instances may be well focused using only relevant information, or they may contain a variety of facts and details including irrelevant data.

There are Many forms of learning can be characterized as a search through a space of possible hypotheses or solutions. To make learning more efficient. It is necessary to constrain this search process or reduce the search space. One method of achieving this is through the use of background knowledge which can be used to constrain the search space or exercise control operations which limit the search process.

Feedback is essential to the learner component since otherwise it would never know if the knowledge structures in the knowledge base were improving or if they were adequate for the performance of the given tasks. The feedback may be a simple yes or no type of evaluation, or it

may contain more useful information describing why a particular action was good or bad. Also , the feedback may be completely reliable, providing an accurate assessment of the performance or it may contain noise, that is the feedback may actually be incorrect some of the time. Intuitively , the feedback must be accurate more than 50% of the time; otherwise the system carries useful information, the learner should also to build up a useful corpus of knowledge quickly. On the other hand, if the feedback is noisy or unreliable, the learning process may be very slow and the resultant knowledge incorrect.

Learning Neural Network

Perceptron

- The perceptron an invention of (1962) Rosenblatt was one of the earliest neural network models.
- Also, It models a neuron by taking a weighted sum of its inputs and sending the output 1 if the sum is greater than some adjustable threshold value (otherwise it sends 0).

$$g(x) = \sum_{k=0}^{n} WkXk \qquad Output(x) = \begin{cases} 1 & \text{if } g(x) > 0 \\ 0 & \text{if } g(x) > 0 \end{cases}$$

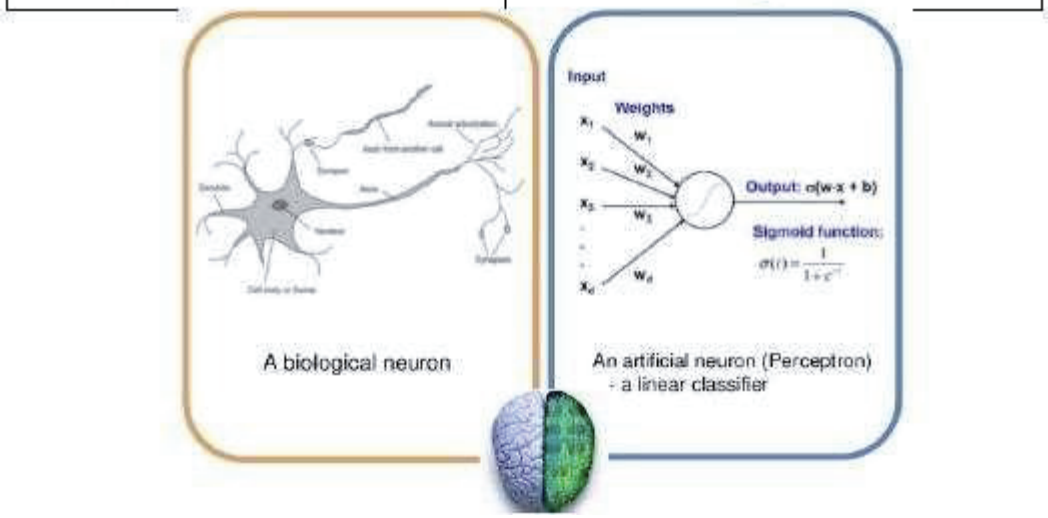


Figure: A neuron & a Perceptron

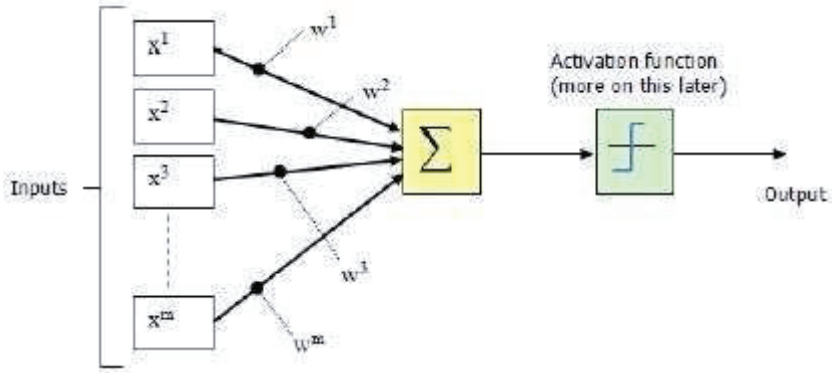


Figure: Perceptron with adjustable threshold

- In case of zero with two inputs g(x) = w0 + w1x1 + w2x2 = 0
- x2 = -(w1/w2)x1 – (w0/w2) → equation for a line
- the location of the line is determined by the weight w0 w1 and w2
- if an input vector lies on one side of the line, the perceptron will output 1
- if it lies on the other side, the perception will output 0
- Moreover, Decision surface: a line that correctly separates the training instances corresponds to a perfectly function perceptron.

