ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Course	:	**ARTIFICIAL INTELLIGENCE**
Course Code	:	**24FMCA34T**
Branch	:	**MCA**
Prepared by	:	**S.THABREEZ BASHA**
Designation	:	**Assistant Professor**
Department	:	**MCA**

ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Title of the Course	**:**	**ARTIFICIAL INTELLIGENCE**
Category	**:**	**PC**
Course Code	**:**	**24FMCA34T**
Branch	**:**	**MCA**
Semester	**:**	**III Semester**

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

COURSE OBJECTIVES:

- Introduce the fundamental concepts and history of Artificial Intelligence.
- Understand and apply techniques in constraint satisfaction problems, adversarial search, and logical reasoning.
- Develop a deep understanding of First-Order Logic.
- Explore knowledge representation techniques and automated planning methods.
- Learn to handle uncertainty in AI through probabilistic reasoning.

UNIT I **11 Hrs**

INTRODUCTION: What is Artificial Intelligence? The Foundations of Artificial Intelligence, The History of Artificial Intelligence, Risks and Benefits of Artificial Intelligence.
INTELLIGENT AGENTS: Agents and Environments, The Nature of Environments.
SOLVING PROBLEMS BY SEARCHING: Problem-Solving Agents, Example Problems, Search Algorithms, Uninformed Search Strategies.

UNIT II **11 Hrs**

CONSTRAINT SATISFACTION PROBLEMS: Defining Constraint Satisfaction Problems, Constraint Propagation: Inference in CSPs, Backtracking Search for Constraint Satisfaction Problems, Local Search for Constraint Satisfaction Problems.
ADVERSARIAL SEARCH AND GAMES: Game Theory, Optimal Decisions in Games, Heuristic Alpha- Beta Tree Search.
LOGICAL AGENTS: Knowledge-Based Agents, Logic, Propositional Logic: A Very Simple Logic, Propositional Theorem Proving: Inference and proofs, Proof by resolution, Horn clauses and definite clauses, Forward and backward chaining, Effective Propositional Model Checking.

UNIT III **10 Hrs**

FIRST-ORDER LOGIC: Representation, Syntax and Semantics of First-Order Logic, Using First-Order Logic, Knowledge Engineering in First-Order Logic.
INFERENCE IN FIRST-ORDER LOGIC: Propositional vs. First-Order Inference, Unification and First- Order Inference, Forward Chaining, Backward Chaining.

UNIT IV **10 Hrs**

KNOWLEDGE REPRESENTATION: Onto logical Engineering, Categories and Objects, Events, Reasoning Systems for Categories, Reasoning with Default Information.
AUTOMATED PLANNING: Definition of Classical Planning, Algorithms for Classical Planning, Heuristics for Planning, Hierarchical Planning, Time, Schedules, and Resources.

UNIT V **10 Hrs**

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

PROBABILISTIC REASONING: Representing Knowledge in an Uncertain Domain, The Semantics of Bayesian Networks, Exact Inference in Bayesian Networks.

TEXT BOOK:

1. Stuart Russell and PeterNorvig. Artificial Intelligence A Modern Approach, Fourth Edition. Pearson Education.

REFERENCE BOOKS:

1. .E.Richand K.Knight. Artificial Intelligence, TMH.
2. Patrick Henny Winston. Artificial Intelligence, Pearson Education.
3. Shivani Goel. Artificial Intelligence. Pearson Education.

COURSE OUTCOMES:

The Student will be able to

1. Comprehend the fundamental concepts of Artificial Intelligence.
2. Describe the constraint satisfaction problems.
3. Apply the concepts of First-Order Logic.
4. Summarize the knowledge Representation and develop automated planning solutions.
5. Analyze uncertainty using probabilistic methods.

CO-PO MAPPING:

Course Outcomes	Foundation Knowledge	Problem Analysis	Development of Solutions	Modern Tool Usage	Individual and Teamwork	Project Management and Finance	Ethics	Life-long Learning
24FMCA034T.1	2	2	1	-	-	-	-	1
24FMCA034T.2	2	2	1	-	-	-	-	-
24FMCA034T.3	3	2	1	-	-	-	-	-
24FMCA034T.4	2	2	1	-	-	-	-	-
24FMCA034T.5	3	3	2	-	-	-	-	-

UNIT- I

FOUNDATIONS OF AI

WHAT IS AI?

Artificial Intelligence (AI) is a branch of Science which deals with helping machines finding solutions to complex problems in a more human-like fashion. This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way. A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behaviour appears. AI is generally associated with Computer Science, but it has many important links with other fields such as Maths, Psychology, Cognition, Biology and Philosophy, among many others. Our ability to combine knowledge from all these fields will ultimately benefit our progress in the quest of creating an intelligent artificial being.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavour. In this sense, it is truly a universal field.

HISTORY OF AI

The origin of artificial intelligence lies in the earliest days of machine computations. During the 1940s and 1950s, AI begins to grow with the emergence of the modern computer. Among the first researchers to attempt to build intelligent programs were Newell and Simon. Their first well known program, logic theorist, was a program that proved statements using the accepted rules of logic and a problem-solving program of their own design. By the late fifties, programs existed that could do a passable job of translating technical documents and it was seen as only a matter of extra databases and more computing power to apply the techniques to less formal, more ambiguous texts. Most problem-solving work revolved around the work of Newell, Shaw and Simon, on the general problem solver (GPS). Unfortunately, the GPS did not fulfil its promise and did not because of some simple lack of computing capacity. In the 1970's the most important concept of AI was developed known as Expert System which exhibits as a set rules the knowledge of an expert. The application area of expert system is very large. The 1980's saw the development of neural networks as a method learning examples.

Prof. Peter Jackson (University of Edinburgh) classified the history of AI into three periods as:

1. Classical
2. Romantic
3. Modern

1. Classical Period:

It was started from 1950. In 1956, the concept of Artificial Intelligence came into existence. During this period, the main research work carried out includes game plying, theorem proving and concept of state space approach for solving a problem.

2. Romantic Period:

It was started from the mid 1960 and continues until the mid 1970. During this period people were interested in making machine understand, that is usually mean the understanding of natural language. During this period the knowledge representation technique "semantic net" was developed.

3. Modern Period:

It was started from 1970 and continues to the present day. This period was developed to solve more complex problems. This period includes the research on both theories and practical aspects of Artificial Intelligence. This period includes the birth of concepts like Expert system, Artificial Neurons, Pattern Recognition etc. The research of the various advanced concepts of Pattern Recognition and Neural Network are still going on.

Components of AI

There are three types of components in AI

1) Hardware Components of AI

a) Pattern Matching
b) Logic Representation
c) Symbolic Processing
d) Numeric Processing
e) Problem Solving
f) Heuristic Search
g) Natural Language processing
h) Knowledge Representation
i) Expert System
j) Neural Network
k) Learning
l) Planning
m) Semantic Network

2) Software Components

a) Machine Language
b) Assembly language
c) High level Language
d) LISP Language
e) Fourth generation Language
f) Object Oriented Language
g) Distributed Language
h) Natural Language
i) Particular Problem Solving Language

3) Architectural Components

a) Uniprocessor
b) Multiprocessor
c) Special Purpose Processor
d) Array Processor
e) Vector Processor
f) Parallel Processor
g) Distributed Processor

Definition of Artificial intelligence

1. AI is the study of how to make computers do things which at the moment people do better. This is ephemeral as it refers to the current state of computer science and it excludes a major area; problems that cannot be solved well either by computers or by people at the moment.

2. AI is a field of study that encompasses computational techniques for performing tasks that apparently require intelligence when performed by humans.

3. AI is the branch of computer science that is concerned with the automation of intelligent behaviour. A I is based upon the principles of computer science namely data structures used in knowledge representation, the algorithms needed to apply that knowledge and the languages and programming techniques used in their implementation.

4. AI is the field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes.

5. AI is about generating representations and procedures that automatically or autonomously solve problems heretofore solved by humans.

6. A I is the part of computer science concerned with designing intelligent computer systems, that is, computer systems that exhibit the characteristics we associate with intelligence in human behaviour such as understanding language, learning, reasoning and solving problems.

7. A I is the study of mental faculties through the use of computational models.

8. A I is the study of the computations that make it possible to perceive, reason, and act.

9. A I is the exciting new effort to make computers think machines with minds, in the full and literal sense.

10. AI is concerned with developing computer systems that can store knowledge and effectively use the knowledge to help solve problems and accomplish tasks. This brief statement sounds a lot like one of the commonly accepted goals in the education of humans. We want students to learn (gain knowledge) and to learn to use this knowledge to help solve problems and accomplish tasks.

STRONG AND WEAK AI

There are two conceptual thoughts about AI namely the Weak AI and Strong AI. The strong AI is very much promising about the fact that the machine is almost capable of solve a complex problem like an intelligent man. They claim that a computer is much more efficient to solve the problems than some of the human experts. According to strong AI, the computer is not merely a tool in the study of mind, rather the appropriately programmed computer is really a mind. Strong AI is the supposition that some forms of artificial intelligence can truly reason and solve problems. The term strong AI was originally coined by John Searle.

In contrast, the weak AI is not so enthusiastic about the outcomes of AI and it simply says that some thinking like features can be added to computers to make them more useful tools. It says that computers to make them more useful tools. It says that computers cannot be made intelligent equal to human being, unless constructed significantly differently. They claim that computers may be similar to human experts but not equal in any cases. Generally weak AI refers to the use of software to study or accomplish specific problem solving that do not encompass the full range of human cognitive abilities. An example of weak AI would be a chess program. Weak AI programs cannot be called "intelligent" because they cannot really think.

THE STATE OF THE ART

The increasingly advanced technology provides researchers with new tools that are capable of achieving important goals, and these tools are great starting points in and of themselves. Among the achievements of recent years, the following are some specific domains:

- **Machine learning;**
- **Reinforcement learning;**
- **Deep learning;**
- **Natural language processing.**

Machine Learning (ML)

Machine learning is a subcategory of AI that often uses statistical techniques to give machines the ability to absorb data without explicitly receiving instructions to do so. This process is known as 'training' a 'model' using a learning 'algorithm', which progressively improves performance on a specific activity. The successes achieved in this field have encouraged researchers to push harder on the accelerator.

Reinforcement Learning (RL)

Reinforcement learning is an area of ML that is related to software agents that learn 'goal-oriented' behavior by trying and making mistakes in environments that provide rewards in response to the agents' actions (called 'Policy') toward achieving the objectives.

This field is perhaps the one that has most captured the attention of researchers in the last decade.

Deep Learning

Also within ML, deep learning takes inspiration from the activity of neurons within the brain to learn how to recognize complex patterns through learned data. This is thanks to the use of algorithms, mainly statistical calculations. The word 'deep' refers to the large number of

levels of neurons that ML models simultaneously, which helps acquire rich representations of data to obtain performance gains.

Natural Language Processing (NLP)

Natural language processing is the mechanism by which machines acquire the ability to analyze, understand, and manipulate textual data. 2019 was a great year for NPL with Google AI's BERT and Transformer, Allen Institute's ELMo, OpenAI's Transformer, Ruder and Howard's ULMFit, and finally, Microsoft's MT-DNN. All of these have shown that pre-taught language models can substantially improve performance on a wide variety of NLP tasks.

INTELLIGENT AGENTS

AGENTS

An AI system is composed of an agent and its environment. The agents act in their environment. The environment may contain other agents.

An **agent** is anything that can perceive its environment through **sensors** and acts upon that environment through **effectors.**

- A **human agent** has sensory organs such as eyes, ears, nose, tongue and skin parallel to the sensors, and other organs such as hands, legs, mouth, for effectors.
- A **robotic agent** replaces cameras and infrared range finders for the sensors, and various motors and actuators for effectors.
- A **software agent** has encoded bit strings as its programs and actions.

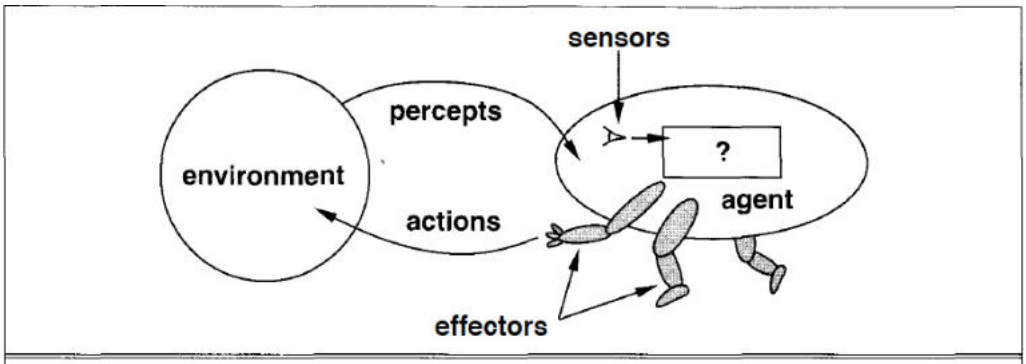


Figure 2.1 Agents interact with environments through sensors and effectors.

Agent Terminology

- **Performance Measure of Agent** − It is the criteria, which determines how successful an agent is.
- **Behavior of Agent** − It is the action that agent performs after any given sequence of percepts.
- **Percept** − It is agent's perceptual inputs at a given instance.
- **Percept Sequence** − It is the history of all that an agent has perceived till date.
- **Agent Function** − It is a map from the precept sequence to an action.

How Agents Should Act

A rational agent is one that does the right thing. Obviously, this is better than doing the wrong thing, but what does it mean? As a first approximation, we will say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding how and when to evaluate the agent's success.

We use the term performance measure for the how—the criteria that determine how successful an agent is. Obviously, there is not one fixed measure suitable for all agents. We could ask the agent for a subjective opinion of how happy it is with its own performance, but some agents would be unable to answer, and others would delude themselves. (Human agents in particular are notorious for "sour grapes"—saying they did not really want something after they are unsuccessful at getting it.) Therefore, we will insist on an objective performance measure imposed by some authority. In other words, we as outside observers establish a standard of what it means to be successful in an environment and use it to measure the performance of agents.

As an example, consider the case of an agent that is supposed to vacuum a dirty floor. A plausible performance measure would be the amount of dirt cleaned up in a single eight-hour shift. A more sophisticated performance measure would factor in the amount of electricity consumed and the amount of noise generated as well. A third performance measure might give highest marks to an agent that not only cleans the floor quietly and efficiently, but also finds time to go windsurfing at the weekend.'

The when of evaluating performance is also important. If we measured how much dirt the agent had cleaned up in the first hour of the day, we would be rewarding those agents that start fast (even if they do little or no work later on), and punishing those that work consistently. Thus, we want to measure performance over the long run, be it an eight-hour shift or a lifetime.

We need to be careful to distinguish between rationality and omniscience. An omniscient agent knows the actual outcome of its actions, and can act accordingly; but omniscience is; impossible in reality.

In summary, what is rational at any given time depends on four things:

- The performance measure that defines degree of success.
- Everything that the agent has perceived so far. We will call this complete perceptual history the percept sequence.
- What the agent knows about the environment.
- The actions that the agent can perform.

This leads to a definition of an ideal rational agent: For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

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

Rationality of an agent depends on the following −

- The **performance measures**, which determine the degree of success.
- Agent's **Percept Sequence** till now.
- The agent's **prior knowledge about the environment**.
- The **actions** that the agent can carry out.

A rational agent always performs right action, where the right action means the action that causes the agent to be most successful in the given percept sequence. The problem the agent solves is characterized by Performance Measure, Environment, Actuators, and Sensors (PEAS).

ENVIRONMENTS

In this section, we will see how to couple an agent to an environment. In all cases, however, the nature of the connection between them is the same: actions are done by the agent on the environment, which in turn provides percepts to the agent. First, we will describe the different types of environments and how they affect the design of agents. Then we will describe environment programs that can be used as testbeds for agent programs.

Properties of environments

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

Accessible vs. inaccessible

If an agent's sensory apparatus gives it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.

Deterministic vs. nondeterministic.

If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the

environment is inaccessible, however, then it may appear to be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. Thus, it is often better to think of an environment as deterministic or nondeterministic/rom the point of view of the agent.

Episodic vs. nonepisodic

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

Static vs. dynamic

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is semidynamic.

Discrete vs. continuous

If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep through a range of continuous values.

We will see that different environment types require somewhat different agent programs to deal with them effectively. It will turn out, as you might expect, that the hardest case is inaccessible, nonepisodic, dynamic, and continuous.

Environment programs

The generic environment program in Figure 2.14 illustrates the basic relationship between agents and environments. In this book, we will find it convenient for many of the examples and exercises to use an environment simulator that follows this program structure. The simulator takes one or more agents as input and arranges to repeatedly give each agent the right percepts and receive back an action. The simulator then updates the environment based on the actions, and possibly other dynamic processes in the environment that are not considered to be agents (rain, for example). The environment is therefore defined by the initial state and the update function. Of course, an agent that works in a simulator ought also to work in a real environment that provides the same kinds of percepts and accepts the same kinds of actions.

```
procedure RUN-ENVIRONMENT(state, UPDATE-FN, agents, termination)
  inputs: state, the initial state of the environment
          UPDATE-FN, function to modify the environment
          agents, a set of agents
          termination, a predicate to test when we are done

  repeat
     for each agent in agents do
        PERCEPT[agent] ← GET-PERCEPT(agent,state)
     end
     for each agent in agents do
        ACTION[agent] ← PROGRAM[agent](PERCEPT[agent])
     end
     state ← UPDATE-FN(actions, agents, state)
  until termination(state)
```

Figure 2.14 The basic environment simulator program. It gives each agent its percept, gets an action from each agent, and then updates the environment.

THE NATURE OF ENVIRONMENTS

Some programs operate in the entirely **artificial environment** confined to keyboard input, database, computer file systems and character output on a screen.

In contrast, some software agents (software robots or softbots) exist in rich, unlimited softbots domains. The simulator has a **very detailed, complex environment**. The software agent needs to choose from a long array of actions in real time. A softbot designed to scan the online preferences of the customer and show interesting items to the customer works in the **real** as well as an **artificial** environment.

The most famous **artificial environment** is the **Turing Test environment**, in which one real and other artificial agents are tested on equal ground. This is a very challenging environment as it is highly difficult for a software agent to perform as well as a human.

Turing Test

- The success of an intelligent behavior of a system can be measured with Turing Test.
- Two persons and a machine to be evaluated participate in the test. Out of the two persons, one plays the role of the tester. Each of them sits in different rooms. The tester is unaware of who is machine and who is a human. He interrogates the questions by typing and sending them to both intelligences, to which he receives typed responses.
- This test aims at fooling the tester. If the tester fails to determine machine's response from the human response, then the machine is said to be intelligent.

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.

So far, we have talked about agents by describing their behavior—the action that is performed after any given sequence of percepts. Now, we will have to bite the bullet and talk about how the insides work.

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

The relationship among agents, architectures, and programs can be summed up as follows:

agent = architecture + program

Before we design an agent program, we must have a pretty good idea of the possible percepts and actions, what goals or performance measure the agent is supposed to achieve, and what sort of environment it will operate in. These come in a wide variety. Figure 2.3 shows the basic elements for a selection of agent types.

It may come as a surprise to some readers that we include in our list of agent types programs that seem to operate in the entirely artificial environment defined by keyboard input and character output on a screen. "Surely," one might say, "this is not a real environment, is it?" In fact, what matters is not the distinction between "real" and "artificial" environments, but the complexity of the relationship among the behavior of the agent, the percept sequence generated by the environment, and the goals that the agent is supposed to achieve. Some "real" environments are actually quite simple. For example, a robot designed to inspect parts as they come by on a conveyer belt can make use of a number of simplifying assumptions: that the lighting is always just so, that the only thing on the conveyer belt will be parts of a certain kind, and that there are only two actions—accept the part or mark it as a reject.

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

Figure 2.3 Examples of agent types and their PAGE descriptions.

Agent Programs

Intelligent Agents will all have the same skeleton, namely, accepting percepts from an environment and generating actions. The early versions of agent programs will have a very simple form (Figure 2.4). Each will use some internal data structures that will be updated as new percepts arrive. These data structures are operated on by the agent's decision-making procedures to generate an action choice, which is then passed to the architecture to be executed.

There are two things to note about this skeleton program. First, even though we defined the agent mapping as a function from percept sequences to actions, the agent program receives only a single percept as its input. It is up to the agent to build up the percept sequence in memory, if it so desires. In some environments, it is possible to be quite successful without storing the percept sequence, and in complex domains, it is infeasible to store the complete sequence.

```
function SKELETON-AGENT(percept) returns action
  static: memory, the agent's memory of the world

  memory ← UPDATE-MEMORY(memory, percept)
  action ← CHOOSE-BEST-ACTION(memory)
  memory ← UPDATE-MEMORY(memory, action)
  return action
```

Figure 2.4 A skeleton agent. On each invocation, the agent's memory is updated to reflect the new percept, the best action is chosen, and the fact that the action was taken is also stored in memory. The memory persists from one invocation to the next.

Second, the goal or performance measure is not part of the skeleton program. This is because the performance measure is applied externally to judge the behavior of the agent, and

it is often possible to achieve high performance without explicit knowledge of the performance measure (see, e.g., the square-root agent).

Example

At this point, it will be helpful to consider a particular environment, so that our discussion can become more concrete. Mainly because of its familiarity, and because it involves a broad range of skills, we will look at the job of designing an automated taxi driver.

We must first think about the percepts, actions, goals and environment for the taxi. They are summarized in Figure 2.6 and discussed in turn.

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.

The taxi will need to know where it is, what else is on the road, and how fast it is going. This information can be obtained from the percepts provided by one or more controllable TV cameras, the speedometer, and odometer. To control the vehicle properly, especially on curves, it should have an accelerometer; it will also need to know the mechanical state of the vehicle, so it will need the usual array of engine and electrical system sensors. It might have instruments that are not available to the average human driver: a satellite global positioning system (GPS) to give it accurate position information with respect to an electronic map; or infrared or sonar sensors to detect distances to other cars and obstacles. Finally, it will need a microphone or keyboard for the passengers to tell it their destination.

The actions available to a taxi driver will be more or less the same ones available to a human driver: control over the engine through the gas pedal and control over steering and braking. In addition, it will need output to a screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles.

What performance measure would we like our automated driver to aspire to? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time and/or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits. Obviously, some of these goals conflict, so there will be trade-offs involved.

Finally, were this a real project, we would need to decide what kind of driving environment the taxi will face. Should it operate on local roads, or also on freeways? Will it be in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not? Will it always be driving on the right, or might we want it to be flexible enough to drive on the left in case we want to operate taxis in Britain or Japan? Obviously, the more restricted the environment, the easier the design problem.

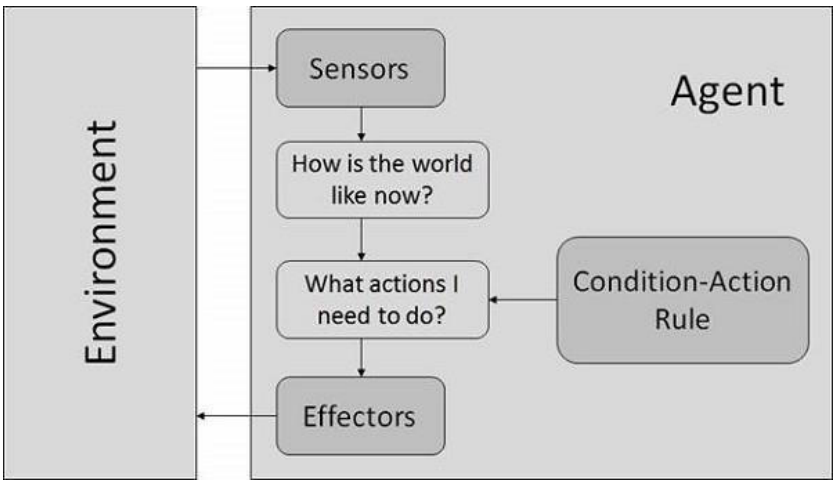
We will consider four types of agent program:

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

1. Simple Reflex Agents

- They choose actions only based on the current percept.
- They are rational only if a correct decision is made only on the basis of current precept.
- Their environment is completely observable.

Condition-Action Rule − It is a rule that maps a state (condition) to an action.



2. Model Based Reflex Agents

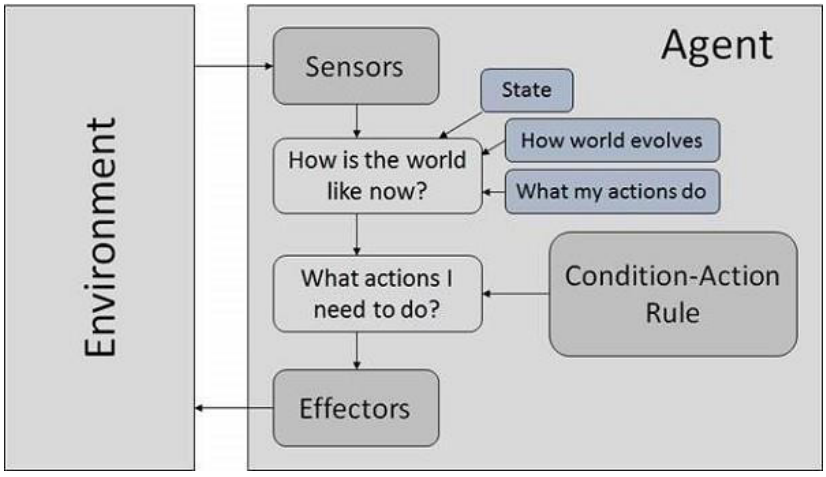
They use a model of the world to choose their actions. They maintain an internal state.

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.

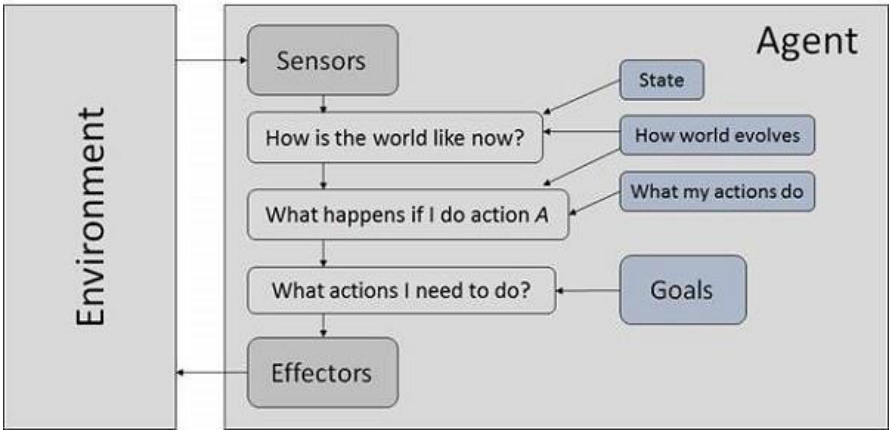


3. Goal Based Agents

They choose their actions in order to achieve goals.

Goal-based approach is more flexible than reflex agent since the knowledge supporting a decision is explicitly modeled, thereby allowing for modifications.

Goal − It is the description of desirable situations.

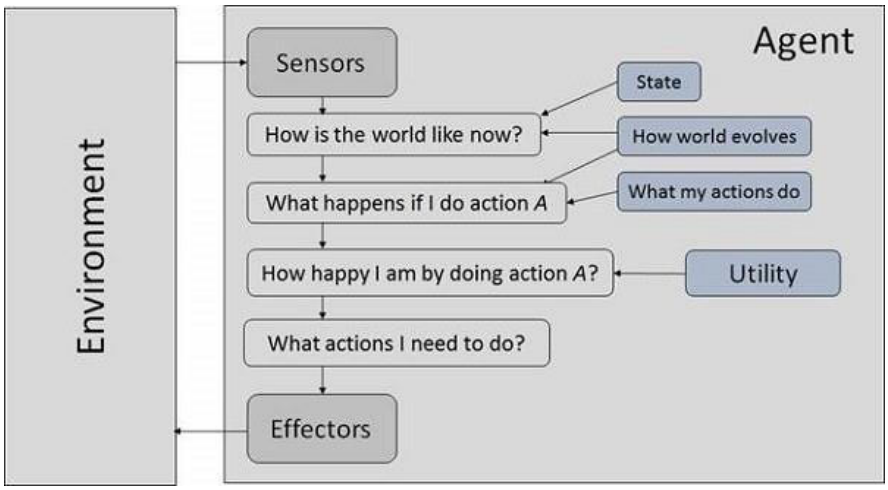


4. Utility Based Agents

They choose actions based on a preference (utility) for each state.

Goals are inadequate when −

- There are conflicting goals, out of which only few can be achieved.
- Goals have some uncertainty of being achieved and you need to weigh likelihood of success against the importance of a goal.



****

SOLVING PROBLEMS BY SEARCHING

In previous chapter, we saw that simple reflex agents are unable to plan ahead. They are limited in what they can do because their actions are determined only by the current percept. Furthermore, they have no knowledge of what their actions do nor of what they are trying to achieve.

In this chapter, we describe one kind of goal-based agent called a problem-solving agent. Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states. We discuss informally how the agent can formulate an appropriate view of the problem it faces. The problem type that results from the formulation process will depend on the knowledge available to the agent: principally, whether it knows the current state and the outcomes of actions. We then define more precisely the elements that constitute a "problem" and its "solution," and give several examples to illustrate these definitions. Given precise definitions of problems, it is relatively straightforward to construct a search process for finding solutions.

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. As we mentioned in previous chapter, the task is somewhat simplified if the agent can adopt a goal and aim to satisfy it. Let us first look at how and why an agent might do this.

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. We will discuss this process in more detail. For now, let us 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.***

Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do, and then removing that step from the sequence. Once the solution has been executed, the agent will find a new goal.

EXAMPLE PROBLEMS

The range of task environments that can be characterized by well-defined problems is vast. We can distinguish between so-called, toy problems, which are intended to illustrate or exercise various problem-solving methods, and so-called real-world problems, which tend to be more difficult and whose solutions people actually care about. In this section, we will give examples of both. By nature, toy problems can be given a concise, exact description. This means that they can be easily used by different researchers to compare the performance of algorithms. Real-world problems, on the other hand, tend not to have a single agreed-upon description, but we will attempt to give the general flavor of their formulations.

1. Toy Problems

The 8-puzzIe

The 8-puzzle, an instance of which is shown in Figure 3.4, 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. One important trick is to notice that rather than use operators such as "move the 3 tile into the blank space," it is more sensible to have operators such as "the blank space changes places with the tile to its left." This is because there are fewer of the latter kind of operator.

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 3.4.
- Path cost: each step costs 1, so the path cost is just the length of the path.

The 8-puzzle belongs to the family of sliding-block puzzles. This general class is known to be NP-complete, so one does not expect to find methods significantly better than the search algorithms described in this chapter and the next. The 8-puzzle and its larger cousin, the 15-puzzle, are the standard test problems for new search algorithms in Al.

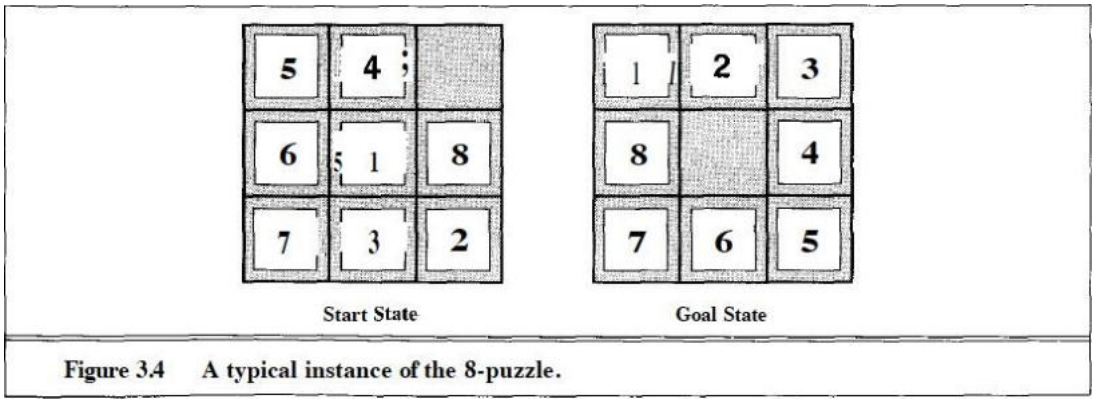


Figure 3.4 A typical instance of the 8-puzzle.

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.

The 8-queens problem can be defined as follows: Place 8 queens on an (8 by 8) chess board such that none of the queens attacks any of the others. A configuration of 8 queens on the board is shown in figure 1, but this does not represent a solution as the queen in the first column is on the same diagonal as the queen in the last column.

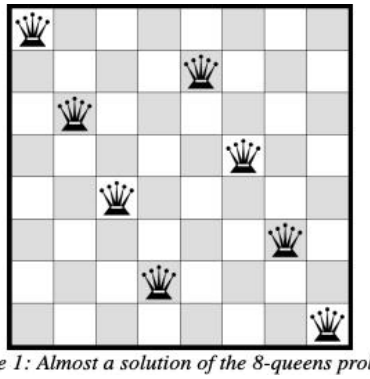


Figure 1: Almost a solution of the 8-queens problem

Although efficient special-purpose algorithms exist for this problem and the whole n queens family, it remains an interesting test problem for search algorithms. There are two main kinds of formulation. The incremental formulation involves placing queens one by one, whereas the complete-state formulation starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts; algorithms are thus compared only on search cost. Thus, we have the following goal test and path cost:

- Goal test: 8 queens on board, none attacked.
- Path cost: zero.

There are also different possible states and operators. Consider the following simple-minded formulation:

- States: any arrangement of 0 to 8 queens on board.
- Operators: add a queen to any square.

In this formulation, we have 648 possible sequences to investigate. A more sensible choice would use the fact that placing a queen where it is already attacked cannot work, because subsequent placings of other queens will not undo the attack. So we might try the following:

- States: arrangements of 0 to 8 queens with none attacked.
- Operators: place a queen in the left-most empty column such that it is not attacked by any other queen.

It is easy to see that the actions given can generate only states with no attacks; but sometimes no actions will be possible. For example, after making the first seven choices (left-to-right) in Figure 1, there is no action available in this formulation. The search process must try another choice. A quick calculation shows that there are only 2057 possible sequences to investigate. The right formulation makes a big difference to the size of the search space. Similar considerations apply for a complete-state formulation. For example, we could set the problem up as follows:

- States: arrangements of 8 queens, one in each column.
- Operators: move any attacked queen to another square in the same column.

This formulation would allow the algorithm to find a solution eventually, but it would be better to move to an unattacked square if possible.

2. Real-world problems

Route finding

We have already seen how route finding is defined in terms of specified locations and transitions! along links between them. 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. The last application is somewhat more complicated, because airline travel has a very complex path cost, in terms of money, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on. Furthermore, the actions in the problem do not have completely known outcomes: flights can be late or overbooked, connections can be missed, and fog or emergency maintenance can cause delays.

Touring and travelling salesperson problems

Consider the problem, "Visit every city in Figure 3.3 at least once, starting and ending in Bucharest." This seems very similar to route finding, because the operators still correspond to trips between adjacent cities. But for this problem, the state space must record more information. In addition to the agent's location, each state must keep track of the set of cities the agent has visited. So the initial state would be "In Bucharest; visited {Bucharest}," a typical intermediate state would be "In Vaslui; visited {Bucharest,Urziceni,Vaslui}," and the goal test would check if the agent is in Bucharest and that all 20 cities have been visited. 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.The problem is NP-hard (Karp, 1972), but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for travelling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit board drills.

VLSI Layout

The design of silicon chips is one of the most complex engineering design tasks currently undertaken, and we can give only a brief sketch here. 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. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire using the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Robot navigation

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a simple, circular robot moving on a flat

surface, the space is essentially two-dimensional. When the robot has arms and legs that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite.

SEARCHING FOR SOLUTIONS

We have seen how to define a problem, and how to recognize a solution. The remaining part— finding a solution—is done by a search through the state space. The idea is to maintain and extend a set of partial solution sequences. In this section, we show how to generate these sequences and how to keep track of them using suitable data structures.

Generating action sequences

To solve the route-finding problem from Arad to Bucharest, for example, we start off with just the initial state, Arad. The first step is to test if this is a goal state. Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad." Because this is ; not a goal state, we need to consider some other states. This is done by applying the operators; to the current state, thereby generating a new set of states. The process is called expanding the state. In this case, we get three new states, "in Sibiu," "in Timisoara," and "in Zerind," because there is a direct one-step route from Arad to these three cities. If there were only one possibility; we would just take it and continue. But whenever there are multiple possibilities, we must make a choice about which one to consider further.

This is the essence of search—choosing one option and putting the others aside for later, in ' case the first choice does not lead to a solution. Suppose we choose Zerind. We check to see if it is a goal state (it is not), and then expand it to get "in Arad" and "in Oradea." We can then choose any of these two, or go back and choose Sibiu or Timisoara. We continue choosing, testing, and expanding until a solution is found, or until there are no more states to be expanded. The choice of which state to expand first is determined by the search strategy.

