Introduction to Artificial Intelligence

UNIT-I

- Introduction to AI
- Intelligent Agents,
- Problem-Solving Agents,
- Searching for Solutions
- Breadth-first search,
- Depth-first search,
- Hill-climbing search,
- Simulated annealing search,
- Local Search in Continuous Spaces.

Introduction to AI

Introduction to AI

- 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 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, , 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.

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, logic and inference, algorithms, programming languages, and system building.

- **Mathematics and Physics**

Application of AI:

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

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
- 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.

- Laws of thought: Think Rationally
- Cognitive Science: Think Human-Like
- Turing Test: Act Human-Like
- Rational agent: Act Rationally

The difference between strong AI and weak AI:

- Strong AI: Strong AI powered machines can exhibit strong human cognitive abilities.

- Weak AI : Weak AI Powered machines do not have mind of their own

  Alexa and siri are the best examples.

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 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.

The State of the Art: What can AI do Today?

- Autonomous planning and scheduling(NASAs Remote Agent Program)
- Game Playing(IBM's Deep Blue)
- Autonomous Control(ALVINN Computer Vision System- trained to steer a car)
- Diagnosis
- Logistics Planning(DART –Dynamic Analysis and Replanning Tool)
- Robotics(Hip Nav-3D modelshows internal Anatomy of patient and guid to insertion of Hip replacement)
- Language understanding and problem solving(PROVERB Program- solves cross word puzzels better then humans)

Intelligent Agents

Agents and Environments:

- An Agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.
  - A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
  - A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
  - A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

- Percept: We use the term percept to refer to the agent's perceptual inputs at any given instant.
- Percept Sequence: An agent's percept sequence is the complete history of everything the agent has ever perceived.
- Agent function: Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any given percept sequence to an action.
- Agent program Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

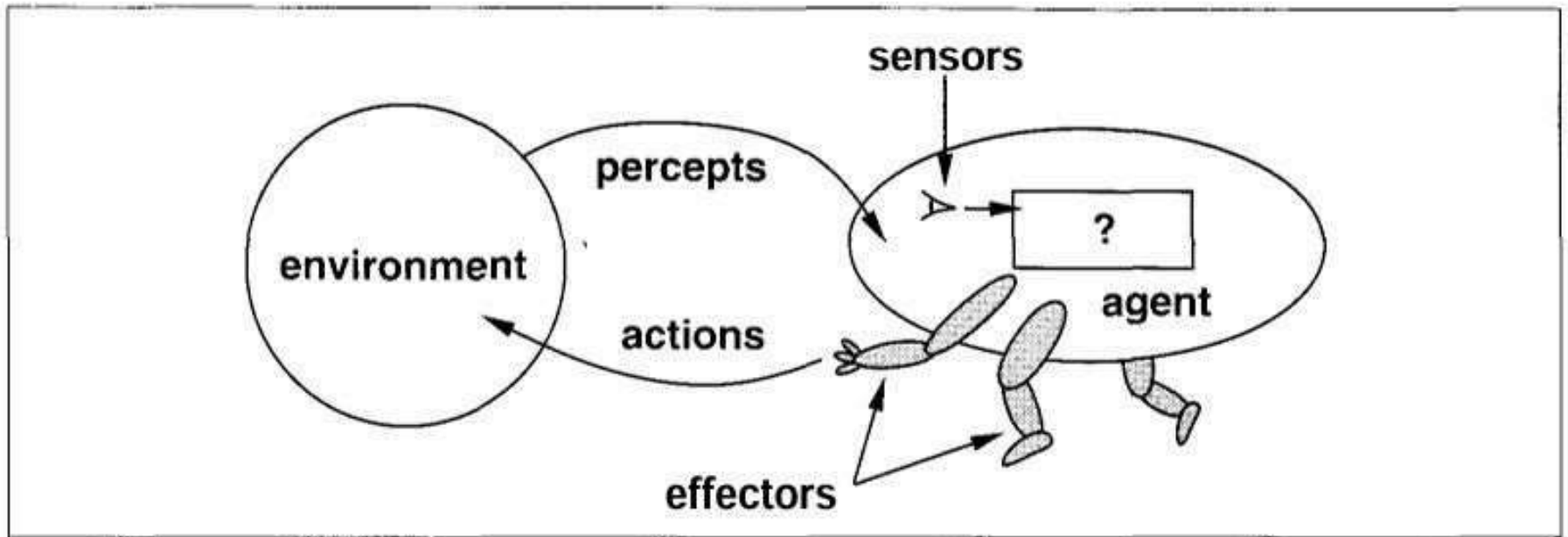


Figure 2.1 Agents interact with environments through sensors and effectors.

example-the vacuum-cleaner world

- 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.

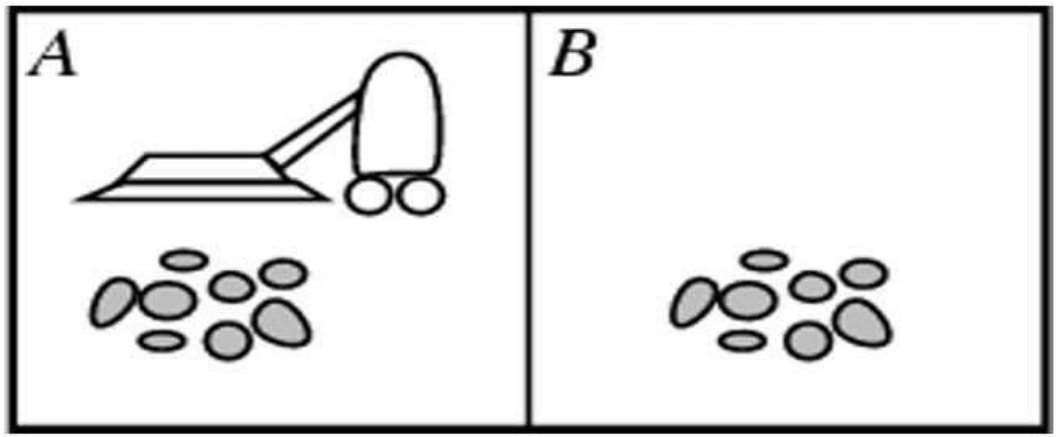


Fig A vacuum-cleaner world with just two locations.

An Effector is any device that affects the physical environment. An actuator is the actual mechanism that enables the effector to execute an action.

THE CONCEPT OF RATIONALITY

- Rationality is nothing but status of being reasonable, sensible, and having good sense of judgment.
- Rationality is concerned with expected actions and results depending upon what the agent has perceived. Performing actions with the aim of obtaining useful information is an important part of rationality.

- What is Ideal Rational Agent?
- An ideal rational agent is the one, which is capable of doing expected actions to maximize its performance measure, on the basis of –
  - Its percept sequence
  - Its built-in knowledge base

The Structure of Agents

- Agent’s structure can be viewed as –
- Agent = Architecture + Agent Program
- Architecture = the machinery that an agent executes on.
- Agent Program = an implementation of an agent function.

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

Figure 23 Examples of agent types and their PAGE descriptions.

Agent Type	Percepts	Actions	Goals	Environment
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers

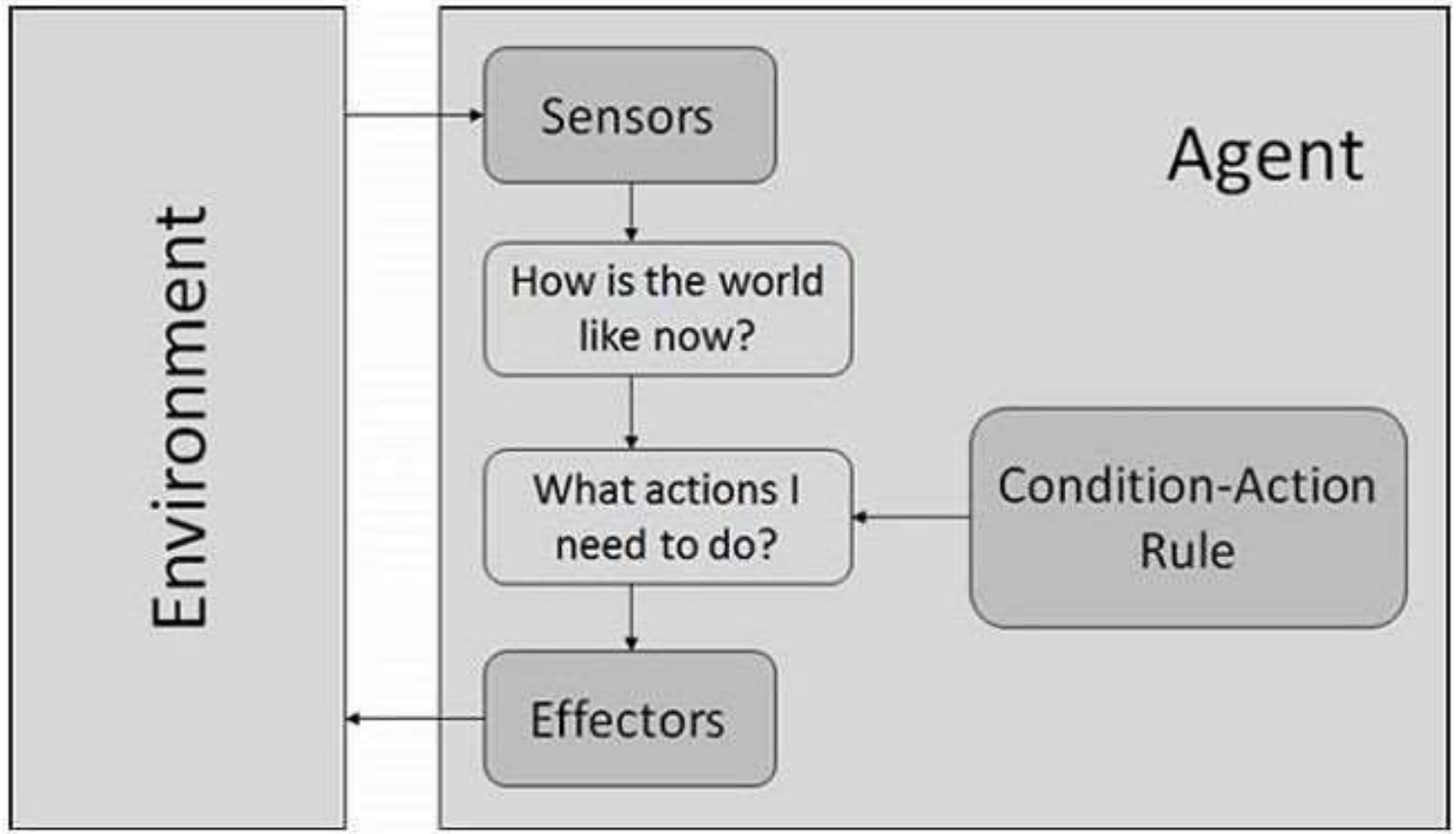
Figure 2.6 The taxi driver agent type.

We will consider four types of agent program:

- Simple reflex agents
- Model based reflex agents (Agents that keep track of the world)
- Goal-based agents
- Utility-based agents
- Learning agents

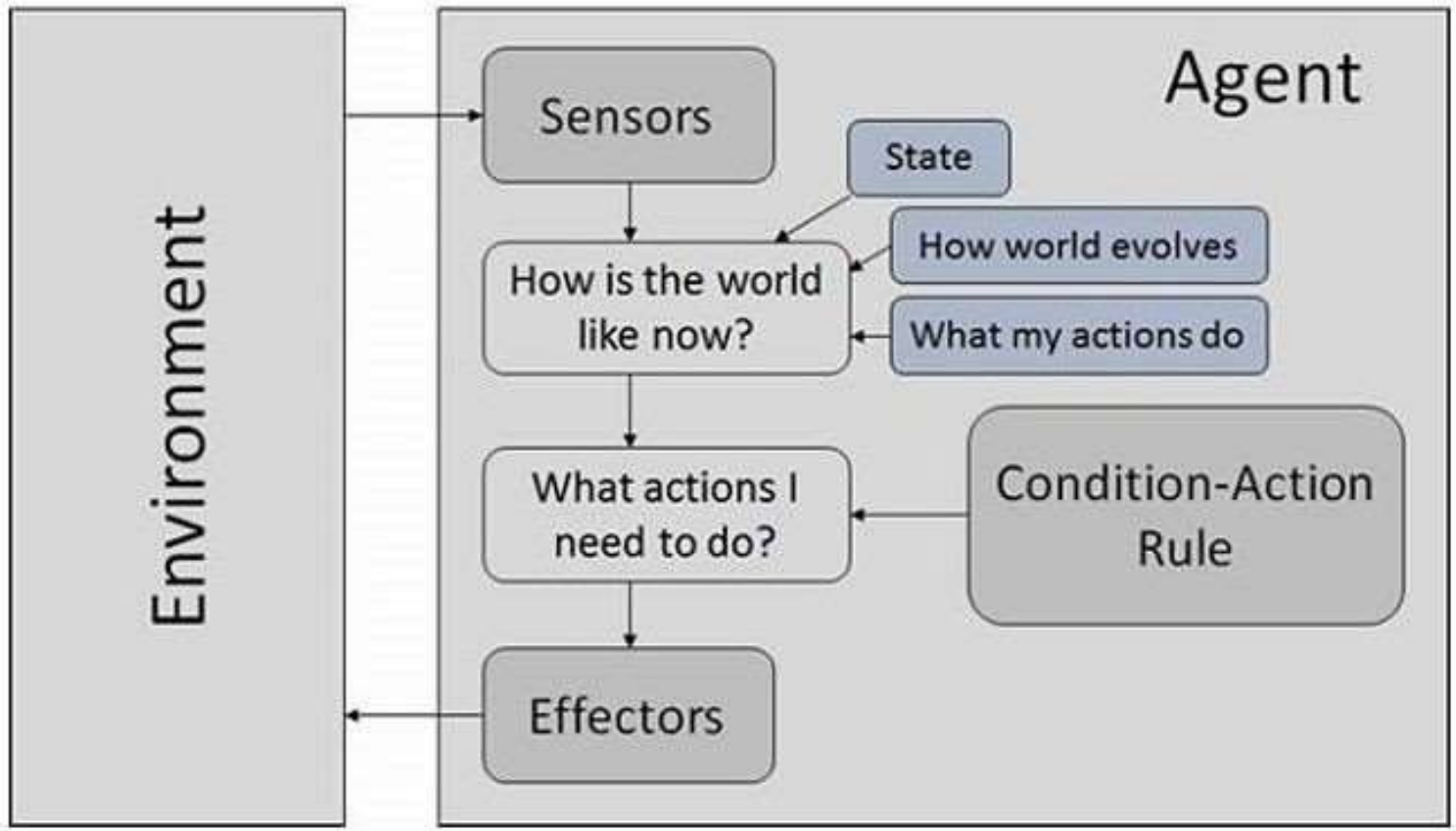
1. Simple Reflex Agents

- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision and action process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.
- Problems for the simple reflex agent design approach:
  - They have very limited intelligence
  - They do not have knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment.


Environment
Sensors
Agent
How is the world like now?
What actions I need to do?
Condition-Action Rule
Effectors

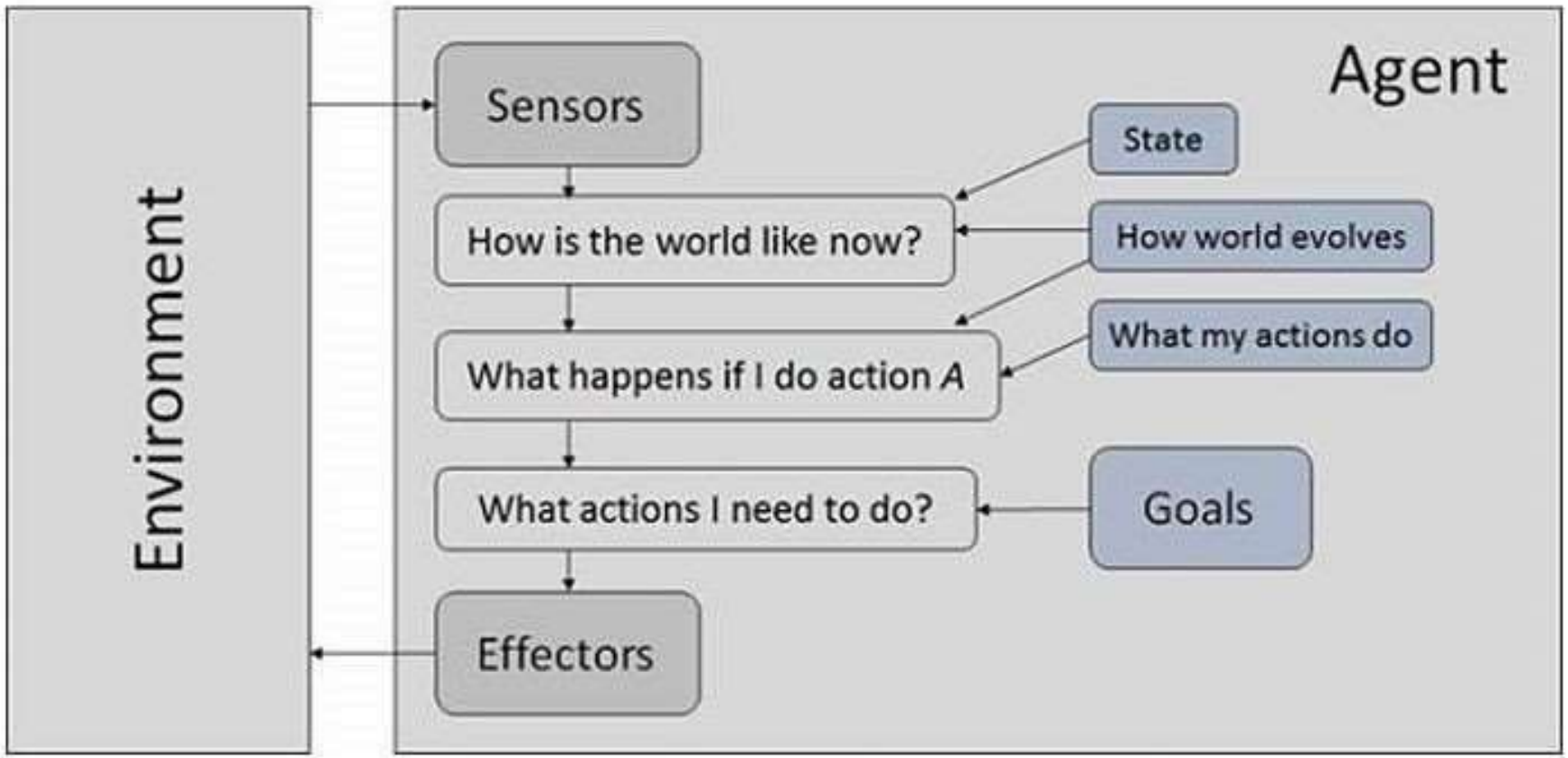
2. Model Based Reflex Agents

- They use a model of the world to choose their actions. They maintain an internal state.
- Model – knowledge about “how the things happen in the world”.
- Internal State – It is a representation of unobserved aspects of current state depending on percept history.
- Updating the state requires the information about –
- • How the world evolves.
- • How the agent’s actions affect the world.


Environment
Agent
Sensors
State
How world evolves
How is the world like now?
What my actions do
What actions I need to do?
Condition-Action Rule
Effectors

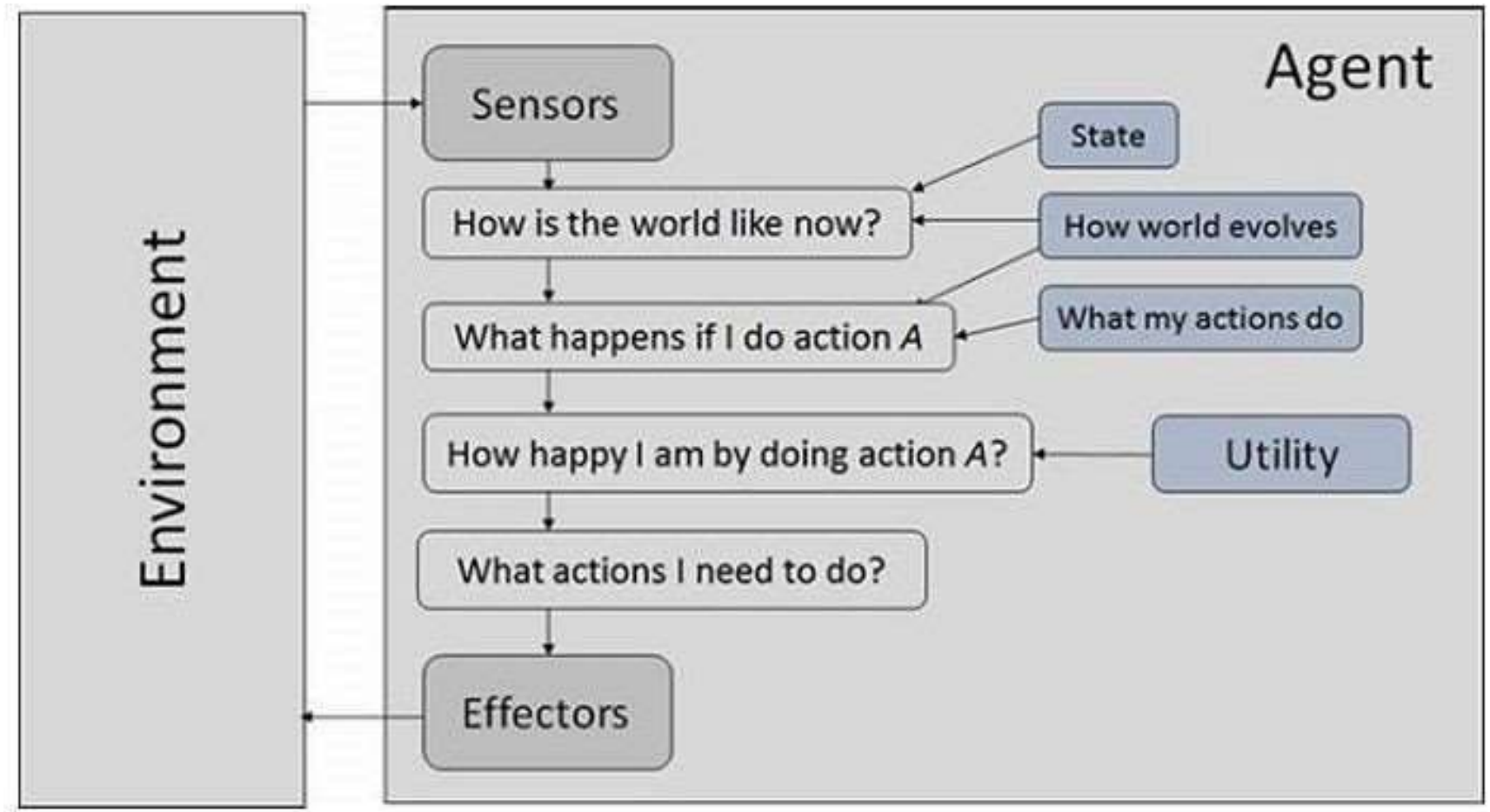
3.Goal-Based Agents

- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.


Environment
Agent
Sensors
How is the world like now?
What happens if I do action A
What actions I need to do?
Effectors
State
How world evolves
What my actions do
Goals

4. Utility Based Agents

- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals.


Environment
Agent
Sensors
State
How is the world like now?
How world evolves
What happens if I do action A
What my actions do
How happy I am by doing action A?
Utility
What actions I need to do?
Effectors

Properties of Environments

Environments come in several flavors. The principal distinctions to be made are as follows:

- Fully Observable vs. partially Observable
- Deterministic vs. nondeterministic(Stochastic).
- Episodic vs. nonepisodic(Sequential)
- Single Agent vs. Multi-Agent
- Static vs. dynamic
- Discrete vs. continuous

Fully observable / Partially Observable

Fully observable

- An agent can always see the entire state of an environment.
- **Example**: Chess

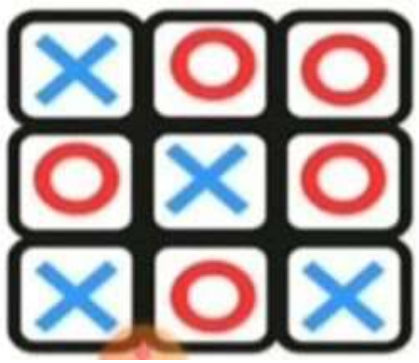
Partially Observable

- An agent can never see the entire state of an environment.
- **Example**: Card game

Deterministic / Stochastic

Deterministic

- An agent's current state and selected action can completely determine the next state of the environment.
- **Example**: Tic tac toe



Stochastic

- A stochastic environment is random in nature and cannot be determined completely by an agent.
- **Example**: Ludo (Any games that involve dice)

Episodic / Sequential

Episodic

- Only the current percept is required for the action
- Every episode is independent of each other
- **Example**: part picking robot

Sequential

- An agent requires memory of past actions to determine the next best actions.
- The current decision could affect all future decisions.
- **Example**: Chess

Single-agent / Multi-agent

Single-agent

- If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
- **Example**: maze

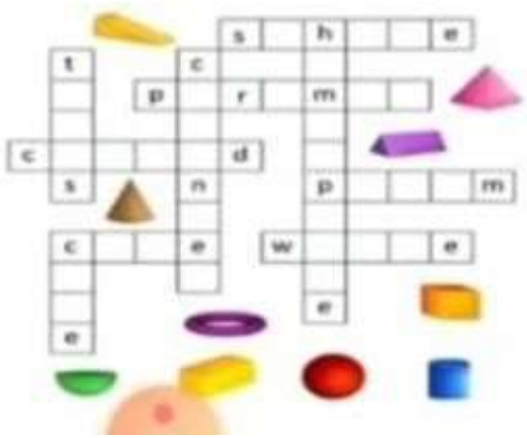
Multi-agent

- if multiple agents are operating in an environment, then such an environment is called a multi-agent environment.
- **Example**: football

Static / Dynamic

Static

- The environment does not change while an agent is acting
- **Example**: Crossword puzzles



Dynamic

- The environment may change over time
- **Example**: roller coaster ride
  Taxi driving

Discrete / Continuous

Discrete

- The environment consists of a finite number of actions that can be deliberated in the environment to obtain the output.
- **Example**: chess

Continuous

- The environment in which the actions performed cannot be numbered ie.,is not discrete, is said to be continuous.
- **Example**: Self-driving cars

PROBLEM SOLVING AGENTS

PROBLEM SOLVING AGENTS

- Intelligent agents are supposed to act in such a way that the environment goes through a sequence of states that maximizes the performance measure. In its full generality, this specification is difficult to translate into a successful agent design.
- The task is somewhat simplified if the agent can adopt a goal and aim to satisfy it.

- Goal formulation, based on the current situation, is the first step in problem solving.
- As well as formulating a goal, the agent may wish to decide on some other factors that affect the desirability of different ways of achieving the goal.
- Problem formulation is the process of deciding what actions and states to consider, and follows goal formulation.
- Assume that the agent will consider actions at the level of driving from one major town to another. The states it will consider therefore correspond to being in a particular town.

- In general, then, an agent with several immediate options of unknown value can decide what to do by first examining; different possible sequences of actions that lead to states of known value, and then choosing the best one. This process of looking for such a sequence is called search.
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase.

Formulating Problems

- Well Defined problems and Solutions
- A problem is really a collection of information that the agent will use to decide what to do.
- The basic elements of a problem definition are the states and actions. To capture these formally, we need the following:
  - The **initial state** that the agent knows itself to be in.
  - The term **operator** is used to denote the description of an action in terms of which state will be reached by carrying out the action in a particular state. (An alternate formulation uses a **successor function** S. Given a particular state x, S(x) returns the set of states reachable from x by any single action.)

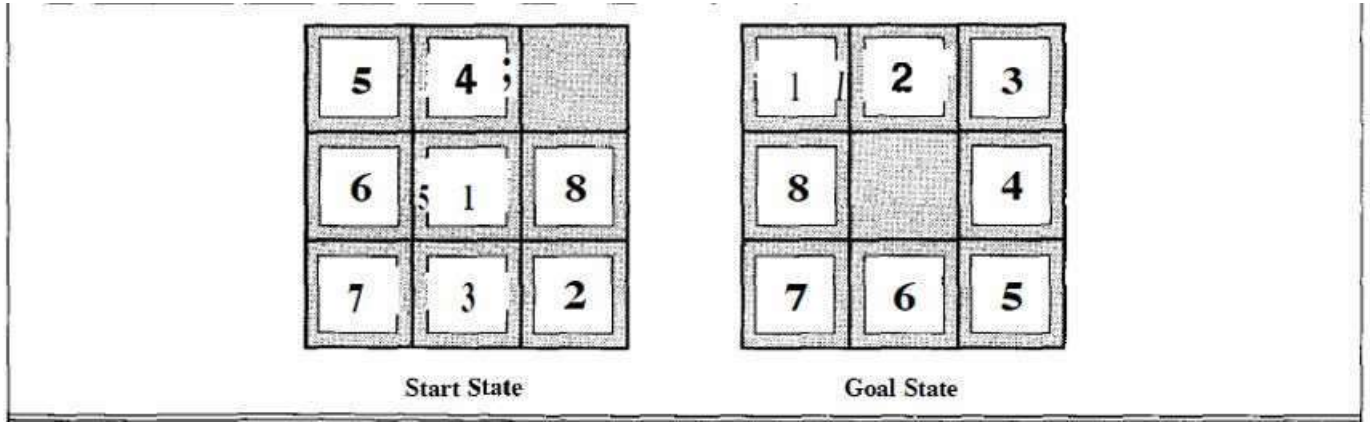
- state space of the problem: the set of all states reachable from the initial state by any sequence of actions. A path in the state space is simply any sequence of actions leading from one state to another.
- The goal test, which the agent can apply to a single state description to determine if it is a goal state.
- A path cost function is a function that assigns a cost to a path

- Measuring problem-solving performance
- The effectiveness of a search can be measured in at least three ways.
- First, does it rind a solution at all?
- Second, is it a good solution (one with a low path cost)?
- Third, what is the search cost associated with the time and memory required to find a solution?
- The total cost of the search is the sum of the path cost and the search cost.