Perceptron Learning Algorithm

Given: A classification problem with n input feature ($x_1, x_2, \ldots, x_n$) and two output classes. Compute A set of weights ($w_0, w_1, w_2, \ldots, w_n$) that will cause a perceptron to fire whenever the input falls into the first output class.

1. Create a perceptron with n+ 1 input and n+ 1 weight, where the x_0 is always set to 1.
2. Initialize the weights ($w_0, w_1, \ldots, w_n$) to random real values.
3. Iterate through the training set, collecting all examples *misclassified* by the current set of weights.
4. If all examples are classified correctly, output the weights and quit.
5. Otherwise, compute the vector sum S of the misclassified input vectors where each vector has the form ($x_0, x_1, \ldots, X_n$). In creating the sum, add to S a vector x if x is an input for which the perceptron incorrectly fails to fire, but – x if x is an input for which the perceptron incorrectly fires. Multiply sum by a scale factor η.
6. Moreover, Modify the weights ($w_0, w_1, \ldots, w_n$) by adding the elements of the vector S to them.
7. Go to step 3.

- The perceptron learning algorithm is a search algorithm. It begins with a random initial state and finds a solution state. The search space is simply all possible assignments of real values to the weights of the perception, and the search strategy is gradient descent.
- The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists.
- Moreover, This brings us to an important question. What problems can a perceptron solve? Recall that a single-neuron perceptron is able to divide the input space into two regions.
- Also, The perception can be used to classify input vectors that can be separated by a linear boundary. We call such vectors linearly separable.
- Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. It was the inability of the basic perceptron to solve such simple problems that are not linearly separable or non-linear.

Genetic Learning

Supervised Learning

Supervised learning is the machine learning task of inferring a function from labeled training data.

Moreover, The training data consist of a set of training examples.

In supervised learning, each example a pair consisting of an input object (typically a vector) and the desired output value (also called the supervisory signal).

Training set

A training set a set of data used in various areas of information science to discover potentially predictive relationships.

Training sets used in artificial intelligence, machine learning, genetic programming, intelligent systems, and statistics.

In all these fields, a training set has much the same role and often used in conjunction with a test set.

Testing set

A **test set** is a set of data used in various areas of information science to assess the strength and utility of a predictive relationship.

Moreover, Test sets are used in artificial intelligence, machine learning, genetic programming, and statistics. In all these fields, a test set has much the same role.

Accuracy of classifier: Supervised learning

In the fields of science, engineering, industry, and statistics. The accuracy of a measurement system is the degree of closeness of measurements of a quantity to that quantity's actual (true) value.

Sensitivity analysis: Supervised learning

Similarly, Local Sensitivity as correlation coefficients and partial derivatives can only use, if the correlation between input and output is linear.

Regression: Supervised learning

In statistics, **regression analysis** is a statistical process for estimating the relationships among variables.

Moreover, It includes many techniques for modeling and analyzing several variables. When the focus on the relationship between a dependent variable and one or more independent variables. More specifically, regression analysis helps one understand how the typical value of the dependent variable (or 'criterion variable') changes when any one of the independent variables varied. Moreover, While the other independent variables held fixed.

1. *Learning from Example : Induction*

A process of learning by example. The system tries to induce a general rule from a set of observed instances. The learning methods extract rules and patterns out of massive data sets.

The learning processes belong to supervised learning, does classification and constructs class definitions, called induction or concept learning.

The techniques used for constructing class definitions (or concept leaning) are :

- Winston's Learning program
- Version Spaces
- Decision Trees

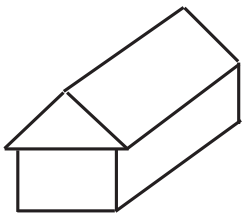
1.1 *Winston's Learning*

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

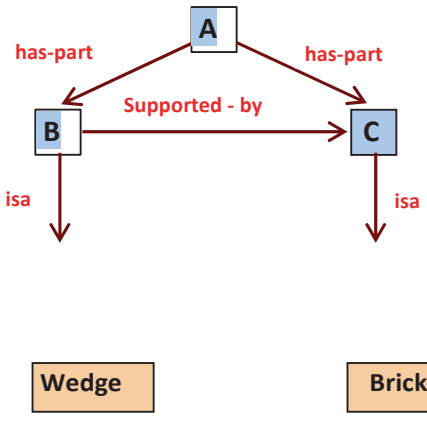
Example : Concepts such a "house".

- Start with input, a line drawing of a blocks world structure. It learned
  Concepts House, Tent, Arch as :
  brick (rectangular block) with a **wedge** (triangular block) suitably placed on top of it, **tent** – as 2 wedges touching side by side, or an **arch** – as 2 non-touching bricks supporting a third wedge or brick.
- The program for Each concept is learned through near miss. A near miss is an object that is not an instance of the concept but a very similar to such instances.
- The program uses procedures to analyze the drawing and construct a semantic net representation.
- An example of such an structural for the house is shown below.