It is helpful to think of the search process as building up a search tree that is superimposed over the state space. The root of the search tree is a search node corresponding to the initial state. The leaf nodes of the tree correspond to states that do not have successors in the tree, either because they have not been expanded yet, or because they were expanded, but generated the empty set. At each step, the search algorithm chooses one leaf node to expand. Figure 3.8 shows some of the expansions in the search tree for route finding from Arad to Bucharest. The general search algorithm is described informally in Figure 3.9.

It is important to distinguish between the state space and the search tree. For the route finding problem, there are only 20 states in the state space, one for each city. But there are an infinite number of paths in this state space, so the search tree has an infinite number of nodes. For example, in Figure 3.8, the branch Arad-Sibiu-Arad continues Arad-Sibiu-Arad-Sibiu-Arad, and so on, indefinitely. Obviously, a good search algorithm avoids following such paths.

Data structures for search trees

There are many ways to represent nodes, but in this chapter, 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

It is important to remember the distinction between nodes and states. A node is a bookkeeping data structure used to represent the search tree for a particular problem instance as generated by a particular algorithm. A state represents a configuration (or set of configurations) of the world. Thus, nodes have depths and parents, whereas states do not. (Furthermore, it is quite possible for two different nodes to contain the same state, if that state is generated via two different sequences of actions.) The EXPAND function is responsible for calculating each of the components of the nodes it generates.

SEARCH STRATEGIES

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.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

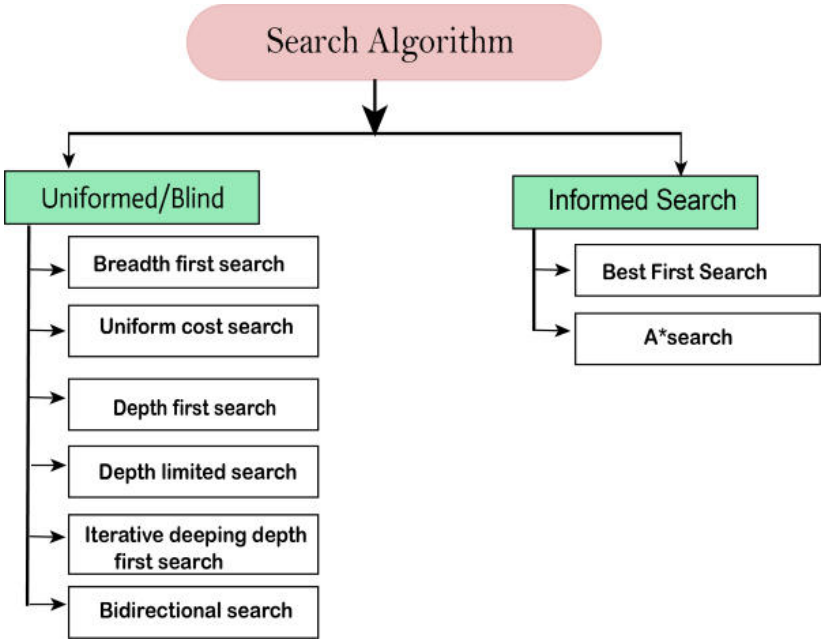
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



1. UNINFORMED SEARCH (Blind 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 applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

1. 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.

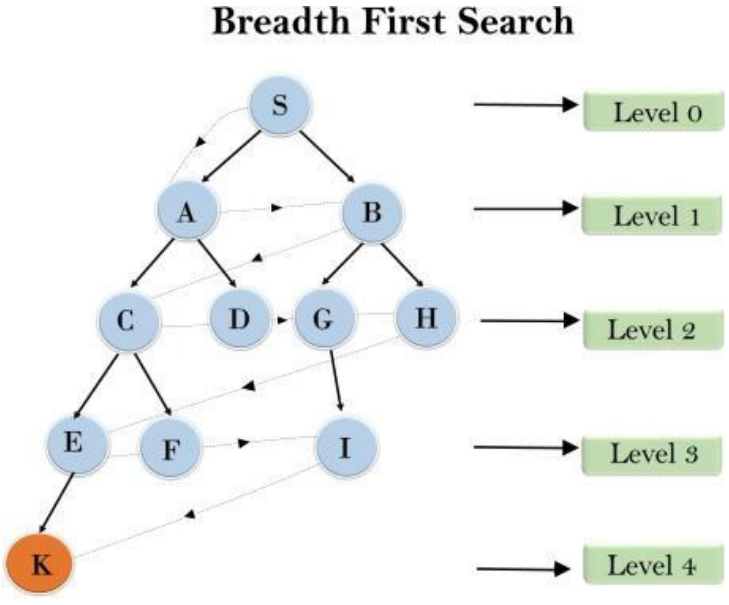
Disadvantages:

- t 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



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.

$T(b) = 1+b^2+b^3+ \quad + b^d = O(b^d)$

Space Complexity:
Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
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.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- 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.
- The process of the DFS algorithm is similar to the BFS algorithm.

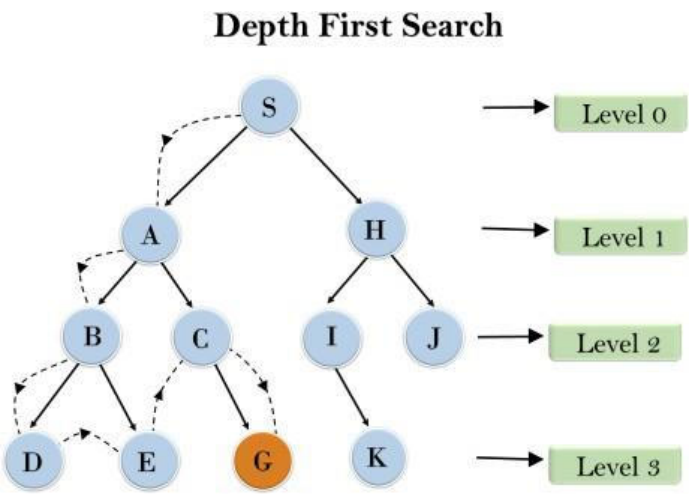
Advantage:

- 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.

Example
In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
Root node--->Left node---- > right node.



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:
$T(n)= 1+ n^2+ n^3 +..... + n^m=O(n^m)$
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.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
Depth-limited search can be terminated with two Conditions of failure:

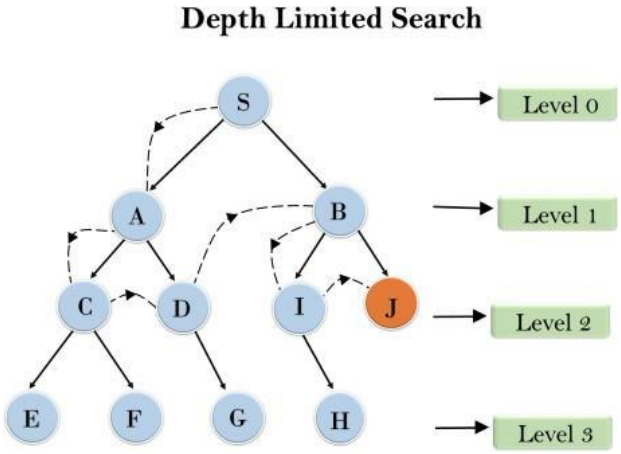
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

Advantages:
Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.
Time Complexity: Time complexity of DLS algorithm is **O(b^{ℓ})**.
Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)**.
Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

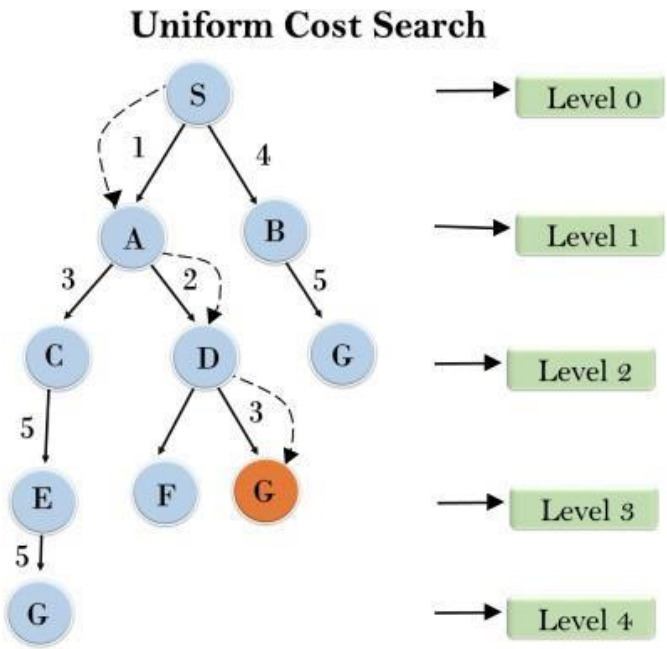
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.
-

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example



Completeness:
Uniform-cost search is complete, such as if there is a solution, UCS will find it.
Time Complexity:
Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$/.
Space Complexity:
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$.
Optimal:
Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

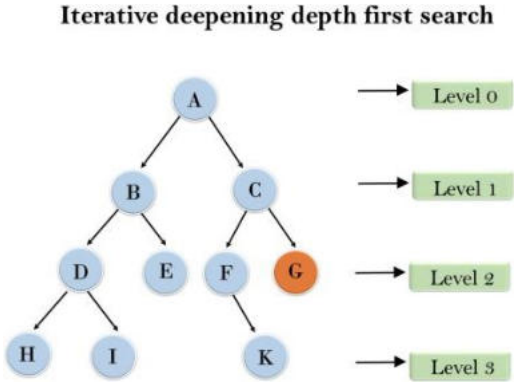
Advantages:

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

Disadvantages:

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

Example
Following tree structure is showing the iterative deepening depth-first search.
IDDFS algorithm performs various iterations until it does not find the goal node.
The iteration performed by the algorithm is given as:



1'st Iteration ----> A
2'nd Iteration ---> A, B, C
3'rd Iteration ----- >A, B, D, E, C, F, G
4'th Iteration ----- >A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

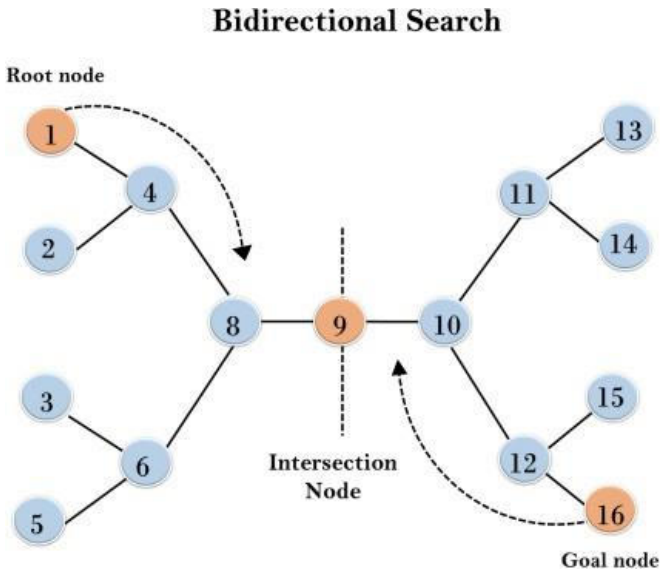
Completeness:
This algorithm is complete is if the branching factor is finite.
Time Complexity:
Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $\mathbf{O(b^d)}$.
Space Complexity:
The space complexity of IDDFS will be **O(bd)**.
Optimal:
IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm
Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.
Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:
- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:
- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example
In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.
Time Complexity: Time complexity of bidirectional search using BFS is $O(b^d)$.
Space Complexity: Space complexity of bidirectional search is $O(b^d)$.
Optimal: Bidirectional search is Optimal.

2. 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.

Heuristics function:
Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it 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:

1. $h(n) <= h^*(n)$

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.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1. Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

$f(n)= g(n)$.

Were, $h(n)$= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Advantages:

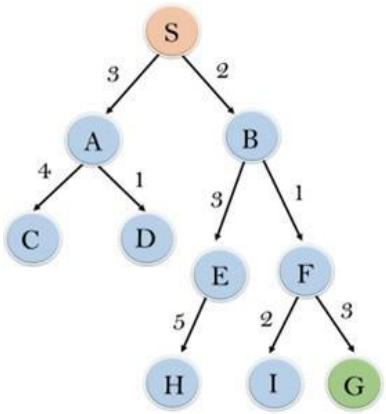
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

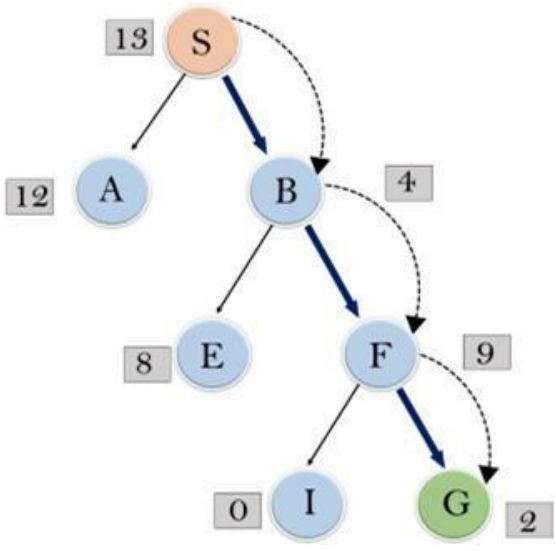
Example

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

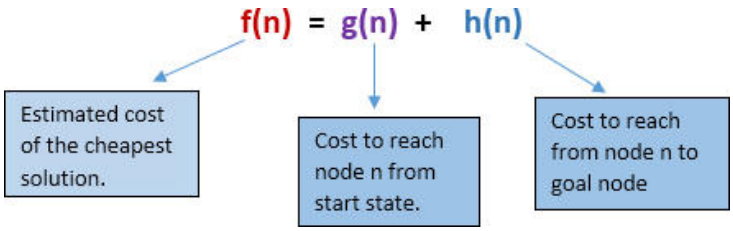
Hence the final solution path will be: **S----> B----->F --- > G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.
Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.
Optimal: Greedy best first search algorithm is not optimal.

2. A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Advantages:

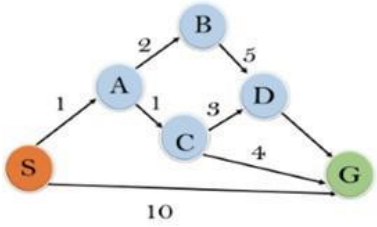
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

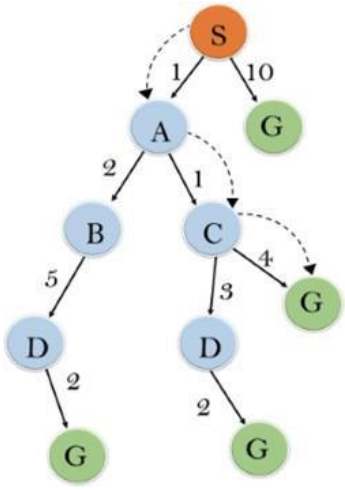
Example

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

****

UNIT-2

CONSTRAINT SATISFACTION PROBLEMS:

A Constraint Satisfaction Problem is a mathematical problem where the solution must meet a number of constraints. In CSP the objective is to assign values to variables such that all the constraints are satisfied. Many AI applications use CSPs to solve decision-making problems that involve managing or arranging resources under strict guidelines.

Common applications of CSPs include:

Scheduling: It assigns resources like employees or equipment while respecting time and availability constraints.

Planning: Organize tasks with specific deadlines or sequences.

Resource Allocation: Distributing resources efficiently without overuse

Components of Constraint Satisfaction Problems

CSPs are composed of three key elements:

1. Variables: These are the things we need to find values for. Each variable represents something that needs to be decided. For example, in a Sudoku puzzle each empty cell is a variable that needs a number. Variables can be of different types like yes/no choices (Boolean), whole numbers (integers) or categories like colors or names.

2. Domains: This is the set of possible values that a variable can have. The domain tells us what values we can choose for each variable. In Sudoku the domain for each cell is the numbers 1 to 9 because each cell must contain one of these numbers. Some domains are small and limited while others can be very large or even infinite.

3. Constraints: These are the rules that restrict how variables can be assigned values. Constraints define which combinations of values are allowed.

There are different types of constraints:

- Unary constraints apply to a single variable like "this cell cannot be 5".
- Binary constraints involve two variables like "these two cells cannot have the same number".
- Higher-order constraints involve three or more variables like "each row in Sudoku must have all numbers from 1 to 9 without repetition".

Types of Constraint Satisfaction Problems

CSPs can be classified into different types based on their constraints and problem characteristics:

Binary CSPs: In these problems each constraint involves only two variables. Like in a scheduling problem the constraint could specify that task A must be completed before task B.

Non-Binary CSPs: These problems have constraints that involve more than two variables. For instance in a seating arrangement problem a constraint could state that three people cannot sit next to each other.

Hard and Soft Constraints: Hard constraints must be strictly satisfied while soft constraints can be violated but at a certain cost. This is often used in real-world applications where not all constraints are equally important.

Representation of Constraint Satisfaction Problems (CSP)

In CSP it involves the interaction of variables, domains and constraints. Below is a structured representation of how CSP is formulated:

1. Finite Set of Variables (V_1, V2…V_n): The problem consists of a set of variables each of which needs to be assigned a value that satisfies the given constraints.
2. Non-Empty Domain for Each Variable (D1,D2…Dn): Each variable has a domain a set of possible values that it can take. For example, in a Sudoku puzzle the domain could be the numbers 1 to 9 for each cell.
3. Finite Set of Constraints (C1,C2…$C_{m)}$: Constraints restrict the possible values that variables can take. Each constraint defines a rule or relationship between variables.
4. Constraint Representation: Each constraint C_i is represented as a pair of (scope, relation) where:
    - Scope: The set of variables involved in the constraint.
    - Relation: A list of valid combinations of variable values that satisfy the constraint.

Example: Let's say you have two variables $V_1$and V_2.A possible constraint could be V1≠V2, which means the values assigned to these variables must not be equal. There:

Scope: The variables V_1 and V_2

Scope: The variables $V_1$is not equal to V_2

Some relations might include explicit combinations while others may rely on abstract relations that are tested for validity dynamically.

Solving Constraint Satisfaction Problems Efficiently:

CSP use various algorithms to explore and optimize the search space ensuring that solutions meet the specified constraints. Here's a breakdown of the most commonly used CSP algorithms:

1. Backtracking Algorithm

The backtracking algorithm is a depth-first search method used to systematically explore possible solutions in CSPs. It operates by assigning values to variables and backtracks if any assignment violates a constraint.

How it works:

- The algorithm selects a variable and assigns it a value.
- It recursively assigns values to subsequent variables.
- If a conflict arises i.e a variable cannot be assigned a valid value then algorithm backtracks to the previous variable and tries a different value.
- The process continues until either a valid solution is found or all possibilities have been exhausted.

This method is widely used due to its simplicity but can be inefficient for large problems with many variables.

2. Forward-Checking Algorithm

The forward-checking algorithm is an enhancement of the backtracking algorithm that aims to reduce the search space by applying local consistency checks.

How it works:

- For each unassigned variable the algorithm keeps track of remaining valid values.
- Once a variable is assigned a value local constraints are applied to neighboring variables and eliminate inconsistent values from their domains.
- If a neighbor has no valid values left after forward-checking the algorithm backtracks.

This method is more efficient than pure backtracking because it prevents some conflicts before they happen reducing unnecessary computations.

3. Constraint Propagation Algorithms

Constraint propagation algorithms further reduce the search space by enforcing local consistency across all variables.

How it works:

- Constraints are propagated between related variables.
- Inconsistent values are eliminated from variable domains by using information gained from other variables
- These algorithms filter the search space by making inferences and by remove values that would led to conflicts.

Constraint propagation is used along with other CSP methods like backtracking to make the search faster.

Example problem: Map coloring

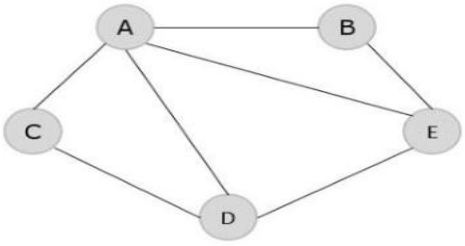
Map colouring problem states that given a graph G {V, E} where V and E are the set of vertices and edges of the graph, all vertices in V need to be coloured in such a way that no two adjacent vertices must have the same colour. The real-world applications of this algorithm are assigning mobile radio frequencies, making schedules, designing Sudoku, allocating registers etc.

Map Colouring Algorithm:

With the map colouring algorithm, a graph G and the colours to be added to the graph are taken as an input and a coloured graph with no two adjacent vertices having the same colour is achieved.
Algorithm:

- Initiate all the vertices in the graph.
- Select the node with the highest degree to colour it with any colour.
- Choose the colour to be used on the graph with the help of the selection colour function so that no adjacent vertex is having the same colour.
- Check if the colour can be added and if it does, add it to the solution set.
- Repeat the process from step 2 until the output set is ready.

EXAMPLE:



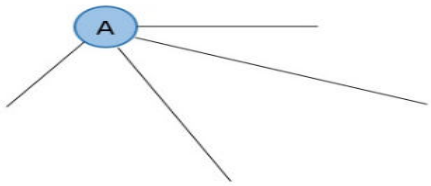
Step 1:

Find degrees of all the vertices –

A	4
B	2
C	2
D	3
E	3

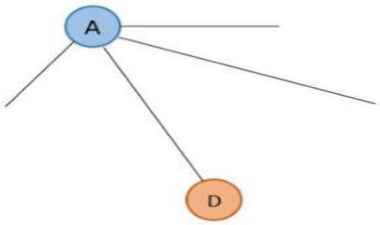
Step 2:

Choose the vertex with the highest degree to colour first, i.e., A and choose a colour using selection colour function. Check if the colour can be added to the vertex and if yes, add it to the solution set.



Step 3

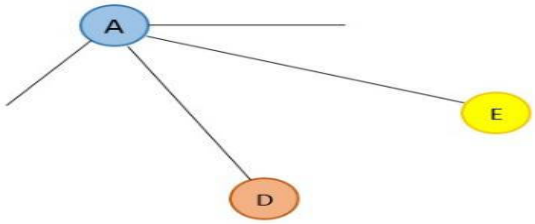
Select any vertex with the next highest degree from the remaining vertices and colour it using selection colour function.
D and E both have the next highest degree 3, so choose any one between them, say D.



D is adjacent to A, therefore it cannot be coloured in the same colour as A. Hence, choose a different colour using selection colour function.

Step 4:

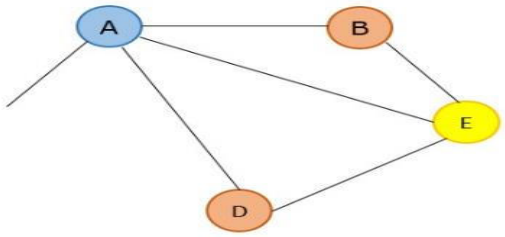
The next highest degree vertex is E, hence choose E.



E is adjacent to both A and D, therefore it cannot be coloured in the same colours as A and D. Choose a different colour using selection colour function.

Step 5:

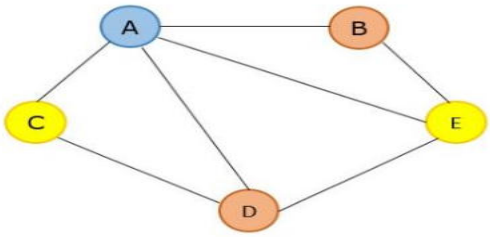
The next highest degree vertices are B and C. Thus, choose any one randomly.



B is adjacent to both A and E, thus not allowing to be coloured in the colours of A and E but it is not adjacent to D, so it can be coloured with Ds colour.

Step 6

The next and the last vertex remaining is C, which is adjacent to both A and D, not allowing it to be coloured using the colours of A and D. But it is not adjacent to E, so it can be coloured in Es colour.



Constraint Propagation: inference in CSPs

Constraint propagation is used in constraint satisfaction problems (CSPs) where the goal is to assign values to variables from a predefined domain while satisfying specific constraints. This process helps reduce the search space by narrowing down the possible values for each variable making the search for solutions more efficient. It is important in areas such as scheduling, planning and resource allocation.

Key Concepts in Constraint Propagation:
1. Variables: In constraint satisfaction problems (CSPs), variables are the elements that need to be assigned specific values. These variables represent the unknowns in a problem.
2. Domains: It refers to the set of all possible values that can be assigned to a variable. Each variable has its own domain and this helps define the range of potential solutions. By narrowing down the domain of variables using constraints, we make the search for solutions more efficient.
3. Constraints: They are the rules that describe how the variables are allowed to interact with each other. These rules specify what combinations of variable assignments are acceptable and which are not.

Working of Constraint Propagation:
Constraint propagation works by iteratively narrowing down the domains of variables based on the constraints. The process iteratively refines these domains based on the constraints, making it easier to find a valid solution. Essentially, it reduces the search space, enabling the solver to concentrate on optimal solutions.

Working of Constraint Propagation
1. Initialization: Begin with the initial domains of all variables. At the start, each variable is allowed to take any value from its respective domain without any constraints applied yet.
2. Propagation: Apply constraints to reduce the domains of variables. This means examining how the values of one variable affect the values of other variables based on the given constraints. For example, if two variables must have different values, the domain of one variable is updated to eliminate any values that conflict with the other.
3. Iteration: Repeat the propagation step until no further reductions can be made. When the process reaches a point where no more values can be eliminated from the domains, we say that the propagation has reached a "stable state."

Example:
Let's see a simple example to understand how constraint propagation works in action.
Consider two variables X and Y, each with the domain {1, 2, 3} and the constraint that $X \neq Y$ (i.e X and Y must not have the same value).
Step 1: Initially both X and Y have the domain {1, 2, 3}.
Step 2: If X is assigned the value 1, the constraint $X \neq Y$ implies that Y cannot be 1. Therefore, Y's domain is reduced to {2, 3}.
Step 3: Now, if Y is assigned the value 2, the constraint $X \neq Y$ tells us that X cannot be 2. So, X's domain is reduced to {1, 3}.
Step 4: This process continues with further assignments but once no more values can be eliminated, we have reached a stable state.

At this point, the domains of X and Y are reduced and the remaining possible values for each variable make it easier to find a valid solution that satisfies the constraints.
Algorithms for Constraint Propagation
There are several well-known algorithms used for constraint propagation. These algorithms help improve the efficiency of solving CSPs by simplifying the problem through various consistency checks.
Some common algorithms include:

1. Arc Consistency (AC-3): This algorithm ensures that for every value of one variable, there is a corresponding consistent value in another variable that satisfies the constraints between them. It's used as a preprocessing step to simplify CSPs before applying more complex algorithms.
2. Path Consistency: It extends arc consistency by considering triples of variables. It ensures that for every pair of variables, there is a consistent value in the third variable further narrowing down the domains and simplifying the problem.
3. k-Consistency: It generalizes arc and path consistency to include k variables. It ensures that for every subset of k-1 variables, there is a consistent value for the kth variable. This provides a greater reduction in domains but can be more computationally expensive.
4. Singleton Consistency: This method ensures that for each variable, the domain is reduced to only one possible value (a singleton). This is a more restrictive form of consistency and it's used when the problem has reached a point where each variable must have a unique value. It's useful in some specific applications like puzzle solving.
5. Generalized Arc Consistency (GAC): This is an extension of the arc consistency algorithm. It ensures that for every pair of variables connected by a constraint, every value in the domain of one variable must have a compatible value in the domain of the other variable.

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] - {Xj} do
        add (Xk, Xi) to queue

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

Solving Sudoku with Constraint Satisfaction Problem (CSP) Algorithms:

Step 1: Define the Problem (Sudoku Puzzle Setup)
The first step is to define the Sudoku puzzle as a 9x9 grid where 0 represents an empty cell. We also define a function print_ Sudoku to display the puzzle in a human readable format.

```
puzzle = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
          [6, 0, 0, 1, 9, 5, 0, 0, 0],
          [0, 9, 8, 0, 0, 0, 0, 6, 0],
          [8, 0, 0, 0, 6, 0, 0, 0, 3],
          [4, 0, 0, 8, 0, 3, 0, 0, 1],
          [7, 0, 0, 0, 2, 0, 0, 0, 6],
          [0, 6, 0, 0, 0, 0, 2, 8, 0],
          [0, 0, 0, 4, 1, 9, 0, 0, 5],
          [0, 0, 0, 0, 8, 0, 0, 7, 9]]


def print_sudoku(puzzle):
    for i in range(9):
        if i % 3 == 0 and i != 0:
            print("- - - - - - - - - - - ")
```

```
        for j in range(9):
            if j % 3 == 0 and j != 0:
                print(" | ", end="")
            print(puzzle[i][j], end=" ")
        print()

print("Initial Sudoku Puzzle:\n")
print_sudoku(puzzle)
```

OUTPUT:

```
Initial Sudoku Puzzle:

5 3 0  | 0 7 0  | 0 0 0
6 0 0  | 1 9 5  | 0 0 0
0 9 8  | 0 0 0  | 0 6 0
- - - - - - - - - - - -
8 0 0  | 0 6 0  | 0 0 3
4 0 0  | 8 0 3  | 0 0 1
7 0 0  | 0 2 0  | 0 0 6
- - - - - - - - - - - -
0 6 0  | 0 0 0  | 2 8 0
0 0 0  | 4 1 9  | 0 0 5
0 0 0  | 0 8 0  | 0 7 9
```

Step 2: Create the CSP Solver Class
We define a class CSP to handle the logic of the CSP algorithm. This includes functions for selecting variables, assigning values and checking consistency between variables and constraints.

Step 3: Implement Helper Functions for Backtracking
We add helper methods for selecting unassigned variables, ordering domain values and checking consistency with constraints. These methods ensure that the backtracking algorithm is efficient.

Step 4: Define Variables, Domains and Constraints
Next we define the set of variables, their possible domains and the constraints for the Sudoku puzzle. Variables represent the cells and domains represent possible values. Constraints should ensure that each number only appears once per row, column and 3x3 subgrid.

Step 5: Solve the Sudoku Puzzle Using CSP
We create an instance of CSP class and call the solve method to find the solution to the Sudoku puzzle. The final puzzle with the solution is then printed.

```
csp = CSP(variables, domains, constraints)
sol = csp.solve()
solution = [[0 for _ in range(9)] for _ in range(9)]
for (i, j), val in sol.items():
    solution[i][j] = val
print("\n******* Solution *******\n")
print_sudoku(solution)
```

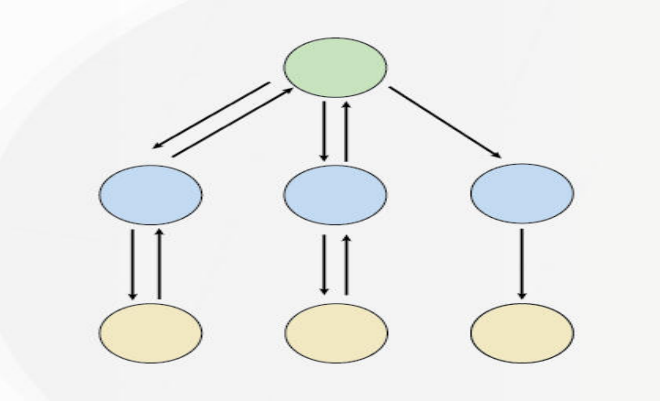
OUTPUT:

```
******* Solution *******

5 3 4   | 6 7 8   | 9 1 2
6 7 2   | 1 9 5   | 3 4 8
1 9 8   | 3 4 2   | 5 6 7
- - - - - - - - - - - -
8 5 9   | 7 6 1   | 4 2 3
4 2 6   | 8 5 3   | 7 9 1
7 1 3   | 9 2 4   | 8 5 6
- - - - - - - - - - - -
9 6 1   | 5 3 7   | 2 8 4
2 8 7   | 4 1 9   | 6 3 5
3 4 5   | 2 8 6   | 1 7 9
```

Output

Backtracking Search for CSPs:

Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they backtrack and try another until they find the right one. It's like solving a puzzle by testing different pieces until they fit together perfectly.



Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.

It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

Working of Backtracking Algorithm:
A backtracking algorithm works by recursively exploring all possible solutions to a problem. It starts by choosing an initial solution, and then it explores all possible extensions of that solution. If an extension leads to a solution, the algorithm returns that solution. If an extension does not lead to a solution, the algorithm backtracks to the previous solution and tries a different extension.
The following is a general outline of how a backtracking algorithm works:

1. Choose an initial solution.
2. Explore all possible extensions of the current solution.
3. If an extension leads to a solution, return that solution.
4. If an extension does not lead to a solution, backtrack to the previous solution and try a different extension.
5. Repeat steps 2-4 until all possible solutions have been explored.

Example of Backtracking Algorithm:

Example: Finding the shortest path through a maze

Input: A maze represented as a 2D array, where 0 represents an open space and 1 represents a wall.

Algorithm:

Start at the starting point.
For each of the four possible directions (up, down, left, right), try moving in that direction.
If moving in that direction leads to the ending point, return the path taken.
If moving in that direction does not lead to the ending point, backtrack to the previous position and try a different direction.
Repeat steps 2-4 until the ending point is reached or all possible paths have been explored.

```
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.

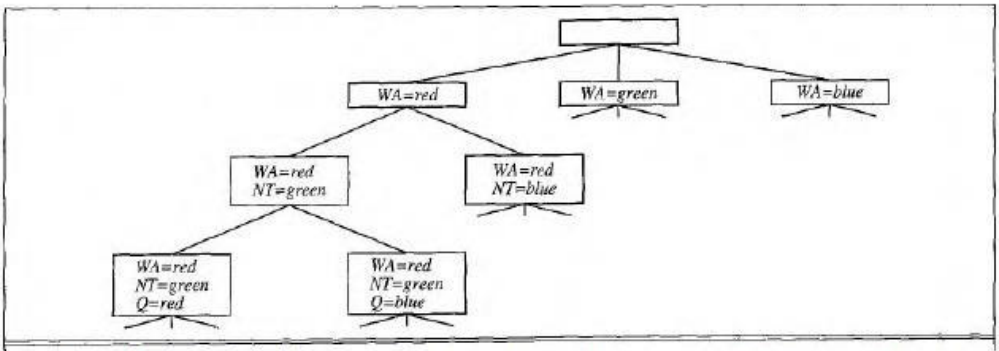


Figure 5.4 Part of the search tree generated by simple backtracking for the map-coloring problem in Figure 5.1.

Use of Backtracking Algorithm:

Backtracking algorithms are best used to solve problems that have the following characteristics:

- There are multiple possible solutions to the problem.
- The problem can be broken down into smaller sub problems.
- The sub problems can be solved independently.

Backtracking algorithms are used in a wide variety of applications, including:

- Solving puzzles (e.g., Sudoku, crossword puzzles)
- Finding the shortest path through a maze
- Scheduling problems
- Resource allocation problems
- Network optimization problems
- Combinatorial problems, such as generating permutations, combinations, or subsets.

Local Search for Search for CSPs:

Local search algorithms are important in artificial intelligence as they can quickly find good answers, especially when finding the perfect solution would take too long or too much effort. They are useful for big or complex problems where checking every possible option isn't practical.

- It focus only on the current solution and the ones directly related to it rather than looking everywhere.
- Ideal for real-world tasks like puzzles, timetables or route finding.

Working of Local Search Algorithms

Step 1: Pick a starting point: Start with a possible solution which is often random but sometimes based on rule.

- **Step 2:** Find the neighbors:
- Neighbors are similar solutions we can get by making small, simple changes to the current one.
- For example, in a puzzle, swapping two pieces creates a neighbor.

Step 3: Compare: Look around at all neighbors to see if any are better.

Step 4: Move: If a better neighbor exists, move to it, making it our new "current" solution.
Step 5: Repeat: Keep searching from the new point, following the same steps.
Step 6: Stop: When none of the neighbors are better or after enough tries.

For example: In the 8-queens problem, the initial state might be a random configuration of 8 queens in 8 columns, and the successor function picks one queen and considers moving it Elsewhere in its column. Another possibility would be start with the 8 queens, one per column In a permutation of the 8 rows, and to generate a successor by having two queens swap rows.

```
function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max-steps, the number of steps allowed before giving up

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

Figure 5.8 The *MIN-CONFLICTS* algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The *CONFLICTS* function counts the number of constraints violated by a particular value, given the rest of the current assignment.

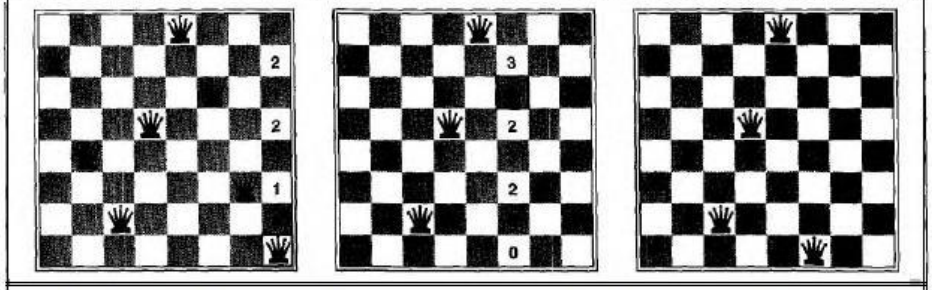


Figure 5.9 A two-step solution for an 8-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its colunnn. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflict square, breaking ties randomly.