Example Problems

Toy problems The 8-puzzle:

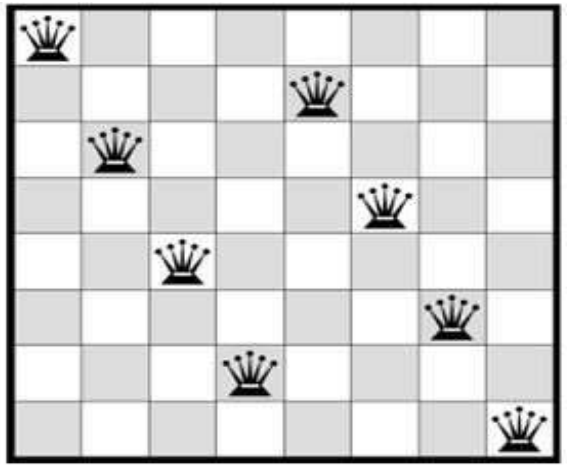
The 8-puzzle, an instance of which is shown in Figure, consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach the configuration shown on the right of the figure.



- This leads us to the following formulation:
- States: a state description specifies the location of each of the eight tiles in one of the nine squares. For efficiency, it is useful to include the location of the blank.
- Operators: blank moves left, right, up, or down.
- Goal test: state matches the goal configuration shown in Figure.
- Path cost: each step costs 1, so the path cost is just the length of the path.

The 8-queens problem

- The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.



- This figure doesn’t represent a solution because the queen in the first column is on the same diagonal as the queen in the last column.

- Goal test: 8 queens on board, none attacked.
- Path cost: zero.
- States: any arrangement of 0 to 8 queens on board.
- Operators: add a queen to any square.

- The vacuum world
- States: one of the eight states shown in Figure 3.2 (or Figure 3.6).
- Operators: move left, move'right, suck.
- Goal test: no dirt left in any square.
- Path cost: each action costs 1.

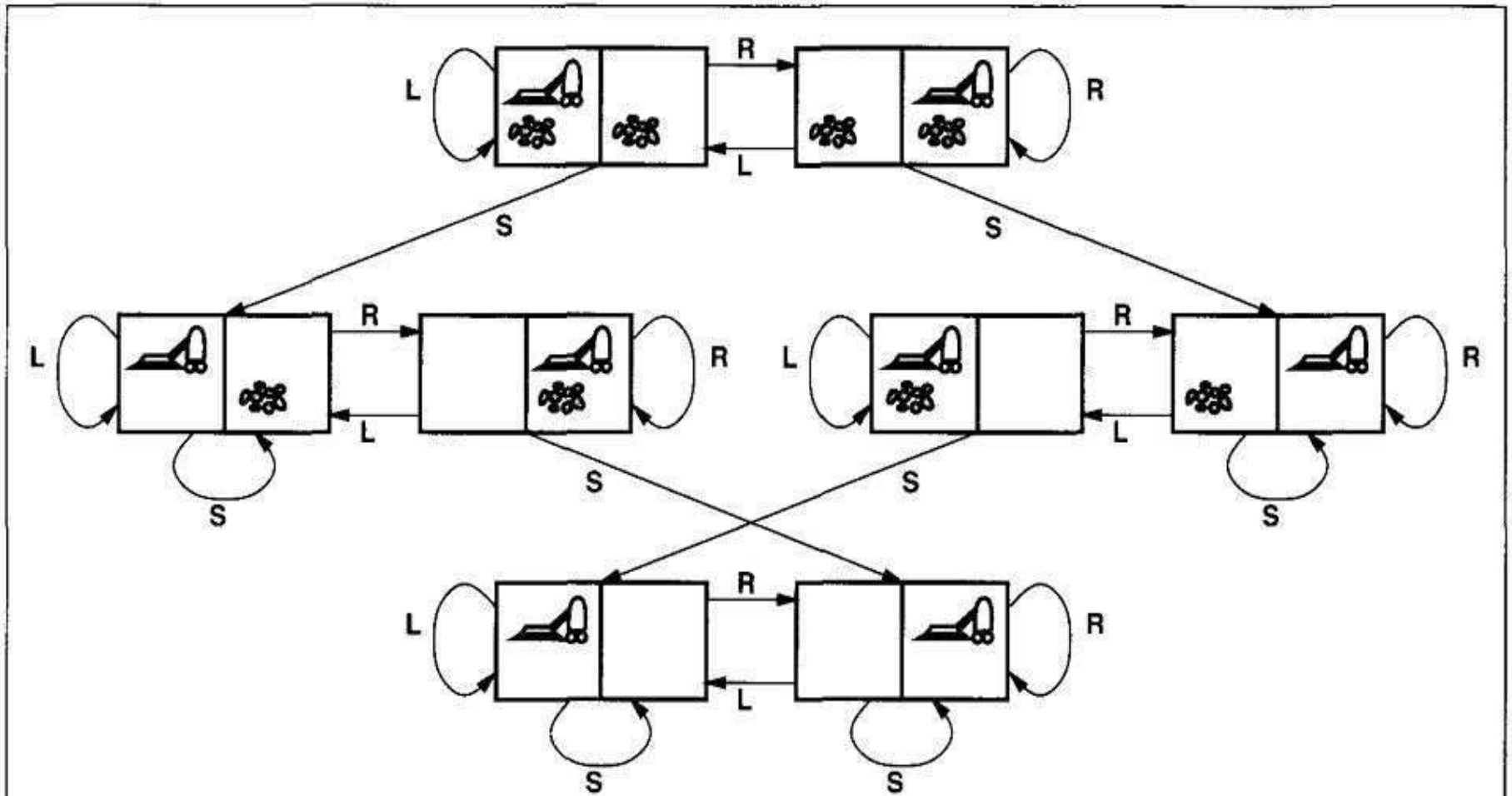


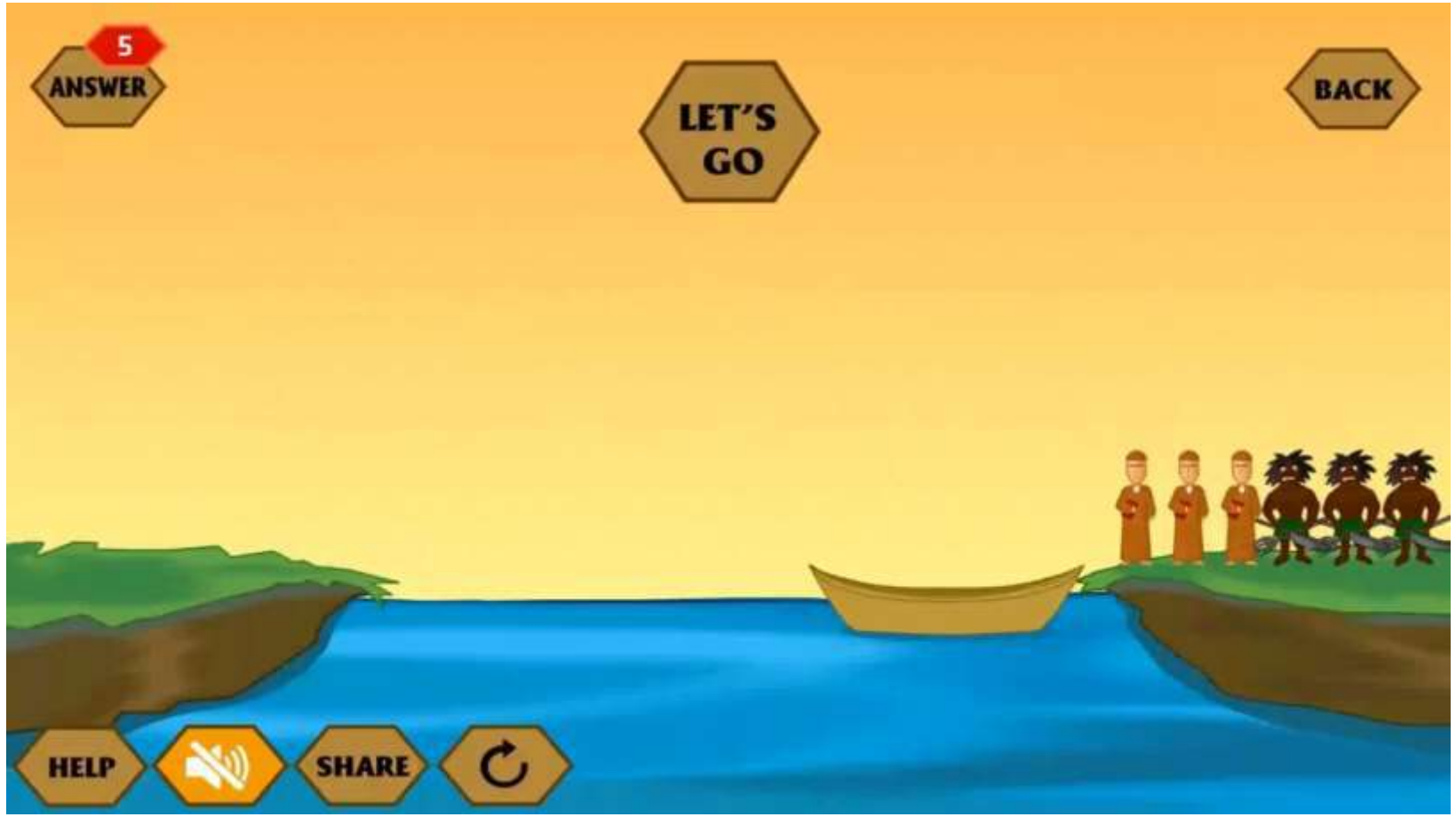
Figure 3.6 Diagram of the simplified vacuum state space. Arcs denote actions. L = move left, R = move riaht, S = suck.

- Cryptarithmetic
- In cryptarithmetic problems, letters stand for digits and the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct. Usually, each letter must stand for a different digit. The following is a well-known example

```
 FORTY        Solution:    29786      F=2, O=9, R=7, etc.
+  TEN                      850
+  TEN                      850
 -----                    -----
 SIXTY                    31486
```

- The following formulation is probably the simplest:
- States: a cryptarithmetic puzzle with some letters replaced by digits.
- Operators: replace all occurrences of a letter with a digit not already appearing in the puzzle.
- Goal test: puzzle contains only digits, and represents a correct sum.
- Path cost: zero. All solutions equally valid.

- Missionaries and cannibals
- In the missionaries and cannibals problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board.
- Find the smallest number of crossings that will allow everyone to cross the river safely.
- States: Combination of missionaries, cannibals, and boat on each side of river.. Thus, the start state is (3,3,1).
- Operators: from each state the possible operators are to take either one missionary, one cannibal, two missionaries, two cannibals, or one of each across in the boat. (Raid the boat with maximum two persons across the river in either direction)
- Goal test: reached state (0,0,0)-Move all missionaries and cannibals across the river.
- Path cost: number of crossings(Require the minimum number of crossings)


5
ANSWER
LET'S GO
BACK
HELP
SHARE

Real-world problems

- **Route finding**

Route-finding algorithms are used in a variety of applications, such as routing in computer networks, automated travel advisory systems, and airline travel planning systems.

- **Touring and travelling sales person problems**

The travelling salesperson problem (TSP) is a famous touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.

- **VLSI layout**

A typical VLSI chip can have as many as a million gates, and the positioning and connections of every gate are crucial to the successful operation of the chip. Computer-aided design tools are used in every phase of the process. Two of the most difficult tasks are cell layout and channel routing. These come after the components and connections of the circuit have been fixed; the purpose is to lay out the circuit on the chip so as to minimize area and connection lengths, thereby maximizing speed.

- **Robot navigation**

Robot navigation is a generalization of the route-finding problem.

- **Assembly sequencing**

Automatic assembly of complex objects by a robot was first demonstrated by FREDDY the robot (Michie, 1972). Progress since then has been slow but sure, to the point where assembly of objects such as electric motors is economically feasible. In assembly problems, the problem is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.

SEARCHING FOR SOLUTIONS

- Data structures for search trees There are many ways to represent nodes, we will assume a node is a data structure with five components:
- the state in the state space to which the node corresponds
- the node in the search tree that generated this node (this is called the parent node);
- the operator that was applied to generate the node;
- the number of nodes on the path from the root to this node (the depth of the node);
- the path cost of the path from the initial state to the node.
- The node data type is thus:

datatype node

components: STATE, PARENT-NODE, OPERATOR, DEPTH, PATH-COST

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

Figure 3.9 An informal description of the general search algorithm.

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Figure 3.10 The general search algorithm. (Note that QUEUING-FN is a variable whose value will be a function.)

SEARCH STRATEGIES

- we will evaluate strategies in terms of four criteria:
- COMPLETENESS: is the strategy guaranteed to find a solution when there is one?
- TIME COMPLEXITY :how long does it take to find a solution?
- SPACE COMPLEXITY :how much memory does it need to perform the search?
- OPTIMALITY: does the strategy find the highest-quality solution when there are several different solutions?

Search Algorithm Terminologies

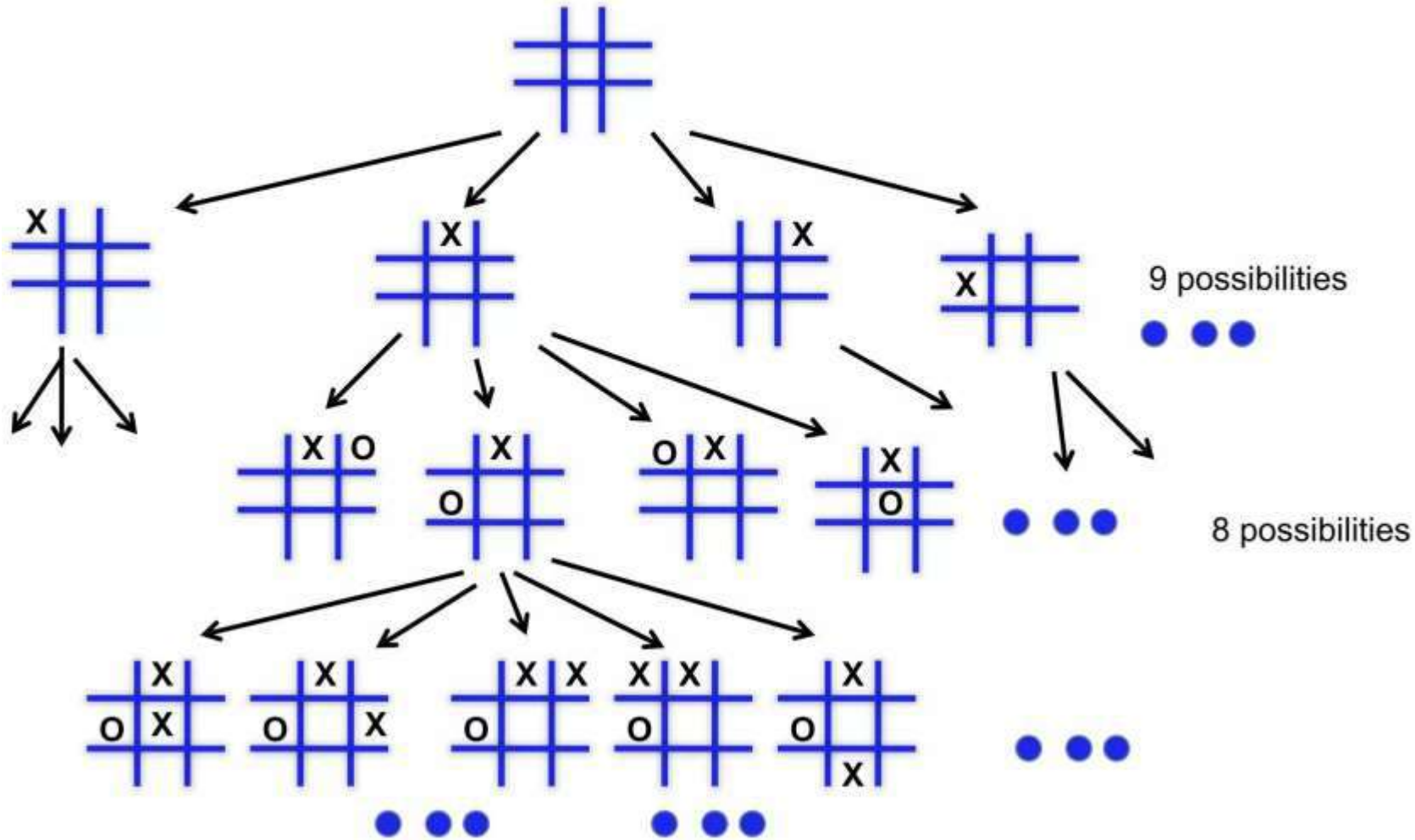
- **Search:**

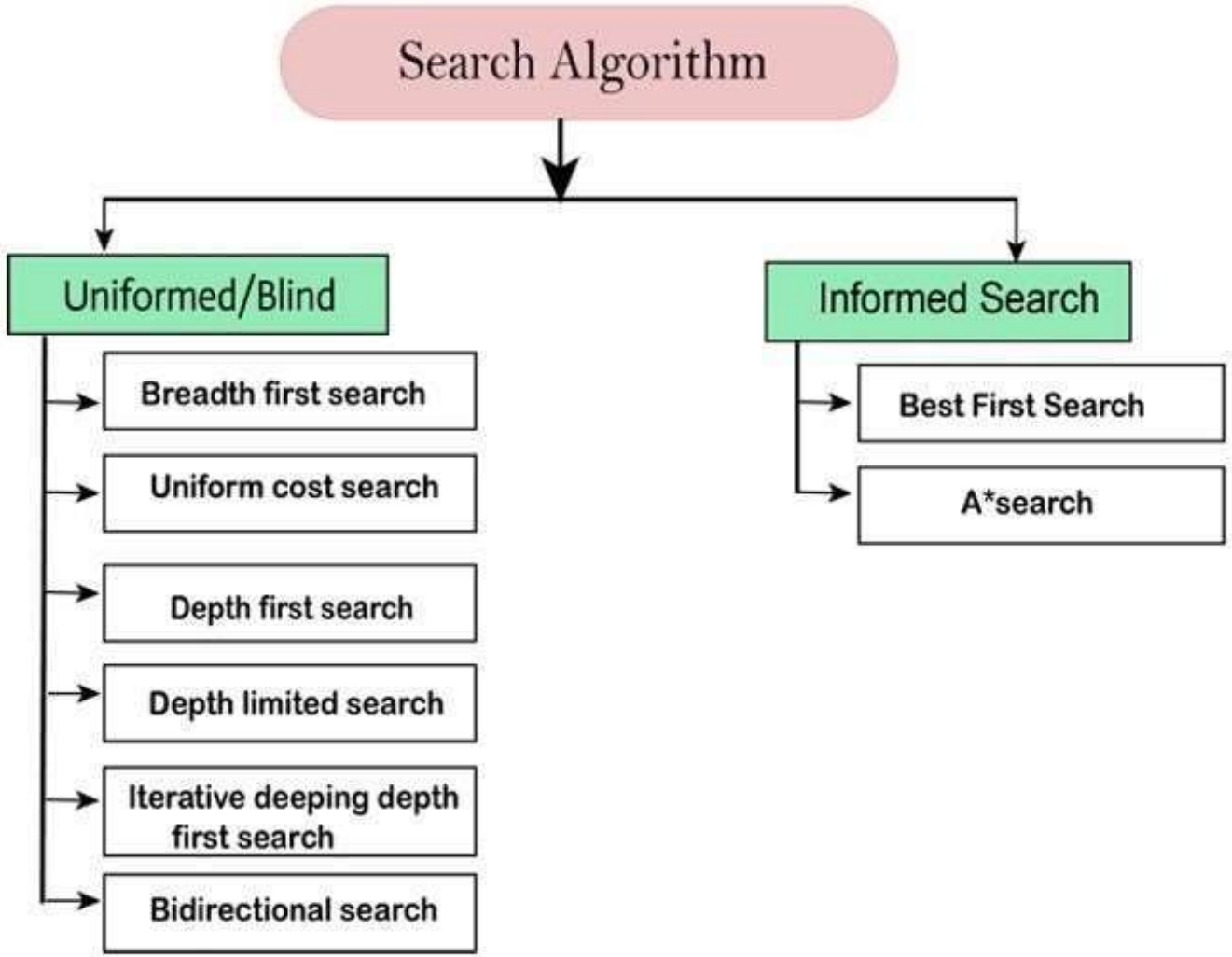
  Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

  1. Search Space: Search space represents a set of possible solutions, which a system may have.
  2. Start State: It is a state from where agent begins the search.
  3. Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.

- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

State Space Tree




Search Algorithm
Uniformed/Blind
Breadth first search
Uniform cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Informed Search
Best First Search
A*search

- Uninformed search: The term means that they have no information about the number of steps or the path cost from the current state to the goal—all they can do is distinguish a goal state from a nongoal state. Uninformed search is also sometimes called blind search.
- Informed search or heuristic search : informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path.

BREADTH-FIRST SEARCH

- Breadth-first search is the most common search strategy for traversing a tree or graph.
- This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

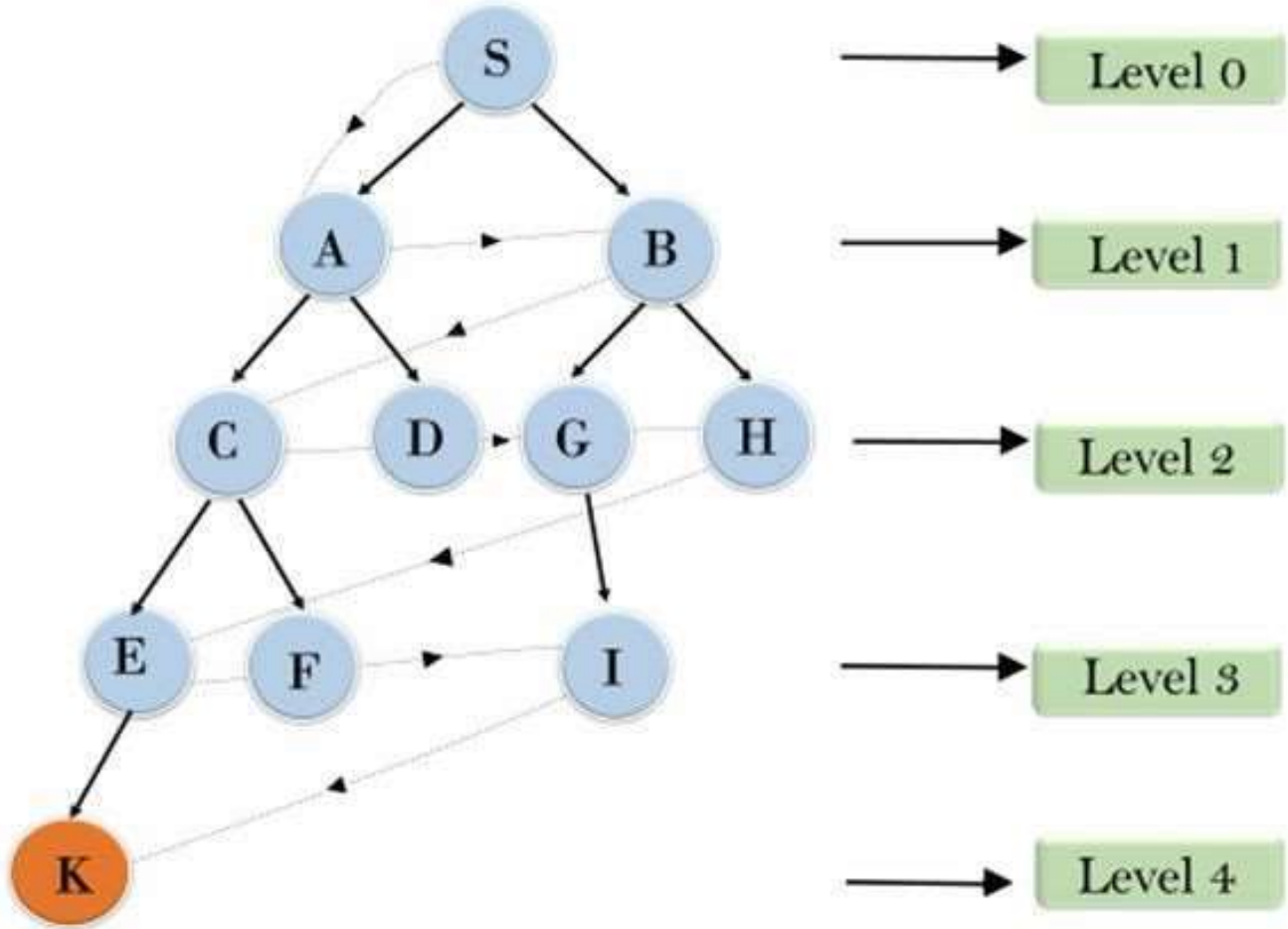
- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

- it requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node. Example :

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be: S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

Breadth First Search



Time Complexity:

- Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$$\Gamma(b) = 1+b+b^2+b^3+ \ldots + b^d = O(b^d)$$

Space Complexity:

- Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

Completeness:

- BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality:

- BFS is optimal if path cost is a non-decreasing function of the depth of the node.

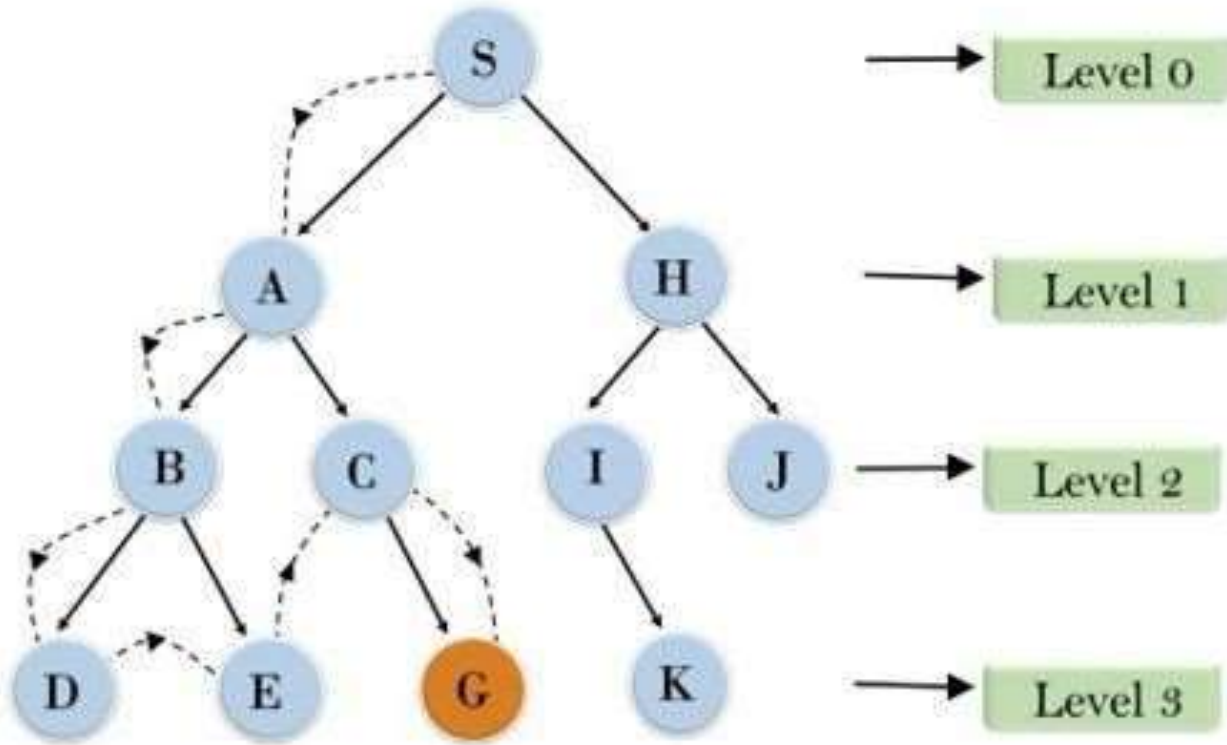
DEPTH-FIRST SEARCH

- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation. Advantage:
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Depth First Search



- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.
- Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n)= 1+ n^2+ n^3 + \ldots\ldots\ldots + n^m=O(n^m)$$

- Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)
- Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).
- Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

INFORMED SEARCH (Heuristic Search)

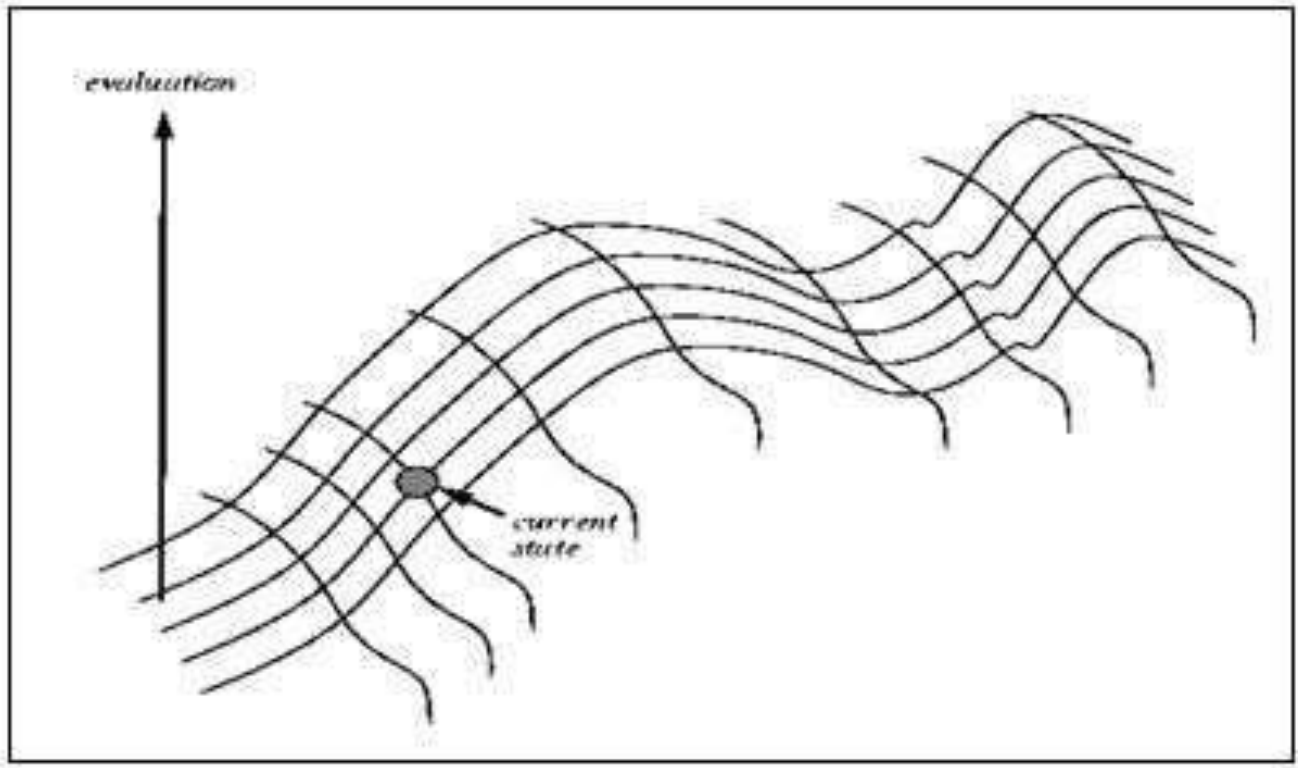
- Informed search algorithms use domain knowledge.
- In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy.
- Informed search is also called a Heuristic search.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

- Admissibility of the heuristic function is given as:
- $h(n) <= h^*(n)$
- Here h(n) is heuristic cost, and h*(n) is the estimated cost.
- Hence heuristic cost should be less than or equal to the estimated cost.