Object - house



Semantic net



- Node **A** represents entire structure, which is composed of two parts : node **B**, a

Wedge, and node **C**, a Brick.

Links in network include supported-by**,** has-part, and isa**.**

- ***Winston's Program***
  - Winston's program followed 3 basic steps in concept formulation:

    1. Select one known instance of the concept. Call this the concept definition.
    2. Examine definitions of other known instance of the concept. Generalize the definition to include them.
    3. Examine descriptions of near misses. Restrict the definition to exclude these.

  - Both steps 2 and 3 of this procedure rely heavily on comparison process by which similarities and differences between structures can be detected.
  - Winston's program can be similarly applied to learn other concepts such as "ARCH".

LEARNING DECISION TREES:

- Come up with a set of attributes to describe the object or situation.
- Collect a complete set of examples (training set) from which the decision tree can derive a hypothesis to define (answer) the goal predicate.

Decision Tree Example:

Problem: decide whether to wait for a table at a restaurant, based on the following attributes:

1. Alternate: is there an alternative restaurant nearby?
2. Bar: is there a comfortable bar area to wait in?
3. Fri/Sat: is today Friday or Saturday?
4. Hungry: are we hungry?

5. Patrons: number of people in the restaurant (None, Some, Full)

6. Price: price range ($, $$, $$$)

7. Raining: is it raining outside?

8. Reservation: have we made a reservation?

9. Type: kind of restaurant (French, Italian, Thai, Burger)

10. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

Logical Representation of a Path

r [Patrons(r, full)□ Wait_Estimate(r, 10-30)□ Hungry(r, yes)] □ Will_Wait(r)

Expressiveness of Decision Trees

- Any Boolean function can be written as a decision tree
- E.g., for Boolean functions, truth table row → path to leaf:
- Trivially, there is a consistent decision tree for any training set with one path to leaf for each

example (unless f nondeterministic in x) but it probably won't generalize to new examples

- Prefer to find more compact decision trees Limitations
  - Can only describe one object at a time.
  - Some functions require an exponentially large decision tree.
- E.g. Parity function, majority function
- Decision trees are good for some kinds of functions, and bad for others.
- There is no one efficient representation for all kinds of functions.

Principle Behind the Decision-Tree-Learning Algorithm

- Uses a general principle of inductive learning often called Ockham's razor: "The most likely hypothesis is the simplest one that is consistent with all observations."

- Decision trees can express any function of the input attributes.

DECISION TREE LEARNING ALGORITHM:

- Aim: find a small tree consistent with the training examples

- Idea: (recursively) choose "most significant" attribute as root of (sub)tree Choosing an attribute tests:

- Idea: a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"

- Patrons? is a better choice Attribute-based representations

- Examples described by attribute values (Boolean, discrete, continuous)

- E.g., situations where I will/won't wait for a table:

- Classification of examples is positive (T) or negative (F) Using information theory

- To implement Choose-Attribute in the DTL algorithm

- Information Content (Entropy):

$I(P(v1), \ldots, P(vn)) = \Sigma i=1 \; -P(vi) \log2 P(vi)$

- For a training set containing p positive examples and n negative examples:

A chosen attribute A divides the training set E into subsets E1, … , Ev according to their values for A, where A has v distinct values. Information Gain (IG) or reduction in entropy from the attribute test: remainder (A),

- Choose the attribute with the largest IG

- For the training set, p = n = 6, I(6/12, 6/12) = 1 bit

- Patrons has the highest IG of all attributes and so is chosen by the DTL algorithm as the root

Assessing the performance of the learning algorithm:

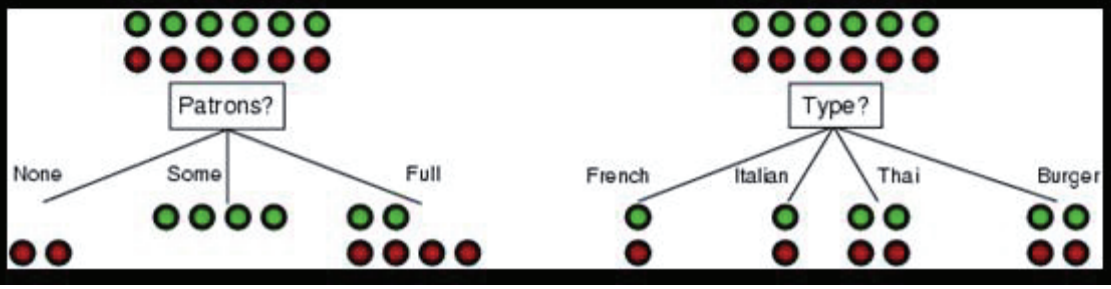
- A learning algorithm is good if it produces hypotheses that do a good job of predicating the classifications of unseen examples