Game Theory:

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

2- Person Games:

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

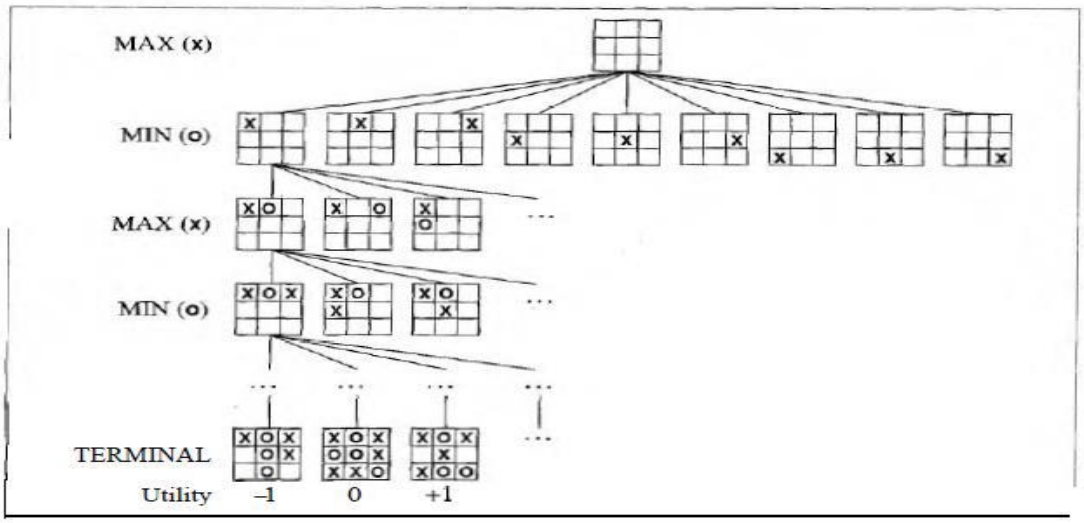
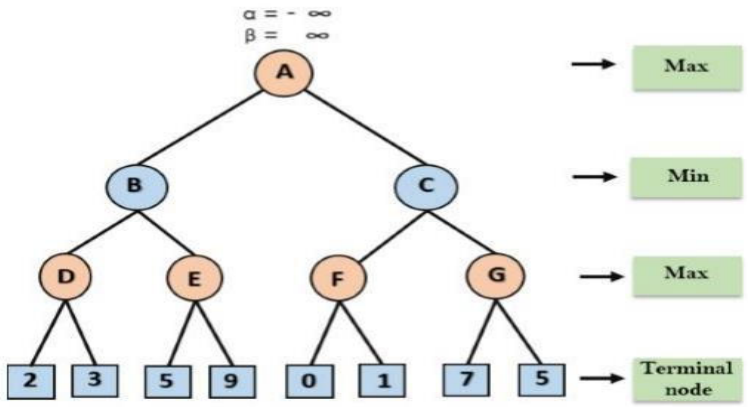


Figure 6.1 A (partial) search tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Optimal Decisions in Games:

An optimal decision in a game is the strategy or move that yields the best possible outcome for a player, assuming both players are playing rationally and trying to maximize their own results, as determined by game theory. This involves using mathematical and conceptual techniques to analyze potential actions, predict opponents' moves, and apply algorithms like the Minimax algorithm (often with alpha-beta pruning) to navigate the game's state and determine the most advantageous path to victory.



MiniMax Algorithm:

1. Generate the whole game tree.
2. Apply the utility function to leaf nodes to get their values.
3. Usetheutilityofnodesatleveltoderivetheutilityofnodesatleveln-1.
4. Continue backing up values towards the root (one layer at a time).
5. Eventually the backed up values reach the top of the tree, at which point Max chooses the move that yield the highest value. This is called the minimax decision because it maximize the utility

for Max on the assumption that Min will play perfectly to minimize it.

Algorithm: MINIMAX (Depth-First Version)

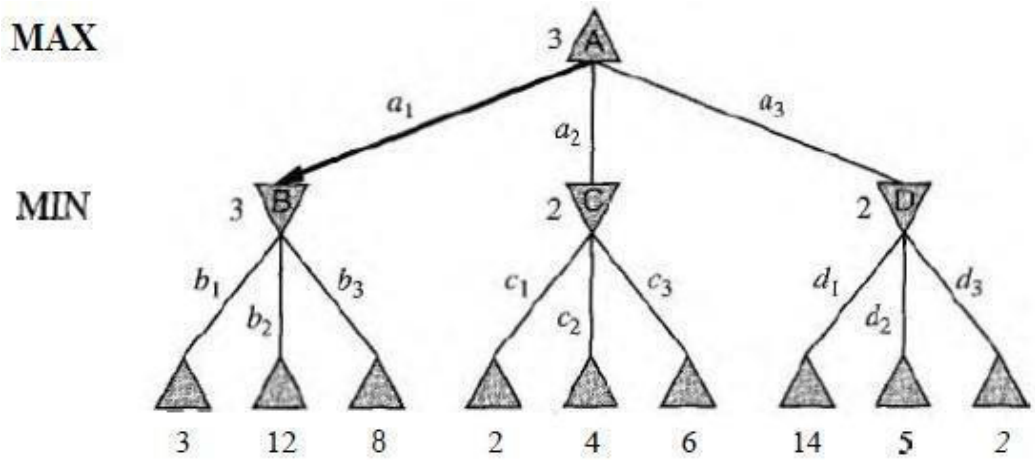
To determine the minimax value $V(J)$, do the following:

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

6.

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

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

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

Example:



Properties of Minimax:

- Complete : Yes(if tree is finite)
- Optimal : Yes(against tan optimal opponent)
- Time complexity : $O(b^m)$
- Space complexity : O(bm)(depth-first exploration)
- Forchess,b≈35,m≈100for"reasonable"games

  →Exact solution completely infeasible.

Limitations:

- Not always feasible to traverse entire tree
- Time limitations

Alpha-Beta pruning Algorithm:

- **Pruning:** eliminating a branch of these arch tree from consideration without exhaustive examination of each node
- α-β**Pruning:** the basic idealist prune portions of these arch tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.
- *Alpha-beta pruning* is used on top of minimax search to detect paths that do not need to be Explored. The intuition is:
- The MAX player is always trying to maximize the score. Call this α.
- The MIN player is always trying to minimize the score. Call this β.
- **Alpha cut off**: Given a Max node n, cutoff the search below (i.e., don't generate or examine any more of n's children) if alpha(n) >=beta(n)

  (Alpha increases and passes beta from below)
- **Beta cutoff.:** Given a Min node n, cutoff the search below (i.e., don't generate or examine any more of n's children) if beta(n) <=alpha(n)

  (Beta decrease and passes alpha from above)

- CarryalphaandbetavaluesdownduringsearchPruningoccurswheneveralpha>=beta

Algorithm:

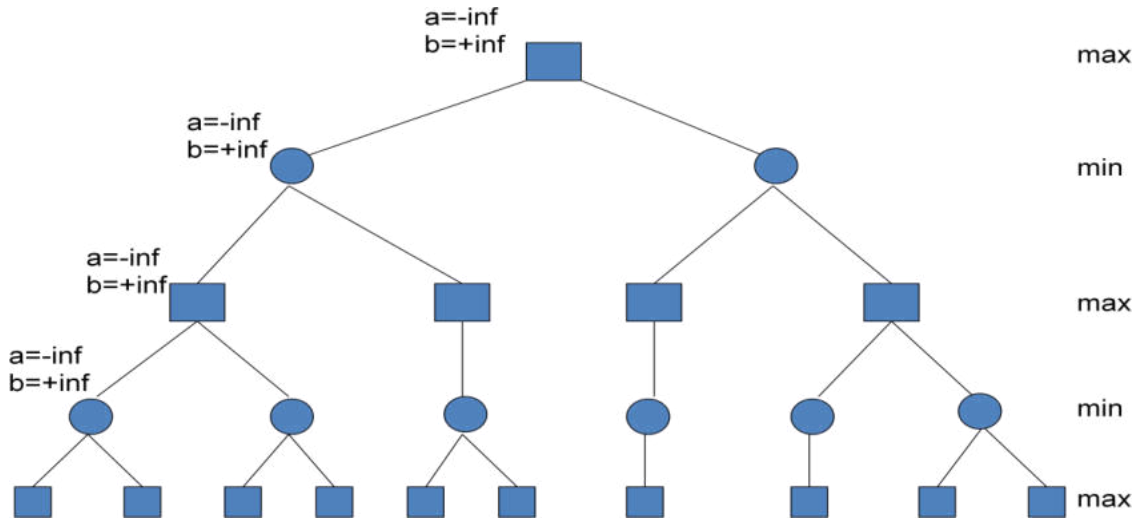
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

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

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

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

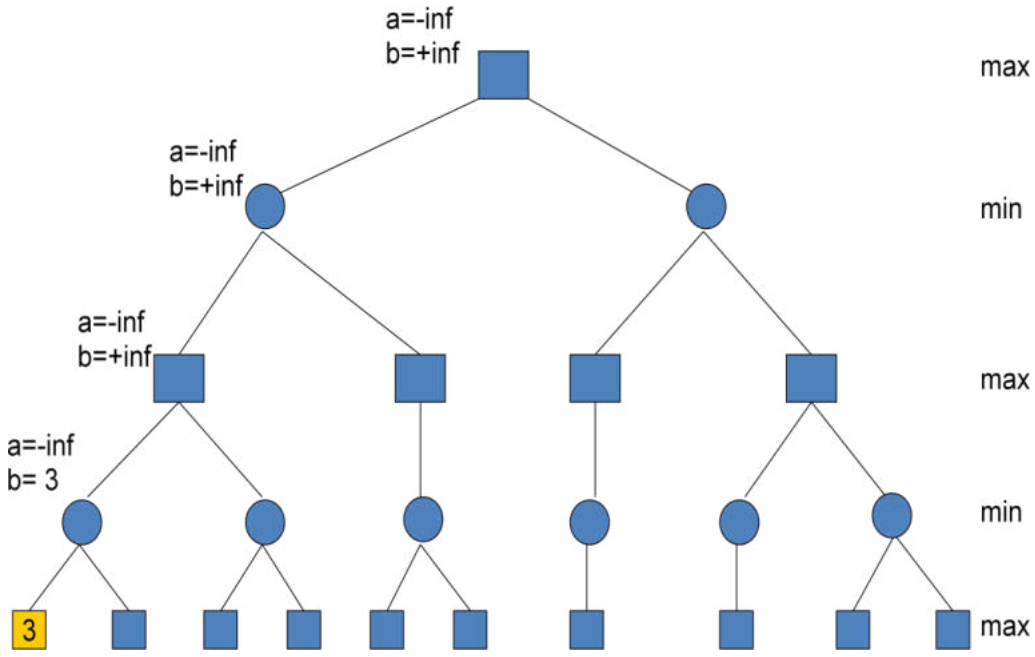
```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

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

Example:

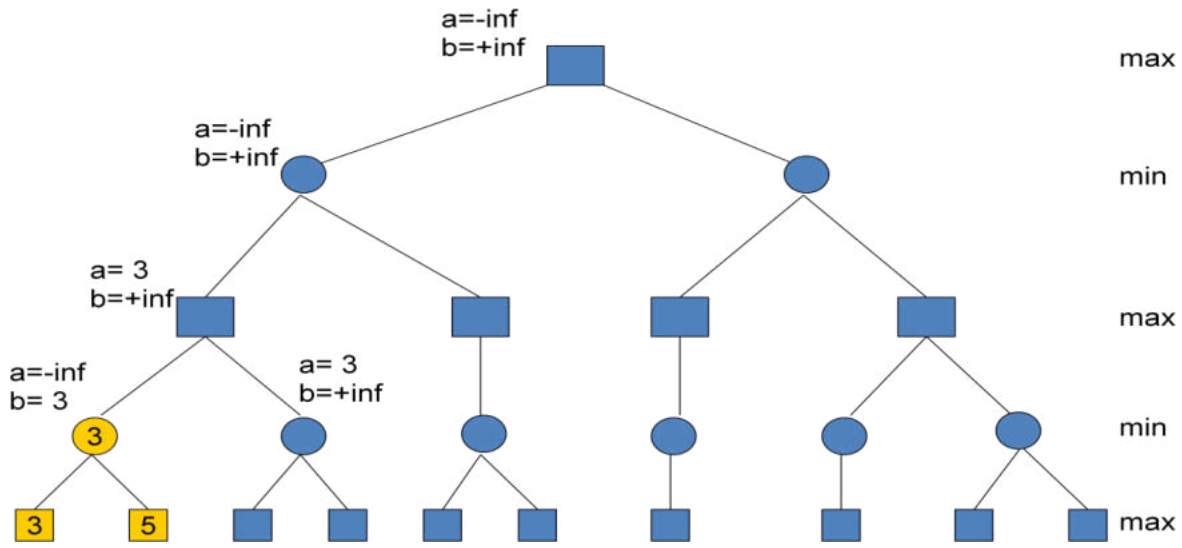
1) **Setup phase: Assign to each left-most(or right-most)internal node of the tree, variables: alpha=-infinity, beta= +infinity**



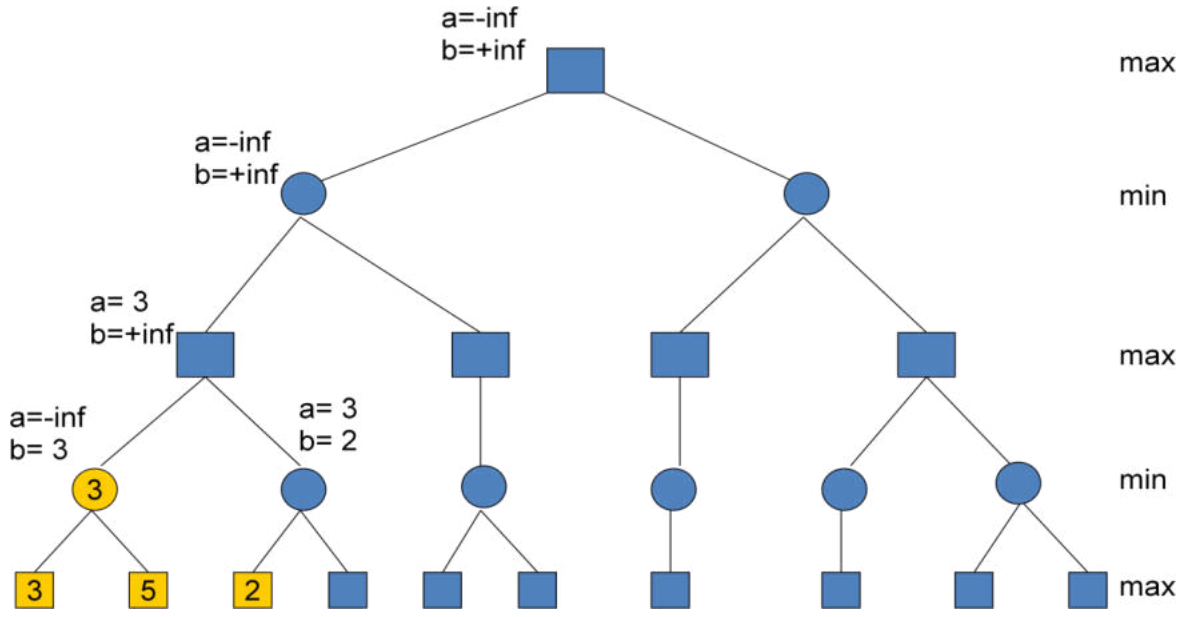
2) **Look at first computed final configuration value. It'sa3. Parent is a min node, so set the beta (min)value to 3.**



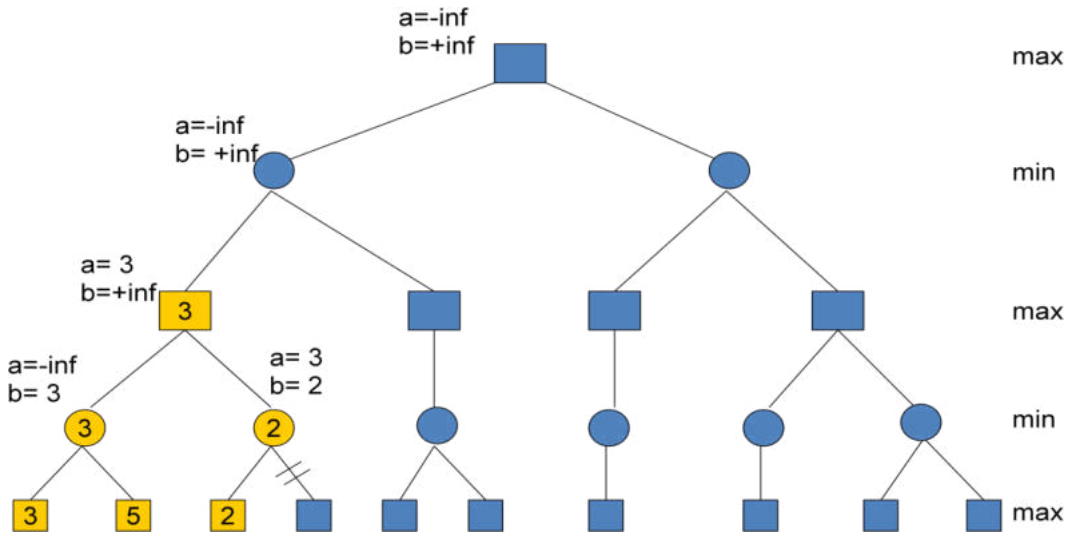
3) Look at next value, 5.Since parent is a min node, we want the minimum of 3 and 5 which is3.Parent min node is done– fill alpha (max) value of its parent max node. Always set alpha for max nodes and beta for min nodes. Copy the state of the max parent node in to the second unevaluated min child.



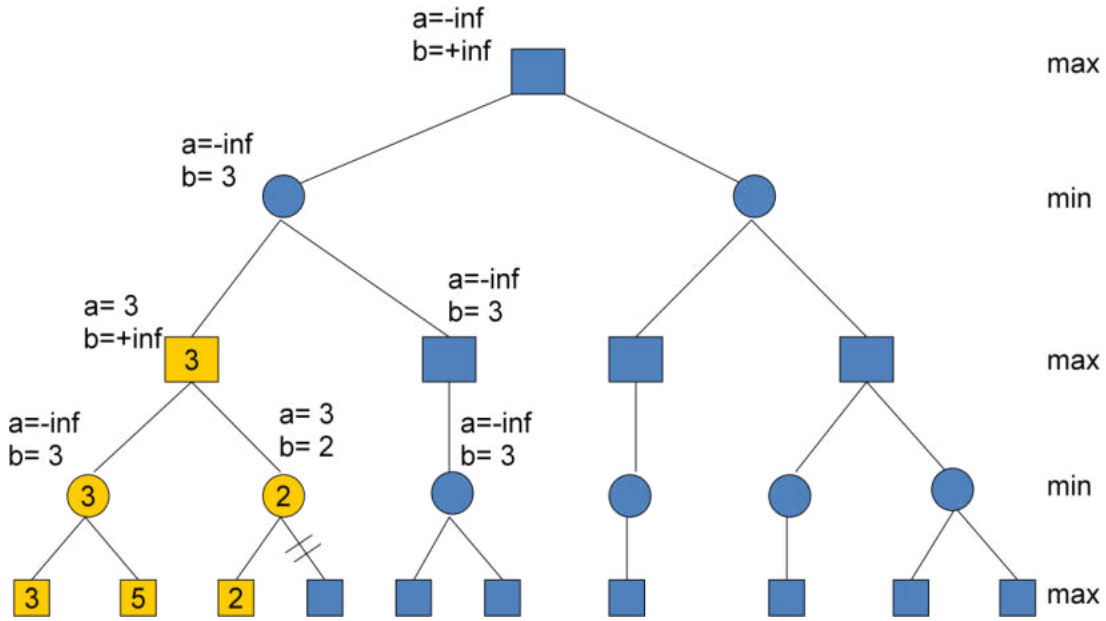
4) Look at next value, 2.Since parent node is min with b=+in f,2 is smaller, change b.



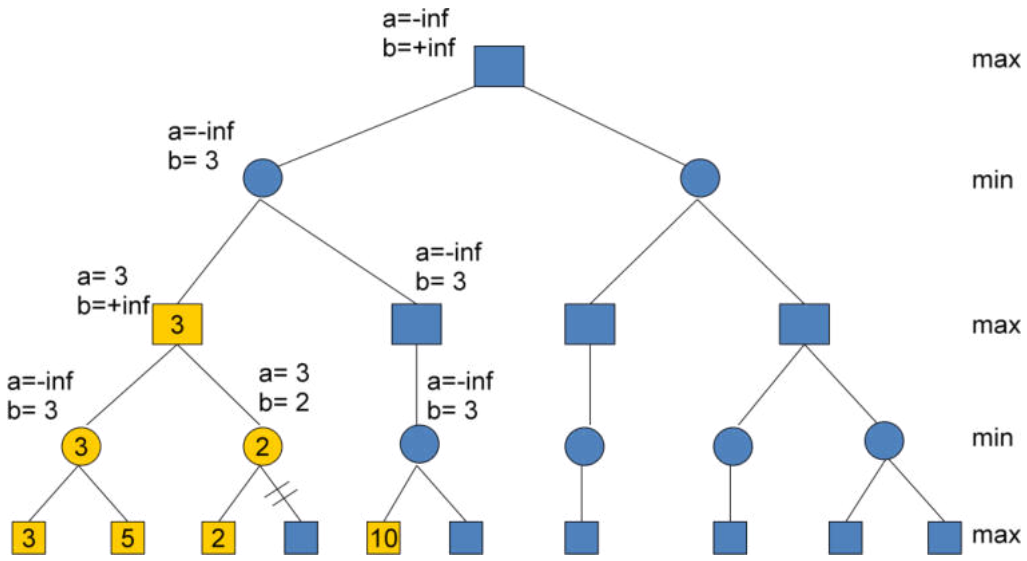
5) Now, the min parent node has a max value of 3 and min value of 2.The value of the 2nd child does not matter. If it is >2, 2 will be selected for min node. If it is <2, it will be selected for min node, but since it is<3 it will not get selected for the parent max node. Thus, we prune the right sub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



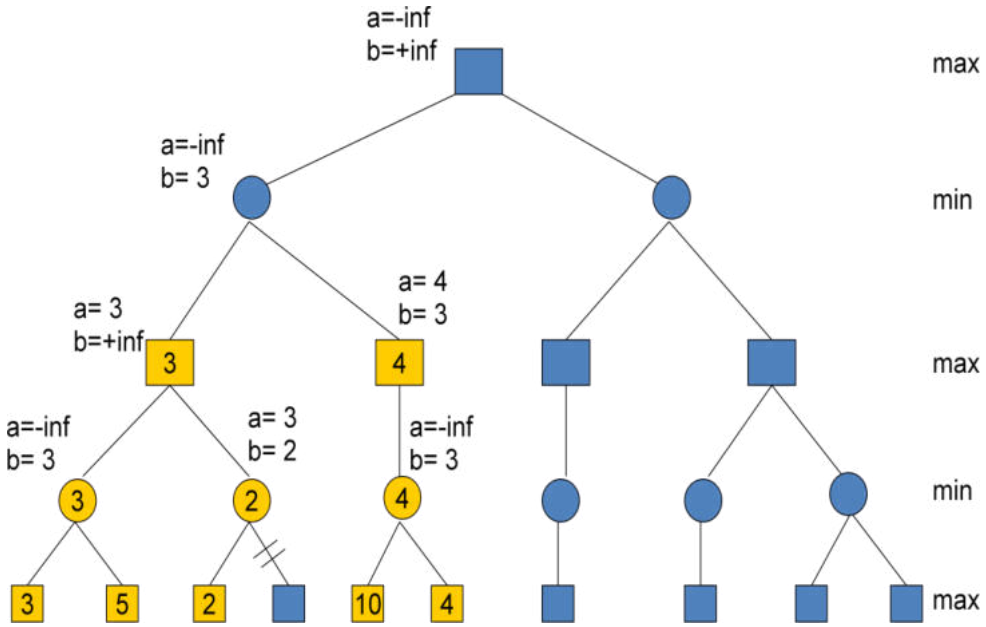
6) Max node is now done and we can set the bet a value of its parent and propagate node state to sibling sub tree's left-most path.



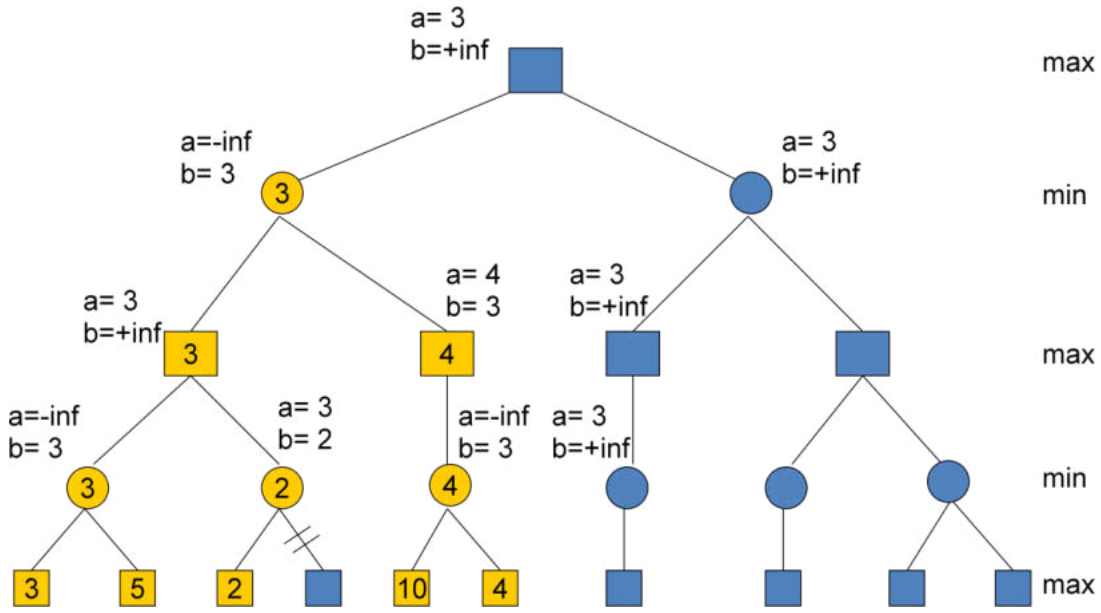
7) The next node is 10. 10 is not smaller than 3, so state of parent does not change. We still have to look at the 2nd child since alpha is still –inf.



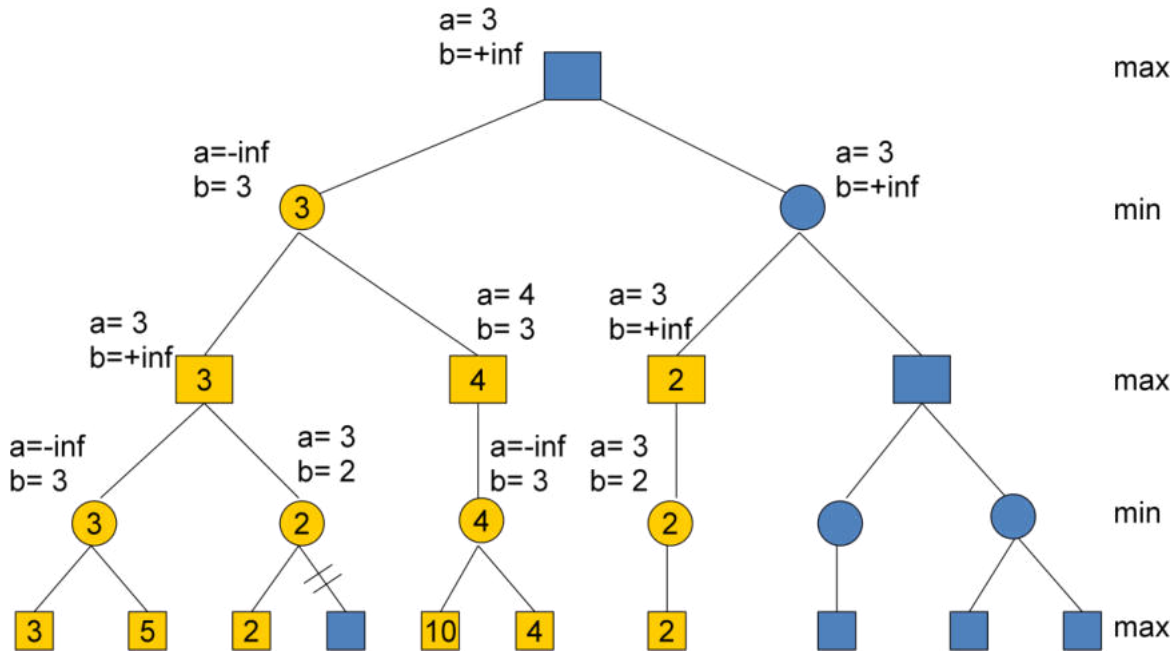
8) The next node is 4. Smallest value goes to the parent min node. Min sub tree is done, so the parent max node gets the alpha (max) value from the child. Note that if the max node had a 2nd sub tree, we can prune its in ce a>b.



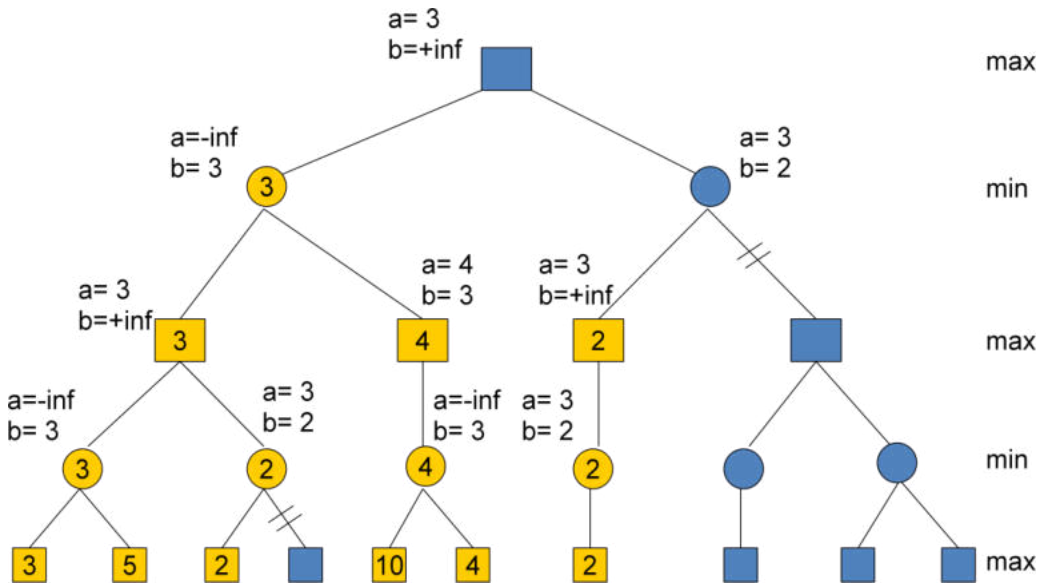
9) Continue propagating value up the tree, modifying the corresponding alpha/beta values. Also propagate the state of root node down the left-most path of the right sub tree.



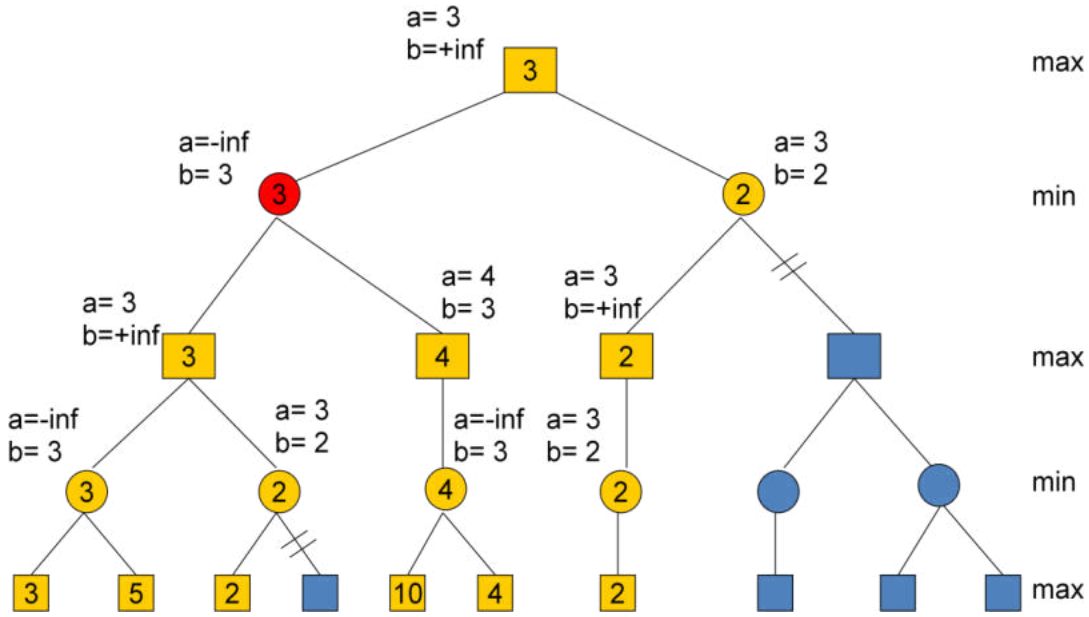
10) Next value is a 2.Weset the beta (min) value of the min parentto2.Since no other children exist, we propagate the value up the tree.



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



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



Knowledge-Based System:

Human's claim that how intelligence is achieved- not by purely reflect mechanisms but by process of reasoning that operate on internal representation of knowledge. In AI these techniques for intelligence are present in Knowledge Based Agents.

- A knowledge-based system is a system that uses artificial intelligence techniques to store and reason with knowledge. The knowledge is typically represented in the form of rules or facts, which can be used to draw conclusions or make decisions.
- One of the key benefits of a knowledge-based system is that it can help to automate decision-making processes. For example, a knowledge-based system could be used to diagnose a medical condition, by reasoning over a set of rules that describe the symptoms and possible causes of the condition.
- Another benefit of knowledge-based systems is that they can be used to explain their decisions to humans. This can be useful, for example, in a customer service setting, where a knowledge-based system can help a human agent understand why a particular decision was made.
- Knowledge-based systems are a type of artificial intelligence and have been used in a variety of applications including medical diagnosis, expert systems, and decision support systems.

Various levels of knowledge-based agents

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. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

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. At the logical level we can expect to the automated taxi agent to reach to the destination B.

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. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Actions performed by an agent:

Inference System is used when we want to update some information (sentences) in Knowledge-Based System and to know the already present information. This mechanism is done by TELL and ASK operations. They include inference i.e. producing new sentences from old. Inference must accept needs when one asks a question to KB and answer should follow from what has been Told to KB. Agent also has a KB, which initially has some background Knowledge. Whenever, agent program is called, it performs some actions.

Actions done by KB Agent:

1. It TELLS what it recognized from the environment and what it needs to know to the knowledge base.
2. It ASKS what actions to do? and gets answers from the knowledge base.
3. It TELLS the which action is selected , then agent will execute that action.

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

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

Figure 7.1 A generic knowledge-based agent.

The wumpus world:

The Wumpus World is a classic example of a knowledge-based agent in AI. It involves reasoning, knowledge representation and planning. In this agent uses its knowledge of the world to make decisions and navigate safely in the given environment. In this article, we will learn more about it.

Understanding Wumpus World Problem:

Wumpus World is a 4x4 grid consisting of 16 rooms. Agent starts at Room[1,1] facing right and its goal is to retrieve treasure while avoiding hazards such as pits and the Wumpus. Agent must navigate through grid using its limited sensory input to make decisions that will keep it safe and allow it to successfully collect the treasure and exit the cave.

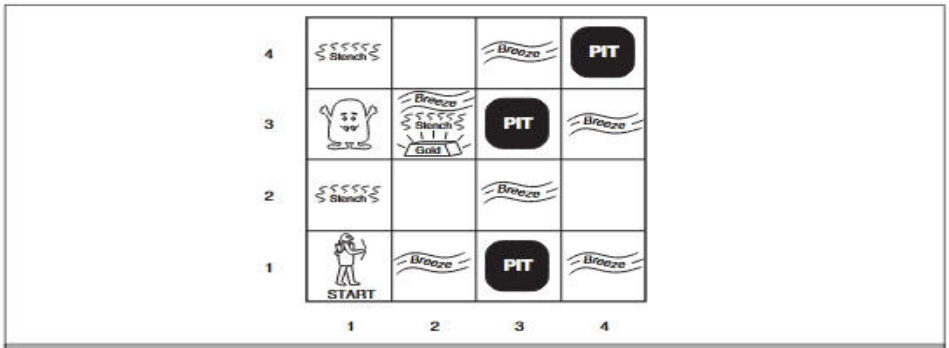


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

Key Elements:

Pits: If the agent steps into a pit it falls and dies. A breeze in adjacent rooms suggests nearby pits.

Wumpus: A creature that kills agent if it enters its room. Rooms next to the Wumpus have a stench. Agent can use an arrow to kill the Wumpus.

Treasure: Agent's main objective is to collect the treasure (gold) which is located in one room.

Breeze: Indicates a pit is nearby.

Stench: Indicates the Wumpus is nearby.

Agent must navigate carefully avoiding dangers to collect treasure and exit safely.

PEAS Description

PEAS stands for Performance Measures, Environment, Actuators and Sensors which describe agent's capabilities and environment.

1. Performance measures: Rewards or Punishments

- Agent gets gold and return back safe = +1000 points
- Agent dies (pit or Wumpus)= -1000 points
- Each move of the agent = -1 point
- Agent uses the arrow = -10 points

2. Environment: A setting where everything will take place.

- A cave with 16(4x4) rooms.
- Rooms adjacent (not diagonally) to the Wumpus are stinking.
- Rooms adjacent (not diagonally) to the pit are breezy.
- Room with gold glitters.
- Agent's initial position - Room[1, 1] and facing right side.
- Location of Wumpus, gold and 3 pits can be anywhere except in Room[1, 1].

3. Actuators: Devices that allow agent to perform following actions in the environment.

- Move forward: Move to next room.
- Turn right/left: Rotate agent 90 degrees.
- Shoot: Kill Wumpus with arrow.
- Grab: Take treasure.
- Release: Drop treasure

4. Sensors: Devices help the agent in sensing following from the environment.

- Breeze: Detected near a pit.
- Stench: Detected near the Wumpus.
- Glitter: Detected when treasure is in the room.
- Scream: Triggered when Wumpus is killed.
- Bump: Occurs when hitting a wall.

How the Agent Operates with PEAS

Perception: Agent uses sensory inputs (breeze, stench, glitter) to detect its surroundings and understand the environment.

Inference: Agent applies logical reasoning to find location of hazards. For example if it detects a breeze, it warns that a pit is nearby or if there's a stench it suspects the Wumpus is in an adjacent room.

Planning: Based on its deductions agent plans its next move avoiding risky areas like rooms with suspected pits or the Wumpus.

Action: Agent performs planned action such as moving to a new room, shooting arrow at the Wumpus or taking the treasure.

This process repeats till the agent finds the cave using its sensory inputs, reasoning and planning to achieve its goal safely.

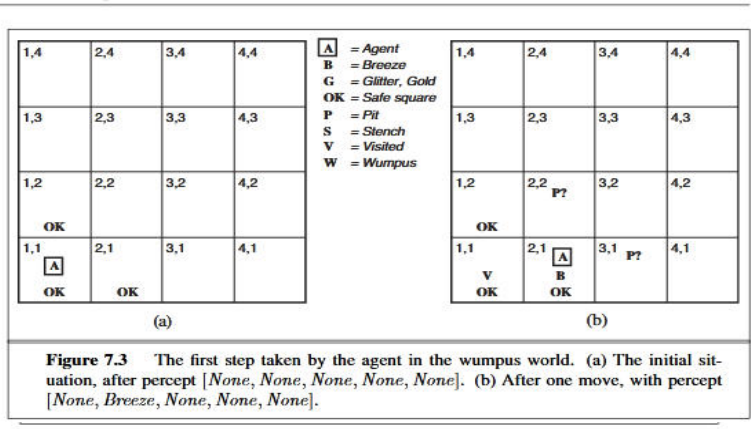


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

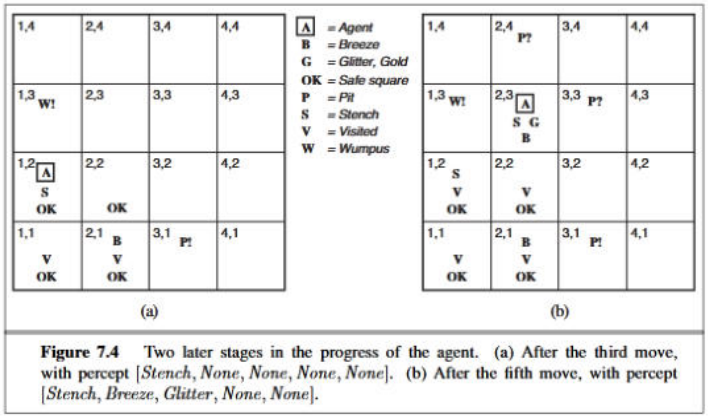


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

LOGIC:

The fundamental concepts of logical representation and reasoning.These beautiful ideas are independent of any of logic's particular forms. we said that knowledge bases consist of sentences. These sentences are expressed according to the syntax of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "x + y = 4" is a well-formed sentence, whereas "x4y+ =" is not.

Logic must also define the semantics or meaning of sentences. The semantics defines the truth of each sentence with respect to each possible world. For example, the semantics for arithmetic specifies that the sentence "x + y = 4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between." 1When we need to be precise, we use the term model in place of "possible world." whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentence.

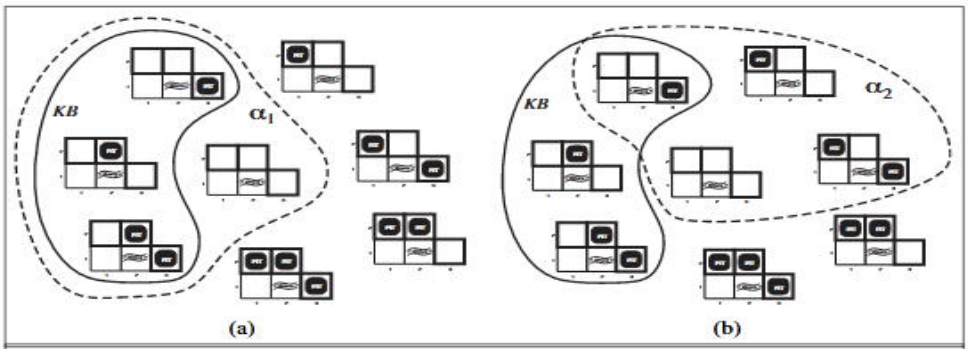


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

Propositional Logic: A Very Simple Logic:

We now present a simple but powerful logic called propositional logic. We cover the syntax of propositional logic and its semantics—the way in which the truth of sentences is determined. Then we look at entailment—the relation between a sentence and another sentence that follows from it—and see how this leads to a simple algorithm for logical inference. Everything takes place, of course, in the wumpus world.

Syntax:

The syntax of propositional logic defines the allowable sentences. The atomic sentences Consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other Letters or subscripts, for example: P , Q, R, W1,3 and North. The names are arbitrary but Are often chosen to have some mnemonic value—we use W1,3 to stand for the proposition That the wumpus is in [1,3]. (Remember that symbols such as W1,3 are atomic, i.e., W , 1, And 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed Meanings: True is the always-true proposition and False is the always-false proposition. Complex sentences are constructed from simpler sentences, using parentheses and logical

connectives. There are five connectives in common use:
¬ (not). A sentence such as ¬W1,3 is called the negation of W1,3. A literal is either an
∧ (and). A sentence whose main connective is ∧, such as W1,3 ∧ P3,1, is called a conjunction; its parts are the conjuncts. (The ∧ looks like an "A" for "And.")
∨ (or). A sentence using ∨, such as (W1,3 ∧ P3,1)∨ W2,2, is a disjunction of the disjuncts (W1,3 ∧ P3,1) and W2,2. (Historically, the ∨ comes from the Latin "vel," which means "or." For most people, it is easier to remember ∨ as an upside-down ∧.)
⇒ (implies). A sentence such as (W1,3 ∧ P3,1) ⇒ ¬W2,2 is called an implication (or conditional). Its premise or antecedent is (W1,3 ∧ P3,1), and its conclusion or consequent is ¬W2,2. Implications are also known as rules or if–then statements. The implication symbol is sometimes written in other books as ⊃ or →.
⇔ (if and only if). The sentence W1,3 ⇔ ¬W2,2 is a biconditional. Some other books write this as ≡

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.

Semantics:
Having specified the syntax of propositional logic, we now specify its semantics. The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the truth value—true or false—for every proposition symbol.
For example, if the sentences in the knowledge base make use of the proposition symbols P1,2, P2,2, and P3,1, then one possible model is m1 = {P1,2 = false, P2,2 = false, P3,1 = true} . With three proposition symbols, there are 23 = 8 possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. P1,2 is just a symbol; it might mean "there is a pit in [1,2]" or "I'm in Paris today and tomorrow."
The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:
• True is true in every model and False is false in every model.
• The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m1 given earlier, P1,2 is false.
For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):

• ¬P is true iff P is false in m.
• P ∧ Q is true iff both P and Q are true in m.
• P ∨ Q is true iff either P or Q is true in m.
• P ⇒ Q is true unless P is true and Q is false in m.
• P ⇔ Q is true iff P and Q are both true or both false in m.
The rules can also be expressed with truth tables that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation.

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

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

A simple inference procedure:
Our goal now is to decide whether KB |= α for some sentence α. For example, is ¬P1,2 entailed by our KB? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of true or false to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are B1,1, B2,1, P1,1, P1,2, P2,1, P2,2, and P3,1. With seven symbols, there are 27 = 128 possible models; in three of these, KB is true (Figure 7.9). In those three models, ¬P1,2 is true, hence there is no pit in [1,2]. On the other hand, P2,2 is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2]

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. *KB* is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic
  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
      else return true // when KB is false, always return true
  else do
      P ← FIRST(symbols)
      rest ← REST(symbols)
      return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
              and
              TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

Propositional Theorem Proving:

So far, we have shown how to determine entailment by model checking: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by theorem proving—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

$$\alpha \equiv \beta \text{ if and only if } \alpha \models \beta \text{ and } \beta \models \alpha .$$

The second concept we will need is validity. A sentence is valid if it is true in all models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as tautologies—they are necessarily true. Because the sentence True is true in all models, every valid sentence

is logically equivalent to True. What good are valid sentences? From our definition of entailment, we can derive the deduction theorem, which was known to the ancient

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

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

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

Inference and proofs:

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

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

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the sentence β can be inferred. For example, if (WumpusAhead ∧ WumpusAlive) ⇒ Shoot and (WumpusAhead ∧ WumpusAlive) are given, then Shoot can be inferred.
Another useful inference rule is And-Elimination, which says that, from a conjunction, any of the conjuncts can be inferred:

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

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

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

Not all inference rules work in both directions like this.

For example, we cannot run Modus
Ponens in the opposite direction to obtain $\alpha \Rightarrow \beta$ and α from β.
Let us see how these inference rules and equivalences can be used in the wumpus world.
We start with the knowledge base containing R1 through R5 and show how to prove $\neg P_{1,2}$,
that is, there is no pit in [1,2]. First, we apply biconditional elimination to R2 to obtain

$R6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$.
Then we apply And-Elimination to R6 to obtain
$R7 : ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$.
Logical equivalence for contrapositives gives
$R8 : (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}))$.
Now we can apply Modus Ponens with R8 and the percept R4 (i.e., $\neg B_{1,1}$), to obtain
$R9 : \neg(P_{1,2} \vee P_{2,1})$.
Finally, we apply De Morgan's rule, giving the conclusion
$R10 : \neg P_{1,2} \wedge \neg P_{2,1}$.
That is, neither [1,2] nor [2,1] contains a pit.
We found this proof by hand, but we can apply any of the search algorithms in Chapter 3
to find a sequence of steps that constitutes a proof. We just need to define a proof problem as
follows:

- **INITIAL S TATE** : 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

Proof by resolution:

Resolution algorithm is a rule used in Artificial Intelligence (AI) for logical reasoning. It helps our AI system to figure out if the given statement is logically proven from a set of known facts or not. It operates mainly on statements expressed in Conjunctive Normal Form (CNF) and is most commonly used in Propositional Logic and First-Order Predicate Logic.

For example:

If you have a statement like "It is raining OR it is sunny," the algorithm will try to determine if this is always true, sometimes true or never true based on the information provided.
The algorithm works by systematically combining logical statements until it either finds a contradiction meaning the statement is false or confirms that the statement is true.
Working of Resolution Algorithm

The Resolution Algorithm works on clauses that are small logical statements connected by "AND" or "OR." It works by:

Step 1: Convert Statement into Logical Forms

First, input sentence is converted into a standard format called Conjunctive Normal Form (CNF) means we break down the complex statements into simple parts connected by "AND" and "OR."

If the Original statement: "If it is raining then the ground is wet." then its CNF is: "NOT(Raining) OR Wet.

Step 2: Combine Clauses Using Resolution Rule

The core idea of the Resolution Algorithm is the resolution rule which combines two clauses to produce a new clause.

If you have two clauses say A OR B and NOT(A) OR C you can combine them to get B OR C. This process eliminates one variable (A in this case) and simplifies the problem.

Step 3: Repeat Until You Find an Answer

The algorithm keeps applying the resolution rule to pairs of clauses until one of two things happens:

Contradiction Found : If the algorithm produces an empty clause (written as FALSE) it means the original set of statements is inconsistent and cannot be true at the same time.

No Contradiction : If no empty clause is found after trying all combinations then the statements are consistent and can coexist..

Example: Resolution in Wumpus World

Let's consider the Wumpus World a well-known AI environment. In this scenario the resolution algorithm can be used to make logical inferences about the state of the environment. Suppose the agent is at position [1,1] and there is no breeze. This implies that there are no pits in adjacent squares. The knowledge base (KB) for this situation might look like:

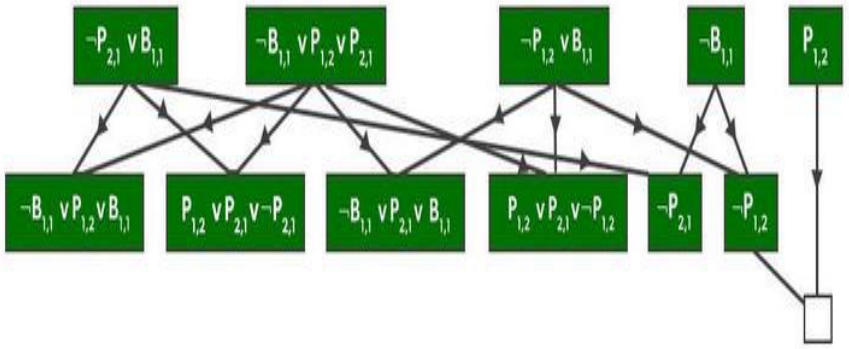
KB=(B1,1⇔(P1,2∨P2,1))∧¬B1,1KB=(B1,1⇔(P1,2∨P2,1))∧¬B1,1

Here B_{1,1} represents the presence of a breeze and P_{1,2} and P_{2,1} represent the presence of pits. We want to determine whether there is a pit in P_{1,2}, i.e verify α = P_{1,2}.

The resolution algorithm works by converting KB ∧ ¬α into CNF and applying the resolution rule to derive a contradiction. The clauses generated in the process might include:

1. B_{1,1} ∨ P_{1,2}
2. ¬B_{1,1}
3. P_{1,2} ∨ ¬P_{1,2} (which resolves to True)

The empty clause is derived when P_{1,2} is resolved with ¬P_{1,2} indicating that the query is entailed by the knowledge base.



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

  clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new ← { }
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

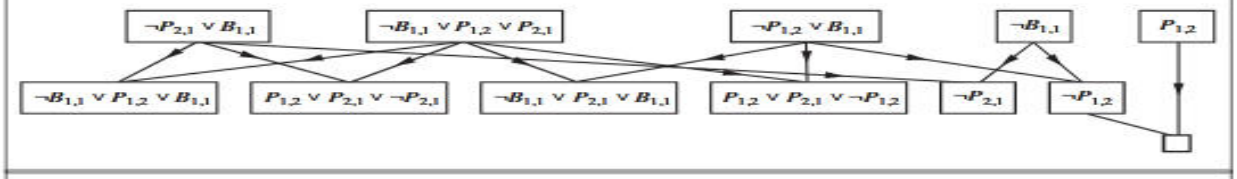


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

Horn clauses and definite clause:

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.
One such restricted form is the definite clause, which is a disjunction of literals of which exactly one is positive. For example, the clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) is a definite clause, whereas (¬B1,1 ∨ P1,2 ∨ P2,1) is not.
Slightly more general is the Horn clause, which is a disjunction of literals of which at most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called goal clauses. Horn clauses are closed under resolution: if you resolve

two Horn clauses, you get back a Horn clause.

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.13.) For example, the definite clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) can be written as the implication (L1,1 ∧ Breeze) ⇒ B1,1. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. In Horn form, the premise is called the body and the conclusion is called the head. sentence consisting of a single positive literal, such as L1,1, is called a fact. It too can be written in implication form as True ⇒ L1,1, but it is simpler to write just L1,1.

2. Inference with Horn clauses can be done through the forward-chaining and backward-chaining algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for logic programming, which is discussed in Chapter 9.

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

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

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.

Forward changing:

Forward changing is a data-driven inference technique. It starts with the available data and applies rules to infer new data until a goal is reached. This method is commonly used in situations where the initial data set is extensive, and the goal is to derive conclusions from it.

How Forward Chaining Works:

1. Start with Known Facts: The inference engine begins with the known facts in the knowledge base.
2. Apply Rules: It looks for rules whose conditions are satisfied by the known facts.
Infer New Facts: When a rule is applied, new facts are inferred and added to the knowledge base.
3. Repeat: This process is repeated until no more rules can be applied or a specified goal is achieved.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional Horn clauses
          q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known to be true in KB

  while agenda is not empty do
      p ← POP(agenda)
      if p = q then return true
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
              if count[c] = 0 then
                  PUSH(HEAD[c], agenda)
  return false
```

