Artificial Intelligence

DIGITAL NOTES
B. TECH III YEAR – I SEM (R22)

CMR ENGINEERING COLLEGE, HYDERABAD

B.TECH III Year I Sem. **L T P C**
3 0 0 3

CS503PC: ARTIFICIAL INTELLIGENCE

(Common to CSE/CSD/IT)

Prerequisites:
1. Programming for problem solving, Data Structures.

Course Objectives:
- To learn the distinction between optimal reasoning Vs. human like reasoning
- To understand the concepts of state space representation, exhaustive search, heuristic search together with the time and space complexities.
- To learn different knowledge representation techniques.
- To understand the applications of AI, namely game playing, theorem proving, and machine learning.

Course Outcomes:
- Understand search strategies and intelligent agents
- Understand different adversarial search techniques
- Apply propositional logic, predicate logic for knowledge representation
- Apply AI techniques to solve problems of game playing, and machine learning.

UNIT - I
Introduction to AI, Intelligent Agents, problem-Solving Agents, Searching for Solutions, **Uninformed Search Strategies:** Breadth-first search, Uniform cost search, Depth-first search, Iterative deepening Depth-first search, Bidirectional search, Informed (Heuristic) Search Strategies: Greedy best-first search, A* search, Heuristic Functions, Beyond Classical Search: Hill-climbing search, Simulated annealing search, Local Search in Continuous Spaces

UNIT - II
Problem Solving by Search-II and Propositional Logic
Adversarial Search: Games, Optimal Decisions in Games, Alpha–Beta Pruning, Imperfect Real-Time Decisions.
Constraint Satisfaction Problems: Defining Constraint Satisfaction Problems, Constraint Propagation, Backtracking Search for CSPs, Local Search for CSPs, The Structure of Problems. Propositional Logic: Knowledge-Based Agents, The Wumpus World, Logic, Propositional Logic, Propositional Theorem Proving: Inference and proofs, Proof by resolution, Horn clauses and definite clauses, Forward and backward chaining, Effective Propositional Model Checking, Agents Based on Propositional Logic.

UNIT - III
Logic and Knowledge Representation
First-Order Logic: Representation, Syntax and Semantics of First-Order Logic, Using First-Order Logic, Knowledge Engineering in First-Order Logic.
Inference in First-Order Logic: Propositional vs. First-Order Inference, Unification and Lifting, Forward Chaining, Backward Chaining, Resolution.

UNIT - IV
Knowledge Representation: Ontological Engineering, Categories and Objects, Events. Mental Events and Mental Objects, Reasoning Systems for Categories, Reasoning with Default Information.
Classical Planning: Definition of Classical Planning, Algorithms for Planning with State-Space Search, Planning Graphs, other Classical Planning Approaches, Analysis of Planning approaches.
UNIT - V
Uncertain knowledge and Learning Uncertainty: Acting under Uncertainty, Basic Probability Notation, Inference Using Full Joint Distributions, Independence, Bayes' Rule and Its Use
Probabilistic Reasoning: Representing Knowledge in an Uncertain Domain, The Semantics of Bayesian Networks, Efficient Representation of Conditional Distributions, Approximate Inference in Bayesian Networks, Relational and First-Order Probability, Other Approaches to Uncertain Reasoning; Dempster-Shafer theory.

-------COMPUTER SCIENCE AND ENGINEERING-DATA SCIENCE 2022-23
CMRENGINEERING COLLEGE (AUOTONOMOUS), HYDERABAD.

TEXT BOOK:
1. Artificial Intelligence: A Modern Approach, Third Edition, Stuart Russell and Peter Norvig, Pearson Education.

REFERENCE BOOKS:
1. Artificial Intelligence, 3rd Edn, E. Rich and K. Knight (TMH)
2. Artificial Intelligence, 3rd Edn., Patrick Henry Winston, Pearson Education.
3. Artificial Intelligence, Shivani Goel, Pearson Education.
4. Artificial Intelligence and Expert systems – Patterson, Pearson Education

Artificial Intelligence

UNIT I:

Introduction, Intelligent Systems, Foundations of AI, Sub areas of AI and its applications. Solving Problems by Searching: Problem-Solving Agents, Searching for Solutions, Uninformed Search Strategies: Breadth-first search, Uniform cost search, Depth-first search, Iterative deepening Depth-first search, Bidirectional search, Informed (Heuristic) Search Strategies: Greedy best-first search, A* search, Heuristic Functions.

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.

Foundations of Artificial Intelligence:

- **Philosophy**

  e.g., foundational issues (can a machine think?), issues of knowledge and believe, mutual knowledge

- **Psychology and Cognitive Science**

  e.g., problem solving skills
- **Neuro-Science**

  e.g., brain architecture
- **Computer Science And Engineering**

  e.g., complexity theory, algorithms, logic and inference, programming languages, and system building.
- **Mathematics and Physics**

  e.g., statistical modeling, continuous mathematics,
- **Statistical Physics, and Complex Systems**.

Sub Areas of AI:

1) **Game Playing**

   Deep Blue Chess program beat world champion Gary Kasparov
2) **Speech Recognition**

   PEGASUS spoken language interface to American Airlines' EAASY SABRE reseration system, which allows users to obtain flight information and make reservations over the telephone. The 1990s has seen significant advances in speech recognition so that limited systems are now successful.
3) **Computer Vision**

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

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

   a. **Diagnostic Systems** : MYCIN system for diagnosing bacterial infections of the blood and suggesting treatments. Intellipath pathology diagnosis system (AMA approved). Pathfinder medical diagnosis system, which suggests tests and makes diagnoses. Whirlpool customer assistance center.

b. **System Configuration**

DEC's XCON system for custom hardware configuration. Radiotherapy treatment planning.

c. **Financial Decision Making**

Credit card companies, mortgage companies, banks, and the U.S. government employ AI systems to detect fraud and expedite financial transactions. For example, AMEX credit check.

d. **Classification Systems**

Put information into one of a fixed set of categories using several sources of information. E.g., financial decision making systems. NASA developed a system for classifying very faint areas in astronomical images into either stars or galaxies with very high accuracy by learning from human experts' classifications.

5) **Mathematical Theorem Proving**

Use inference methods to prove new theorems.

6) **Natural Language Understanding**

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

7) **Scheduling and Planning**

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

8) **Artificial Neural Networks:**

9) **Machine Learning**

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 andenvironments:

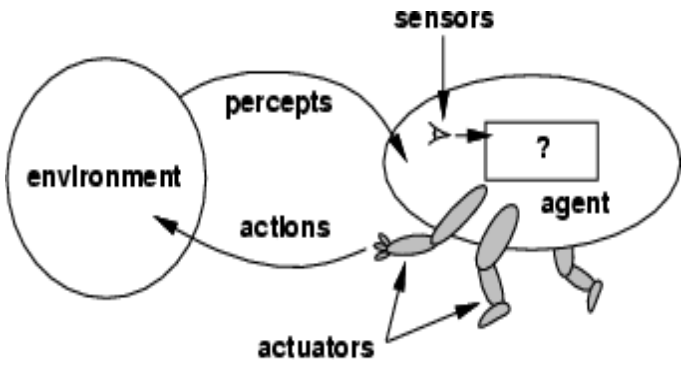


Fig 2.1: Agents and Environments

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 foractuators.
- ✓ A *robotic agent* might have cameras and infrared range finders for sensors and various motors foractuators.
- ✓ 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 PerceptSequence:

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 Agentprogram

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**.

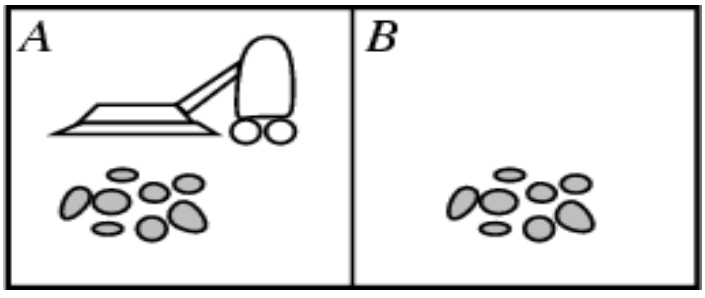


Fig 2.1.5: A vacuum-cleaner world with just two locations.

2.1.6 Agentfunction

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.

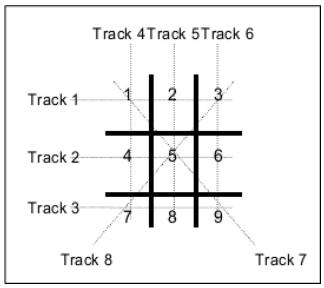


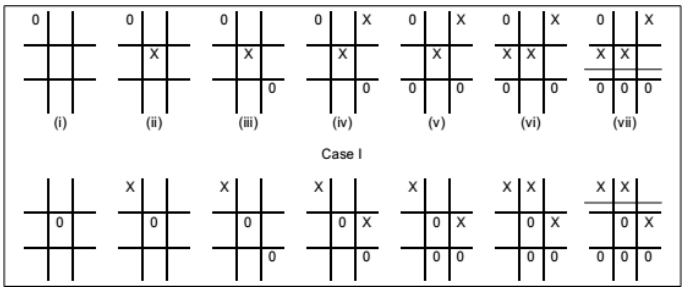
Fig 1.

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

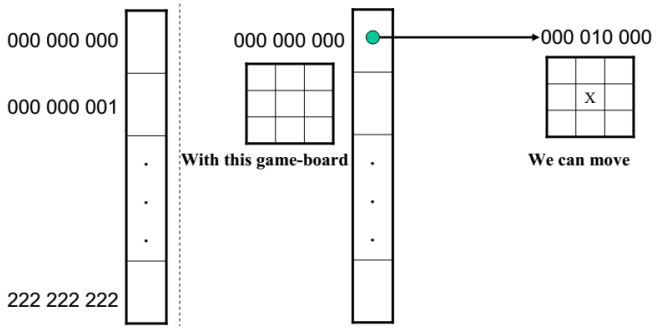
→ the vector is a ternary number

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)**

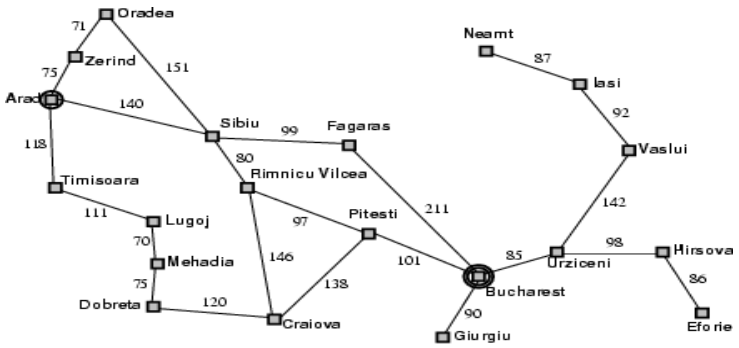
Specify a set of rules that describe the actions **(operations)** available

State-Space Problem Formulation:

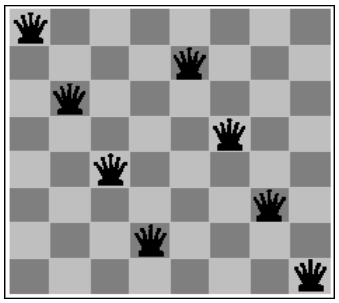
Example: A problem is defined by four items:

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 ≤x≤ 4, and 0 ≤y ≤3. Our initial state: (0, 0)

Goal Predicate - state = (2, y) where 0≤ y≤ 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) y > 0	→ (x,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	(x,y)	→ (x+y, 0)

into 4-gal jug	0 < x+y 4 and y 0	
8. Pour all of water from 4-gal jug	(x,y)	→ (0, x+y)
into 3-gal jug	0 < x+y 3 and x 0	

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

Control strategies

Control Strategies means how to decide which rule to apply next during the process of searching for a solution to a problem.

Requirement for a good Control Strategy

1. **It should cause motion**

   In water jug problem, if we apply a simple control strategy of starting each time from the top of rule list and choose the first applicable one, then we will never move towards solution.

2. **It should explore the solution space in a systematic manner**

   If we choose another control strategy, let us say, choose a rule randomly from the applicable rules then definitely it causes motion and eventually will lead to a solution. But one may arrive to same state several times. This is because control strategy is not systematic.

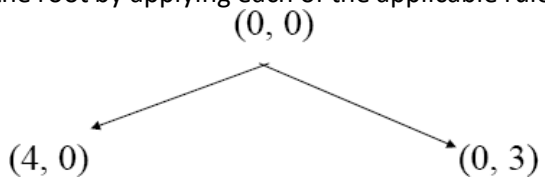
Systematic Control Strategies (Blind searches):

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.

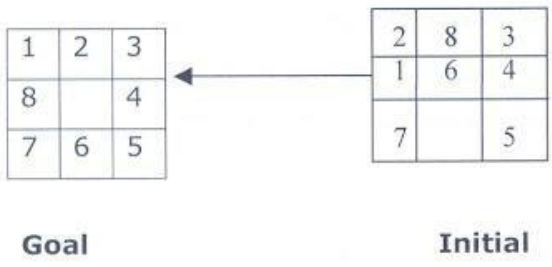
Generate all the offspring of the root by applying each of the applicable rules to the initial state**.**



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.

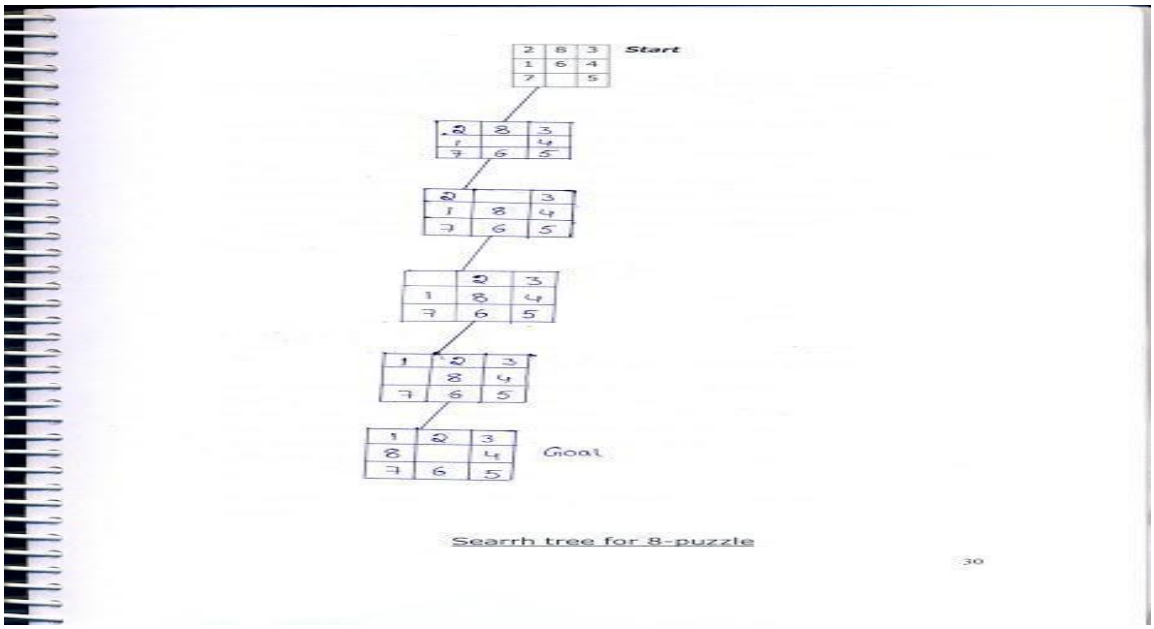
A move transforms one problem state into another state. The 8-puzzle is conveniently interpreted as having the following for moves. Move empty space (blank) to the left, move blank up, move blank to the right and move blank down,. These moves are modeled by production rules that operate on the state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

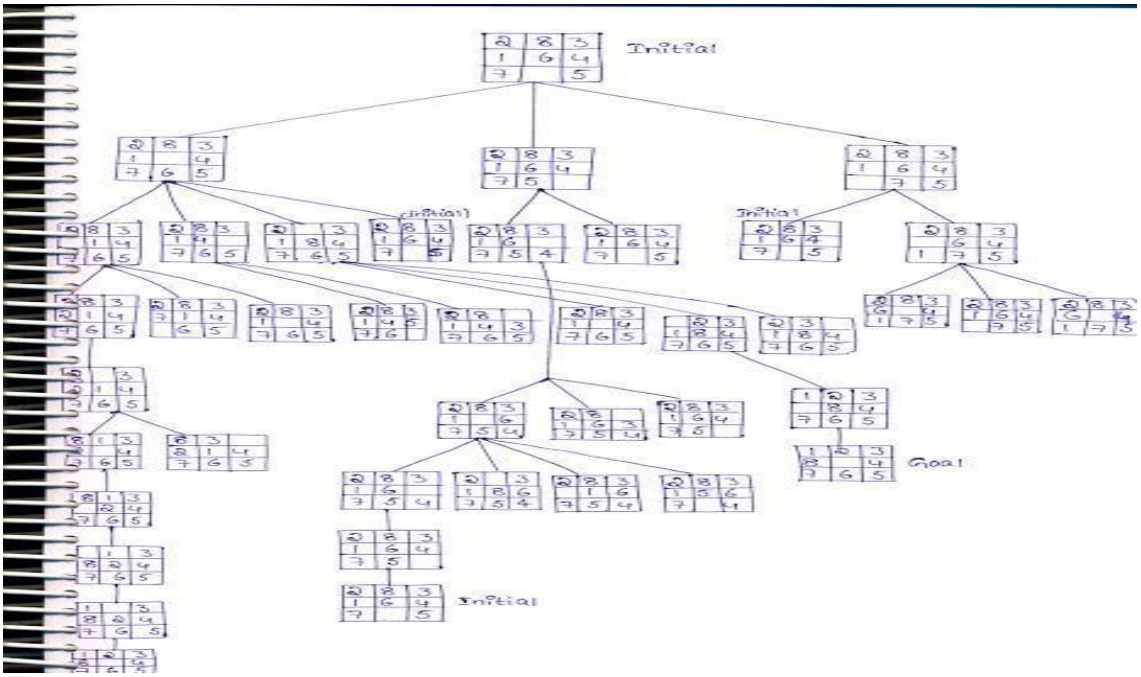
The rules each have preconditions that must be satisfied by a state description in order for them to be applicable to that state description. Thus the precondition for the rule associated with “move blank up” is derived from the requirement that the blank space must not already be in the top row.

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.

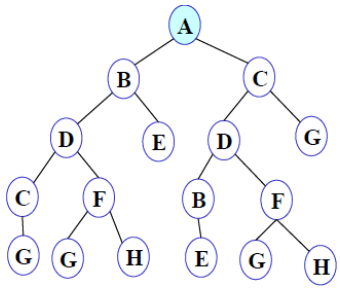


Figure 1

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.

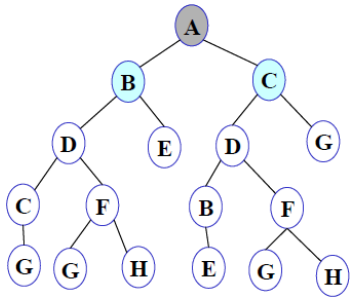


Figure 2

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.

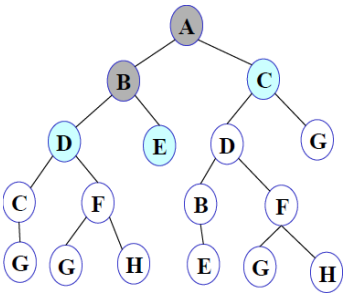


Figure 3

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.

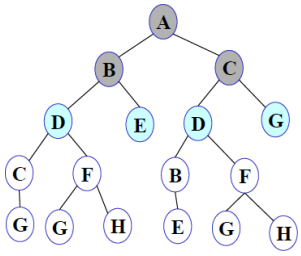


Figure 4

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.

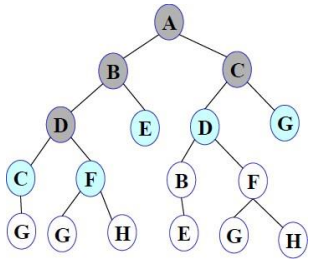


Figure 5

FRINGE: E D G C F

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

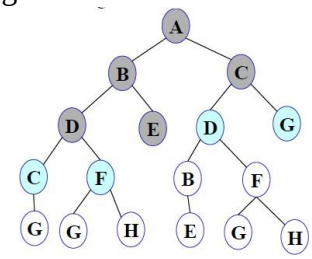


Figure 6

FRINGE: D G C F

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

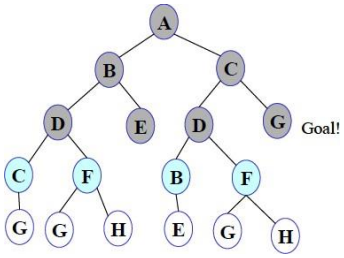


Figure 7

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.

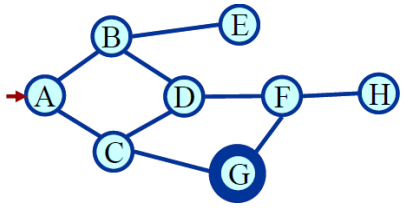
Depth- First- Search.

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.

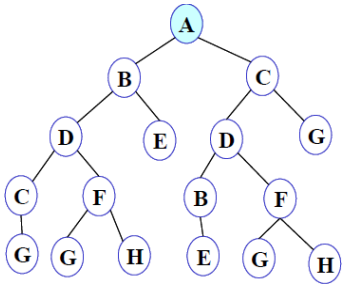


Figure 1

FRINGE: A

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

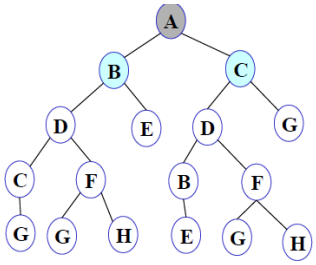


Figure 2

FRINGE: B C

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

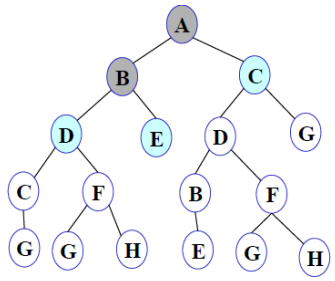


Figure 3

FRINGE: D E C

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

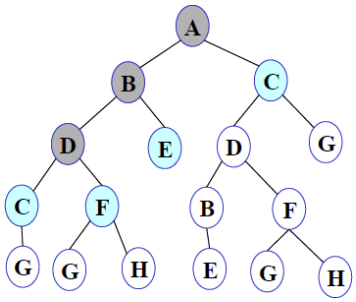


Figure 4

FRINGE: C F E C

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

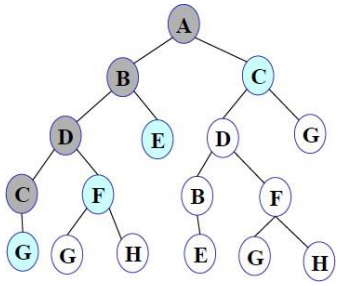


Figure 5

FRINGE: G F E C

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

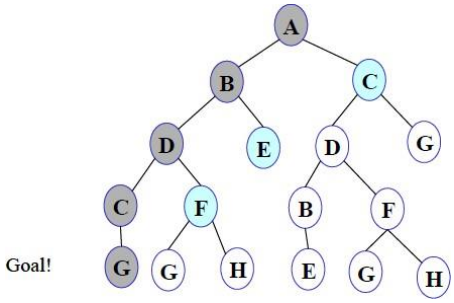


Figure 6

FRINGE: *G* F E C

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

Depth first searchis:

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:

$$f(n) = h(n).$$

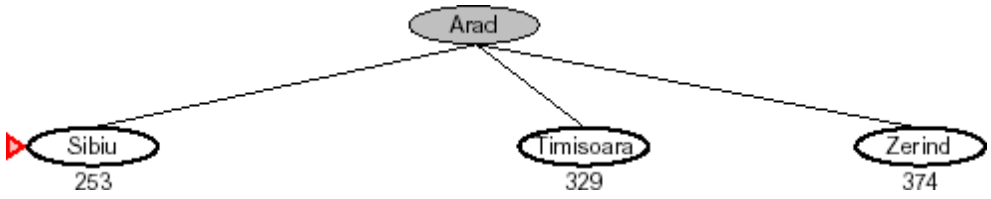
Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in **Figure 8.1**. For example, the initial state is In (Arad), and the straight line distance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

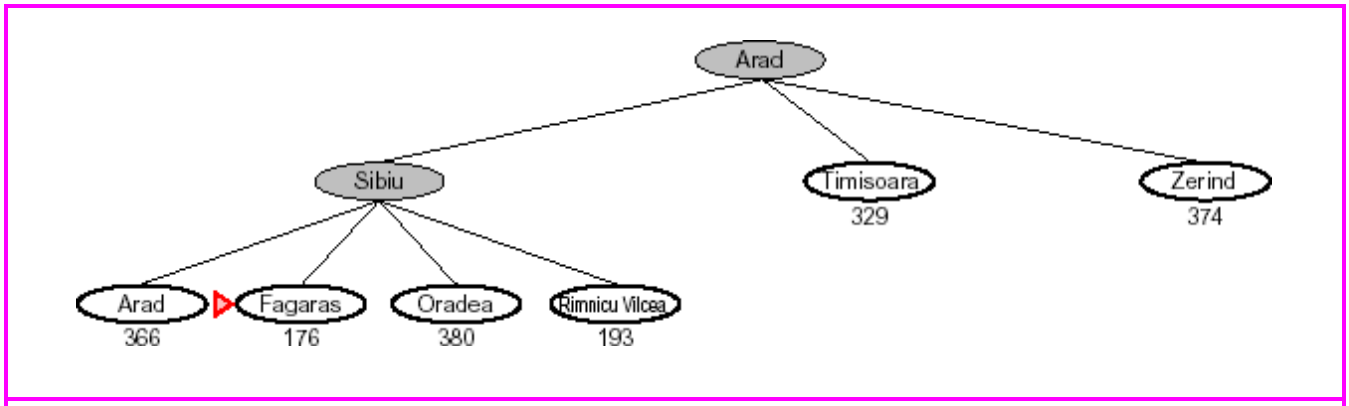
Figure 8.1: Values of h_{SLD}-straight-line distances to B u c h a r e s t.

The Initial State

After Expanding Arad



After Expanding Sibiu



After Expanding Fagaras

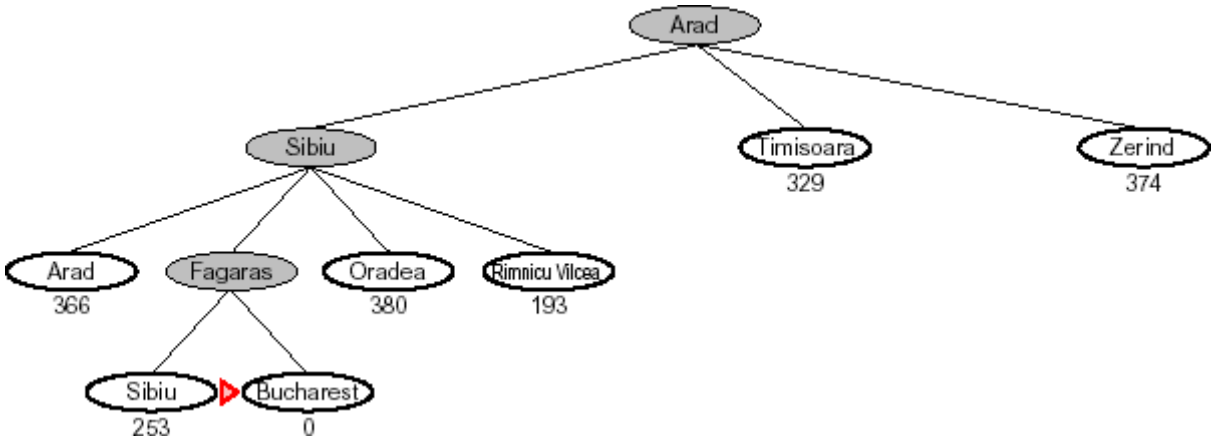


Figure 8.2: Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

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:

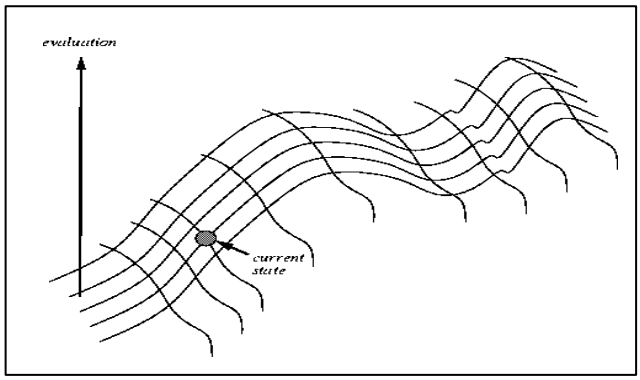
Hill Climbing Algorithm

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

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←l **to** ∞ **do**
T ←*schedule[t]*
if *T* = 0 **then return** *current*
next ←a randomly selected successor of *current*
Δ E←***VALUE****[next]* – ***VALUE****[current]*
if Δ E> 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.