HILL CLIMBING SEARCH

- Hill Climbing Algorithm
- 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.


evaluation
current
state

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: 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
```

Figure 4.14 The hill-climbing search algorithm.

- 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

- Plateau: a plateau is an area of the state space where the evaluation function is essentially flat. The search will conduct a random walk.
- Local maxima: a local maximum, as opposed to a global maximum, is a peak that is lower than the highest peak in the state space. Once on a local maximum, the algorithm will halt even though the solution may be far from satisfactory.
- Ridges: a ridge may have steeply sloping sides, so that the search reaches the top of the ridge with ease, but the top may slope only very gently toward a peak. Unless there happen to be operators that move directly along the top of the ridge, the search may oscillate from side to side, making little progress.

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.

- 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 E 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 SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability of downward steps

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

Figure 4.15 The simulated annealing search algorithm.

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.

Introduction to Artificial Intelligence

UNIT-III

- First-Order Logic - 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
- Forward Chaining
- Backward Chaining
- Resolution
- Knowledge Representation: Ontological Engineering
- Categories and Objects
- Events

First-Order Logic

- In propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. so we required some more powerful logic, such as **first-order logic.**
- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- First-order logic is also known as **Predicate logic or First-order predicate logic.** First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.

- **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ……
- **Relations:** **It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between
- **Function:** Father of, best friend, third inning of, end of, ……
- As a natural language, first-order logic also has two main parts:
  - **Syntax**
  - **Semantics**

Syntax and semantics of First-Order logic:

- The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.
- **Basic Elements of First-order logic:**
- Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

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

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as Predicate (term1, term2, ……, term n).
- **Example: Ravi and Ajay are brothers: ⇒ Brothers(Ravi, Ajay).**
  Chinfiy is a cat: ⇒ cat (Chinfiy).

Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
  - **Universal Quantifier, (for all, everyone, everything)**
  - **Existential quantifier, (for some, at least one).**

Universal Quantifier:

- Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.
- The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.
- In universal quantifier we use implication "→".

- If x is a variable, then ∀x is read as:

For all x

For each x

For every x.

Example:

All man drinfi coffee.

∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

- Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.
- It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

- In Existential quantifier we always use AND or Conjunction symbol (∧).
- If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

There exists a 'x.'

For some 'x.'

For at least one 'x.'

- Example:

Some boys are intelligent.

∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

- The main connective for universal quantifier $\forall$ is implication $\rightarrow$.
- The main connective for existential quantifier $\exists$ is and $\wedge$.

Properties of Quantifiers:

- In universal quantifier, $\forall x \forall y$ is similar to $\forall y \forall x$.
- In Existential quantifier, $\exists x \exists y$ is similar to $\exists y \exists x$.
- $\exists x \forall y$ is not similar to $\forall y \exists x$.

- Nested quantifiers:We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type.
- For example, “Brothers are siblings” can be written as

$\forall x \ \forall y$ Brother(x,y) $\Rightarrow$ Sibling(x,y) . Or

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

“Everybody loves somebody” means that for every person, there is someone that person loves: $\forall x \ \exists y$ Loves(x,y)

On the other hand, to say “There is someone who is loved by everyone,” we write $\exists y \ \forall x$ Loves(x,y) .

- Connections between $\forall$ and $\exists$:The two quantifiers are actually intimately connected with each other, through negation.

- “Everyone likes ice cream” means that there is no one who does not like ice cream:

∀ x Likes(x,IceCream) is equivalent to ¬ ∃ x ¬Likes(x,IceCream) .

- Equality:First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the equality symbol to signify that two terms refer to the same object. For example, Brother (John)=Henry
- Ex: ¬(x=y) which is equivalent to $x \neq y$.

- Some Examples of FOL using quantifier:

1. All birds fly.

$\forall$x bird(x) $\rightarrow$fly(x).

2. Every man respects his parent.

In this, the predicate is "**respect(x, y),**" **where x=man, and y= parent.**

$\forall$x man(x) $\rightarrow$ respects (x, parent).

3. Some boys play cricfiet.

In this, the predicate is "**play(x, y),**" where x= boys, and y= game.

$\exists$x boys(x) $\rightarrow$ play(x, cricfiet).

4. Not all students lifie both Mathematics and Science.

In this, the predicate is "**lifie(x, y),**" **where x= student, and y= subject.**

Convert the following sentences into predicate logic.

i. x is hungry.

ii. "Alice is a parent of Bob.“

iii. x is married to y if and only if x is a spouse of y.“

iv. "if x is human, then x is mortal.“

v. All white birds are beautiful.

vi. All Students are smart

vii. Every Macintosh computer uses electricity

viii. All mammals are warm blooded.

Using First-order logic

Assertions and queries in first order logic:

- Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called ASSERTIONS.
- TELL(KB, King(John)) .
- TELL(KB, Person(Richard)) .
- TELL(KB, $\forall$ x King(x) $\Rightarrow$ Person(x)) .
- Questions asked with ASK are called queries or goals
- ASK(KB, Person(John))
- ASK(KB, $\exists$ x Person(x)) .-returns true not useful
- ASKVARS(KB,Person(x))- : return a stream/list of all possible binding. i.e,returns{x/John} and {x/Richard}.
- Such an answer is called a substitution or binding list.

The kinship domain:

- example we consider is the domain of family relationships, or kinship. This domain includes facts such as “Elizabeth is the mother of Charles” and “Charles is the father of William” and rules such as “One’s grandmother is the mother of one’s parent.”
- The objects in our domain are people. We have two unary predicates, Male and Female.
- Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt , and Uncle.
- We use functions for Mother and Father, because every person has exactly one of each of these.

- one’s mother is one’s female parent:

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

- One’s husband is one’s male spouse:

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

- Male and female are disjoint categories:

$\forall x\ Male(x) \Leftrightarrow \neg Female(x)$.

- Parent and child are inverse relations:

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

Numbers, sets, and lists

NUMBERS

- We describe here the theory of natural numbers or non-negative integers. We need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor).

  NatNum(0) . $\forall$ n NatNum(n) $\Rightarrow$ NatNum(S(n)) .

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

SETS

- Adjoining an element already in the set has no effect:

  $\forall x,s\ x \in s \Leftrightarrow s=\{x|s\}$.

- A set is a subset of another set if and only if all of the first set's members are members of the second set:

  $\forall s1,s2\ s1 \subseteq s2 \Leftrightarrow (\forall x\ x \in s1 \Rightarrow x \in s2)$.

- Two sets are equal if and only if each is a subset of the other: $\forall s1,s2\ (s1 = s2) \Leftrightarrow (s1 \subseteq s2 \wedge s2 \subseteq s1)$.

- An object is in the intersection of two sets if and only if it is a member of both sets:

  $\forall x,s1,s2\ x \in (s1 \cap s2) \Leftrightarrow (x \in s1 \wedge x \in s2)$.

- An object is in the union of two sets if and only if it is a member of either set:

  $\forall x,s1,s2\ x \in (s1 \cup s2) \Leftrightarrow (x \in s1 \vee x \in s2)$.

LISTS

- Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list.
- We can use the vocabulary of Lisp for lists: Nil is the constant list with no elements; Cons, Append, First, and Rest are functions
- The empty list is [].
- The term Cons(x,y),where y is a nonempty list, is written [x|y].
- The term Cons(x,Nil) (i.e., the list containing the element x) is written as [x].
- A list of several elements, such as [A,B,C], corresponds to the nested term Cons(A,Cons(B,Cons(C,Nil)))

Knowledge Engineering in First-order logic

- The process of constructing a knowledge-base in first-order logic is called as knowledge-engineering. In knowledge-engineering, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as knowledge engineer.

- In this topic, we will understand the Knowledge engineering process in an electronic circuit domain, which is already familiar. This approach is mainly suitable for creating special-purpose knowledge base.

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

1. Identify the task.

2. Assemble the relevant knowledge.

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

4. Encode general knowledge about the domain.

5. Encode a description of the specific problem instance.

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

7. Debug the knowledge base.

1. Identify the task.

- Figure out the range of questions that can be asked.
- Figure out the range of facts available for each specific instance of the problem.

2. Assemble the relevant knowledge.

- Engineer needs to be an expert in the domain or might need to work with real experts to extract what they know-this process is called knowledge acquisition.
- It is similar to requirements analysis-just getting an rough idea about the KB scope and what are the rules.

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

- Translate domain-level concepts into logic-level names.
- We need to decide which one should be objects or unary predicates.
- We need to decide which are functions and binary predicates.
- Once decided, 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.
- The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

4. Encode general knowledge about the domain.

The knowledge engineer writes down the axioms for all the vocabulary terms.

- Tries to specify the meaning of the terms , and enabling the expert to check the content.
- In this step we can elaborate on issues that need to be fixed by repeating the previous step.

5. Encode a description of the specific problem instance.

When ontology is well defined, It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors.

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

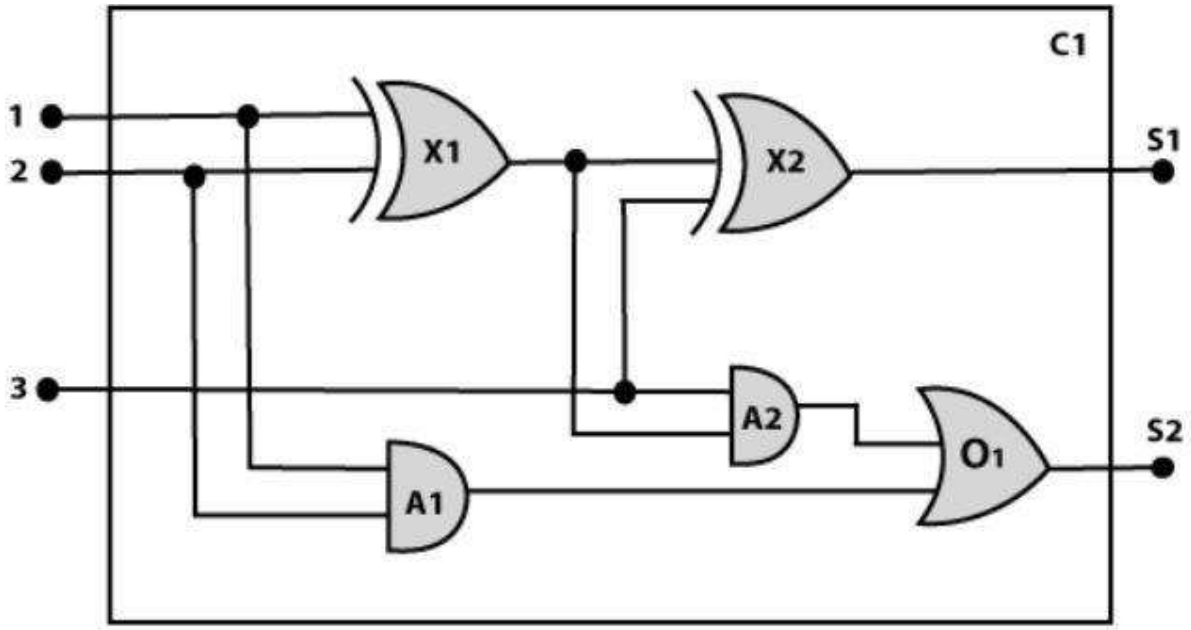
we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.

7. Debug the knowledge base.

- The answers to queries will rarely be correct on the first try. More precisely, the answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting.
- For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly.

The knowledge-engineering process:

- Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (**One-bit full adder**) which is given below



1. Identify the task:

- The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks.
- At the first level or highest level, we will examine the functionality of the circuit:
  - **Does the circuit add properly?**
  - **What will be the output of gate A2, if all the inputs are high?**
- At the second level, we will examine the circuit structure details such as:
  - **Which gate is connected to the first input terminal?**
  - **Does the circuit have feedbacfi loops?**

2. Assemble the relevant knowledge:

- In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:
- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: **AND, OR, XOR, and NOT**.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

- The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate(X1)**. The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit (C1)**.
- For the terminal, we will use predicate: **Terminal(x)**.
- For gate input, we will use the function **In(1, X1)** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, X1)**.
- The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.
- The connectivity between gates can be represented by predicate **Connect(Out(1, X1), In(1, X2))**.
- We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

- To encode the general knowledge about the logic circuit, we need some following rules:
- If two terminals are connected then they have the same input signal, it can be represented as:

$\forall$ t1, t2 Terminal (t1) $\wedge$ Terminal (t2) $\wedge$ Connect (t1, t2) $\rightarrow$ Signal (t1) = Signal (t2).

- Signal at every terminal will have either value 0 or 1, it will be represented as:

$\forall$ t Terminal (t) $\rightarrow$ Signal (t) = 1 $\vee$ Signal (t) = 0.

- Connect predicates are commutative:

$\forall$ t1, t2 Connect(t1, t2) $\rightarrow$ Connect (t2, t1).

- Representation of types of gates:

$\forall$ g Gate(g) $\wedge$ r = Type(g) $\rightarrow$ r = OR $\vee$ r = AND $\vee$ r = XOR $\vee$ r = NOT.

- Output of AND gate will be zero if and only if any of its input is zero.

$\forall$ g Gate(g) $\wedge$ Type(g) = AND $\rightarrow$ Signal (Out(1, g)) = 0 $\Leftrightarrow$ $\exists$ n Signal (In(n, g)) = 0.

- Output of OR gate is 1 if and only if any of its input is 1:

$\forall$ g Gate(g) $\wedge$ Type(g) = OR $\rightarrow$ Signal (Out(1, g)) = 1 $\Leftrightarrow$ $\exists$ n Signal (In(n, g)) = 1

- Output of XOR gate is 1 if and only if its inputs are different:

$\forall$ g Gate(g) $\wedge$ Type(g) = XOR $\rightarrow$ Signal (Out(1, g)) = 1 $\Leftrightarrow$ Signal (In(1, g)) $\neq$ Signal (In(2, g)).

- Output of NOT gate is invert of its input:

$\forall$ g Gate(g) $\wedge$ Type(g) = NOT $\rightarrow$ Signal (In(1, g)) $\neq$ Signal (Out(1, g)).

- All the gates in the above circuit have two inputs and one output (except NOT gate).

$\forall$ g Gate(g) $\wedge$ Type(g) = NOT $\rightarrow$ Arity(g, 1, 1)

$\forall$ g Gate(g) $\wedge$ r =Type(g) $\wedge$ (r= AND $\vee$r= OR $\vee$r= XOR) $\rightarrow$ Arity (g, 2, 1).

- All gates are logic circuits:

$\forall$ g Gate(g) $\rightarrow$ Circuit (g).

5. Encode a description of the problem instance:

- Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.
- For the given circuit C1, we can encode the problem instance in atomic sentences as below:
- Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

1. For XOR gate: Type(x1)= XOR, Type(X2) = XOR
2. For AND gate: Type(A1) = AND, Type(A2)= AND
3. For OR gate: Type (O1) = OR.

- And then represent the connections between all the gates.

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

- In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:
- What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

1. ∃ i1, i2, i3 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2 ∧ Signal (In(3, C1))= i3 ∧ Signal (Out(1, C1)) =0 ∧ Signal (Out(2, C1))=1

7. Debug the knowledge base:

- Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.
- In the knowledge base, we may have omitted assertions like

Inference in First-Order Logic

- Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand **some basic terminologies** used in FOL.
- **Substitution:**

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F**[**a**/**x**], so it refers to substitute a constant "**a**" in place of variable "**x**".

- **Equality:** First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

  Example: Brother (John) = Smith.

- As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith.** The equality symbol can also be used with negation to represent that two terms are not the same objects.
- **Example: ¬(x=y) which is equivalent to x ≠y.**

- **FOL inference rules for quantifier:**

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

Universal Generalization

Universal Instantiation

Existential Instantiation

Existential introduction

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).

- It can be represented as: $\frac{P(C)}{\forall x\ P(x)}$
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.
- Example: Let's represent, P(c): "A byte contains 8 bits", so for ∀ x P(x) "All bytes contain 8 bits.", it will also be true.

2. Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.
- The new KB is logically equivalent to the previous KB.
- As per UI, we can infer any sentence obtained by substituting a ground term for the variable.
- The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from ∀ x P(x) for any object in the universe of discourse.
- It can be represented as: $\frac{\forall x\ P(x).}{P(C)}$

- **Example:1.**

IF "Every person like ice-cream"=> $\forall x\ P(x)$ so we can infer that

"John likes ice-cream" => P(c)

- **Example: 2.**

Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:

$\forall x\ king(x) \wedge greedy\ (x) \rightarrow Evil\ (x),$

So from this information, we can infer any of the following statements using Universal Instantiation:

$King(John) \wedge Greedy\ (John) \rightarrow Evil\ (John),$

$King(Richard) \wedge Greedy\ (Richard) \rightarrow Evil\ (Richard),$

$King(Father(John)) \wedge Greedy\ (Father(John)) \rightarrow Evil\ (Father(John)),$

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\ P(x)}{P(C)}$
- **Example:**

From the given sentence: ∃x Crown(x) ∧ OnHead(x, John),

- So we can infer: Crown(K) ∧ OnHead(K, John), as long as K does not appear in the knowledge base.
- The above used K is a constant symbol, which is called Skolem constant.
- The Existential instantiation is a special case of Skolemization process.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(C)}{\exists x\ P(x)}$
- **Example:** Let's say that,

"Priyanka got good marks in English."

"Therefore, someone got good marks in English."

Unification

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ1 and Ψ2 be two atomic sentences and σ be a unifier such that, $\Psi 1\sigma = \Psi 2\sigma$, then it can be expressed as UNIFY(Ψ1, Ψ2).
- Example: Find the MGU for Unify{King(x), King(John)}
- Let Ψ1 = King(x), Ψ2 = King(John),

- Substitution θ= {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).

- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.
- E.g. Let's say there are two different expressions, P(x, y), and P(a, f(z)).

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)..... (i)

P(a, f(z))..... (ii)

Substitute x with a, and y with f(z) in the first expression, and it will be represented as a/x and f(z)/y.

- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: [a/x, f(z)/y].

Conditions for Unification:

- Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

- **Unification Algorithm:**

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

  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(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

---

function UNIFY-VAR(var, x, θ) **returns** a substitution

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

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B)

- **Implementation of the Algorithm**
- **Step.1:** Initialize the substitution set to be empty.
- **Step.2:** Recursively unify atomic sentences:

1. Check for Identical expression match.
2. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:
   1. Substitute t_i / v_i in the existing substitutions
   2. Add t_i /v_i to the substitution setlist.
   3. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

Forward Chaining

- Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.
- The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward –chaining approach is commonly used in the expert system, such as CLIPS(C Language Integrated Production System), business, and production rule systems.

Example:

- **"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."**

  Prove that **"Robert is criminal."**

- To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL:

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

and r are variables).

- **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)**
- Country A has some missiles. **∃p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation,

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

$\exists$ p Missiles(p) $\wedge$ Owns (A, p) $\rightarrow$ Sells (Robert, p, A) .....(4)

- Missiles are weapons.

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

- Enemy of America is known as hostile.

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

- Country A is an enemy of America.

Enemy (A, America) .....(7)

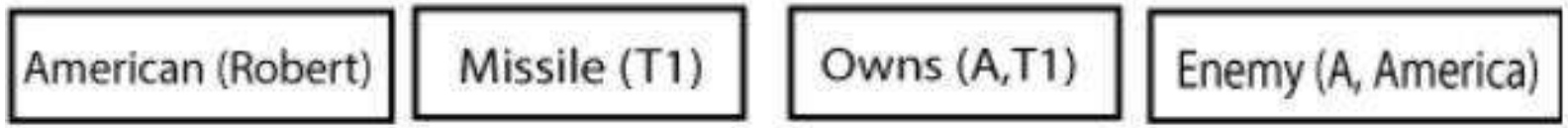
- Robert is American

American(Robert). .....(8)

Forward chaining proof:

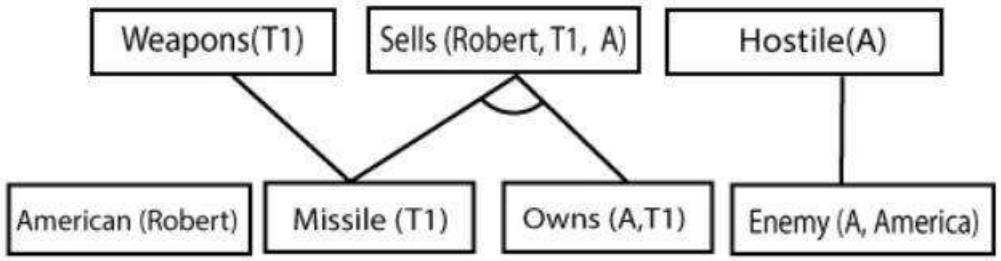
Step-1:

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



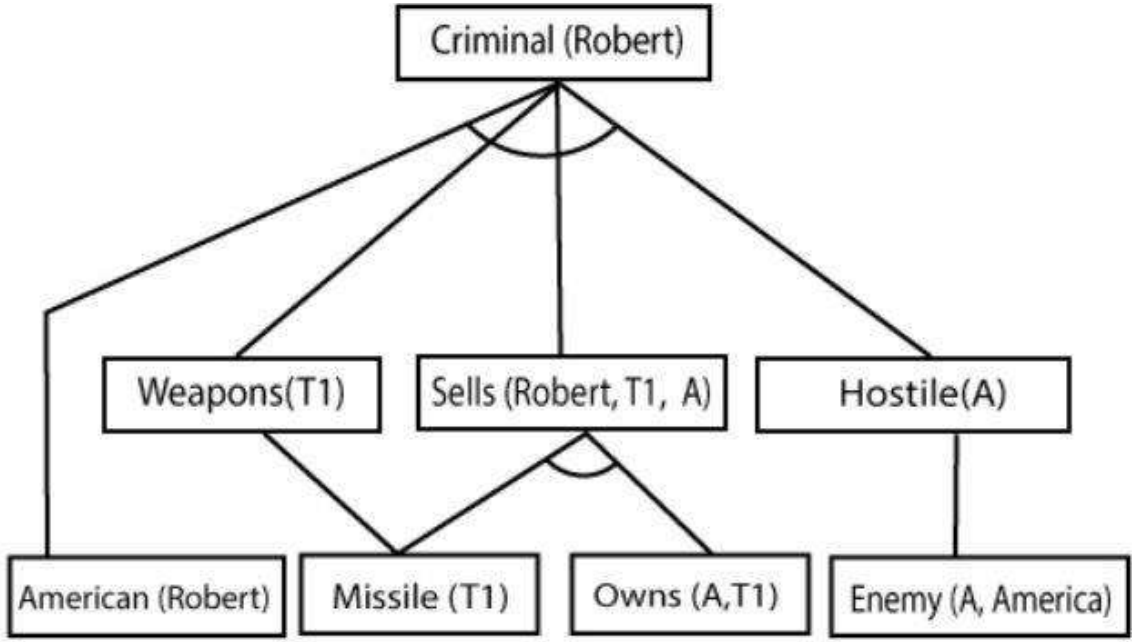
Step-2:

- At the second step, we will see those facts which infer from available facts and with satisfied premises.
- Rule–(1) does not satisfy premises, so it will not be added in the first iteration.
- Rule–(2) and (3) are already added.
- Rule–(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).
- Rule–(6) is satisfied with the substitution(p/A), so Hostile(A) is added and v

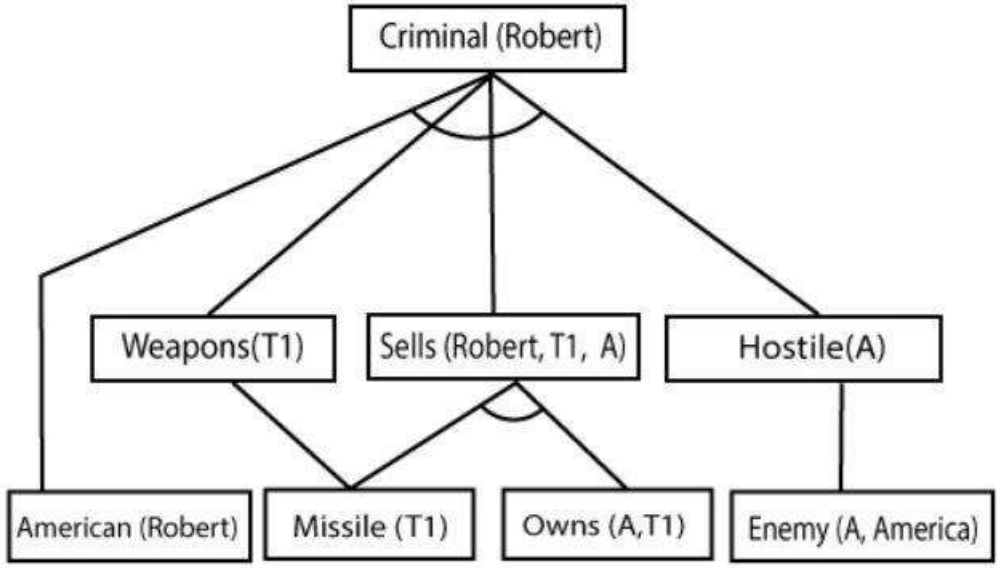


Step-3:

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



Forward chaining proof:



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

Backward Chaining:

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

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.

- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a depth-first search strategy for proof.

Example:

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

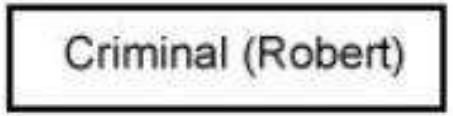
- American (p) ⋀ weapon(q) ⋀ sells (p, q, r) ⋀ hostile(r) → Criminal(p) ...(1)
- Owns(A, T1) ……..(2)
- Missile(T1) …….(3)
- ?p Missiles(p) ⋀ Owns (A, p) → Sells (Robert, p, A) ……(4)
- Missile(p) → Weapons (p) …….(5)
- Enemy(p, America) →Hostile(p) ……..(6)
- Enemy (A, America) ……..(7)
- American(Robert). …….(8)

Backward-Chaining proof:

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

Step-1:

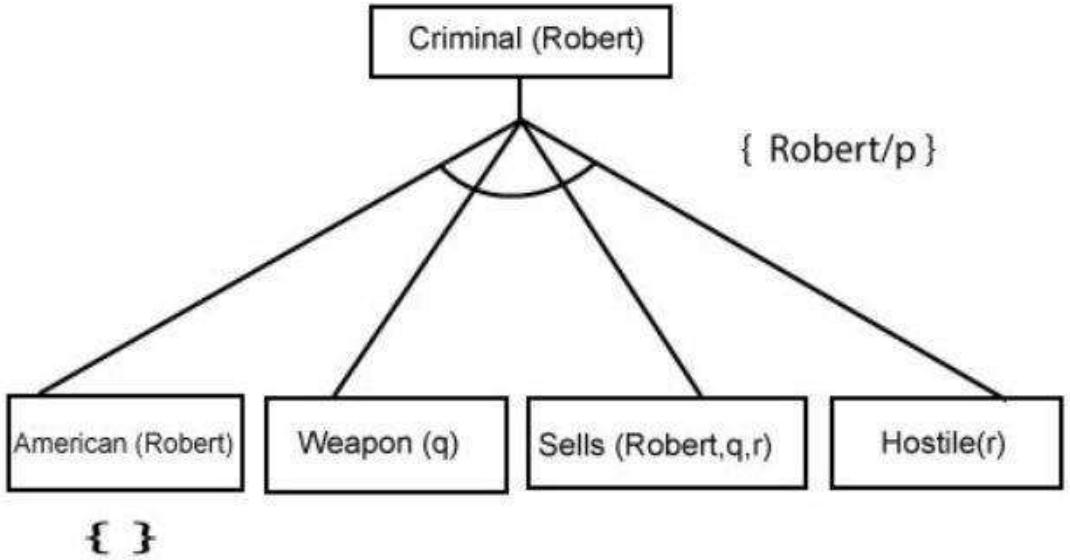
- At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.



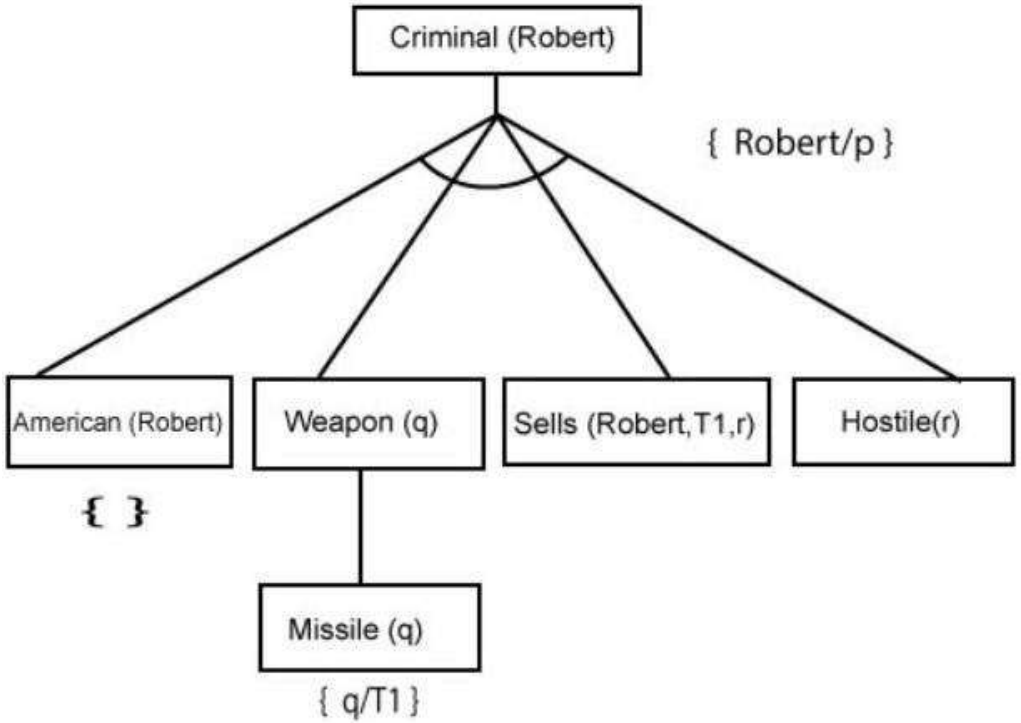
Step-2:

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

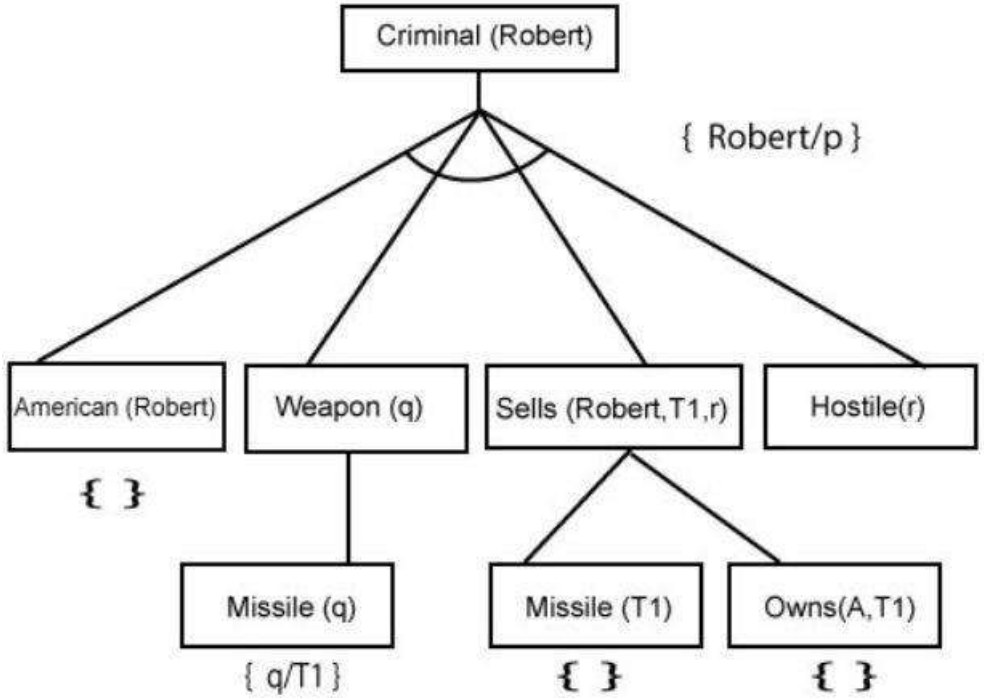
- Here we can see American (Robert) is a fact, so it is proved here.



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

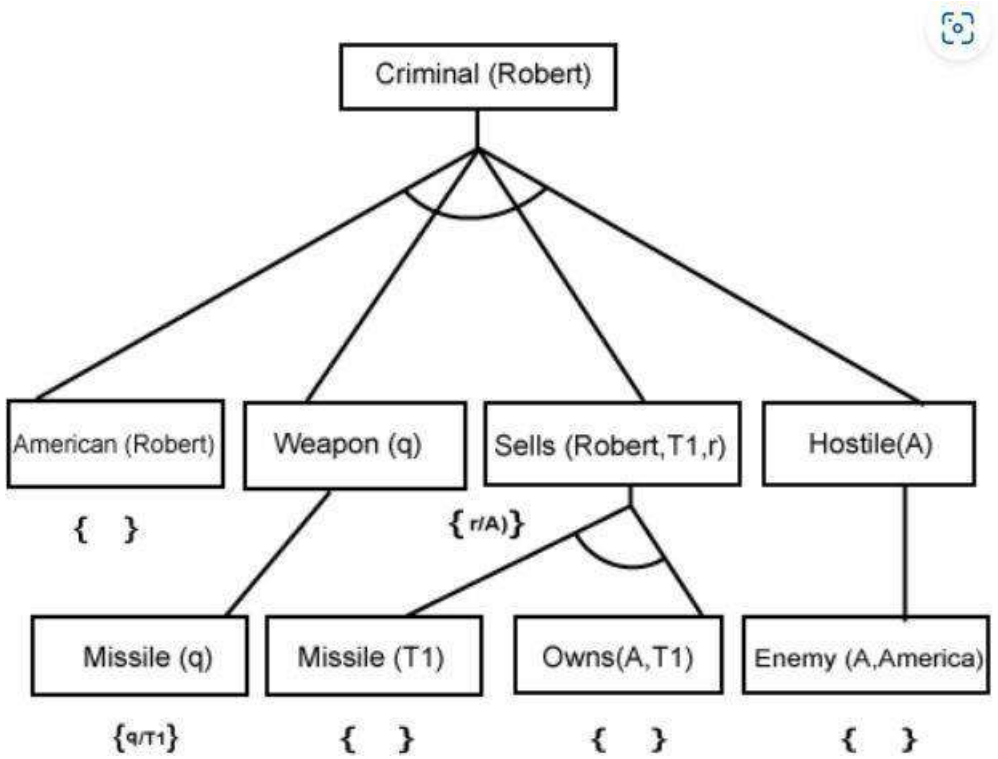


- **Step-4:**
- At step–4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.

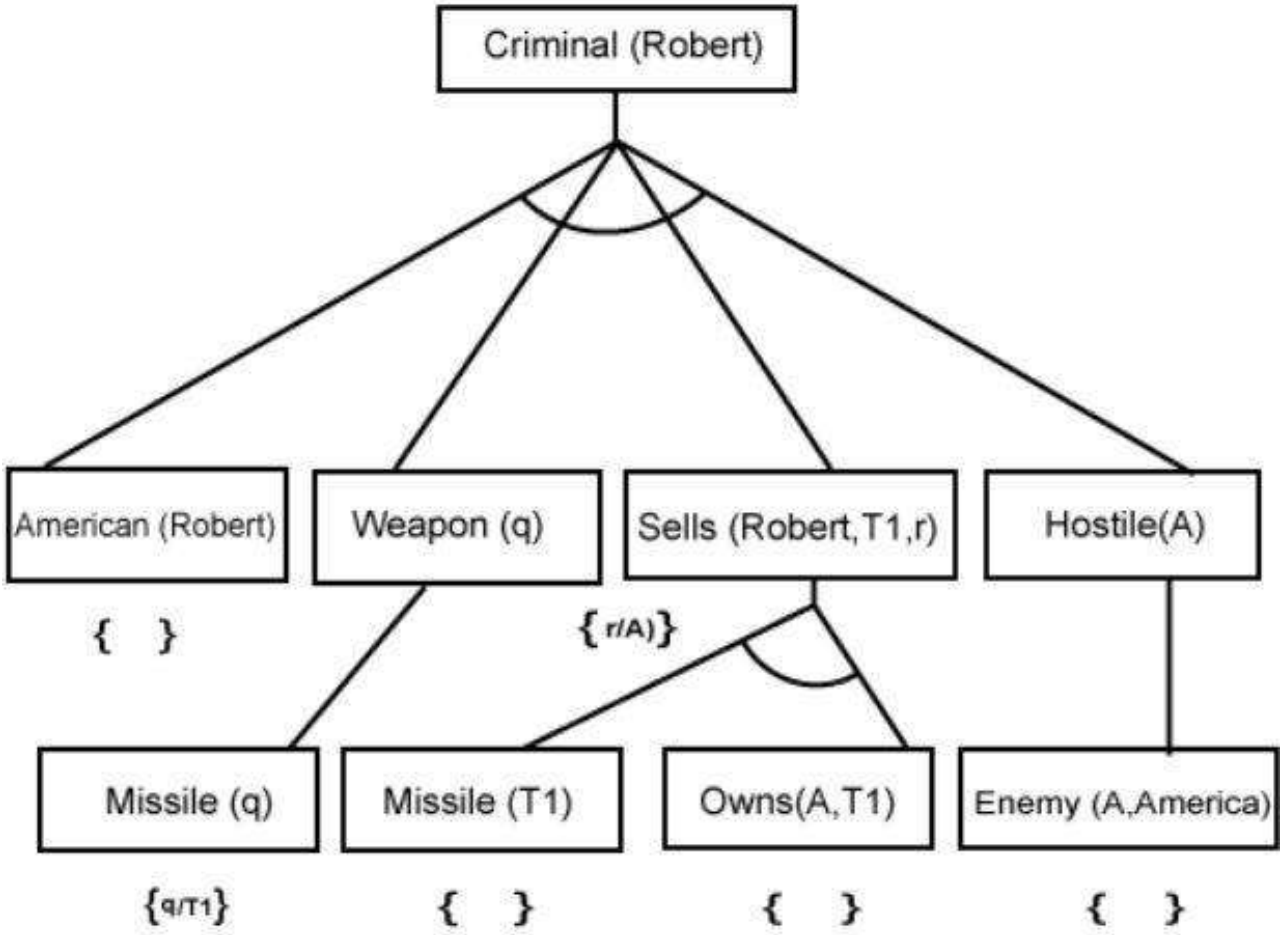


Step-5:

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



- Backward-Chaining proof:



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

Resolution

Conjunctive normal form for first-order logic

- Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements.
- Unification is a key concept in proofs by resolutions.
- Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form.**
- **Clause**: Disjunction of literals (an atomic sentence) is called a **clause.** It is also known as a unit clause.
- **Conjunctive Normal Form**: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF.**

The resolution inference rule:

- The resolution rule for first-order logic is simply a lifted version of the propositional rule. Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

$$\frac{l_1 \vee \ldots\ldots \vee l_k, \qquad m_1 \vee \ldots\ldots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \ldots\ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots\ldots \vee l_k \vee m_1 \vee \ldots\ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots\ldots \vee m_n)}$$

- Where l_i and m_j are complementary literals.
- This rule is also called the **binary resolution rule** because it only resolves exactly two literals.

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

Example:

1. John lifies all fiind of food.
2. Apple and vegetable are food
3. Anything anyone eats and not fiilled is food.
4. Anil eats peanuts and still alive
5. Harry eats everything that Anil eats.
   Prove by resolution that:
6. John lifies peanuts.

Step-1: Conversion of Facts into FOL

- In the first step we will convert all the given statements into its first order logic.

a. $\forall x$: food(x) $\rightarrow$ likes(John, x)

b. food(Apple) $\wedge$ food(vegetables)

c. $\forall x \forall y$: eats(x, y) $\wedge \neg$ killed(x) $\rightarrow$ food(y)

d. eats (Anil, Peanuts) $\wedge$ alive(Anil).

e. $\forall x$: eats(Anil, x) $\rightarrow$ eats(Harry, x)

f. $\forall x$: $\neg$ killed(x) $\rightarrow$ alive(x)

g. $\forall x$: alive(x) $\rightarrow \neg$ killed(x)

} **added predicates.**

h. likes(John, Peanuts)

- **Step-2: Conversion of FOL into CNF**
- In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.
  - **Eliminate all implication (→) and rewrite**

    a. ∀x ¬ food(x) V likes(John, x)

    b. food(Apple) Λ food(vegetables)

    c. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)

    d. eats (Anil, Peanuts) Λ alive(Anil)

    e. ∀x ¬ eats(Anil, x) V eats(Harry, x)

    f. ∀x¬ [¬ killed(x)] V alive(x)

    g. ∀x ¬ alive(x) V ¬ killed(x)

    h. likes(John, Peanuts).

- **Move negation (¬)inwards and rewrite**

  a. ∀x ¬ food(x) V likes(John, x)

  b. food(Apple) Λ food(vegetables)

  c. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)

  d. eats (Anil, Peanuts) Λ alive(Anil)

  e. ∀x ¬ eats(Anil, x) V eats(Harry, x)

  f. ∀x ¬killed(x)] V alive(x)

  g. ∀x ¬ alive(x) V ¬ killed(x)

  h. likes(John, Peanuts).

- **Rename variables or standardize variables**

  a. ∀x ¬ food(x) V likes(John, x)

  b. food(Apple) Λ food(vegetables)

  c. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)

  d. eats (Anil, Peanuts) Λ alive(Anil)

  e. ∀w¬ eats(Anil, w) V eats(Harry, w)

  f. ∀g ¬killed(g)] V alive(g)

  g. ∀k ¬ alive(k) V ¬ killed(k)

  h. likes(John, Peanuts).

- **Eliminate existential instantiation quantifier by elimination.**

  In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.

- **Drop Universal quantifiers.**
  In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

  a. ¬ food(x) V likes(John, x)

  b. food(Apple)

  c. food(vegetables)

  d. ¬ eats(y, z) V killed(y) V food(z)

  e. eats (Anil, Peanuts)

  f. alive(Anil)

  g. ¬ eats(Anil, w) V eats(Harry, w)

  h. killed(g) V alive(g)

  i. ¬ alive(k) V ¬ killed(k)

  j. likes(John, Peanuts).

Note: Statements "food(Apple) Λ food(vegetables)" and "eats (Anil, Peanuts) Λ alive(Anil)" can be written in two separate statements.

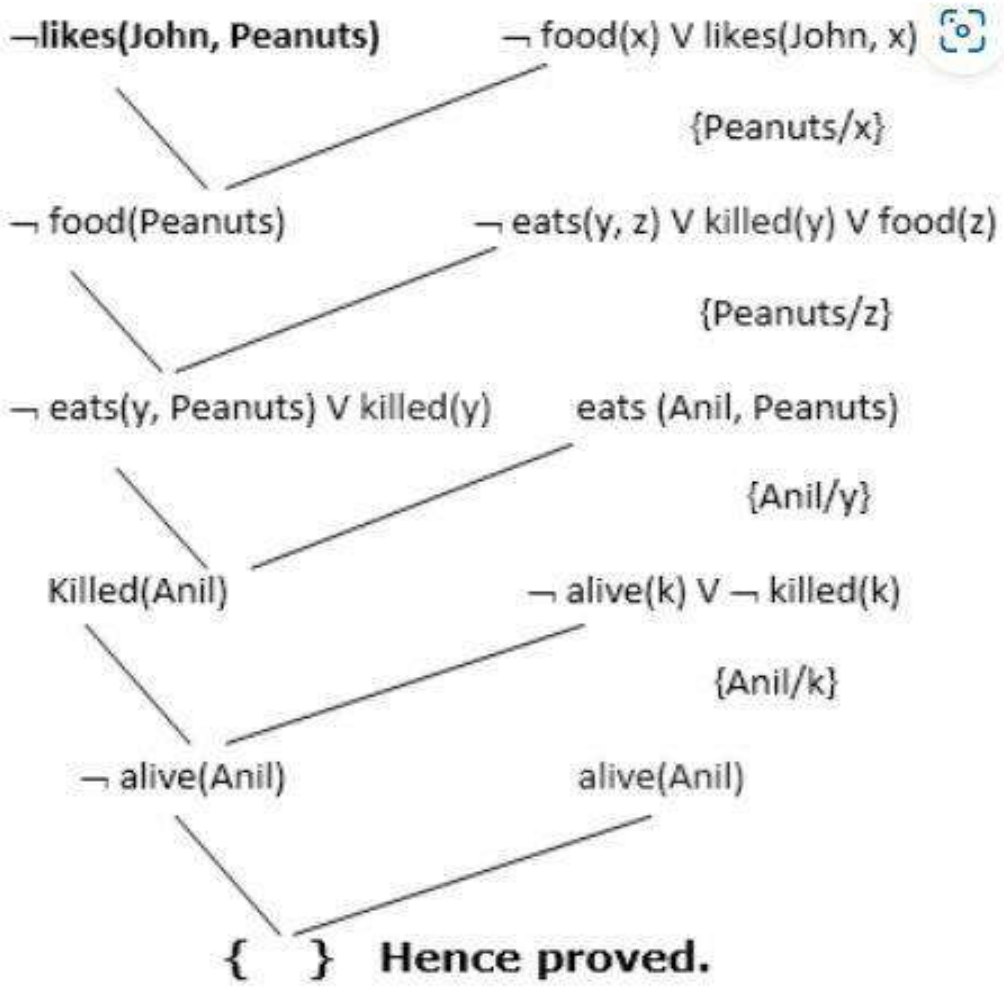
- **Distribute conjunction ∧ over disjunction ¬.**
  This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:

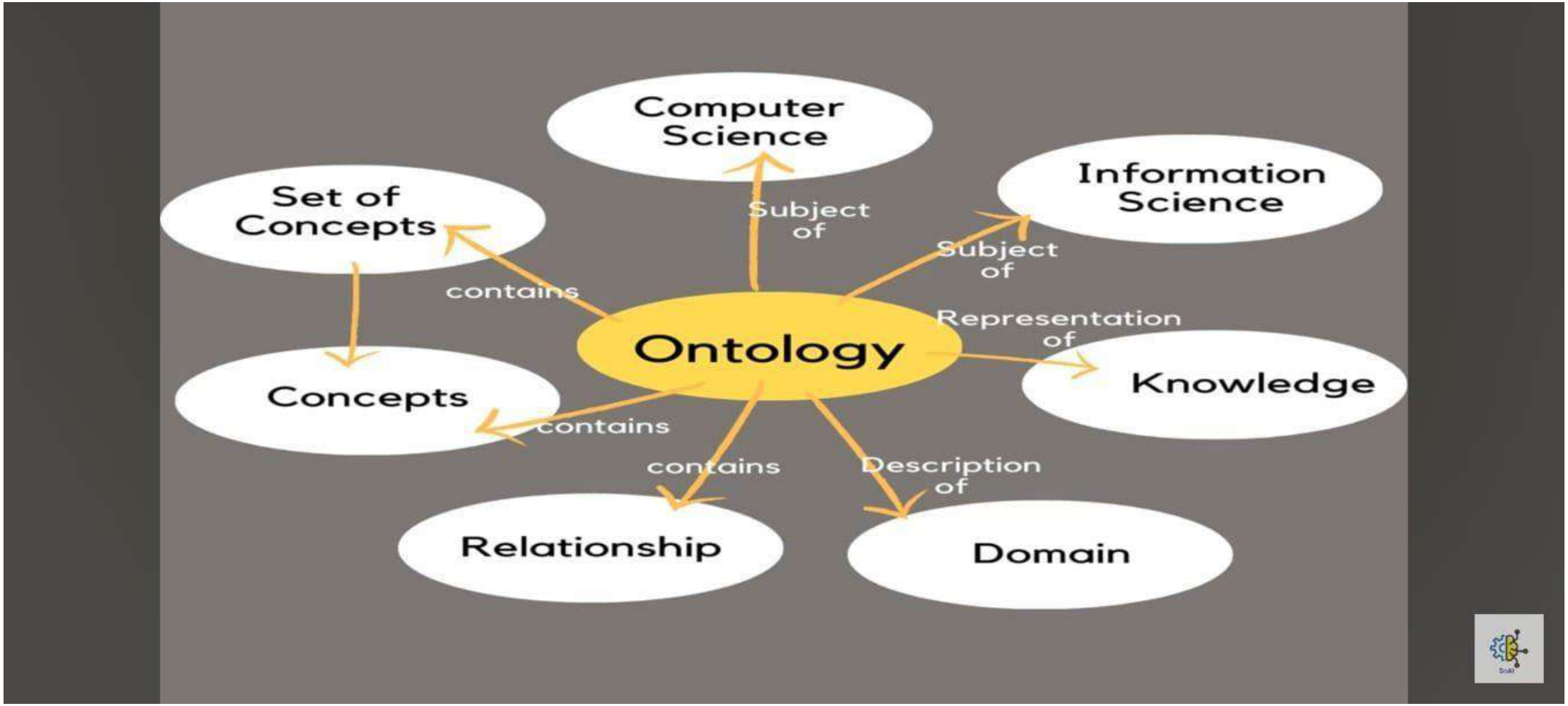


Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

- In the first step of resolution graph, ¬**lifies**(**John, Peanuts**) , and **lifies**(**John, x**) get resolved(canceled) by substitution of {**Peanuts**/**x**}, and we are left with ¬**food**(**Peanuts**)
- In the second step of the resolution graph, ¬ **food**(**Peanuts**) , and **food**(**z**) get resolved (canceled) by substitution of { **Peanuts**/**z**}, and we are left with ¬ **eats**(**y, Peanuts**) **V fiilled**(**y**) .
- In the third step of the resolution graph, ¬ **eats**(**y, Peanuts**) and **eats** (**Anil, Peanuts**) get resolved by substitution {**Anil**/**y**}, and we are left with **Killed**(**Anil**) .
- In the fourth step of the resolution graph, **Killed**(**Anil**) and ¬ **fiilled**(**fi**) get resolve by substitution {**Anil**/**fi**}, and we are left with ¬**alive**(**Anil**) .
- In the last step of the resolution graph ¬ **alive**(**Anil**) and **alive**(**Anil**) get resolved.

Knowledge Representation:
Ontological Engineering



- An ontology in the context of artificial intelligence is a formal and explicit representation of a conceptualization of a domain.
- **Ontologies** are formal definitions of vocabularies that allow us to define difficult or complex structures and new relationships between vocabulary terms and members of classes that we define. Ontologies generally describe specific domains such as scientific research areas.

Different Ontology Languages:

- **CycL –** It was developed for the Cyc project and is based on First Order Predicate Calculus.
- **Rule Interchange Format (RIF) –** It is the language used for combining ontologies and rules.
- **Open Biomedical Ontologies (OBO) –** It is used for various biological and biomedical ontologies.
- **Web Ontology Language (OWL) –** It is developed for using ontologies over the World Wide Web (WWW).

Categories and Objects

- Ontological engineering in AI involves the creation and management of ontologies, which are formal representations of knowledge that define concepts and their relationships in a particular domain.

- Categories and objects are fundamental components of ontological engineering, as they help organize and structure knowledge for various AI applications. Here's a breakdown of categories and objects in AI ontological engineering:

CATEGORIES

- Categories are high-level concepts or classes that help classify and organize objects or entities within a specific domain.
- In ontological engineering, categories are used to group related objects based on their shared characteristics or attributes.
- Categories can have subcategories to create a hierarchical structure that represents the relationships between different concepts. For example, in a medical ontology, "Diseases" might be a category, and under it, you might have subcategories like "Cardiovascular Diseases," "Infectious Diseases," and so on.

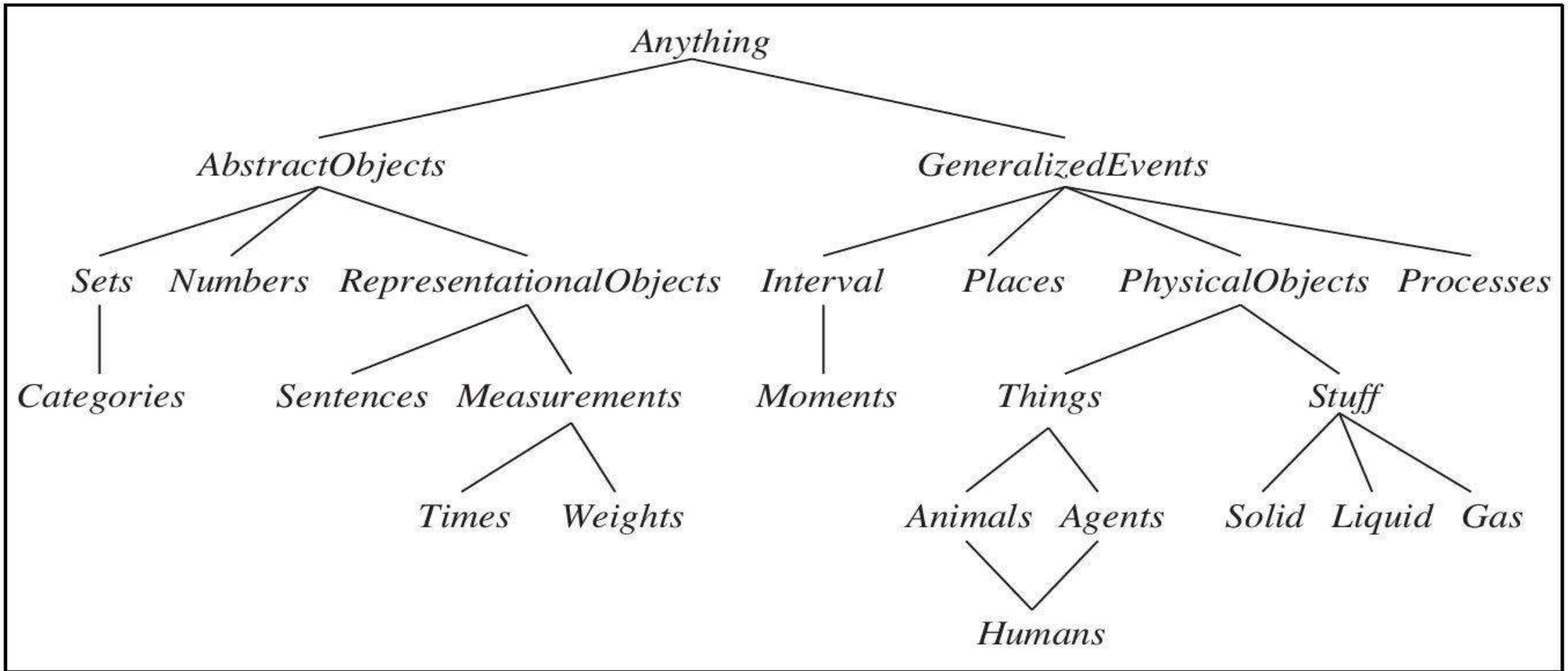


Figure 12.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example. We will see in Section 12.3.3 why physical objects come under generalized events.

Examples

- An object is a member of a category.
  $BB_9 \in Basketballs$
- A category is a subclass of another category.
  $Basketballs \subset Balls$
- All members of a category have some properties.
  $(x \in Basketballs) \Rightarrow Spherical(x)$
- Members of a category can be recognized by some properties.
  $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties.
  $Dogs \in DomesticatedSpecies$

OBJECTS

- Objects, also known as instances, are individual entities or things that belong to a particular category.
- They represent concrete items or elements within a domain.
- Each object is characterized by a set of attributes or properties that describe its features or characteristics.
- For example, in a real estate ontology, "House" and "Apartment" might be objects within the "Residential Properties" category, and each instance of a house or apartment would have attributes like "Number of Bedrooms," "Location," and "Price."

- Categories also serve to make predictions about objects once they are classified. There are two choices for representing categories in first-order logic: predicates and objects.

Physical composition

- We use the general PartOf relation to say that one thing is part of another.
- Objects can be grouped into PartOf hierarchies

PartOf (Bucharest , Romania)

PartOf (Romania, EasternEurope)

PartOf (EasternEurope, Europe)

PartOf (Europe, Earth) .

The PartOf relation is transitive and reflexive; that is, PartOf (x, y) ,PartOf (y, z) $\Rightarrow$ PartOf (x, z) .

Measurements

- In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called measures

Examples

- Diameter (Basketball 12)=Inches(9.5) .
- ListPrice(Basketball 12)=$(19) .

Events

- **Situation calculus** is limited in its applicability: it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time.
- **Event calculus**, which is based on points of time rather than on situations.
- Events are described as instances of event categories. The event E1 of Shankar flying from San Francisco to Washington, D.C. is described as

  $E1 \in Flyings \wedge Flyer(E1,Shankar) \wedge Origin(E1,SF) \wedge Destination(E1,DC)$.
- If this is too verbose, we can define an alternative three-argument version of the category of flying events and say

  $E1 \in Flyings(Shankar, SF, DC)$.

- We represent time intervals by a (start, end) pair of times; that is, i =(t1,t2) is the time interval that starts at t1 and ends at t2. The complete set of predicates for one version of the event calculus is

$T(f,t)$	Fluent f is true at time t
$Happens(e,i)$	Event e happens over the time interval i
$Initiates(e,f,t)$	Event e causes fluent f to start to hold at time t
$Terminates(e,f,t)$	Event e causes fluent f to cease to hold at time t
$Clipped(f,i)$	Fluent f ceases to be true at some point during time interval i
$Restored(f,i)$	Fluent f becomes true sometime during time interval i

Processes

- The events we have seen so far are **discrete events**—they have a definite structure. Shankar's trip has a beginning, middle, and end. If interrupted halfway, the event would be something different—it would not be a trip from San Francisco to Washington, but instead a trip from San Francisco to somewhere over Kansas.
- On the other hand, the category of events denoted by ***Flyings*** has a different quality. If we take a small interval of Shankar's flight, say, the third 20-minute segment (while he waits anxiously for a bag of peanuts), that event is still a member of ***Flyings.*** In fact, this is true for any subinterval. Categories of events with this property are called **process categories** or **liquid event categories.** Any process ***e*** that happens over an interval also happens over any subinterval:

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

- The distinction between **liquid and nonliquid events** is exactly analogous to the difference between substances, or stuff, and individual objects, or things. In fact, some have called liquid events temporal substances, whereas substances like butter are spatial substances

Fluents and objects

- Physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time.