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

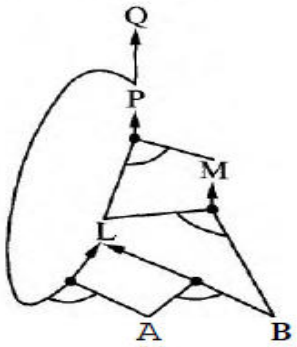
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

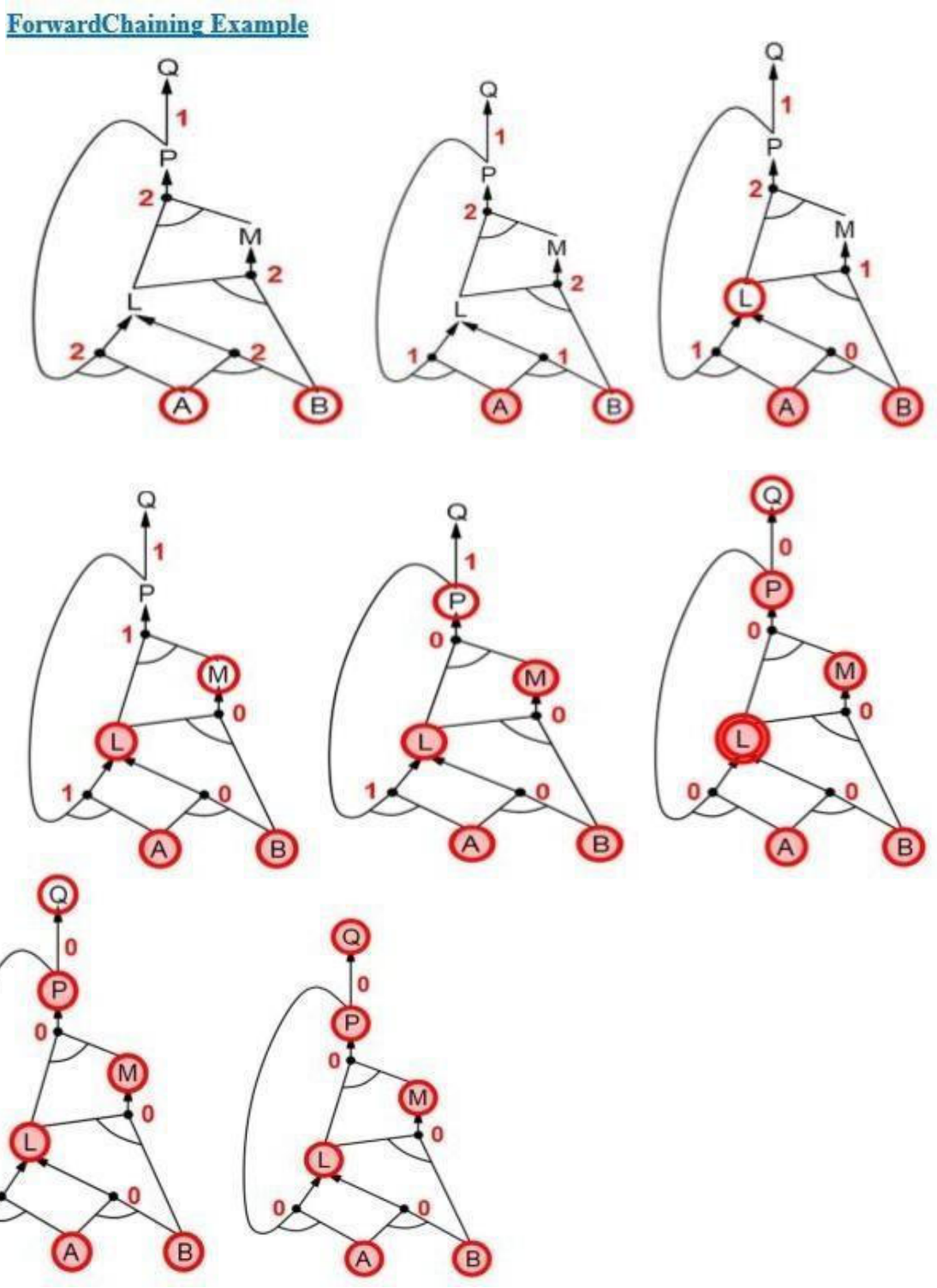
B

(a)



(b)

Figure 7.15 (a) A simple knowledge base of Horn clauses. (b) The corresponding AND–OR graph.



Example of Forward Chaining

Consider a medical diagnosis system where rules are used to diagnose diseases based on symptoms:

Fact: The patient has a fever.

Rule: If a patient has a fever and a rash, they might have measles.

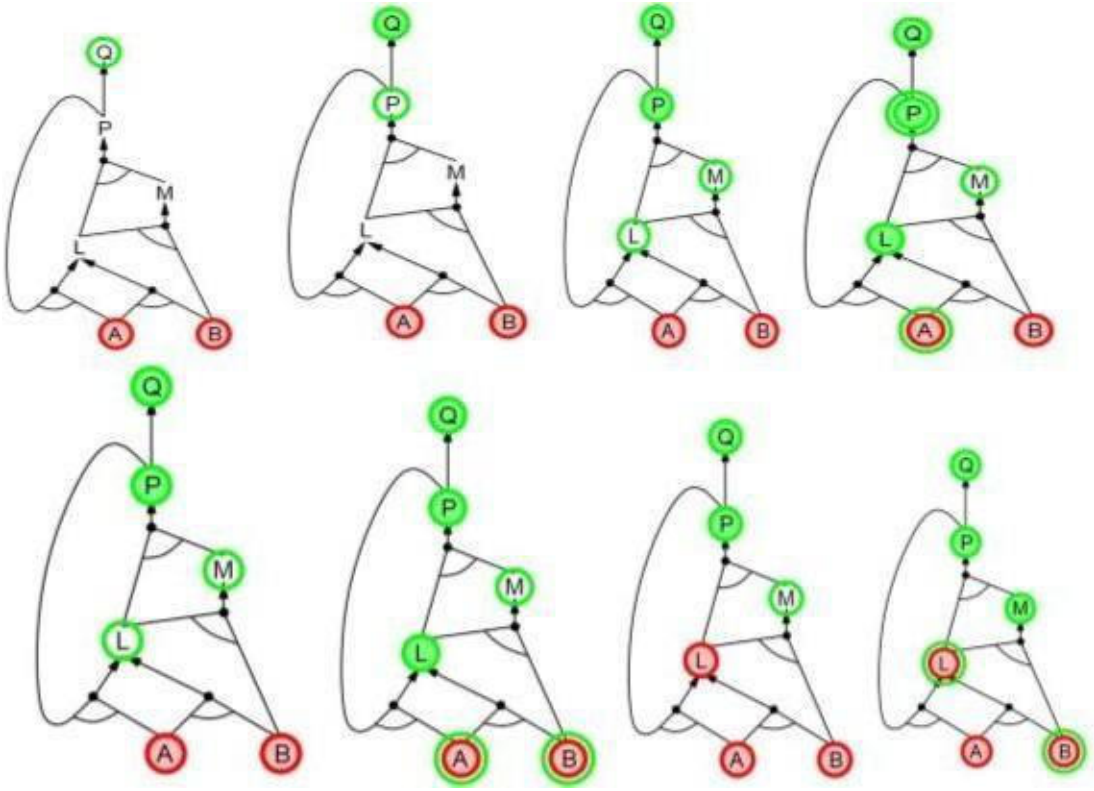
Starting with the known fact (fever), the system checks for other symptoms (rash). If the patient also has a rash, the system infers the possibility of measles.

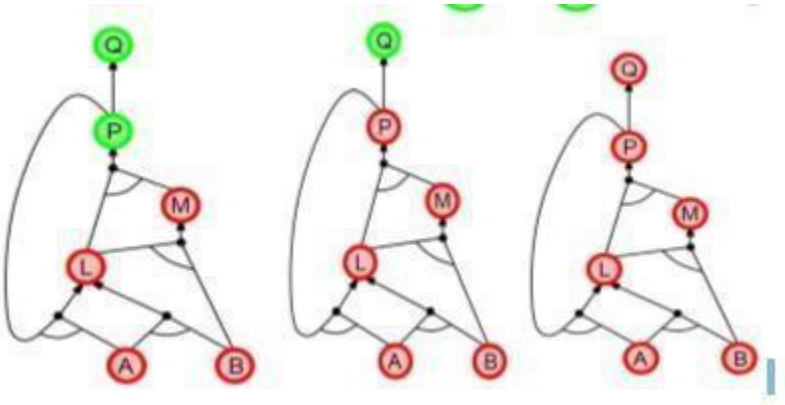
Backward Chaining:

Backward chaining is a goal-driven inference technique. It starts with the goal and works backward to determine which facts must be true to achieve that goal. This method is ideal for situations where the goal is clearly defined, and the path to reach it needs to be established.

How Backward Chaining Works

1. Start with a Goal: The inference engine begins with the goal or hypothesis it wants to prove.
2. Identify Rules: It looks for rules that could conclude the goal.
3. Check Conditions: For each rule, it checks if the conditions are met, which may involve proving additional sub-goals.
4. Recursive Process: This process is recursive, working backward through the rule set until the initial facts are reached or the goal is deemed unattainable.





Example of Backward Chaining
In a troubleshooting system for network issues:
Goal: Determine why the network is down.
Rule: If the router is malfunctioning, the network will be down.
The system starts with the goal (network down) and works backward to check if the router is malfunctioning, verifying the necessary conditions to confirm the hypothesis.

Effective propositional model checking:

Effective propositional model checking is a technique in Artificial Intelligence for verifying if a logical statement is true under all conditions that satisfy a given knowledge base, typically by creating a truth table for all possible scenarios. While effective for smaller problems, its performance can be significantly improved by using algorithms like the Davis-Putnam-Logemann-Loveland (DPLL) algorithm or local search methods. These algorithms incorporate optimizations such as early termination, pure symbol heuristics, and unit propagation to solve the problem more efficiently than a brute-force truth table enumeration.

How Truth Table Model Checking Works (and its limitations)

1. Define Propositions:
Identify all individual propositional symbols (e.g., A, B, C) in your knowledge base.
2. Create a Truth Table:
Systematically assign every possible combination of "true" and "false" to these symbols, creating 2^n rows, where n is the number of propositions.
3. Evaluate the Knowledge Base (KB):
For each row (a potential model), determine if the statements in your knowledge base are all true.
4. Check the Conclusion:
In the rows where the knowledge base is true, examine the truth value of the conclusion you are trying to prove.

5. Determine Validity:

If the conclusion is true in every single row where the knowledge base is true, then the knowledge base logically entails the conclusion.

Limitations: The main limitation of this method is the exponential growth of the truth table with each additional propositional symbol, making it inefficient for larger or more complex systems.

More Effective Techniques

To make propositional model checking more practical, various algorithms are used:

DPLL Algorithm:(Davis-Putnam-Logemann-Loveland)

- A **backtracking-based** search algorithm.
- Used in **SAT solvers**.
- Optimizes the checking by pruning the search space using rules like **unit propagation** and **pure literal elimination**.

This is a backtracking-based algorithm that improves upon basic truth-table enumeration.

Early Termination: Detects if a formula must be true or false early, even with only a partial model.

Pure Symbol Heuristic: If a symbol always appears with the same polarity (e.g., always "A", never "¬A"), it can be assigned a truth value that satisfies that symbol's clauses.

Unit Clause Heuristic: If a clause contains only one unassigned literal, that literal's truth value can be set to satisfy the clause immediately.

Effective propositional model checking:

Effective propositional model checking is a technique in Artificial Intelligence for verifying if a logical statement is true under all conditions that satisfy a given knowledge base, typically by creating a truth table for all possible scenarios. While effective for smaller problems, its performance can be significantly improved by using algorithms like the Davis-Putnam-Logemann-Loveland (DPLL) algorithm or local search methods. These algorithms incorporate optimizations such as early termination, pure symbol heuristics, and unit propagation to solve the problem more efficiently than a brute-force truth table enumeration.

How Truth Table Model Checking Works (and its limitations)

1. Define Propositions:

Identify all individual propositional symbols (e.g., A, B, C) in your knowledge base.

2. Create a Truth Table:

Systematically assign every possible combination of "true" and "false" to these symbols, creating 2^n rows, where n is the number of propositions.

3. Evaluate the Knowledge Base (KB):

For each row (a potential model), determine if the statements in your knowledge base are all true.

4. Check the Conclusion:

In the rows where the knowledge base is true, examine the truth value of the conclusion you are trying to prove.

5. Determine Validity:

If the conclusion is true in every single row where the knowledge base is true, then the knowledge base logically entails the conclusion.

Limitations: The main limitation of this method is the exponential growth of the truth table with each additional propositional symbol, making it inefficient for larger or more complex systems.

More Effective Techniques

To make propositional model checking more practical, various algorithms are used:

DPLL Algorithm:(Davis-Putnam-Logemann-Loveland)

- A **backtracking-based** search algorithm.
- Used in **SAT solvers**.
- Optimizes the checking by pruning the search space using rules like **unit propagation** and **pure literal elimination**.

This is a backtracking-based algorithm that improves upon basic truth-table enumeration.

Early Termination: Detects if a formula must be true or false early, even with only a partial model.

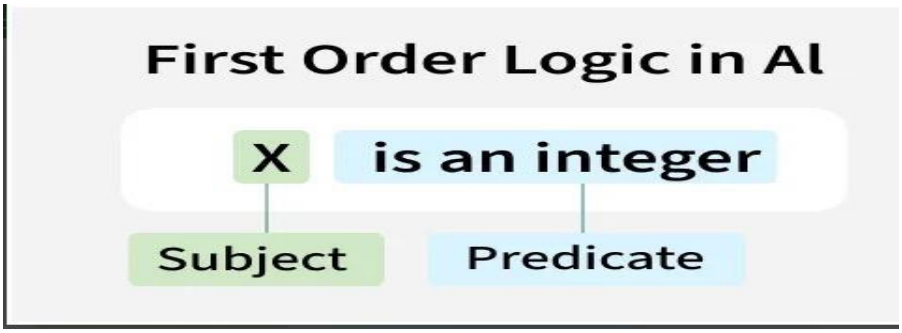
Pure Symbol Heuristic: If a symbol always appears with the same polarity (e.g., always "A", never "¬A"), it can be assigned a truth value that satisfies that symbol's clauses.

Unit Clause Heuristic: If a clause contains only one unassigned literal, that literal's truth value can be set to satisfy the clause immediately.

UNIT-3

First Order Logic:

First-Order Logic (FOL) also known as predicate logic helps us represent knowledge, reason through problems and understand language. By building on propositional logic and adding quantifiers and predicates, FOL allows us to express more complex relationships and make decisions based on logical reasoning. This makes it an important part of how we create intelligent systems that can reason and interact with the world in a meaningful way. It is used in many fields including mathematics, philosophy, linguistics and computer science.



Key components of First-Order Logic:

FOL extends propositional logic by introducing quantifiers and predicates, making it more expressive and versatile. Let's see various key components of FOL:
1. Constants: These represent specific objects or entities (Example: Alice, 2, NewYork).
2. Variables: These stand for unspecified objects or entities (Example: x, y, z).
3. Predicates: These define properties or relationships (Example: Likes(Alice, Bob) means "Alice likes Bob").
4. Functions: It map objects to other objects (Example: MotherOf(x) refers to the mother of x).
5. Quantifiers: These define the scope of variables:

- **Universal Quantifier (∀):** Applies a predicate to all elements (Example: ∀x (Person(x) -> Mortal(x)) means "All persons are mortal").
- **Existential Quantifier (∃):** Shows the existence of at least one element (Example: ∃x (Person(x) ∧ Likes(x, IceCream)) means "Someone likes ice cream").
- **Logical Connectives:** Include conjunction (∧), disjunction (∨), implication (→), biconditional (↔) and negation (¬).

Syntax and Semantics Logic:

FOL's syntax defines how to construct valid logical expressions, while semantics gives meaning to those expressions based on an interpretation which provides a domain of discourse and assigns meanings to constants, predicates and functions.

For example, consider the domain of natural numbers. The predicate GreaterThan(x, y) holds if x is greater than y.

If x = 5 and y = 3, GreaterThan(5, 3) is true.

Example: Logical Reasoning with FOL

Consider the following statements:

- **∀x (Cat(x) -> Mammal(x))** (All cats are mammals)
- **∀x (Mammal(x) -> Animal(x))** (All mammals are animals)
- **Cat(Tom)** (Tom is a cat)

From these, we can logically infer:

- **Mammal(Tom)** (Tom is a mammal)
- **Animal(Tom)** (Tom is an animal)

Propositional Logic Vs First-Order Logic

Propositional Logic (PL)	First-Order Logic (FOL)
Represents entire statements as true or false (e.g "It is raining")	Represents relationships, properties and generalizations (e.g "All cats are mammals")
No quantifiers	Uses quantifiers (∀ for "all", ∃ for "some") to express generalizations and existence
Basic logical operations (AND, OR, NOT)	Advanced reasoning through unification, resolution and inference rules
Simple tasks like decision-making and circuit design	Complex tasks like knowledge representation, reasoning and language processing

Syntax and Semantics of First-Order Logic:

First-order logic (FOL), also known as first-order predicate logic, is a fundamental formal system used in mathematics, philosophy, computer science, and linguistics for expressing and reasoning about relationships between objects in a domain. In artificial intelligence (AI), first-order logic (FOL) serves as a cornerstone for representing and reasoning about knowledge. Its syntax and semantics provide a robust framework for encoding information in a precise and structured manner, enabling AI systems to perform tasks such as automated reasoning, planning, and natural language understanding.

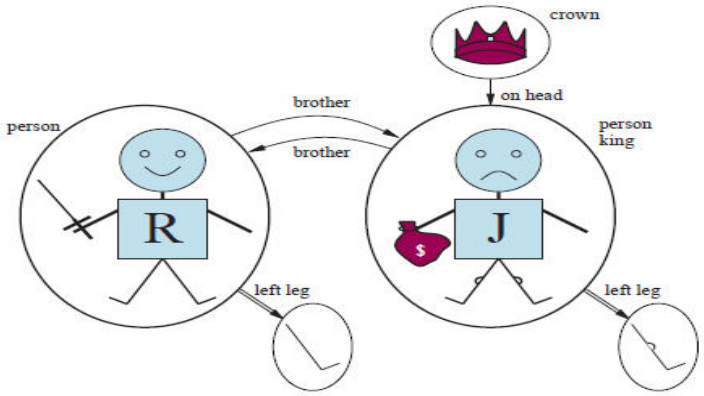
1. **Models for first-order logic**
2. **Symbols and interpretations**
3. **Terms**
4. **Atomic sentences**
5. **Complex sentences**
6. **Quantifiers**
7. **Equality**
8. **Database Semantics**

1. Models for first-order logic:

Models for first-order logic are much more interesting. First, they have objects in them! The domain of a model is the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object.

The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of tuples of objects that are related.(A tuple is a collection of objects arranged in a fixed order and is written with angle brackets Surrounding the objects.) Thus, the brotherhood relation in this model is the set

{<Richard the Lionheart, King John>, <King John,Richard the Lionheart>}



For example, each person has one left leg, so the model has a unary "left leg" function—a mapping from a one-element tuple to an object—thatincludes the following mappings:

⟨Richard the Lionheart⟩ → Richard's left leg
⟨King John⟩ → John's left leg.