- Test the algorithm's prediction performance on a set of new examples, called a test set.

```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value v_i of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← DTL(examples_i, attributes − best, MODE(examples))
            add a branch to tree with label v_i and subtree subtree
        return tree
```

Choosing an attribute tests:

- **Idea:** a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"



Patrons has the highest IG of all attributes and so is chosen by the DTL algorithm as the root

- Choose the attribute with the largest IG
- For the training set, p = n = 6, I(6/12, 6/12) = 1 bit
- Assessing the performance of the learning algorithm:
- A learning algorithm is good if it produces hypotheses that do a good job of predicating the classifications of unseen examples

UNIT V

Expert systems:

Expert system = knowledge + problem-solving methods..... A knowledge base that captures the domain-specific knowledge and an inference engine that consists of algorithms for manipulating the knowledge represented in the knowledge base to solve a problem presented to the system.

Expert systems (ES) are one of the prominent research domains of AI. It is introduced by the researchers at Stanford University, Computer Science Department.

What are Expert Systems?

The expert systems are the computer applications developed to solve complex problems in a particular domain, at the level of extra-ordinary human intelligence and expertise.

Characteristics of Expert Systems

- High performance
- Understandable
- Reliable
- Highly responsive

Capabilities of Expert Systems

The expert systems are capable of −

- Advising
- Instructing and assisting human in decision making
- Demonstrating
- Deriving a solution
- Diagnosing
- Explaining
- Interpreting input
- Predicting results
- Justifying the conclusion
- Suggesting alternative options to a problem

They are incapable of −

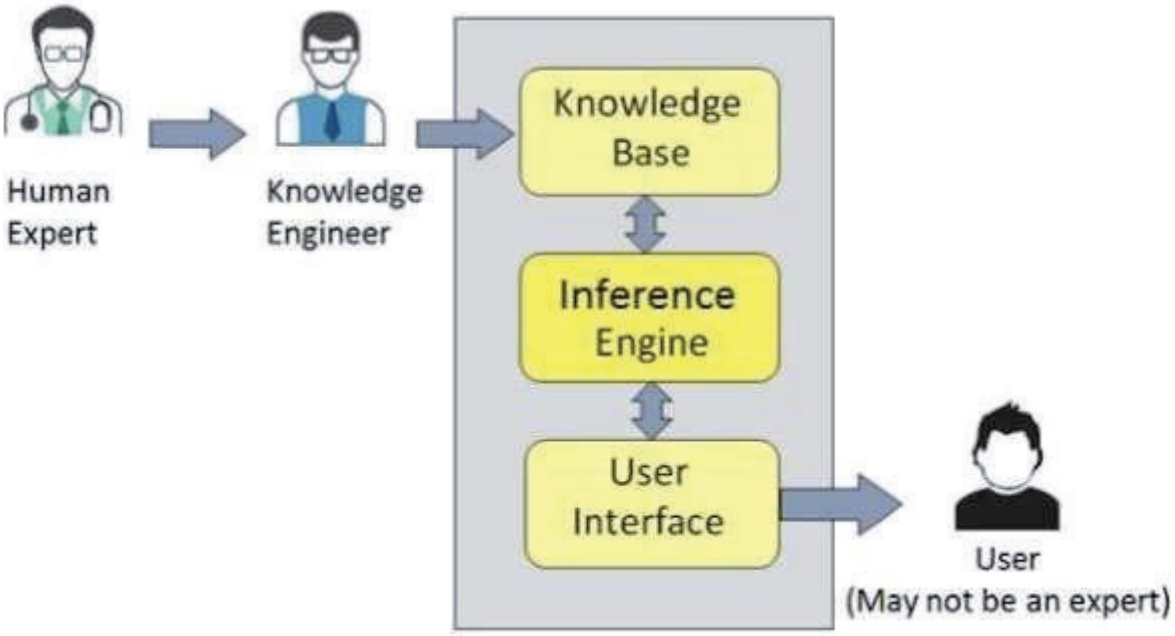
- Substituting human decision makers
- Possessing human capabilities

- Producing accurate output for inadequate knowledge base
- Refining their own knowledge

Components of Expert Systems

The components of ES include −

- Knowledge Base
- Inference Engine
- User Interface



Let us see them one by one briefly −

Knowledge Base

It contains domain-specific and high-quality knowledge. Knowledge is required to exhibit intelligence. The success of any ES majorly depends upon the collection of highly accurate and precise knowledge.

What is Knowledge?

The data is collection of facts. The information is organized as data and facts about the task domain. Data, information, and past experience combined together are termed as knowledge.

Components of Knowledge Base

The knowledge base of an ES is a store of both, factual and heuristic knowledge.