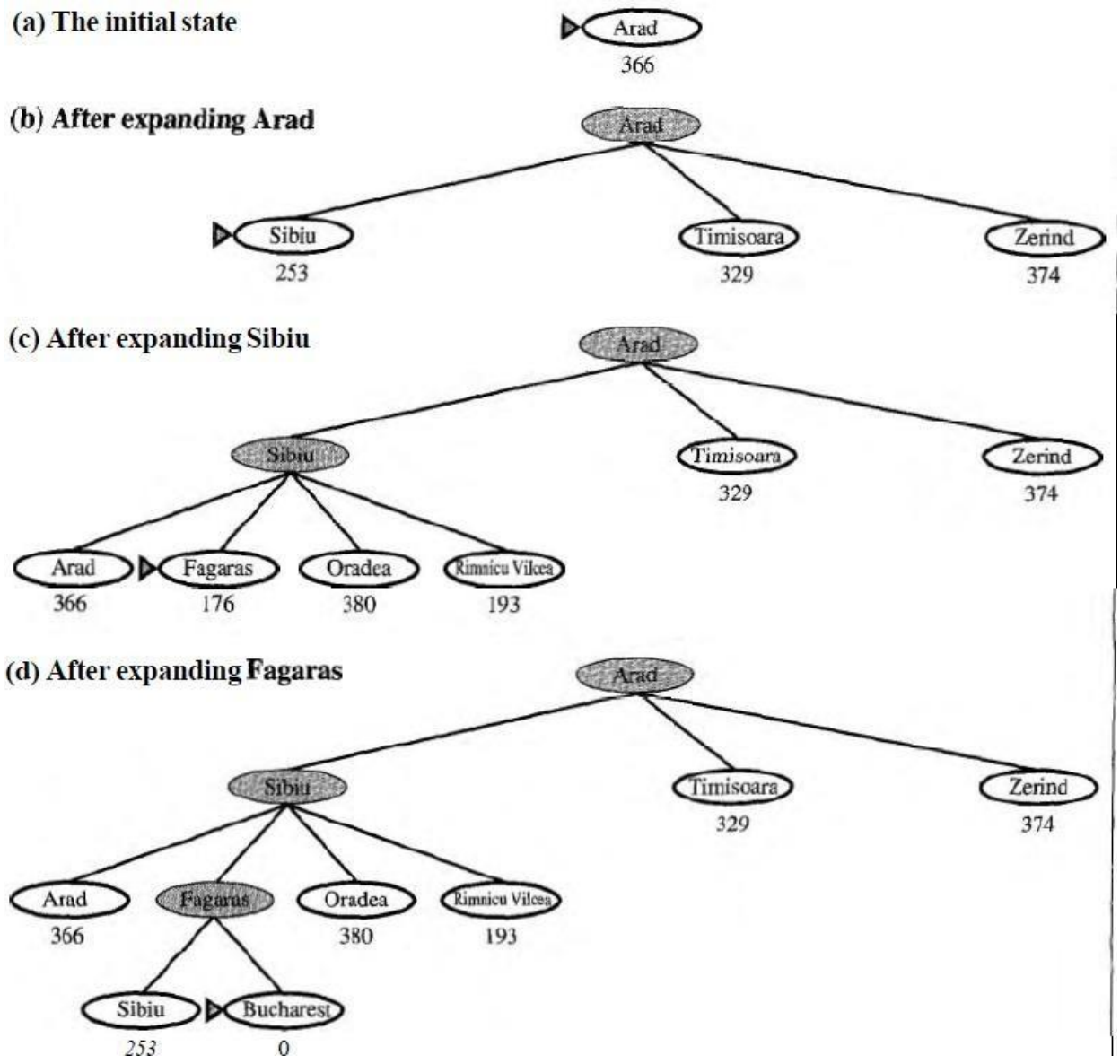
Algorithm:

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

his 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

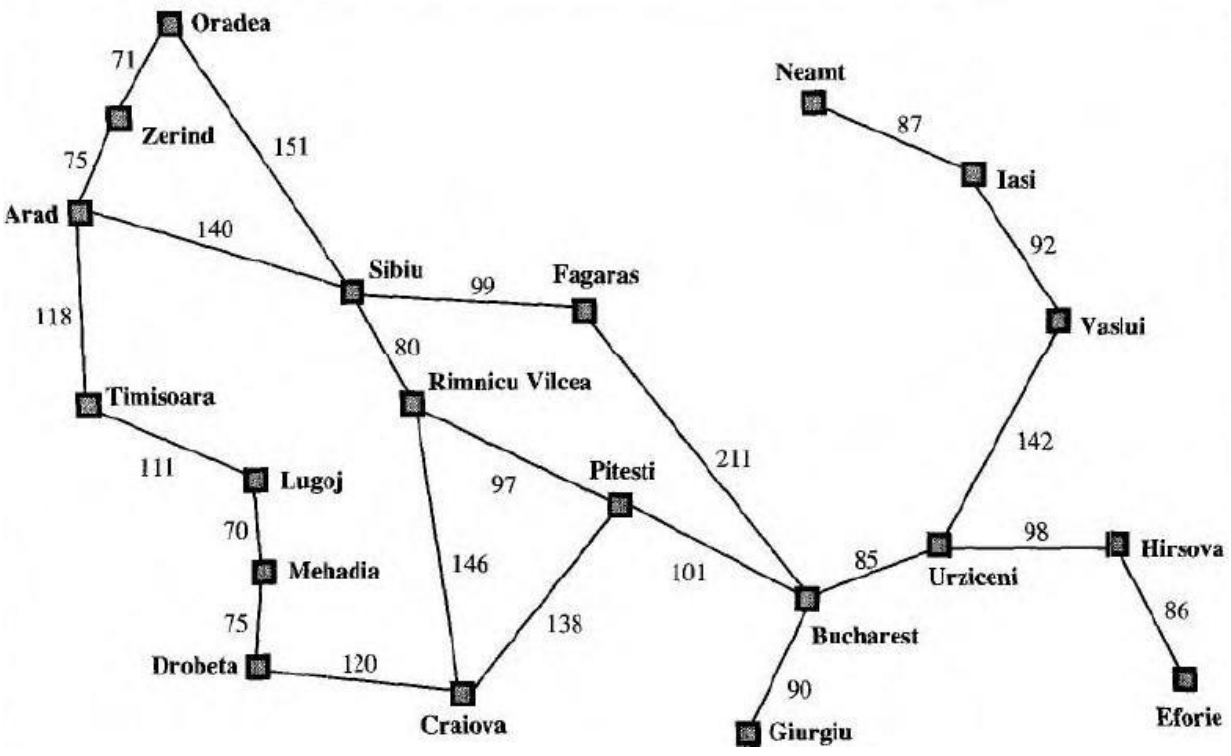
Move m to OPEN.

Description:

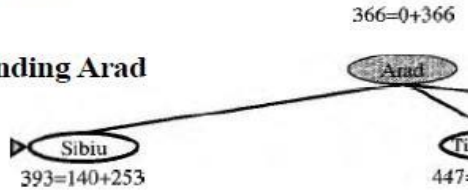
- A* begins at a selected node. Applied to this node is the "cost" of entering this node (usually zero for the initial node). A* then estimates the distance to the goal node from the current node. This estimate and the cost added together are the heuristic which is assigned to the path leading to this 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

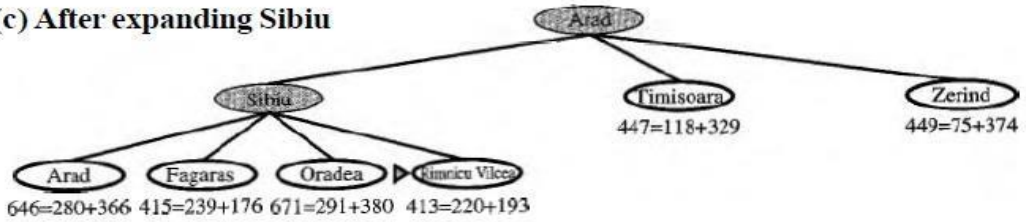


(a) The initial state

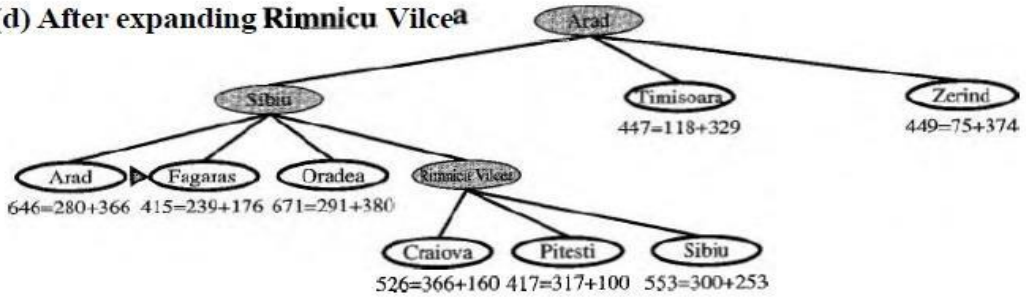


(b) After expanding Arad

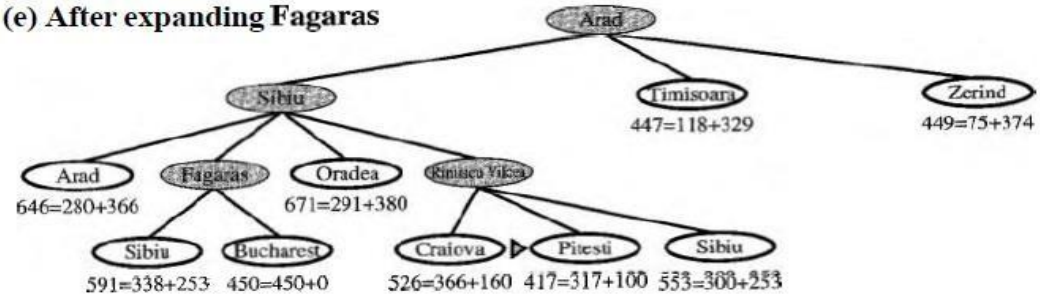
(c) After expanding Sibiu



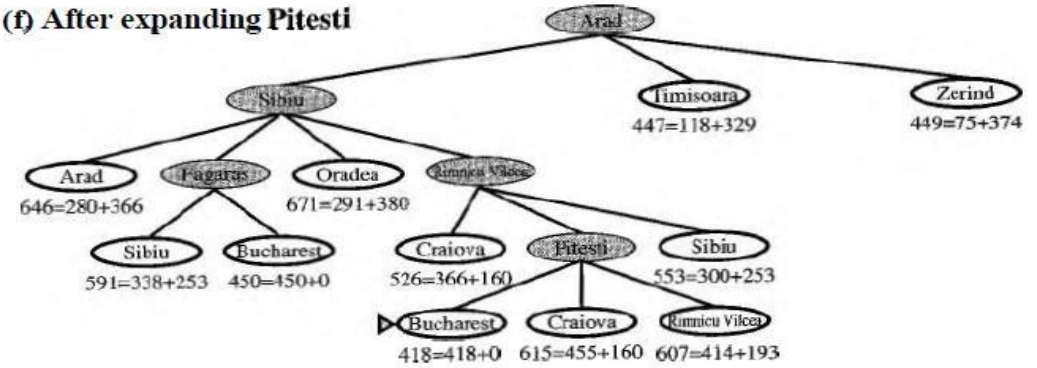
(d) After expanding Rimnicu Vilcea



(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.

Iterative Deeping A* Algorithm:

Algorithm:

Let L be the list of visited but not expanded node, and

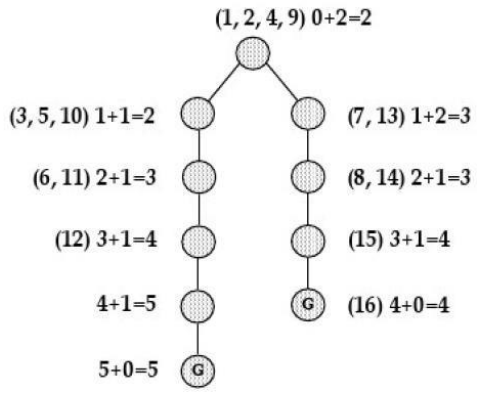
C the maximum depth

1) Let C=0
2) Initialize Lto the initial state (only)
3) If List empty increase C and goto 2),

   else

   extract a node n from the front of L
4) If n is a goal node,

   SUCCEED and return the path from the initial state to n
5) Remove n from L. If the level is smaller than C, insert at the front of L all the children n' of n with f(n') ≤ C
6) Goto 3)



- IDA* is complete & optimal Space usage is linear in the depth of solution. Each iteration is depth first search, and thus it does not require a priority queue.
- Iterative deepening A* (IDA*) eliminates the memory constraints of A* search algorithm without sacrificing solution optimality.
- Each iteration of the algorithm is a depth-first search that keeps track of the cost, f(n) = g(n) + h(n), of each node generated.
- As soon as a node is generated whose cost exceeds a threshold for that iteration, its path is cut off, and the search backtracks before continuing.
- The cost threshold is initialized to the heuristic estimate of the initial state, and in each successive iteration is increased to the total cost of the lowest-cost node that was pruned during the previous iteration.
- The algorithm terminates when a goal state is reached whose total cost dees not exceed the current threshold.

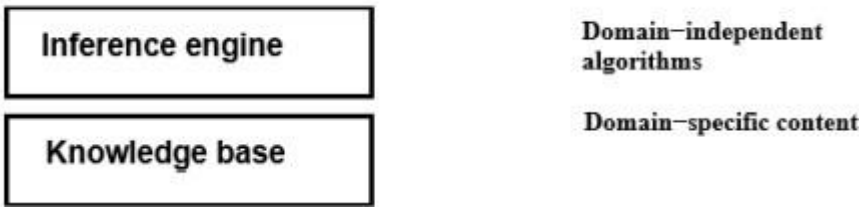
UNIT-III

PROPOSITIONAL LOGIC

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 to know – Then it can Ask itself what to do— answers should follow from the KB Agents can be viewed at the knowledge level i.e., what they know, regardless of how implemented or at the implementation level i.e., data structures in KB and algorithms that manipulate them.

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.

The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only redeeming feature of this bleak environment is the possibility of

finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Section 2.3, by the PEAS description: • Performance measure: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken, and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.

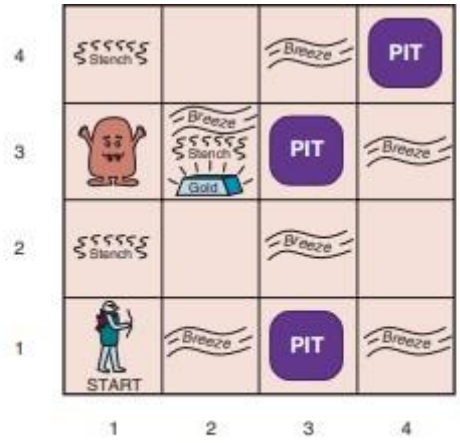


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing east (rightward).

• Environment: A 4×4 grid of rooms, with walls surrounding the grid. The agent always starts in the square labeled [1,1], facing to the east. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

• Actuators: The agent can move Forward, TurnLeft by 90◦ , or TurnRight by 90◦ . The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action Grab can be used to pick up the gold if it is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first Shoot action has any effect. Finally, the action Climb can be used to climb out of the cave, but only from square [1,1].

• Sensors: The agent has five sensors, each of which gives a single bit of information: – In the squares directly (not diagonally) adjacent to the wumpus, the agent will perceive a Stench. 1 – In the squares directly adjacent to a pit, the agent will perceive a Breeze. – In the square where the gold is, the agent will perceive a Glitter. – When an agent walks into a wall, it will perceive a Bump. – When the wumpus is killed, it emits a woeful Scream that can be perceived anywhere in the cave. The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [Stench,Breeze,None,None,None].

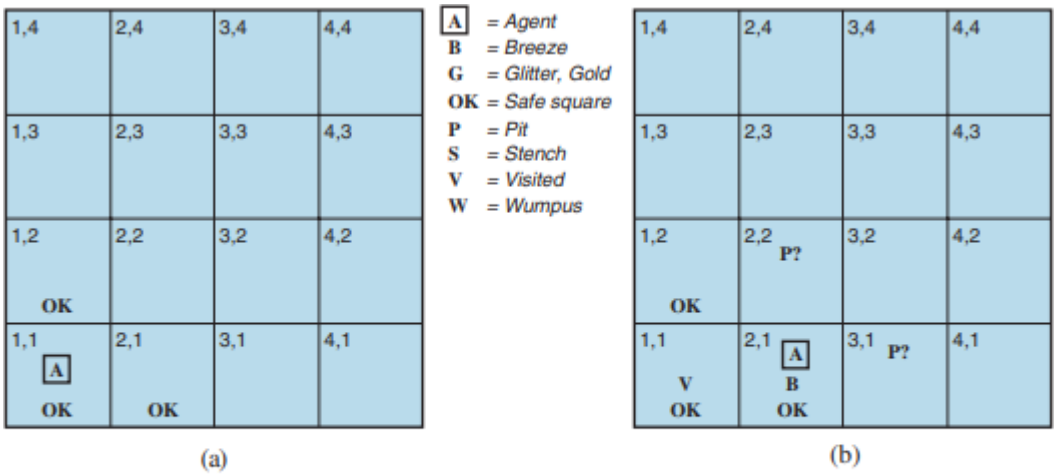


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After moving to [2,1] and perceiving [*None*, *Breeze*, *None*, *None*, *None*].

We can characterize the wumpus environment along the various dimensions given in Chapter 2. Clearly, it is deterministic, discrete, static, and single-agent. (The wumpus doesn't move, fortunately.) It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable: the agent's location, the wumpus's state of health, and the availability of an arrow. As for the locations of the pits and the wumpus: we could treat them as unobserved parts of the state— in which case, the transition model for the environment is completely known, and finding the locations of pits completes the agent's knowledge of the state. Alternatively, we could say that the transition model itself is unknown because the agent doesn't know which Forward actions are fatal—in which case, discovering the locations of pits and wumpus completes the agent's knowledge of the transition model. For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment;

overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits. Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 7.2. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 7.3 and 7.4). The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an —A‖ and —OK,‖ respectively, in square [1,1]. The first percept is [None,None,None,None,None], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 7.3(a) shows the agent's state of knowledge at this point.

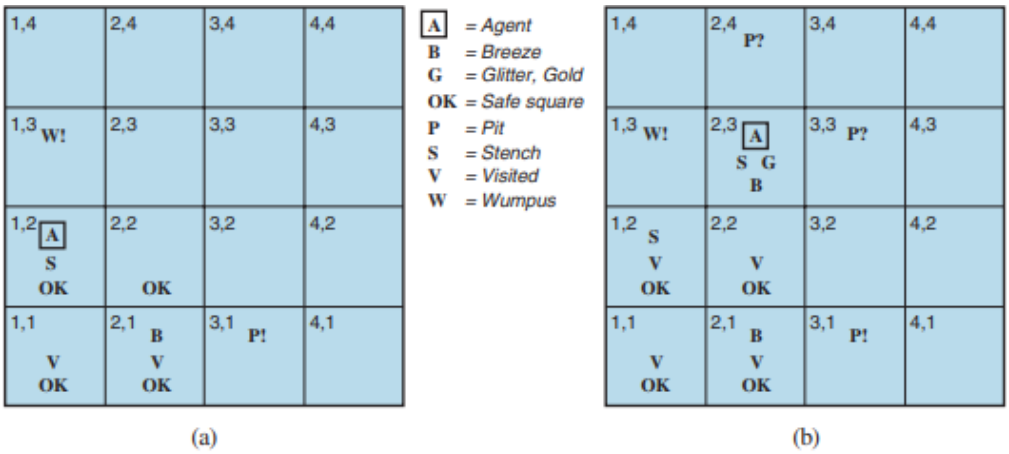


Figure 7.4 Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench*, *None*, *None*, *None*, *None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by —B‖) in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation —P?‖ in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2]. The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules

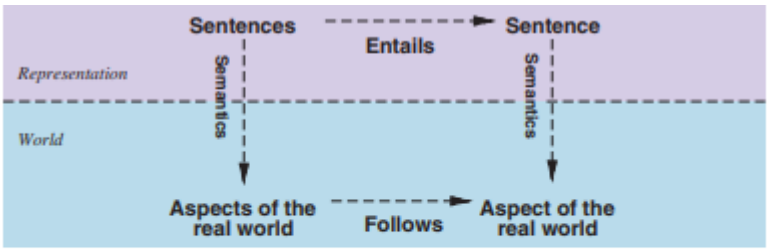
of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step. The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home. Note that in each case for which the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct. This is a fundamental property of logical reasoning. In the rest of this chapter, we describe how to build logical agents that can represent information and draw conclusions .

Logic:

A logic must also define the semantics, or meaning, of sentences. The semantics defines Truth the truth of each sentence with respect to each possible world. For example, the semantics Possible world for arithmetic specifies that the sentence ―x+y=4‖ is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no ―in between.‖2 Model When we need to be precise, we use the term model in place of ―possible world.‖ Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which has a fixed truth value (true or false) for every relevant sentence. Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence x+y=4 is true when there are four people in total. Formally, the possible models are just all possible assignments of nonnegative integers to the variables x and y. Each such assignment determines the truth of any sentence of arithmetic whose variables are x and y. If Satisfaction a sentence α is true in model m, we say that m satisfies α or sometimes m is a model of α. We use the notation M(α) to mean the set of all models of α. Now that we have a notion of truth, we are ready to talk about logical reasoning. This Entailment involves the relation of logical

entailment between sentences—the idea that a sentence follows logically from another sentence. In mathematical notation, we write α |= β to mean that the sentence α entails the sentence β. The formal definition of entailment is this: α |= β if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

α |= β if and only if M(α) ⊆ M(β)



gure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of nstructing new physical configurations from old ones. Logical reasoning should ensure that : new configurations represent aspects of the world that actually follow from the aspects at the old configurations represent.

Propositional Logic: A Very Simple Logic

Syntax :

The syntax of propositional logic defines the allowable sentences. The atomic sentences Atomic sentences consist of a single proposition symbol. Each such symbol stands for a proposition that can Proposition symbol be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, W1,3 and Facing East. The names are arbitrary but are often chosen to have some mnemonic value—we use W1,3 to stand for the proposition that the wumpus is in [1,3]. (Remember that symbols such as W1,3 are atomic, i.e., W, 1, and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: True is the always-true proposition and False is the always-false proposition. Complex sentences are constructed from simpler sentences, using parentheses and operators Complex sentences called logical connectives. There are five connectives in common use: Logical connectives ¬ (not). A sentence such as ¬W1,3 is called the negation of W1,3. A literal is either an Negation atomic sentence (a positive literal) or a negated atomic sentence (a negative literal). Literal ∧ (and). A sentence whose main connective is ∧, such as W1,3 ∧P3,1, is called a conjunction; its parts are the conjuncts. (The ∧ looks like an ―A‖ for ―And.‖) Conjunction ∨ (or). A

sentence whose main connective is ∨, such as (W1,3 ∧P3,1)∨W2,2, is a disjunction; its parts are disjuncts—in this example, (W1,3 ∧P3,1) and W2,2. Disjunction ⇒ (implies). A sentence such as (W1,3 ∧P3,1) ⇒ ¬W2,2 is called an implication (or con- Implication ditional). Its premise or antecedent is (W1,3 ∧P3,1), and its conclusion or consequent Premise is ¬W2,2. Implications are also known as rules or if–then statements. The implication Conclusion symbol is sometimes written in other books as ⊃ or →. Rules ⇔ (if and only if). The sentence W1,3 ⇔ ¬W2,2 is a biconditional.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Semantics :

The semantics defines the rules for determining the truth of a sentence with respect to a particular Truth value model. In propositional logic, a model simply sets the truth value—true or false—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols P1,2, P2,2, and P3,1, then one possible model is m1 = {P1,2 =false, P2,2 =false, P3,1 =true}. With three proposition symbols, there are 23 =8 possible models.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

Propositional Theorem Proving

In this section, we show how entailment Theorem proving can be done by theorem proving—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking. Before we plunge into the details of theorem-proving algorithms, we will need some Logical equivalence additional concepts related to entailment. The first concept is logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. (Note that ≡ is used to make claims about sentences, while ⇔ is used as part of a sentence.) For example, we can easily show (using truth tables) that P∧Q and Q∧P are logically equivalent; other equivalences are shown in Figure 7.11. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent if and only if each of them entails the other: $\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$. Validity The second concept we will need is validity. A sentence is valid if it is true in all models. For Tautology example, the sentence P∨ ¬P is valid. Valid sentences are also known as tautologies—they are necessarily true. Because the sentence True is true in all models, every valid sentence is logically equivalent to True. What good are valid sentences? From our definition of entail Deduction theorem, we can derive the deduction theorem, which was known to the ancient Greeks: I For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid. (Exercise 7.DEDU asks for a proof.) Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 7.10 does—or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to True. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference. Satisfiability The final concept we will need is satisfiability. A sentence is satisfiable if it is true in, or satisfied by, some model. For example, the knowledge base given earlier, (R1 ∧R2 ∧ R3 ∧ R4 ∧ R5), is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences,

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

in propositional logic—the SAT problem—was the first problem proved to be NP-complete. SAT Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 5 ask whether the constraints are satisfiable by some assignment. Validity and satisfiability are of course connected: α is valid iff ¬α is unsatisfiable; contrapositively, α is satisfiable iff ¬α is not valid. We also have the following useful result: α |= β if and only if the sentence (α∧ ¬β) is unsatisfiable. J Proving β from α by checking the unsatisfiability of (α ∧ ¬β) corresponds exactly to the standard mathematical proof technique of reductio ad absurdum (literally, ‖reduction to an absurd thing‖). It is also called proof by refutation or proof by contradiction. One assumes a Refutation sentence β to be false and shows that this leads to a contradiction with known axioms α. This Contradiction contradiction is exactly what is meant by saying that the sentence (α∧ ¬β) is unsatisfiable.

Inference and proofs:

This section covers inference rules that can be applied to derive a proof—a chain of conclu- Inference rules sions that leads to the desired goal. The best-known rule is called Modus Ponens (Latin for Proof mode that affirms) and is written

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the sentence β can be inferred. For example, if (WumpusAhead ∧ WumpusAlive) ⇒ Shoot and (WumpusAhead ∧ WumpusAlive) are given, then

Shoot can be inferred. Another useful inference rule is And-Elimination, which says that, from a conjunction, And-Elimination any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

For example, from (WumpusAhead∧WumpusAlive), WumpusAlive can be inferred.

By considering the possible truth values of α and β, one can easily show once and for all that Modus Ponens and And-Elimination are sound. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models. All of the logical equivalences in Figure 7.11 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain α ⇒ β and α from β. Let us see how these inference rules and equivalences can be used in the wumpus world. We start with the knowledge base containing R1 through R5 and show how to prove ¬P1,2, that is, there is no pit in [1,2]:

1. Apply biconditional elimination to R2 to obtain R6 : (B1,1 ⇒ (P1,2 ∨P2,1)) ∧ ((P1,2 ∨P2,1) ⇒ B1,1). 2. Apply And-Elimination to R6 to obtain R7 : ((P1,2 ∨P2,1) ⇒ B1,1).