2. Symbols and interpretations:

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: constant symbols, which stand for objects; predicate symbols, which stand for relations; and function symbols, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols Richard and John; the predicate symbols Brother, On Head, Person, King, and Crown; and the function symbol Left Leg. As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an arity that fixes the number of arguments.

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

3. **Terms:**

A term is a logical expression that refers to an object. Constant symbols are terms, but it is Not always convenient to have a distinct symbol to name every object. In English we might Use the expression “King John’s left leg” rather than giving a name to his leg. This is what Function symbols are for: instead of using a constant symbol, we use LeftLeg(John). In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a “subroutine call” that “returns a value.” There is no LeftLeg subroutine that takes a person as input and returns a leg. We can reason about left legs.

4. **Atomic sentences:**

We have terms for referring to objects and predicate symbols for referring to relations, we can combine them to make atomic sentences that state facts. An atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as

Brother(Richard;John):

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John. Atomic sentences can have complex terms as arguments.

Married(Father(Richard);Mother(John))

Thus, states that Richard the Lionheart’s father is married to King John’s mother

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

5. Complex sentences

We can use logical connectives to construct more complex sentences, with the same syntax and semantics as in propositional calculus.

Here are four sentences that are true in the model

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$
$$King(Richard) \vee King(John)$$
$$\neg King(Richard) \Rightarrow King(John).$$

6. Quantifiers:

Quantifiers are used in logic to show how much a statement is true for a set of things. They show whether something is true for all members of the group, or only for some of them. The two main types are:

Universal Quantifier (∀):

The universal quantifier (∀) indicates that a predicate is true for all elements in a given domain.

Example: ∀ x ∈N, P(x)

Translation: "For all natural numbers xxx, xxx is even."

Ex:

the universally quantified sentence is equivalent to asserting the following five sentences:

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

Existential Quantifier (∃):

The existential quantifier (∃) indicates that there exists at least one element in a given domain for which the predicate is true.

Example: ∃x ∈N, P(x)

Translation: "There exists a natural number x such that x is even."

EX:

Richard the Lionheart is a crown ∧ Richard the Lionheart is on John's head;
King John is a crown ∧ King John is on John's head;
Richard's left leg is a crown ∧ Richard's left leg is on John's head;
John's left leg is a crown ∧ John's left leg is on John's head;
The crown is a crown ∧ the crown is on John's head.

Connections between ∀ and ∃

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x\ \neg Likes(x, Parsnips) \quad \text{is equivalent to} \quad \neg\exists x\ Likes(x, Parsnips).$$

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

$$\forall x\ Likes(x, IceCream) \quad \text{is equivalent to} \quad \neg\exists x\ \neg Likes(x, IceCream).$$

Because ∀ is really a conjunction over the universe of objects and ∃ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

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

7. Equality:

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

For example:

Father (John) = Henry

Says that the object referred to by Father (John) and the object referred to by Henry are the Same. Because an interpretation fixes the referent of any term, determining the truth of an Equality sentence is simply a matter of seeing that the referents of the two terms are the same Object.

8. Database Semantics:

Suppose that we believe that Richard has Two brothers, John and Geoffrey. We could write

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard), \tag{8.3}$$

but that wouldn't completely capture the state of affairs. First, this assertion is true in a model where Richard has only one brother—we need to add $John \neq Geoffrey$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \wedge John \neq Geoffrey$$
$$\wedge \forall x\ Brother(x, Richard) \Rightarrow (x = John \vee x = Geoffrey).$$

Using First-Order Logic:

Now that we have defined an expressive logical language, let's learn how to use it. we provide example sentences in some simple domains. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge. We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world.

1. Assertions and queries in firrst-order logic:

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such Sentences are called assertions. For example, we can assert that John is a king, Richard is a Person, and all kings are persons:

TELL(KB, King(John))
TELL(KB,Person(Richard))
TELL(KB, ∀x King(x) => Person(x))

We can ask questions of the knowledge base using ASK. For example,

ASK(KB, King(John))

2. The kinship domain:

The first 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."

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's parent who is female:

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

One's husband is one's male spouse:

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

Parent and child are inverse relations:

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

A grandparent is a parent of one's parent:

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

A sibling is another child of one's parent:

$$\forall x,y \;\; Sibling(x,y) \;\Leftrightarrow\; x \neq y \wedge \exists p \;\; Parent(p,x) \wedge Parent(p,y) .$$

3. Numbers, sets, and lists:

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or nonnegative 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). The Peano axioms define natural numbers and addition. Natural numbers are defined recursively:

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

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

$$\forall n \;\; 0 \neq S(n) .$$
$$\forall m,n \;\; m \neq n \Rightarrow S(m) \neq S(n) .$$

Now we can define addition in terms of the successor function:

$$\forall m \;\; NatNum(m) \Rightarrow +(0,m) = m .$$
$$\forall m,n \;\; NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m),n) = S(+(m,n)) .$$

4. **The wumpus world:**

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

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

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

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

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

$$\text{ASKVARS}(KB, BestAction(a, 5)),$$

which returns a binding list such as $\{a/Grab\}$. The agent program can then return *Grab* as the action to take. The raw percept data implies certain facts about the current state. For example:

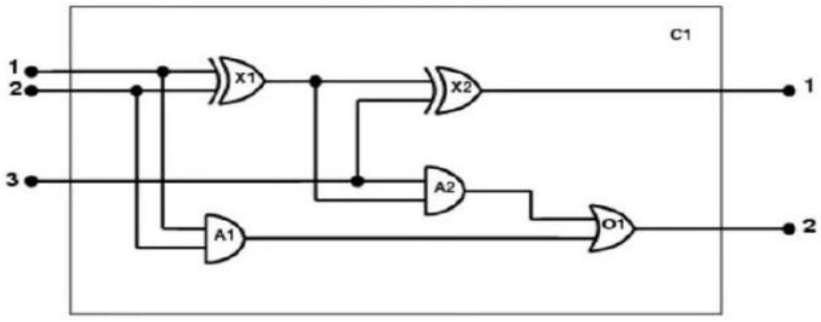
$$\begin{aligned}
&\forall t, s, g, w, c\ \ Percept([s, Breeze, g, w, c], t) \Rightarrow Breeze(t)\\
&\forall t, s, g, w, c\ \ Percept([s, None, g, w, c], t) \Rightarrow \neg Breeze(t)\\
&\forall t, s, b, w, c\ \ Percept([s, b, Glitter, w, c], t) \Rightarrow Glitter(t)\\
&\forall t, s, b, w, c\ \ Percept([s, b, None, w, c], t) \Rightarrow \neg Glitter(t)
\end{aligned}$$

Knowledge Engineering in First-Order Logic:

Knowledge Engineering in First-Order Logic (FOL) involves encoding knowledge that is represented through objects, predicates, functions, and logical rules to enable AI systems to reason. It provides facts, axioms, and inference rules used to model complex relationships so machines can learn and make logical choices.

We will analyze how the knowledge engineering approach is applied in electronic circuits. This strategy works very well to develop custom-built knowledge bases for specific functions.

- One-bit full adder



Steps of Knowledge Engineering in FOL:

1. **Identify the task:**

The first stage in the procedure is to define the task; in the case of a digital circuit, various reasoning tasks can be accomplished.

- Identify the task similar to PEAS (Performance measure, Environment, Actuators, Sensors) design.
- Knowledge engineer must describe the range of the question that the knowledge base will support.
- There are many reasoning tasks associated with digital circuits. At the highest level, we will examine the functionality circuit.
- For example, does the circuit in actually add properly? (circuit verification)
- What are the expected outputs for given inputs?
- At the second level we will examine circuit's structure.
- For example, what are all the gates connected to the first input terminal?
- Does the circuit contain feedback loops?

2. Gather Relevant Knowledge:

We need to understand how gates interpret input signals.

- The circuit is made up of AND, OR, XOR, and NOT gates. AND, OR, and XOR have two inputs, while NOT only has one. Each gate has only one output, and circuits, like gates, have distinct input and output terminals.
- Irrelevant knowledge: Our study is unaffected by component characteristics such as size, shape, color, or cost.

3. **Choose a Vocabulary:**

To accurately represent the digital circuit in First-Order Logic (FOL), we establish a structured language that includes gates, terminals, connections, and signal states. The key components are as follows.

- **Gates and Types:** Each gate is designated as an object (e.g., Gate(X1)) and given a type (Type(X1) = XOR). Circuits are similarly labeled (Circuit(1)).
- **Terminals and connections:** Terminals are defined using Terminal(x), with In(i, X1) and Out(i, X1) representing inputs and outputs, respectively. Connections are represented as Connected(Out(1, X1), In(1, X2).
- **Signals and Values:** Signals at terminals are represented by Signal(t), where 1 (ON) and 0 (OFF) indicate active or inactive states.

4**. Encode general knowledge about the domain:**
These are all the axioms we need.

If two terminals are connected, then they have the same signal –

```
t1, t2 Terminal (t1)  Terminal (t2)  Connect (t1, t2) → Signal (t1) =
Signal (2)
```

The signal at every terminal is either 1 or 0 –

```
t Terminal (t) → Signal (t) = 1 Signal (t) = 0.
```

Connect is commutative –

```
t1, t2 Connect(t1, t2) → Connect (t2, t1).
```

They are four types of gates and represented as below –

```
g Gate(g)  r = Type(g) → r = OR r = AND r = XOR r = NOT.
```

AND gate's output is 0 if and only if any of its input is 0 –

```
g Gate(g)  Type(g) = AND → Signal (Out(1, g))= 0  n Signal (In(n, g))= 0.
```

An OR gate's output is 1 if and only if any of its input is 1 –

```
g Gate(g)  Type(g) = OR → Signal (Out(1, g))= 1  n Signal (In(n, g))= 1
```

An XOR gate's output is 1 if and only if its inputs are different –

```
 g Gate(g)  Type(g) = XOR → Signal (Out(1, g)) = 1  Signal (In(1, g))
Signal (In(2, g)).
```

A NOT gate's output is different from it's input (negation) –

```
g Gate(g)  Type(g) = NOT → Signal (In(1, g))  Signal (Out(1, g)).
```

The gates (except for NOT) have two inputs and one output.

```
g Gate(g)  Type(g) = NOT → Arity(g, 1, 1)
    g Gate(g)  r =Type(g)  (r= AND r= OR r= XOR) → Arity(g, 2, 1).
```

All gates are logic circuits –

```
g Gate(g) → Circuit (g).
```

5. **Encode a description of the problem instance:**

To define Circuit C1, we classify its components and generate linkages between them. This means that we describe the circuit and its gates as atomic sentences in First Order Logic (FOL).

First, we categorize the circuit and its component gates.

```
Circuit (C1) Arity(C1, 3, 2)
Gate(X1) A Type(X1) = XOR
Gate(X2) A Type(X2) = XOR
Gate(A1) A Type(A1) = AND
Gate(A2) A Type(A2) = AND
Gate(01) A Type(01) = OR
```

Second, we represent the connections between gates –

For example, Connect(Out(1, X1), In(1, X2)) represents X1s output going into X2s input.

6. **Pose queries to the inference procedure and get answers:**
In this phase, we will find all possible input combinations for all terminals in the adder circuit. What input combinations make C1's first output to be 0 and its second output to be

```
  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
```

This is a simple example of verification of circuits.

7. **Debug the knowledge base:**
We can test the knowledge base with small faults and see improper behavior.
For example: Failure to assert 1 0 may lead to wrong answers for any input save for particular ones like 000 and 110. Debugging through observing each gate's output helps pinpoint such faults.

Inference in First-Order Logic:

Inference in First Order Logic

- Inference in FOL is used to generate new sentences from existing sentences.
- Definition:
- An expression X logically follows form a set S, if every interpretation that satisfies S also satisfies X
- The function of logical inference is to produce new sentence that logically follow a given set of FOL sentence.

Universal Instantiation (UI) / Universal Elimination

- UI says that we can infer (produce) any sentence obtained by substituting a ground term for the variable.
- we use the notion of **Substitutions** for these instantiations.
- Let SUBST(θ,α) denote the result of applying the substitution θ to the sentence α

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

for any variable v and ground term g.

Substitutions

- E.g., KB contains "all greedy kings are evil"
- $\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ yields (eliminate $\forall$)
- SUBST(x/John)
- King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
- SUBST(x/Richard)
- King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)
- SUBST(x/Father(John))
- King(Father(John)) $\wedge$ Greedy(Father(John)) $\Rightarrow$ Evil(Father(John))

Existential Instantiation (EI) / Existential Elimination

- For any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base:

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

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

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

- Skolem Constant

provided C_1 is a new constant symbol which is not in KB but satisfy all properties of 'x', called a Skolem constant

(skolemization – replacing variables with ground terms)

Reduction to Propositional Inference

Suppose the KB contains just the following:

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

- Instantiating the universal sentence in all possible ways, we have:

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

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

Problems with Propositionalization

- Propositionalization seems to generate lots of irrelevant sentences.
- with function symbols, there are infinitely many ground terms,
  - e.g., *Father(Father(Father(John)))*

Inference in Propositional Logic:

Propositional Logic - Reasoning Patterns Inference Rules

- Standard patterns of inference that can be applied to derive chains of conclusions that lead to the desired goal. These patterns of inference are called inference rules.
- The useful inference rules are
  1. Modus Ponens
  2. And Elimination
- These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

Inference Rules – AND Elimination

- And Elimination
- From a conjunction, any of the conjuncts can be inferred

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

- Example
- (WumpusAhead $\wedge$ WumpusAlive), WumpusAlive can be inferred.

Inference Rules – Modus Ponens

- Modus Ponens
  - Given: S1 $\Rightarrow$ S2 and S1, derive S2
- Whenever sentences of the form $\alpha \Rightarrow \beta$ and α are given, then sentence β can be inferred

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

- (WumpusAhead $\wedge$ WumpusAlive) $\Rightarrow$ Shoot And (WumpusAhead $\wedge$ WumpusAlive),
- Shoot can be inferred

Example Proof By Deduction (Wumpus World)

- Knowledge
- **S1:** $B_{22} \Leftrightarrow (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})$ ***rule***
- **S2:** $\neg B_{22}$ ***observation***
- Inferences
- S3: $(B_{22} \Rightarrow (P_{21} \vee P_{23} \vee P_{12} \vee P_{32})) \wedge ((P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \Rightarrow B_{22})$ *[S1, bi elim]*
- S4: $((P_{21} \vee P_{23} \vee P_{12} \vee P_{32}) \Rightarrow B_{22})$ *[S3, And elim]*
- S5: $(\neg B_{22} \Rightarrow \neg(P_{21} \vee P_{23} \vee P_{12} \vee P_{32}))$ *[contrapos]*
- S6: $\neg(P_{21} \vee P_{23} \vee P_{12} \vee P_{32})$ *[S2,S6, MP]*
- S7: $\neg P_{21} \wedge \neg P_{23} \wedge \neg P_{12} \wedge \neg P_{32}$ *[S6, DeMorg]*

Propositional vs First-Order Logic Inference:

Feature	Propositional Inference	First-Order Inference
Level of Logic	Propositional Logic	First-Order Logic (Predicate Logic)
Basic Elements	Propositions (atomic sentences)	Objects, predicates, functions, and quantifiers
Expressiveness	Less expressive (can't represent object properties or relations)	More expressive (can represent object properties, relations, and quantification)
Variables	No variables	Uses variables (e.g., x, y)
Quantifiers	Not supported	Supports ∀ (for all), ∃ (there exists)
Example Sentence	$P \land Q \rightarrow R$	$\forall x$ (Human(x) → Mortal(x))
Inference Methods	Truth tables, resolution, forward/backward chaining	Unification, resolution, forward/backward chaining
Computational Complexity	Generally lower	Generally higher
Use Cases	Simple rule-based systems, digital circuits	Natural language processing, knowledge representation
Decidability	Decidable	Semi-decidable (not always decidable)

Unification in First-Order Logic (FOL):

Unification is a key concept in first-order logic (FOL) that involves finding a common substitution for two logical expressions, making them identical. This process is crucial for automated reasoning, theorem proving, and various inference methods in artificial intelligence.

- All FOL inference techniques depend on unified reasoning.
- The algorithm returns failure if two expressions do not match.
- The replacement variables achieved from unification are called Most General Unifier, or simply MGU.

Example

Let us say we have the following predicates − P(x, Dog) and P(Alex, y). To unify them, we must find substitutions such that both predicates become the same. The solution is −

- x → Alex
- y → Dog

After substitution, both predicates become P(Alex, Dog), meaning they are unified.

Conditions for Unification:

Following are some basic conditions for unification −

- To achieve unification, the predicate names in both formulations must be same. For example, Loves(x, y) and Hates(A, B) cannot be combined since "Loves" and "Hates" are not synonyms.
- Both expressions must have an equal number of parameters. If one expression is P(A, B) and another is P(x, y, z), they cannot be combined because the first has two arguments and the latter has three.
- A constant can only unify with itself. For example, if one phrase contains Apple and another contains Orange, unification fails since the two are distinct things.
- A variable can be swapped for a constant or another variable. For example, we can unify P(x, B) and P(A, y) by assigning x to A and y to B.
- A variable cannot be replaced with an expression containing itself. It is impossible to unify x with f(x) because it would create a cycle, and therefore, unification is not possible.
- If the same variable occurs twice in different places in an expression, unification fails. For instance, P(x, x) and P(A, B) cannot be unified since x cannot simultaneously be A and B.

Unification Algorithm:

Unification is a crucial technique in first-order logic (FOL) that allows two logical expressions to be made equal by finding an appropriate substitution. The UNIFY algorithm determines whether two expressions can be unified and provides the necessary substitutions to achieve this.

Algorithm: UNIFY (Expression1, Expression2)

Step 1: Check if either expression is a variable or constant.

- If the expressions are identical, return an empty substitute (no changes are needed).
- If one of them is a variable, check if the variable occurs in the other expression (to avoid circular dependency). If so, return FAILURE. Otherwise, substitute the variable with the corresponding term.
- If neither is a variable expression, then proceed to the next step.

Step 2: Compare the names of the predicates

- If the predicate names in the two expressions are different, unification cannot occur as they represent different relationships.

Step 3: Verify the number of arguments.

- If the number of parameters in the two expressions is not the same, unification is not possible. The algorithm will return a FAILURE response.

Step 4: Create an empty substitution set.

- Establish an empty set to hold any variable substitutions required for unification.

Step 5: Attempt to unify each argument recursively.

- Examine the arguments of the two expressions −
- Recursively use the UNIFY function on each pair of parameters.
- If a failure occurs at any point, return a FAILURE.
- If a substitution is identified, apply it to the remaining expressions.

Step 6: Return to the Most General Unifier (MGU)

- Once all the words have been successfully unified, present the final substitution set, known as the Most General Unifier (MGU).
- This ensures that the expressions are simplified to be equal using the simplest substitutions available.

Implementation of the Unification Algorithm
Example: Unifying Employee Roles

Determine the Most General Unifier (MGU) between WorksAt(TutorialsPoint, X, Manager) and WorksAt(TutorialsPoint, John, Y).

Step 1: Start with an empty set of substitutes.
Step 2: Combine the atomic statements step by step.

- The predicate "WorksAt" is identical, so we can proceed.
- The first argument "TutorialsPoint" matches in both cases.
- For the second argument, "X" is a variable, so we substitute X with John.
- For the third parameter, "Manager" and "Y": since Y is a variable, we substitute Y with Manager.
- Unified Expression: WorksAt (TutorialsPoint, John, Manager). Unifier: {X/John, Y/Manager}.

Forward chaining in AI with FOL proof:

Forward Chaining is one of the two methodologies using an inference engine, the other one being backward chaining. It starts with a base state and uses the inference rules and available knowledge in the forward direction till it reaches the end state. The process iterates till the final state is reached.

Now let's understand forward chaining with FOL with an example:

We will list down the facts initially and then convert facts to First-order Logic (FOL) using inference laws until we reach the goal state.

Some Fact:
It is crime for Americans to sell the weapon to the enemy of America
Country Nono is an enemy of America
Nono has some Missiles
All missiles were sold to Nono by Colonel
Colonel is American
Missile is a weapon

Goal:
Colonel is criminal

Steps to be followed:
Fact conversion to FOL:

Step1 : It is crime for Americans to sell the weapon to the enemy of America

$$FOL : American(x)^{W}eapon(y)^{E}nemy(z, America)^{S}ell(x, y, z) \rightarrow Criminal(x)$$

Step2 : Country Nono is enemy of America

```
FOL : Enemy(Nono, America)
```

Step3 : Nono has some Missile

```
FOL : Owns(Nono ,x)
      Missile(x)
```

Step4 : All missiles were sold to Nono by Colonel

```
FOL : Missile(x) ^ Owns(Nono,x) → Sell(Colonel,x,Nono)
```

Step5 : Colonel is American

```
FOL : American(Colonel)
```

Step6 : Missile is weapon

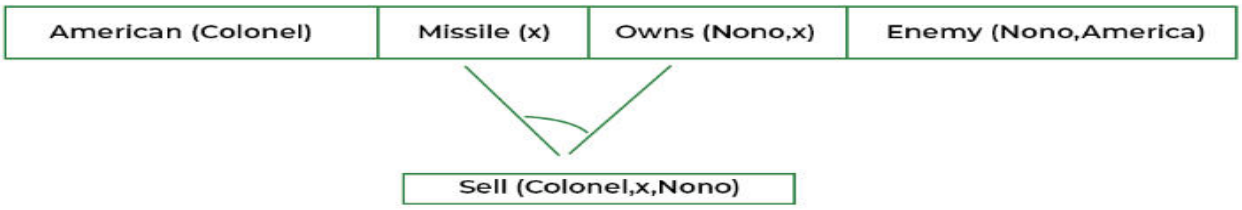
```
FOL : Missile(x)  → Weapon (x)
```

Proof:
Iteration 1:

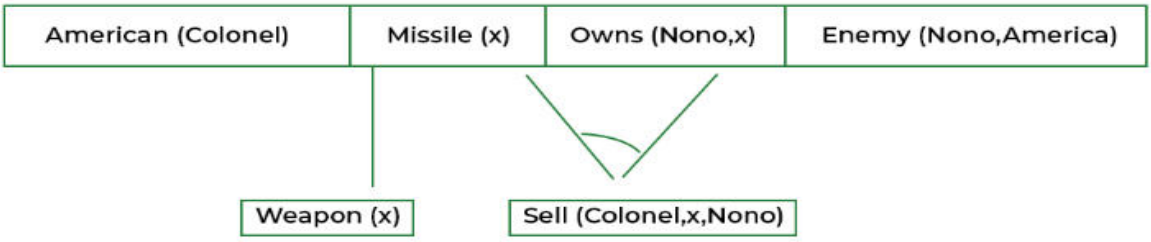
- Start with the knowledge base/known facts : The facts that are not deduced from any other [present in LHS of the statements]

American(Colonel)	Missile(x)	Owns(Nono , x)	Enemy(Nono , America)

- Add inferences one by one to link the connections
    - Rule (1) doesn't satisfy the so it will not be added
    - Rule (2) and Rule(3) FOL are already added
    - Rule (4) matches the inference so we will add it.



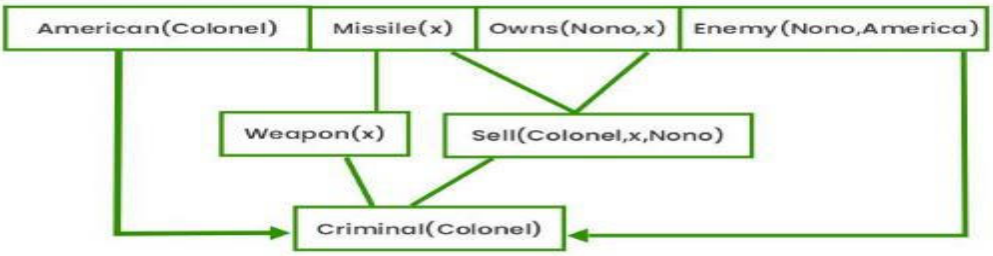
- Rule(5) doesn't have RHS so nothing added
- Rule (6) has weapon in RHS derived from Missile(x) , which is already a part so we will add weapon in our next state



That completes our first iteration

Iteration 2:

- Rule(1) has all the LHS conditions satisfied as we can see in the first iteration. So all the four FOL in LHS are available, now we can add Criminal(x) from RHS in our next state



We reached the goal state to deduce that: Colonel is a criminal

Forward chaining is data driven since we use available data to reach the final state.

Inference Engine
Forward Chaining Algorithm

Inference Engine

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

Horn Clause and Definite clause

- Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm.
- Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.
- **Definite clause:** A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.
- **Horn clause:** A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.
- **Example: (¬ p V ¬ q V k)**. It has only one positive literal k.
- It is equivalent to $p \wedge q \rightarrow k$.

What do you mean by forward chaining algorithm?

- 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.
- It is a bottom-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.

- **The Forward-chaining algorithm Starting from the known facts,**
- **it triggers the entire rules whose premises are satisfied,**
- **adding their conclusions to the known facts.**
- **The process repeats until the query is answered or no new facts are called.**

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.

Facts Conversion into FOL

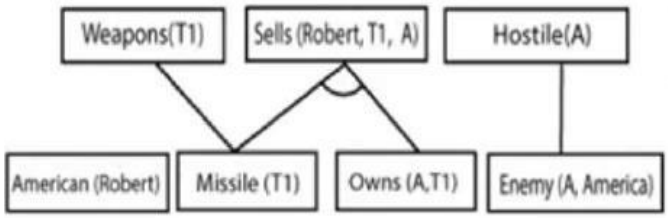
- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. **∃p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.
  Owns(A, T1) .....(2)
  Missile(T1) .....(3)
- All of the missiles were sold to country A by Robert.
  ∃p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) .....(4)
- Missiles are weapons.
  Missile(p) → Weapons (p) .....(5)
- Enemy of America is known as hostile.
  Enemy(p, America) →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

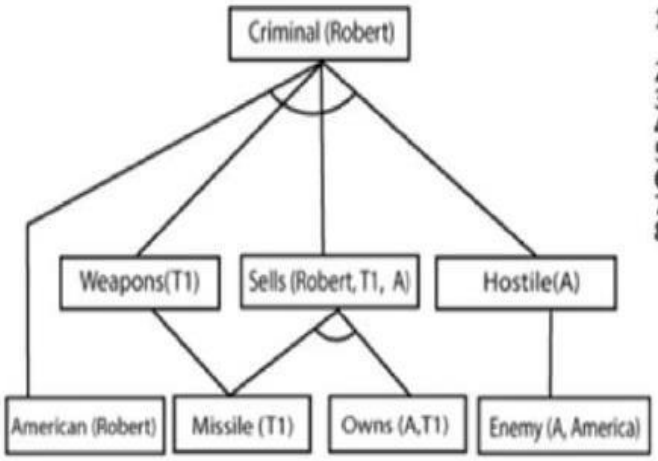
1. **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p)**
2. **Owns(A, T1)**
3. **Missile(T1)**
4. **∃p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A)**
5. **Missile(p) → Weapons (p)**
6. **Enemy(p, America) →Hostile(p)**
7. **Enemy (A, America)**
8. **American(Robert).**

- **Step-2**
- At the second step, we will see those facts which infer from available facts and with satisfied premises.
- Rule-(1) does not satisfy premises, so it will not be added in the first iteration.
- Rule-(2) and (3) are already added.
- Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).
- Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



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

- **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**



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

Inference Engine
Backward Chaining Algorithm

Inference Engine

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

Define backward chaining algorithm.

- 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.
- 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 also called as goal-driven approach, as a list of goals decides which rules are selected and used.

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.

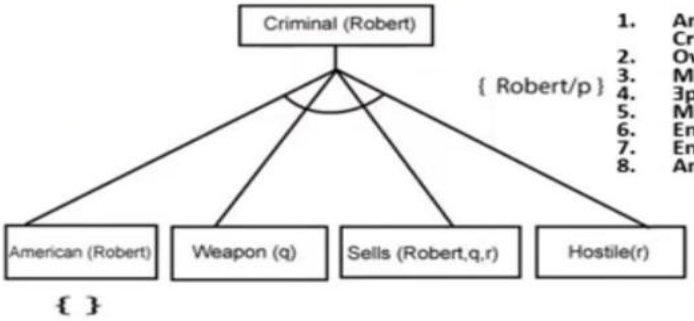
Facts Conversion into FOL

- 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, introducing new Constant T1.
  Owns(A, T1) .....(2)
  Missile(T1) .....(3)
- All of the missiles were sold to country A by Robert.
  ∃p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) .....(4)
- Missiles are weapons.
  Missile(p) → Weapons (p) .....(5)
- Enemy of America is known as hostile.
  Enemy(p, America) →Hostile(p) .....(6)
- Country A is an enemy of America.
  Enemy (A, America) .....(7)
- Robert is American
  American(Robert). .....(8)

Backward-Chaining proof

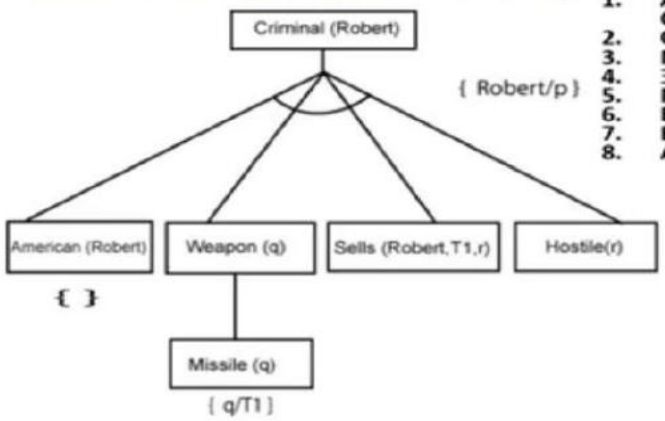
- In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert).** And from the goal fact, we will infer other facts, and at last, we will prove those facts true.
- So our goal fact is "Robert is Criminal,"
- **Step-1:**
- At the first step, we will take the goal fact.

- **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**



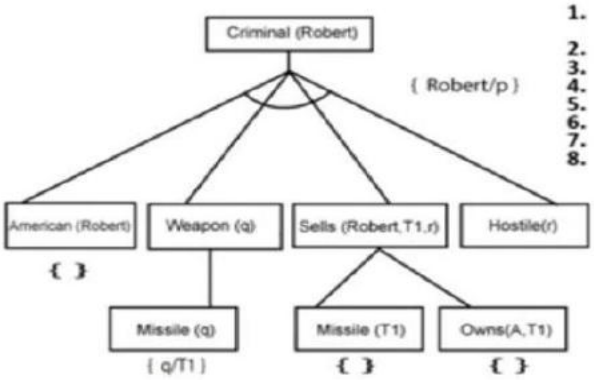
1. American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p)
2. Owns(A, T1)
3. Missile(T1)
4. ∃p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A)
5. Missile(p) → Weapons (p)
6. Enemy(p, America) →Hostile(p)
7. Enemy (A, America)
8. American(Robert).

- **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.



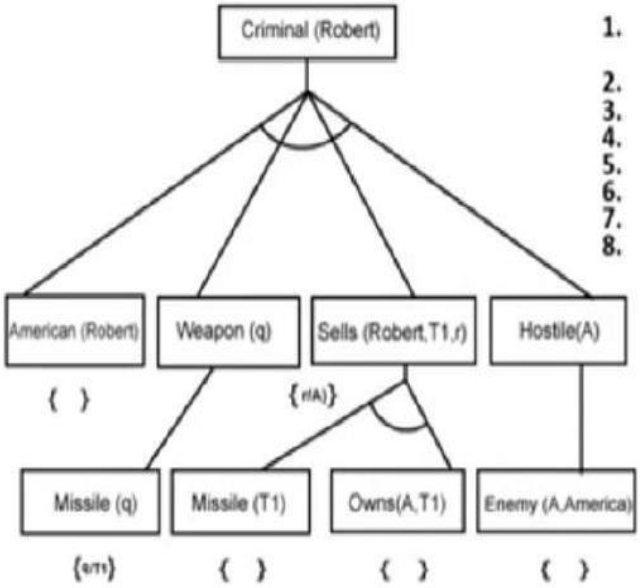
1. American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p)
2. Owns(A, T1)
3. Missile(T1)
4. ∃p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A)
5. Missile(p) → Weapons (p)
6. Enemy(p, America) →Hostile(p)
7. Enemy (A, America)
8. American(Robert).

- **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.



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

- **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



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

ARTIFICIAL INTELLIGENCE

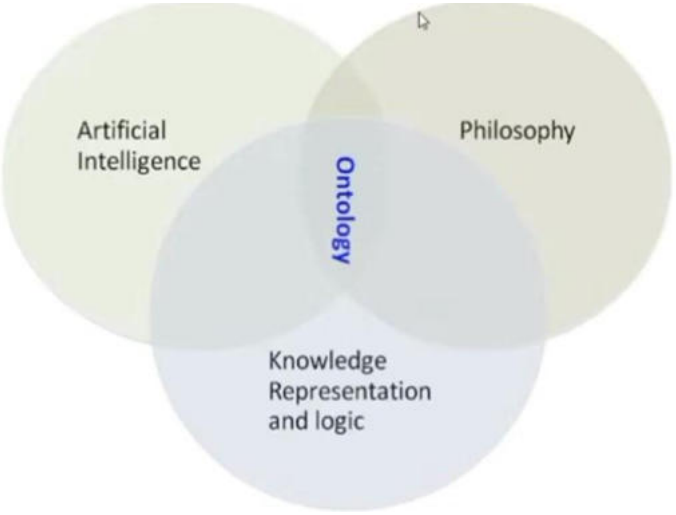
UNIT 4 - KNOWLEDGE REPRESENTATION AND PLANNING

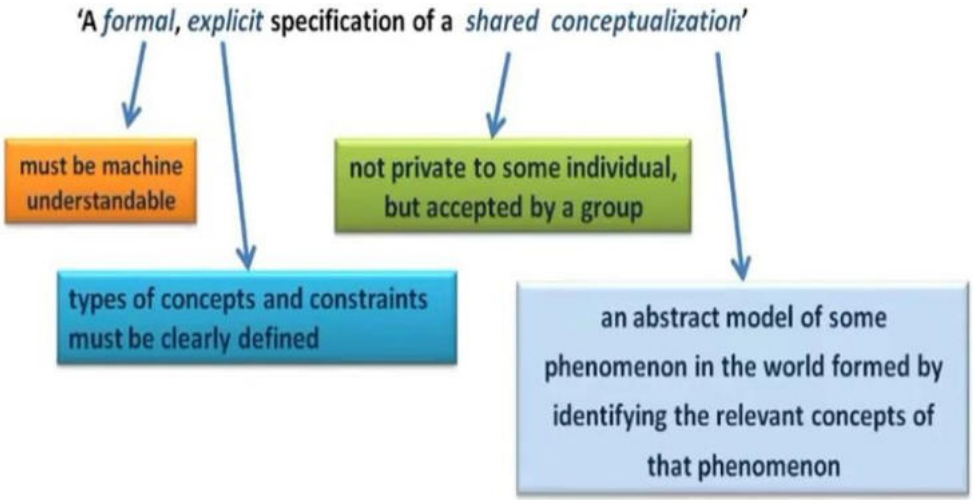
Ontological engineering- Categories and objects- Events- Mental objects and modal logic- Reasoning systems for categories- Reasoning with default information- Classical planning- Algorithms for classical planning -Heuristics for planning- Hierarchical planning -non-deterministic domains Time, schedule, and resources – Analysis

ONTOLOGICAL ENGINEERING

Ontology:

- It is a branch of metaphysics that studies the nature of being, the nature of existence.
- Ontologies enable more effective knowledge representation and reasoning in AI Systems.
- Ontologies provide a structured and formal way to represent knowledge.





- Conceptualization is an abstract simplified view of some selected part of the world, containing objects and concepts.