- For example, USA can be thought of as an event that began in, say, 1776 as a union of 13 states and is still in progress today as a union of 50. We can describe the changing properties of USA using state fluents, such as Population(USA).A property of the USA that changes every four or eight years, barring mishaps, is its president. One might propose that President(USA) is a logical term that denotes a different object at different times. Unfortunately, this is not possible, because a term denotes exactly one object in a given model structure. (The term President(USA,t) can denote different objects, depending on the value of t, but our ontology keeps time indices separate from fluents.)

Introduction to Artificial Intelligence

UNIT-II

- Games – Optimal Decisions in Games,
- Alpha–Beta Pruning,
- Defining Constraint Satisfaction Problems,
- Backtracking Search for CSPs,
- Constraint Propagation,
- Knowledge-Based Agents
- Logic- Propositional Logic,
- Propositional Theorem Proving: Inference and proofs,
- Proof by resolution,
- Horn clauses and definite clauses.

Games – Optimal Decisions in Games

- We will consider the general case of a game with two players, whom we will call MAX and MIN.
- A game can be formally defined as a kind of search problem with the following components:
- The initial state, which includes the board position and an indication of whose move it is.
- A set of operators, which define the legal moves that a player can make

- A terminal test, which determines when the game is over. States where the game has ended are called terminal states.
- A utility function (also called a payoff function), which gives a numeric value for the outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent by the values +1, —1, or 0. Some games have a wider variety of possible outcomes; for example, the payoffs in backgammon range from +192 to —192.

The MINIMAX search procedure

- The minimax search is a depth first and depth limited procedure.
- This is two ply Game
  - Ex:- chess , checkers go , tic-tac-toe.

- The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions.
- Now we can apply the static evolution function to those positions and simply choose the best one.
- After doing so, we can back that value up to the starting position to represent our evolution of it.
- Here we assume that static evolution function returns larger values to indicate good situations for us.
- So our goal is to maximize the value of the static evaluation function of the next board position.
- The opponents' goal is to minimize the value of the static evaluation function.

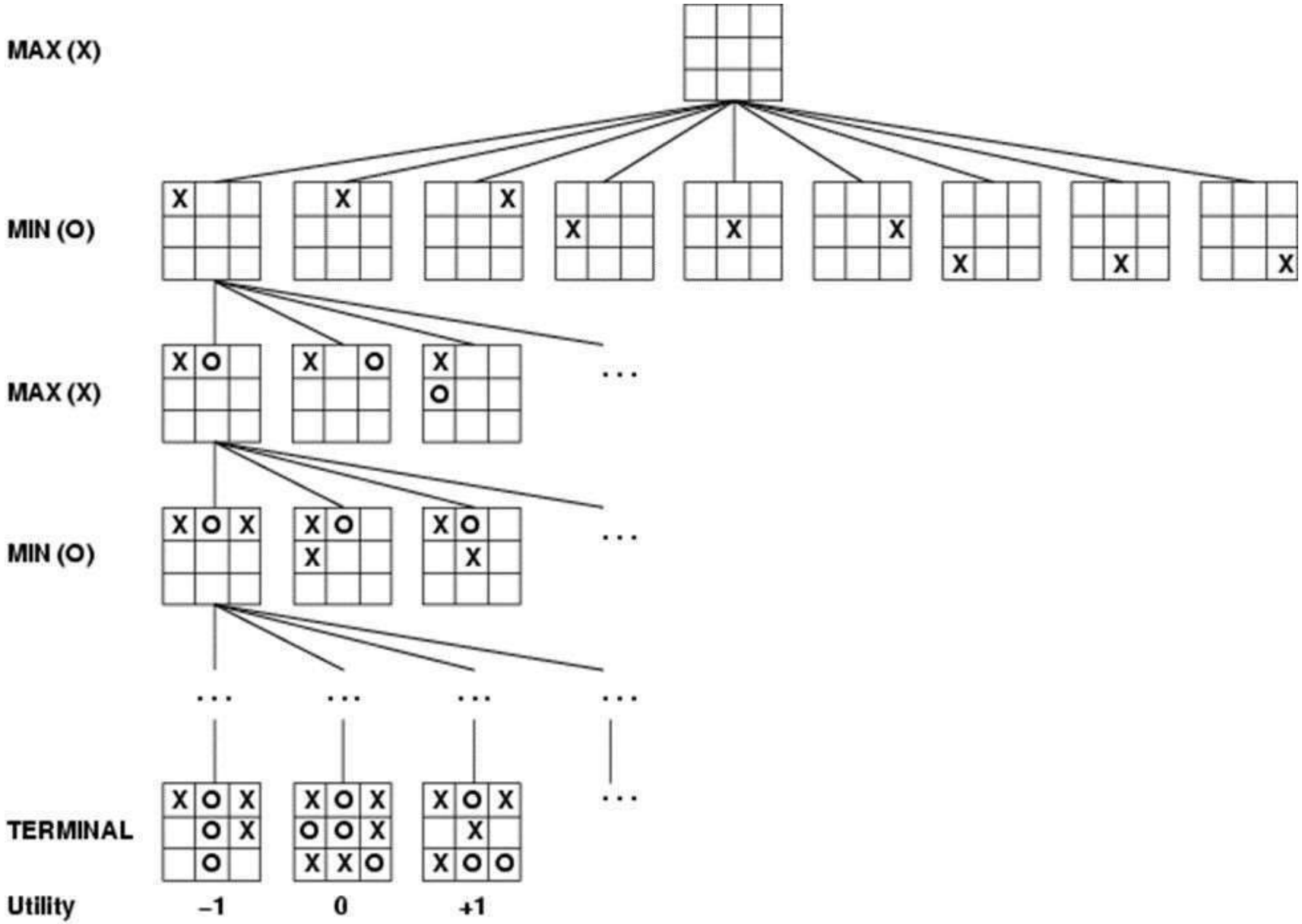
- *The alternation of maximizing and minimizing at alternate ply when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.*

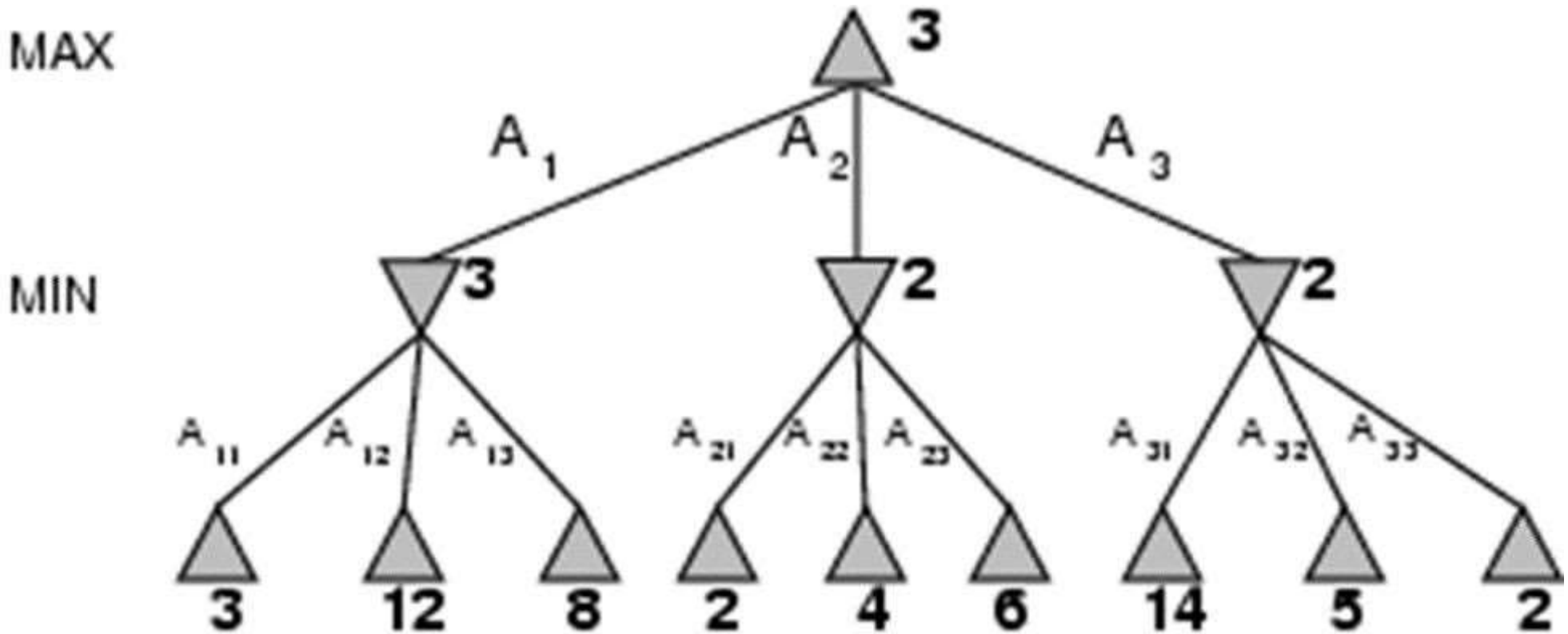
- S_0- Initial state
- Player(s)- who's the player in state 'S'
- Action (S)- Possible moves from state 'S'
- Result(S,a)- The state after action 'a' is taken on state 'S'
- Terminal(S)- Returns True if S is a Terminal state
- Utility(S,P)- The objective function in state 'S' for player 'P'

The **minimax algorithm** is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is. The algorithm consists of five steps:

- Generate the whole game tree, all the way down to the terminal states.
- Apply the utility function to each terminal state to get its value.
- Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree. Consider the leftmost three leaf nodes in Figure below In the V node above it, MIN has the option to move, and the best MIN can do is choose A11, which leads to the minimal outcome, 3. Thus, even though the utility function is not immediately applicable to this V node, we can assign it the utility value 3, under the assumption that MIN will do the right thing. By similar reasoning, the other two V nodes are assigned the utility value 2.

- Continue backing up the values from the leaf nodes toward the root, one layer at a time.
- Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value. In the topmost A node of Figure below, MAX has a choice of three moves that will lead to states with utility 3, 2, and 2, respectively. Thus, MAX's best opening move is A1. This is called the minimax decision, because it maximizes the utility under the assumption that the opponent will play perfectly to minimize it


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
–1
0
+1


MAX
MIN
3
A 1
A 2
A 3
3
2
2
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] — MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]

---------------------------------------------------------------

function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return  UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

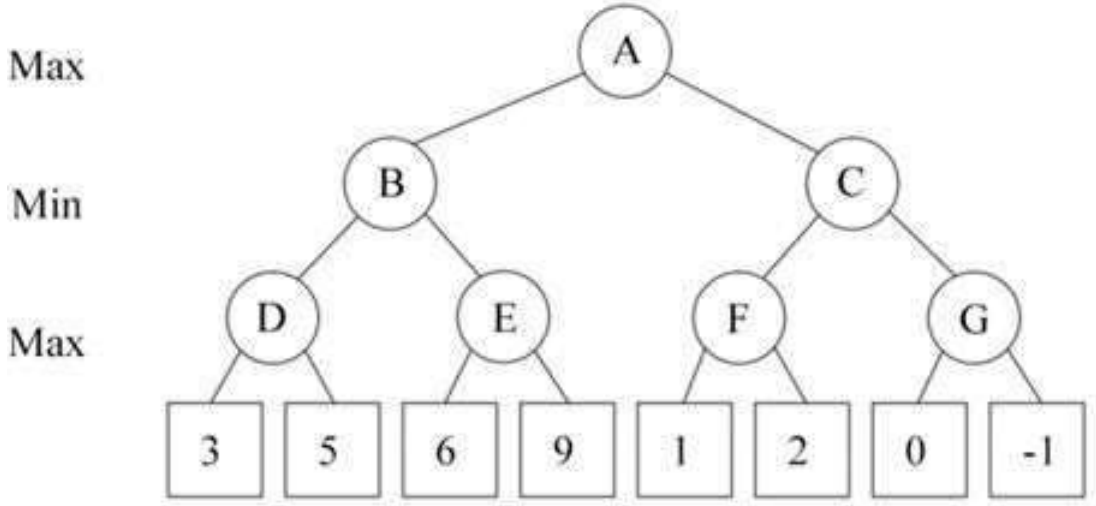
Figure 5.3 An algorithm for calculating minimax decisions. It returns the operator that corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The function MINIMAX-VALUE goes through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

Alpha-Beta Pruning

- Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor.
- This allows us to search much faster and even go into deeper levels in the game tree.
- It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.
- Let’s define the parameters alpha and beta.

- Alpha is the best value that the maximizer currently can guarantee at that level or above.
- Beta is the best value that the minimizer currently can guarantee at that level or above.

- Let’s make above algorithm clear with an example.



- The initial call stars from A. The value of alpha here is -INFINITY and the value of beta is +INFINITY. These values are passed down to subsequent nodes in the tree. At A the maximizer must choose max of B and C, so A calls B first
- At B it the minimizer must choose min of D and E and hence calls D first.
- At D, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at D is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- D now looks at its right child which returns a value of 5.At D, alpha = max(3, 5) which is 5. Now the value of node D is 5

- D returns a value of 5 to B. At B, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. B now calls E to see if he can get a lower value than 5.
- At E the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at B and that is what B passed down to E
- Now E looks at its left child which is 6. At E, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and E returns 6 to B
- Note how it did not matter what the value of E‘s right child is. It could have been +INF or -INF, it still wouldn’t matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of B. This way we dint have to look at that 9 and hence saved computation time.

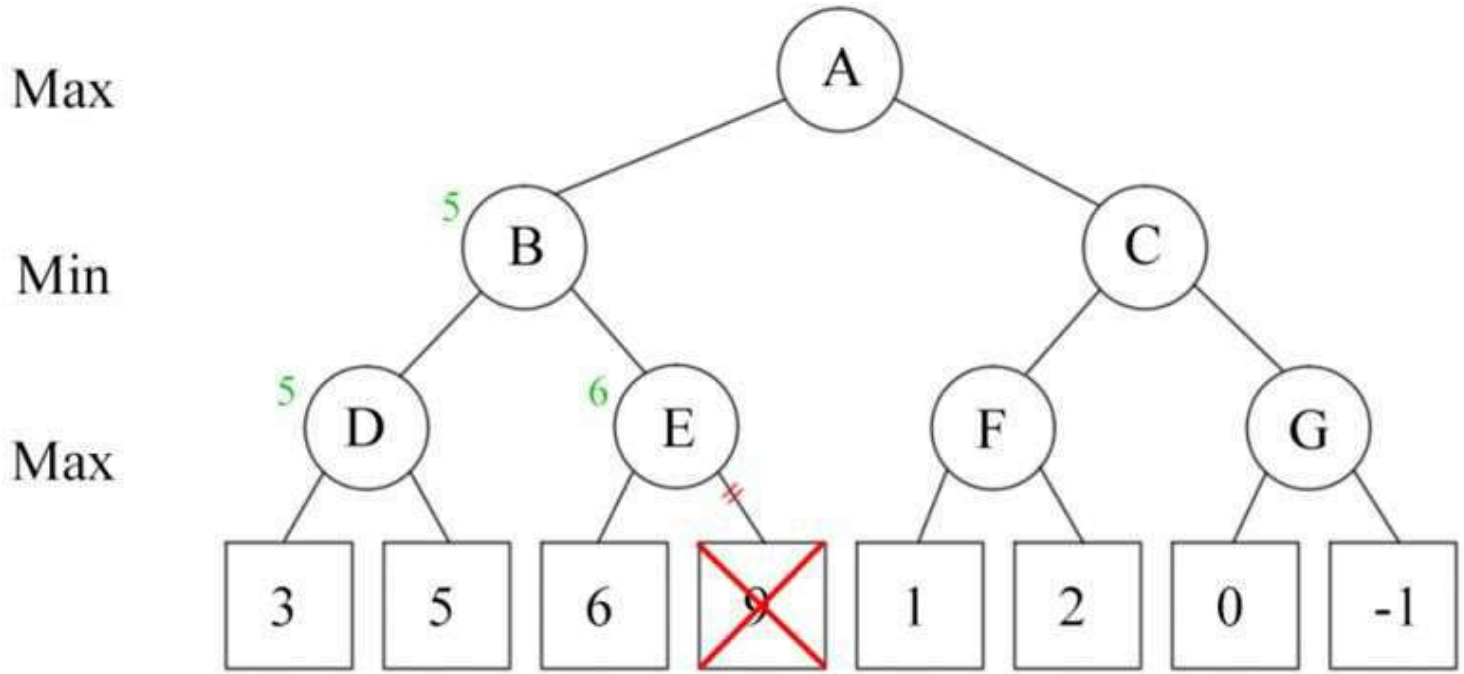
- E returns a value of 6 to B. At B, beta = min(5, 6) which is 5.The value of node B is also 5
- So far this is how our game tree looks. The 9 is crossed out because it was never computed.

- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.

- At **C**, alpha = 5 and beta = +INF. **C** calls **F**

- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.

- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5.


Max
Min
Max
A
5
B
C
5
D
6
E
F
G
3
5
6
9
1
2
0
-1

- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes false as beta = 2 and alpha = 5. So it breaks and it dose not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

Constraint Satisfaction Problems

- **Constraint Satisfaction Problem (CSP)** is a fundamental topic in artificial intelligence (AI) that deals with solving problems by identifying constraints and finding solutions that satisfy those constraints.
- The **goal** of AI is to create intelligent machines that can perform tasks that usually require human intelligence, such as reasoning, learning, and problem-solving. One of the key approaches in AI is the use of constraint satisfaction techniques to solve complex problems.
- **CSP** is a specific type of problem-solving approach that involves identifying constraints that must be satisfied and finding a solution that satisfies all the constraints. CSP has been used in a variety of **applications**, including scheduling, planning, resource allocation, and automated reasoning.

Constraint Satisfaction Problem (CSP)

- A **Constraint Satisfaction Problem** in artificial intelligence involves a set of variables, each of which has a domain of possible values, and a set of constraints that define the allowable combinations of values for the variables. The goal is to find a value for each variable such that all the constraints are satisfied.
- More formally, a CSP is defined as a triple (X,D,C), where:
- X is a set of variables { $x1$,$x2$, ..., xn}.
- D is a set of domains {$D1$, $D2$, ..., Dn}, where each Di is the set of possible values for xi.
- C is a set of constraints {$C1$, $C2$, ..., Cm}, where each Ci is a constraint that restricts the values that can be assigned to a subset of the variables.
- The goal of a CSP is to find an assignment of values to the variables that satisfies all the constraints. This assignment is called a solution to the CSP.

- A state of the problem is defined by an assignment of values to some or all of the variables, {Xi = vi,Xj =vj , ……} . An assignment that does not violate any constraints is called a **consistent or legal assignment.**
- A **complete assignment** is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an objective function.

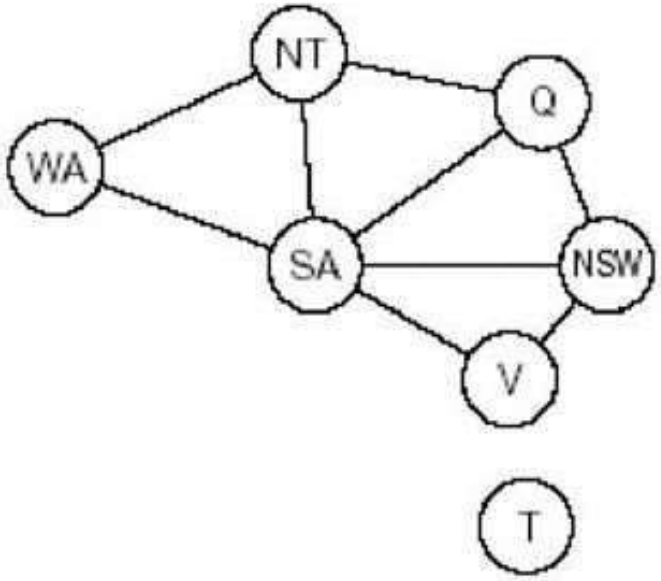
- **Examples of CSP are Map Coloring, Graph Coloring, Sudoku, Scheduling …**
- **Example:** The map coloring problem.
- We are given the task of coloring each region red, green, or blue in such a way that the neighboring regions must not have the same color.
- To formulate this as CSP, we define the variable to be the regions: WA, NT, Q, NSW, V, SA, and T.
- The domain of each variable is the set {red, green, blue}.
- The constraints require neighboring regions to have distinct colors: for example, the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.
- (The constraint can also be represented as the inequality WA ≠ NT). There are many possible solutions, such as {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}.

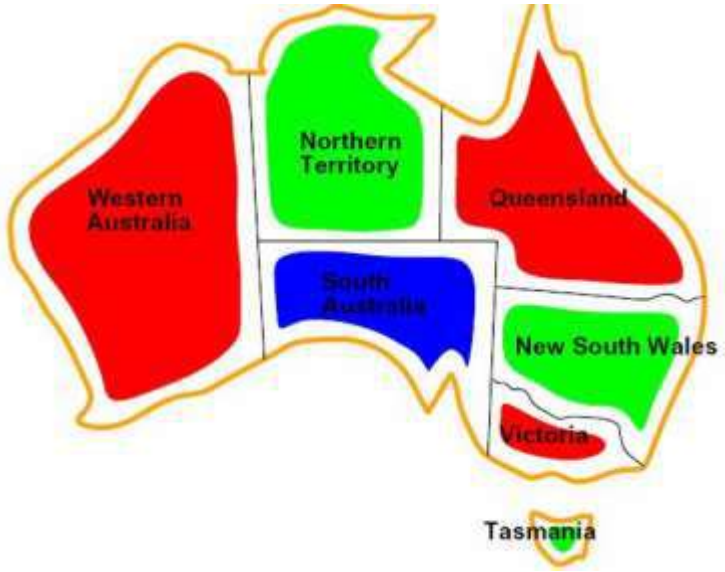
- Map of Australia showing each of its states and territories

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

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

Constraint graph: nodes are variables, arcs show constraints





CSP can be viewed as a standard search problem as follows:

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

- There are 3 methods to economically beneficial to something like a parameter:

1. Consistent or Legal Assignment: A task is referred to as consistent or legal if it complies with all laws and regulations.
2. Complete Assignment: An assignment in which each variable has a number associated to it and that the CSP solution is continuous. One such task is referred to as a completed task.
3. A partial assignment is one that just gives some of the variables values. Projects of this nature are referred to as incomplete assignment.

Domain Categories within CSP

- The domain of a variable in a Constraint satisfaction problem in artificial intelligence can be categorized into three types:

  finite, infinite, and continuous.

Finite domains have a finite number of possible values, such as colors or integers.

Infinite domains have an infinite number of possible values, such as real numbers.

Continuous domains have an infinite number of possible values, but they can be represented by a finite set of parameters, such as the coefficients of a polynomial function.

Types of Constraints in CSP

- Several types of constraints can be used in a Constraint satisfaction problem in artificial intelligence, including:
- **Unary Constraints:**
  A unary constraint is a constraint on a single variable. For example, Variable A not equal to “Red”.
- **Binary Constraints:**
  A binary constraint involves two variables and specifies a constraint on their values. For example, a constraint that two tasks cannot be scheduled at the same time would be a binary constraint.
- **Global Constraints:**
  Global constraints involve more than two variables and specify complex relationships between them. For example, a constraint that no two tasks can be scheduled at the same time if they require the same resource would be a global constraint.

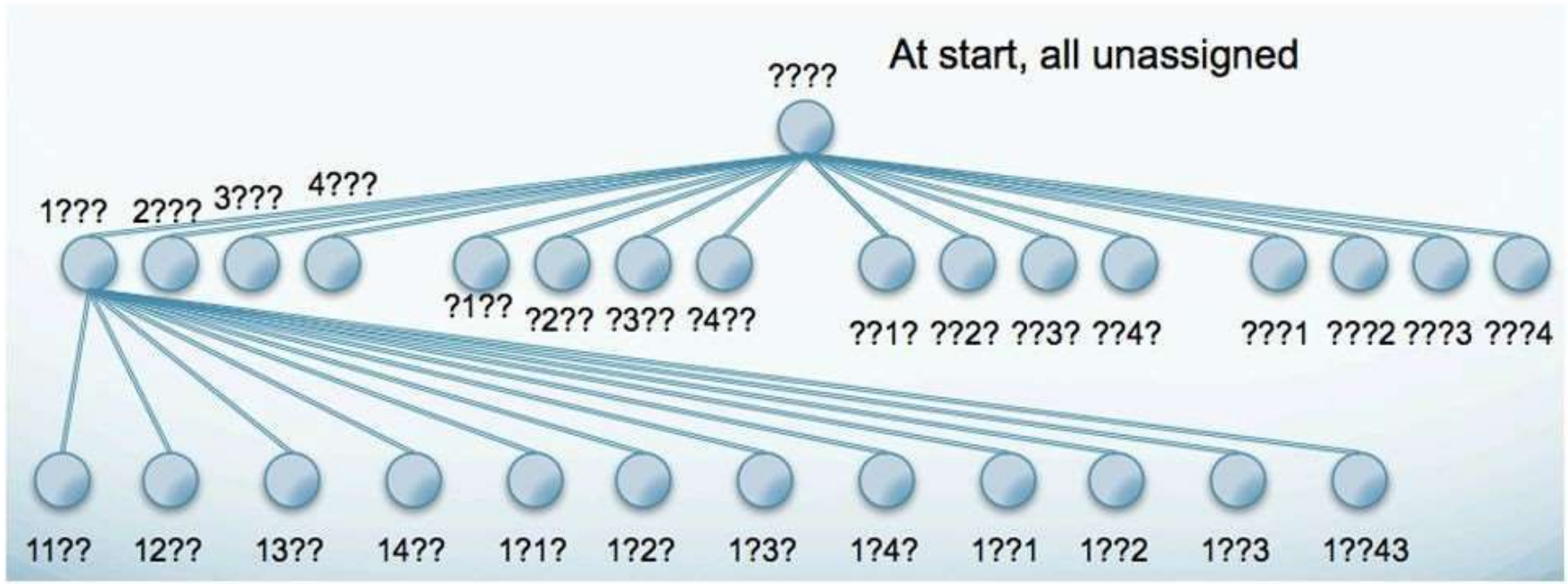
Backtracking Search for CSPs

- The branching factor at the top level is nd, because any of d values can be assigned to any of n variables. At the next level, the branching factor is (n−1)d, and so on for n levels. We generate a tree with $n!.d^n$ leaves, even though there are only d^n possible complete assignments!
- Commutativity: A problem is commutative if the order of application of any given set of actions has no effect on the outcome.
- CSPs are commutative: Regardless of the assignment order, the same partial solution is reached for a defined set of assignment values.

$n = 4$ variables each taking $d = 4$ values

$b = nd$

$b=(n-1)d$



Generate a search tree of $n!d^n$ but there are only d^n possible assignments!

- The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- we find general-purpose methods that address the following questions:

1. Which variable should be assigned next, and in what order should its values be tried?
2. What are the implications of the current variable assignments for the other unassigned variables?
3. When a path fails—that is, a state is reached in which a variable has no legal values— can the search avoid repeating this failure in subsequent paths?

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, csp)

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

Figure 5.3 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES can be used to implement the general-purpose heuristics discussed in the text.

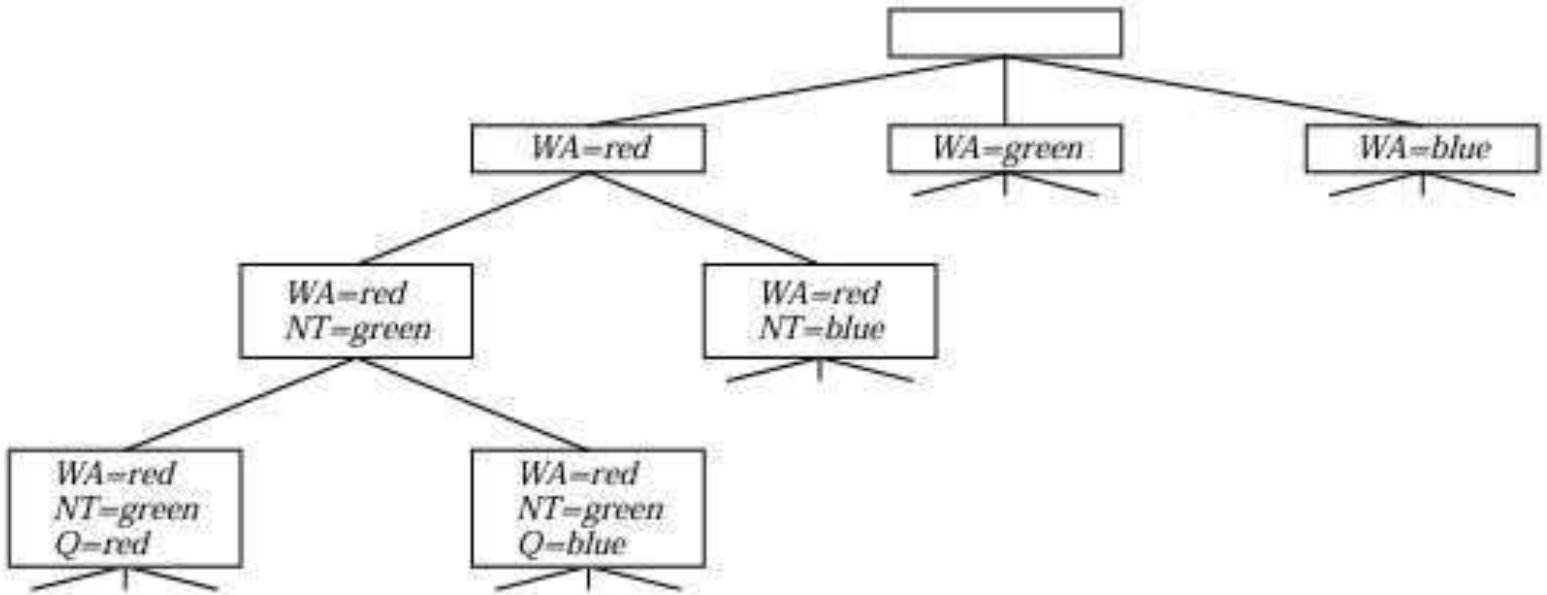


Figure 5.4 Part of the search tree generated by simple backtracking for the map-coloring problem in Figure 5.1.

Variable and Value ordering

- The backtracking algorithm contains the line
- var ← SELECT- UNASSIGNED VARIABLE(VARIABLES[csp],assignment,csp)
- By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES[csp].

Next variable?

- Random or static
- Variable with the fewest legal values: Minimum remaining values (MRV) heuristic (the most constrained var, the fail-first heuristic)
- Variable with the largest number of constraints on other unassigned variables, reduces b on future choices (Degree heuristic)

Variable's value?

- Value that leaves most choices for the neighboring variables in the constraint graph, max flexibility (least-constraining value heuristic), fail-last

Example

Variables: A,B,C,D Domain: {1,2,3}

Constraints: A ≠ B, A=D, C < B, C < D

Variable order: alphabetical **Value order: Descending**

- (A=3)
- (A=3, B=3) inconsistent with A ≠ B
- (A=3, B=2)
- (A=3, B=2, C=3) inconsistent with C < B
- (A=3, B=2, C=2) inconsistent with C < B
- (A=3, B=2, C=1)
- (A=3, B=2, C=1,D=3) VALID

Example

Variables: A,B,C,D Domain: {1,2,3}

Constraints: A ≠ B, A=D, C < B, C < D

Variable order: alphabetical **Value order:** Ascending

- (A=1)
- (A=1,B=1) inconsistent with A ≠ B
- (A=1,B=2)
- (A=1,B=2,C=1)
- (A=1,B=2,C=1,D=1) inconsistent with C < D
- (A=1,B=2,C=1,D=2) inconsistent with A=D
- (A=1,B=2,C=1,D=3) inconsistent with A=D

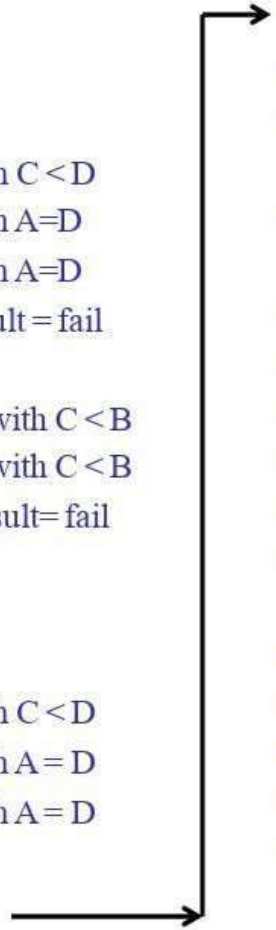
Example

Variables: A,B,C,D Domain: {1,2,3}

Constraints: $A \neq B$, $A=D$, $C < B$, $C < D$

Variable order: alphabetical **Value order:** Ascending

- (A=1)
- (A=1,B=1) inconsistent with $A \neq B$
- (A=1,B=2)
- (A=1,B=2,C=1)
- (A=1,B=2,C=1,D=1) inconsistent with $C < D$
- (A=1,B=2,C=1,D=2) inconsistent with $A=D$
- (A=1,B=2,C=1,D=3) inconsistent with $A=D$
- No valid assignment for D, return result = fail
  - Backtrack to (A=1,B=2,C=)
- Try (A=1,B=2,C=2) but inconsistent with $C < B$
- Try (A=1,B=2,C=3) but inconsistent with $C < B$
- No other assignments for C, return result= fail
  - Backtrack to (A=1,B=)
- (A=1,B=3)
- (A=1,B=3,C=1)
- (A=1,B=3,C=1,D=1) inconsistent with $C < D$
- (A=1,B=3,C=1,D=2) inconsistent with $A = D$
- (A=1,B=3,C=1,D=3) inconsistent with $A = D$
- Return result = fail
  - Backtrack to (A=1,B=3,C=)



- (A=1,B=3,C=2) inconsistent with $C < B$
- (A=1,B=3,C=3) inconsistent with $C < B$
- No remaining assignments for C, return fail
  - Backtrack to (A=1,B=)
- No remaining assignments for B, return fail
  - Backtrack to A
- (A=2)
- (A=2,B=1)
- (A=2,B=1,C=1) inconsistent with $C < B$
- (A=2,B=1,C=2) inconsistent with $C < B$
- (A=2,B=1,C=3) inconsistent with $C < B$
- No remaining assignments for C, return fail
  - Backtrack to (A=2,B=?)
- (A=2,B=2) inconsistent with $A \neq B$
- (A=2,B=3)
- (A=2,B=3,C=1)
- (A=2,B=3,C=1,D=1) inconsistent with $C < D$
- (A=2,B=3,C=1,D=2) **ALL VALID**

Propagating Information through constraints

Forward Checking:

One way to make better use of constraints during search is called forward checking. When ever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is in consistent with the value chosen for X.

Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q=*green*	(R)	B	(G)	R B	R G B	B	R G B
After V=*blue*	(R)	B	(G)	R	(B)		R G B

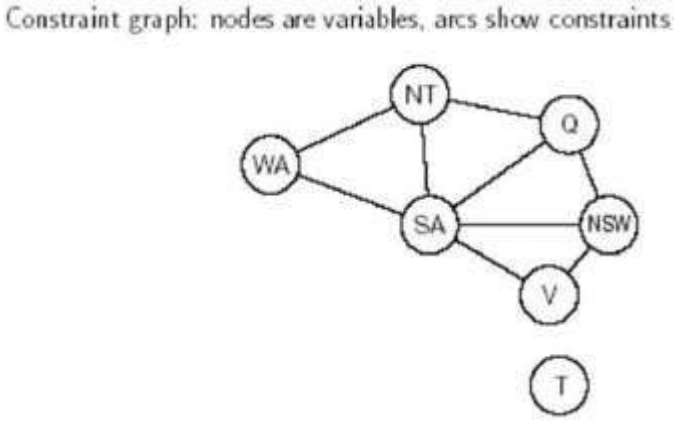
Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

- after V =blue, the domain of SA is empty. Hence, forward checking has detected that the partial assignment {WA=red,Q=green,V =blue} is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

Constraint Propagation:

Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables;

The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking. Here, “arc” refers to a directed arc in the constraint graph, such as the arc from SA to NSW.