- Factual Knowledge − It is the information widely accepted by the Knowledge Engineers and scholars in the task domain.
- Heuristic Knowledge − It is about practice, accurate judgement, one's ability of evaluation, and guessing.

Knowledge representation

It is the method used to organize and formalize the knowledge in the knowledge base. It is in the form of IF-THEN-ELSE rules.

Knowledge Acquisition

The success of any expert system majorly depends on the quality, completeness, and accuracy of the information stored in the knowledge base.

The knowledge base is formed by readings from various experts, scholars, and the Knowledge Engineers. The knowledge engineer is a person with the qualities of empathy, quick learning, and case analyzing skills.

He acquires information from subject expert by recording, interviewing, and observing him at work, etc. He then categorizes and organizes the information in a meaningful way, in the form of IF-THEN-ELSE rules, to be used by interference machine. The knowledge engineer also monitors the development of the ES.

Inference Engine

Use of efficient procedures and rules by the Inference Engine is essential in deducting a correct, flawless solution.

In case of knowledge-based ES, the Inference Engine acquires and manipulates the knowledge from the knowledge base to arrive at a particular solution.

In case of rule based ES, it −

- Applies rules repeatedly to the facts, which are obtained from earlier rule application.
- Adds new knowledge into the knowledge base if required.
- Resolves rules conflict when multiple rules are applicable to a particular case.

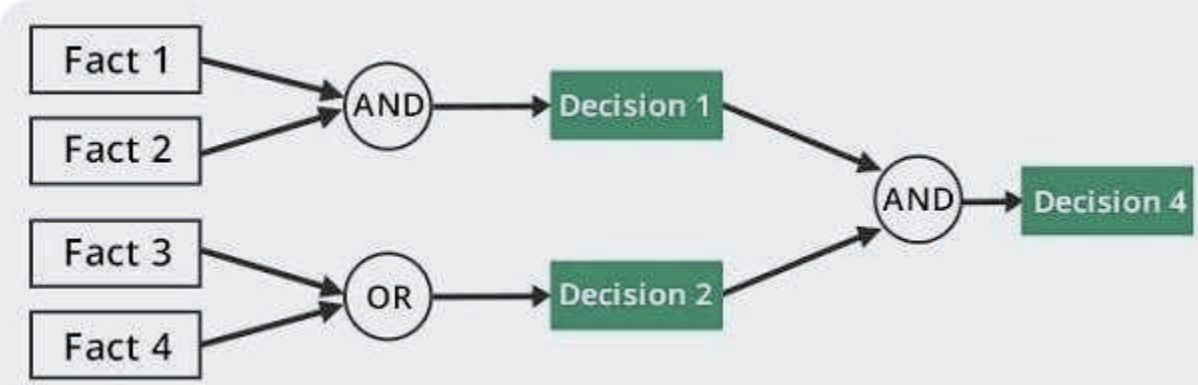
To recommend a solution, the Inference Engine uses the following strategies −

- Forward Chaining
- Backward Chaining

Forward Chaining

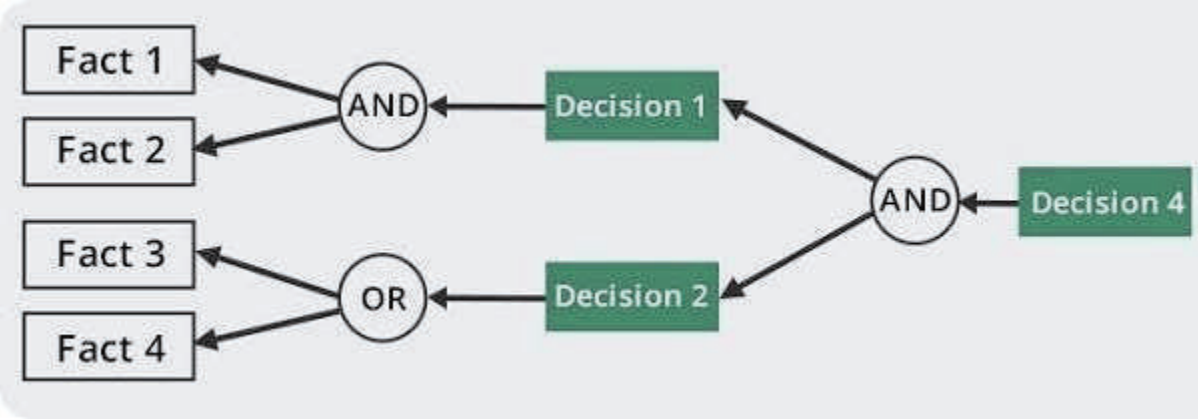
It is a strategy of an expert system to answer the question, “What can happen next?”