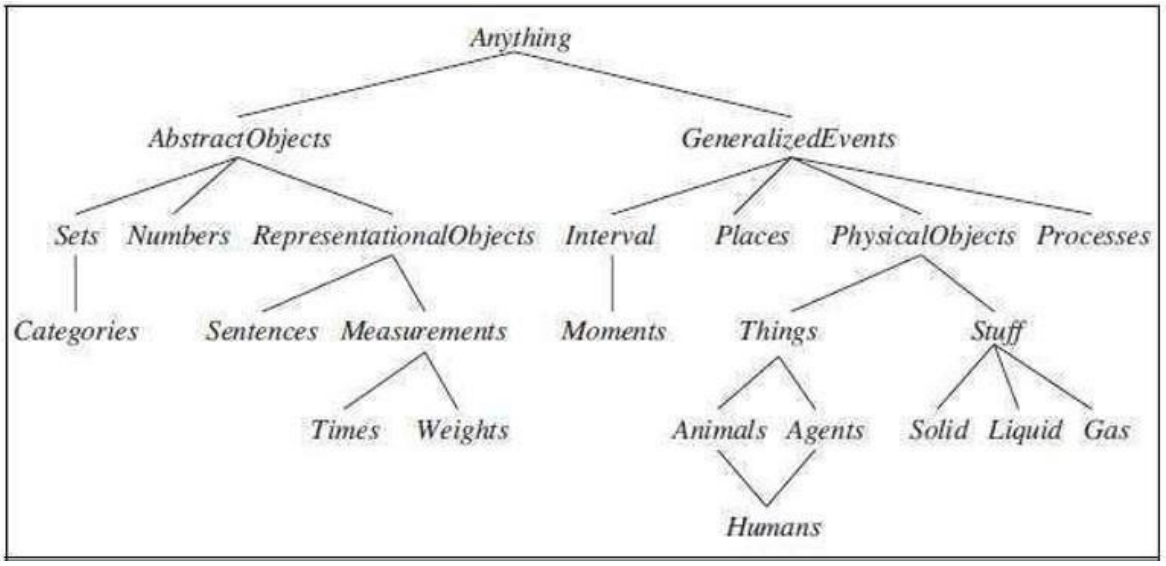
- An ontology is an explicit description of a domain:

  Concepts

  Properties and attributes of concepts

  Constraints on properties and attributes.

Ontology of the world:



Ontologies are expressed in the form of:

- Categories
- Measures
- Composite objects
- Time
- Changes
- Events
- Physical objects
- Beliefs

Ontological engineering:

- **It** is a discipline within artificial intelligence (AI) and computer science that focuses on creating and managing ontologies to enable more effective knowledge representation and reasoning in AI systems.
- Ontological engineering is a vital component of artificial intelligence that enables

  Better knowledge representation,

  Reasoning

  Interoperability across various applications and domains.

- It provides a structured foundation for AI systems to understand and work with complex knowledge, making them more intelligent and capable of supporting a wide range of tasks.
- Ontological Engineering is a set of tasks related to the development of ontologies for a particular domain. It involves

  Defining concepts in the domain(classes)

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

  Defining which attributes and properties can have constraints on their values

  Defining individuals and filling in the properties.

Key aspects of Ontological Engineering in AI:

1) Ontology Development: Ontological engineering involves designing and building ontologies for specific domains.

2) Knowledge Representation: Ontologies provide a structured and formal way to represent knowledge.

3) Semantic Web: Ontologies play a fundamental role in making web content machine-readable and understandable by both humans and machines.

4) Reasoning: Ontologies are used for inferencing and reasoning within AI systems.

5) Natural Language Processing: By mapping text to ontological concepts, AI systems can extract more meaningful information and improve language understanding.

6) Data Integration and Knowledge Management: Helps to structure and organize data, making it more accessible and useful for decision support and analytics.

CATEGORIES AND OBJECTS

CATEGORIES

- The organization of objects into categories is a vital part of knowledge representation.
- Although interaction with the world takes place at the level of individual objects, much reasoning takes place at the level of categories.
- For example, a shopper would normally have the goal of buying a basketball, rather than a particular **basketball such as BB9**.

There are two choices for representing categories in first-order logic:

1. Predicates – Basketball (b)
2. Object – We can reify the category as an object, Basketballs

- We could then say Member(b, Basketballs), which we will abbreviate as b Є Basketballs, to say that b is a member of the category of basketballs.
- We say Subset(Basketballs, Balls), abbreviated as Basketballs Balls, to say that Basketballs is a subcategory of Balls. We will use subcategory, subclass, and subset interchangeably.

Category Organization - Inheritance and taxonomy

- If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit, then we can infer that every apple is edible. We say that the individual apples inherit the property of edibility, in this case from their membership in the Food category.
- Subclass relations organize categories into a taxonomy, or taxonomic hierarchy

An object is a member of a category.

$BB9 \in Basketballs$

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.

$Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$

Relations between Categories:

Disjoint:

- Two or more categories are disjoint if they have no members in common.
- Example : Disjoint({Males, Females})
- A member of male category cannot be a female and a member of female category cannot be a male. We say that two or more categories are disjoint if they have no members in common.

Partition (Disjoint Exhaustive Decomposition)

- Disjoint Exhaustive Decomposition – A partition is a disjoint exhaustive decomposition
- A set of categories S constitutes an exhaustive decomposition of a category C, if all members of the set C are covered by the categories in S.
- Males and females constitute an exhaustive decomposition of the animals. A disjoint exhaustive decomposition is known as a partition.

$$Disjoint(\{Animals, Vegetables\})$$
$$ExhaustiveDecomposition(\{Americans, Canadians, Mexicans\}, NorthAmericans)$$
$$Partition(\{Males, Females\}, Animals) .$$

Physical Composition

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

  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) ⇒ PartOf (x, z) .

  PartOf (x, x) .

Therefore, we can conclude PartOf (Bucharest , Earth)

Composite object

Categories of composite objects are often characterized by structural relations among parts. For example, a biped has two legs attached to a body

- E.g. Biped(a) ⇒

$$(\exists l_1, l_2, b)(Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge Attached(l_1, b) \wedge Attached(l_2, b) \wedge \neq l_2 \wedge (\forall l_3)(Leg(l_3) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)))$$

Measurements

The values that we assign for the object properties are called measures.

- Eg: Height, Mass, Cost.
- If the line segment is called L1, we can write **Length(L1) = Inches(1.5) = Centimeters(3.81) .**

Conversion between units is done by equating multiples of one unit to another

- Centimeters(2.54 × d) = Inches(d)
- Diameter (Basketball 12) = Inches(9.5) .

Measurements are comparable

- For example, we might well believe that Norvig's exercises are tougher than Russell's, and that one scores less on tougher exercises:

e1 ∈ Exercises ∧ e2 ∈ Exercises ∧ Wrote(Norvig, e1) ∧ Wrote(Russell , e2) ⇒ Difficulty(e1) > Difficulty(e2) .

OBJECTS

The real world can be seen as consisting of primitive objects (e.g., atomic particles) and composite objects built from them. By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects individually.

Individuation

- A significant portion of reality that seems to defy any obvious individuation—division into distinct objects. We give this portion the generic name stuff.
- Any part of a butter-object is also a butter-object:

b Є Butter∧ PartOf (p, b) ⇒ p Є Butter .

Intrinsic and extrinsic properties

Intrinsic properties : They belong to the very substance of the object, rather to the object as a whole.

Eg: density, boiling point,flavor, color, ownership, and so on

Extrinsic properties : They are not retained under subdivision. They do depend on the size of a sample.Eg: weight, length, shape, and so on

- The category Stuff is the most general substance category, specifying no intrinsic properties.
- The category Thing is the most general discrete object category, specifying no extrinsic properties.

EVENTS

- Events are represented in two ways: 1) Situation Calculus 2) Event Calculus

- Situation calculus-it was designed to describe a world in which actions are discrete, instantaneous, and happen one at a time.
- Event calculus-based on points of time rather than on situations.

Example for Situation Calculus

- Consider a continuous action, such as filling a bathtub.
- Situation calculus can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens during the action.
- It also can't describe two actions happening at the same time—such as brushing one's teeth while waiting for the tub to fill.

Example for Event Calculus

- Events are described as instances of event categories.
- Event calculus, is based on points of time rather than on situations.

Events & fluents

- In the event calculus, fluents are reified. This means that they are not formalized by means of predicates but by means of functions.
- To assert that a fluent is actually true at some point in time 't', we use the predicate T , as in
  T (At(Shankar , Berkeley), t).

  T (f, t) Fluent f is true at time t
- The fluent **At(Shankar , Berkeley)** is an object that refers to the fact of Shankar being in Berkeley, but does not by itself say anything about whether it is true.
- The event E1 of Shankar flying from San Francisco to Washington, D.C. is described as

  E1 ∈ Flyings(Shankar , SF , DC) or

  E1 ∈ Flyings ∧ Flyer (E1, Shankar) ∧ Origin(E1, SF) ∧ Destination(E1,DC) .

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

Process or Liquid Event

- Ex: 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.
- Categories of events with this property are called process categories or liquid event.

Time intervals

- Two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration.

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

MENTAL EVENTS AND MENTAL OBJECTS

- Knowledge about one's own knowledge and reasoning processes is useful for controlling inference.
- For example, suppose Alice asks "what is the square root of 1764" and Bob replies "I don't know." If Alice insists "think harder," Bob should realize that with some more thought, this question can in fact be answered.
- Propositional attitudes are used to represent mental events – An agent can have toward mental objects: attitudes such as **Believes, Knows, Wants, Intends, and Informs**. The difficulty is that these attitudes do not behave like "normal" predicates.

For example:

The following sentence asserts that Lois knows that Superman can fly:

$$Knows(Lois, CanFly(Superman))$$

If "Superman is Clark Kent is true", then the inferential rules conclude that "Lois knows that Clark Kent can fly". This is an example of **Referential Transparency.** But in reality, Lois doesn't actually know that clark can fly.

$$Superman = Clark \; (Superman = Clark) \text{ and } (Knows(Lois, CanFly(Superman)) \vDash Knows(Lois, CanFly(Clark))$$

Referential Transparency

- An expression always evaluates to the same result in any context
- E.g. if agent knows that 2+2=4 and 4<5, then agent should know that 2+2<5 built into inferential rules of most formal logic languages

Referential Opacity

- Not referential transparent

- We want referential opacity for propositional attitudes because terms do matter and not all agents know which terms are co-referential
- Not directly possible in most formal logic languages (except Modal Logic)

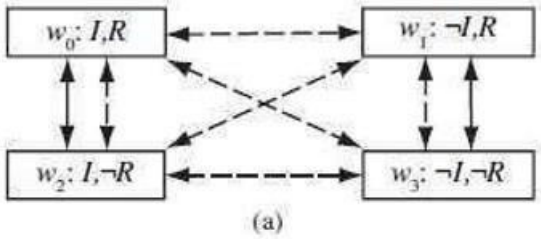
MODAL LOGIC

- Regular logic is concerned with single modality (the modality of truth), allowing us to express "P is true"
- In modal logic a model consists of a collection of possible worlds (instead of 1 true world)
- Modal logic includes modal operators that takes sentences (rather than terms) as arguments. E.g. "A knows P" is represented with notation $K_A P$ where K is the modal operator for knowledge, A is the agent(written as the subscript), and P is a sentence)
- Syntax of modal logic is the same as first-order logic, with the addition that sentences can also be formed with modal operators semantics of modal logic is more complicated.
- In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't.
- For example, we can say that, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows:

$$K_{Lois}\,[K_{Clark}\,Identity(Superman, Clark) \vee K_{Clark}\neg Identity(Superman, Clark)]$$

Example:

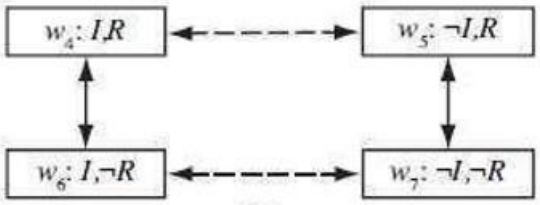
- The worlds are connected in a graph by accessibility relations (one relation for each modal operator).
- Consider the possible worlds with accessibility relations $K_{Superman}$ (solid arrows), K_{Lois} (Dotted arrows).
- The proposition R means "The weather report for tomorrow is rain"
- The proposition R means "Superman's secret identity is Clark Kent"



(a)

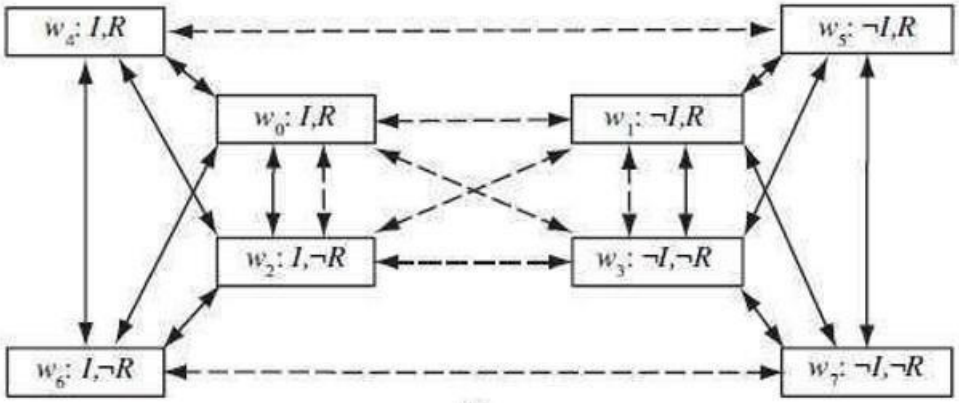
- **In the world A, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report.** So in w_0 the worlds w_0 and w_2 are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from

each other; she doesn't know anything about the report or if Clark is Superman.But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows I, or he knows ¬ I. Lois does not know which is the case, but either way she knows Superman knows.



(b)

- **In the world B, it is common knowledge that Lois has seen the weather report.** So in w4 she knows rain is predicted and in w6 she knows rain is not predicted Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows R or she knows ¬R.



(c)

- **In the world C, we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report.**

- We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds. Lois does know, so we don't need to add any arrows for her.

- In w0 Superman still knows I but not R, and now he does not know whether Lois knows R. From what Superman knows, he might be in w0 or w2, in which case Lois does not know whether R is true, or he could be in w4, in which case she knows R, or w6, in which Case she knows ¬R.

REASONING SYSTEMS FOR CATEGORIES

TWO WAYS FOR ORGANIZING AND REASONING WITH CATEGORIES

Semantic networks

- Semantic networks provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership

Description logics

- Description logics provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

SEMANTIC NETWORKS

- A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links.

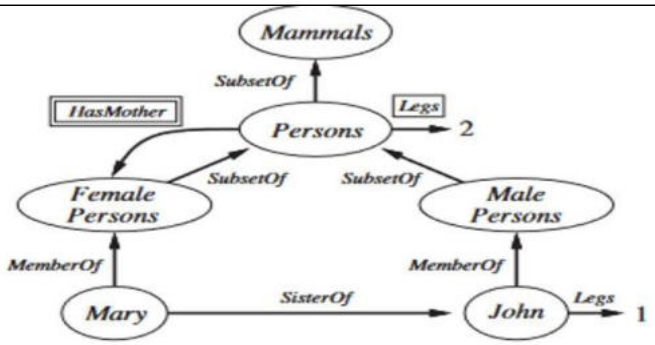


Figure 12.5 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

- The above semantic network has a MemberOf link between Mary and FemalePersons, corresponding to the logical assertion Mary ∈ FemalePersons ;

- similarly, the SisterOf linkbetween Mary and John corresponds to the assertion SisterOf (Mary, John). We can connect categories using SubsetOf links, and so on. It is such fun drawing bubbles and arrows that one can get carried away.

- For example, we know that persons have female persons as mothers, so can we draw a HasMother link from Persons to FemalePersons? The answer is no, because HasMother is a relation between a person and his or her mother, and categories do not have mothers.

- HasMother link is mentioned with double lined box.

This link asserts that

$$\forall x \;\; x \in Persons \;\Rightarrow\; [\forall y \;\; HasMother(x, y) \;\Rightarrow\; y \in FemalePersons] .$$

We might also want to assert that persons have two legs—that is,

$$\forall x \;\; x \in Persons \;\Rightarrow\; Legs(x, 2) .$$

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure 12.5 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information. The default semantics is enforced naturally by the inheritance algorithm, because it follows links upwards from the object itself (John in this case) and stops as soon as it finds a value. We say that the default is **overridden** by the more specific value. Notice that we could also override the default number of legs by creating a category of *OneLeggedPersons*, a subset of *Persons* of which *John* is a member.

We can retain a strictly logical semantics for the network if we say that the *Legs* assertion for *Persons* includes an exception for John:

$$\forall x \;\; x \in Persons \wedge x \neq John \;\Rightarrow\; Legs(x, 2) \; .$$

For a *fixed* network, this is semantically adequate but will be much less concise than the network notation itself if there are lots of exceptions. For a network that will be updated with more assertions, however, such an approach fails—we really want to say that any persons as yet unknown with one leg are exceptions too. Section 12.6 goes into more depth on this issue and on default reasoning in general.

DESCRIPTION LOGIC:

- Description logics are notations that are designed to make it easier to describe definitions and properties of categories.
- The principal inference tasks for description logics are **subsumption** (checking if one category is a subset of another by comparing their definitions)
- **Classification** – Checking whether an object belongs to a category.
- Some systems also include **consistency** of a category definition—whether the membership criteria are logically satisfiable

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

Figure 12.7 The syntax of descriptions in a subset of the CLASSIC language.

Any description in CLASSIC can be translated into an equivalent first-order sentence, but some descriptions are more straightforward in CLASSIC .

Example 1:

Bachelors are unmarried adult males

$$Bachelor = And(Unmarried, Adult, Male) .$$

The equivalent in first-order logic would be

$$Bachelor(x) \Leftrightarrow Unmarried(x) \wedge Adult(x) \wedge Male(x) .$$

Example 2:

For example, to describe the set of men with at least three sons who are all unemployed and married to doctors, and at most two daughters who are all professors in physics or math departments, we would use

$$And(Man, AtLeast(3, Son), AtMost(2, Daughter),$$
$$All(Son, And(Unemployed, Married, All(Spouse, Doctor))),$$
$$All(Daughter, And(Professor, Fills(Department, Physics, Math)))) .$$

REASONING WITH DEFAULT INFORMATION

The reasoning is the mental process of deriving logical conclusion and making predictions from available knowledge, facts, and beliefs. Or we can say, "**Reasoning is a way to infer facts from existing data.**" It is a general process of thinking rationally, to find valid conclusions.

In artificial intelligence, the reasoning is essential so that the machine can also think rationally as a human brain, and can perform like a human.

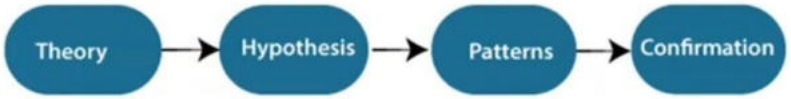
Types of Reasoning

- Deductive reasoning
- Inductive reasoning
- Abductive reasoning
- Common Sense Reasoning
- Monotonic Reasoning
- Non-monotonic Reasoning

1. Deductive reasoning:

- Deductive reasoning is deducing new information from logically related known information. It is the form of valid reasoning, which means the argument's conclusion must be true when the premises are true.

Example:
Premise-1: All the human eats veggies
Premise-2: Suresh is human.
Conclusion: Suresh eats veggies.
The general process of deductive reasoning is given below:

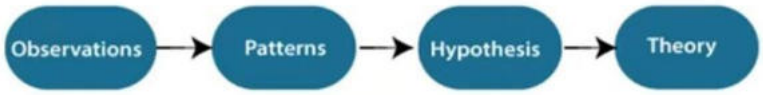


2. Inductive Reasoning:

- Inductive reasoning is a form of reasoning to arrive at a conclusion using limited sets of facts by the process of generalization. It starts with the series of specific facts or data and reaches to a general statement or conclusion.

Inductive Reasoning

Premise: All of the pigeons we have seen in the zoo are white.
Conclusion: Therefore, we can expect all the pigeons to be white.



3. Abductive reasoning:

- Abductive reasoning is a form of logical reasoning which starts with single or multiple observations then seeks to find the most likely explanation or conclusion for the observation.

Example:
Implication: Cricket ground is wet if it is raining
Axiom: Cricket ground is wet.
Conclusion It is raining.

4. Common Sense Reasoning

- Common sense reasoning is an informal form of reasoning, which can be gained through experiences.

Example:
One person can be at one place at a time.
If I put my hand in a fire, then it will burn.
The above two statements are the examples of common sense reasoning which a human mind can easily understand and assume.

5. Monotonic Reasoning:

- In monotonic reasoning, once the conclusion is taken, then it will remain the same even if we add some other information to existing information in our knowledge base. In monotonic reasoning, adding knowledge does not decrease the set of prepositions that can be derived.

Example:
Earth revolves around the Sun.
It is a true fact, and it cannot be changed even if we add another sentence in knowledge base like, "The moon revolves around the earth" Or "Earth is not round," etc.

6. Non-monotonic Reasoning

- In Non-monotonic reasoning, some conclusions may be invalidated if we add some more information to our knowledge base.
- Logic will be said as non-monotonic if some conclusions can be invalidated by adding more knowledge into our knowledge base.

Example: Let suppose the knowledge base contains the following knowledge:
Birds can fly
Penguins cannot fly
Pitty is a bird

So from the above sentences, we can conclude that **Pitty can fly**. However, if we add one another sentence into knowledge base "**Pitty is a penguin**", which concludes "**Pitty cannot fly**", so it invalidates the above conclusion.

CIRCUMSCRIPTION

- The idea is to specify particular predicates that are assumed to be "as false as possible"—that is, false for every object except those for which they are known to be true.
- For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say Abnormal 1(x), and write

  Bird (x) ∧ ¬Abnormal 1(x) ⇒ Flies(x) .

- If we say that **Abnormal 1 is to be circumscribed**, a circumscriptive reasoner is entitled to assume ¬Abnormal 1(x) unless Abnormal 1(x) is known to be true.
- This allows the conclusion Flies(Penguin) to be drawn from the premise Bird (Penguin), but the conclusion no longer holds if Abnormal 1(Penguin) is asserted.

MODEL PREFERENCE

- Circumscription can be viewed as an example of a model preference logic
- A sentence is entailed (with default status) if it is true in all preferred models of the KB,as opposed to the requirement of truth in all models in classical logic.
- Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

$$Republican(Nixon) \wedge Quaker(Nixon) .$$
$$Republican(x) \wedge \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x) .$$
$$Quaker(x) \wedge \neg Abnormal_3(x) \Rightarrow Pacifist(x) .$$

- If we circumscribe Abnormal 2 and Abnormal 3, there are two preferred models:
  1) Abnormal 2(Nixon) and Pacifist(Nixon) hold
  2) Abnormal 3(Nixon) and ¬Pacifist(Nixon) hold
- If we wish, in addition, **to assert that religious beliefs take precedence over political beliefs**, We can use a formalism called prioritized circumscription to give preference to models where **Abnormal 3 is minimized.**

DEFAULT LOGIC

Default logic is a formalism in which default rules can be written to generate contingent, non-monotonic conclusions. A default rule looks like this:

$$Bird(x) : Flies(x)/Flies(x) .$$

This rule means that if Bird (x) is true, and if Flies(x) is consistent with the knowledge base,then Flies(x) may be concluded by default. In general, a default rule has the form

$$P : J_1, \ldots, J_n/C$$

where P is called the prerequisite, C is the conclusion, and Ji are the justifications

The Richard Nixon example can be represented in default logic with one fact and two default rules:

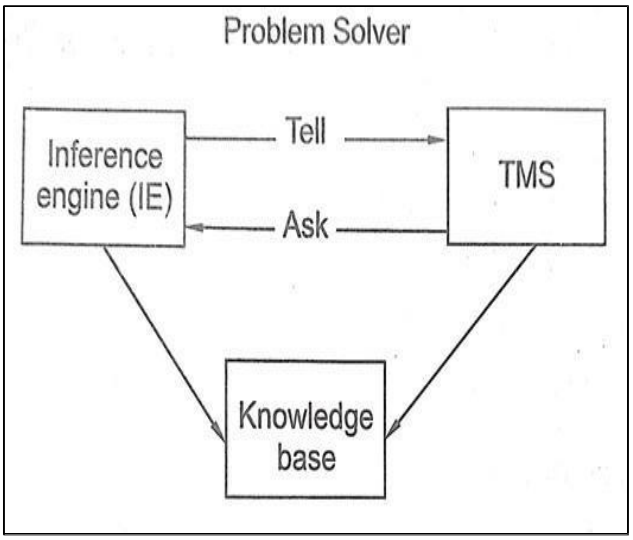
$$Republican(Nixon) \wedge Quaker(Nixon) .$$
$$Republican(x) : \neg Pacifist(x)/\neg Pacifist(x) .$$
$$Quaker(x) : Pacifist(x)/Pacifist(x) .$$

- To interpret what the default rules mean, we define the notion of an extension.
- Extension S consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from S and the justifications of every default conclusion in S are consistent with S.
- As in the case of the preferred models in circumscription, we have two possible extensions for the Nixon diamond: one wherein he is a pacifist and one wherein he is not. Prioritized schemes exist in which some default rules can be given precedence over others, allowing some ambiguities to be resolved.

TRUTH MAINTENANCE SYSTEMS

- Many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain.
- Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called belief revision.
- Suppose that a knowledge base KB contains a sentence P—perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and we want to execute TELL(KB, ¬P). To avoid creating a contradiction, we must first execute RETRACT(KB, P).
- Problems arise, however, if any additional sentences were inferred from P and asserted in the KB. For example, the implication P ⇒ Q might have been used to add Q.

- The obvious "solution"—retracting all sentences inferred from P—fails because such sentences may have other justifications besides P. For example, if R and R ⇒ Q are also in the KB, then Q does not have to be removed after all.



- One simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from P1 to Pn. When the call RETRACT (KB, Pi) is made, the system reverts to the state just before Pi was added, thereby removing both Pi and any inferences that were derived from Pi. The sentences Pi+1 through Pn can then be added again.

Justification Truth Maintenance System

- In a JTMS, each sentence in the knowledge base is annotated with a justification consisting of the set of sentences from which it was inferred.
- For example, if the knowledge base already contains P ⇒ Q, then TELL(P) will cause Q to be added with the justification {P, P ⇒ Q}. In general, a sentence can have any number of justifications. Justifications make retraction efficient.
- Given the call RETRACT(P), the JTMS will delete exactly those sentences for which P is a member of every justification. So, if a sentence Q had the single justification {P, P ⇒ Q}, it would be removed; if it had the additional justification {P, P ∨ R ⇒ Q}, it would still be removed.

Assumption-based Truth maintenance system

Assumption based truth maintenance system	Justification based truth maintenance system
An ATMS represents all the states that have ever been considered at the same time	In a JTMS, the maintenance of justifications allows you to move quickly from one state to another by making a few retractions and assertions, but at any time only one state is represented
Each sentence has a label that consists of a set of assumption sets. The sentence holds just in those cases in which all the assumptions in one of the assumption sets hold.	JTMS simply labels each sentence as being in or out,

Explanations and Assumptions in Truth Management System

- Truth maintenance systems also provide a mechanism for generating explanations.
- An **explanation** of a sentence P is a set of sentences E such that E entails P.
- If the sentences in E are already known to be true, then E simply provides a sufficient basis for proving that P must be the case.
- But explanations can also **include assumptions**— sentences that are not known to be true, but would suffice to prove P if they were true.
- Computational Complexity– NP hard

CLASSICAL PLANNING

What is planning in AI?

- The planning in Artificial Intelligence is about the decision making tasks performed by the robots or computer programs to achieve a specific goal.
- The execution of planning is about choosing a sequence of actions with a high likelihood to complete the specific task.

Planning Domain Definition Language:

- PDDL, the Planning Domain Definition Language in which a state of the world is represented by a collection of variables that allows us to express all Actions with one action schema.
- PDDL describes the four things we need to define a search problem:

  The initial state
  The actions that are available in a state
  the result of applying an action
  the goal test.

- Each state is represented as a conjunction of fluents that are ground, functionless atoms.
- Eg: A state in a package delivery problem might be At(Truck 1, Melbourne) ∧ At(Truck 2, Sydney).
- Actions are described by a **set of action schemas** that implicitly define the ACTIONS(s) and RESULT(s, a) functions needed to do a problem-solving search.
- Classical planning concentrates on problems where most actions leave most things unchanged.

Action schema

- A set of ground (variable-free) actions can be represented by a single action schema. The schema is a lifted representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic.
- The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect
- For example, here is an action schema for flying a plane from one location to another:

$$
\begin{array}{l}
Action(Fly(p, from, to), \\
\quad \text{PRECOND}: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\
\quad \text{EFFECT}: \neg At(p, from) \wedge At(p, to))
\end{array}
$$

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

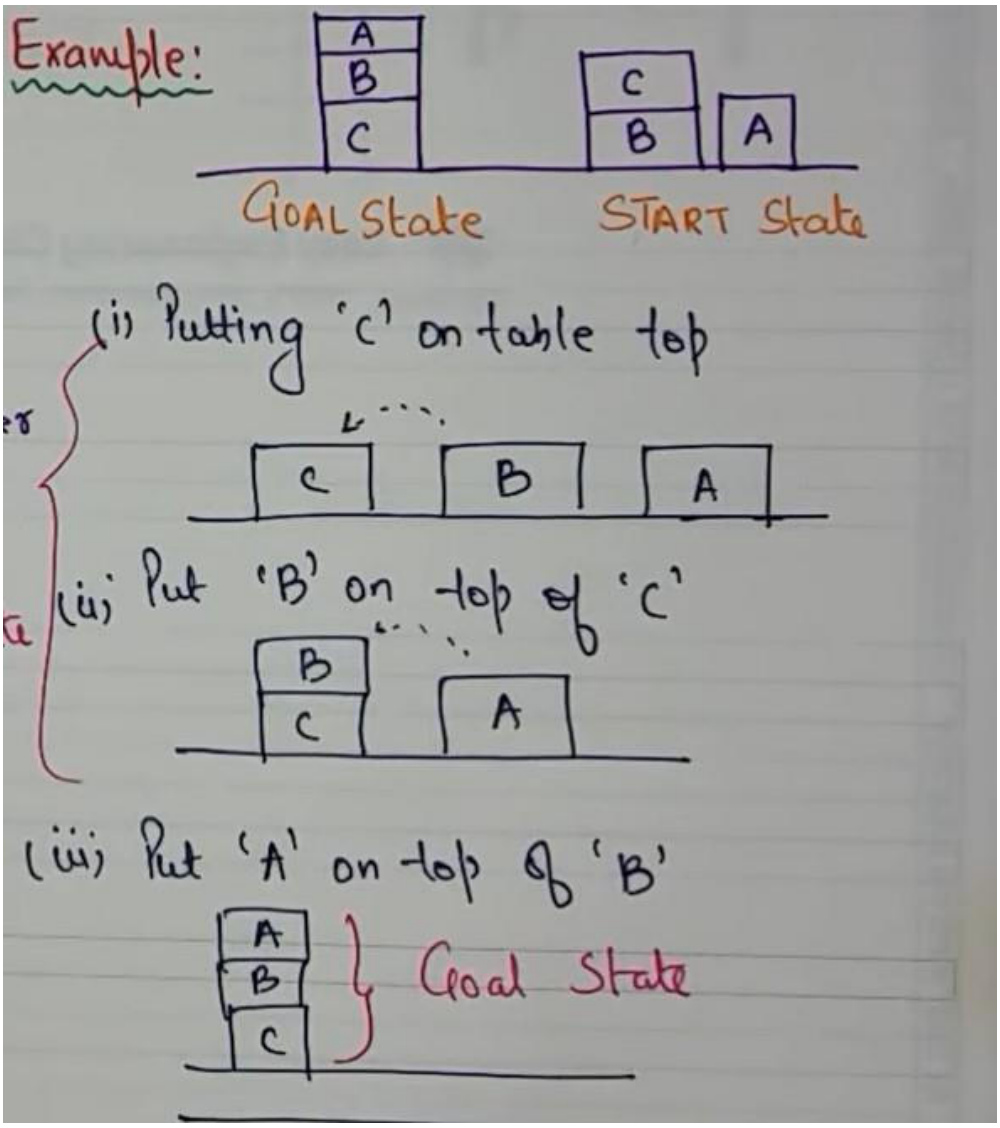
Figure 10.1 A PDDL description of an air cargo transportation planning problem.

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
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

- This domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another.
- A robot arm can pick up a block and move it to another position, either on the table or on top of another block.
- The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stack of what other blocks.s of blocks, specified in terms of what blocks are on top



$Init(On(A, Table) \land On(B, Table) \land On(C, A)$
$\land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
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 $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

Components of Planning System

- The planning consists of following important steps:

1. Choose the best rule for applying the next rule based on the best available heuristics.

2. Apply the chosen rule for computing the new problem state.
3. Detect when a solution has been found.
4. Detect dead ends so that they can be abandoned and the system's effort is directed in more fruitful directions.
5. Detect when an almost correct solution has been found.

Goal stack planning:

- This is one of the most important planning algorithms, which is specifically used by STRIPS.
- The stack is used in an algorithm to hold the action and satisfy the goal. A knowledge base is used to hold the current state, actions.
- Goal stack is similar to a node in a search tree, where the branches are created if there is a choice of an action.

The important steps of the algorithm are as stated below:

- Start by pushing the original goal on the stack. Repeat this until the stack becomes empty. If stack top is a compound goal, then push its unsatisfied subgoals on the stack.
- If stack top is a single unsatisfied goal then, replace it by an action and push the action's precondition on the stack to satisfy the condition.
- If stack top is an action, pop it from the stack, execute it and change the knowledge base by the effects of the action.
- If stack top is a satisfied goal, pop it from the stack.

Non-linear planning

- This planning is used to set a goal stack and is included in the search space of all possible subgoal orderings. It handles the goal interactions by interleaving method.

Advantage of non-Linear planning

- Non-linear planning may be an optimal solution with respect to plan length (depending on search strategy used).

Disadvantages of Nonlinear planning

- It takes larger search space, since all possible goal orderings are taken into consideration.
- Complex algorithm to understand.

Algorithm

1. Choose a goal 'g' from the goalset

2. If 'g' does not match the state, then
    - Choose an operator 'o' whose add-list matches goal g
    - Push 'o' on the opstack
    - Add the preconditions of 'o' to the goalset
3. While all preconditions of operator on top of opstack are met in state
    - Pop operator o from top of opstack
    - state = apply(o, state)
    - plan = [plan; o]

ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

The two planners are:

1. Forward (progression) state-space search/Forward Planner/Progression Planner
2. Backward (regression) relevant-states search/Backward Planner/Regression Planner

1. Forward (progression) state-space search/Forward Planner/Progression Planner

- A forward planner, or progression planner starts at the initial state, and applies actions in an effort to find a path to the goal state for many years, forward planning was thought to be too inefficient to be practical.
- Forward search is **prone to exploring irrelevant actions**.
- Example: Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B.
- There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo.
- Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport).
- On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000 41 node.

2. Backward(Regression)state-space search/Backward Planner/Regression Planner

- A backward planner, or regression planner, starts at the goal state, and works backwards from that to try to find a path to the state.
- For many decades, backward planners were thought to be inherently more efficient than forward planners because they only consider actions that are relevant to the goal the reason for their presumed superiority was they result in smaller branching factors because they focus on relevant states.
- The PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it.
- Example: Suppose the goal is to deliver a specific piece of cargo to SFO: At(C2, SFO).
- That suggests the action **Unload(C2, p' , SFO)**:

$$\begin{array}{l} Action(Unload(C_2, p', SFO), \\ \quad \text{PRECOND: } In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO) \\ \quad \text{EFFECT: } At(C_2, SFO) \wedge \neg In(C_2, p') . \end{array}$$

Example: Consider the goal Own(0136042597), given an initial state with 10 Billion ISBN's and a single action schema.

$$A = Action(Buy(i), \text{PRECOND: } ISBN(i), \text{EFFECT: } Own(i))$$

- **Forward search without heuristic** would start enumerating the 10 million ground BUY actions.
- But, with Backward search, we would unify the goal Own(0136042597)with the (standardized) Own(i'), yielding the substitution θ = { i' / 0136042597}. Then, we would regress over the action Subst(θ, A') to yield the predecessor state description ISBN(0136042597).

Heuristics for Planning:

- A heuristic functions estimates the distance from a states to the goal and that if we can derive an admissible heuristic for this distance—one that does not overestimate—then we can use A∗ search to find optimal solutions.
- An admissible heuristic can be derived by defining a relaxed problem that is easier to solve.
- The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.
- It appears now that the **best planners are forward planners,** the essential reason is that very good general-purpose, and problem-specific, heuristics have been found for forward planners, but similarly good heuristics have not been found for backwards planners.

- For example, one good general-purpose forward planning heuristic is to can relax this problem to make it easier:

  By adding more edges to the graph,making it strictly easier to find a path.

  By **grouping multiple nodes together**, forming an abstraction of the state space that has fewer states, and thus is easier to search.