```
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff we remove a value
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint between X_i and X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

Figure 5.7 The arc consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be made arc-consistent (and thus the CSP cannot be solved). The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

AC-Algorithm

1. Turn each binary constraint into two arcs.
2. Add all arcs to agenda.
3. Repeat until agenda is empty.
    i) Take an arc(x_i, x_j)off the agenda and check it.
    ii) For every value of x_i, there must be some value of x_j.
    iii) Remove any inconsistent values from x_i.
    iv) If x_i has changed , add all arcs of the form (x_k, x_i) to the agenda.

- Applying arc consistency has resulted in early detection of an inconsistency that is not detected by pure forward checking.
- Example: A={1,2,3}

B={1,2,3}

C={1,2,3}

A>B

B=C

Arcs

A>B

B<A

B=C

C=B

Knowledge based Agents

- An intelligent agent needs knowledge about the real world for taking decisions and reasoning to act efficiently.
- Knowledge-based agents are those agents who have the capability of maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently.
- Knowledge-based agents are composed of two main parts:

  Knowledge-base and Inference system.
- A knowledge-based agent must able to do the following:

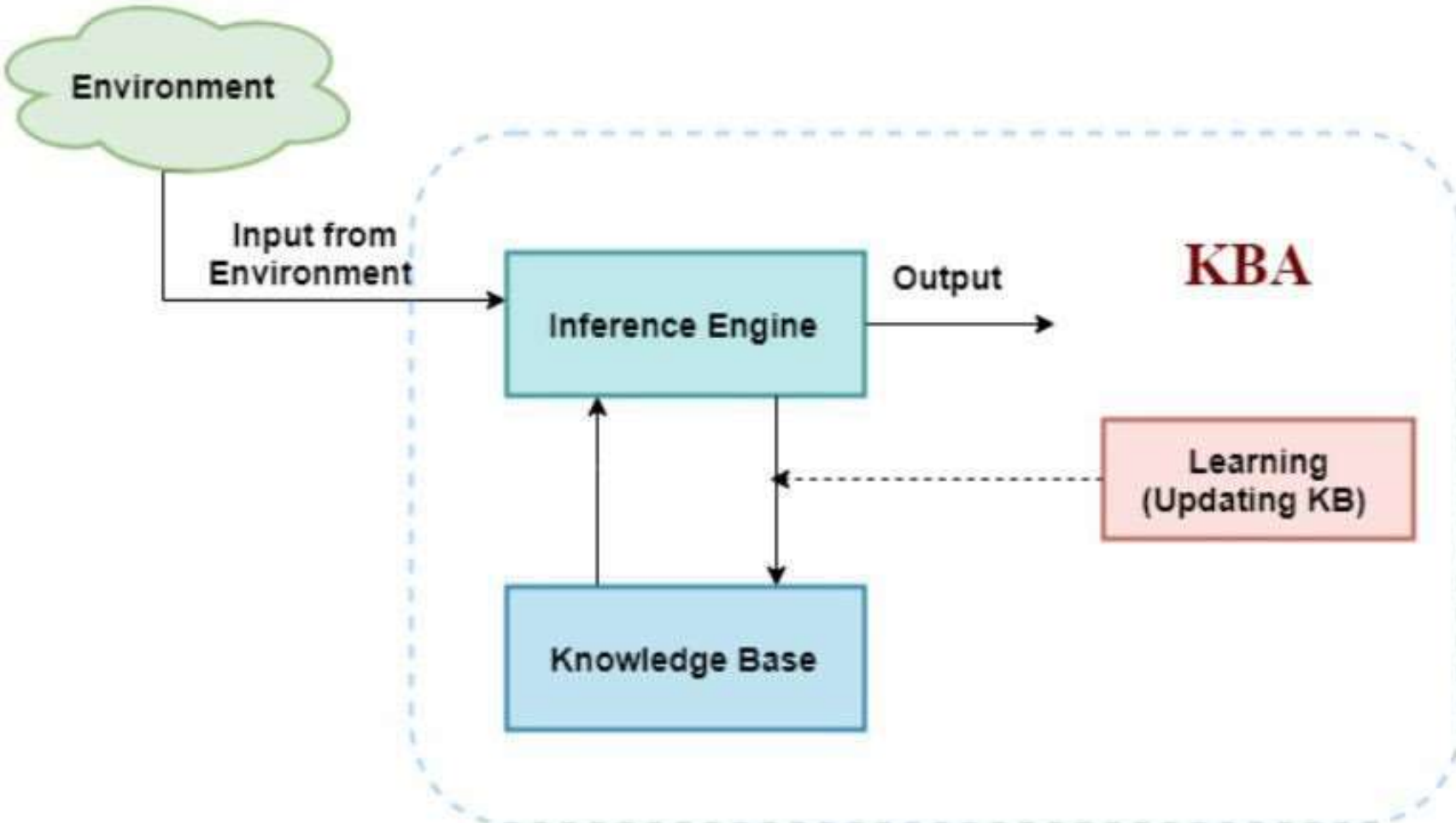
  An agent should be able to represent states, actions, etc.

  An agent should be able to incorporate new percepts

  An agent can update the internal representation of the world

  An agent can deduce the internal representation of the world

  An agent can deduce appropriate actions.


Environment
Input from Environment
Inference Engine
Output
KBA
Learning (Updating KB)
Knowledge Base

- **Knowledge base:** Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.
- **Inference** means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.
- Operations Performed by KBA

Following are three operations which are performed by KBA in order to show the intelligent behavior:

1. **TELL:** This operation tells the knowledge base what it perceives from the environment.
2. **ASK:** This operation asks the knowledge base what action it should perform.
3. **Perform:** It performs the selected action.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time.

MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time.

Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed.

- **Various levels of knowledge-based agent:**

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level.

Approaches to designing a knowledge-based agent:

- There are mainly two approaches to build a knowledge-based agent:

1. Declarative approach: We can create a knowledge-based agent by initializing with an empty knowledge base and telling the agent all the sentences with which we want to start with. This approach is called Declarative approach.

2. Procedural approach: In the procedural approach, we directly encode desired behavior as a program code. Which means we just need to write a program that already encodes the desired behavior or agent.

Logic- Propositional Logic

- Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.
- **Following are some basic facts about propositional logic:**
- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.

- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called tautology, and it is also called a valid sentence.
- A proposition formula which is always false is called Contradiction.
- Statements which are questions, commands, or opinions are not propositions such as "Where is Rohini", "How are you", "What is your name", are not propositions.

- **Syntax of propositional logic:**

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Sentences:

- **Atomic Sentences:** Atomic sentences are the simple sentences. It consists of a single proposition symbol. These are the sentences which must be either true or false.
- **Compound Sentences :** Compound **Sentences** are constructed by combining simpler or atomic **Sentences**, using parenthesis and logical connectives.

- **Semantics of propositional logic:**

The semantics defines the rules for determining the truth of a sentence with respect to a particular model.

- **Logical Connectives:**

Logical connectives are used to connect two simpler **Sentences** or representing a sentence logically. We can create compound **Sentences** with the help of logical connectives. There are mainly five connectives

- **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
- **Conjunction:** A sentence which has ⋀ connective such as, P ⋀ Q is called a conjunction.

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

  P= Rohan is intelligent,

  Q= Rohan is hardworking. → P⋀ Q.
- **Disjunction:** A sentence which has ⋁ connective, such as P ⋁ Q. is called disjunction, where P and Q are the propositions.

  Example: "Ritika is a doctor or Engineer",

  Here P= Ritika is Doctor. Q= Ritika is Engineer, so we can write it as P ⋁ Q.
- **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as

  If it is raining, then the street is wet.

  Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
- **Biconditional:** A sentence such as P⇔ Q is a Biconditional sentence,

  example If I am breathing, then I am alive

  P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

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*

Precedence of connectives:

Sentence → *AtomicSentence* | *ComplexSentence*

AtomicSentence → *True* | *False* | *P* | *Q* | *R* | ...

ComplexSentence → (*Sentence*) | [*Sentence*]
| ¬ *Sentence*
| *Sentence* ∧ *Sentence*
| *Sentence* ∨ *Sentence*
| *Sentence* ⇒ *Sentence*
| *Sentence* ⇔ *Sentence*

OPERATOR PRECEDENCE : ¬, ∧, ∨, ⇒, ⇔

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

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

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

- **Limitations of Propositional logic:**
- We cannot represent relations like ALL, some, or none with propositional logic. Example:
  - **All the girls are intelligent.**
  - **Some apples are sweet.**
- Propositional logic has limited expressive power.

Propositional Theorem Proving

- logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$.
- Validity: A sentence is valid if it is true in all models. Valid sentences are also known as tautologies.
- deduction theorem:
  - For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.
- Satisfiability: A sentence is satisfiable if it is true in, or satisfied by, some model.
- The problem of determining the satisfiability of sentences in propositional logic—the SAT problem

Deduction Theorem

Deduction Theorem
For any sentences α and β, α |= β if and only if the sentence (α ⇒ β) is valid

α |= β means that if α is true, then β must also be true. If α is false, then β may be true or false

We also know that the only way (α ⇒ β) will be false is when α is true and β is false

But α = true and β = false cannot happen as α |= β

As the above cannot happen, (α ⇒ β) will be true in all models i.e. (α ⇒ β) is a valid sentence

Hence, we can decide if α |= β by checking that (α ⇒ β) is true in every model

Satisfiable

A sentence is **satisfiable** if it is true in, or satisfied by, some model

Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence

α is valid iff $\neg\alpha$ is unsatisfiable

α is satisfiable iff $\neg\alpha$ is not valid

$\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid

$\alpha \models \beta$ if and only if the sentence $(\neg\alpha \vee \beta)$ is valid

$\alpha \models \beta$ if and only if the sentence $\neg(\neg\alpha \vee \beta)$ is unsatisfiable

$\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ Is called proof by **refutation** or proof by **contradiction**

- proof by refutation or proof by contradiction
- 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.

  Proving β from α by checking the unsatisfiability of (α ⋀¬β) corresponds exactly to the standard mathematical proof technique .It is also called proof by refutation or proof by contradiction.

Inference and proofs

- In **propositional theorem proving**, we use **inference rules** to create a **proof**—a series of conclusions that lead to the desired result.
- One well-known inference rule is **Modus Ponens**, which allows us to affirm a conclus $\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$ en premises.
- Another useful rule is **And-Elimination**, which states that any of the conjuncts can be inferre $\frac{\alpha \wedge \beta}{\alpha}$ a conjunction.
- These rules are sound, meaning they lead to valid conclusions without enumerating all possible models.

- 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.

One final property of logical systems is monotonicity, which says that the set of entailed sentences can only increase as information is added to the knowledge base.

For any sentences α and β, if KB $\models \alpha$ then KB $\wedge \beta \models \alpha$.

Sample Rules of Inference

- Modus Ponens: $\{\alpha \Rightarrow \beta, \alpha\} \vdash \beta$
- And Elimination: $\{\alpha \wedge \beta\} \vdash \alpha$; $\{\alpha \wedge \beta\} \vdash \beta$
- And Introduction: $\{\alpha, \beta\} \vdash \alpha \wedge \beta$
- Or introduction: $\{\alpha\} \vdash \alpha \vee \beta$
- Double negation Elimination: $\{\neg\neg\alpha\} \vdash \alpha$
- Implication Elimination: $\{\alpha \Rightarrow \beta\} \vdash \neg\alpha \vee \beta$
- Unit resolution: $\{\alpha \vee \beta, \neg\beta\} \vdash \alpha$
- Resolution: $\{\alpha \vee \beta, \neg\beta \vee \gamma\} \vdash \alpha \vee \gamma$

Sample Proof

If John is not married he is a bachelor. $(\neg P \Rightarrow Q)$
John is not a bachelor. $(\neg Q)$
Therefore, he is married. (P)

$\neg P \Rightarrow Q$	
$\neg\neg P \vee Q$	Implication elimination
$P \vee Q, \quad \neg Q$	Double negation elimination
P	Unit resolution

Could also check validity of:

$((\neg P \Rightarrow Q) \wedge \neg Q) \Rightarrow P$

Form of modus tollens reasoning

Proof by resolution

- **Resolution:** An inference rule that yields a complete inference algorithm when coupled with any complete search algorithm.
- **Clause:** A disjunction of literals. (e.g. $A \vee B$). A single literal can be viewed as a **unit clause** (a disjunction of one literal).
- **Unit resolution inference rule:** Takes a clause and a literal and produces a new clause.

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

- where each l is a literal, l_i and m are **complementary literals** (one is the negation of the other).

- **Full resolution rule:** Takes 2 clauses and produces a new clause.

$$\frac{l_1 \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_n}{l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee vm_n}$$

- where l_i and m_j are complementary literals.
- Notice: The resulting clause should contain only one copy of each literal. The removal of multiple copies of literal is called **factoring**.
- The resolution rule is sound and complete.

Horn clauses and definite clauses.

- **Definite clause:** A disjunction of literals of which exactly one is positive. (e.g. $\neg L_{1,1} \vee \neg Breeze \vee B_{1,1}$)
- 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.
- **Horn clause:** A disjunction of literals of which at most one is positive. (All definite clauses are Horn clauses.)
- In Horn form, the premise is called the **body** and the conclusion is called the **head**.
- A sentence consisting of a single positive literal is called a **fact**, it too can be written in implication form.
- Horn clause are closed under resolution: if you resolve 2 horn clauses, you get back a horn clause.
- Inference with horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms.
- Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.
- **Goal clause:** A clause with no positive literals.

$$
\begin{aligned}
\mathit{CNFSentence} &\rightarrow \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} &\rightarrow \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} &\rightarrow \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} &\rightarrow P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} &\rightarrow \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{aligned}
$$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the k-CNF sentence, which is a CNF sentence where each clause has at most k literals.

Introduction to Artificial Intelligence

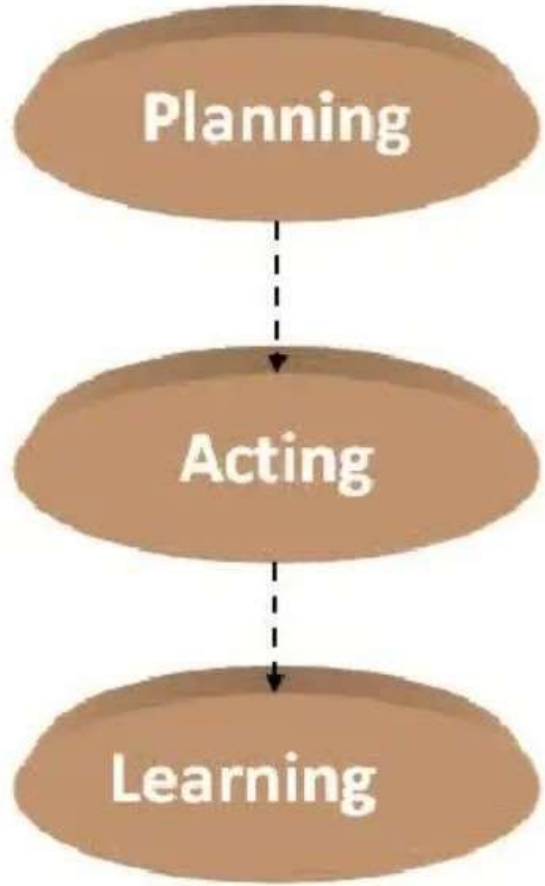
UNIT-IV

- Planning - Definition of Classical Planning
- Algorithms for Planning with State Space Search
- Planning Graphs
- Other Classical Planning Approaches
- Analysis of Planning approaches
- Hierarchical Planning.

Definition of Classical Planning

Classical Planning is the planning where an agent takes advantage of the problem structure to construct complex plans of an action. The agent performs three tasks in classical planning:

- **Planning:** The agent plans after knowing what is the problem.
- **Acting:** It decides what action it has to take.
- **Learning:** The actions taken by the agent make him learn new things.


Planning
Acting
Learning

A language known as **PDDL(Planning Domain Definition Language)** which is used to represent all actions into one action schema.

PDDL describes the four basic things needed in a search problem:

- **Initial state:** It is the representation of each state as the conjunction of the ground and functionless atoms.
- **Actions:** It is defined by a set of action schemas which implicitly define the **ACTION()** and **RESULT()** functions.
- **Result:** It is obtained by the set of actions used by the agent.

- **Goal Test:** The goal is just like a precondition: a conjunction of literals (positive or negative).
- **Example: Air Cargo Transport**

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

Example: Air Cargo Transport

- Problem involves loading and unloading cargo and flying it from place to another place
- Three actions:
  - ***Load:*** *This action is taken to load cargo*
  - ***Unload:*** *This action is taken to unload the cargo, when it reaches its destination*
  - ***Fly:*** *This action is taken to fly from one place to another*

Therefore air cargo transport problem based on loading and unloading the cargo and flying it from one place to another.

- PDDL Description for air cargo transport
- Actions affect two predicates:
  - ***In(c,p):*** cargo c is in plane p
  - ***At(x,a):*** object x (plane or cargo) is at airport a
  - When a plane flies from one airport to another all cargo inside moves with it.
  - In FOL we could use quantifiers (i.e. forall), but PDDL doesn't have (forall)
  - The approach we use is to say that a piece of cargo ceases to be ***At*** anywhere when it is ***In*** a plane; the cargo only becomes ***At*** the new airport when it is unloaded.

- A plan solution to the problem
  - [Load (C1,P1,SFO),Fly(P1,SFO,JFK),Unload(C1,P1,JFK), Load(C2,P2,JFK),Fly(P2,JFK,SFO),Unload(C2,P2,SFO)] .
- Finally, there is the problem of spurious actions such as Fly(P1,JFK,JFK), which should be a no-op, but which has contradictory effects (according to the definition, the effect would include At(P1,JFK)∧¬At(P1,JFK)).
- The correct approach is to add inequality preconditions saying that the **from** and **to** airports must be different;

- **Example: The Spare Tire Problem**
- Problem of changing a flat tire
  - **Goal:** properly get and mount the spare tire onto the car's axle
  - **Initial state:** flat tire on the axle and a good spare tire in the trunk
  - **Actions:** There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. A solution to the problem is [Remove(Flat,Axle),Remove(Spare,Trunk),PutOn(Spare,Axle)].
  - **Assumptions:** We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.
  - A **solution** to the problem is

[Remove(Flat,Axle),Remove(Spare,Trunk),PutOn(Spare,Axle)].

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
PRECOND: $At(obj, loc)$
EFFECT: $\neg At(obj, loc) \land At(obj, Ground))$
$Action(PutOn(t, Axle),$
PRECOND: $Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)$
EFFECT: $\neg At(t, Ground) \land At(t, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
$\land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))$

Figure 10.2 The simple spare tire problem.

Example: The Blocks World

- Famous planning domain
- A set of cube-shaped blocks sitting on a table
- Blocks can be stacked, only one block can fit directly on top of another
- Robot arm picks up a block and moves it to another position
  - On the table or on top of another block
  - Arm can pick only one object at a time (can’t pick an object that has another one on it)

- Goal: build one or more stacks of blocks, specifying which blocks are on top of what other blocks i.e. Get block A on B and B on C.
- On(b,x) means that block b is on x, where x can be another block or the table
- Move(b,x,y) is the action of moving block b from the top of x to the top of y if b and y are clear.
  - After the move is made, b is still clear but y is not. A first attempt at the Move schema is

    Action(Move(b,x,y),

    PRECOND:On(b,x) ∧Clear(b) ∧ Clear(y), EFFECT:On(b,y) ∧ Clear(x) ∧¬On(b,x) ∧¬Clear(y)) .
  - Doesn't maintain the *Clear* property when x or y is the table
    - The table should not become clear: Clear(table)
  - To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

    Action(MoveToTable(b,x), PRECOND:On(b,x) ∧Clear(b), EFFECT:On(b,Table) ∧ Clear(x) ∧¬On(b,x)) .

$Init(On(A, Table) \land On(B, Table) \land On(C, A)$
$\quad \land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$
$(b \neq x) \land (b \neq y) \land (x \neq y),$
EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),$
EFFECT: $On(b, Table) \land Clear(x) \land \neg On(b, x))$

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

Complexity of Classical Planning

- **PlanSAT**: Question of whether there exists any plan that solves a planning problem.
- **Bounded PlanSAT**: Asks whether there is a solution of length ***k*** or less, can be used to find an optimal plan
- Both decision problems are decidable for classical planning
  - If we add function symbols to the language
    - Number of states becomes infinite
    - PlanSAT becomes semidecidable
      - An algorithm that will terminate with the correct answer for any solvable problem ***exists***, but may not terminate on unsolvable problems
    - Bounded PlanSAT is still decidable when we add function symbols
- Complexity of PlanSAT and Bounded PlanSAT is in class PSPACE, larger and more difficult than NP

Planning with State-Space Search

- The most straightforward approach of planning algorithm, is state-space search
  - Forward state-space search (Progression)
  - Backward state-space search (Regression)
- The descriptions of actions in a planning problem, and specify both preconditions and effects
- It is possible to search in both direction: either forward from the initial state or backward from the goal
- We can also use the explicit action and goal representations, to derive effective heuristics automatically.

Forward State Space Search(Progression)

Problem Formulation for Progression

- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Progression

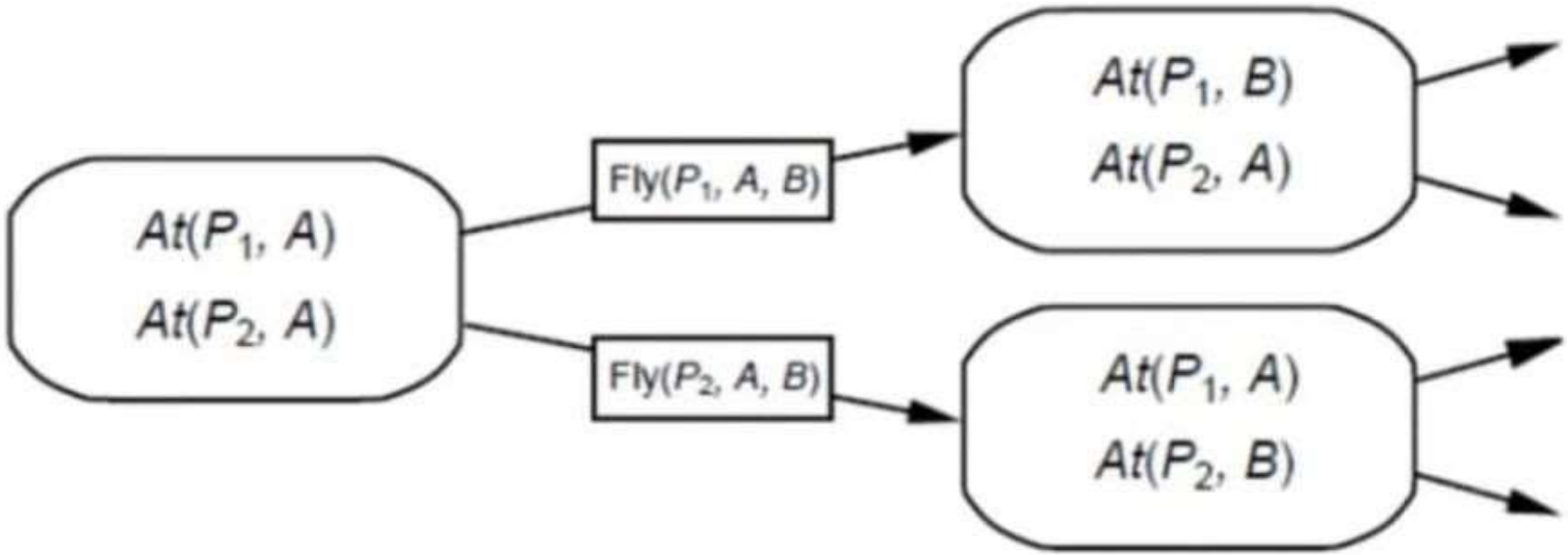
- From initial state, search forward by selecting operators whose preconditions can be unified with literals in the state
- New state includes positive literals of effect; the negated literals of effect are deleted
- Search forward until goal unifies with resulting state
- This is forward state-space search using STRIPS operators

Example : Transportation of air cargo between airports.