3. Logical equivalence for contrapositives gives R8 : (¬B1,1 ⇒ ¬(P1,2 ∨P2,1)).

4. Apply Modus Ponens with R8 and the percept R4 (i.e., ¬B1,1), to obtain R9 : ¬(P1,2 ∨P2,1). 5. Apply De Morgan's rule, giving the conclusion R10 : ¬P1,2 ∧ ¬P2,1 . That is, neither [1,2] nor [2,1] contains a pit. Any of the search algorithms in Chapter 3 can be used to find a sequence of steps that constitutes a proof like this. We just need to define a proof problem as follows:

• INITIAL STATE: the initial knowledge base.

• ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.

- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove Thus, searching for proofs is an alternative to enumerating models. In many practical cases I finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are.

Proof by resolution

We have argued that the inference rules covered so far are sound, but we have not discussed the question of completeness for the inference algorithms that use them.
For example, if we removed the biconditional elimination rule, the proof in the preceding section would not go through. The current section introduces a single inference rule, resolution, that yields a complete inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the wumpus world. Let us consider the steps leading up to Figure 7.4(a): the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

$R_{11}: \quad \neg B_{1,2}$.
$R_{12}: \quad B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$.

By the same process that led to R_{10} earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pitless):

$R_{13}: \quad \neg P_{2,2}$.
$R_{14}: \quad \neg P_{1,3}$.

We can also apply biconditional elimination to R_3, followed by Modus Ponens with R_5, to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

$R_{15}: \quad P_{1,1} \vee P_{2,2} \vee P_{3,1}$.

Now comes the first application of the resolution rule: the literal $\neg P_{2,2}$ in R_{13} *resolves with* the literal $P_{2,2}$ in R_{15} to give the **resolvent**

$R_{16}: \quad P_{1,1} \vee P_{3,1}$.

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal $\neg P_{1,1}$ in R_1 resolves with the literal $P_{1,1}$ in R_{16} to give

$R_{17}: \quad P_{3,1}$.

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the **unit resolution** inference rule

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  while true do
      for each pair of clauses C_i, C_j in clauses do
          resolvents ← PL-RESOLVE(C_i, C_j)
          if resolvents contains the empty clause then return true
          new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

Figure 7.13 A simple resolution algorithm for propositional logic. PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

Horn clauses and definite clause:

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm. One such restricted form is the definite clause, which is a disjunction of literals of which Definite clause exactly one is positive. For example, the clause (¬L1,1 ∨ ¬BreezeVB1,1) is a definite clause, whereas (¬B1,1 ∨P1,2 ∨P2,1) is not, because it has two positive clauses. Slightly more general is the Horn clause, which is a disjunction of literals of which at Horn clause 248 Chapter 7 Logical Agents most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive Goal clauses literals; these are called goal clauses. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause. One more class is the k-CNF sentence, which is a CNF sentence where each clause has at most k literals. Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.DISJ.) For example, the definite clause (¬L1,1 ∨¬BreezeVB1,1) can be written as the implication (L1,1∧Breeze) ⇒ B1,1. In the implication form, the sentence is easier to understand: it

says that if the agent is in [1,1] and there is a breeze percept, then [1,1] is breezy. In Body Horn form, the premise is called the body and the conclusion is called the head. A Head sentence consisting of a single positive literal, such as L1,1, is called a fact. It too can Fact be written in implication form as True ⇒ L1,1, but it is simpler to write just L1,1. Forward-chaining

2. Inference with Horn clauses can be done through the forward-chaining and backwardBackward-chaining chaining algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for logic programming.

3. Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base—a pleasant surprise.

Forward and backward chaining:

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if L1,1 and Breeze are known and (L1,1 ∧Breeze) ⇒ B1,1 is in the knowledge base, then B1,1 can be added. This process continues until the query q is added or until no further inferences can be made. The algorithm is shown in Figure 7.15; the main point to remember is that it runs in linear time. The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.16(b) shows the same knowledge base drawn as an AND–OR graph (see Chapter 4). In AND–OR graphs, multiple edges joined by an arc indicate a conjunction—every edge must be proved—while multiple edges without an arc indicate a disjunction—any edge can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail. It is easy to see that forward chaining is sound: every inference is essentially an application of Modus Ponens. Forward chaining is also

complete: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the inferred table (after the algorithm reaches a fixed point where no new inferences are possible). The table contains true for each symbol inferred during the process, and false for all other symbols. We I can view the table as a logical model; moreover, every definite clause in the original KB is true in this model.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
    p ← POP(queue)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to queue
  return false
```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *queue* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are not yet proven. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

Effective Propositional Model Checking

we describe two families of efficient algorithms for general propositional inference based on model checking: one approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the ―technology‖ of propositional logic. This section can be skimmed on a first reading of the chapter. The algorithms we describe are for checking satisfiability: the SAT problem. (As noted in Section 7.5, testing entailment, α |= β, can be done by testing unsatisfiability of α∧ ¬β.) We

mentioned that the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of propositional satisfiability algorithms closely resemble the backtracking algorithms of Section 5.3 and the local search algorithms of Section 5.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

A complete backtracking algorithm

The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962).

It embodies three improvements,

• Early termination: The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if any literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete. For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C. Similarly, a sentence is false if any clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space.

• Pure symbol heuristic: A pure symbol is a symbol that always appears with the same Pure symbol ―sign‖ in all clauses. For example, in the three clauses$(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains B=false, then the clause $(\neg B \vee \neg C)$ is

already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

• Unit clause heuristic: A unit clause was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model. For example, if the model contains B=true, then (¬B∨ ¬C)simplifies to ¬C, which is a unit clause. Obviously, for this clause to be true,C must be set to false. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.KNOW). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to false, (C∨A) becomes a unit clause, causing true to be assigned to A. This ―cascade‖ of forced assignments is called unit propagation.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
         DPLL(clauses, rest, model ∪ {P=false})
```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

1. Component analysis (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called components, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.

2. Variable and value ordering (as seen in Section 5.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and

always tries the value true before false. The degree heuristic (see page 177) suggests choosing the variable that appears most frequently over all remaining clauses.

3. Intelligent backtracking : Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of conflict clause learning to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.

4. Random restarts : Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.

5. Clever indexing (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things.

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of value flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for each i = 1 to max_flips do
     if model satisfies clauses then return model
     clause ← a randomly selected clause from clauses that is false in model
     if RANDOM(0, 1) ≤ p then
        flip the value in model of a randomly selected symbol from clause
     else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

Agents Based on Propositional Logic:

The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history. This requires writing down a complete logical model of the effects of actions. We then show how logical inference can be used by an agent in the wumpus world. We also show how the agent can keep track of the world efficiently without going back into the percept history for each inference. Finally, we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals, provided its knowledge base is true in the actual world.

The current state of the world:

The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent's experience in a particular world. In this section, we focus on the problem

of deducing the current state of the wumpus world—where am I, is that square safe, and so on.

The agent knows that the starting square contains no pit (¬P1,1) and no wumpus (¬W1,1). Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus. Thus, we include a large collection of sentences of the following form: B1,1 ⇔ (P1,2 ∨P2,1) S1,1 ⇔ (W1,2 ∨W2,1) ⋯ The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, we have to say that there is at least one wumpus: W1,1 ∨W1,2 ∨⋯ ∨W4,3 ∨W4,4 . Then we have to say that there is at most one wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free: ¬W1,1 ∨ ¬W1,2 ¬W1,1 ∨ ¬W1,3 ⋯ ¬W4,3 ∨ ¬W4,4 .

Then we have to say that there is at most one wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free: ¬W1,1 ∨ ¬W1,2 ¬W1,1 ∨ ¬W1,3 ⋯ ¬W4,3 ∨ ¬W4,4 .

A hybrid agent:

The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the atemporal axioms—those that don't depend on t, such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t, such as the successor-state axioms. (The next section explains why the agent doesn't need axioms for future time steps.) Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited. The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares. Route planning is done with A∗ search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations.

Logical state estimation

The agent program works quite well, but it has one major weakness: as time goes by, the computational expense involved in the calls to ASK goes up and up. This happens mainly because the required inferences have to go back further and further in time and involve more and more proposition symbols. Obviously, this is unsustainable—we cannot have an agent whose time to process each percept grows in proportion to the length of its life! What we really need is a constant update time—that is, independent of t. The obvious answer is to save, or cache, the results of inference, so that the inference process at the next time step can Caching build on the results of earlier steps instead of having to start again from scratch.

```
function HYBRID-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter,bump,scream]
  persistent: KB, a knowledge base, initially the atemporal "wumpus physics"
              t, a counter, initially 0, indicating time
              plan, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal "physics" sentences for time t
  safe ← {[x,y] : ASK(KB, OK^t_{x,y}) = true}
  if ASK(KB, Glitter^t) = true then
    plan ← [Grab] + PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
  if plan is empty then
    unvisited ← {[x,y] : ASK(KB, L^{t'}_{x,y}) = false for all t' ≤ t}
    plan ← PLAN-ROUTE(current, unvisited ∩ safe, safe)
  if plan is empty and ASK(KB, HaveArrow^t) = true then
    possible_wumpus ← {[x,y] : ASK(KB, ¬ W_{x,y}) = false}
    plan ← PLAN-SHOT(current, possible_wumpus, safe)
  if plan is empty then        // no choice but to take a risk
    not_unsafe ← {[x,y] : ASK(KB, ¬ OK^t_{x,y}) = false}
    plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
  if plan is empty then
    plan ← PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
  action ← POP(plan)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action

function PLAN-ROUTE(current,goals,allowed) returns an action sequence
  inputs: current, the agent's current position
          goals, a set of squares; try to plan a route to one of them
          allowed, a set of squares that can form part of the route

  problem ← ROUTE-PROBLEM(current, goals,allowed)
  return SEARCH(problem)       // Any search algorithm from Chapter 3
```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to choose actions. Each time HYBRID-WUMPUS-AGENT is called, it adds the percept to the knowledge base, and then either relies on a previously-defined plan or creates a new plan, and pops off the first step of the plan as the action to do next.

Pros and Cons of Pro. Logic

- Propositional logic is declarative
- Propositional logic allows partial/disjunctive/negated information
  - unlike most data structures and databases, which are domain-specific
- Propositional logic is compositional
  - meaning of $B_{1,1} \wedge P_{1,2}$ is derived from meaning of $B_{1,1}$ and of $P_{1,2}$
- Meaning in propositional logic is context-independent
  - (unlike natural language, where meaning depends on context)
- Propositional logic has very limited expressive power
  - (unlike natural language)
  - E.g., cannot say "pits cause breezes in adjacent squares"
    - except by writing one sentence for each square

Our Approach

- Adopt the foundation of propositional logic – a declarative, compositional semantics that is context-independent and unambiguous – but with more expressive power, borrowing ideas from natural language while avoiding its drawbacks

- Important elements in natural language:
  - Objects (squares, pits, wumpuses)
  - Relations among objects (is adjacent to, is bigger than) or *unary relations* or properties (is red, round)
  - Functions (father of, best friend of)
  - **First-order logic (FOL)** is built around the above 3 elements

Identify Objects, Relations, Functions

- One plus two equals three.
- Squares neighboring the wumpus are smelly.
- Evil King John ruled England in 1200.

Prop. Logic vs. FOL

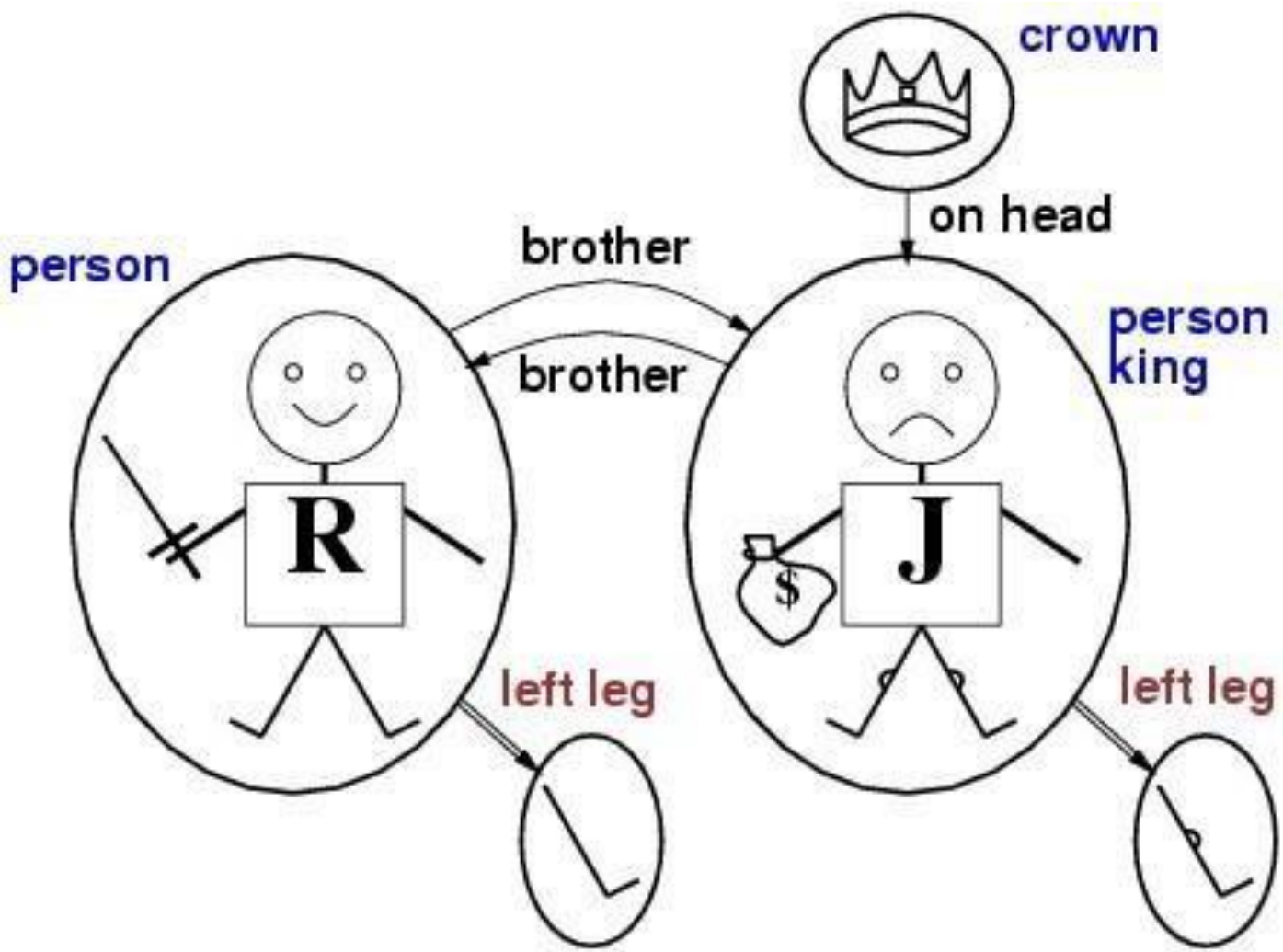
- Propositional logic assumes that there are facts that either hold or do not hold
- FOL assumes more: the world consists of objects with certain relations among them that do or do not hold
- Other special-purpose logics:
  - Temporal logic: facts hold at particular times / T,F,U
    - E.g., "I am always hungry", "I will eventually be hungry"
  - Higher-order logic: views the relations and functions in FOL as objects themselves / T,F,U
  - Probability theory: facts / degree of belief [0, 1]
  - Fuzzy logic: facts with degree of truth [0, 1] / known interval values
    - E.g, "the temperature is very hot, hot, normal, cold, very cold"

First-Order Logic (FOL)

- Whereas propositional logic assumes the world contains facts

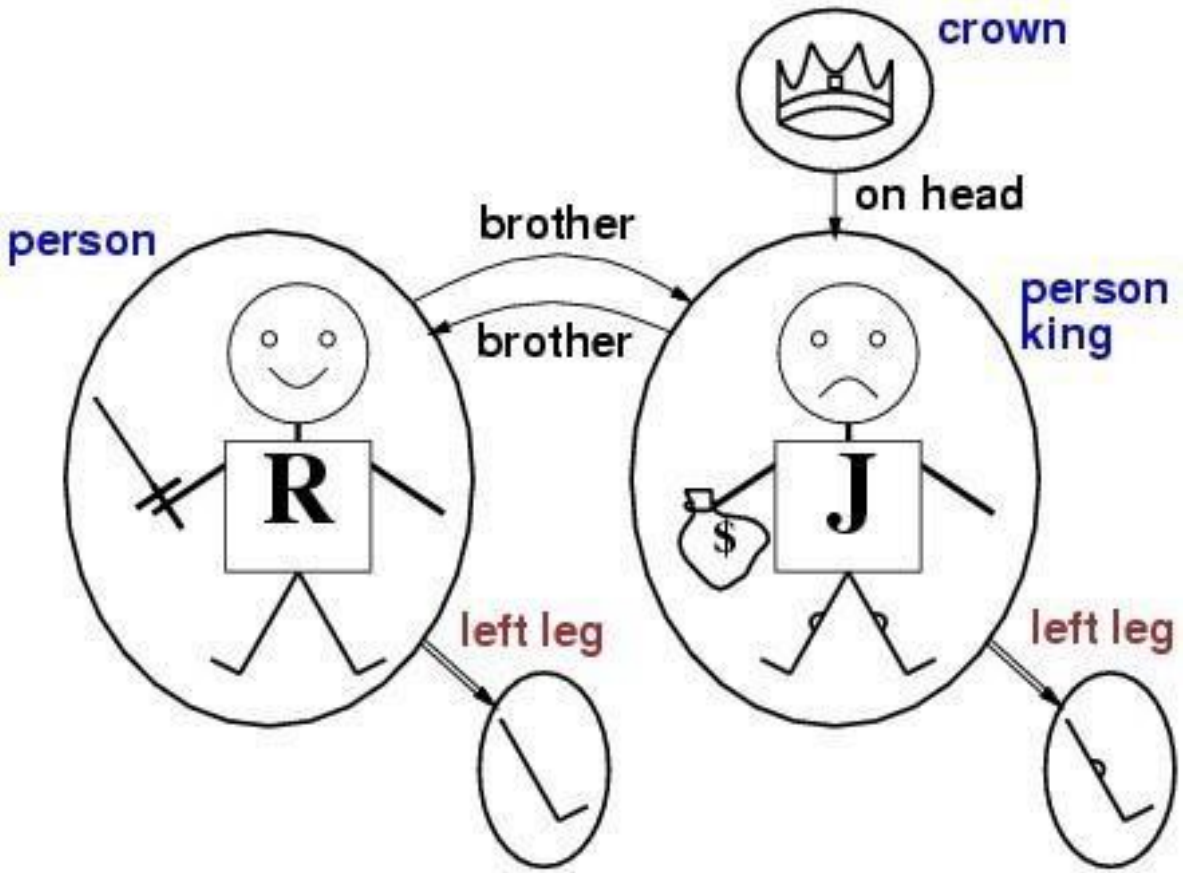
- First-order logic (like natural language) assumes the world contains
  - Objects: people, houses, numbers, colors, baseball games, wars, …
  - Relations: red, round, prime, brother of, bigger than, part of, comes between, …
  - Functions: father of, best friend, one more than, plus, …

Models for FOL: Example



Example

- Five objects
- Two binary relations
- Three unary relations
- One unary function



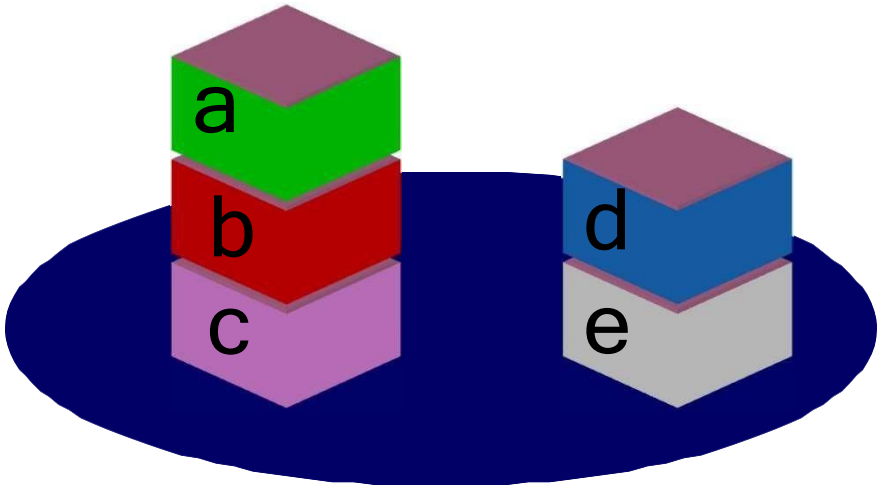
Syntax of FOL: Basic Elements

- Basic symbols: objects (constant symbols), relations (predicate symbols), and functions (functional symbols).
- Constants — King John, 2, Wumpus...
- Predicates — Brother, >,...
- Functions — Sqrt, LeftLegOf,...
- Variables — x, y, a, b,...
- Connectives — □, □, □, □, □
- Equality — =
- Quantifiers — □, □
- A legitimate expression of predicate calculus is called well-formed formula (wff), or simply, sentence

Relations (Predicates)

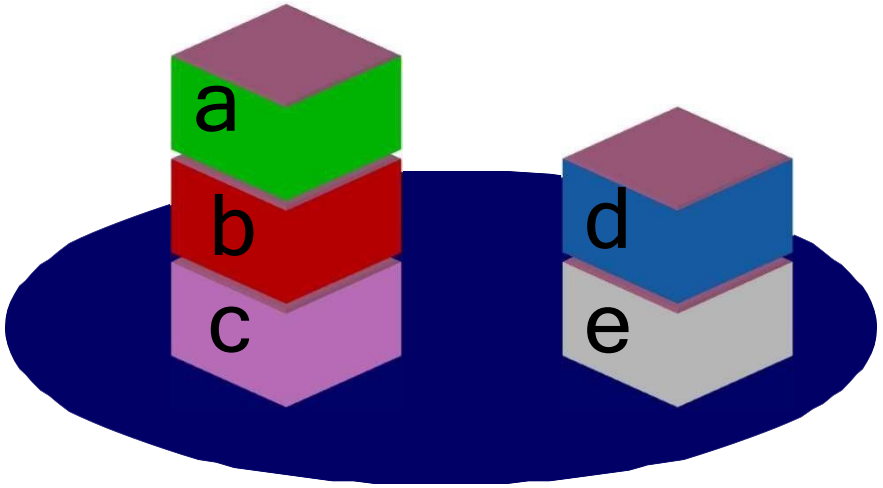
- A relation is the set of tuples of objects
  - Brotherhood relationship {<Richard, John>, <John, Richard>}
  - Unary relation, binary relation, …

- Example: set of blocks {a, b, c, d, e}
- The "On" relation includes:
  - On = {<a,b>, <b,c>, <d,e>}
  - the predicate On(A,B) can be interpreted as <a,b> ∈ On.



Functions

- In English, we use “King John’s left leg” rather than giving a name to his leg, where we use “function symbol”
- hat(c) = b
- hat(b) = a
- hat(d) is not defined



Terms and Atomic Sentences

Atomic sentence = *predicate* ($term_1$,...,$term_n$)
or $term_1 = term_2$

Term = *function* ($term_1$,...,$term_n$)
or *constant* or *variable*

- Atomic sentence states facts
- Term refers to an object
- For example:
  - *Brother*(*KingJohn*,*RichardTheLionheart*)
  - *Length*(*LeftLegOf*(*Richard*))
  - *Married*(*Father*(*Richard*), *Mother*(*John*))

Composite Sentences

- Complex sentences are made from atomic sentences using connectives
- $\neg S$, $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, $S_1 \Leftrightarrow S_2$

For example:

$$Sibling(John, Richard) \Rightarrow Sibling(Richard, John)$$

$$\neg Brother(LeftLeg(Richard), John)$$

$$King(Richard) \vee King(John)$$

$$\neg King(Richard) \Rightarrow King(John)$$

Intended Interpretation

- The semantics relate sentences to models to determine truth
- Interpretation specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols
  - *Richard* refers to Richard the Lionheart and *John* refers to the evil King John
  - *Brother* refers to the brotherhood relation; *Crown* refer to the set of objects that are crowns
  - *LeftLeg* refers to the "left leg" function
  - There are many other possible interpretations that relate symbols to model; *Richard* refers to the crown
  - If there are 5 objects in the model, how many possible interpretations are there for two symbols *Richard* and *John*?

Intended Interpretation (Con't)

- The truth of any sentence is determined by a model and an interpretation for the sentence's model
- The entailment and validity are defined in terms of all possible models and all possible interpretations
- The number of domain elements in each model may be unbounded; thus the number of possible models is unbounded
- Checking entailment by enumeration of all possible models is **NOT** doable for FOL

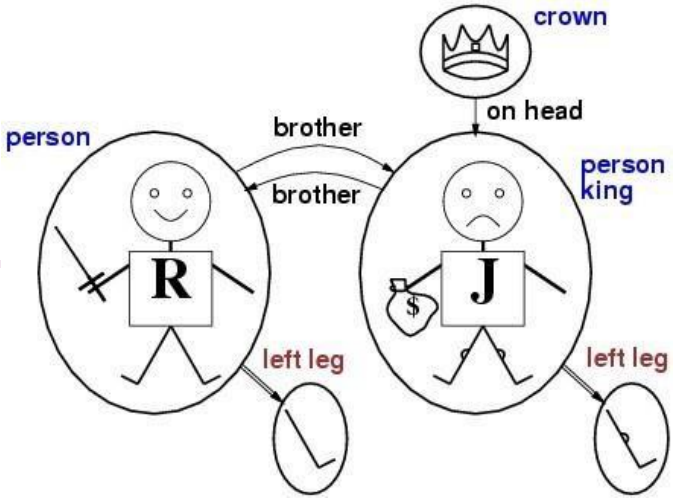
In General

- Sentences are true with respect to a model and an interpretation
- Model contains objects (domain elements) and relations among them
- Interpretation specifies referents for

constant symbols	→	objects
predicate symbols	→	relations
function symbols	→	functional relations

- Once we have a logic that allows objects, it is natural to want to express properties of entire collections of objects, instead of enumerating the objects by name Quantifiers

Universal Quantifier



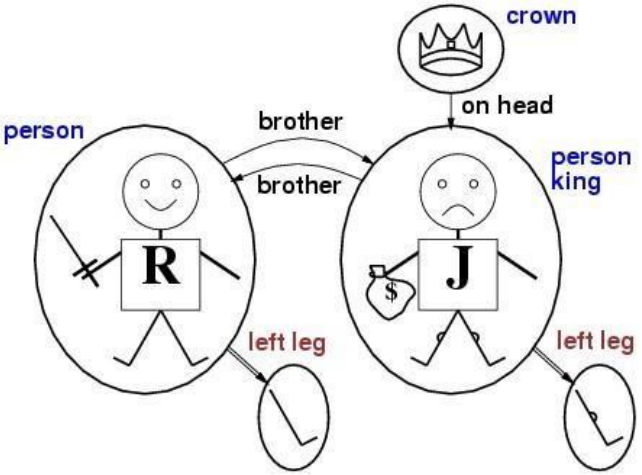
- Universal quantification (□)
  - "All kings are person": □x King(x) □ Person(x)
  - For all x, if x is a king, then x is a person

- In general, □x P is true in a given model under a given interpretation if P is true in all possible extended interpretations

- In the above example, x could be one of the following:
  - Richard, John, Richard's left leg, John's left leg, Crown
  - 5 extended interpretations

- A common mistake: □x (King(x) ^ Person(x))

Existential Quantifier



- Existential quantification (∃)
  - ∃x Crown(x) ∧ OnHead(x, John)
  - There is an x such that x is a crown and x is on the John's head
- In general, ∃x P is true in a given model under a given interpretation if P is true in at least one extended interpretation that assigns x to a domain element
- In the above example, "Crown(crown) ∧ OnHead(crown, John)" is true
- Common mistake: ∃x Crown(x) ⇒ OnHead(x, John)

Nested Quantifiers

- Nested quantifiers
  - ∀x ∀y [Brother(x, y) ⇒ Sibling(x, y)]
  - ∀x ∀y Loves(x, y)
  - ∀y ∀x Loves(x, y)
  - ∀x ∀y is not the same as ∀y ∀x
  - The order of quantification is important

- Quantifier duality: each can be expressed using the other
  - ∀x Likes(x,IceCream) ≡ ¬∃x ¬Likes(x,IceCream)
  - ∀x Likes(x,Broccoli) ≡ ¬∃x ¬Likes(x,Broccoli)

Equality

- $term_1 = term_2$ is true under a given interpretation if and only if $term_1$ and $term_2$ refer to the same object
  - Can be used to state facts about a given function
    - E.g., Father(John) = Henry
  - Can be used with negation to insist that two terms are not the same object
  - E.g., definition of *Sibling* in terms of *Parent*:
  - ∀x,y *Sibling(x,y)* ⇔ [¬(x = y) ∧ ∃m,f ¬(m = f) ∧ Parent(m,x) ∧ Parent(f,x) ∧ Parent(m,y) ∧ Parent(f,y)]

Using FOL

- Sentences are added to a KB using TELL, which is called assertions
- Questions asked using ASK are called queries
- Any query that is logically entailed by the KB should be answered affirmatively
- The standard form for an answer is a substitution or binding list, which is a set of variable/term pairs

$$TELL(KB,\ King(John))$$

$$TELL(KB,\ \exists x\ King(x) \wedge Person(x))King(John))$$

$$ASK\ \ (KB,Person(John))$$

$$ASK\,(KB,$$

$$ASK\,(KB,\ \exists x\ Person(x)) \qquad answer:\{x/\ John\}$$

Example: The Kinship Domain

- An example KB includes things like:
  - Fact:
    - "Elizabeth is the mother of Charles"
    - "Charles is the father of William"
  - Rules:
    - One's grandmother is the mother of one's parent"
- Object: people
- Unary predicate: Male, Female
- Binary predicate: Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle
- Function: Mother, Father

Example: The Kinship Domain

One's mom is one's female parent.