1. Add more edges to the graph ---ignore preconditions heuristic

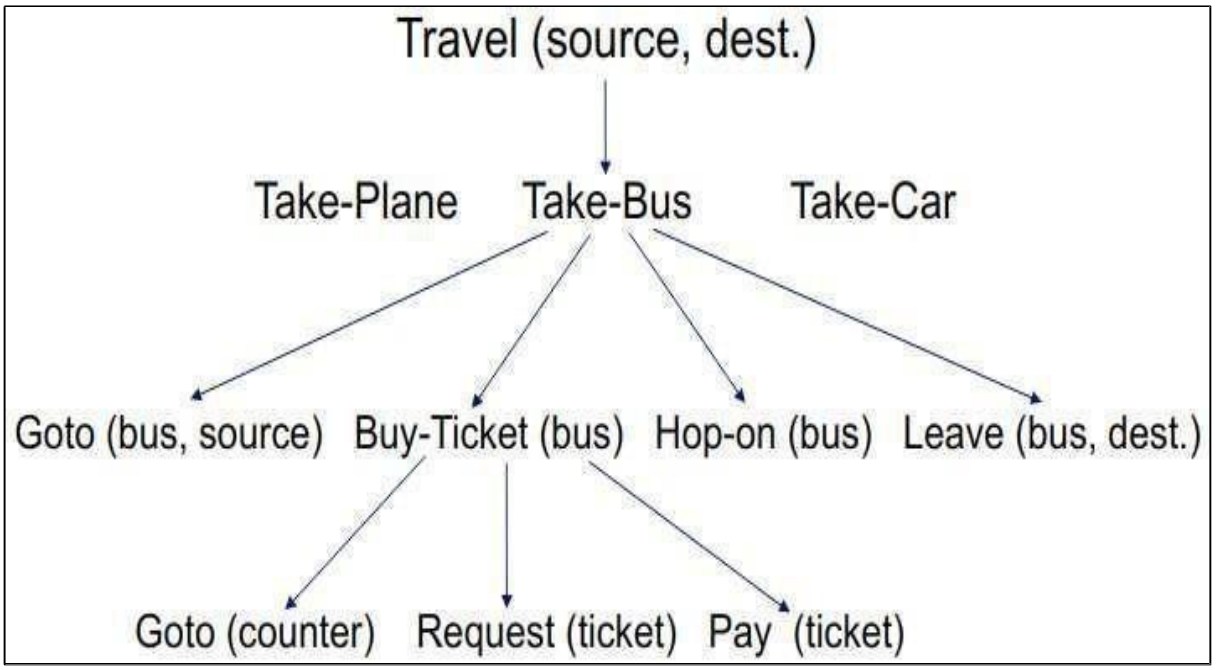
- First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal.
- The ignore preconditions heuristic, drops all preconditions from actions.
- This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others.

2. Ignore delete lists

- Assume for a moment that all goals and preconditions contain only positive literals .We want to create a **relaxed version of the original problem** that will be easier to solve, and where the length of the solution will serve as a good heuristic.
- We can do that by removing the delete lists from all actions (i.e., removing all negative literals from effects). That makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action
- **Relaxations that decrease the number of states** by forming a state abstraction—a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

HIERARCHICAL PLANNING

- Hierarchical planning is a planning method based on Hierarchical Task Network (HTN) or HTN planning.
- HTN planning is often formulated with a single "top level" action called Act, where the aim is to find an implementation of Act that achieves the goal.
- In HTN planning, the initial plan is viewed as a very high level description of what is to be done. This plan is refined by applying decomposition actions. Each action decomposition reduces a higher level action to a partially ordered set of lower level actions. This decomposition continues until only the primitive actions remain in the plan.
- Consider the example of a **hierarchical plan to travel from a certain source to destination**,



- In the above hierarchical planner diagram, suppose we are Travelling from source "Mumbai" to Destination "Goa".
- Then, you can plan how to travel: whether by Plane, Bus, or a Car. Suppose, you choose to travel by "Bus".Then, "Take-Bus" plan can be further broken down into set of actions like: **Goto Mumbai – Bus stop, Buy-Ticket for Bus, Hop-on Bus, & Leave for Goa.**
- Now, the four actions in previous point can be individually broken down. Take, "By-Ticket for Bus".
- It can be decomposed into: **Go to Bus stop counter, Request Ticket & Pay for Ticket.**
- Thus, each of these actions can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.
- Here, we also use the concept of "One level Partial Order Planner" where say, if you plan to take a trip, you need to decide a location first. This can be done by One Level Planner as:
- Switch on computer > Start web browser > Open Redbus website > Select date > Select class > Select bus > …

Advantages of Hierarchical Planning:

- The key benefit of hierarchical structure is that, at each level of the hierarchy, plan is reduced to a small number of activities at the next lower level,so the computational cost of finding the correct way to arrange those activities for the current problem is small.
- HTN methods can create the very large plans required by many real-world applications.
- Hierarchical structure makes it easy to fix problems in case things go wrong.
- For complex problems hierarchical planning is much more efficient than single level planning.

Disadvantages of Hierarchical Planning:

- Many of the HTN planners require a Deterministic environment.
- Many of the HTN planners are unable to handle uncertain outcomes of actions.

Searching for primitive solutions:

- Repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal.
- One possible implementation based on breadth-first tree search. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps.
- It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

Searching for abstract solutions:

- Write precondition-effect descriptions of the HLAs,just as we write down what the primitive actions do. From the descriptions, it ought to be easy to prove that the high-level plan achieves the goal.
- Every high-level plan that "claims" to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal.
- This property has been called the downward refinement property for HLA descriptions.

Planning and acting in non deterministic domains

- Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view

$$Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table))$$
$$Goal(Color(Chair, c) \land Color(Table, c))$$

- There are two actions: removing the lid from a paint can and painting an object using the paint from an open can

$$
\begin{aligned}
&Action(RemoveLid(can),\\
&\quad \text{PRECOND: } Can(can)\\
&\quad \text{EFFECT: } Open(can))\\
&Action(Paint(x, can),\\
&\quad \text{PRECOND: } Object(x) \wedge Can(can) \wedge Color(can, c) \wedge Open(can)\\
&\quad \text{EFFECT: } Color(x, c))
\end{aligned}
$$

$$
\begin{aligned}
&Percept(Color(x, c),\\
&\quad \text{PRECOND: } Object(x) \wedge InView(x)\\
&Percept(Color(can, c),\\
&\quad \text{PRECOND: } Can(can) \wedge InView(can) \wedge Open(can)
\end{aligned}
$$

Three Planning Approaches

- Sensorless planning (also called conformant planning).
  - Handles domains where the state of the world is not fully known.
  - Comes up with plans that work in all possible cases.
- Contingent planning (also called conditional planning).
  - Handles domains where the effects of an action are not deterministic.
  - Approach: plan ahead for different possible results of each action.
- Execution monitoring and online replanning.
  - Handles domains where the effects of an action are not deterministic, or where, more generally, things in the world may unexpectedly change.
  - While executing the plan, before performing each action, monitor the environment.
  - If the enviroment is different than expected, replan.

Sensorless (Conformant) Planning

- Example:
  - You have a wall made of bricks.
  - You have a can of white paint.
  - Action: *Paint(brick)*, effect: *Color(brick, white)*.
  - Goal: every brick should be painted white.
- Suppose the world is not fully observable.
  - We actually cannot observe the color of a brick.
- Suppose that the world is deterministic.
  - The effects of an action are known in advance.
- What plan would ensure achieving the goal?
  - Paint all bricks, regardless of their initial color (which we don't know anyway).
  - It may be overkill, since some bricks may already be white, but it is the only plan that guarantees achieving the goal.

Sensorless (Conformant) Planning

- Limitations:
  - While there are a few domains simple enough to allow for sensorless planning:
  - Many real world domains are too complicated for this approach, and you can't come up with plans that work regardless of what the state of the world is.

Contingent Planning

- Example: this was the definition of actions for the blocks world:

```
Action(Move(block, from, to),
   PRECOND: On(block, from) AND Clear(block) AND Clear(to)
   EFFECT: On(block, to) AND NOT(On(block, from)) AND
            Clear(from) AND NOT(Clear(to))

Action(MoveToTable(block, from),
   PRECOND: On(block, from) AND Clear(block)
   EFFECT: On(block, Table) AND NOT(On(block, from)) AND Clear(from)
```

Contingent Planning

- We also need to allow plans to have if statements.
- Example:

move(A, B, C)
if *on(A, C)* **then** *done*
else:
  move(A, table, C)
  if *on(A, C)* **then** *done*
 else:
   move(A, table, C)
   if *on(A, C)* **then** *done* ...

Online Replanning

- In online replanning, we do not have to modify the action definitions.
- The plan we make is the same as in the deterministic case:

move(A, B, C)

- When executing the plan:
- Before executing *move(A, B, C)*, we check if the state of the world is as expected.
- If so, we execute the action.
- After we execute the action, we check if the goal is indeed achieved.
  - If not, we replan.

Online Replanning

- Before executing the next action, the agent monitors the enviroment.
- There are different choices as to exactly what to monitor.
  - **Action monitoring:** verify that the preconditions of the next action still hold.
  - **Plan monitoring:** verify that the remaining plan will still succeed, given the current state of the world.
  - **Goal monitoring:** check to see if, given the current state of the world, there is a better plan to follow.
- Action monitoring is the most simple and efficient.
- Goal monitoring can take advantage of unexpected changes that may make it easier to achieve the goal.

Time, schedules, and resources

- The PDDL language allows events (actions) and ordering of events, but not time *duration*
- In real-life planning, we must take duration, delays, etc. into account (not just ordering)
- *Job shop scheduling*:
  - The problem is to complete a set of jobs
  - Each job consists of a set of actions, with given duration and resource requirements
  - Determine a schedule that minimizes total time *(makespan)* needed while respecting resource constraints
- Must extend representation language to express duration and resource constraints

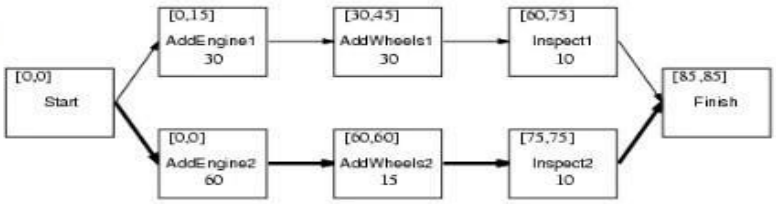
Example - Assembling two cars

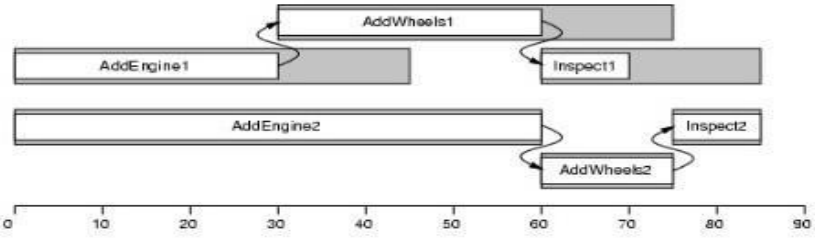
- ***Jobs***(*{AddEngine1 < AddWheels1 < Inspect1}, {AddEngine2 < AddWheels2 < Inspect2}*)
- ***Resources***(*EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500)*)
- ***Action***(*AddEngine1,* DURATION:30, USE: *EngineHoists(1)*)
- ***Action***(*AddEngine2,* DURATION:60, USE: *EngineHoists(1)*)
- ***Action***(*AddWheels1,* DURATION:30, CONSUME: *LugNuts(20),* USE: *WheelStations(1)*)
- ***Action***(*AddWheels2,* DURATION:15, CONSUME: *LugNuts(20),* USE: *WheelStations(1)*)
- ***Action***(*InspectI,* DURATION:10, USE: *Inspectors(1)*)

Scheduling - No resource constraints

- Partial order plan produced by e.g. POP
- To create a *schedule,* we must place actions on a timeline
- Can use *critical path* method (CPM): the longest path, no slack – determines total duration
- Shortest duration schedule, given partial-order plan:

  85 minutes





Planning and scheduling

- The approach shown here is common in real-world AI applications for manufacturing scheduling, airline scheduling, etc. :
  - √ First generate partial order plan without timing information (*planning*)
  - √ Then use separate algorithm to find optimal (or satisfactory) time behavior (*scheduling*)
- In some cases it may be better to *interleave* planning and scheduling, e.g. to consider temporal constraints already at the planning stage

UNIT-5

Uncertain knowledge and Learning

- Knowledge Reasoning : work with facts/assertions; develop rules of logical inference
- Planning: work with applicability/effects of actions; develop searches for actions which achieve goals/avert disasters.
- Expert Systems: develop by hand a set of rules for examining inputs, updating internal states and generating outputs
- Learning approach: use probabilistic models to tune performance based on many data examples.
- Probabilistic AI: emphasis on noisy measurements, approximation in hard cases, learning, algorithmic issues.
    - logical assertions ⇒ probability distributions
    - logical inference ⇒ conditional probability distributions
    - logical operators ⇒ probabilistic generative models

Probabilistic reasoning

Causes of uncertainty:

Following are some leading causes of uncertainty to occur in the real world.

- Information occurred from unreliable sources.
- Experimental Errors
- Equipment fault
- Temperature variation
- Climate change.

Probabilistic reasoning is a way of knowledge representation where we apply the concept of probability to indicate the uncertainty in knowledge. In probabilistic reasoning, we combine probability theory with logic to handle the uncertainty.

We use probability in probabilistic reasoning because it provides a way to handle the uncertainty that is the result of someone's laziness and ignorance.

In the real world, there are lots of scenarios, where the certainty of something is not confirmed, such as "It will rain today," "behavior of someone for some situations," "A match between two teams or two players." These are probable sentences for which we can assume that it will happen but not sure about it, so here we use probabilistic reasoning.

Probability: Probability can be defined as a chance that an uncertain event will occur. It is the numerical measure of the likelihood that an event will occur. The value of probability always remains between 0 and 1 that represent ideal uncertainties.

$0 \leq P(A) \leq 1$, where P(A) is the probability of an event A.

P(A) = 0, indicates total uncertainty in an event A.

P(A) =1, indicates total certainty in an event A.

We can find the probability of an uncertain event by using the below formula.
P(¬A) = probability of a not happening event.
P(¬A) + P(A) = 1.
Event: Each possible outcome of a variable is called an event.
Sample space: The collection of all possible events is called sample space.
Random variables: Random variables are used to represent the events and objects in the real world.
Prior probability: The prior probability of an event is probability computed before observing new information.
Posterior Probability: The probability that is calculated after all evidence or information has taken into account. It is a combination of prior probability and new information.
Conditional probability:
Conditional probability is a probability of occurring an event when another event has already happened.

Let's suppose, we want to calculate the event A when event B has already occurred, "the probability of A under the conditions of B", it can be written as:

Where P(A⋀B)= Joint probability of a and B
P(B)= Marginal probability of B.

If the probability of A is given and we need to find the probability of B, then it will be given as:

It can be explained by using the below Venn diagram, where B is occurred event, so sample space will be reduced to set B, and now we can only calculate event A when event B is already occurred by dividing the probability of P(A⋀B) by P(B).
Example:
In a class, there are 70% of the students who like English and 40% of the students who likes English and mathematics, and then what is the percent of students those who like English also like mathematics?
Solution:
Let, A is an event that a student likes Mathematics
B is an event that a student likes English.
Hence, 57% are the students who like English also like Mathematics.

Why Reason Probabilistically?

- In many problem domains it isn't possible to create complete, consistent models of the world. Therefore agents (and people) must act in uncertain worlds (which the real world is).
- Want an agent to make rational decisions even when there is not enough information to prove that an action will work.
- Some of the reasons for reasoning under uncertainty:
  - **True uncertainty**. E.g., flipping a coin.
  - **Theoretical ignorance**. There is no complete theory which is known about the problem domain. E.g., medical diagnosis.
  - **Laziness**. The space of relevant factors is very large, and would require too much work to list the complete set of antecedents and consequents. Furthermore, it would be too hard to use the enormous rules that resulted.
  - **Practical ignorance**. Uncertain about a particular individual in the domain because all of the information necessary for that individual has not been collected.
- Probability theory will serve as the formal language for representing and reasoning with uncertain knowledge.

Bayes' theorem:

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, which is fundamental to Bayesian statistics

It is a way to calculate the value of P(B|A) with the knowledge of P(A|B).

Bayes' theorem allows updating the probability prediction of an event by observing new information of the real world.

Example: If cancer corresponds to one's age then by using Bayes' theorem, we can determine the probability of cancer more accurately with the help of age.

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:

1. $P(A \wedge B) = P(A|B)\, P(B)$ or

Similarly, the probability of event B with known event A:

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

Equating right hand side of both the equations, 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:

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

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 neck?

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

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

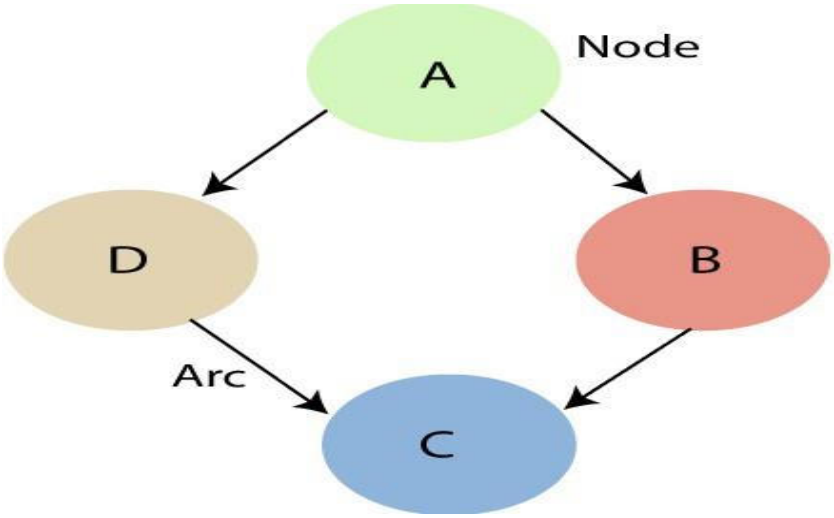
Hence, we can assume that 1 patient out of 750 patients has meningitis disease with a stiff neck.

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

- **Directed Acyclic Graph**
- **Table of conditional probabilities.**

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

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



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

The Bayesian network has mainly two components:Causal Component

- **Actual numbers**

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

Representing Belief about Propositions

- Rather than reasoning about the truth or falsity of a proposition, reason about the belief that a proposition or event is true or false
- For each primitive proposition or event, attach a **degree of belief** to the sentence
- Use **probability theory** as a formal means of manipulating degrees of belief
- Given a proposition, A, assign a probability, P(A), such that 0 <= P(A) <= 1, where if A is true, P(A)=1, and if A is false, P(A)=0. Proposition A must be either true or false, but P(A) summarizes our degree of belief in A being true/false.
  - Examples
  - P(Weather=Sunny) = 0.7 means that we believe that the weather will be Sunny with 70% certainty. In this case Weather is a random variable that can take on values in a domain such as {Sunny, Rainy, Snowy, Cloudy}.
  - P(Cavity=True) = 0.05 means that we believe there is a 5% chance that a person has a cavity. Cavity is a Boolean random variable since it can take on possible values *True* and *False*.
  - Example: P(A=a ^ B=b) = P(A=a, B=b) = 0.2, where A=My_Mood, a=happy, B=Weather, and b=rainy, means that there is a 20% chance that when it's raining my mood is happy.

- We will assume that in a given problem domain, the programmer and expert identify all of the relevant propositional variables that are needed to reason about the domain.
- Each of these will be represented as a **random variable**, i.e., a variable that can take on values from a set of mutually exclusive and exhaustive values called the **sample space** or **partition** of the random variable. Usually this will mean a sample space {*True*, *False*}.
- For example, the proposition *Cavity* has possible values *True* and *False* indicating whether a given patient has a cavity or not. A random variable that has True and False as its possible values is called a **Boolean random variable**.

  More generally, propositions can include the equality predicate with random variables and the possible values they can have.
  For example, we might have a random variable *Color* with possible values *red*, *green*, *blue*, and *other*.

  Then P(Color=red) indicates the likelihood that the color of a given object is red.
  Similarly, for Boolean random variables we can ask P(A=True), which is abbreviated to P(A), and P(A=False), which is abbreviated to P(~A).

Axioms of Probability Theory

Probability Theory provides us with the formal mechanisms and rules for manipulating propositions represented probabilistically. The following are the three axioms of probability theory:

- 0 <= P(A=a) <= 1 for all *a* in sample space of A
- P(True)=1, P(False)=0
- P(A v B) = P(A) + P(B) - P(A ^ B)

From these axioms we can show the following properties also hold:

- P(~A) = 1 - P(A)
- P(A) = P(A ^ B) + P(A ^ ~B)
- Sum{P(A=a)} = 1, where the sum is over all possible values *a* in the sample space of A

Joint Probability Distribution

Given an application domain in which we have determined a sufficient set of random variables to encode all of the relevant information about that domain, we can completely specify all of the possible probabilistic information by constructing the **full joint probability distribution**,
P(V1=v1, V2=v2, ..., Vn=vn), which assigns probabilities to all possible combinations of values to all random variables.

For example, consider a domain described by three Boolean random variables, Bird, Flier, and Young. Then we can enumerate a table showing all possible interpretations and associated probabilities:

Bird	Flier	Young	Probability
T	T	T	0.0
T	T	F	0.2
T	F	T	0.04
T	F	F	0.01
F	T	T	0.01
F	T	F	0.01
F	F	T	0.23
F	F	F	0.5

Notice that there are 8 rows in the above table representing the fact that there are 2^3 ways to assign values to the three Boolean variables. More generally, with *n* Boolean variables the table will be of size 2^n. And if *n* variables each had *k* possible values, then the table would be size k^n.

Also notice that the sum of the probabilities in the right column must equal 1 since we know that the set of all possible values for each variable are known. This means that for *n* Boolean random variables, the table has 2^n-1 values that must be determined to completely fill in the table.

If all of the probabilities are known for a full joint probability distribution table, then we can compute *any* probabilistic statement about the domain. For example, using the table above, we can compute

- P(Bird=T) = P(B) = 0.0 + 0.2 + 0.04 + 0.01 = 0.25
- P(Bird=T, Flier=F) = P(B, ~F) = P(B, ~F, Y) + F(B, ~F, ~Y) = 0.04 + 0.01 = 0.05

Conditional Probabilities

- Conditional probabilities are key for reasoning because they formalize the process of accumulating evidence and updating probabilities based on new evidence.
- For example, if we know there is a 4% chance of a person having a cavity, we can represent this as the **prior** (aka unconditional) probability P(Cavity)=0.04.
- Say that person now has a symptom of a toothache, we'd like to know what is the **posterior** probability of a Cavity given this new evidence. That is, compute P(Cavity | Toothache).
- If P(A|B) = 1, this is equivalent to the sentence in Propositional Logic B => A. Similarly, if P(A|B)
  =0.9, then this is like saying B => A with 90% certainty.
- In other words, we've made implication fuzzy because it's not absolutely certain.
- Given several measurements and other "evidence", E1, ..., Ek, we will formulate queries as P(Q | E1, E2, ..., Ek) meaning "what is the degree of belief that Q is true given that we know E1, ..., Ek *and nothing else*."

  Conditional probability is defined as: P(A|B) = P(A ^ B)/P(B) = P(A,B)/P(B)

  One way of looking at this definition is as a normalized (using P(B)) joint probability (P(A,B)).
- Example Computing Conditional Probability from the Joint Probability Distribution Say we want to compute P(~Bird | Flier) and we know the full joint probability distribution function given above.
- We can do this as follows:
- P(~B|F) = P(~B,F) / P(F)
- = (P(~B,F,Y) + P(~B,F,~Y)) / P(F)
- = (.01 + .01)/P(F)

  Next, we could either compute the marginal probability P(F) from the full joint probability distribution, or, as is more commonly done, we could do it by using a process called **normalization**, which first requires computing

  P(B|F) = P(B,F) / P(F)

= (P(B,F,Y) + P(B,F,~Y)) / P(F)
= (0.0 + 0.2)/P(F)

Now we also know that P(~B|F) + P(B|F) = 1, so substituting from above and solving for P(F) we get P(F) = 0.22. Hence, P(~B|F) = 0.02/0.22 = 0.091.

While this is an effective procedure for computing conditional probabilities, it is intractable in general because it means that we must compute and store the full joint probability distribution table, which is exponential in size.

- **Some important rules related to conditional probability are:**

- Rewriting the definition of conditional probability, we get the **Product Rule**: P(A,B) = P(A|B)P(B)
- **Chain Rule**: P(A,B,C,D) = P(A|B,C,D)P(B|C,D)P(C|D)P(D), which generalizes the product rule for a joint probability of an arbitrary number of variables. Note that ordering the variables results in a different expression, but all have the same resulting value.
- **Conditionalized version of the Chain Rule**: P(A,B|C) = P(A|B,C)P(B|C)
- **Bayes's Rule**: P(A|B) = (P(A)P(B|A))/P(B), which can be written as follows to more clearly emphasize the "updating" aspect of the rule: P(A|B) = P(A) * [P(B|A)/P(B)] Note: The terms P(A) and P(B) are called the **prior** (or **marginal**) probabilities. The term P(A|B) is called the **posterior** probability because it is derived from or depends on the value of B.
- **Conditionalized version of Bayes's Rule**: P(A|B,C) = P(B|A,C)P(A|C)/P(B|C)
- **Conditioning (aka Addition) Rule**: P(A) = Sum{P(A|B=b)P(B=b)} where the sum is over all possible values *b* in the sample space of B.
- P(~B|A) = 1 - P(B|A)

Assuming conditional independence of B and C given A, we can simplify Bayes's Rule for two pieces of evidence B and C:

- P(A | B,C) = (P(A)P(B,C | A))/P(B,C)
- = (P(A)P(B|A)P(C|A))/(P(B)P(C|B))
- = P(A) * [P(B|A)/P(B)] * [P(C|A)/P(C|B)]
- = (P(A) * P(B|A) * P(C|A))/P(B,C)

Probabilistic Notation:

Artificial Intelligence (AI) heavily relies on probabilistic models to make decisions, predict outcomes, and learn from data. These models are articulated and implemented using probabilistic notation, a formal system of symbols and expressions that enables precise communication of stochastic concepts and relationships. This article provides a comprehensive overview of probabilistic notation in AI.

What is Probabilistic Notation?

Probabilistic notation refers to the symbols and conventions used to represent and manipulate probabilities and statistical concepts. This notation is fundamental in fields such as statistics, machine learning, and artificial intelligence, where dealing with uncertainty and variability is crucial. Here are some key elements of probabilistic notation:

Basic Probabilistic Notations

Here are some key elements of probabilistic notation, which form the foundation for more advanced probabilistic models in AI:

1. Probability Notation:

Probability Notation	Description
$P(A)$	The probability of event A occurring
$P(A')$	The probability of event A not occurring
$P(A \cap B)$	The probability of both A and B occurring at the same time
$P(A \cup B)$	The probability of either A or B occurring
$P(A \cap B')$	The probability of A occurring but not B
$P(A' \cup B)$	The probability of either A not occurring or B occurring

2. Conditional Probability:

- **P(A | B)**: The probability of event A occurring given that event B has occurred. This is fundamental in AI for updating beliefs based on new evidence.
- Bayes' Theorem: $P(A \mid B) = P(B)P(B \mid A) \cdot P(A)$, which provides a way to update probabilities based on new data.

3. Joint Probability:

The probability of both A and B occurring, which can also be written as P(A∩B). This is essential for understanding the relationships between multiple variables.

4. Marginal Probability:

The probability of event A P(A) occurring, regardless of other events. This is derived by summing or integrating over the joint probabilities of A with all other possible events.

Advanced Probabilistic Notations

1. Random Variables:

- X: A random variable representing a possible outcome.
- P(X = x): The probability that the random variable X takes the value x.
- P(X ≤ x): The probability that the random variable X takes a value less than or equal to x.

2. Probability Distributions:

- Probability Mass Function (PMF): For discrete random variables, $P(X = x)$ denotes the PMF.
- Probability Density Function (PDF): For continuous random variables, $f_X(x)$ denotes the PDF.
- **Cumulative Distribution Function (CDF)**: $F_X(x) = P(X \leq x)$ gives the cumulative probability up to x.

3. Expectation and Variance:

- E[X]: The expected value or mean of the random variable X.
- Var(X): The variance of the random variable X, representing the spread of its possible values.

4. Covariance and Correlation:

- Cov(X, Y): The covariance between random variables X and Y, indicating the degree to which they change together.
- Corr(X, Y): The correlation coefficient between X and Y, a normalized measure of their linear relationship.

Applications of Probabilistic Notation in AI:

1. Bayesian Networks:

- Bayesian networks use directed acyclic graphs (DAGs) to represent the probabilistic relationships among a set of variables. Nodes represent random variables, and edges represent conditional dependencies.
- Joint Probability Distribution: The joint probability distribution of a Bayesian network is the product of the conditional probabilities of each node given its parents.

2. Hidden Markov Models (HMMs):

- Hidden Markov Models (HMMs) are used to model systems that have hidden states influencing observable events. They are widely used in speech recognition, natural language processing, and bioinformatics.

- **Transition Probability**: $P(s_t \mid s_{t-1})$ represents the probability of transitioning from state $s_{t-1}$to state s_t.
- **Emission Probability**: $P(o_t \mid s_t)$represents the probability of observingo_t given the state s_t.

3. Markov Decision Processes (MDPs):

- Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker.

- **Transition Model**: $P(s\prime \mid s, a)$ denotes the probability of transitioning to state *s'* from state *s* after taking action *a*.
- **Reward Function**: $R(s, a)$represents the reward received after taking action *a* in state *s*.

4. Gaussian Processes (GPs):

- GPs are used for regression and classification tasks in machine learning. They define a distribution over functions and provide a principled way to incorporate uncertainty in predictions.
  - **Mean Function**: $m(x) = E[f(x)]$gives the mean of the function values.
  - **Covariance Function**: $k(x, x\prime) = Cov(f(x), f(x\prime))$ defines the covariance between function values at *x* and *x'*.

5. Probabilistic Graphical Models (PGMs):

- PGMs use graphs to encode the conditional independence structure between random variables. They include Bayesian networks (directed) and Markov networks (undirected).
- Factorization: The joint probability distribution in PGMs is factored into a product of smaller, local distributions, facilitating efficient computation.

Importance of Probabilistic Notation in AI
Probabilistic notation is vital in AI for several reasons:

- **Handling Uncertainty**: Real-world data is often noisy and incomplete. Probabilistic models enable AI systems to make robust predictions and decisions despite uncertainty.
- **Learning from Data:** Many machine learning algorithms, including Bayesian methods and probabilistic graphical models, rely on probabilistic notation to learn from data and update beliefs.
- **Inference:** Probabilistic notation provides the tools for performing inference, allowing AI systems to deduce new information from existing knowledge.
- **Communication:** A standardized probabilistic notation facilitates clear and precise communication of complex probabilistic concepts among researchers and practitioners.
- **Decision Making**: Probabilistic models support decision-making under uncertainty, a common scenario in real-world applications like robotics, finance, and healthcare.

Joint Probability | Concept, Formula and Examples:

Probability theory is a cornerstone of statistics, offering a powerful tool for navigating uncertainty and randomness in various fields, including business. One key concept within probability theory is Joint Probability, which enables us to analyse the likelihood of multiple events occurring simultaneously.

What is Joint Probability in Business Statistics?

In the realm of business statistics, Joint Probability refers to the likelihood of two or more events happening together or in conjunction with each other. It helps answer questions such as, "What is the probability of both event A and event B occurring in a business context?"

Joint probability offers valuable insights into the likelihood of multiple events happening together. This helps us in several ways:

1. Co-occurrence: Joint probability helps us understand how likely it is for two or more events to happen at the same time. This is important for seeing how events are connected and the probability of them occurring together.

2. Risk Evaluation: In areas like finance and insurance, joint probability helps us assess the risk when multiple events overlap. For instance, it can estimate the chance of multiple financial instruments facing losses simultaneously.

3. Quality Check: Businesses can use joint probability to gauge the reliability and quality of their products or processes. It shows the likelihood of multiple defects or issues occurring at once, which allows for proactive quality improvement efforts.

4. Event Relationships: Joint probability can indicate if events are related or not. If joint probability significantly differs from the product of individual probabilities, it suggests events are connected, and the occurrence of one affects the likelihood of the other.

5. Decision Support: When businesses need to make choices involving multiple factors or events, joint probability provides a numerical foundation for decision-making. It helps assess how different variables together impact the desired outcome.

6. Resource Management: In situations with limited resources, understanding joint probability helps optimize resource allocation. For example, in supply chain management, it can estimate the chance of multiple supply chain disruptions happening at the same time, enabling better risk management strategies.

Formula for Joint Probability:

The formula for calculating joint probability hinges on whether the events are independent or dependent:

1. For Independent Events

When events A and B are independent, meaning that the occurrence of one event does not impact the other, we use the multiplication rule:

$P(A \cap B) = P(A) \times P(B)$

Here, P(A) is the probability of occurrence of event A, P(B) is the probability of occurrence of event B, and $P(A \cap B)$ is the joint probability of events A and B.

2. For Dependent Events

Events are often dependent on each other, meaning that one event's occurrence influences the likelihood of the other. Here, we employ a modified formula:

$P(A \cap B) = P(A) \times P(B|A)$

Here, P(A) is the probability of occurrence of event A, P(B|A) is the conditional probability of occurrence of event B when event A has already occurred, and P(A∩B) is the joint probability of events A and B.

Examples of Joint Probability

Example 1: Independent Events

Suppose you are running an e-commerce platform, and you want to find the probability of a customer purchasing a red shirt (event A) and a blue hat (event B) independently. Find out the Joint Probability where

P(A): The probability of a customer buying a red shirt is 0.3.

P(B): The probability of a customer purchasing a blue hat is 0.2.

Solution:

P(A∩B) = P(A) x P(B)
P(A∩B) = P(customer buying a red shirt) x P(customer buying a blue hat)
P(A∩B) = 0.3 x 0.2
P(A∩B) = 0.6

Example 2: Dependent Events

Imagine you are in the insurance business, and you want to determine the probability of a customer filing a claim (event A) and receiving a payout (event B), given that a claim was filed. Find out the Joint Probability where

P(A): The probability of a customer filing a claim is 0.1.

$P(B|A)$: The probability of a customer receiving a payout given that a claim was filed is 0.8.

Solution:

P(A∩B) = P(A) x P(B|A)

P(A∩B) = P(customer filing a claim) x P(customer receiving a payout given that a claim was filed)

P(A∩B) = 0.1 x 0.8

P(A∩B) = 0.08

P(B|A): The probability of a customer receiving a payout given that a claim was filed is 0.8.

Solution:

$P(A \cap B) = P(A) \times P(B|A)$

$P(A \cap B) = P(\text{customer filing a claim}) \times P(\text{customer receiving a payout given that a claim was filed})$

$P(A \cap B) = 0.1 \times 0.8$

$P(A \cap B) = 0.08$

Difference between Joint Probability and Conditional Probability

Joint Probability ($P(A \cap B)$)

Joint Probability addresses the simultaneous occurrence of events A and B without considering any specific order or sequence. It quantifies the combined probability of events occurring together, providing insights into their co-occurrence in a business context.

Conditional Probability ($P(B|A)$)

Conditional Probability focuses on the probability of event B happening, given that event A has already occurred. This kind of probability is utilised when the occurrence of one event influences the likelihood of another event, making it a valuable tool for understanding cause-and-effect relationships in business statistics.

Basis	Joint Probability	Conditional Probability
Definition	Probability of multiple events occurring together.	Probability of an event occurring given another event has occurred.
Application	Provides insights into the combined occurrence of events, often used in risk assessment, quality control, and event co-occurrence analysis.	Useful for understanding cause-and-effect relationships; i.e., helps predict outcomes based on known information.
Focus	Focuses on events occurring together, regardless of order.	Focuses on events that depend on or are influenced by the occurrence of another event.
Example	Probability of a customer buying both a red shirt (A) and a blue hat (B) independently.	Probability of a customer buying a blue hat (B) given that he has already bought a red shirt (A).

Conditional Independence Representation:

Conditional independence is a fundamental concept in artificial intelligence (AI), particularly in the fields of probabilistic reasoning and graphical models. It simplifies the representation and computation of complex probabilistic models by specifying the independence relationships among random variables. Understanding conditional independence is crucial for building efficient models like Bayesian Networks and Markov Random Fields.

What is Conditional Independence?

Conditional independence describes a situation where two random variables, A and B, are independent of each other given a third variable, C.

What is Conditional Independence?

Conditional independence describes a situation where two random variables, **A** and **B**, are independent of each other given a third variable, **C**.

Mathematically, this is expressed as:

$$P(A, B|C) = P(A|C) \cdot P(B|C)$$

Equivalently, in terms of conditional probabilities:

$$P(A \mid B, C) = P(A \mid C) \text{ or } P(B \mid A, C) = P(B \mid C)$$

The intuition is:

- Without C, A and B may be dependent (knowing A may give information about B).
- Once C is known, A and B no longer provide any additional information about each other.

Examples

1. Weather and Traffic Example

In this case, we can consider 'Rain', 'Accidents' and 'Wet Roads':

$P(\text{Rain, Accidents} \mid \text{Wet Roads}) = P(\text{Rain} \mid \text{Wet Roads}) \cdot P(\text{Accidents} \mid \text{Wet Roads})$

Intuition: If you know the roads are wet (C), the information about rain (A) does not provide extra insight about the likelihood of accidents (BBB). Here, A and B are conditionally independent given C.