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
EFFECT: $\neg At(p, from) \land At(p, to))$

Forward (progression) state-space search

- Starting from the initial state and using the problem's actions to search forward for the goal state.



FSSS Algorithm

- (1) compute whether or not a state is a goal state,
- (2) find the set of all actions that are applicable to a state, and
- (3) compute a successor state, that is the result of applying an action to a state
- Algorithm takes as input the statement P = (O, s0, g) of a planning problem P. (O contains a list of actions)
- If P is solvable, then Forward-search(O, s0, g) returns a solution plan; otherwise it returns failure.

Algorithm for FSSS

1. Forward-search(O, s0, g)
2. $s \leftarrow s0$
3. $\pi \leftarrow$ the empty plan
4. loop
    1. if s satisfies g then return π
    2. applicable $\leftarrow$ {a | a is a ground instance of an operator in O, and precond(a) is true in s}
    3. if applicable = $\emptyset$ then return failure
    4. Non-deterministically choose an action a $\in$ applicable
    5. $s \leftarrow \gamma(s, a)$
    6. $\pi \leftarrow \pi.a$

Backward State Space Search(Regression)

Problem Formulation for Regression

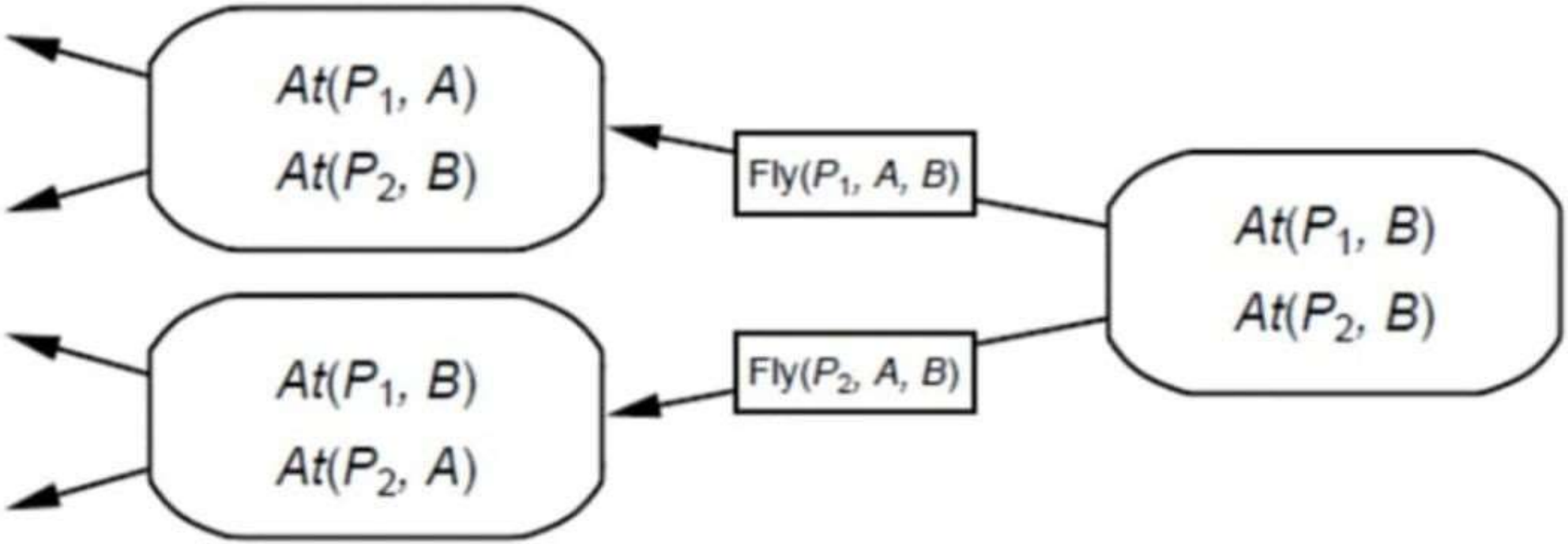
- Initial state:
  - Initial state of the planning problem
- Actions:
  - Applicable to the current state.
  - First actions' preconditions are satisfied, Successor states are generated
  - Add positive literals to add list and negative literals to delete list.
- Goal test:
  - Whether the state satisfies the goal of the planning
- Step cost:
  - Each action is 1 (assumed)

Regression

- The goal state must unify with at least one of the positive literals in the operator's effect
- Its preconditions must hold in the previous situation, and these become subgoals which might be satisfied by the initial conditions
- Perform backward chaining from goal
- Again, this is just state-space search using STRIPS operators

Backward (regression) state-space search:

- Backward state search starting from the goal state(s)
- Using the inverse of the actions to search backward for the initial state.



BSSS Algorithm

- Start at the goal,
- Test the goal is initial state, otherwise
- Apply inverses of the planning operators to produce subgoals,
- The algorithm will stop, if we produce a set of subgoals that satisfies the initial state.

BSSS Algorithm

1. Backward-search(*O*, *s*0, *g*)
2. π the empty plan
3. loop
    1. if *s*0 satisfies *g* then return π
    2. *applicable* $\leftarrow$ {*a* | *a* is a ground instance of an operator in *O* that is relevant for *g*}
    3. if *applicable* = $\emptyset$ then return failure
    4. Non-deterministically choose an action $a \in applicable$
    5. $\pi \leftarrow a.\ \pi$
    6. $g \leftarrow \gamma^{-1}(g, a)$

Heuristics for State-Space Search

- How to find an admissible heuristic estimate?
  - Distance from a state to the goal?
  - Look at the effects of the actions and at the goals and guess how many actions are needed
- NP-hard
- Relaxed problem
- Subgoal independence assumption:
- The cost of solving a conjunction of subgoals, is approximated by the sum of the costs of solving each subgoal independently

PLANNING GRAPHS

- Planning Graph can give better heuristic estimates.
- Here we can extract a solution directly from the planning graph, using a specialized algorithm such as GRAPHPLAN
- A planning graph consists of a sequence of **levels** that correspond to time steps in the plan, where level 0 is the initial state.
- Each level contains a set of literals and a set of actions.
- The literals are true at that time step, depending on, the actions executed at preceding time steps.
- Actions *could* have their preconditions, that should be satisfied at that time step, depending on the literals actually hold.

- Planning graphs are an efficient way to create a representation of a planning problem, that can be used to
  - Achieve better heuristic estimates
  - Directly construct plans
- Planning graphs only work for propositional problems.

Example - The “have cake and eat cake too” problem.

$$Init(Have(Cake))$$
$$Goal(Have(Cake) \land Eaten(Cake))$$
$$Action(Eat(Cake)$$
$$\text{PRECOND: } Have(Cake)$$
$$\text{EFFECT: } \neg Have(Cake) \land Eaten(Cake))$$
$$Action(Bake(Cake)$$
$$\text{PRECOND: } \neg Have(Cake)$$
$$\text{EFFECT: } Have(Cake)$$

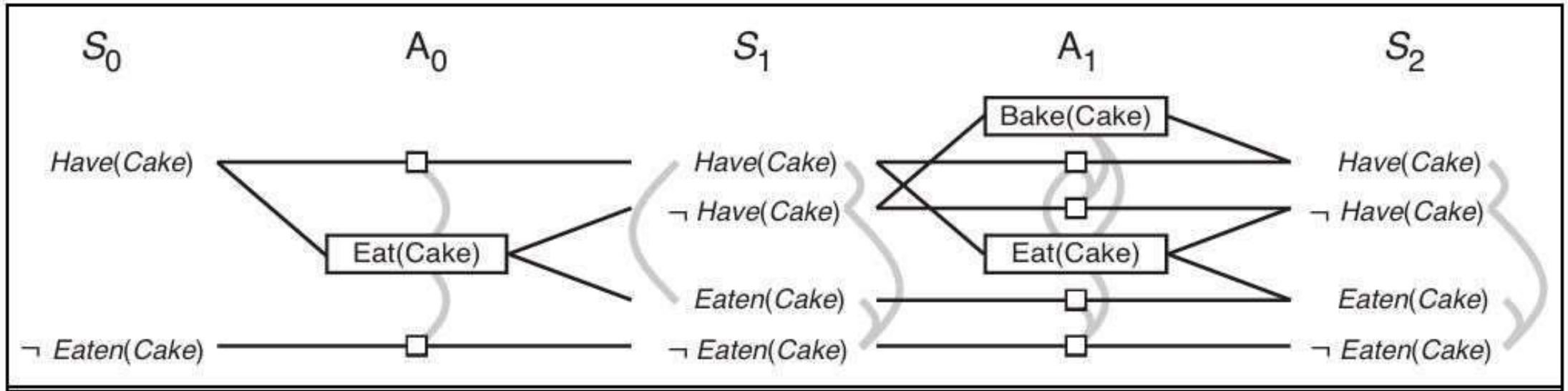


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

- Repeat process until graph levels off:
  - two consecutive levels are identical, or
  - contain the same amount of literals

Mutual exclusion

- A mutex relation holds between two actions when:
  - Inconsistent effects: one action negates the effect of another.
  - Interference: one of the effects of one action, is the negation of a precondition of the other.
  - Competing needs: one of the preconditions of one action, is mutually exclusive with the precondition of the other.
- A mutex relation holds between two literals when:
  - one is the negation of the other
  - each possible action pair that could achieve the literals is mutex (inconsistent support).

THE GRAPHPLAN ALGORITHM

```
function GRAPHPLAN(problem) returns solution or failure
    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← CONJUNCTS(problem.GOAL)
    nogoods ← an empty hash table
    for tl = 0 to ∞ do
        if goals all non-mutex in S_t of graph then
            solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
            if solution ≠ failure then return solution
        if graph and nogoods have both leveled off then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

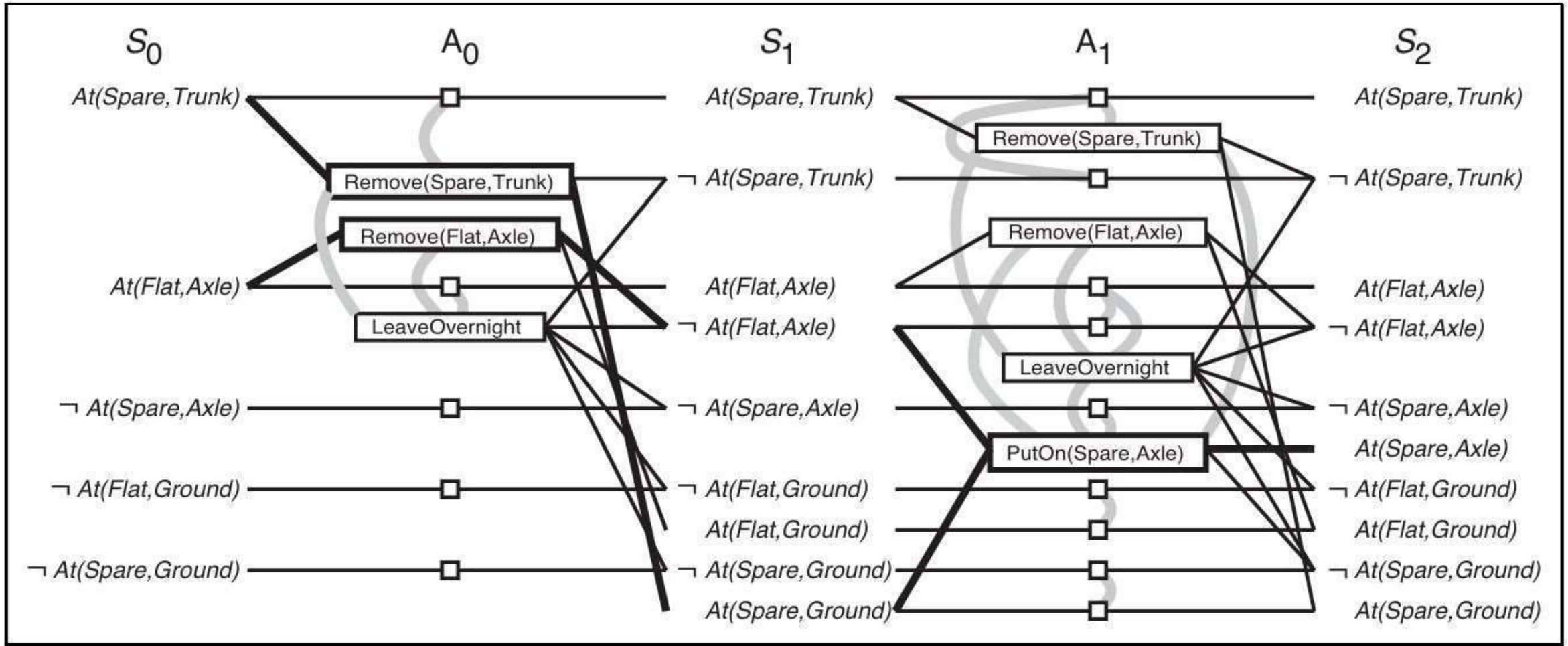


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

- **Inconsistent effects**: Remove(Spare,Trunk) is mutexwith LeaveOvernight because one has the effect At(Spare,Ground) and the other has its negation.
- **Interference:** Remove(Flat,Axle) is mutexwith LeaveOvernight because one has the precondition At(Flat,Axle) and the other has its negation as an effect.
- **Competing needs:** PutOn(Spare,Axle) is mutex with Remove(Flat,Axle) because one has At(Flat,Axle) as a precondition and the other has its negation.
- **Inconsistent support:** At(Spare,Axle) is mutex with At(Flat,Axle) in S2 because the only way of achieving At(Spare,Axle) is by PutOn(Spare,Axle), and that is mutex with the persistence action that is the only way of achieving At(Flat,Axle). Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

Termination of GRAPHPLAN

The properties are as follows:

- **Literals increase monotonically:** Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- **Actions increase monotonically:** Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of the monotonic increase of literals; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
- **Mutexes decrease monotonically:** If two actions are mutex at a given level Ai, then they will also be mutex for all previous levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level Si nor actions that cannot be executed at level Ai. We can see that “mutexes decrease monotonically” is true if you consider that these invisible literals and actions are mutex with everything.

Other Classical Planning Approaches

- Most popular approaches for fully automated planning
  - Translating to a Boolean satisfiability (SAT) problem
  - Forward state-space search with carefully crafted heuristics
  - Search using a planning graph
- Other approaches
  - Planning as first-order logical deduction: Situation calculus
  - Planning as constraint satisfaction
  - Planning as refinement of partially ordered plans

Classical planning as Boolean satisfiability

We show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

- Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.
- Define the initial state: assert F0 for every fluent F in the problem's initial state, and ¬F for every fluent not mentioned in the initial state.
- Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants.

- Add successor-state axioms: For each fluent F, add an axiom of the form $F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$,where ActionCauses F is a disjunction of all the ground actions that have F in their add list, and ActionCausesNot F is a disjunction of all the ground actions that have F in their delete list.
- Add precondition axioms: For each ground action A, add the axiom $A^t \sim PRE(A)^t$, that is, if an action is taken at time t, then the preconditions must have been true.
- Add action exclusion axioms: say that every action is distinct from every other action. The resulting translation is in the form that we can hand to SATPLAN to find a solution.

Planning as first-order logical deduction: Situation calculus

- Some problems cannot express in PDDL. So we can do it in First-order logic using quantifiers. Ex: move all cargo from A to B regardless of how many pieces of cargo there are.
- The propositional logic representation of planning problems also has limitations.
- These limitations can be replaced by a representation called **Situation Calculus .**
  The ontology of the situation calculus is made up of situations, actions, and fluents.
- A fluent could be anything whose value can be changed. These are essentially functions and predicates that ary from one situation to another. A fluent has the value true if it holds in a situation, it has the value false if it does not hold in the situation.
- Situation calculus has each action being defined by two axioms. One of them is a possibility axiom and the other is the effect maximum.

- The possibility axiom tells us when it is possible to execute the action and the effect axiom tells us what happens when the possible action is executed.

- Situation Calculus work like this:
  - The initial state is called a situation. If s is a situation and a is an action, then RESULT(s,a) is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same.
  - A function or relation that can vary from one situation to the next is a fluent. By convention, the situation s is always the last argument to the fluent.
  - Each action's preconditions are described with a possibility axiom that says when the action can be taken.
  - Each fluent is described with a successor-state axiom that says what happens to the fluent, depending on what action is taken.
  - We need unique action axioms so that the agent can deduce that.
  - A solution is a situation (and hence a sequence of actions) that satisfies the goal.

Planning as refinement of partially ordered plans

Partial-order planning

- A **linear planner** builds a plan as a **totally ordered sequence** of plan steps
- A **non-linear planner (aka partial-order planner)** builds up a plan as a set of steps with some temporal constraints
  - constraints like S1<S2 if step S1 must come before S2.
- One **refines** a partially ordered plan (POP) by either:
  - **adding a new plan step**, or
  - **adding a new constraint** to the steps already in the plan.

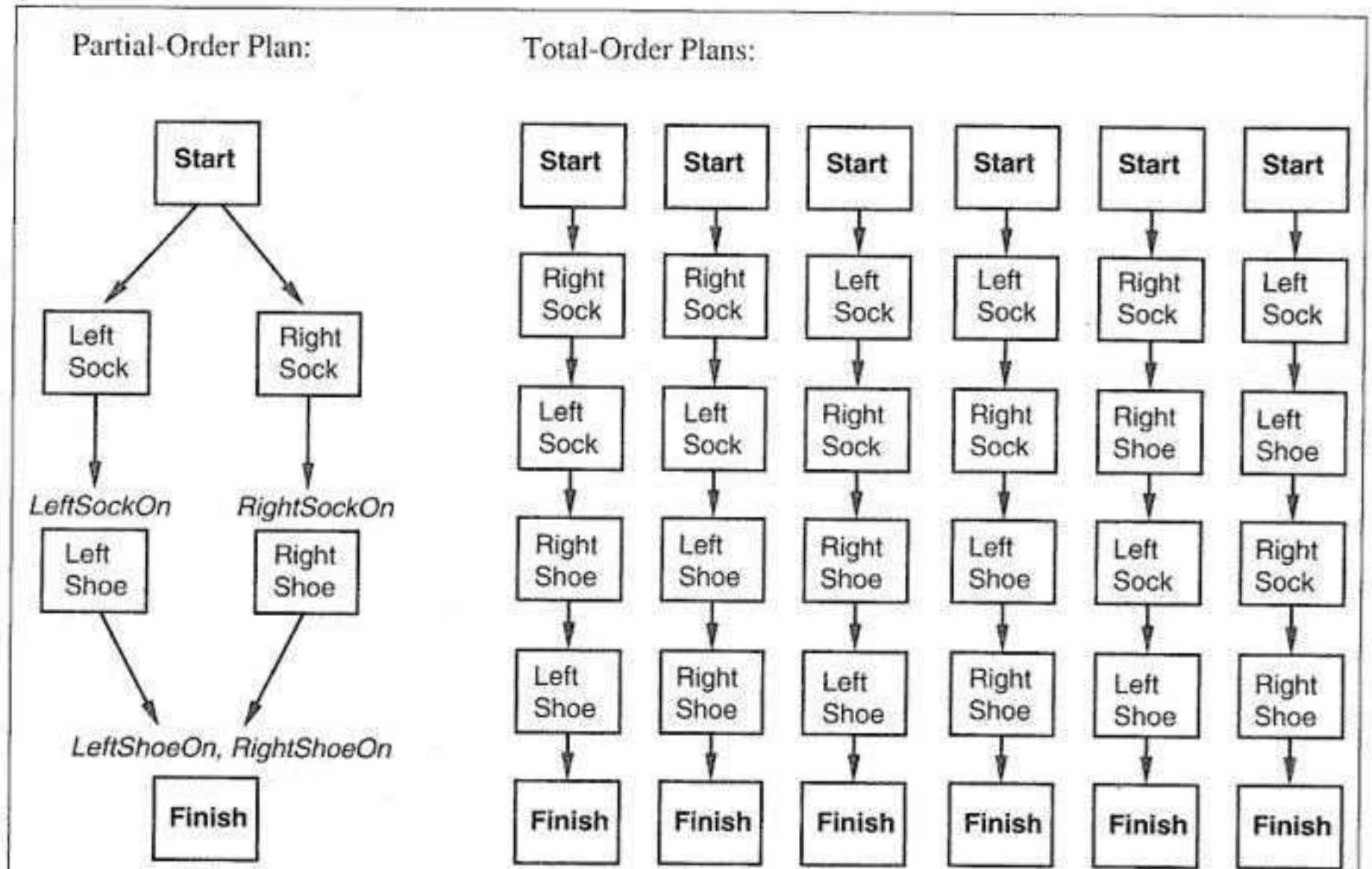


Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

Analysis of Planning approaches

- Planning combines the two major areas of AI we have covered so far: **search and logic**. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution.
- Planning is foremost an exercise in controlling combinatorial explosion. **If there are n propositions in a domain, then there are 2^n states.** As we have seen, planning is PSPACE hard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions.
- Sometimes it is possible to solve a problem efficiently by recognizing that negative SERIALIZABLE SUBGOAL interactions can be ruled out. We say that a problem has serializable subgoals if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals.

Hierarchical Planning

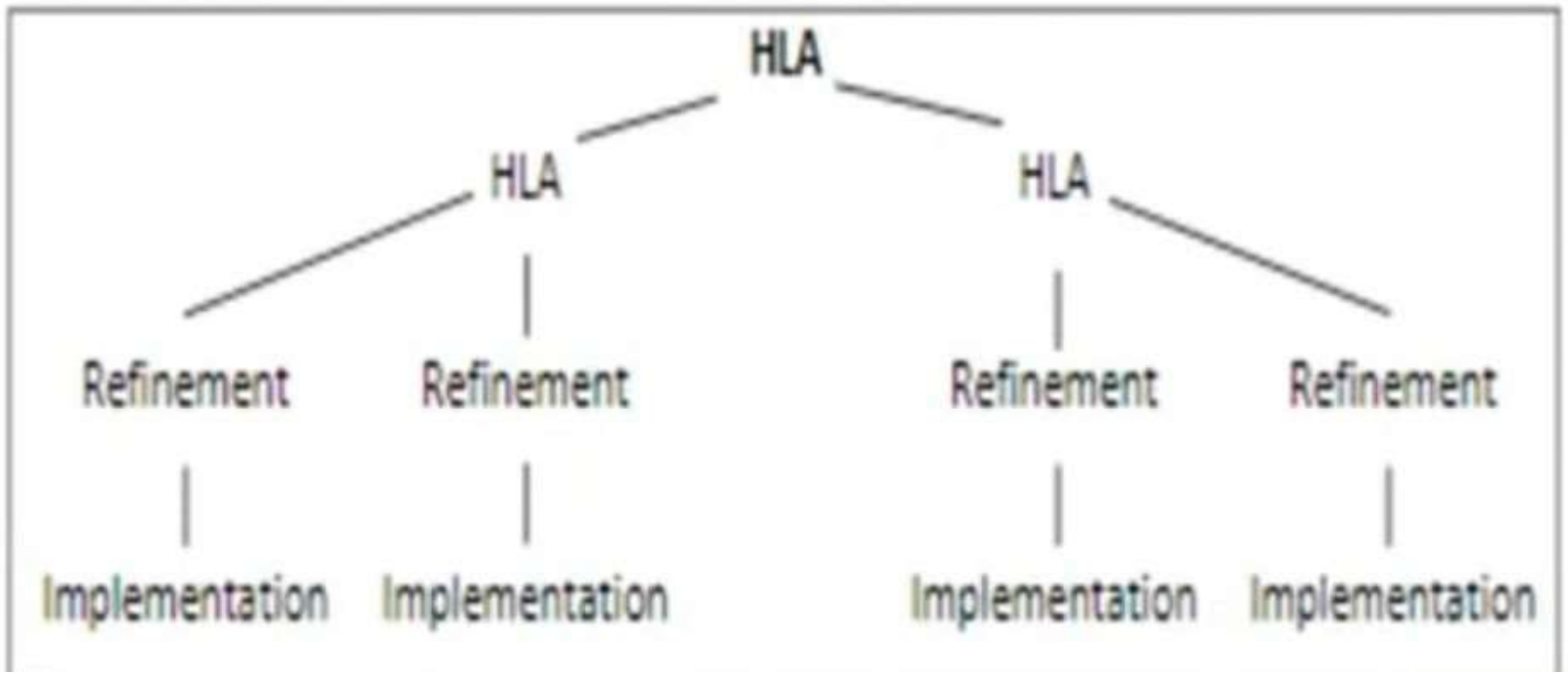
Hierarchical (HTN) Planning

- In order to solve hard problems, a problem solver may have to generate long plans,
- It is important to be able to eliminate some of the details of the problem until a solution that addresses the main aspect is found.
- Then an attempt can be made to fill the appropriate details.
- Early attempts to do this involved the use of macro operators.
- But in this approach, no details were eliminated from actual descriptions of the operators.
- As an example, suppose you want to visit a friend in London but you have a limited amount of cash to spend.

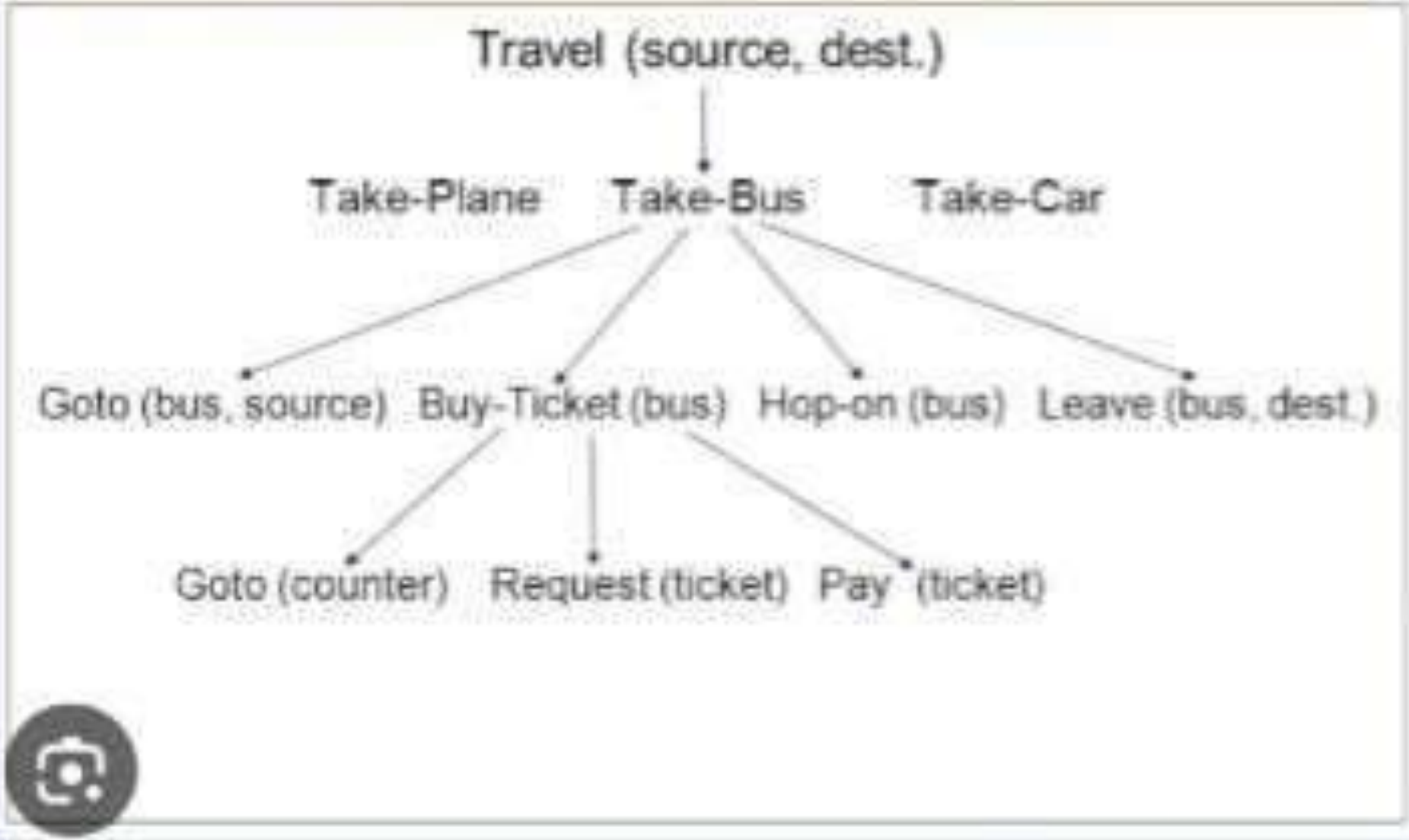
Structure of Hierarchical (HTN) Planning

- Hierarchical Task Networks (from here on abbreviated as HTN) is a hierarchical planning technique where actions are divided into different levels, or hierarchies.
- There are different types of actions residing within these hierarchies, one of them being High Level Actions (from here on abbreviated as HLA).
- These HLAs could be any action that you can divide into smaller actions called refinements. These refinements may be sequences of actions or even other HLAs.

- Refinements in turn are broken down into implementations(also called primitive actions) which refers to any action that has no further refinements or implementations.



Hierarchical Plan - Example



```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier ← a FIFO queue with [Act] as the only element
  loop do
    if EMPTY?(frontier) then return failure
    plan ← POP(frontier)  /* chooses the shallowest plan in frontier */
    hla ← the first HLA in plan, or null if none
    prefix,suffix ← the action subsequences before and after hla in plan
    outcome ← RESULT(problem.INITIAL-STATE, prefix)
    if hla is null then  /* so plan is primitive and outcome is its result */
      if outcome satisfies problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is $[Act]$. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, $outcome$.