Here, the Inference Engine follows the chain of conditions and derivations and finally deduces the outcome. It considers all the facts and rules, and sorts them before concluding to a solution. This strategy is followed for working on conclusion, result, or effect. For example, prediction of share market status as an effect of changes in interest rates.



Backward Chaining

With this strategy, an expert system finds out the answer to the question, "Why this happened?" On the basis of what has already happened, the Inference Engine tries to find out which conditions could have happened in the past for this result. This strategy is followed for finding out cause or reason. For example, diagnosis of blood cancer in humans.



User Interface

User interface provides interaction between user of the ES and the ES itself. It is generally Natural Language Processing so as to be used by the user who is well-versed in the task domain. The user of the ES need not be necessarily an expert in Artificial Intelligence.

It explains how the ES has arrived at a particular recommendation. The explanation may appear in the following forms −

- Natural language displayed on screen.
- Verbal narrations in natural language.
- Listing of rule numbers displayed on the screen.

The user interface makes it easy to trace the credibility of the deductions.

Requirements of Efficient ES User Interface

- It should help users to accomplish their goals in shortest possible way.
- It should be designed to work for user's existing or desired work practices.
- Its technology should be adaptable to user's requirements; not the other way round.
- It should make efficient use of user input.

Expert Systems Limitations

No technology can offer easy and complete solution. Large systems are costly, require significant development time, and computer resources. ESs have their limitations which include −

- Limitations of the technology
- Difficult knowledge acquisition
- ES are difficult to maintain
- High development costs

Applications of Expert System

The following table shows where ES can be applied.

Application	Description
Design Domain	Camera lens design, automobile design.
Medical Domain	Diagnosis Systems to deduce cause of disease from observed data, conduction medical operations on humans.
Monitoring Systems	Comparing data continuously with observed system or with prescribed behavior such as leakage monitoring in long petroleum pipeline.
Process Control Systems	Controlling a physical process based on monitoring.
Knowledge Domain	Finding out faults in vehicles, computers.
Finance/Commerce	Detection of possible fraud, suspicious transactions, stock market trading, Airline scheduling, cargo scheduling.

Expert System Technology

There are several levels of ES technologies available. Expert systems technologies include −

- Expert System Development Environment − The ES development environment includes hardware and tools. They are −
  - Workstations, minicomputers, mainframes.
  - High level Symbolic Programming Languages such as LISt Programming (LISP) and PROgrammation en LOGique (PROLOG).
  - Large databases.
- Tools − They reduce the effort and cost involved in developing an expert system to large extent.
  - Powerful editors and debugging tools with multi-windows.
  - They provide rapid prototyping
  - Have Inbuilt definitions of model, knowledge representation, and inference design.
- Shells − A shell is nothing but an expert system without knowledge base. A shell provides the developers with knowledge acquisition, inference engine, user interface, and explanation facility. For example, few shells are given below −

- Java Expert System Shell (JESS) that provides fully developed Java API for creating an expert system.
- *Vidwan*, a shell developed at the National Centre for Software Technology, Mumbai in 1993. It enables knowledge encoding in the form of IF-THEN rules.

Development of Expert Systems: General Steps

The process of ES development is iterative. Steps in developing the ES include −

Identify Problem Domain

- The problem must be suitable for an expert system to solve it.
- Find the experts in task domain for the ES project.
- Establish cost-effectiveness of the system.

Design the System

- Identify the ES Technology
- Know and establish the degree of integration with the other systems and databases.
- Realize how the concepts can represent the domain knowledge best.

Develop the Prototype

From Knowledge Base: The knowledge engineer works to −

- Acquire domain knowledge from the expert.
- Represent it in the form of If-THEN-ELSE rules.

Test and Refine the Prototype

- The knowledge engineer uses sample cases to test the prototype for any deficiencies in performance.
- End users test the prototypes of the ES.

Develop and Complete the ES

- Test and ensure the interaction of the ES with all elements of its environment, including end users, databases, and other information systems.
- Document the ES project well.
- Train the user to use ES.

Maintain the ES

- Keep the knowledge base up-to-date by regular review and update.
- Cater for new interfaces with other information systems, as those systems evolve.

Benefits of Expert Systems

- Availability − They are easily available due to mass production of software.
- Less Production Cost − Production cost is reasonable. This makes them affordable.
- Speed − They offer great speed. They reduce the amount of work an individual puts in.
- Less Error Rate − Error rate is low as compared to human errors.
- Reducing Risk − They can work in the environment dangerous to humans.
- Steady response − They work steadily without getting motional, tensed or fatigued.

Expert System.

DEFINITION - An expert system is a computer program that simulates the judgement and behavior of a human or an organization that has expert knowledge and experience in a particular field. Typically, such a system contains a knowledge base containing accumulated experience

and a set of rules for applying the knowledge base to each particular situation that is described to the program. Sophisticated expert systems can be enhanced with additions to the knowledge base or to the set of rules.