$\forall m, c \; Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m, c)$

One's husband is one's male spouse.

$\forall w, h \; Husband(h, w) \Leftrightarrow Male(h) \wedge Spouse(h, w)$

Paren and child are inverse relations.

$\forall p, c \; Parent(p, c) \Leftrightarrow Child(c, p)$

A grandparent is a parent of one's parent.

$\forall g, c \; Grandparent(g, c) \Leftrightarrow \exists p \; Parent(g, p) \wedge Parent(p, c)$

Practice:

Male and female are disjoint categories

A sibling is another child of one's parents

Answer

- Male and female are disjoint categories:
  - ∀x Male(x) ⇒ ¬Female(x)
- A sibling is another child of one's parent:
  - ∀x,y Sibling(x, y) ⇔ x≠y ^ ∃p Parent(p,x) ^ Parent(p, y)

The Wumpus World

- A percept is a binary predicate:
  - Percept([Stench, Breeze, Glitter, None, None], 5)
- Actions are logical terms:
  - Turn(Right), Turn(Left), Forward, Shoot, Grab, Release, Climb
- Query:
  - ?a BestAction(a, 5)
  - Answer: {a/Grab}
- Perception: percept implies facts:
  - ? t, s, b, m, c Percept([s, b, Glitter, m, c], t) ? Glitter(t)
- Reflex behavior:
  - ? t Glitter(t) ? BestAction(Grab, t)

The Wumpus World

- The Environment:
  - Objects: squares, pits and the wumpus
  - ∀x,y,a,b *Adjacent*([x,y],[a,b]) ⇔

[a,b] ∈ {[x+1,y], [x-1,y],[x,y+1],[x,y-1]}

  - A unary predicate *Pit*(x)
  - ∀s,t *At*(Agent,s,t) ∧ *Breeze*(t) ⇒ *Breezy*(s), the agent infers properties of the square from properties of its current percept
  - Breezy() has no time argument
  - Having discovered which places are breezy or smelly, not breezy or not smelly, the agent can deduce where the pits are and where the wumpus is

Diagnostic Rules

- Diagnostic rule: lead from observed effects to hidden causes
  - If the square is breezy then adjacent square(s) must contain a pit

    ∀s Breezy(s) ⇒ ∃r Adjacent(r,s) ∧ Pit(r)
  - If the square is not breezy, no adjacent square contains a pit

    ∀s ¬Breezy(s) ⇒ (∀r Adjacent(r,s) ⇒ ¬Pit(r))
  - Combining both:

    ∀s Breezy(s) ⇔ ∃r Adjacent(r,s) ∧ Pit(r)

Causal Rules

- Causal rule: some hidden property of the world causes certain percept to be generated
  - A pit causes all adjacent squares to be breezy

    ∀r Pit(r) ⇒ [∀s Adjacent(r,s) ⇒ Breezy(s)]
  - If all squares adjacent to a given square are pitless, the square will not be breezy

    ∀s [∀r [Adjacent(r,s)⇒ ¬Pit(r)] ⇒ ¬Breezy(s)

Summary

- First-order logic:
  - objects and relations are semantic primitives
  - syntax: constants, functions, predicates, equality, quantifiers

- Increased expressive power: sufficient to define wumpus world

Exercises

- Represent the following sentences in FOL
  - Some students took French in spring 2001.
  - Every student who takes French passes it.
  - Only one student took Greek in spring 2001.
  - The best score in Greek is always higher than the best score in French.
- Let the basic vocabulary be as follows:

$Student(x)$

$Takes(x, c, s)$: student x takes course c in semester s;
$Passes(x, c, s)$: student x passes course c in semester s;
$Score(x, c, s)$: the score obtained by student x in course c in semester s;
$x > y$: x is greater than y;
F and G: specific French and Greek courses (one could also interpret these sentences as referring to *any* such course, in which case one could use a predicate $Subject(c, f)$ meaning that the subject of course c is field f;

Inference in FOL

- Two ideas:
  - convert the KB to propositional logic and use propositional inference
  - a shortcut that manipulates on first-order sentences directly (resolution, will not be introduced here)

Universal Instantiation

- Universal instantiation
  - infer any sentence by substituting a ground term (a term without variables) for the variable
  - $$\frac{\forall v\ \alpha}{\text{Subst}(\{v/g\},\ \alpha)}$$
  - for any variable v and ground term g
- Examples
  - $\forall x$ *King*(x) $\Rightarrow$ *Greedy*(x) $\Rightarrow$ *Evil*(x) yields:
  - *King*(*John*) $\Rightarrow$ *Greedy*(*John*) $\Rightarrow$ *Evil*(*John*)
  - *King*(*Richard*) $\Rightarrow$ *Greedy*(*Richard*) $\Rightarrow$ *Evil*(*Richard*)
  - *King*(*Father*(*John*)) $\Rightarrow$ *Greedy*(*Father*(*John*)) $\Rightarrow$ *Evil*(*Father*(*John*))

Existential Instantiation

- Existential instantiation
  - For any sentence α, variable v, and constant symbol k that does NOT appear elsewhere in the knowledge base, replace v with k

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

- E.g., $\exists x$ *Crown*(x) $\wedge$ *OnHead*(x, *John*) yields:

$$\textit{Crown}(C_1) \wedge \textit{OnHead}(C_1, \textit{John})$$

provided C_1 is a new constant symbol, called a Skolem constant

Exercise

- Suppose a knowledge base contains just one sentence, $\exists x$ AsHighAs(x, Everest). Which of the following are legitimate results of applying Existential Instantiation?
  - AsHighAs(Everest, Everest)
  - AsHighAs(Kilimanjaro, Everest)
  - AsHighAs(Kilimajaro, Everest) and AsHighAs(BenNevis, Everest)

Reduction to Propositional Inference

Suppose the KB contains just the following:

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

- Instantiating the universal sentence in all possible ways, we have:
  King(John) $\wedge$ *Greedy(John)* $\Rightarrow$ *Evil(John)* *King(Richard)* $\wedge$ *Greedy(Richard)* $\Rightarrow$ *Evil(Richard)* *King(John)*
  Greedy(John)
  Brother(Richard,John)

- The new KB is propositionalized: proposition symbols are
- *King(John), Greedy(John), Evil(John), King(Richard)*, etc.

- What conclusion can you get?

Reduction Cont'd.

- Every FOL KB can be propositionalized so as to preserve entailment
- (A ground sentence is entailed by new KB iff entailed by original KB)
- Idea: propositionalize KB and query, apply resolution, return result

Problems with Propositionalization

- Propositionalization seems to generate lots of irrelevant sentences

- E.g., from:

  $\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

  $King(John)$

  $\forall y\ Greedy(y)$

  $Brother(Richard,John)$

- It seems obvious that *Evil*(*John*) is true, but propositionalization produces lots of facts such as *Greedy*(*Richard*) that are irrelevant

- With p k-ary predicates and n constants, there are $p \cdot n^k$ instantiations

Problems with Propositionalization

- When the KB includes a function symbol, the set of possible ground term substitutions is infinite
  - Father(Father(…(John)…))

- Theorem:
  - If a sentence is entailed by the original, first-order KB, then there is a proof involving just a finite subset of the propositionalized KB
  - First instantiation with constant symbols
  - Then terms with depth 1 (Father(John))
  - Then terms with depth 2 …
  - Until the sentence is entailed

Unification and Lifting

Propositionalization approach is rather inefficient

A First-Order Inference Rule

- If there is some substitution θ that makes the premise of the implication identical to sentences already in the KB, then we assert the conclusion of the implication, after applying θ

$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

$King(John)$

$Greedy(John)$

What's θ here?

- Sometimes, we need to do is find a substitution θ both for the variables in the implication and in the sentence to be matched

$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

$King(John)$

$\forall y\ Greedy(y)$

$Brother(Richard,John)$

What's θ here?

Generalized Modus Ponens

- For atomic sentences p_i, p_i', and q, where there is a substitution θ such that Subst(θ, p_i') = Subst(θ, p_i), for all i:

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

p_1' is *King*(*John*) p_1 is *King*(*x*)

p_2' is *Greedy*(*y*) p_2 is *Greedy*(*x*)

θ is {x/John,y/John} q is *Evil*(*x*)

Subst(θ, q) is *Evil*(*John*)

Can be used with KB of definite clauses (exactly one positive literal)

Unification

- GMP is a lifted version of Modus Ponens – raises Modus Ponens from propositional to first-order logic
- What's the key advantage of GMP over propositionalization?
- Lifted inference rules require finding substitutions that make different logical expressions look identical
- This process is called **unification** and is a key component of all first-order inference algorithms

Unification

- The unify algorithm takes two sentences and returns a unifier for them if one exists:
  - UNIFY(p, q) = θ where Subst(θ, p) = Subst(θ, q)

p	q	θ
Knows(John,x)	Knows(John,Jane)	
Knows(John,x)	Knows(y,OJ)	
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x,OJ)	

Unification

- The unify algorithm takes two sentences and returns a unifier for them if one exists:
  - UNIFY(p, q) = θ where Subst(θ, p) = Subst(θ, q)

p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x,OJ)	

Unification

- The unify algorithm takes two sentences and returns a unifier for them if one exists:
  - UNIFY(p, q) = θ where Subst(θ, p) = Subst(θ, q)

p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x,OJ)	

Unification

- The unify algorithm takes two sentences and returns a unifier for them if one exists:
  - UNIFY(p, q) = θ where Subst(θ, p) = Subst(θ, q)

p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}
Knows(John,x)	Knows(x,OJ)	

Unification

- The unify algorithm takes two sentences and returns a unifier for them if one exists:
  - UNIFY(p, q) = θ where Subst(θ, p) = Subst(θ, q)

p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}
Knows(John,x)	Knows(x,OJ)	{fail}

Standardizing apart: renaming variables to avoid name clashes

UNIFY(Knows(John, x), Knows(z, OJ)) = {x/OJ, z/John}

Unification Contd.

- To unify *Knows(John, x)* and *Knows(y, z)*,
  θ = { y/John, x/z } or θ = { y/John, x/John, z/John}

- The first unifier is more general than the second
- There is a single most general unifier (MGU)
- MGU = {y/John, x/z}

Exercise

- For each pair of atomic sentences, give the most general unifier if it exists:
  - P(A, B, B), P(x, y, z)
  - Q(y, G(A, B)), Q(G(x, x), y)
  - Older(Father(y), y), Older(Father(x), John)
  - Knows(Father(y), y), Knows(x, x)

Example Knowledge Base

- 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.
- Prove that Colonel West is a criminal
  - American(x): x is an American
  - Weapon(x): x is a weapon
  - Hostile(x): x is a hostile nation
  - Criminal(x): x is a criminal
  - Missile(x): x is a missile
  - Owns(x, y): x owns y
  - Sells(x, y, z): x sells y to z
  - Enemy(x, y): x is an enemy of y
  - Constants: America, Nono, West

Example Knowledge Base

First-Order Definite Clauses

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

$American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \Rightarrow Criminal(x)$

Nono … has some missiles, i.e., $\exists x\ Owns(Nono,x) \land Missile(x)$:

$Owns(Nono,M_1)$ and $Missile(M_1)$

… all of its missiles were sold to it by Colonel West

$Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

Missiles are weapons:

$Missile(x) \Rightarrow Weapon(x)$

An enemy of America counts as "hostile":

$Enemy(x, America) \Rightarrow Hostile(x)$

West, who is American …

$American(West)$

The country Nono, an enemy of America …

$Enemy(Nono,America)$

Forward Chaining

- Starting from known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. This process will continue until query is answered.

- When a new fact P is added to the KB:
  - For each rule such that P unifies with a premise
    - if the other premises are already known
    - then add the conclusion to the KB and continue chaining
- Forward chaining is data-driven:
  - inferring properties and categories from percepts

Forward Chaining Algorithm

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*

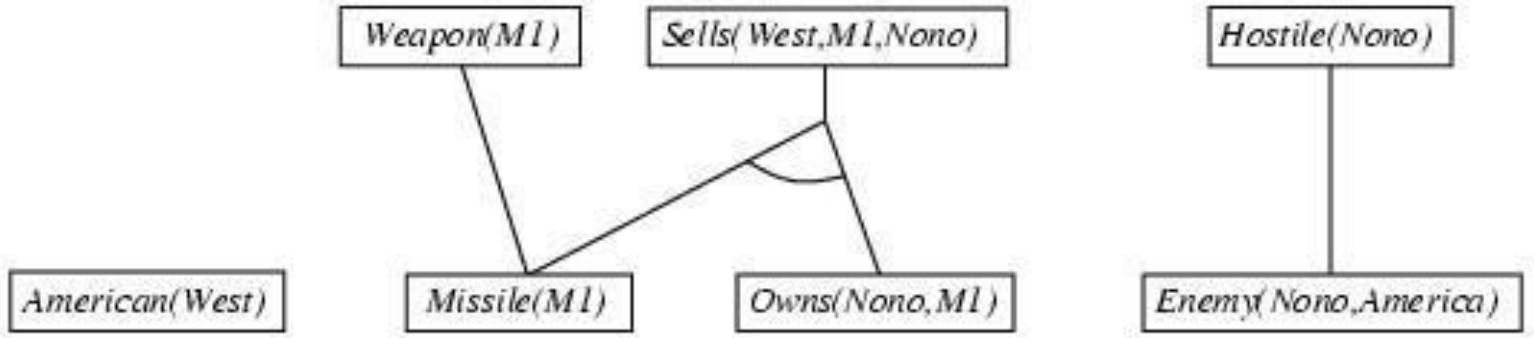
repeat until *new* is empty
  new $\leftarrow \{\}$
  for each sentence r **in** KB **do**
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
    for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
        for some $p'_1,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

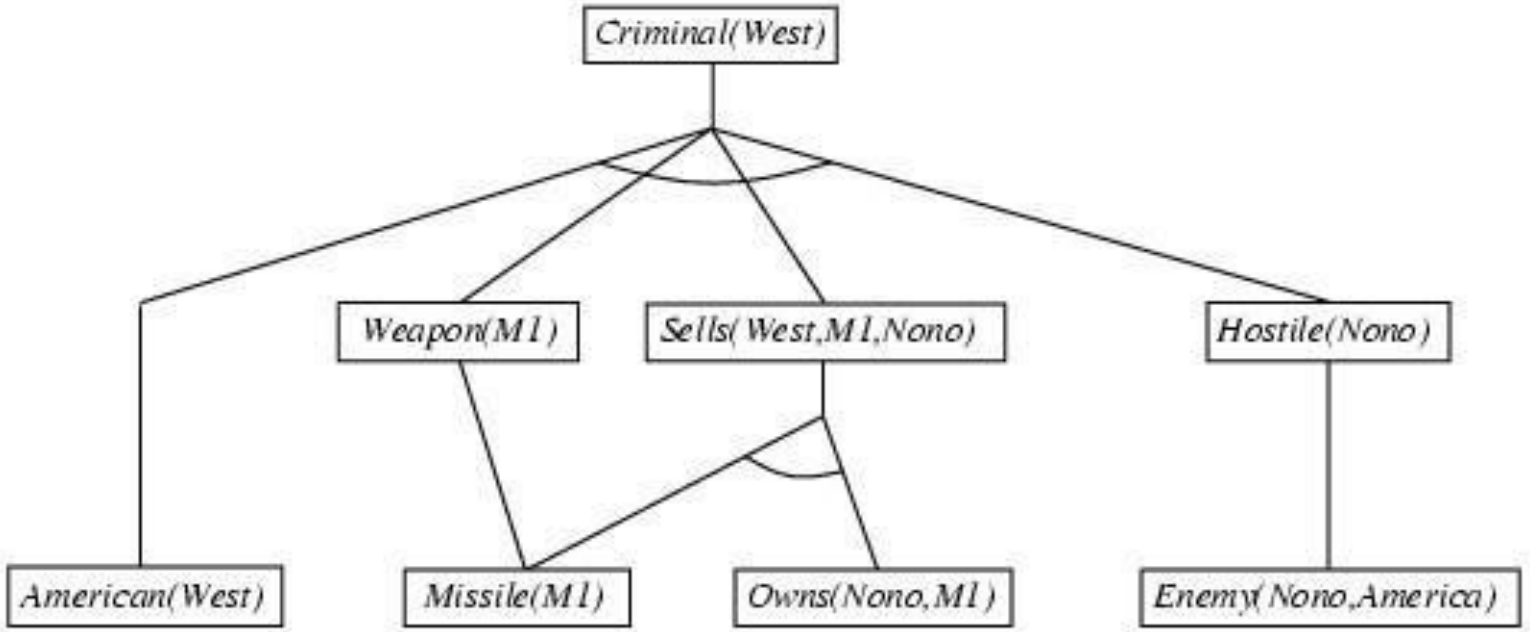
American(West)

Missile(M1)

Owns(Nono,M1)

Enemy(Nono,America)


Weapon(M1)
Sells(West,M1,Nono)
Hostile(Nono)
American(West)
Missile(M1)
Owns(Nono,M1)
Enemy(Nono,America)


Criminal(West)
Weapon(M1)
Sells(West,M1,Nono)
Hostile(Nono)
American(West)
Missile(M1)
Owns(Nono,M1)
Enemy(Nono,America)

Analysis of Forward Chaining

- A fixed point of the inference process can be reached
- The algorithm is sound and complete
- Its efficiency can be improved

FC Algorithm Analysis

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*

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

Sources of Complexity

- 1. "Inner loop" involves finding all possible unifiers such that the premise of a rule unifies with facts in the KB
  - Often called "pattern matching", very expensive
- 2. The algorithm rechecks every rule on every iteration to see whether its premises are met
- 3. It might generate many facts that are irrelevant to the goal

Matching Rules Against Known Facts

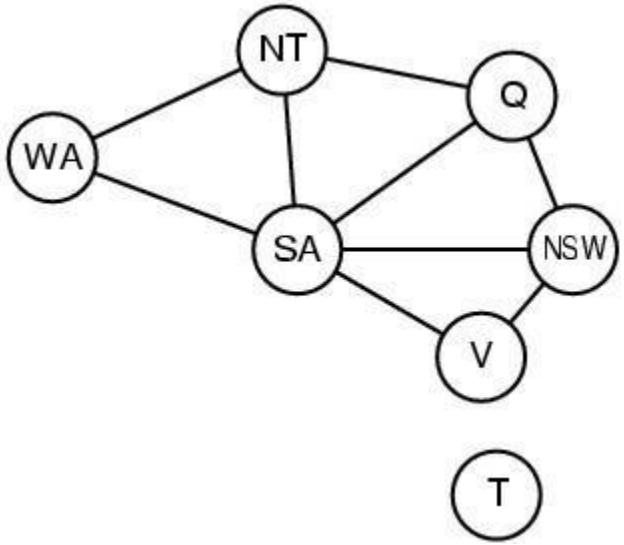
- Consider a rule:
  - *Owns(Nono, x)Missile(x) Sells(West, x, Nono)*
  - We find all the objects owned by Nono in constant time per object;
  - Then, for each object, we check whether it's a missile.
  - If more objects and very few missiles inefficient
  - Conjunct ordering problem: NP-hard
  - Heuristics?

Pattern Matching and CSP

- 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 owned by Nono
- We can actually express every finite-domain CSP as a single definite clause together with some associated ground facts

Diff(wa, nt) Diff(wa, sa)
Diff(nt, q) Diff(nt, sa)
Diff(nsw, v) Diff(nsw, v)
Diff(v, sa) Colorable()

Diff(R, B) Diff(R, G)
Diff(G, R) Diff(G, B)
Diff(B, R) Diff(B, G)



Incremental Forward Chaining

- Observation:
  - Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t-1
- Modification:
  - At iteration t, we check a rule only if its premise includes a conjunct p_i that unifies with a fact p_i' newly inferred at iteration t-1
  - Many real systems operate in an "update" mode wherein forward chaining occurs in response to each new fact that is TOLD to the system

Irrelevant Facts

- FC makes all allowable inferences based on known facts, even if they are irrelevant to the goal at hand
- Similar to FC in propositional context

- Solution?

Backward Chaining

- $p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q$
- When a query q is examined:
  - if a matching fact q' is known, return the unifier
  - for each rule whose consequent q' matches q
  - attempt to prove each premise of the rule by backward chaining

- Backward chaining is the basis for "logic programming," e.g., Prolog

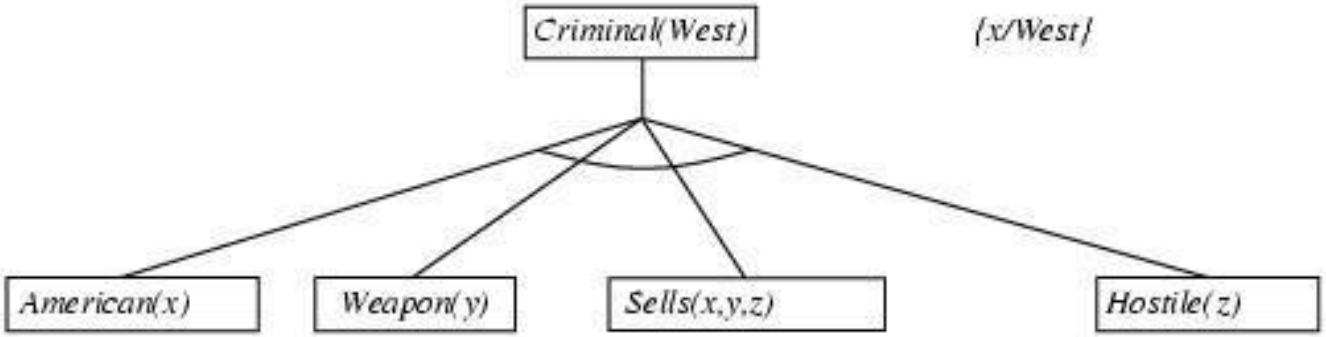
Backward Chaining Algorithm

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, the current substitution, initially the empty substitution $\{\}$
  local variables: ans, a set of substitutions, initially empty

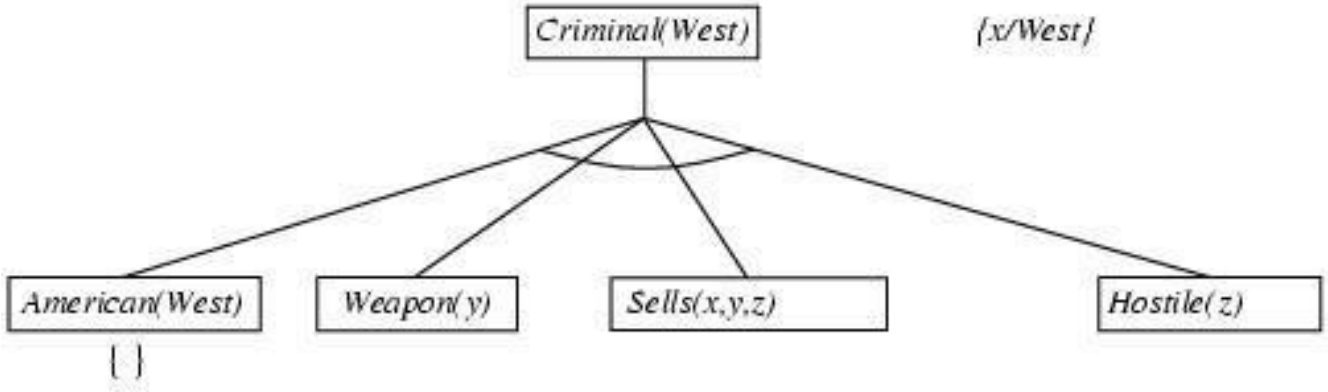
  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each r **in** KB where STANDARDIZE-APART(r) $= (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    $ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n | \text{REST}(goals)]$,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