Introduction to Artificial Intelligence

UNIT-V

- Probabilistic Reasoning-Acting under Uncertainty
- Basic Probability Notation
- 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

Uncertainty

- When an agent knows enough facts about its environment, the logical plans and actions produces a guaranteed work.
- Unfortunately, *agents never have access to the whole truth about their environment*. Agents act under **uncertainty.**

Nature of Uncertain Knowledge

- The Diagnosis : medicine, automobile repair, or whatever-is a task that almost always involves uncertainty
- Let us try to write rules for dental diagnosis using first-order logic, so that we can see how the logical approach breaks down. Consider the following rule:
  - $\forall p$ *Symptom(p, Toothache)* $\Rightarrow$ *Disease(p, Cavity)* .
- The problem is that this rule is wrong.
- Not all patients with toothaches have cavities; some of them have gum disease, swelling, or one of several other problems
  - $\forall p$ *Symptom(p, Toothache)* $\Rightarrow$ *Disease(p, Cavity) V Disease(p, GumDisease) V Disease(p, Swelling) . . .*

Nature of Uncertain Knowledge…

- to make the rule true, we have to add almost unlimited list of possible causes.
- We could try a causal rule:
  - $\forall p$ *Disease(p, Cavity)* $\Rightarrow$ *Symptom(p, Toothache)*.
- But this rule is also not right either; not all cavities cause pain
- Toothache and a Cavity are unconnected, so the judgement may go wrong.

Nature of Uncertain Knowledge…

- This is a type of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, dating, and so on.
- The agent take action, only a **degree of belief** in the relevant sentences.
- Our main tool for dealing with degrees of belief will be **probability theory**
- **The Probability** assigns to each sentence a numerical degree of belief between 0 and 1.

Uncertainty and rational decisions

- Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called decision theory: **Decision theory = probability theory + utility theory .**
- The fundamental idea of decision theory is that an agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action. This is called the principle of **maximum expected utility (MEU).**

Probability

- Probabilities are used to compute the truth of given statement, written as numbers between 0 and 1, that describes how likely an event is to occur.
- 0 indicates impossibility and 1 indicates certainly.
  - 1. Tossing a coin 2. Tolling a dice
- Probability based reasoning
  - understanding from knowledge
  - how much of uncertainty present in that event.

Probability

- *Probability provides a way of* **summarizing** *the uncertainty, that comes from our laziness and ignorance.*
- Toothache problem - an 80% chance , a probability of 0.8 that the patient has a cavity if he or she has a toothache.
- The 80% summarizes those cases, but both toothache and cavity are unconnected.
- The missing 20% summarizes, all other possible causes of toothache, that we are too lazy or ignorant to confirm or deny.

Laziness: too many antecedents/consequents
Theoretical Ignorance: No complete knowledge
Practical Ignorance: not all tests can be run

- Probabilities between 0 and **1** correspond to intermediate degrees of belief in the truth of the sentence.
- The sentence itself is in *fact* either true or false.
- It is important to note that a degree of belief is different from a degree of truth.
- A probability of 0.8 does not mean "80% true" but rather an 80% degree of belief-that is, a fairly strong expectation.
- Thus, probability theory makes the same ontological commitment as logic-namely, that facts either do or do not hold in the world.
- Degree of truth, as opposed to degree of belief, is the subject of **fuzzy logic**

- In probability theory, a sentence such as
- "The probability that the patient has a cavity is 0.8“,
- is about the agent's beliefs, not directly about the world.
- These percepts create the **evidence,** which are based on probability statements.

Types of random variables

- Boolean random variables
  - Cavity domain (true,false), if Cavity = true then cavity, or
  - if Cavity = false then ¬cavity
- Discrete random variables – countable domain
  - *Weather* might be *(sunny, rainy, cloudy, snow)*
- Continuous random variables – finite set real numbers with equal intervals e.g. interval(0.1)

Atomic events

- The concept of an **atomic event** is useful in understanding the foundations of probability theory.
- It is a complete specification of the state of the world about which the agent is uncertain.
- It can be an assignment of particular values, to all the variables of which the world is composed

Atomic events…

- Atomic events have some important properties
- They are mutually exclusive -at most one can actually be the case.
- The set of all possible atomic events is exhaustive – at least one must be the case.
- Any particular atomic event entails the truth or falsehood of every proposition, whether simple or complex
- Any proposition is logically equivalent to the disjunction of all atomic events that required the truth of proposition.

Prior Probability

- The **unconditional** or **prior probability** associated with a proposition *a,* is the degree of belief according to the absence of any other information;
- it is written as *P(a).*
- For example, if the prior probability that one have a cavity is 0.1, then we would write
  - *P(Cavity = true) = 0.1* or *P(cavity)* = 0.1 .
- It is important to remember that *P(a)* can be used only when there is no other information.

Prior Probability...

- we will use an expression **P(** *Weather),* which denotes a ***vector*** of values, for the probabilities of each individual state of the weather.
  - *P(Weather = sunny) = 0.7*
  - *P(Weather = rain) = 0.2*
  - *P(Weather = cloudy) = 0.08*
  - *P(Weather = snow) = 0.02* .
- we may simply write
- *P(Weather) = (0.7, 0.2, 0.08, 0.02)* .
- This statement defines a prior **probability distribution** for the random variable *Weather.*

Conditional Probability

- The **conditional** or **posterior** probabilities notation is *P(alb),*
- where *a* and *b* are any proposition.
- This is read as "the probability of *a,* given that *all* we know is *b."*
- For example, *P(cavity| toothache) = 0.8*
- if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8.

Conditional probabilities...

- Conditional probabilities can be defined in terms of unconditional probabilities.
- The equation is

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

- whenever $P(b) > 0$.
- This equation can also be written as
- $P(a \wedge b) = P(a/\ b)\ P(b)$ which is called the **product rule.**

Basic Axioms of Probability

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

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

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

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

- The probability of a disjunction is given by

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

BAYES's Rule and Its use

- Bayes' theorem is also known as **Bayes' rule, Bayes' law**, or **Bayesian reasoning**, which determines the probability of an event with uncertain knowledge.
- In probability theory, it relates the conditional probability and marginal probabilities of two random events.
- Bayes' theorem was named after the British mathematician **Thomas Bayes**. The **Bayesian inference** is an application of Bayes' theorem.

- Bayes' theorem can be derived using product rule and conditional probability of event A with known event B:
- As from product rule we can write: $P(A \wedge B) = P(A|B)\, P(B)$ or
- Similarly, the probability of event B with known event A:

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

- Equating right hand side of both the equations , and dividing by P(B) we will get:

$$P(A|B) = \frac{P(B|A)\, P(A)}{P(B)} \quad \text{....(a)}$$

- The above equation (a) is called as Bayes' rule or Bayes' theorem. This equation is basic of most modern AI systems for probabilistic inference.
- It shows the simple relationship between joint and conditional probabilities.
- Here ,P(A|B) is known as posterior, which we need to calculate, and it will be read as Probability of hypothesis A when we have occurred an evidence B.
- P(B|A) is called the likelihood, in which we consider that hypothesis is true, then we calculate the probability of evidence.
- P(A) is called the prior probability, probability of hypothesis before considering the evidence
- P(B) is called marginal probability, pure probability of an evidence.
- In the equation (a), in general, we can write P (B) = P(A)*P(B|Ai), hence the Bayes' rule can be written as:

$$P(A_i|B) = \frac{P(A_i) \cdot P(B|A_i)}{\sum_{i=1}^{k} P(A_i) \cdot P(B|A_i)}$$

- Where A_1, A_2, A_3,....., A_n is a set of mutually exclusive and exhaustive events.

Applying Bayes' rule:

- Bayes' rule allows us to compute the single term P(B|A) in terms of P(A|B), P(B), and P(A). This is very useful in cases where we have a good probability of these three terms and want to determine the fourth one. Suppose we want to perceive the effect of some unknown cause, and want to compute that cause, then the Bayes' rule becomes:

$$\mathbf{P(cause|effect) = \frac{P(effect|cause)\,P(cause)}{P(effect)}}$$

Example-1:

Question: what is the probability that a patient has diseases meningitis with a stiff necfi?

Given Data:

A doctor is aware that disease meningitis causes a patient to have a stiff neck, and it occurs 80% of the time. He is also aware of some more facts, which are given as follows:

- The Known probability that a patient has meningitis disease is 1/30,000.
- The Known probability that a patient has a stiff neck is 2%.

Let a be the proposition that patient has stiff neck and b be the proposition that patient has meningitis. , so we can calculate the following as:

P(a|b) = 0.8

P(b) = 1/30000

P(a)= .02

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} = \frac{0.8*(\frac{1}{30000})}{0.02} = 0.001333333.$$

Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

Application of Bayes' theorem in Artificial intelligence:

- It is used to **calculate the next step of the robot** when the already executed step is given.
- Bayes' theorem is helpful in **weather forecasting**.
- It can solve the **Monty Hall problem**.

Representing Knowledge in an Uncertain Domain

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

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

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

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

The full specification is as follows:

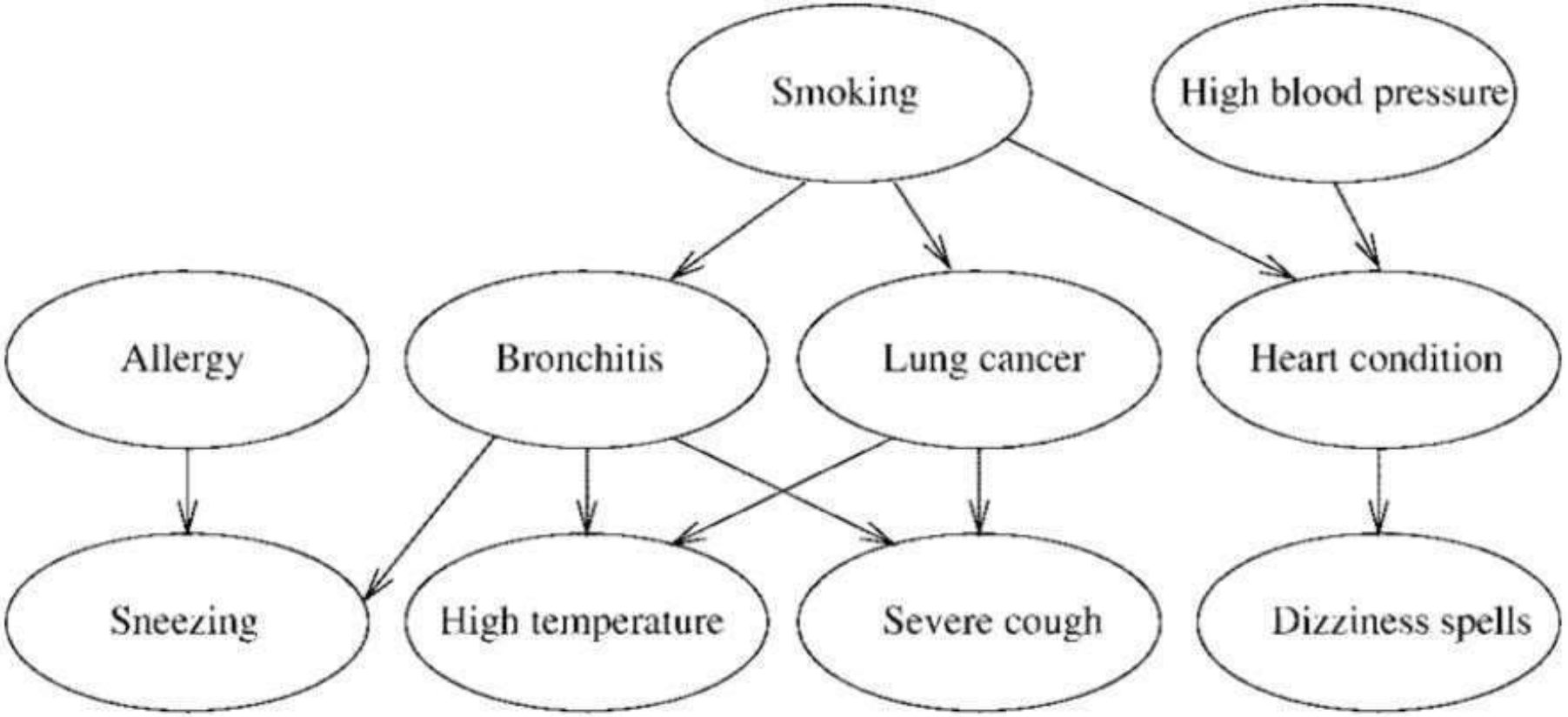
1. Each **node** corresponds to a **random variable**, which may be discrete or continuous.

2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y,X is said to be a parent of Y. The graph has no directed cycles (and hence is a directed acyclic graph, or DAG).

3. Each node Xi has a conditional probability distribution P(Xi|Parents(Xi)) that quantifies the effect of the parents on the node.

- The topology of the network—the set of nodes and links—specifies the conditional independence relationships that hold in the domain.

Topology of Bayesian network



Topology of simplified Bayes net for medical diagnosis.

The Semantics of Bayesian Networks

- There are two ways to understand the semantics of the Bayesian network, which is given below:
- **1. To understand the networfi as the representation of the Joint probability distribution.**
- It is helpful to understand how to construct the network.
- **2. To understand the networfi as an encoding of a collection of conditional independence statements.**
- It is helpful in designing inference procedure.

Example

- To prevent break-ins, Harry put a brand-new burglar alarm at his house. The alarm consistently reacts to a break-in, but it also reacts to little earthquakes. David and Sopia, two of Harry's neighbours, have agreed to call Harry at work when they hear the alarm. David always calls Harry when the alarm goes off, but occasionally he gets distracted by the listening to loud music, so occasionally she doesn't hear the alarm.

Problem

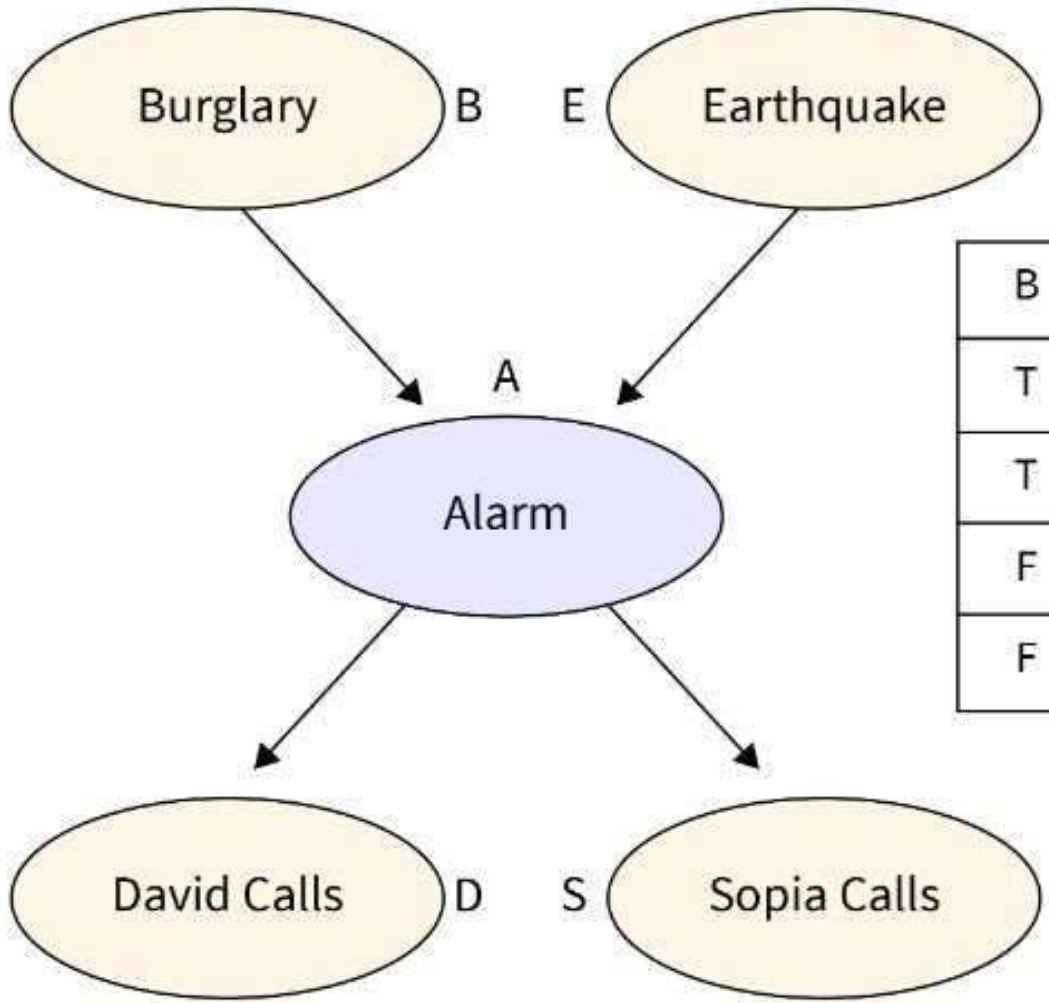
- Determine the likelihood that the alarm went off but that burglary nor an earthquake had taken place, and that both David Sopia had phoned Harry.

Here, we'd want to calculate the likelihood of a burglar alarm

Solution

- Here is provided the Bayesian network for the afore mentioned issue. The network topology demonstrates that earthquake and burglary are the parent nodes of the alarm and directly affect the likelihood that the alarm will sound, but David and Sopia's calls depend on the likelihood that the alarm will sound.
- According to the network, our presumptions did not directly observe the break-in, did not see the tiny earthquake, and did not consult with one another before contacting.
- The conditional probabilities table, or CPT, contains the conditional distributions for each node.
- Because every entry in the table represents an exhaustive set of scenarios for the variable, each row in the CPT must add up to 1.
- A boolean variable in CPT has 2^kprobability for each of its k boolean parents. Hence, CPT will contain 4 probability values if there are two parents.

- **List Of All Events Occurring in a Bayesian Network**
- Burglary (B)
- Earthquake(E)
- Alarm(A)
- David Calls(D)
- Sopia calls(S)



T	0.02
F	0.998

T	0.001
F	0.999

B	E	P(A=T)	P(A=F)
T	T	0.94	0.06
T	F	0.95	0.04
F	T	0.69	0.69
F	F	0.999	0.999

A	P(D=T)	P(D=F)
T	0.91	0.09
F	0.05	0.95

A	P(S=T)	P(S=F)
T	0.75	0.25
F	0.02	0.98

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

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

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

$$=0.00068045$$

Applications of Bayesian Networks in AI

- Some of the most common applications of Bayesian networks in AI include:

- **Prediction and classification:** Bayesian belief networks can be used to predict the probability of an event or classify data into different categories based on a set of inputs. This is useful in areas such as fraud detection, medical diagnosis, and image recognition.
- **Decision making:** Bayesian networks can be used to make decisions based on uncertain or incomplete information. For example, they can be used to determine the optimal route for a delivery truck based on traffic conditions and delivery schedules.
- **Risk analysis:** Bayesian belief networks can be used to analyze the risks associated with different actions or events. This is useful in areas such as financial planning, insurance, and safety analysis.
- **Anomaly detection:** Bayesian networks can be used to detect anomalies in data, such as outliers or unusual patterns. This is useful in areas such as cybersecurity, where unusual network traffic may indicate a security breach.
- **Natural language processing:** Bayesian belief networks can be used to model the probabilistic relationships between words and phrases in natural language, which is useful in applications such as language translation and sentiment analysis.

Efficient Representation of Conditional Probability

- Even if the maximum number of parents **k** is small, filling in the CPT for a node requires up to **O(2^k)**numbers.
- Usually, such relationships are describable by a **canonical distribution,** the complete table can be specified by naming the pattern and perhaps supplying a few parameters—much easier than supplying an exponential number of parameters.
- A **deterministic node** has its value specified exactly by the values of its parents, with no uncertainty.

- The relationship between parent and child can be numerical.
- Uncertain relationships can often be characterized by so-called noisy logical relation ships. The standard example is the noisy-OR relation, which is a generalization of the logical OR.
- Ex: Fever is true if and only if Cold, Flu,or Malaria is true.
- In medicine noisy-OR and noisy-MAX distributions are used to model relationships among diseases and symptoms. With 448 nodes and 906 links, it requires only 8,254 values instead of 133,931,430 for a network with full CPTs.
- Many real-world problems involve continuous quantities, such as height, mass, temperature, and money; in fact, much of statistics deals with random variables whose domains are **continuous.**

- Continuous variables have an infinite number of possible values, so it is impossible to specify conditional probabilities explicitly for each value.
- One possible way to handle continuous variables is to avoid them by using **discretization**—that is, dividing up the possible values into a fixed set of intervals.
- Temperatures could be divided into $(<0^o),(0^o - 100^o),(> 100^o)$
- A network with both discrete and continuous variables is called a **hybrid Bayesian network.**

Approximate inference in Bayesian Networks

Basic idea:

1. Draw N samples from a sampling distribution S.
2. Compute an approximate posterior probability P
3. Show this converges to the true probability P

Outline:

1. Direct sampling: Sampling from an empty network
2. Rejection sampling: reject samples disagreeing with the evidence
3. Likelihood weighting: use evidence to weight samples
4. Markov chain Monte Carlo (MCMC): sample from a stochastic process whose stationary distribution is the true posterior

Direct sampling:

Samples:

s_1: $(\neg a, \neg b, c, d)$;
s_2: $(a, \neg b, \neg c, d)$;
s_3: $(\neg a, \neg b, c, \neg d)$;
s_4: $(\neg a, \neg b, \neg c, \neg d)$;
s_5: $(\neg a, b, c, d)$;
s_6: $(a, \neg b, \neg c, \neg d)$;
s_7: $(\neg a, b, \neg c, d)$;
s_8: $(a, b, \neg c, \neg d)$;
s_9: $(\neg a, \neg b, \neg c, d)$;
s_{10}: $(a, \neg b, c, \neg d)$;

Queries:

$\mathbf{P}(\neg c)$

$\mathbf{P}(\neg a, b)$

$\mathbf{P}(\neg b \mid \neg d)$

$\mathbf{P}(a \mid b, \neg c)$

Prior Sampling:

Function *prior-sampling(bayes net)*

for $i = 1, 2, \ldots, n$ **do**

Sample x_i from $P(X_i | Parents(X_i))$;

return $(x_1, x_2, \ldots, x_n)$;

end

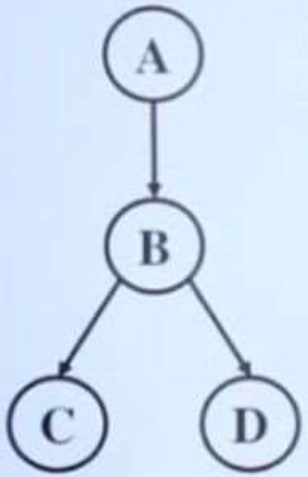
Rejection sampling:

A	P(A)
a	1/5
¬a	4/5

A	B	P(B|A)
a	b	1/5
¬a	b	1/2
a	¬b	4/5
¬a	¬b	1/2

B	C	P(C|B)
b	c	1/4
¬b	c	3/4
b	¬c	2/5
¬b	¬c	3/5

B	D	P(D|B)
b	d	1/2
¬b	d	4/5
b	¬d	1/2
¬b	¬d	1/5



s_1: $(\neg a, \neg b, c, d)$;
s_2: $(a, \neg b, \neg c, d)$;
s_3: $(\neg a, \neg b, c, \neg d)$;
s_4: $(\neg a, \neg b, \neg c, \neg d)$

s_5: $(\neg a, b, c, d)$;
s_6: $(a, \neg b, \neg c, \neg d)$;
s_7: $(\neg a, b, \neg c, d)$;
s_8: $(a, b, \neg c, \neg d)$

Cross out the samples that would be rejected by rejection sampling to estimate $P(a \mid \neg c, \neg d)$

Rejection Sampling:

Function *rejection-sampling(bayes-net, evidence-instantiation)*

for $i = 1, 2, \ldots, n$ **do**

Sample x_i from $P(X_i|Parents(X_i))$;

if x_i *not consistent with evidence* **then**

Reject: return – no sample is generated in this cycle

end

return $(x_1, x_2, \ldots, x_n)$;

end

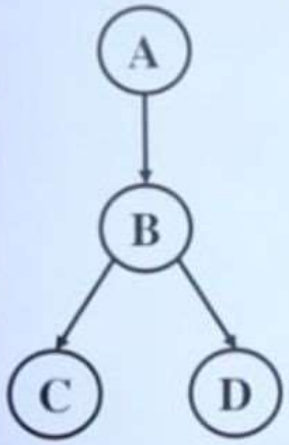
Likelihood weighting:

A	P(A)
a	1/5
¬a	4/5

A	B	P(B|A)
a	b	1/5
¬a	b	1/2
a	¬b	4/5
¬a	¬b	1/2

B	C	P(C|B)
b	c	1/4
¬b	c	3/4
b	¬c	2/5
¬b	¬c	3/5

B	D	P(D|B)
b	d	1/2
¬b	d	4/5
b	¬d	1/2
¬b	¬d	1/5



s_1: $(a, \neg b, \neg c, \neg d)$;
s_2: $(\neg a, b, \neg c, \neg d)$;
s_3: $(a, b, \neg c, \neg d)$;

Estimate the likelihood weight of each sample and thereby estimate $P(a \mid \neg c, \neg d)$

Likelihood Weighting:

Function *likelihood-weighting(bayes-net, evidence-instantiation)*

$w = 1.0$;
for $i = 1, 2, \ldots, n$ **do**
  Sample x_i from $P(X_i|Parents(X_i))$;
  if x_i *is an evidence variable* **then**
    $X_i = observation\ x_i$ for X_i ;
    Set $w = w * P(x_i|Parents(X_i))$;
  end
  else
    Sample x_i from $P(X_i|Parents(X_i))$;
  end
  return $(x_1, x_2, \ldots, x_n)$, w;
end

Markov Chain Monte Carlo (MCMC):

- Markov Chain Monte Carlo (MCMC) is a probabilistic sampling technique to approximate the posterior distribution of variables in Bayesian network. It is particularly useful in dealing with high-dimensional and complex probability distributions.
- MCMC algorithms work quite differently from rejection sampling and likelihood weighting. Instead of generating each sample from scratch, MCMC algorithms generate each sample by making a random change to the preceding sample.
- Common MCMC algorithms include Metropolis Hastings and Gibbs Sampling.

- It is therefore helpful to think of an MCMC algorithm as being in a particular current state specifying a value for every variable and generating a next state by making random changes to the current state.
- The **Gibbs sampling algorithm** for Bayesian networks starts with an arbitrary state (with the evidence variables fixed at their observed values) and generates a next state by randomly sampling a value for one of the nonevidence variables Xi.
- The evidence variables are fixed to their observed values and the nonevidence variables are initialized randomly. Now the nonevidence variables are sampled repeatedly in an arbitrary order.
- In a Markov Chain, the future state depends on the current state and is independent of the previous state

Gibbs Sampling Algorithm

- Fix the values of observed variables.
- Set the values of all non-observed variables randomly.
- Perform a random walk through the space of complete variable assignments. On each move:
  1. Pick a variable X
  2. Calculate P(X=true | all other variables)
  3. Set X to true with that probability.
- Repeat many times. Frequently with which any variable X is true is it's posterior probability.
- Converges to true posterior when frequencies stop changing significantly.

Stationary Distribution means the distribution when the probability Stop changing.

Summary: Bayesian Networks Sampling

- Direct sampling/Prior Sampling(P(Q):very Simple and inefficient
- Rejection sampling P(Q|e): bit more efficient
- Likelihood weighting P(Q|e): Evidence only effects down stream random variables
- Markov chain Monte Carlo (MCMC) P(Q|e): Efficient and effects both down stream and up stream random variables

Difference between Markov networks and Bayesian networks

- Markov networks can have loops or cycles, while networks cannot.

- Markov networks are better for showing when things are related to each other in a more equal or balanced way. Bayesian networks are better for showing when one thing has a bigger effect on another thing.

Relational and First-order Probability Models

- **Relational probability models:**

Like first-order logic, RPMs have constant, function, and predicate symbols. (It turns out to be easier to view predicates as functions that return true or false.) We will also assume a type signature for each function, that is, a specification of the type of each argument and the function's value. If the type of each object is known, many spurious possible worlds are eliminated by this mechanism.

For the book-recommendation domain, the types are Customer and Book, and the type signatures for the functions and predicates are as follows:

Honest : Customer $\rightarrow$ { true, false} *Kindness :* Customer $\rightarrow$ {1,2,3,4,5}

Quality : Book $\rightarrow$ {1,2,3,4,5}

Recommendation : Customer $\times$ Book $\rightarrow$ {1,2,3,4,5}

The constant symbols will be whatever customer and book names appear in the retailer's data

- Given the constants and their types, together with the functions and their type signatures, the random variables of the RPM are obtained by instantiating each function with each possible combination of objects: Honest(C1), Quality(B2), Recommendation(C1,B2), and so on.

Honest(c) ~ 0.99,0.01

Kindness(c) ~ 0.1,0.1,0.2,0.3,0.3

Quality(b) ~ 0.05,0.2,0.4,0.2,0.15

Recommendation(c,b) ~ RecCPT(Honest(c),Kindness(c),Quality(b))

where RecCPT is a separately defined conditional distribution with 2×5×5=50 rows, each with 5 entries.

We can refine the model by introducing a **context-specific independence** to reflect the fact that dishonest customers ignore quality when giving a recommendation; moreover, kindness plays no role in their decisions.

A context-specific independence allows a variable to be independent of some of its parents given certain values of others;

thus, *Recommendation(c,b)* is independent of *Kindness(c)* and *Quality(b)* when *Honest(c)=false:*

Recommendation(c,b)~ **if** Honest(c) **then** HonestRecCPT(Kindness(c),Quality(b))

else <0.4,0.1,0.0,0.1,0.4> .