2. Student Performance Example

In this case, we can consider 'Hours Studied', 'Test Score', and 'IQ Level':

$P(\text{Hours Studied, Test Score} \mid \text{IQ Level}) = P(\text{Hours Studied} \mid \text{IQ Level}) \cdot P(\text{Test Score} \mid \text{IQ Level})$

Intuition: Given the IQ level (C), the hours studied (A) and the test score (B) may become conditionally independent. Once the IQ level is known, knowing how much a student studied (A) does not provide additional insight about their test score (B). Here, A and B are conditionally independent given C.

Importance in AI

Simplification of Models: Conditional independence reduces the number of parameters needed to specify a probabilistic model.

Efficient Computation: It enables faster inference and learning by reducing computational complexity.

Modular Design: Models can be constructed in a modular fashion by isolating independent components.

Understanding Causality: Conditional independence is key to modeling causal relationships.

Representations in AI

Conditional independence is commonly represented in the following ways:

1. Bayesian Networks

Bayesian Networks (BNs) are directed acyclic graphs (DAGs) where nodes represent random variables, and edges represent dependencies. Conditional independence is encoded using the Markov property: a node is conditionally independent of its non-descendants given its parents.

For example, for three nodes, A → B → C: B is conditionally independent of A given C.

2. Markov Random Fields (MRFs)

MRFs are undirected graphs where nodes represent random variables, and edges represent direct dependencies. Conditional independence is encoded by the absence of an edge: two variables are conditionally independent given all other variables if there is no direct edge between them.

3. d-Separation

In Bayesian Networks, d-separation is used to determine if two sets of variables are conditionally independent given a third set. It provides a graphical criterion to identify independence relationships.

Bayes' theorem in Artificial intelligence:

Bayes Theorem in AI is perhaps the most fundamental basis for probability and statistics, more popularly known as Bayes' rule or Bayes' law. It allows us to revise our assumptions or the probability that an event will occur, given new information or evidence.

In this article, we will see how the Bayes theorem is used in AI.

Bayes' Theorem in AI

In probability theory, Bayes' theorem talks about the relation of the conditional probability of two random events and their marginal probability. In short, it provides a way to calculate the value of P(B|A) by using the knowledge of P(A|B).

Bayes' theorem is the name given to the formula used to calculate conditional

$$P(A \mid B) = P(A \cap B)/P(B) = (P(A) * P(B \mid A))/P(B)$$

where,

- *P(A) is the probability that event A occurs.*
- *P(B) defines the probability that event B occurs.*
- *P(A|B) is the probability of the occurrence of event A given that event B has already occurred.*
- *P(B|A) can now be read as: Probability of event B occurring given that event A occurred.*
- *p(A∩B) is the probability events A and B will happen together.*

Key terms in Bayes' Theorem

The Bayes' Theorem is a basic concept in probability and statistics. It gives a model of updating beliefs or probabilities when the new evidence is presented. This theorem was named after Reverend Thomas Bayes and has been applied in many fields, ranging from artificial intelligence and machine learning to data analysis.

The Bayes' Theorem encompasses four major elements:

Prior Probability (P(A)): The probability or belief in an event A prior to considering any additional evidence, it represents what we know or believe about A based on previous knowledge.

Likelihood P(B|A): the probability of evidence B given the occurrence of event A. It determines how strongly the evidence points toward the event.

Evidence (P(B)): Evidence is the probability of observing evidence B regardless of whether A is true. It serves to normalize the distribution so that the posterior probability is a valid probability distribution.

Posterior Probability P(A|B): The posterior probability is a revised belief regarding event A, informed by some new evidence B. It answers the question, "What is the probability that A is true given evidence B observed?"

Using these components, Bayes' Theorem computes the posterior probability P(A|B), which represents our updated belief in A after considering the new evidence.

In artificial intelligence, probability and the Bayes Theorem are especially useful when making decisions or inferences based on uncertain or incomplete data. It enables us to rationally update our beliefs as new evidence becomes available, making it an indispensable tool in AI, machine learning, and decision-making processes.

How Bayes theorem is relevant in AI?

Bayes' theorem is highly relevant in AI due to its ability to handle uncertainty and make decisions based on probabilities. Here's why it's crucial:

Probabilistic Reasoning: In many real-world scenarios, AI systems must reason under uncertainty. Bayes' theorem allows AI systems to update their beliefs based on new evidence. This is essential for applications like autonomous vehicles, where the environment is constantly changing and sensors provide noisy information.

Machine Learning: Bayes' theorem serves as the foundation for Bayesian machine learning approaches. These methods allow AI models to incorporate prior knowledge and update their beliefs as they see more data. This is particularly useful in scenarios with limited data or when dealing with complex relationships between variables.

Classification and Prediction: In classification tasks, such as spam email detection or medical diagnosis, Bayes' theorem can be used to calculate the probability that a given input belongs to a **particular class**. This allows AI systems to make more informed decisions based on the available evidence.

Anomaly Detection: Bayes' theorem is used in anomaly detection, where AI systems identify unusual patterns in data. By modeling the normal behavior of a system, Bayes' theorem can help detect deviations from this norm, signaling potential anomalies or security threats.

Overall, Bayes' theorem provides a powerful framework for reasoning under uncertainty and is essential for many AI applications, from decision-making to pattern recognition.

Mathematical Derivation of Bayes' Rule

Bayes' Rule is derived from the definition of conditional probability. Let's start with the definition:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

This equation states that the probability of event A given event B is equal to the probability of both events happening (the intersection of A and B) divided by the probability of event B.

Similarly, we can write the conditional probability of event B given event A:

$$P(B \mid A) = \frac{P(A \cap B)}{P(A)}$$

By rearranging this equation, we get:

$$P(A \cap B) = P(B \mid A) \cdot P(A)$$

Now, we have two expressions for $P(A \cap B)$, since both expressions are equal to $P(A \cap B)$, we can set them equal to each other:

$$P(A \mid B) \cdot P(B) = P(B \mid A) \cdot P(A)$$

To get $P(A|B)$, we divide both sides by $P(B)$:

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

Importance of Bayes' Theorem :

Bayes' Theorem is extremely important in artificial intelligence (AI) and related fields.

Probabilistic Reasoning: In AI, many problems involve uncertainty, so probabilistic reasoning is an important technique. Bayes' Theorem enables artificial intelligence systems to model and reason about uncertainty by updating beliefs in response to new evidence. This is important for decision-making, pattern recognition, and predictive modeling.

Machine Learning: Bayes' Theorem is a fundamental concept in machine learning, specifically Bayesian machine learning. Bayesian methods are used to model complex relationships, estimate model parameters, and predict outcomes. Bayesian models enable the principled handling of uncertainty in tasks such as classification, regression, and clustering.

Data Science: Bayes' Theorem is used extensively in Bayesian statistics. It is used to estimate and update probabilities in a variety of settings, including hypothesis testing, Bayesian inference, and Bayesian optimization. It offers a consistent framework for modeling and comprehending data.

Example of Bayes' Rule Application

One of the good old example of Bayes' Rule in AI is its application in spam email classification. This example demonstrates how Bayes' Theorem is used to classify emails as spam or non-spam based on the presence of certain keywords.

Consider an email filtering system that needs to determine whether an incoming email is spam or not based on the presence of the word "win" in the email. We are given the following probabilities:

P(S): The prior probability that any given email is spam.

P(H): The prior probability that any given email is not spam (ham).

P(W|S): The probability that the word "win" appears in a spam email.

P(W|H): The probability that the word "win" appears in a non-spam email.

P(W): The probability that the word "win" appears in any email.

Given Data

P(S)=0.2 (20% of emails are spam)

P(H)=0.8 (80% of emails are not spam)

P(W|S)=0.6 (60% of spam emails contain the word "win")

P(W|H)=0.1P (10% of non-spam emails contain the word "win")

We want to find P(S|W), the probability that an email is spam given that it contains the word "win".

Applying Bayes rule we get:

$$P(S \mid W) = \frac{P(W)}{P(W|S) \cdot P(S)}$$

First, we need to calculate P(W), the probability that any email contains the word "win". Using the law of total probability:

$$P(W) = P(W \mid S) \cdot P(S) + P(W \mid H) \cdot P(H)$$

Substituting the given values:

$$P(W) = (0.6 \cdot 0.2) + (0.1 \cdot 0.8) = 0.2$$

Now, we can use Bayes' Rule to find P(S|W):

$$P(S \mid W) = \frac{P(W|S) \cdot P(S)}{P(W)},$$

substituting the values:

$$P(S \mid W) = \frac{0.6 \cdot 0.2}{0.2} = 0.6$$

Thus we can conclude that the probability that an email is spam given that it contains the word "win" is 0.6, or 60%. This means that if an email contains the word "win," there is a 60% chance that it is spam.

In a real-world AI system, such as an email spam filter, this calculation would be part of a larger model that considers multiple features (words) within an email. The filter uses these probabilities, along with other algorithms, to classify emails accurately and efficiently. By continuously updating the probabilities based on incoming data, the spam filter can adapt to new types of spam and improve its accuracy over time.

Uses of Bayes Rule in Artificial Intelligence

Bayes' theorem in Al is used to draw probabilistic conclusions, update beliefs, and make decisions based on available information. Here are some important applications of Bayes' rule in AI.

Bayesian Inference: In Bayesian statistics, the Bayes' rule is used to update the probability distribution over a set of parameters or hypotheses using observed data. This is especially important for machine learning tasks like parameter estimation in Bayesian networks, hidden Markov models, and probabilistic graphical models.

Naive Bayes Classification: In the field of natural language processing and text classification, the Naive Bayes classifier is widely used. It uses Bayes' theorem to calculate the likelihood that a

document belongs to a specific category based on the words it contains. Despite its "naive" assumption of feature independence, it works surprisingly well in practice.

Bayesian Networks: Bayesian networks are graphical models that use Bayes' theorem to represent and predict probabilistic relationships between variables. They are used in a variety of AI applications, such as medical diagnosis, fault detection, and decision support systems.

Spam Email Filtering: In email filtering systems, Bayes' theorem is used to determine whether an incoming email is spam or not. The model calculates the likelihood of seeing specific words or features in spam or non-spam emails and adjusts the probabilities accordingly.

Reinforcement Learning: Bayes' rule can be used to model the environment in a probabilistic manner. Bayesian reinforcement learning methods can help agents estimate and update their beliefs about state transitions and rewards, allowing them to make more informed decisions.

Bayesian Optimization: In optimization tasks, Bayes' theorem can be used to represent the objective function as a probabilistic surrogate. Bayesian optimization techniques make use of this model to iteratively explore and exploit the search space in order to efficiently find the optimal solution. This is commonly used for hyperparameter tuning and algorithm parameter optimization.

Anomaly Detection: The Bayes theorem can be used to identify anomalies or outliers in datasets. Deviations from the normal distribution can be quantified by modeling it, which aids in anomaly detection for a variety of applications, including fraud detection and network security.

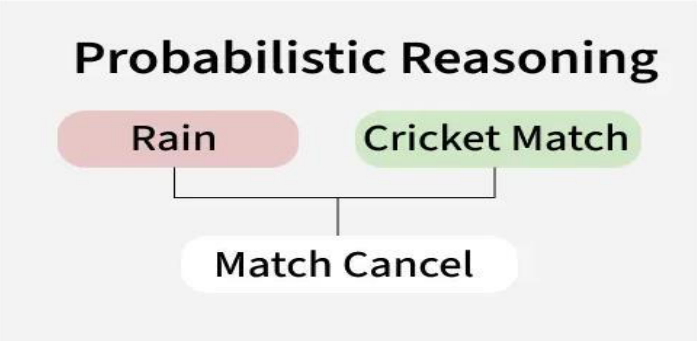
Personalization: In recommendation systems, Bayes' theorem can be used to update user preferences and provide personalized recommendations. By constantly updating a user's preferences based on their interactions, the system can recommend more relevant content.

Robotics and Sensor Fusion: In robotics, the Bayes' rule is used to combine sensors. It uses data from multiple sensors to estimate the state of a robot or its environment. This is necessary for tasks like localization and mapping.

Medical Diagnosis: In healthcare, Bayes' theorem is used in medical decision support systems to update the likelihood of various diagnoses based on patient symptoms, test results, and medical history.

Probabilistic Reasoning:

Probabilistic reasoning in Artificial Intelligence (AI) is a method that uses probability theory to manage and model uncertainty in decision-making. Unlike traditional systems that depend on precise information, it understands that real-world data is often incomplete, unclear or noisy. By giving probabilities to different possibilities, AI systems can make better decisions, predict outcomes and solve problems even when things are uncertain. This approach is important for building smart systems that can work in changing environments and make good choices in complex situations.



Need for Probabilistic Reasoning :

Probabilistic reasoning with artificial intelligence is important to different tasks such as:

Machine Learning: It helps algorithms learn from incomplete or noisy data, refining predictions over time.

Robotics: It enables robots to navigate and interact with dynamic and unpredictable environments.

Natural Language Processing: It helps AI understand human language which is often ambiguous and depends on context.

Decision Making: It allows AI systems to evaluate different outcomes and make better decisions by considering the likelihood of various possibilities.

Key Concepts in Probabilistic Reasoning

Probabilistic reasoning helps AI systems make decisions and predictions when they have to deal with uncertainty. It uses different ideas and models to understand how likely things are even when we don't have all the answers. Let's see some of the important concepts:

1**. Probability**: It is a way to measure how likely something is to happen, typically expressed as a number between 0 and 1. In AI, we use probabilities to understand and make predictions when the information we have is uncertain or incomplete.

2**. Bayes' Theorem**: It helps AI systems update their beliefs when they get new information. It's like changing our mind about something based on new evidence. This is useful when we need to adjust our predictions after learning new facts.

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

Where:

- $P(A \mid B)$: Probability of A happening, given B has happened (posterior probability).
- $P(B \mid A)$: Probability of B happening, given A happened.
- $P(A)$: Prior probability of A.
- $P(B)$: Probability of observing B.

3. **Conditional Probability**: It is the chance of an event happening, given that something else has already happened. This helps when the outcome depends on something that happened before.

4. **Random Variables**: They are values that can change or vary based on uncertainty. In simple terms, these are the things AI tries to predict or estimate. The possible outcomes of these variables depend on probability.

Types of Probabilistic Models

There are different models that use these concepts to help AI systems make sense of uncertainty. Let's see some of them:

Bayesian Networks: They are graphs that show how different variables are connected with probabilities. Each node represents a variable and the edges show how they depend on each other. These networks help us understand how one piece of information can affect another.

Markov Models: They predict the future state of a system based only on the present state with no regard for the past. This is known as the "memoryless" property which means the future depends only on the current situation not the history that led to it.

Hidden Markov Models (HMMs): They extend Markov models by introducing hidden states that cannot be directly observed. These models help infer the hidden states based on observable data, using statistical techniques to estimate the likelihood of these unobservable conditions.

Probabilistic Graphical Models: It combine the features of Bayesian networks and Hidden Markov Models, allowing more complex relationships between variables to be represented. It provide a framework for managing uncertainty in large systems where many variables are connected and interact with each other.

Markov Decision Processes (MDPs): They are used for decision-making, particularly in reinforcement learning. It model an agent's interaction with an environment where the agent takes actions that affect the state of the environment and receives rewards or penalties based on those actions.

Techniques in Probabilistic Reasoning

Inference: It calculates the probability of an outcome based on known data. Exact methods like variable elimination and approximate methods like Markov Chain Monte Carlo (MCMC) are used, depending on the complexity of the system.

Learning: It updates the parameters of probabilistic models as new data comes in, improving predictions. Techniques like maximum likelihood estimation and Bayesian estimation allow models to adapt and become more accurate over time.

Decision Making: AI uses probabilistic reasoning to make decisions that maximize expected rewards. Partially Observable Markov Decision Processes (POMDPs) are used when some information is hidden.

How Probabilistic Reasoning Enhances AI Systems?

Probabilistic reasoning helps AI systems navigate uncertainty, enabling them to make better decisions even when information is unclear. Let's see how it works:

Quantifying Uncertainty: Probabilistic reasoning turns uncertainty into probabilities. Instead of a simple "yes" or "no," it provides a probability, like "there's a 60% chance of rain tomorrow."

Reasoning with Evidence: As new information comes in, AI systems update their predictions. For example, if dark clouds appear, the chance of rain might rise to 80%. This continuous adjustment helps AI stay accurate.

Learning from Past Experiences: AI systems can improve predictions by learning from historical data. For example, weather predictions become more accurate as the AI system accounts for past seasonal trends.

Effective Decision-Making: It allows AI to make informed decisions by considering the likelihood of different outcomes. It helps AI weigh possible paths and choose the best option even when the future is uncertain.

Applications of Probabilistic Reasoning in AI

Probabilistic reasoning is applicable in a variety of domains which includes:

Robotics: In robotics, probabilistic reasoning helps with navigation and mapping. For example, in Simultaneous Localization and Mapping, robots create maps and track their position in unknown environments.

Healthcare: AI systems use probabilistic models to predict disease likelihood and assist in diagnosis. Bayesian networks can model medical factors like symptoms and test results to guide decisions.

Natural Language Processing: In tasks like speech recognition and translation, models like Hidden Markov Models (HMMs) help AI understand and process ambiguous language.

Finance: Probabilistic reasoning in finance helps predict market trends and assess risks. Techniques like Bayesian inference and Monte Carlo simulations model financial uncertainties for better decision-making.

Advantages of Probabilistic Reasoning

Flexibility: Probabilistic models can handle different kinds of uncertainty and can be adapted to work in various fields from healthcare to robotics.

Robustness: These models remain effective even when the data is noisy or incomplete, making them reliable in real-world scenarios where perfect data is rarely available.

Transparency: It provides a clear framework for understanding and explaining uncertainty which helps build trust and improve the interpretability of AI decisions.

Scalability: It can scale to handle large amounts of data and complex systems, making them suitable for applications like big data analysis and large-scale decision-making processes.

Decision Support: These models assist in making informed decisions under uncertainty by calculating the likelihood of different outcomes, helping AI systems choose the best course of action based on expected results.

Challenges of Probabilistic Reasoning

Despite its various advantages, probabilistic reasoning in AI also has several challenges:

Complexity: Some models such as large Bayesian networks can become computationally expensive, especially as the number of variables grows. This can slow down processing and limit scalability.

Data Quality: Probabilistic models heavily rely on accurate and clean data. If the data is noisy, incomplete or biased, the model's predictions can become unreliable, leading to incorrect conclusions.

Interpretability: Understanding how probabilistic models make decisions can be tough, particularly in complex systems or deep learning models. This makes it harder to trust and explain AI decisions to non-experts.

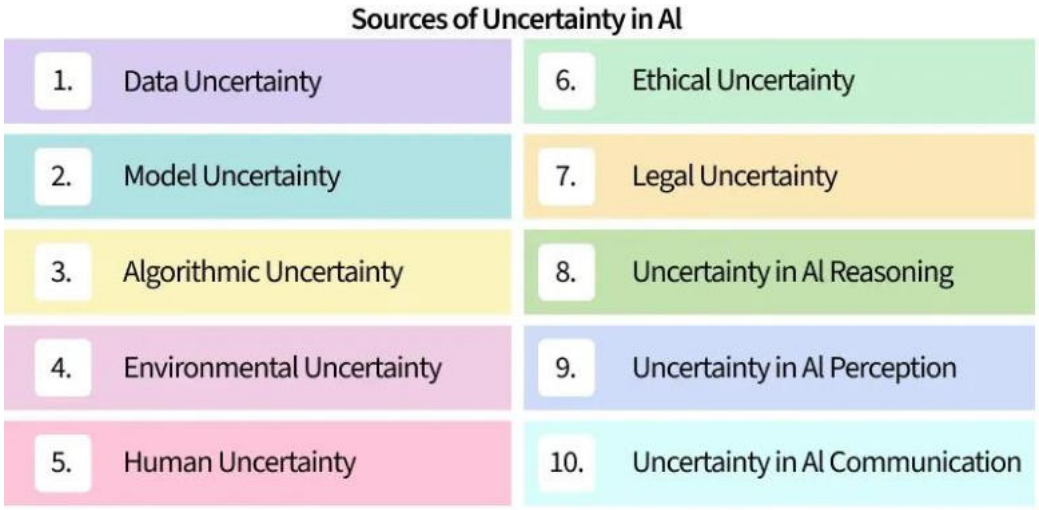
Representing Knowledge in an Uncertain Domain:

Real-world AI systems rarely function under perfect conditions. Instead, they must act in environments filled with incomplete information, noisy data and unpredictable events. Traditional deterministic approaches assumes full and accurate knowledge, often fail in such scenarios. This challenge has led to the development of complex techniques for reasoning under uncertainty.

Understanding Uncertain Domains:

An uncertain domain refers to an environment where information is incomplete or unpredictable. Unlike deterministic systems where outcomes are precisely predictable, AI systems operating in uncertain domains must handle:

- Incomplete Information: Missing data needed for fully informed decisions
- Ambiguity: Inputs that support multiple interpretations
- Noise: Data corruption due to measurement errors or external factors
- Stochastic Processes: Events with probabilistic outcomes that can't be predicted deterministically



Example: A medical diagnosis system must interpret symptoms that suggest multiple diseases, cope with noisy test results and account for incomplete patient histories.

Why Uncertainty Management Matters?

Effective uncertainty handling enables AI systems to make robust decisions even when data is imperfect. Key benefits include:

- Quantifying confidence in predictions to improve decision trustworthiness
- Dynamically updating beliefs as new information arrives
- Maintaining performance where deterministic systems typically fail

This capability is crucial in complex domains like autonomous driving, medical diagnostics and financial forecasting.

Probabilistic Reasoning Approaches:

Uncertainty management often relies on probabilistic reasoning, which uses probability theory to represent and manipulate uncertain knowledge.

1. Bayesian Networks

Bayesian networks use directed acyclic graphs to model probabilistic relationships. Each node represents a variable and edges represent conditional dependencies.

- Enable both predictive (cause to effect) and diagnostic (effect to cause) reasoning
- Efficiently compute posterior probabilities when new evidence is introduced

Example: In medical diagnosis, symptoms like fever and cough link to diseases such as flu or pneumonia. The network estimates probabilities based on observed symptoms.

2. Hidden Markov Models (HMMs)

HMMs handle sequential data where the true system states are hidden. They assume a Markov process, where the next state depends only on the current one.

- Widely used in speech recognition, where sounds are observed but underlying phonetic states are hidden
- Also applied in bioinformatics, NLP and financial modeling

They model both temporal dependencies and uncertainty, making them ideal for time-series analysis.

3. Markov Decision Processes (MDPs)

MDPs model decision-making under uncertainty through:

- States, Actions, Transition Probabilities and Rewards
- Agents learn optimal policies by evaluating possible outcomes over time

Example: An autonomous robot uses MDPs to plan routes through an office, accounting for movement uncertainty and reward-based goals (like avoiding obstacles).

MDPs are foundational to reinforcement learning, where agents learn through interaction with their environment.

Fuzzy Logic and Approximate Reasoning

While probabilistic methods can quantify uncertainty in numerical values, fuzzy logic deals with uncertainty arising from imprecision or vagueness in concepts.

1. Fuzzy Sets and Membership Functions

- Allow partial membership, assigning degrees between 0 and 1
- Reflect human intuition better than rigid, binary logic

Example: For a temperature control system, “warm” might be defined fuzzily, with 25°C having 0.8 membership and 30°C having full membership.

2. Fuzzy Inference Systems

Use linguistic rules for decision-making under uncertainty, such as:

"If temperature is high and humidity is low, then increase cooling moderately."

These systems:

- Combine rule outputs based on input conditions
- Are especially useful in control systems where human expertise is hard to formalize mathematically

Advanced Uncertainty Frameworks

Beyond classical probability and fuzzy systems, advanced frameworks allow richer handling of incomplete data.

1. Dempster-Shafer Theory

Dempster-Shafer Theory generalizes probability by modeling degrees of belief, accounting for incomplete knowledge.

- Useful for combining evidence from multiple sources
- Does not require prior probabilities, making it effective for domains with high uncertainty

Example: In fault diagnosis, sensor data can be combined to assess the likelihood of different failures, accounting for uncertainty and sensor reliability.

2. Belief Networks

Belief networks extend Bayesian networks by allowing imprecise probabilities or intervals rather than exact values.

- Handle uncertainty in relationships between variables
- Enable reasoning even when exact distributions are unknown

Example: In intelligent tutoring systems, belief networks model student understanding with uncertain links between concepts.

3. Case-Based Reasoning (CBR) Under Uncertainty

CBR solves new problems by referencing past cases, adapting known solutions to current situations.

- Must handle missing or incomplete case data
- Often combined with probabilistic methods to assign confidence scores

Example: A support system retrieves similar past issues but adjusts recommendations based on current, possibly incomplete, information.

Modern CBR systems integrate machine learning to improve case similarity assessment and solution adaptation under uncertainty.

Applications Across Domains

Uncertainty management techniques have been successfully applied across various industries:

Medical Diagnosis: Systems use Bayesian networks to model probabilistic relationships between symptoms and diseases. Fuzzy logic helps interpret vague patient inputs, while Dempster-Shafer theory integrates multiple test results with differing reliability levels.

Autonomous Systems: Probabilistic localization methods handle GPS and sensor noise to estimate vehicle position. Fuzzy controllers enable smooth navigation by processing imprecise inputs from the environment.

Natural Language Processing (NLP): Hidden Markov Models (HMMs) are employed for tasks like speech recognition by modeling sequential word patterns. Probabilistic grammars handle syntactic ambiguity and fuzzy matching improves performance in information retrieval and search engines.

Financial Systems: Bayesian models dynamically update predictions based on evolving market conditions. Monte Carlo simulations quantify risk in uncertain financial environments. Fuzzy rule-based systems capture expert trading heuristics for decision-making support.

Implementation Considerations and Challenges:

Despite their effectiveness, uncertainty management techniques face implementation hurdles:

Bayesian Networks: Become computationally intensive with many variables, often needing approximation methods

Fuzzy Systems: Depend on well-designed membership functions and expert-defined rules

Model Selection: The right framework depends on data availability, domain knowledge and desired interpretability

Modern systems often combine approaches:

- Ensemble methods integrate probabilistic, fuzzy and neural models
- Deep learning incorporates uncertainty using dropout or Bayesian neural nets.

Basic Understanding of Bayesian Belief Networks:

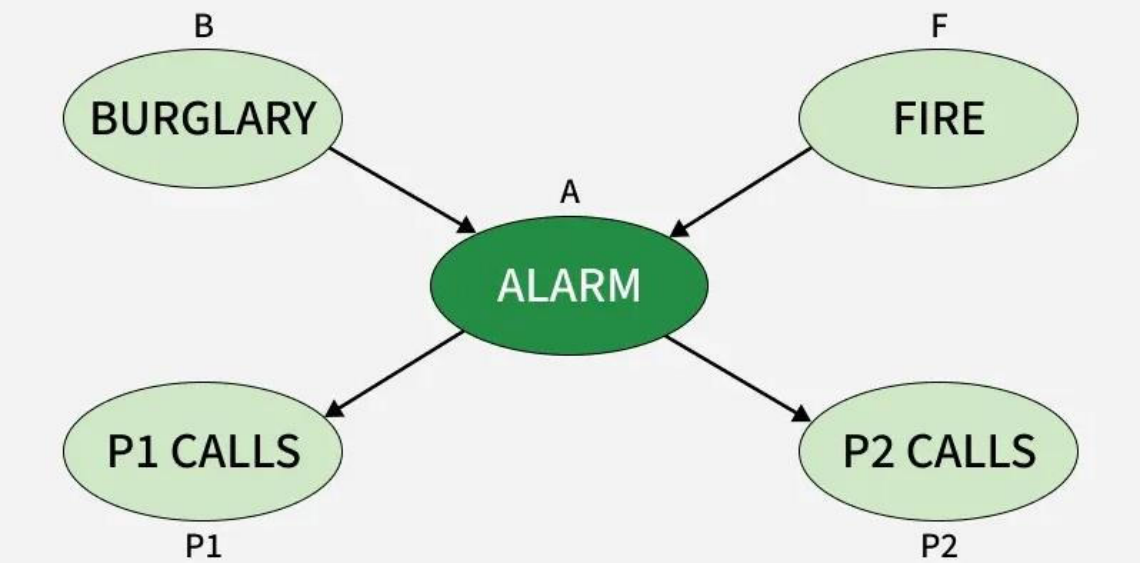
Bayesian Belief Network (BBN) is a graphical model that represents the probabilistic relationships among variables. It is used to handle uncertainty and make predictions or decisions based on probabilities.

- **Graphical Representation**: Variables are represented as nodes in a directed acyclic graph (DAG), and their dependencies are shown as edges.
- **Conditional Probabilities**: Each node's probability depends on its parent nodes, expressed as $P(\text{Variable} \mid \text{Parent})$.
- **Probabilistic Model**: Built from probability distributions, BBNs apply probability theory for tasks like prediction and anomaly detection.

Bayesian Belief Networks are valuable tools for understanding and solving problems involving uncertain events. They are also known as Bayes networks, belief networks, decision networks, or Bayesian models.

Note: A classifier assigns data in a collection to desired categories.

Consider this example:



- In the above figure, we have an alarm 'A' - a node, say installed in a house of a person 'gfg', which rings upon two probabilities i.e burglary 'B' and fire 'F', which are - parent nodes of the alarm node. The alarm is the parent node of two probabilities P1 calls 'P1' & P2 calls 'P2' person nodes.
- Upon the instance of burglary and fire, 'P1' and 'P2' call person 'gfg', respectively. But, there are few drawbacks in this case, as sometimes 'P1' may forget to call the person 'gfg', even after hearing the alarm, as he has a tendency to forget things, quick. Similarly, 'P2', sometimes fails to call the person 'gfg', as he is only able to hear the alarm, from a certain distance.

Calculating Conditional Probability of Events in a Bayesian Network

Find the probability that 'P1' is true (P1 has called 'gfg'), 'P2' is true (P2 has called 'gfg') when the alarm 'A' rang, but no burglary 'B' and fire 'F' has occurred.

=> P (P1, P2, A, ~B, ~F) [where- P1, P2 & A are 'true' events and '~B' & '~F' are 'false' events]

[Note: The values mentioned below are neither calculated nor computed. They have observed values]

Burglary 'B' -

P (B=T) = 0.001 ('B' is true i.e burglary has occurred)

P (B=F) = 0.999 ('B' is false i.e burglary has not occurred)

Fire 'F' -

P (F=T) = 0.002 ('F' is true i.e fire has occurred)

P (F=F) = 0.998 ('F' is false i.e fire has not occurred)

Alarm 'A' :

B	F	P (A=T)	P (A=F)
T	T	0.95	0.05
T	F	0.94	0.06
F	T	0.29	0.71
F	F	0.001	**0.999**

- The alarm 'A' node can be 'true' or 'false' (i.e may have rung or may not have rung). It has two parent nodes burglary 'B' and fire 'F' which can be 'true' or 'false' (i.e may have occurred or may not have occurred) depending upon different conditions.

Person 'P1' :

A	P (P1=T)	P (P1=F)
T	0.95	0.05
F	0.05	0.95

- The person 'P1' node can be 'true' or 'false' (i.e may have called the person 'gfg' or not) . It has a parent node, the alarm 'A', which can be 'true' or 'false' (i.e may have rung or may not have rung ,upon burglary 'B' or fire 'F').

Person 'P2' :

A	P (P2=T)	P (P2=F)
T	0.80	0.20
F	0.01	0.99

- The person 'P2' node can be 'true' or false' (i.e may have called the person 'gfg' or not). It has a parent node, the alarm 'A', which can be 'true' or 'false' (i.e may have rung or may not have rung, upon burglary 'B' or fire 'F').

Solution: Considering the observed probabilistic scan -

With respect to the question — P (P1, P2, A, ~B, ~F) , we need to get the probability of 'P1'. We find it with regard to its parent node - alarm 'A'. To get the probability of 'P2', we find it with regard to its parent node — alarm 'A'.

We find the probability of alarm 'A' node with regard to '~B' & '~F' since burglary 'B' and fire 'F' are parent nodes of alarm 'A'.

From the observed probabilistic scan, we can deduce -

P (P1, P2, A, ~B, ~F)

= P (P1/A) * P (P2/A) * P (A/~B~F) * P (~B) * P (~F)

= 0.95 * 0.80 * 0.001 * 0.999 * 0.998

= 0.00075

Exact Inference in Bayesian Networks:

Bayesian Networks (BNs) are powerful graphical models for probabilistic inference, representing a set of variables and their conditional dependencies via a directed acyclic graph (DAG). These models are instrumental in a wide range of applications, from medical diagnosis to machine learning. Exact inference in Bayesian Networks is a fundamental process used to compute the probability distribution of a subset of variables, given observed evidence on a set of other variables.

Introduction to Bayesian Networks:

A Bayesian Network consists of nodes representing random variables and directed edges representing conditional dependencies between these variables. Each node X_i in the network is associated with a conditional probability table (CPT) that quantifies the effect of the parents' nodes on X_i.

Key Components:

- **Nodes:** Represent random variables which can be discrete or continuous.
- **Edges**: Directed arrows showing dependency; $X \rightarrow Y$ implies that X influences Y.
- **Conditional Probability Tables (CPTs)**: Each node has a CPT that describes the probability of the node given its parents.

Basics of Inference in Bayesian Networks

Inference in Bayesian Networks involves answering probabilistic queries about the network. The most common types of queries are:

- **Marginalization:** Determining the probability distribution of a subset of variables, ignoring the values of all other variables.
- **Conditional Probability:** Computing the probability distribution of a subset of variables given evidence observed on other variables.

Mathematically, if X are the query variables and E are the evidence variables with observed values e, the goal is to compute P(X/E=e)

Methods of Exact Inference

Amongst the extant exact inference methods developed in the context of Bayesian networks. These methods operate under the assumptions of the network structure to achieve efficient probability calculations.

The methods of Exact Inference are:

1. Variable Elimination
2. Junction Tree Algorithm
3. Belief Propagation

Variable Elimination

Variable Elimination is a popular exact inference technique that systematically sums out the variables not of interest. The process involves manipulating and combining the network's CPTs to answer queries efficiently.

Steps:

1. Factorization: Break down the joint probability distribution into a product of factors, each corresponding to a CPT in the network.
2. Elimination: Sequentially eliminate each non-query, non-evidence variable by summing over its values. This step reduces the dimensionality of the problem.
3. Normalization: After all eliminations, normalize the resulting distribution to ensure that it sums to one.

Mathematical Representation:

To compute $P(X \mid E = e)$, one might need to sum out a variable Z not in X or E:

$$P(X \mid E = e) = \alpha \sum_Z P(X, Z, E = e)$$

where α is a normalization constant.

Junction Tree Algorithm

The Junction Tree Algorithm, also known as the Clique Tree Algorithm, is a more structured approach that converts the Bayesian Network into a tree structure called a "junction tree" or "clique tree," where each node (clique) contains a subset of variables that form a complete (fully connected) subgraph in the network.

Steps:

1. **Triangulation**: Modify the network to ensure that every cycle of four or more nodes has a chord (an edge that is not part of the cycle but connects two nodes of the cycle).
2. **Building the Junction Tree**: Form cliques of variables and organize them into a tree structure where each edge represents a conditional independence statement.
3. **Message Passing**: Perform a two-phase message passing (collecting and distributing) to propagate information throughout the tree.

Mathematical Representation: During the message passing phase, messages (functions of probabilities) are passed between cliques. If C_i and C_j are two cliques connected by a separator , the message from C_i to C_j can be calculated as:

$m_{i \to j}(S) = \sum_{Ci \setminus S} \phi_{C_i}(X_{C_i})$

where ϕ_{C_i} is the potential function associated with clique C_i.

Belief Propagation

Belief Propagation (BP) is another exact inference method used particularly in networks that form a tree structure or can be restructured into a tree-like form using the Junction Tree Algorithm. It involves passing messages between nodes and uses these messages to compute marginal probabilities at each node.

Steps:

1. **Initialization**: Each node initializes messages based on its local evidence and conditional probabilities.
2. **Message Passing:** Nodes send and receive messages to and from their neighbors. Each message represents a belief about the state of the sender, conditioned on the evidence.
3. **Belief Update**: Each node updates its belief based on the incoming messages and its initial probability.

Belief Propagation is especially effective in tree-structured networks where messages can be propagated without loops, ensuring that each node's final belief is computed exactly once all messages have been passed.

Challenges of Exact Inference

1. **Exponential Complexity**: Exact approaches like the variable elimination and the junction tree are computationally complex and increase with a rate that is exponential to the number of variables in the network. The diversity of the degrees of freedom further implies that exact inference is not feasible for large networks with a large number of variables.
2. **Memory Requirements:** Most exact inference methods involve the computation of a large table or another structure such as a junction tree which in turn has to be stored in memory. The use of sparse structures or high-dimensional probability distributions may make the memory demands impractical in some cases, especially when the number of variables in the network is large.
3. **Loops and Cycles:** Local computations can be performed using Bayesian networks without loops or cycles that cause problems with exact inference algorithms. Variable elimination can result in suboptimal computations and, in addition, junction tree algorithms may cause more complicated loops.