Backward Chaining Example

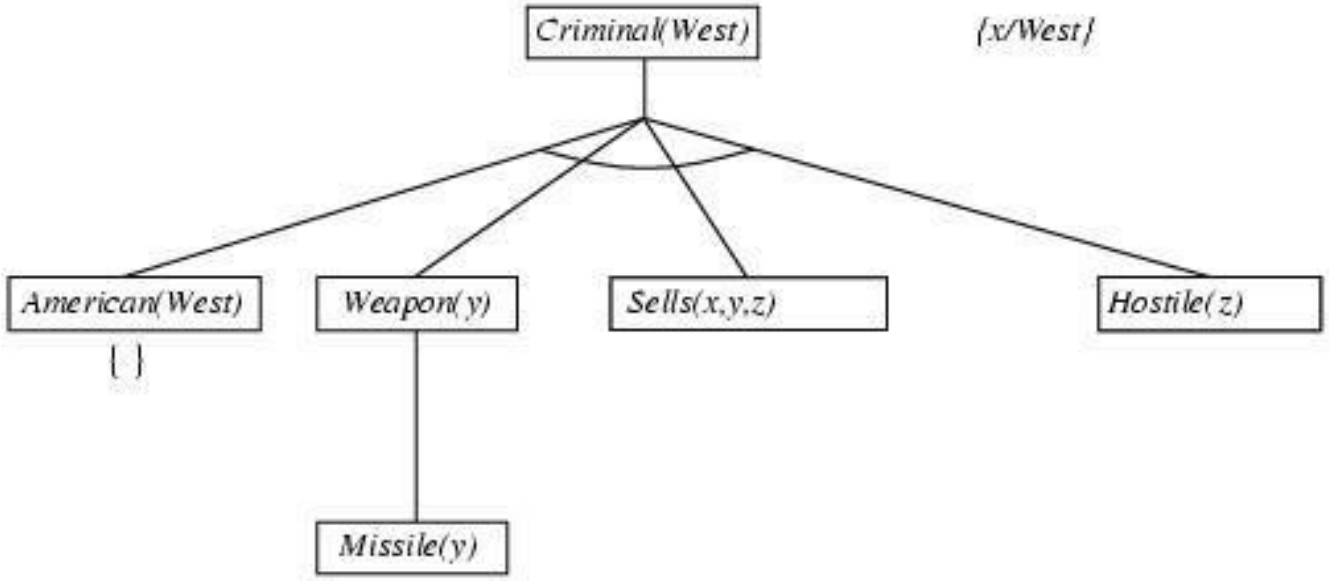
Backward Chaining Example



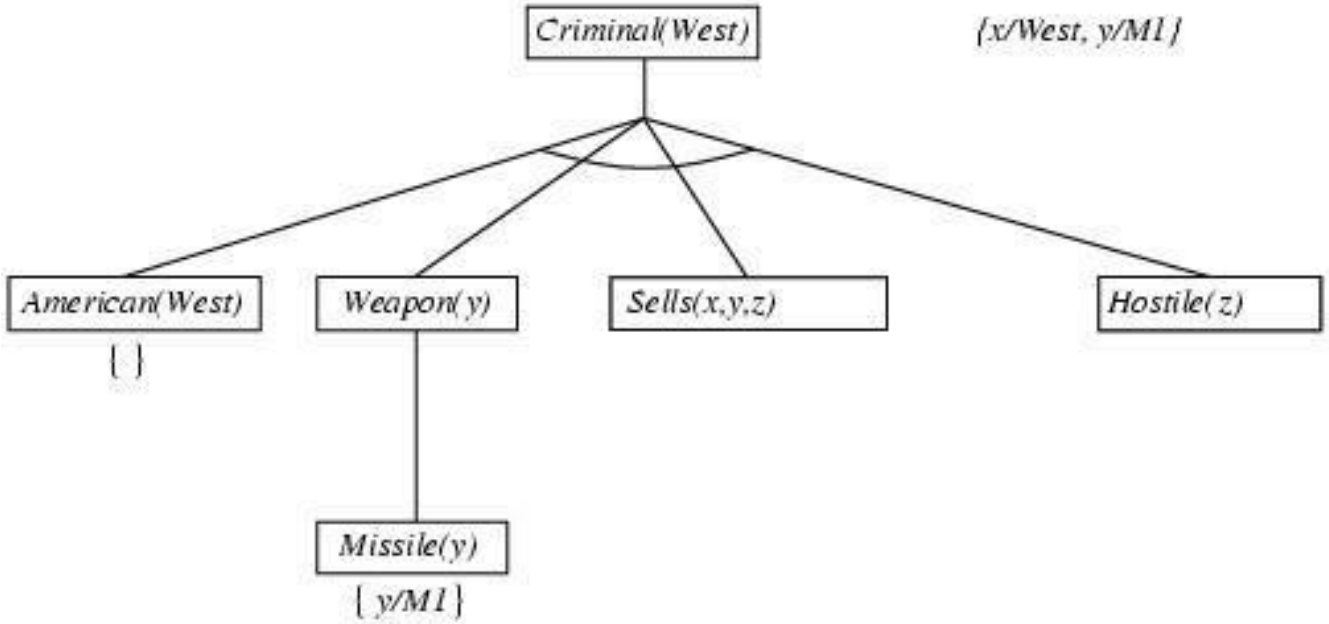
Backward Chaining Example



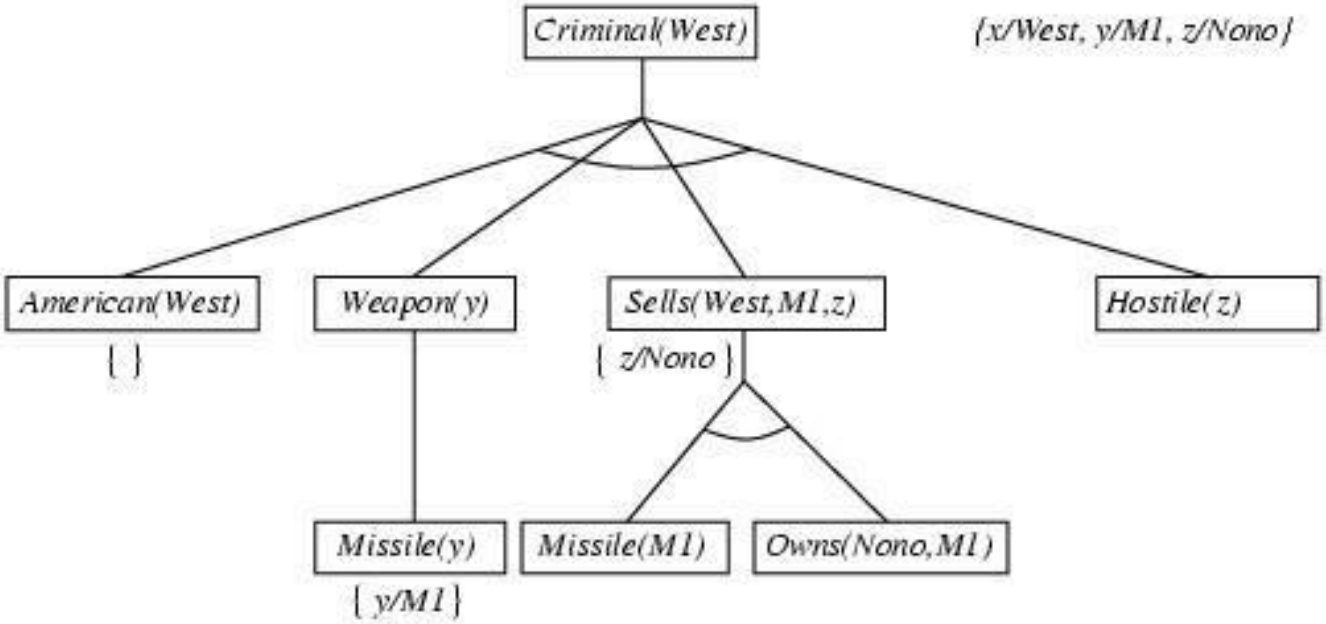
Backward Chaining Example



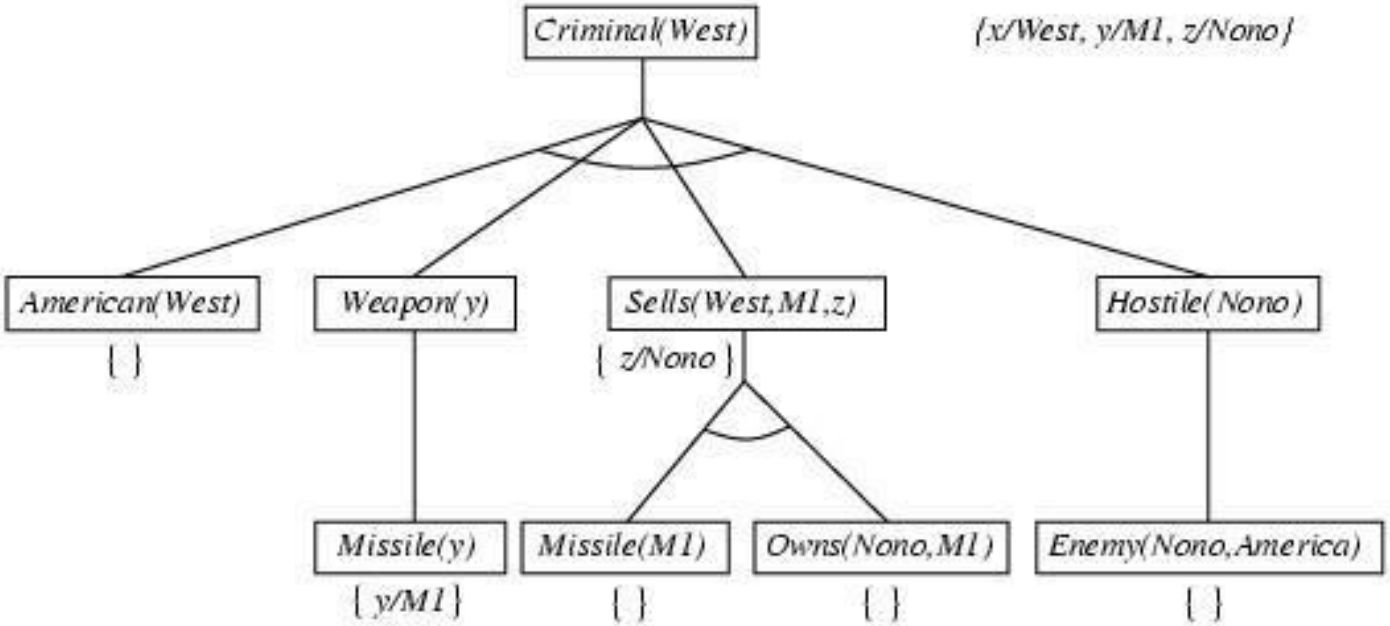
Backward Chaining Example



Backward Chaining Example



Backward Chaining Example



Analysis of Backward Chaining

- Depth-first search: space is linear in size of proof
- Repeated states and incompleteness
  - can be fixed by checking current goal against every goal on stack
- Inefficient due to repeated subgoals
  - can be fixed by caching previous results
- Widely used in logic programming

Logic Programming

- Prolog
- Lisp

- Introduced in CS 352 (symbolic programming)

- http://www.cs.toronto.edu/~sheila/384/w11/simple-prolog-examples.html

Exercise

- Write down logical representations for the following sentences, suitable for use with Generalized Modus Ponens
  - Horses, cows, and pigs are mammals.
  - An offspring of a horse is a horse.
  - Bluebeard is a horse.
  - Bluebeard is Charlie's parent.
  - Offspring and parent are inverse relations.

  - Draw the proof tree generated by exhaustive backward-chaining algorithm for the query ?h Horse(h), where clauses are matched in the order given.

UNIT-V

Knowledge Representation: Ontological Engineering, Categories and Objects, Events. Mental Events and Mental Objects, Reasoning Systems for Categories, Reasoning with Default Information.

Quantifying Uncertainty: Acting under Uncertainty, Basic Probability Notation, Inference Using Full Joint Distributions, Independence, Bayes' Rule and Its Use.

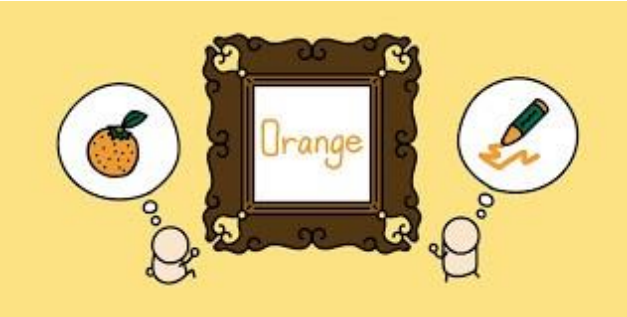
KNOWLEDGE REPRESENTATION

Knowledge-representation is the field of AI dedicated to representing information about the world in a form that a computer system can utilize to solve complex tasks.

• This topic addresses what content to put into knowledge base
• How to represent facts about the world?

Ontology engineering is a set of tasks related to the development of ontologies for a particular domain.

• Google definition: a set of concepts and categories in a subject area or domain that shows their properties and the relations between them.



How to create more general and flexible representations

– Concepts like actions, time, physical objects and beliefs

– Operates on a bigger scale than knowledge engineering

• Representing these abstract concepts is sometimes called **ontological engineering**.

• Define general framework of concepts (because representing everything is challenging) called as **upper ontology** with general concepts at the top and more specific concepts below the hierarchy

• Limitations of logic representation

– Red, green and yellow tomatoes: exceptions and uncertainty

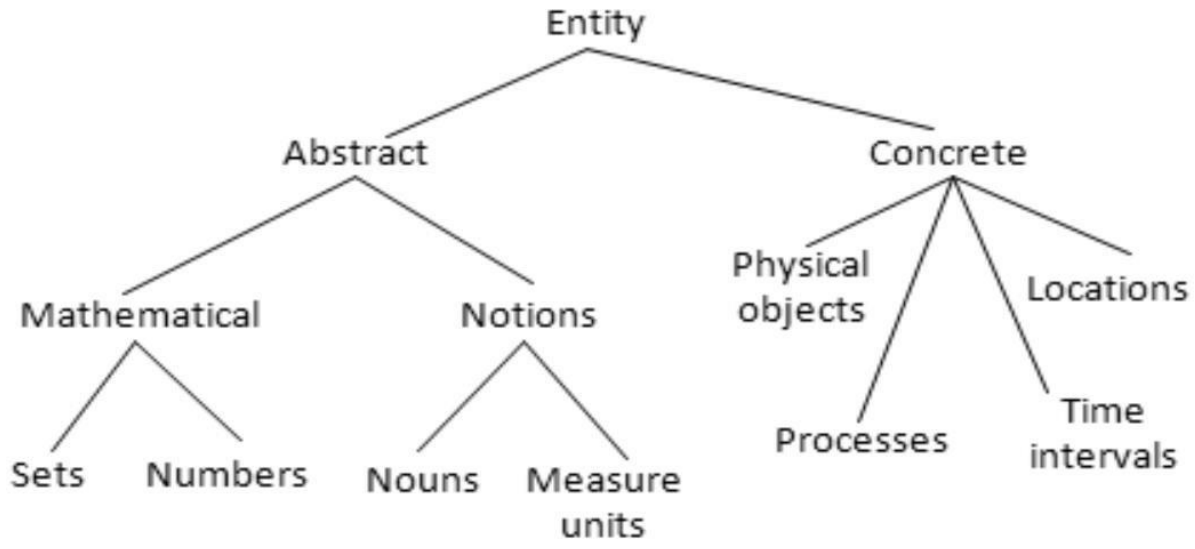
Defining terms in the domain and relations among them

– Defining concepts in the domain(classes)

– Arranging the concepts in a hierarchy(subclass-superclass hierarchy)

– Defining which attributes and properties classes can have and constraints on their values

– Defining individuals and filling in property values



The upper ontology of the world. Each link indicates that the lower concept

is a specialization of the upper one.

General-purpose ontology

A general-purpose ontology should be applicable in more or less any special-purpose domain.

– Add domain-specific axioms

• In any sufficiently demanding domain, different areas of knowledge need to be unified.

– Reasoning and problem solving could involve several areas simultaneously

• What do we need to express?

– Categories, Measures, Composite objects, Time, Space, Change, Events, Processes, Physical Objects, Substances, Mental Objects, Beliefs

Categories and objects

KR requires the organization of objects into categories. Although

– Interaction at the level of the object

– Reasoning at the level of categories

• Categories play a role in predictions about objects

– Based on perceived properties

• Categories can be represented in two ways by FOL

1. Predicates: ***apple(x)***

2. Reification of categories into **objects**: ***apples***

• Category = set of its members

– Example: Member(x, apples), x ∈ apples,

– Subset(apples, fruits), apples ⊂ fruits

Category Organization

Categories serve to organize and simplify the knowledge base through **inheritance**.

• Relation = inheritance:

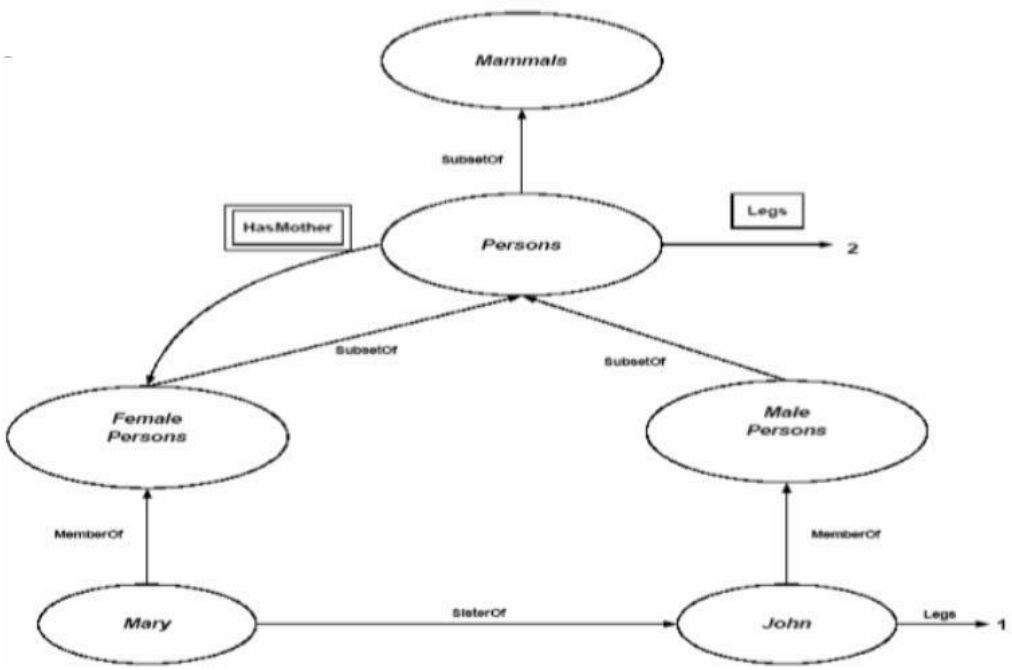
– All instance of food are edible, fruit is a subclass of food and apples is a subclass of fruit then an apple is edible.

– Individual apples **inherit** the property of edibility from food

• Defines a taxonomy

– Subclass relations

organize categories



FOL and Categories

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members.

Example:

• An object is a member of a category

– MemberOf(BB12,Basketballs)

• A category is a subclass of another category

– SubsetOf(Basketballs, Balls)

• All members of a category have some properties

– ∀ x (MemberOf(x, Basketballs) ⇒ Round(x))

• All members of a category can be recognized by some properties

– ∀ x (Orange(x) ∧ Round(x) ∧ Diameter(x)=9.5in ∧

MemberOf(x,Balls) ⇒ MemberOf(x, BasketBalls))

• A category as a whole has some properties

– MemberOf(Dogs, DomesticatedSpecies)

Relations between Categories

Two or more categories are disjoint if they are mutually exclusive

– Disjoint({Animals, Vegetables})

• A decomposition of a class into categories is called exhaustive if each object of the class must belong to atleast one category

– living = {animal, vegetable, fungi, bacteria}

• A partition is an exhaustive decomposition of a class into disjoint subsets.

– student = {undergraduate, graduate}

Natural kinds

Many categories have no clear-cut definitions (e.g., chair, bush, book).

• Tomatoes: sometimes green, orange, red, yellow. Mostly spherical, smaller, larger etc

• One solution: Separate what is true of all instances of a category from what is true only of typical instances.

• subclass using category Typical(Tomatoes)

– Typical(c) ⊆ c

• ∀ x, x ∈ Typical(Tomatoes) ⇒ Red(x) ∧ Spherical(x)

• This helps us to write down useful facts about categories without providing exact definitions.

Physical composition

One object may be part of another:

– PartOf(Bucharest,Romania)

– PartOf(Romania,EasternEurope)

– PartOf(EasternEurope,Europe)

• The PartOf predicate is transitive (and reflexive), so we can infer that PartOf(Bucharest,Europe)

• More generally:

– ∀ x PartOf(x,x)

– ∀ x,y,z PartOf(x,y) ∧ PartOf(y,z) ⇒ PartOf(x,z)

Logical Minimization: Defining an object as the smallest one satisfying certain conditions.

Measurements

Objects have height, mass, cost,....

– **Values that we assign to these properties are called measures**

• **Combine Unit functions with a number to measure line segment object L1**

– **Length(L1) = Inches(1.5) = Centimeters(3.81).**

• **Conversion between units:**

– ∀ **i Centimeters(2.54 x i)=Inches(i).**

• **Some measures have no scale:**

– **Beauty, Difficulty, etc.**

– **Most important aspect of measures is not numerical values, but measures can be orderable.**

– **An apple can have deliciousness .9 or .1**

Objects: Things and stuff

Stuff: Defy any obvious individuation after division into distinct objects (mass nouns)

• **Things: Individuation(count nouns)**

• **Example: Butter(stuff) and Cow(things)**

$b \in Butter \wedge PartOf(p, b) \Rightarrow p \in Butter$

Events

Event calculus, models how the truth value of relations changes because of events occurring at **certain times**.

• Event *E* occurring at time *T* is written as *event(E,T)*.

• It is designed to allow reasoning over intervals of time.

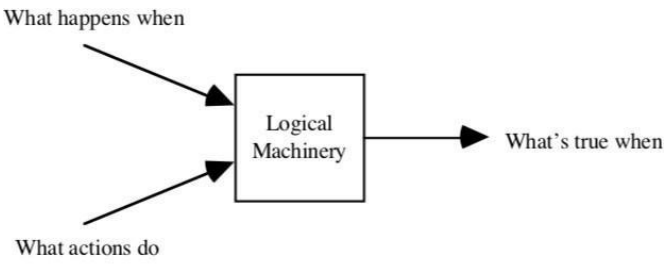


Figure 1: How the Event Calculus Functions

Reified Fluents in Event calculus

Fluents: Is a condition that can change over time.

• In logical approach, fluent is a predicate or function that vary from one situation to the next.

– "the box is on the table" - **On(box, table)**

– if it can change over time - **On(box, table, t)**

– Here "On" is a predicate.

• Fluents can also be represented by functions are said to be reification

• When using reified fluents, a separate predicate is necessary to tell when a fluent is actually true or not.

• For example, **HoldsAt(on(box,table), t)** means that the box is actually on the table at time t, where the predicate **HoldsAt** is the one that tells when fluents are true.

• This representation of reified fluents is used in the event calculus.

Complete set of predicates for one version of the event calculus

- **HoldsAt(f,t) - Fluent f is true at time t**
- **Happens(e, i) - Event e happens at time i**
- **Initiates (e, f , t) - Event e causes fluent f to be true after time t**
- **Terminates (e, f , t) - Event e causes fluent f to cease after time t**
- **Clipped(t, f, t2) - Fluent f ceases to be true at some point during time interval between t and t2**
- **Restored(t, f, t2) - Fluent f becomes true sometime during time interval between t and t2**

The Axioms of the Simple Event Calculus

Collection of axioms relating the various predicates together to represent an action

$$HoldsAt(f,t) \leftarrow [Happens(e,t_1) \wedge Initiates(e,f,t_1) \wedge (t_1 < t) \wedge \neg Clipped(t_1,f,t)]$$

This formula means that the fluent represented by the term f is true at time t if:

1. an event e has taken place: $Happens(e,t_1)$;
2. this took place in the past: $t_1 < t$;
3. this event has the fluent f as an effect: $Initiates(e,f,t_1)$;
4. the fluent has not been made false in the meantime: $Clipped(t_1,f,t)$

• A fluent is true at time *t* if and only if it has been made true in the past and has not been made false in the meantime.

The *Clipped* predicate, stating that a fluent has been made false during an interval, can be axiomatized as follows:

$$\text{Clipped}(t1, f, t2) \Leftrightarrow \exists e, t[\text{Happens}(e, t) \wedge t1 \leq t < t2 \wedge \text{Terminates}(e, f, t)]$$

• The *Restored* predicate, stating that a fluent has been made true during an interval, can be axiomatized as follows:

$$\text{Restored}(t1, f, t2) \Leftrightarrow \exists e, t[\text{Happens}(e, t) \wedge t1 \leq t < t2 \wedge \text{Initiates}(e, f, t)]$$

Example

Happens(Turnoff(LightSwitch1),1:00) – Lightswitch was turned off at exactly 1:00

Processes

Any subinterval of a process is also a member of same process category called process category or liquid category

• Any process e that happens over an interval also happens over any subinterval:

$$(e \in \text{Processes}) \wedge \text{Happens}(e, (t1, t4)) \wedge (t1 < t2 < t3 < t4) \Rightarrow \text{Happens}(e, (t2, t3))$$

Example:

In(Shankar, New Delhi) – Shankar being in New Delhi

T(In(Shankar, New Delhi), Today) – He was in New Delhi all day

Time Interval

Time is important to any agent that takes action

• 2 kinds of time intervals:

1. moments

2. extended intervals

• The distinction is that only moments have zero duration

– **Partition ({Moments , ExtendedIntervals },**

Intervals)

- **i∈Moments ⇔ Duration(i)=Seconds(0)**

Time scale

The moment at midnight (GMT) on Jan 1, 1900 has time 0

• Interval(i) ‹ Duration(i)=(Time(End(i))−Time(Begin(i))).

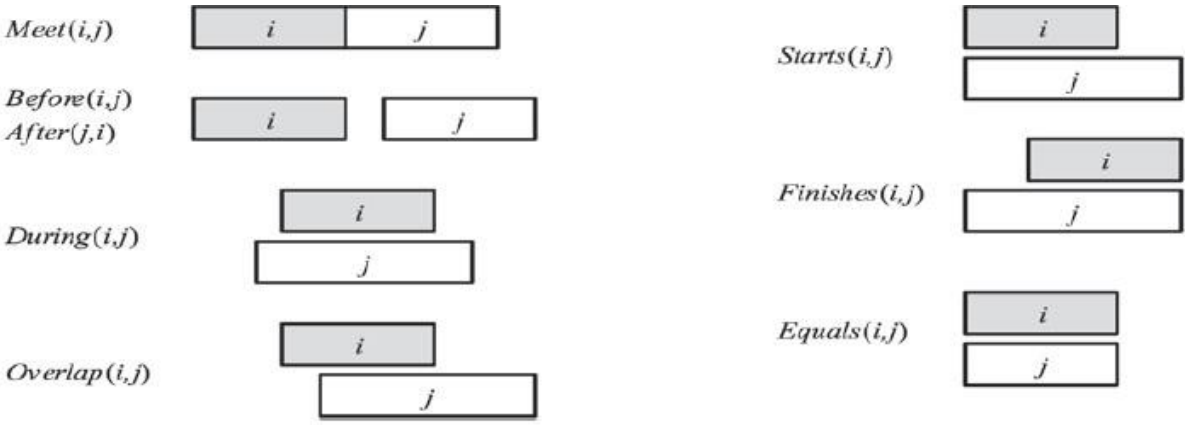
• Time (Begin (AD 1900)) = Seconds (0).

• Time(Begin(AD2001)) = Seconds(3187324800)

• Time(End(AD2001))=Seconds(3218860800)

• Duration(AD2001)= Seconds(31536000)

Predicate of time intervals



$$
\begin{aligned}
Meet(i,j) &\Leftrightarrow End(i) = Begin(j)\\
Before(i,j) &\Leftrightarrow End(i) < Begin(j)\\
After(j,i) &\Leftrightarrow Before(i,j)\\
During(i,j) &\Leftrightarrow Begin(j) < Begin(i) < End(i) < End(j)\\
Overlap(i,j) &\Leftrightarrow Begin(i) < Begin(j) < End(i) < End(j)\\
Begins(i,j) &\Leftrightarrow Begin(i) = Begin(j)\\
Finishes(i,j) &\Leftrightarrow End(i) = End(j)\\
Equals(i,j) &\Leftrightarrow Begin(i) = Begin(j) \wedge End(i) = End(j)
\end{aligned}
$$

CS869
1

Logic of time intervals

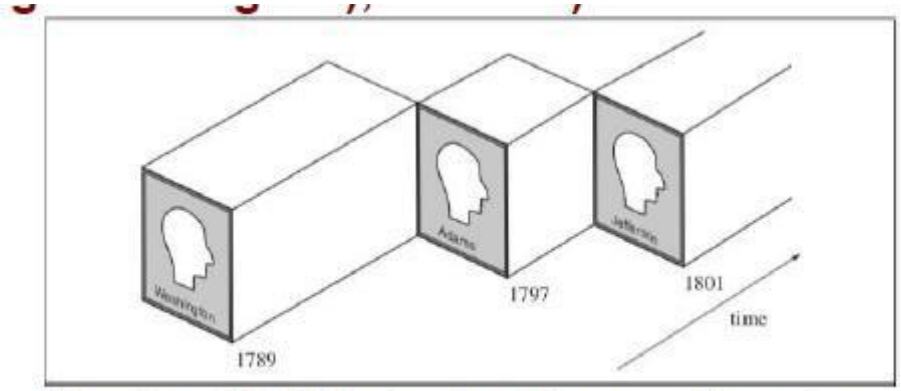
Fluents and objects

Physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time.

• President(USA, t) is a logical term that denotes a different object at different times.

• To say that George Washington was president throughout 1790, we can write

T (Equals (President (USA), GeorgeWashington), AD 1790)



object President(USA) for the first 15 years of its existence

Mental Events and Mental Objects

For a single- agent domains, knowledge about one's own knowledge and reasoning processes is useful for controlling inference.

• In a multiagent domains, it becomes important for an agent to reason about the mental states of the other agents.

• Example Bob and Alice scenario

• It requires a model of the mental objects in someone's head(knowledge base) and the processes that manipulate these mental objects.

A formal theory of beliefs

Relationships between agents and mental objects: believes, knows, wants, intends ... are called propositional attitudes

• Lois knows that Superman can fly:

– Knows (Lois , CanFly (Superman))

• If Superman is Clark, then we must conclude that Lois knows that Clark can fly:

– (Superman = Clark) ∧ Knows (Lois , CanFly (Superman)) |=

Knows(Lois, CanFly(Clark))

• "Can Clark fly?" – No. Need descriptions

• If an agent knows that 2 + 2 = 4 and 4 < 5, then we want an agent to know that 2 + 2 < 5. This property is called referential transparency

For propositional attitudes like *believes* and *knows*, we would like to have referential opacity

Modal logic address this problem

• Modal logic includes special modal operators that take sentences (rather than terms) as arguments.

• Example: "*A* knows *P*" is represented with the notation

$K_A P$,

- where K is the modal operator for knowledge,

- agent A (written as the subscript) and a sentence

Opacity: blur, condition of lacking transparency

<u>Possible Worlds And Accessibility Relations</u>

In modal logic, consider both the possibility that Superman's secret identity is Clark and that it isn't.