Among the best-known expert systems have been those that play chess and that assist in medical diagnosis.

An **expert system** is software that attempts to provide an answer to a problem, or clarify uncertainties where normally one or more human experts would need to be consulted. Expert systems are most common in a specific problem domain, and is a traditional application and/or

subfield of artificial intelligence (AI). A wide variety of methods can be used to simulate the performance of the expert; however, common to most or all are: 1) the creation of a knowledge base which uses some knowledge representation structure to capture the knowledge of

the Subject Matter Expert (SME); 2) a process of gathering that knowledge from the SME and codifying it according to the structure, which is called knowledge engineering; and 3) once the system is developed, it is placed in the same real world problem solving situation as the human SME, typically as an aid to human workers or as a supplement to some information system.

Expert systems may or may not have learning components.

factors

The MYCIN rule-based expert system introduced a quasi-probabilistic approach called certainty factors, whose rationale is explained below.

A human, when reasoning, does not always make statements with 100% confidence: he might venture, "If Fritz is green, then he is probably a frog" (after all, he might be a chameleon). This type of reasoning can be imitated using numeric values called confidences. For example, if it is known that Fritz is green, it might be concluded with 0.85 confidence that he is a frog; or, if it is known that he is a frog, it might be concluded with 0.95 confidence that he hops. These certainty factor (CF) numbers quantify uncertainty in the degree to which the available evidence supports a hypothesis. They represent a degree of confirmation, and are not probabilities in a Bayesian sense. The CF calculus, developed by Shortliffe & Buchanan, increases or decreases the CF associated with a hypothesis as each new piece of evidence becomes available. It can be mapped to a probability update, although degrees of confirmation are not expected to obey the laws of probability. It is important to note, for example, that evidence for hypothesis H may have nothing to contribute to the degree to which Not_h is confirmed or disconfirmed (e.g., although a fever lends some support to a diagnosis of infection, fever does not disconfirm alternative hypotheses) and that the sum of CFs of many competing hypotheses may be greater than one (i.e., many hypotheses may be well confirmed based on available evidence).

The CF approach to a rule-based expert system design does not have a widespread following, in part because of the difficulty of meaningfully assigning CFs a priori. (The above example of

green creatures being likely to be frogs is excessively naive.) Alternative approaches to quasi-probabilistic reasoning in expert systems involve fuzzy logic, which has a firmer mathematical foundation. Also, rule-engine shells such as Drools and Jess do not support probability manipulation: they use an alternative mechanism called salience, which is used to prioritize the order of evaluation of activated rules.

In certain areas, as in the tax-advice scenarios discussed below, probabilistic approaches are not acceptable. For instance, a 95% probability of being correct means a 5% probability of being wrong. The rules that are defined in such systems have no exceptions: they are only a means of achieving software flexibility when external circumstances change frequently. Because rules are stored as data, the core software does not need to be rebuilt each time changes to federal and state tax codes are announced.

Chaining

Two methods of reasoning when using inference rules are forward chaining and backward chaining.

Forward chaining starts with the data available and uses the inference rules to extract more data until a desired goal is reached. An inference engine using forward chaining searches the inference rules until it finds one in which the if clause is known to be true. It then concludes the then clause and adds this information to its data. It continues to do this until a goal is reached. Because the data available determines which inference rules are used, this method is also classified as data driven.

Backward chaining starts with a list of goals and works backwards to see if there is data which will allow it to conclude any of these goals. An inference engine using backward chaining would search the inference rules until it finds one which has a then clause that matches a desired goal. If the if clause of that inference rule is not known to be true, then it is added to the list of goals.

SW Architecture.

The following general points about expert systems and their architecture have been outlined:

1. The sequence of steps taken to reach a conclusion is dynamically synthesized with each new case. The sequence is not explicitly programmed at the time that the system is built.

2. Expert systems can process multiple values for any problem parameter. This permits more than one line of reasoning to be pursued and the results of incomplete (not fully determined) reasoning to be presented.

3. Problem solving is accomplished by applying specific knowledge rather than specific technique. This is a key idea in expert systems technology. It reflects the belief that human experts do not process their knowledge differently from others, but they do possess different knowledge. With this philosophy, when one finds that their expert system does not produce the desired results, work begins to expand the knowledge base, not to re-program the procedures.

End user

There are two styles of user-interface design followed by expert systems. In the original style of user interaction, the software takes the end-user through an interactive dialog. In the following example, a backward-chaining system seeks to determine a set of restaurants to recommend:

Q. Do you know which restaurant you want to go to?

A. No

Q. Is there any kind of food you would particularly like?

A. No

Q. Do you like spicy food?

A. No

Q. Do you usually drink wine with meals?

A. Yes

Q. When you drink wine, is it French wine?

A. Yes

Participants

There are generally three individuals having an interaction in an expert system. Primary among these is the end-user, the individual who uses the system for its problem solving assistance. In the construction and maintenance of the system there are two other roles: the problem domain expert who builds the system and supplies the knowledge base, and a knowledge engineer who assists the experts in determining the representation of their knowledge, enters this knowledge into an explanation module and who defines the inference technique required to solve the problem. Usually the knowledge engineer will represent the problem solving activity in the form of rules. When these rules are created from domain expertise, the knowledge base stores the rules of the expert system.