– Clark=Superman and Clark!=Superman

• Build a model, that consists of a collection of possible worlds rather than just one true world and the worlds are connected in a graph by accessibility relations

• "Bond knows that someone is a spy" is ambiguous.

∃x KBondSpy(x)

which in modal logic means that there is an x that, in all accessible worlds, Bond knows to be a spy.

• Bond just knows that there is at least one spy:

KBond∃x Spy(x)

The modal logic interpretation is that in each accessible world there is an x that is a spy, but it need not be the same x in each world.

<u>Knowledge and belief</u>

Knowledge in terms if AI is something which is always true

• Belief on the other hand deals more with probability

• After extensive study, it is commonly said that knowledge is justified true belief

• **Example:**

– Eating food necessary for living, so we eat.

– Gambling to gain money. Probability. Believe we will win

• **Knows(a, p) – agent a knows that proposition p is true**

• **Lois knows whether Clark can fly if she either knows that Clark can fly or knows that he cannot**

– $\text{Knowswhether}(a, p) \Leftrightarrow \text{knows}(a, p) \vee \text{knows}(a, \neg p)$

Knowledge, time and action

Belief is also something which can change over time. For example:

• **Lois believes Superman flies today**

T(Believes(Lois, flies(Superman)),Today)

• **The same will not be true in 100 years when he died of old age**

<u>Reasoning Systems for Categories</u>

Categories are the primary building blocks of large-scale knowledge representation schemes.

• **This topic describes systems specially designed for organizing and reasoning with categories.**

• **There are two types of reasoning systems:**

1. Semantic networks

2. Description logics

<u>Reasoning systems</u>

Semantic networks

– Visualize knowledge-base in patterns of interconnected nodes and arcs

– Efficient algorithms for inferring of object on the basis of its category membership

Description logics

– Formal language for constructing and combining category definitions

– Efficient algorithms to decide subset and superset relationships between categories.

<u>Semantic Networks</u>

In 1909, Charles S. Peirce proposed a graphical notation of nodes and edges called existential graphs

• A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled arcs/links

• For Example:

– MemberOf link between Mary and FemalePersons, corresponding to the logical assertion

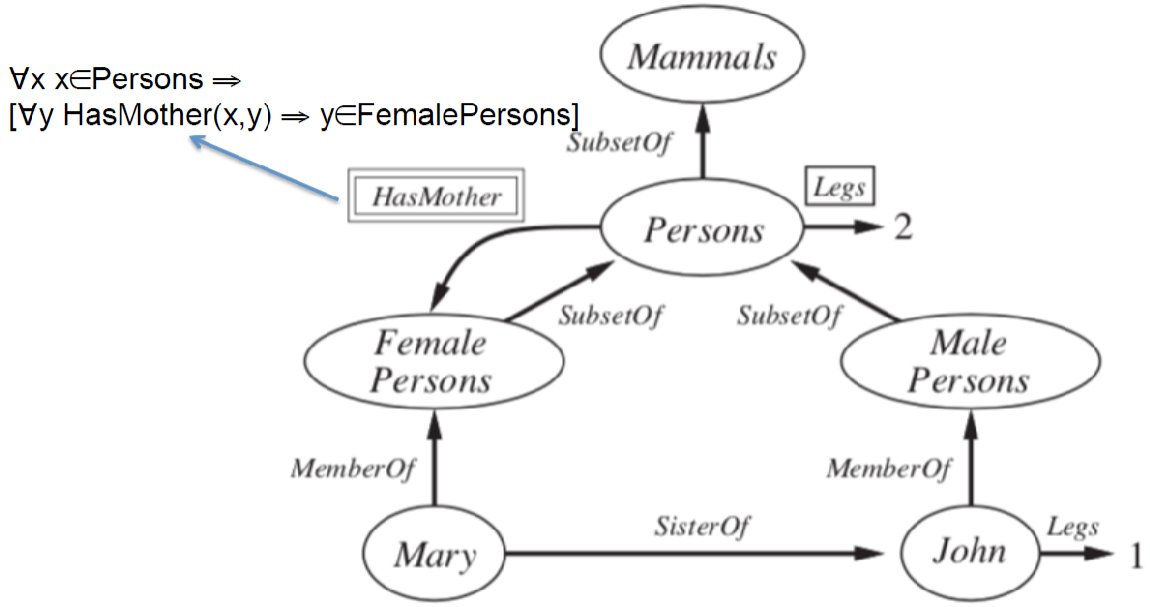
Mary ∈ FemalePersons

– SisterOf link between Mary and John corresponds to the

assertion SisterOf (Mary, John)

– connect categories using SubsetOf links,

Semantic Network Example



A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

Semantic Networks

Allows for inheritance reasoning

– Female persons inherit all properties from person

– Mary inherits the property of having two legs

• The simplicity and efficiency of this inference mechanism compared with logical theorem has been one of the main attractions of semantic networks.

• Multiple Inheritance becomes complicated because two or more conflicting values for answering the query

• For this reason, multiple inheritance is banned in some object-oriented programming (OOP) languages, such as Java

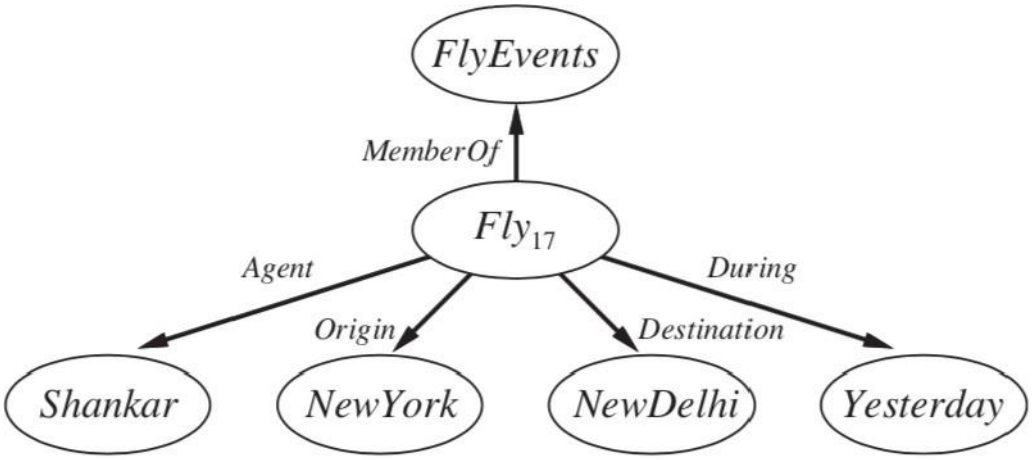
Another form of inference is the use of inverse links

• Example: HasSister is the inverse of SisterOf

• Drawback of semantic network is that the links between bubbles represent only *binary* relations.

• For example, the sentence Fly(Shankar, NewYork, NewDelhi, Yesterday) cannot be asserted directly in a semantic network.

• *But can* obtain the effect of n-ary assertions by reifying the proposition



Semantic network showing representation of logical assertion fly()

Ability to override the default values

• Example:

– John has 1 leg despite the fact that all persons have 2 legs

– This would be contradiction in a strictly logical KB.

<u>Description logics</u>

They are logical notations that are designed to describe definitions and properties about categories

• It is to formalize the semantic network

• Principal inference task is !

– Subsumption: checking if one category is the subset of another by comparing their definitions

– Classification: checking whether an object belongs to a category.

– Consistency: whether the category membership criteria are logically satisfiable

The CLASSIC language is a typical description logic

• Any CLASSIC can be written in FOL

• For example, to say that bachelors are unmarried adult males we would write

• Bachelor = And (Unmarried , Adult , Male)

• The equivalent in first-order logic would be

• Bachelor(x) ⇔ Unmarried(x)∧Adult(x)∧Male(x)

The syntax of descriptions in a subset of the CLASSIC language

$$
\begin{array}{rcl}
Concept & \rightarrow & \textbf{Thing} \mid ConceptName \\
& \mid & \textbf{And}(Concept, \ldots) \\
& \mid & \textbf{All}(RoleName, Concept) \\
& \mid & \textbf{AtLeast}(Integer, RoleName) \\
& \mid & \textbf{AtMost}(Integer, RoleName) \\
& \mid & \textbf{Fills}(RoleName, IndividualName, \ldots) \\
& \mid & \textbf{SameAs}(Path, Path) \\
& \mid & \textbf{OneOf}(IndividualName, \ldots) \\
Path & \rightarrow & [RoleName, \ldots]
\end{array}
$$

Reasoning with Default Information

Default reasoning

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

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

We will discuss two approaches to do this:

- Non-Monotonic logic.
- Default logic.

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

Non-Monotonic Logic

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

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

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

How do we define *consistency*?

One common solution (consistent with PROLOG notation) is

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

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

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

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

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

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

Quaker(Nixon)

Republican(Nixon)

This now leads to our total knowledge becoming inconsistent.

Default Logic

Default logic introduces a new inference rule:

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

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

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

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

Circumscription

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

Circumscription can also cope with default reasoning.

Suppose we know: bird(tweety)

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

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

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

In circumscription this phrase would be stated as:

A bird will fly if it is not abnormal

and can thus be represented by:

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

However, this is not sufficient

We cannot conclude

flies(tweety)

since we cannot prove

$\neg$abnormal(tweety).

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

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

Thus we can rewrite our *default rule* as:

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

and add the following

$\forall x$: $\neg$abnormal(x)

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

If we now add the fact:

penguin(tweety)

Clearly we can prove

abnormal(tweety).

If we circumscribe abnormal now we would add the sentence,

a penguin (tweety) is the abnormal thing:

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

Note the distinction between Default logic and circumscription:

Defaults are sentences in language itself not additional inference rules.

Implementations: Truth Maintenance Systems

Due to Lecture time limitation. This topic is not dealt with in any great depth. Please refer to the further reading section.

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

Basically TMSs:

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

Justification-Based Truth Maintenance Systems (JTMS)

- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- Each supported belief (assertion) in has a justification.
- Each justification has two parts:
    - An *IN-List* -- which supports beliefs held.
    - An *OUT-List* -- which supports beliefs *not* held.

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

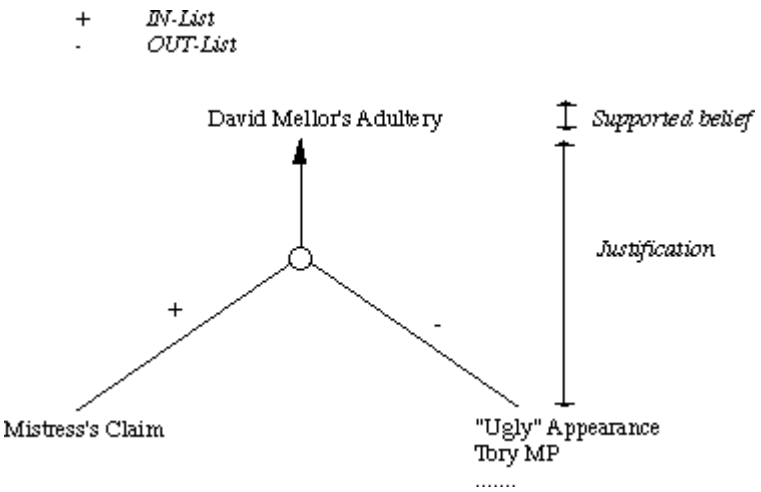


Fig. 20 A JTMS Assertion

Logic-Based Truth Maintenance Systems (LTMS)

Similar to JTMS except:

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

Assumption-Based Truth Maintenance Systems (ATMS)

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

Quantifying Uncertainty

Acting Under Uncertainty

Agents often make decisions based on incomplete information
-partial observability
-nondeterministic actions
Partial solution (see previous chapters): maintain belief states
-represent the set of all possible world states the agent might be in
-generating a contingency plan handling every possible eventuality
Several drawbacks:
-must consider every possible explanation for the observation (even very-unlikely ones) =) impossibly complex belief-states

-contingent plans handling every eventuality grow arbitrarily large
-sometimes there is no plan that is guaranteed to achieve the goal
-Agent's knowledge cannot guarantee a successful outcome..... but can provide some degree of belief (likelihood) on it
-A rational decision depends on both the relative importance of (sub)goals and the likelihood that they will be achieved
-Probability theory offers a clean way to quantify likelihood

Acting Under Uncertainty: Example

Automated taxi to Airport
-Goal: deliver a passenger to the airport on time
-Action A_t : leave for airport t minutes before flight
How can we be sure that A_{90} will succeed?
-Too many sources of uncertainty:
- partial observability (ex: road state, other drivers' plans, etc.)
- uncertainty in action outcome (ex: flat tire, etc.)
- noisy sensors (ex: unreliable traffic reports)
- complexity of modelling and predicting traffic
=) With purely-logical approach it is difficult to anticipate everything that can go wrong
-risks falsehood: "A_{25} will get me there on time" or
- leads to conclusions that are too weak for decision making:
"A_{25} will get me there on time if there's no accident on the bridge , and it doesn't rain and my tires remain intact , and. "
-Over-cautious choices are not rational solutions either
ex: A_{1440} causes staying overnight at the airport

Acting Under Uncertainty: Example (2)

A medical diagnosis
-Given the symptoms (toothache) infer the cause (cavity)
-How to encode this relation in logic?
diagnostic rules:
Toothache ! Cavity (wrong)
Toothache ! (Cavity _ GumProblem _ Abscess _ :::)
(too many possible causes, some very unlikely)
causal rules:
Cavity ! Toothache (wrong)
(Cavity ^ :::) ! Toothache (many possible (con)causes)
Problems in specifying the correct logical rules:
Complexity: too many possible antecedents or consequents
Theoretical ignorance: no complete theory for the domain
Practical ignorance: no complete knowledge of the patient

Summarizing Uncertainty

Probability allows to summarize the uncertainty on effects of
laziness: failure to enumerate exceptions, qualifications, etc.
ignorance: lack of relevant facts, initial conditions, etc.
Probability can be derived from
statistical data (ex: 80% of toothache patients so far had cavities)

some knowledge (ex: 80% of toothache patients has cavities) their combination thereof

Probability statements are made with respect to a state of knowledge (aka evidence), not with respect to the real world

e.g., "The probability that the patient has a cavity, given that she has a toothache, is 0.8":
P(HasCavity(patient) j hasToothAche(patient)) = 0:8

Probabilities of propositions change with new evidence:

"The probability that the patient has a cavity, given that she has a toothache and a history of gum disease, is 0.4":
P(HasCavity(patient)
j hasToothAche(patient) ^ HistoryOfGum(patient)) = 0:4

Making Decisions Under Uncertainty

Ex: Suppose I believe:

P(A_{25} gets me there on time j:::) = 0:04
P(A_{90} gets me there on time j:::) = 0:70
P(A_{120} gets me there on time j:::) = 0:95
P(A_{1440} gets me there on time j:::) = 0:9999
Which action to choose?

=) Depends on tradeoffs among preferences:

missing flight vs. costs (airport cuisine, sleep overnight in airport)

When there are conflicting goals the agent may express preferences among them by means of a utility function.

Utilities are combined with probabilities in the general theory of rational decisions, aka decision theory:

Decision theory = Probability theory + Utility theory

Maximum Expected Utility (MEU): an agent is rational if and only if it chooses the action that yields the maximum expected utility, averaged over all the possible outcomes of the action.

Probabilities Basics:

- Probabilistic assertions: state how likely possible worlds are
- Sample space Ω: the set of all possible worlds
  - $\omega \in \Omega$ is a possible world (aka sample point or atomic event)
  - ex: the dice roll (1,4)
  - the possible worlds are mutually exclusive and exhaustive
  - ex: the 36 possible outcomes of rolling two dice: (1,1), (1,2), ...
- A probability model (aka probability space) is a sample space with an assignment $P(\omega)$ for every $\omega \in \Omega$ s.t.
  - $0 \leq P(\omega) \leq 1$, for every $\omega \in \Omega$
  - $\Sigma_{\omega \in \Omega} P(\omega) = 1$
- Ex: 1-die roll: $P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6$
- An Event A is any subset of Ω, s.t. $P(A) = \Sigma_{\omega \in A} P(\omega)$
  - events can be described by propositions in some formal language
  - ex: $P(Total = 11) = P(5,6) + P(6,5) = 1/36 + 1/36 = 1/18$
  - ex: $P(doubles) = P(1,1) + P(2,2) + ... + P(6,6) = 6/36 = 1/6$

Random Variables

- Factored representation of possible worlds: sets of $\langle variable, value \rangle$ pairs
- Variables in probability theory: Random variables
  - domain: the set of possible values a variable can take on
    ex: Die: $\{1,2,3,4,5,6\}$, Weather: $\{sunny, rain, cloudy, snow\}$, Odd: $\{true, false\}$,
  - a r.v. can be seen as a function from sample points to the domain:
    ex: $Die(\omega)$, $Weather(\omega)$,... ("(ω)" typically omitted)
- Probability Distribution gives the probabilities of all the possible values of a random variable X: $P(X = x_i) \stackrel{\text{def}}{=} \Sigma_{\omega \in X(\omega)} P(\omega)$
  - ex: $P(Odd = true) = P(1) + P(3) + P(5) = 1/6 + 1/6 + 1/6 = 1/2$

Propositions and Probabilities

- We think a proposition a as the event A (set of sample points) where the proposition is true
  - Odd is a propositional random variable of range $\{true, false\}$
  - notation: $a \Longleftrightarrow$ "$A = true$"
- Given Boolean random variables A and B:
  - event a: set of sample points where $A(\omega) = true$
  - event $\neg a$: set of sample points where $A(\omega) = false$
  - event $a \wedge b$: set of sample points where $A(\omega) = true$, $B(\omega) = true$

$\Longrightarrow$ with Boolean random variables, sample points are PL models

- Proposition: disjunction of the sample points in which it is true
  - ex: $(a \vee b) \equiv (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$

  $\Longrightarrow$ $P(a \vee b) = P(\neg a \wedge b) + P(a \wedge \neg b) + P(a \wedge b)$
- Some derived facts:
  - $P(\neg a) = 1 - P(a)$
  - $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$

Probability Distributions

- Probability Distribution gives the probabilities of all the possible values of a random variable
  - ex: Weather: $\{sunny, rain, cloudy, snow\}$

  $\Longrightarrow \mathbf{P}(Weather) = (0.6, 0.1, 0.29, 0.01) \Longleftrightarrow$

  $$\left\{ \begin{array}{ll} P(Weather = sunny) & = 0.6 \\ P(Weather = rain) & = 0.1 \\ P(Weather = cloudy) & = 0.29 \\ P(Weather = snow) & = 0.01 \end{array} \right\}$$

  - normalized: their sum is 1
- Joint Probability Distribution for multiple variables
  - gives the probability of every sample point
  - ex: $\mathbf{P}(Weather, Cavity) =$

$Weather =$	$sunny$	$rain$	$cloudy$	$snow$
$Cavity = true$	0.144	0.02	0.016	0.02
$Cavity = false$	0.576	0.08	0.064	0.08

- Every event is a sum of sample points,

  $\Longrightarrow$ its probability is determined by the joint distribution

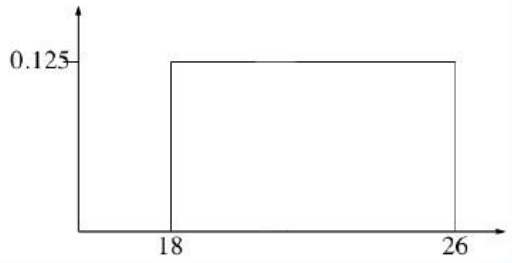
Probability for Continuous Variables

- Express continuous probability distributions:
  - density functions $f(x) \in [0, 1]$ s.t $\int_{-\infty}^{+\infty} f(x)dx = 1$
- $P(x \in [a, b]) = \int_a^b f(x)\ dx$

  $\Longrightarrow P(x \in [val, val]) = 0$, $P(x \in [-\infty, +\infty]) = 1$
  - ex: $P(x \in [20, 22]) = \int_{20}^{22} 0.125\ dx = 0.25$
- Density: $P(x) = P(X = x) \stackrel{def}{=} \lim_{dx \mapsto 0} P(X \in [x, x + dx])/dx$
  - ex: $P(20.1) = \lim_{dx \mapsto 0} P(X \in [20.1, 20.1 + dx])/dx = 0.125$
  - note: $P(v) \neq P(x \in [v, v]) = 0$

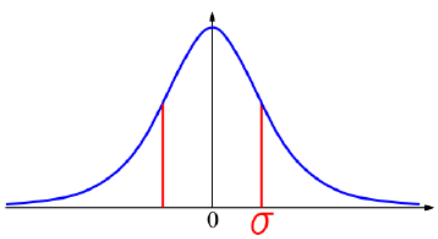
Uniform density between 18 and 26

$f(x) = U[18, 26](x)$



Gaussian density

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



Conditional Probabilities

- Unconditional or prior probabilities refer to degrees of belief in propositions in the absence of any other information (evidence)
  - ex: $P(cavity) = 0.2$, $P(Total = 11) = 1/18$, $P(double) = 1/6$
- Conditional or posterior probabilities refer to degrees of belief in proposition a given some evidence b: P(a|b)
  - evidence: information already revealed
  - ex: $P(cavity|toothache) = 0.6$: p. of a cavity given a toothache (assuming no other information is provided!)
  - ex: $P(Total = 11|die_1 = 5) = 1/6$: p. of total 11 given first die is 5

  $\Longrightarrow$ restricts the set of possible worlds to those where the first die is 5
- Note: $P(a|... \wedge a) = 1$, $P(a|... \wedge \neg a) = 0$
  - ex: $P(cavity|toothache \wedge cavity) = 1$, $P(cavity|toothache \wedge \neg cavity) = 0$
- Less specific belief still valid after more evidence arrives
  - ex: $P(cavity) = 0.2$ holds even if $P(cavity|toothache) = 0.6$
- New evidence may be irrelevant, allowing for simplification
  - ex: $P(cavity|toothache, 49ersWin) = P(cavity|toothache) = 0.8$

- Conditional probability: $P(a|b) \stackrel{\text{def}}{=} \frac{P(a \wedge b)}{P(b)}$, s.t. $P(b) > 0$
  - ex: $P(Total = 11|die_1 = 5) = \frac{P(Total=11 \wedge die_1=5)}{P(die_1=5)} = \frac{1/6 \cdot 1/6}{1/6} = 1/6$
  - observing b restricts the possible worlds to those where b is true
- Production rule: $P(a \wedge b) = P(a|b) \cdot P(b) = P(b|a) \cdot P(a)$
- Production rule for whole distributions: $\mathbf{P}(X, Y) = \mathbf{P}(X|Y) \cdot \mathbf{P}(Y)$
  - ex: $\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather|Cavity)\mathbf{P}(Cavity)$, that is:

    $P(sunny, cavity) = P(sunny|cavity)P(cavity)$

    ...

    $P(snow, \neg cavity) = P(snow|\neg cavity)P(\neg cavity)$
  - a 4×2 set of equations, not matrix multiplication!
- Chain rule is derived by successive application of product rule:

  $\mathbf{P}(X_1, ..., X_n)$

  $= \mathbf{P}(X_1, ..., X_{n-1})\mathbf{P}(X_n|X_1, ..., X_{n-1})$

  $= \mathbf{P}(X_1, ..., X_{n-2})\mathbf{P}(X_{n-1}|X_1, ..., X_{n-2})\mathbf{P}(X_n|X_1, ..., X_{n-1})$

  $= ...$

  $= \prod_{i=1}^{n} \mathbf{P}(X_i|X_1, ..., X_{i-1})$

Logic vs. Probability

Logic	*Probability*
a	$P(a) = 1$
$\neg a$	$P(a) = 0$
$a \to b$	$P(b\|a) = 1$
$\dfrac{(a, a \to b)}{b}$	$\dfrac{P(a) = 1, P(b\|a) = 1}{P(b) = 1}$
$\dfrac{(a \to b, b \to c)}{a \to c}$	$\dfrac{P(b\|a) = 1, P(c\|b) = 1}{P(c\|a) = 1}$

- Proof of $P(b|a) = 1, P(c|b) = 1 \Longrightarrow P(c|a) = 1$
  - $P(b|a) = 1 \Longrightarrow P(\neg b, a) \stackrel{\text{def}}{=} P(\neg b|a)P(a) = 0$
  - $P(c|b) = 1 \Longrightarrow P(\neg c, b) \stackrel{\text{def}}{=} P(\neg c|b)P(b) = 0$
  - $P(\neg c, a) = P(\neg c, a, b) + P(\neg c, a, \neg b) \leq \underbrace{P(\neg c, b)}_{0} + \underbrace{P(a, \neg b)}_{0} = 0$
  - $P(\neg c|a) = P(\neg c, a)/P(a) = 0$
  - $P(c|a) = 1 - P(\neg c|a) = 1$

Probabilistic Inference via Enumeration

Basic Ideas

- Start with the joint distribution **P**(*Toothache*, *Catch*, *Cavity*)
- For any proposition φ, sum the atomic events where φ is true: $P(\varphi) = \Sigma_{\omega\,:\,\omega \models \varphi} P(\omega)$

Example: Generic Inference

- Start with the joint distribution $\mathbf{P}(\mathit{Toothache}, \mathit{Catch}, \mathit{Cavity})$
- For any proposition φ, sum the atomic events where φ is true: $P(\varphi) = \Sigma_{\omega\,:\,\omega \models \varphi} P(\omega)$:
- Ex: $P(\mathit{cavity} \vee \mathit{toothache}) =$
  $0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

Marginalization

- Start with the joint distribution $\mathbf{P}(\mathit{Toothache}, \mathit{Catch}, \mathit{Cavity})$
- Marginalization (aka summing out): sum up the probabilities for each possible value of the other variables:
  $\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y}, \mathbf{z})$
  Ex: $\mathbf{P}(\mathit{Toothache}) = \sum_{\mathbf{z} \in \{\mathit{Catch}, \mathit{Cavity}\}} \mathbf{P}(\mathit{Toothache}, \mathbf{z})$
- Conditioning: variant of marginalization, involving conditional probabilities instead of joint probabilities (using the product rule)
  $\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y} | \mathbf{z}) P(\mathbf{z})$
  Ex: $\mathbf{P}(\mathit{Toothache}) = \sum_{\mathbf{z} \in \{\mathit{Catch}, \mathit{Cavity}\}} \mathbf{P}(\mathit{Toothache} | \mathbf{z}) P(\mathbf{z})$

Marginalization: Example

- Start with the joint distribution $\mathbf{P}(Toothache, Catch, Cavity)$
- Marginalization (aka summing out): sum up the probabilities for each possible value of the other variables:

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

  Ex: $\mathbf{P}(Toothache) = \sum_{\mathbf{z} \in \{Catch, Cavity\}} \mathbf{P}(Toothache, \mathbf{z})$

  $P(toothache) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2$

  $P(\neg toothache) = 1 - P(toothache) = 1 - 0.2 = 0.8$

$\Longrightarrow$ $\mathbf{P}(Toothache) = \langle 0.2, 0.8 \rangle$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

Conditional Probability via Enumeration: Example

- Start with the joint distribution $\mathbf{P}(Toothache, Catch, Cavity)$
- Conditional Probability:

  Ex: $P(\neg cavity | toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)}$

  $= \frac{0.016+0.064}{0.108+0.012+0.016+0.064} = 0.4$

  Ex: $P(cavity | toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)} = \ldots = 0.6$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

Normalization

- Let **X** be all the variables. Typically, we want $\mathbf{P}(\mathbf{Y}|\mathbf{E}=\mathbf{e})$:
  - the conditional joint distribution of the query variables **Y**
  - given specific values **e** for the evidence variables **E**
  - let the hidden variables be $\mathbf{H} \stackrel{\text{def}}{=} \mathbf{X} \setminus (\mathbf{Y} \cup \mathbf{E})$
- The summation of joint entries is done by summing out the hidden variables:
  $\mathbf{P}(Y|\mathbf{E}=\mathbf{e}) = \alpha\mathbf{P}(Y,\mathbf{E}=\mathbf{e}) = \alpha\Sigma_{\mathbf{h}\in\mathbf{H}}\mathbf{P}(Y,\mathbf{E}=\mathbf{e},\mathbf{H}=\mathbf{h})$

  where $\alpha \stackrel{\text{def}}{=} 1/\mathbf{P}(\mathbf{E}=\mathbf{e})$ (different α's for different values of **e**)

  $\Longrightarrow$ it is easy to compute α by normalization
  - note: the terms in the summation are joint entries, because **Y**, **E**, **H** together exhaust the set of random variables **X**
- Complexity: $O(2^n)$, n number of propositions $\Longrightarrow$ impractical

Normalization: Example

- $\alpha \stackrel{\text{def}}{=} 1/P(toothache)$ can be viewed as a normalization constant
- Idea: compute whole distribution on query variable by:
  - fixing evidence variables and summing over hidden variables
  - normalize the final distribution, so that $\sum ... = 1$

Ex:

$\mathbf{P}(Cavity|toothache) = \alpha\mathbf{P}(Cavity \wedge toothache)$
$= \alpha[\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)]$
$= \alpha[\langle 0.108, 0.016\rangle + \langle 0.012, 0.064\rangle]$
$= \alpha\langle 0.12, 0.08\rangle = (normalization) = \langle 0.6, 0.4\rangle$ $[\alpha = 5]$
$\mathbf{P}(Cavity|\neg toothache) = ... = \alpha\langle 0.08, 0.72\rangle = \langle 0.1, 0.9\rangle [\alpha = 1.25]$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

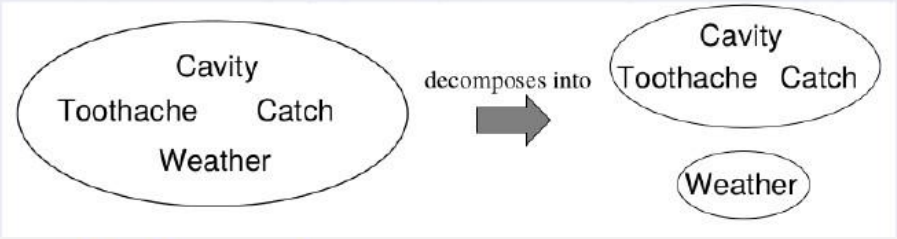
Independence

- Variables X and Y are independent iff $\mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y)$ (or equivalently, iff $\mathbf{P}(X|Y) = \mathbf{P}(X)$ or $\mathbf{P}(Y|X) = \mathbf{P}(Y)$)
  - ex: $\mathbf{P}(Toothache, Catch, Cavity, Weather) = \mathbf{P}(Toothache, Catch, Cavity)\mathbf{P}(Weather)$

  $\Longrightarrow$ e.g. $P(toothache, catch, cavity, cloudy) = P(toothache, catch, cavity)P(cloudy)$

  - typically based on domain knowledge
- May drastically reduce the number of entries and computation

  $\Longrightarrow$ ex: 32-element table decomposed into one 8-element and one 4-element table
- Unfortunately, absolute independence is quite rare



Conditional Independence

- Variables X and Y are conditionally independent given **Z** iff $\mathbf{P}(X, Y|\mathbf{Z}) = \mathbf{P}(X|\mathbf{Z})\mathbf{P}(Y|\mathbf{Z})$ (or equivalently, iff $\mathbf{P}(X|Y, \mathbf{Z}) = \mathbf{P}(X|\mathbf{Z})$ or $\mathbf{P}(Y|X, \mathbf{Z}) = \mathbf{P}(Y|\mathbf{Z})$)
- Consider $\mathbf{P}(Toothache, Cavity, Catch)$
  - if I have a cavity, the probability that the probe catches in it doesn't depend on whether I have a toothache: $P(catch|toothache, cavity) = P(catch|cavity)$
  - the same independence holds if I haven't got a cavity: $P(catch|toothache, \neg cavity) = P(catch|\neg cavity)$

  $\Longrightarrow$ Catch is conditionally independent of Toothache given Cavity: $\mathbf{P}(Catch|Toothache, Cavity) = \mathbf{P}(Catch|Cavity)$

  or, equivalently: $\mathbf{P}(Toothache|Catch, Cavity) = \mathbf{P}(Toothache|Cavity)$, or $\mathbf{P}(Toothache, Catch|Cavity) = \mathbf{P}(Toothache|Cavity)P(Catch|Cavity)$

Conditional Independence [cont.]

- In many cases, the use of conditional independence reduces the size of the representation of the joint distribution dramatically
  - even from exponential to linear!
- Ex:

$$\begin{aligned}&\mathbf{P}(Toothache, Catch, Cavity)\\ &= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch, Cavity)\\ &= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)\\ &= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)\end{aligned}$$

$\Longrightarrow$ Passes from 7 to 2+2+1=5 independent numbers

- $\mathbf{P}(Toothache, Catch, Cavity)$ contains 7 independent entries (the 8th can be obtained as $1 - \sum ...$)
- $\mathbf{P}(Toothache|Cavity)$,$\mathbf{P}(Catch|Cavity)$ contain 2 independent entries (2×2 matrix, each row sums to 1)
- $\mathbf{P}(Cavity)$ contains 1 independent entry

- General Case: if one causes has n independent effects:
  $\mathbf{P}(Cause, Effect_1, ..., Effect_n) = \mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i|Cause)$
  $\Longrightarrow$ reduces from $2^{n+1} - 1$ to $2n + 1$ independent entries

Exercise

Consider the joint probability distribution described in the table in previous section (slide 20 onwards): $\mathbf{P}(Toothache, Catch, Cavity)$

- Consider the example in previous slide:
  $\mathbf{P}(Toothache, Catch, Cavity)$
  $= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch, Cavity)$
  $= \mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)$
  $= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)$
- Compute separately the distributions $\mathbf{P}(Toothache|Catch, Cavity)$, $\mathbf{P}(Catch|Cavity)$, $\mathbf{P}(Cavity)$, $\mathbf{P}(Toothache|Cavity)$.
- Recompute $\mathbf{P}(Toothache, Catch, Cavity)$ in two ways:
  - $\mathbf{P}(Toothache|Catch, Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)$
  - $\mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)$

  and compare the result with $\mathbf{P}(Toothache, Catch, Cavity)$

Bayes' Rule

Bayes' Rule/Theorem/Law

- Bayes' rule: $P(a|b) = \frac{P(a \wedge b)}{P(b)} = \frac{P(b|a)P(a)}{P(b)}$
- In distribution form $\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y)$
  - $\alpha \stackrel{\text{def}}{=} 1/\mathbf{P}(X)$: normalization constant to make $\mathbf{P}(Y|X)$ entries sum to 1 (different α's for different values of X)
- A version conditionalized on some background evidence $\mathbf{e}$: $\mathbf{P}(Y|X,\mathbf{e}) = \frac{\mathbf{P}(X|Y,\mathbf{e})\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})}$

Using Bayes' Rule: The Simple Case

- Used to assess diagnostic probability from causal probability: $P(cause|effect) = \frac{P(effect|cause)P(cause)}{P(effect)}$
  - $P(cause|effect)$ goes from effect to cause (diagnostic direction)
  - $P(effect|cause)$ goes from cause to effect (causal direction)

Example

- An expert doctor is likely to have causal knowledge ...
  $P(symptoms|disease)$ (i.e., $P(effect|cause)$)
  ... and needs producing diagnostic knowledge
  $P(disease|symptoms)$ (i.e., $P(cause|effect)$)
- Ex: let m be meningitis, s be stiff neck
  - $P(m) = 1/50000$, $P(s) = 0.01$ (prior knowledge, from statistics)
  - "meningitis causes to the patient a stiff neck in 70% of cases": $P(s|m) = 0.7$ (doctor's experience)

$$\Longrightarrow P(m|s) = \frac{P(s|m)P(m)}{P(s)} = \frac{0.7 \cdot 1/50000}{0.01} = 0.0014$$

Using Bayes' Rule: Combining Evidence

- A naive Bayes model is a probability model that assumes the effects are conditionally independent, given the cause
  $\Longrightarrow$ $\mathbf{P}(Cause, Effect_1, ..., Effect_n) = \mathbf{P}(Cause) \prod_i \mathbf{P}(Effect_i|Cause)$
  - total number of parameters is linear in n
  - ex: $\mathbf{P}(Cavity, Toothache, Catch) = \mathbf{P}(Cavity)\mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)$

Q: How can we compute $\mathbf{P}(Cause|Effect_1, ..., Effect_k)$?
- ex $\mathbf{P}(Cavity|toothache \wedge catch)$?

Using Bayes' Rule: Combining Evidence [cont.]

Q: How can we compute $\mathbf{P}(Cause|Effect_1, ..., Effect_k)$?
- ex: $\mathbf{P}(Cavity|toothache \wedge catch)$?

A: Apply Bayes' Rule

$$\begin{aligned}
&\mathbf{P}(Cavity|toothache \wedge catch) \\
&= \mathbf{P}(toothache \wedge catch|Cavity)\mathbf{P}(Cavity)/P(toothache \wedge catch) \\
&= \alpha\mathbf{P}(toothache \wedge catch|Cavity)\mathbf{P}(Cavity) \\
&= \alpha\mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)\mathbf{P}(Cavity)
\end{aligned}$$

- $\alpha \stackrel{\text{def}}{=} 1/P(toothache \wedge catch)$ not computed explicitly

- General case:
  $\mathbf{P}(Cause|Effect_1, ..., Effect_n) = \alpha\mathbf{P}(Cause) \prod_i \mathbf{P}(Effect_i|Cause)$
  - $\alpha \stackrel{\text{def}}{=} 1/\mathbf{P}(Effect_1, ..., Effect_n)$ not computed explicitly (one α value for every value of $Effect_1, ..., Effect_n$)

  $\Longrightarrow$ reduces from $2^{n+1} - 1$ to $2n + 1$ independent entries

Artificial Intelligence (BEIT703T)
VII Semester IT
Unit-III
Topic:- Knowledge Representation with Logic and Inferences
Prof. S. C. Sakure

(1) What are the steps associated with the knowledge Engineering process?

Discuss them by applying the steps to any real world application of your choice.

Knowledge Engineering

The general process of constructing knowledge base is called knowledge engineering. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We will illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar,

The steps associated with the knowledge engineering process are :

1 Identify the task.

The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents.

2. Assemble the relevant knowledge. The knowledge engineer might already be an expert in the domain, **or** might need to work with real experts to extract what they know-a process called **knowledge acquisition.**

3. **Decide on a vocabulary of predicates, functions, and constants**. That is, translate the important domain-level concepts into logic-level names. Once the choices have been made. the result is a vocabulary that is known as the **ontology** of the domain. The word ontology means a particular theory of the nature of being or existence.

4. **Encode general knowledge about the domain**. The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. Encode a description of the specific problem instance

For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. **Pose queries to the inference procedure and get answers.** This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.

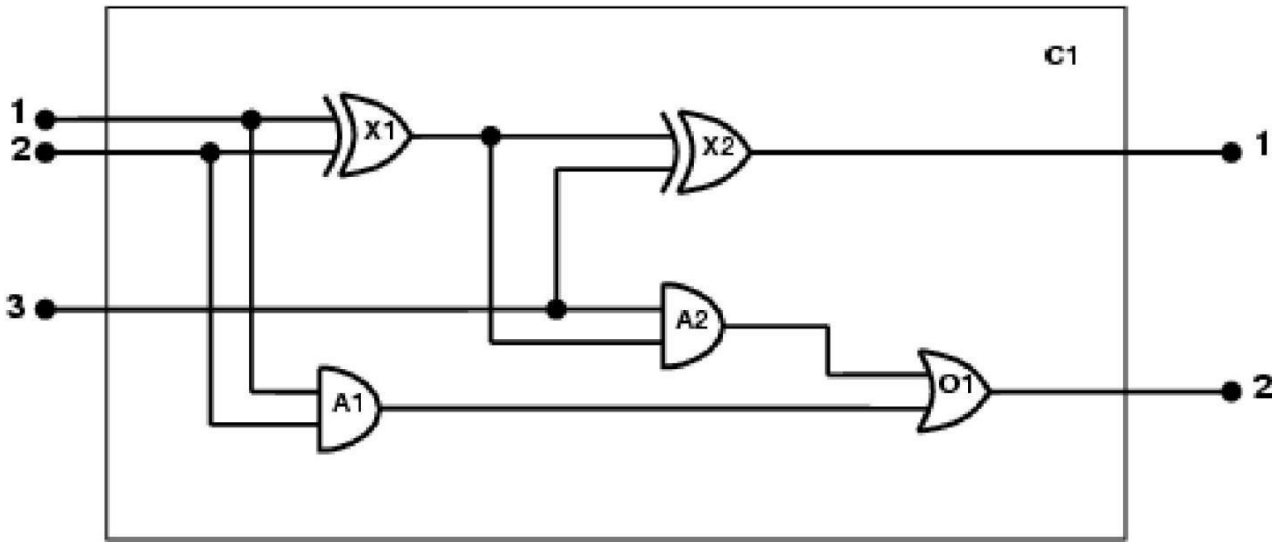
7. Debug the knowledge base.

x NumOfLegs(x,4) => Mammal(x) Is false for reptiles ,amphibians.

To understand this seven-step process better, we now apply it to an extended example-the domain of electronic circuits.

The electronic circuits domain

One bit adder



1 Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section.

2 Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire.

3 Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants: *X I , X2,* and so on

Type(X_1) = XOR
Type(X_1, XOR) –binary predicate
XOR(X_1) –individual type

Agate or circuit can have one or more terminal. For X_1 the terminals are X_1In_1, X_1In_2, X_1Out1

X_1In_1- 1st input of gate X1
X_1In_2 -2nd input of gate X1
X_1Out_1.Output of gate X1

4 Encode general knowledge of the domain

One sign that we have a good ontology is that there are very few general rules which need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules to describe everything we need to know about circuits:

1. If two terminals are connected, then they have the same signal:

   $\forall t_1,t_2$ Connected(t_1, t_2) $\Rightarrow$ Signal(t_1) = Signal(t_2)

2. The signal at every terminal is either 1 or 0 (but not both):

$$\forall t \text{ Signal}(t) = 1 \Leftrightarrow \text{Signal}(t) = 0$$

$$1 \neq 0$$

3. Connected is a commutative predicate:

$$\forall t_1,t_2 \text{ Connected}(t_1, t_2) \Leftrightarrow \text{Connected}(t_2, t_1)$$

4. An OR gate's output is 1 if and only if any of its inputs is 1:

$$\forall g \text{ Type}(g) = \text{OR} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 1 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n,g)) = 1$$

5. An **A.ND** gate's output is 0 if and only if any of its inputs is 0:

$$\forall g \text{ Type}(g) = \text{AND} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 0 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n,g)) = 0$$

6. An XOR gate's output is 1 if and only if its inputs are different:

$$\forall g \text{ Type}(g) = \text{XOR} \Rightarrow \text{Signal}(\text{Out}(1,g)) = 1 \Leftrightarrow \text{Signal}(\text{In}(1,g)) \neq \text{Signal}(\text{In}(2,g))$$

7. A NOT gate's output is different from its input:

$$\forall g \text{ Type}(g) = \text{NOT} \Rightarrow \text{Signal}(\text{Out}(1,g)) \neq \text{Signal}(\text{In}(1,g))$$

5 Encode the specific problem instance

The circuit shown in Figure is encoded as circuit *C1* with the following description. First,we categorize the gates:

$$\text{Type}(X1) = \text{XOR}\ \text{Type}(X2) = \text{XOR}$$

$\text{Type}(X_1) = \text{XOR}$ $\text{Type}(X_2) = \text{XOR}$
$\text{Type}(A_1) = \text{AND}$ $\text{Type}(A_2) = \text{AND}$
$\text{Type}(O_1) = \text{OR}$
$\text{Connected}(\text{Out}(1,X_1),\text{In}(1,X_2))$ $\text{Connected}(\text{In}(1,C_1),\text{In}(1,X_1))$
$\text{Connected}(\text{Out}(1,X_1),\text{In}(2,A_2))$ $\text{Connected}(\text{In}(1,C_1),\text{In}(1,A_1))$
$\text{Connected}(\text{Out}(1,A_2),\text{In}(1,O_1))$ $\text{Connected}(\text{In}(2,C_1),\text{In}(2,X_1))$
$\text{Connected}(\text{Out}(1,A_1),\text{In}(2,O_1))$ $\text{Connected}(\text{In}(2,C_1),\text{In}(2,A_1))$
$\text{Connected}(\text{Out}(1,X_2),\text{Out}(1,C_1))$ $\text{Connected}(\text{In}(3,C_1),\text{In}(2,X_2))$
$\text{Connected}(\text{Out}(1,O_1),\text{Out}(2,C_1))$ $\text{Connected}(\text{In}(3,C_1),\text{In}(1,A_2))$

6 Pose queries to the inference procedure

What combinations of inputs would cause the first output of Cl (the sum bit) to be 0 and the second output of C1 (the carry bit) to be l?

$$\exists i_1,i_2,i_3,o_1,o_2 \text{ Signal}(\text{In}(1,C1)) = i_1 \wedge \text{Signal}(\text{In}(2,C_1)) = i_2 \wedge \text{Signal}(\text{In}(3,C_1)) = i_3 \wedge \text{Signal}(\text{Out}(1,C_1)) = 0 \wedge \text{Signal}(\text{Out}(2,C_1)) = 1$$

The answer are substitutions for the variable i1,i2 and i3 such that the resulting sentence is entailed by the knowledge base, There are three such substitutions:

{i1/1,i2/1,i3/0} {i1/2,i2/0,i3/1} {i1/0,i2/1,i3/1}

What are the possible sets of values of all the terminals for the adder circuit?

$$\exists\, i_1, i_2, i_3, o_1, o_2 \;\; Signal(In(1, C_1)) = i \wedge Signal(In(2,CO) = i_2 \wedge Signal(In(3,_1) = i_3 \wedge Signal(Out(1,CO) = 01 \wedge Signal(Out(2,CO) = 02$$

This final query will return a complete input/output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification.**

7 Debug the knowledge base

The knowledge base is checked with different constraints. For example if the assertion 1#0 is not included in the knowledge base then it is variable to prove any output for the circuit ,expect for the input cases 000 and 110

$\exists i_1,i_2,o$ Signal(In(1,C1)) = i_1 $\wedge$ Signal(In(2,C_1)) = i_2 $\wedge$ Signal(Out(1,X_1))

This claim that no output are known at X1 for the input cases 10 and 01. Therefore the axiom for XOR gate is considered

Signal(Out(1,X_1)) = 1 $\Leftrightarrow$ Signal(In(1,X_1)) $\neq$ Signal(In(2,X_1))

It the input signal are known 1 and 0 , the above sentence reduces to Signal(Out(1,X_1)) = 1 $\Leftrightarrow$ 1#0

Now the system is unable to infer the Signal(Out(1,X_1)) =1

Unification

Generalized Modus Pones

- For atomic sentences p_i, p_i', q where there is a substitution q such that Subst{q, p_i'} = Subst{q, p_i}:

$$\frac{p_1', \quad p_2', \quad \ldots, \quad p_n', \quad (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

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

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

Unification

- It is the process of finding substitutions that make two atomic sentences identical
- Lifted inference rules require finding substitution that make different logical expression look identical this process is called unification
- It is the key component of first order inference logic algorithm

  UNIFY (p,q)=θ Where SUBST(θ,p)= SUBST(q, θ)

  Θ- Unifier of two sentences

  P-S1(x,x)

  1(y,z)

  Assume θ=y

  Unification algorithm returns failure as the result at two factors

  i) Two sentence have different predicate name

  Ex hate(m,c)

  try(m,c)

  ii) Two sentence will have different number of arguments

  Ex hate(m,c,x)

  hate(m,c)

Query Knows (John,x) whom does john knows

The knowledg e base contain the following information

Knows(John,x)

Knows(John,Jane)

Knows(y,Bill)

Knows(y,Mother(y))

Knows(x,Eizabeth)

$$\text{UNIFY}(Knows(John, x),\ Knows(John, Jane)) = \{x/Jane\}$$
$$\text{UNIFY}(Knows(John, x),\ Knows(y, Bill)) = \{x/Bill, y/John\}$$
$$\text{UNIFY}(Knows(John, x),\ Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}$$
$$\text{UNIFY}(Knows(John, x),\ Knows(x, Elizabeth)) = fail \; .$$

Consider the last sentence:

$$\text{UNIFY}(Knows(John, x),\ Knows(x, Elizabeth)) = fail$$

- This fails because *x* cannot take on two values
- But "Everyone knows Elizabeth" and it should not fail
- Must standardize apart one of the two sentences to eliminate reuse of variable

$$\text{UNIFY}(Knows(John, x),\ Knows(z_{17}, Elizabeth)) = \{x/Elizabeth, z_{17}/John\}$$

Standardizing apart eliminates overlap of variables, e.g., *Knows*(z_{17}, *OJ*) by renaming its variables to avoid name clashes

Most General Unifier

Multiple unifiers are possible:

$$\text{UNIFY}(Knows(John, x), Knows(y, z))$$

$$\{y/John, x/z\}$$

or

$$\{y/John, x/John, z/John\}$$

Which is better, Knows (John, z) or Knows (John, John)?

- Second could be obtained from first with extra subs
- First unifier is more general than second because it places fewer restrictions on the values of the variables

There is a single most general unifier for every unifiable pair of expressions

Occur Check

t1 = likes(X, Y)

t2 = likes(g(Y),

An equation that has a variable on one side and a term containing that variable on the other side cannot be solved no matter what the substitute is. So the unification fails. The above phenomenon is called "occurs check".

Unification algorithm

```
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

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  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, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ 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 {var/x} to θ
```

Storage and retrieval

Once the data type for sentences and terms are defined, we need to contain a set of sentences in a KB

i) Stores(s) – store a sentence s

ii) Fetch(s)-returns all unifiers

such that query q unifies with some sentence in KB.

An example is when we ask Knows (John, x) – is an instance of fetching

- The simplest way to implement STORE and FETCH is to keep all the facts in the knowledge base in one long list
- The given query UNIFY(q,s) – given query call for every sentence s in the list, requires O(n) time on an n-element KB.
- Such a process is inefficient in terms of time taken because it will unify each and every statement of knowledge base
- Inefficient because we're performing so many unifies.

- Example: unify Knows (John, x) with Brother (Richard, John)
  There is no need to unify

➢ We can avoid such unification by indexing the facts in the knowledge base

➢ Predicate indexing puts all the *Knows* facts in one bucket and all the *Brother* facts in another

➢ The bucket stored in a hash table for efficient access.

➢ We can large bucket problem with the help of multiple indexing

- Might not be a win if there are lots of clauses for a particular predicate symbol
  - Consider how many people *Know* one another

– Instead index by predicate and first argument
  - Clauses may be stored in multiple buckets

Subsumption lattice

The above set of queries are said to form a collection tree called as Subsumption lattice

How to construct indices for all possible queries that unify with it

- **Example: Employs (AIMA.org, Richard)**

$Employs(AIMA.org, Richard)$ Does AIMA.org employ Richard?
$Employs(x, Richard)$ Who employs Richard?
$Employs(AIMA.org, y)$ Whom does AIMA.org employ?
$Employs(x, y)$ Who employs whom?

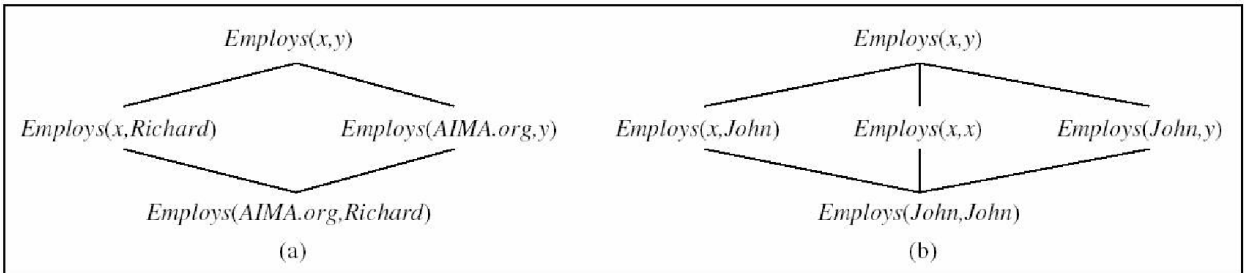


Figure 9.2 (a) The subsumption lattice whose lowest node is the sentence $Employs(AIMA.org, Richard)$. (b) The subsumption lattice for the sentence $Employs(John, John)$.

Properties of subsumption lattice

- The child of any node in the lattice is obtained from its parent by single substitution
- The "highest" common descendent of any two nodes is the result of applying the most general unifier
- The portion of lattice , above any ground fact can bd e construct
- Predicate with *n* arguments will create a lattice with $O(2^n)$ nodes
- Benefits of indexing may be outweighed by cost of storing and maintaining indices

8.Explain resolution concept& its procedure (NOV 2012)(NOV 2011)(MAY 2011)

- Resolution is a procedure used in proving that arguments which are expressible in predicate logic are correct
- Resolution is a procedure that produces proofs by refutation or contradiction
- Resolution lead to refute a theorem-proving technique for sentences in propositional logic and first order logic

- Resolution is a rule of inference
- Resolution is computerized theorem prover

-Resolution is so far only defined for propositional logic. The strategy is that the resolution techniques of propositional logic be adopted in predicate logic

Procedure for resolution

- Convert given proposition in to clausal form
- Convert the negation of the sentence to be proved into clausal form
- Combine the clauses into a set
- Iteratively apply resolution to the set and add the resolvent to the set
- Continue until no further resolvents can be obtained or a null clause is obtained

A Predicate Logic Example

1. Marcus was a man. $man(Marcus)$
2. Marcus was a Pompeian. $Pompeian(Marcus)$
3. All Pompeians were Romans. $\forall x: Pompeian(x) \rightarrow Roman(x)$
4. Caesar was a ruler. $ruler(Caesar)$
5. All Romans were either loyal to Caesar or hated him.
   $\forall x: Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. Everyone is loyal to someone. $\forall x: \exists y: loyalto(x,y)$
7. People only try to assassinate rulers they aren't loyal to.
   $\forall x:\forall y: person(x) \wedge ruler(y) \wedge tryassassinate(x,y) \rightarrow \neg loyalto(x,y)$
8. Marcus tried to assassinate Caesar.
   $tryassassinate(Marcus, Caesar)$
9. All men are people. $\forall x: man(x) \rightarrow person(x)$

Conversion to Clause Form

- Problem:

  $\forall x: [Roman(x) \wedge know(x,Marcus)] \rightarrow [hate(x,Caesar) \vee (\forall y:\exists z: hate(y,z) \rightarrow thinkcrazy(x,y))]$
- Solution:
  - Flatten
  - Separate out quantifiers
- Conjunctive Normal Form: $\neg Roman(x) \vee \neg know(x,Marcus) \vee hate(x,Caesar) \vee \neg hate(y,z) \vee thinkcrazy(x,z)$
- Clause Form
  - Conjunctive normal form

Converting FOL Sentences to CNF

1. Replace ⇔ with equivalent (added):
   - convert $P \Leftrightarrow Q$ to $P \Rightarrow Q \wedge Q \Rightarrow P$
2. Replace ⇒ with equivalent: convert $P \Rightarrow Q$ to $\neg P \vee Q$
3. Reduce scope of ¬ to single literals:
   - convert $\neg\neg P$ to P (DNE)
   - convert $\neg(P \vee Q)$ to $\neg P \wedge \neg Q$ (de Morgan's)
   - convert $\neg(P \wedge Q)$ to $\neg P \vee \neg Q$ (de Morgan's)
   - convert $\neg\forall x\, P$ to $\exists x \neg P$
   - convert $\neg\exists x\, P$ to $\forall x \neg P$
4. Standardize variables apart:
   - each quantifier must have a unique variable name
   - avoids confusion in steps 5 and 6
   - e.g. convert $\forall x\, P \vee \exists x\, Q$ to $\forall x\, P \vee \exists y\, Q$
5. Eliminate existential quantifiers (Skolemize):
   - convert $\exists x\, P(x)$ to $P(C)$ (EE)
     C must be a new constant (Skolem constant)
   - convert $\forall x, y\, \exists z\, P(x, y, z)$ to $\forall x, y\, P(x, y, F(x, y))$
     F() must be a new function (Skolem function) with arguments that are all enclosing universally quantified variables

Skolem Functions in FOL

- **Objective**
  - Want all variables universally quantified
  - Notational variant of FOL w/o existentials
  - Retain implicitly full FOL expressiveness
- **Skolem's Theorem**

Every existentially quantified variable can be replaced by a unique *Skolem function* whose arguments are all the universally quantified variables on which the existential depends, without changing FOL.