Inference rule

An understanding of the "inference rule" concept is important to understand expert systems. An inference rule is a conditional statement with two parts: an if clause and a then clause. This rule is what gives expert systems the ability to find solutions to diagnostic and prescriptive problems. An example of an inference rule is:

If the restaurant choice includes French and the

occasion is romantic, Then the restaurant choice is

definitely Paul Bocuse.

Procedure node interface

The function of the procedure node interface is to receive information from the procedures coordinator and create the appropriate procedure call. The ability to call a procedure and receive information from that procedure can be viewed as simply a generalization of input from the external world. In some earlier expert systems external information could only be obtained in a

predetermined manner, which only allowed certain information to be acquired. Through the knowledge base, this expert system disclosed in the cross-referenced application can invoke any procedure allowed on its host system. This makes the expert system useful in a much wider class of knowledge domains than if it had no external access or only limited external access.

In the area of machine diagnostics using expert systems, particularly self-diagnostic applications, it is not possible to conclude the current state of "health" of a machine without some information. The best source of information is the machine itself, for it contains much detailed information that could not reasonably be provided by the operator.

The knowledge that is represented in the system appears in the rulebase. In the rulebase described in the cross-referenced applications, there are basically four different types of objects, with the associated information:

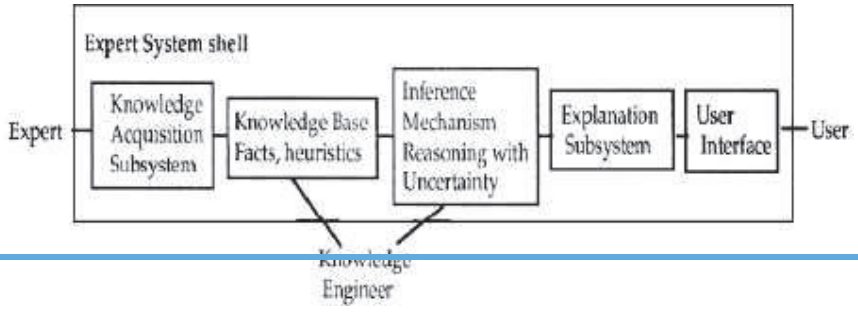
1. Classes: Questions asked to the user.

2. Parameters: Place holders for character strings which may be variables that can be inserted into a class question at the point in the question where the parameter is positioned.

3. Procedures: Definitions of calls to external procedures.

3. Rule nodes: Inferences in the system are made by a tree structure which indicates the rules or logic mimicking human reasoning. The nodes of these trees are called rule nodes. There are several different types of rule nodes.

Expert Systems/Shells. The E.S **shell** simplifies the process of creating a knowledge base. It is the **shell** that actually processes the information entered by a user relates it to the concepts contained in the knowledge base and provides an



assessment or solution for a particular problem.

Knowledge Acquisition

Knowledge acquisition is the process used to define the rules and ontologies required for a knowledge-based system. The phrase was first used in conjunction with expert systems to describe the initial tasks associated with developing an expert system, namely finding and interviewing domain experts and capturing their knowledge via rules, objects, and frame- based ontologies.

Expert systems were one of the first successful applications of artificial intelligence technology to real world business problems. Researchers at Stanford and other AI laboratories worked with doctors and other highly skilled experts to develop systems that could automate complex tasks such as medical diagnosis. Until this point computers had mostly been used to automate highly data intensive tasks but not for complex reasoning. Technologies such as inference engines allowed developers for the first time to tackle more complex problems.

As expert systems scaled up from demonstration prototypes to industrial strength applications it was soon realized that the acquisition of domain expert knowledge was one of if not the most critical task in the knowledge engineering process. This knowledge acquisition process became an intense area of research on its own. One of the earlier works on the topic used Batesonian theories of learning to guide the process.

One approach to knowledge acquisition investigated was to use natural language parsing and generation to facilitate knowledge acquisition. Natural language parsing could be performed on manuals and other expert documents and an initial first pass at the rules and objects could be developed automatically. Text generation was also extremely useful in generating explanations for system behavior. This greatly facilitated the development and maintenance of expert systems. A more recent approach to knowledge acquisition is a re-use based approach. Knowledge can be developed in ontologies that conform to standards such as the Web Ontology Language (OWL). In this way knowledge can be standardized and shared across a broad community of knowledge workers. One example domain where this approach has been successful