- **Examples**
  - "Everybody likes something"

∀(x) ∃ (y) [Person(x) & Likes(x,y)]
∀(x) [Person(x) & Likes(x, S1(x))]
Where S1(x) = "that which x likes"

  - "Every philosopher writes at least one book"

∀(x) ∃(y)[Philosopher(x) & Book(y)) => Write(x,y)]
∀(x)[(Philosopher(x) & Book(S2(x))) => Write(x,S2(x))]

6. Drop quantifiers:
   - all variables are only universally quantified after step 5
   - e.g. convert $\forall x\, P(x) \vee \forall y\, Q(y)$ to $P(x) \vee Q(y)$
   - all variables in KB will be assumed to be universally quantified
7. Distribute $\wedge$ over $\vee$ to get conjunction of disjunctions :
   - convert $(P \wedge Q) \vee R$ to $(P \vee R) \wedge (Q \vee R)$

8. Flatten nested conjunctions and disjunctions:
   - convert $(P \wedge Q) \wedge R$ to $(P \wedge Q \wedge R)$
   - convert $(P \vee Q) \vee R$ to $(P \vee Q \vee R)$

9. Separate each conjunct (added)

   spllit at $\wedge$'s so each conjunct is now a CNF clause

```
(¬Above(x,y) ∨ OnTop(x,y) ∨ OnTop(x,F(x,y))) ∧
(¬Above(x,y) ∨ OnTop(x,y) ∨ Above(F(x,y),y))
```

becomes:

```
¬Above(x,y) ∨ OnTop(x,y) ∨ OnTop(x,F(x,y))
¬Above(x,y) ∨ OnTop(x,y) ∨ Above(F(x,y),y)
```

10. Standardize variables apart in each clause (added)
    - each clause in KB must contain unique variable names
    - now during unification the standardize apart step need only be done on deduced clauses (i.e. resolvents)

```
¬Above(x,y) ∨ OnTop(x,y) ∨ OnTop(x,F(x,y))
¬Above(x,y) ∨ OnTop(x,y) ∨ Above(F(x,y),y)
```

Examples of Conversion to Clause Form

- Example:

$\forall x$: $[Roman(x) \wedge know(x, Marcus)] \rightarrow [hate(x,Caesar) \vee (\forall y: \exists z: hate(y,z) \rightarrow thinkcrazy(x,y))]$

  - Eliminate $\rightarrow$

$\forall x$: $\neg[Roman(x) \wedge know(x, Marcus)] \vee [hate(x,Caesar) \vee (\forall y$:

$\exists \neg z$: *hate*(*y*,*z*) V *thinkcrazy*(*x*,*y*))]

- Reduce scope of ¬.

$\forall x$: [¬*Roman*(*x*) V ¬ *know*(*x*, *Marcus*)] V [*hate*(*x*,*Caesar*) V ($\forall y$: $\forall z$: ¬*hate*(*y*,*z*) V *thinkcrazy*(*x*,*y*))]

- "Standardize" variables:

$\forall x$: *P*(x) V $\forall x$: Q(*x*) converts to $\forall x$: *P*(*x*) V $\forall y$: Q(*y*)

- Move quantifiers. $\forall x$: $\forall y$: $\forall z$: [¬*Roman*(*x*) V ¬ *know*(*x*, *Marcus*)] V [*hate*(*x*,*Caesar*) V (¬*hate*(*y*,*z*) V *thinkcrazy*(*x*,*y*))]
- Eliminate existential quantifiers.

$\exists y$: *President*(*y*) will be converted to *President*(*S*1)

$\forall x$: $\exists y$: *father-of*(*y*,*x*) will be converted to $\forall x$: *father-of*(*S*2(*x*),*x*))

- Drop the prefix.

[¬*Roman*(*x*) ¬ *know*(*x*,*Marcus*)] V [*hate*(*x*, *Caesar*) V (¬ *hate*(*y*,*z*) V *thinkcrazy*(*x*,*y*))]

- Convert to a conjunction of disjuncts.

¬ *Roman*(*x*) V ¬ *know*(*x*,*Marcus*) V *hate*(*x*,*Caesar*) V ¬ *hate*(*y*,*z*) V *thinkcrazy*(*x*,*y*)

The Basis of Resolution and Herbrand's Theorem

- Given:

  winter V *summer*

  ¬ *winter* V cold

  We can conclude:

  summer v *cold*

- Herbrand's Theorem:

To show that a set of clauses *S* is unsatisfiable, it is necessary to consider only interpretations over a particular set, called the *Herbrand universe* of *S*. A set of clauses *S* is unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of *S* is unsatsifable.

Algorithm: Propositional Resolution

1. Convert all the propositions of *F* to clause form.
2. Negate *P* and convert the result to clause form. Add it to the set of clauses obtained in step1.
3. Repeat until either a contradiction is found or no progress can be made:

a) Select two clauses. Call these the parent clauses.

b) Resolve them together.

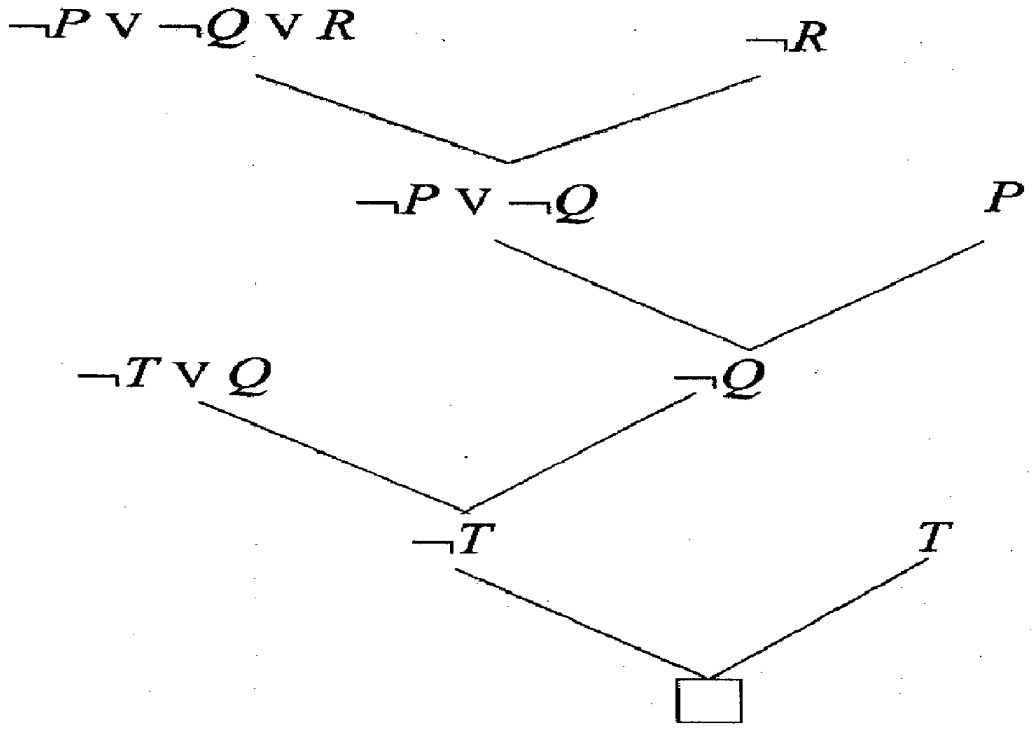
The *resolvent* will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals *L* and ¬ *L* such that one of the parent clauses contains *L* and the other contains ¬*L*, then select one such pair and eliminate both *L* and ¬*L* from the resolvent.

c) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

A Few Facts in Propositional Logic

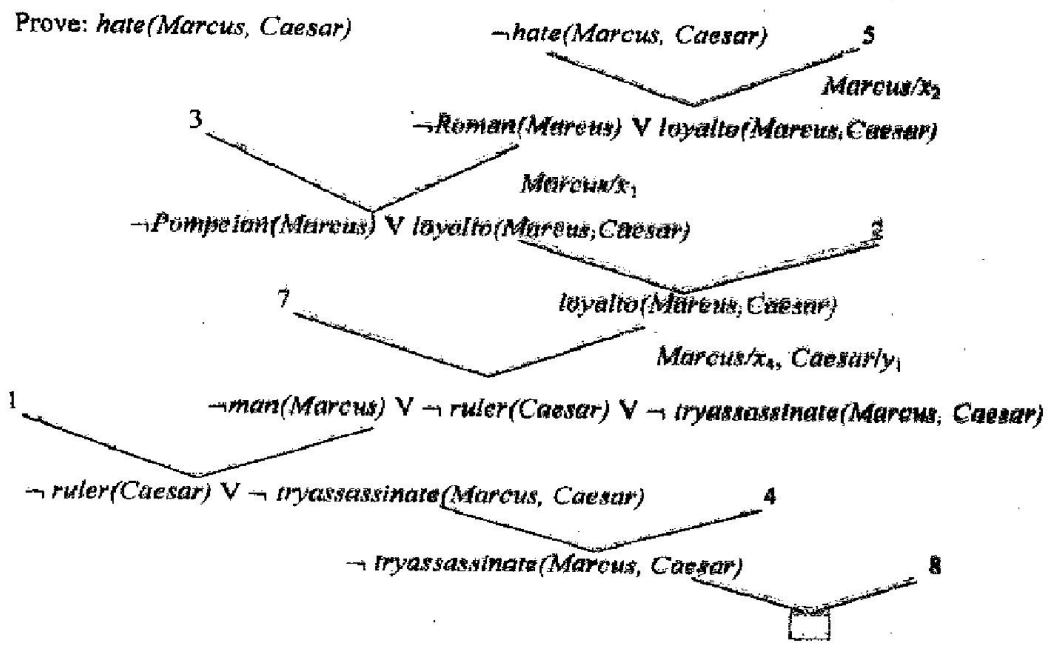
Given Axioms	Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Resolution in Propositional Logic



- Axioms in clause form:
  1. *man(Marcus)*
  2. *Pompeian(Marcus)*
  3. $\neg$ *Pompeian*(x_1) v Roman(x_1)
  4. *Ruler(Caesar)*
  5. $\neg$ *Roman*(x_2) v *loyalto*(x_2, *Caesar*) v *hate*(x_2, *Caesar*)
  6. *loyalto*(x_3, *f1*(x_3))
  7. $\neg$ *person*(x_4) v $\neg$ *ruler*(y_1) v $\neg$ *tryassassinate*(x_4, y_1) v *loyalto* (x_4, y_1)
  8. *tryassassinate*(*Marcus, Caesar*)

9 Man(x5)v person(x5)



Issues with Resolution

- Requires full formal representation in FOL (for conversion to clause form)
- Resolution defines a search space (which clauses will be resolved against which others define the operators in the space) ➔ search method required
- Worst case: resolution is exponential in the number of clauses to resolve. Actual: exponential in average resolvable set (= branching factor)
- Can we define heuristics to guide search for BestFS, or A* or B*? (Not in the general case)

2. Explain Forward and backward chaining (NOV 2013)(MAY 2013)(MAY 2012)(NOV 2011).

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

First-order definite clauses

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

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

- $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$
- Nono… has some missles

∃x Owns(Nono,x) ∧ Missile(x):

$Owns(Nono, M_1)$ and $Missile(M_1)$

- All of its missiles were sold to it by Colonel West

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

- We also need to know that missiles are weapons

$$Missile(x) \Rightarrow Weapon(x)$$

- and we must know that an enemy of America counts as "hostile"

$$Enemy(x, America) \Rightarrow Hostile(x)$$

"

- West, who is American"

$$American(West)$$

Nono, is a nation

Nation(nono)

- The country Nono, an enemy of America

$$Enemy(Nono, America)$$

America is a nation

Nation (America)

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

Analyse the algorithm

Sound

- Does it only derive sentences that are entailed?
- Yes, because only Modus Ponens is used and it is sound

Complete

- Does it answer every query whose answers are entailed by the KB?
- Yes if the clauses are definite clauses

Assume KB only has sentences with no function symbols

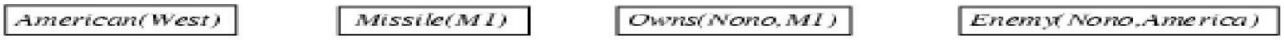
- What's the most number of iterations through algorithm?
- Depends on the number of facts that can be added

  Let *k* be the arity, the max number of arguments of any predicate and
  Let *p* be the number of predicates
  Let *n* be the number of constant symbols
- At most pn^k distinct ground facts
- Fixed point is reached after this many iterations
- A proof by contradiction shows that the final KB is complete

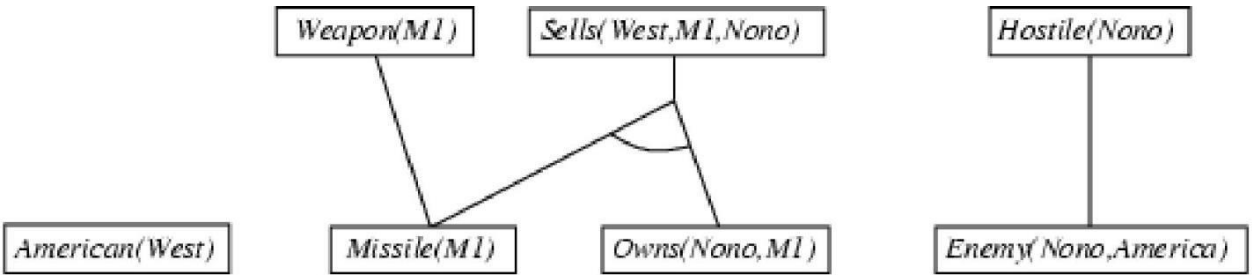
Three sources of complexity

- inner loop requires finding all unifiers such that premise of rule unifies with facts of database
  - this "pattern matching" is expensive
- must check every rule on every iteration to check if its premises are satisfied
- many facts are generated that are irrelevant to goal

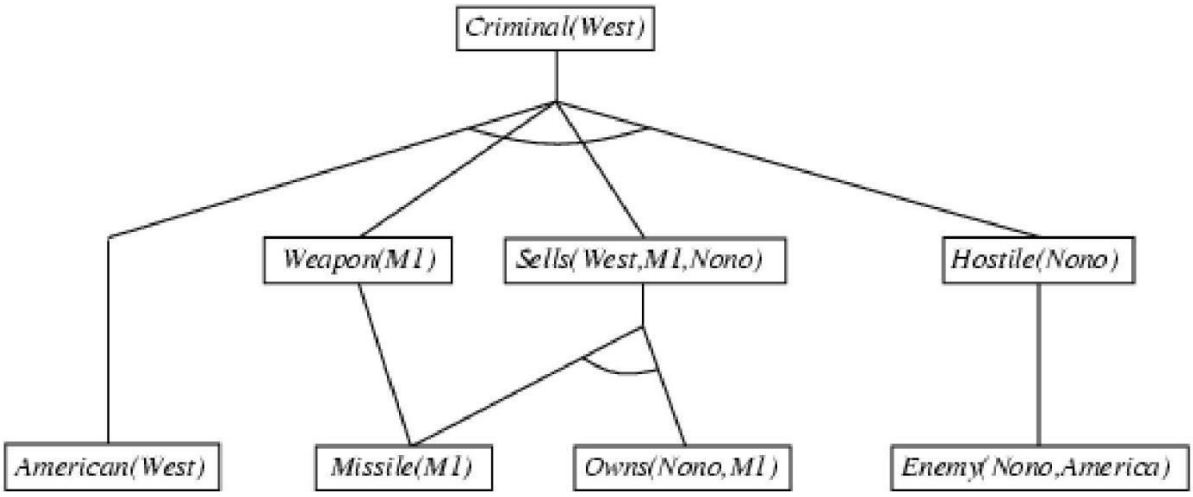
Step1:



Step2:



Step3:



Efficiency of forward chaining

- Incremental forward chaining: no need to match a rule on iteration *k* if a premise wasn't added on iteration *k-1*

  match each rule whose premise contains a newly added positive literal
- Matching itself can be expensive:
- Database indexing allows O(1) retrieval of known facts

  e.g., query *Missile(x)* retrieves *Missile* (M_1)

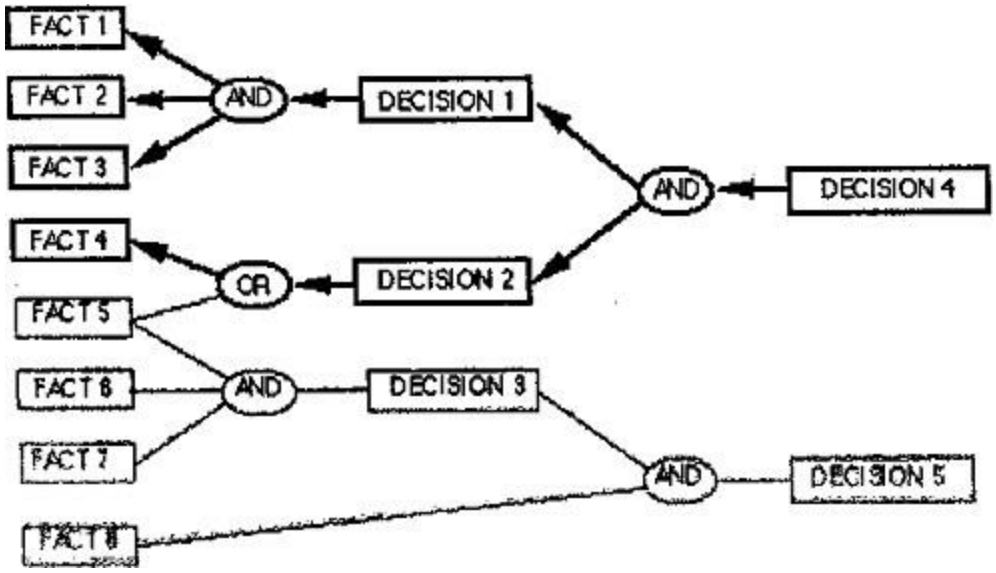
  Forward chaining is widely used in deductive databases

Backward chaining

In backward chaining, we start from a conclusion, which is the hypothesis we wish to prove, and we aim to show how that conclusion can be reached from the rules and facts in the data base. To determine if a decision should be made, work backwards looking for justifications for the decision,

The conclusion we are aiming to prove is called a goal ,and the reasoning in this way is known as goal-driven.

Eventually, a decision must be satisfied by facts



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, the current substitution, initially the empty substitution $\{\}$

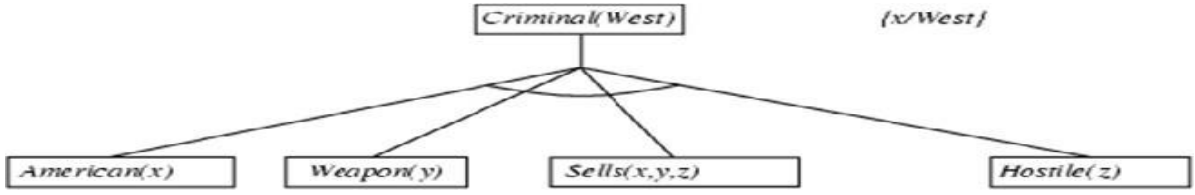
local variables: ans, a set of substitutions, initially empty

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

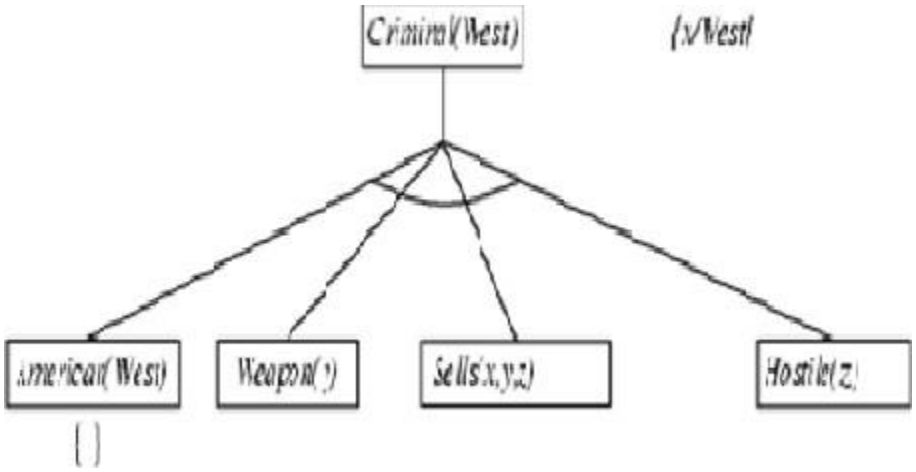
The algorithm uses composition of substitution. COMPOSE(θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution is
SUBST(COMPOSE(θ_1, θ_2), p) = SUBST(θ_2, SUBST(θ_1, p))
In the algorithm , the current variable binding , which are stored in θ, are composed with the binding resulting from unifying the goal with the clause head , given a new set of current bindings for the recursive call.

Step1

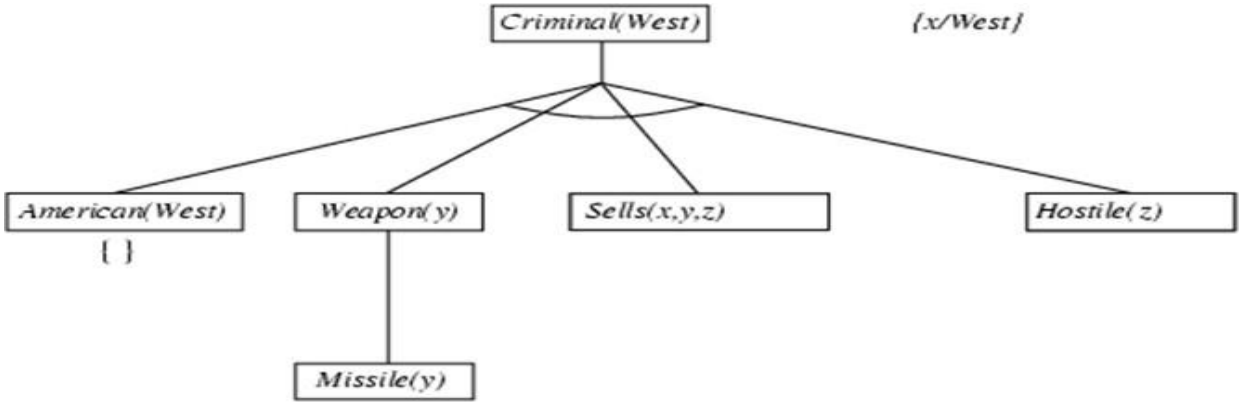
Step2



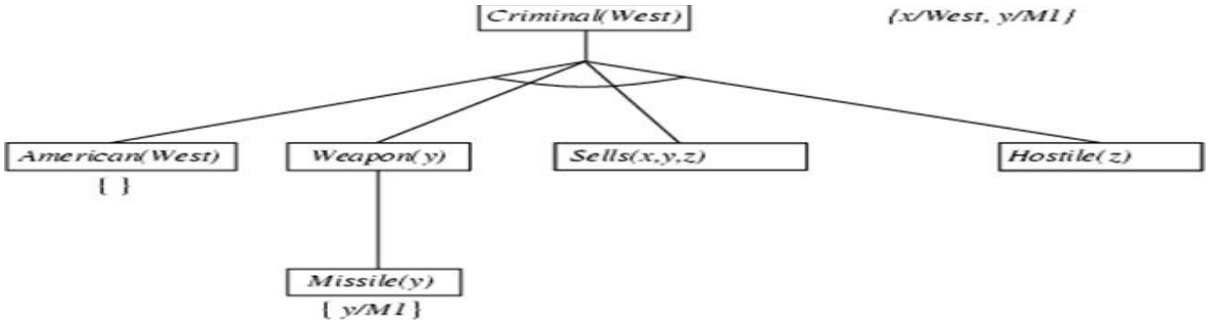
Step 3



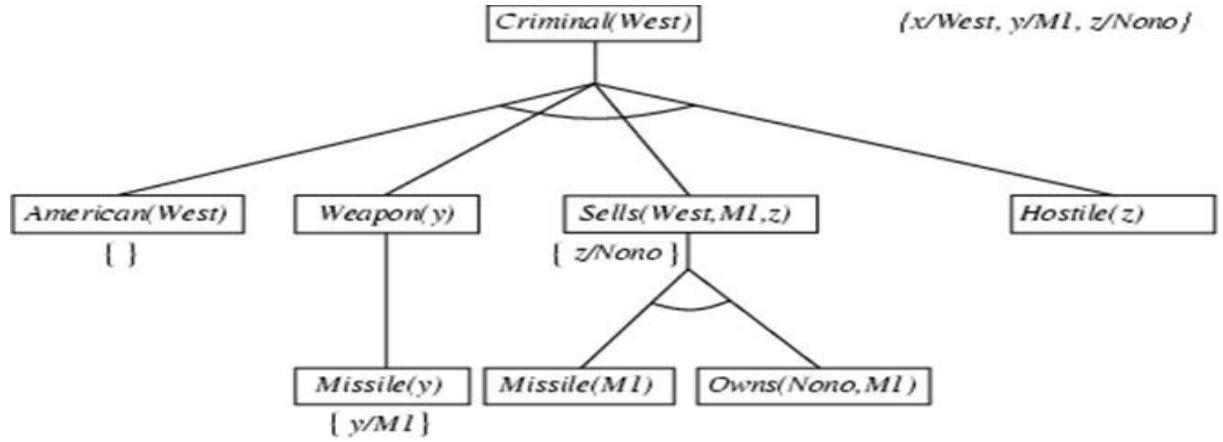
Step 4



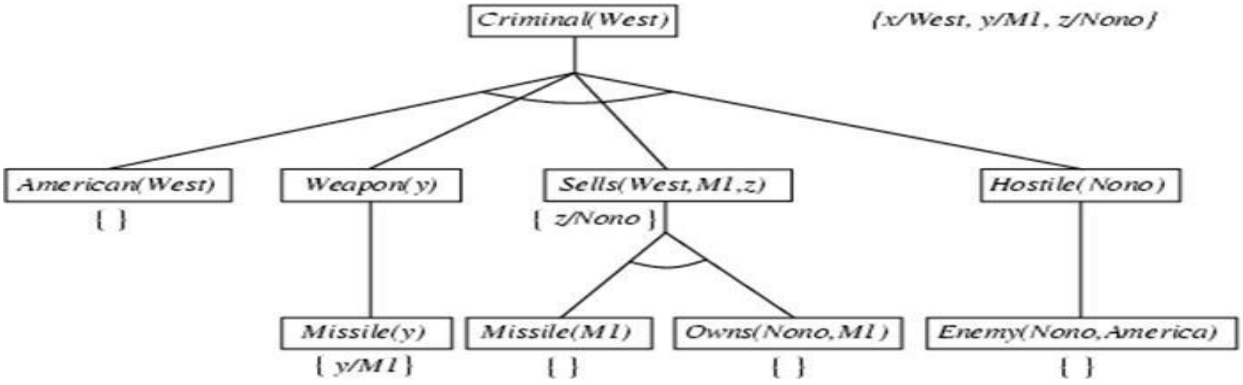
Step 5



Step 6



Step 7



properties of backward chaining

Depth-first recursive proof search: space is linear in size of proof

Incomplete due to infinite loops □ fix by checking current goal against every goal on stack

Inefficient due to repeated subgoals (both success and failure)

□fix using caching of previous results (extra space)

Widely used for logic programming: prolog

Two marks

1 What is Ontological Engineering

Ontology refers to organizing everything in the world into hierarch of categories. Representing the abstract concepts such as Actions,Time,Physical Objects, and Beliefs is called Ontological Engineering

2 1Define diagnostic rules with example?

Diagnostic rules are used in first order logic for inferencing. The diagnostic rules generate hidden causes from observed effect. They help to deduce hidden facts in the world. For example considering the wumpum world.

The diagnostic rule finding 'pit' is,

"If square is breezy some adjacent square must contain pit", which is written as,

∀xBreezy(s)=> ∃rAdjacent(r,s)^ pit (r

Define the first order definite clause? (NOV 2013)

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

King (x) A Greedy (x) ⇒ Evil (x) .
King(John) .
Greedy(y) .

4

S. No	Propositional logic	Predicate logic
1	It is less expressive power	It is more expressive power
2	It cannot represent relationship among objects	It represent relationship among objects
3	It does not use quantifiers	It use quantifiers
4	It does not consider generalization of objects	It consider generalization of objects