ArtificialIntelligence
Unit-I

1. Introduction

1.1 What Is AI?

Some definitions of artificial intelligence:

- Thinking Humanly
- Acting Humanly
- Thinking Rationally
- Acting Rationally

1.1.1 Acting humanly: The Turing test approach

The **Turing test**, proposed by Alan Turing, was designed as a thought experiment that would sidestep the philosophical vagueness of the question "Can a machine think?" The computer would need the following capabilities to pass the Turing test:

- **natural language processing** to communicate successfully in a human language
- **knowledge representation** to store what it knows or hears
- **automated reasoning** to answer questions and to draw new conclusions
- **machine learning** to adapt to new circumstances and to detect and

extrapolate patterns To pass the **total Turing test**, a robot will need

- **computer vision** and speech recognition to perceive the world
- **robotics** to manipulate objects and move about.

1.1.2 Thinking humanly: The cognitive modeling approach

We can learn about human thought in three ways:

- **introspection**–trying to catch our own thoughts as they go by
- **psychological experiments**–observing a person in action
- .
- **brain imaging**–observing the brain in action

The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind.

1.1.3 Thinking rationally: The "laws of thought" approach

Several Greek schools developed various forms of **logic**, which provides a precise notation for

statements bout objects in the world and the relations among them.

1.1.4 Acting rationally: The rational agent approach

An **agent** is just something that acts. A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

AI has focused on the study and constructionof agents that **do the right thing**. This general paradigm is so pervasive that we might call it the **standard model**.

Limited rationality: act appropriately when there is not enough time to do all the computations one might take.

1.1.5 Beneficial machines

The standard model has been a useful guide for AI research since its inception, but it is probably not the right model in the long run. The reason is that the standard model assumes that we will supply a fully specified objective to the machine.

The problem of achieving agreement between our true preferences and the objective we put into the machine is called the *value alignment problem**: the values or objectives put intothe machine must be aligned with those of the human.

We don't want machines that are intelligent in the sense of pursuing *their* objectives; we want them to pursue *our*
objectives. Ultimately, we want agents that are **provably beneficial** to humans.

1.2 The Foundations of Artificial Intelligence

1.2.1 Philosophy

- Can formal rules be used to draw valid conclusions?
- How does the mind arise from a physical brain?
- Where does knowledge come from?
- How does knowledge lead to action?

Descartes was a proponent of **dualism**. He held that there is a part of the human mind (or soul or spirit) that is outside of nature, exempt from physical laws. Animals, on the other hand, did not possess this dual quality; they could be treated as machines.

An alternative to dualism is **materialism**, which holds that the brain's operation according to the laws of physics constitutes the mind. Free will is simply the way that the perception of available choices appears to the choosing entity.

The **empiricism** is characterized by a dictum of John Locke (1632–1704): "Nothing is in the understanding, which was not first in the senses."

The **logical positivism** holds that all knowledge can be characterized by logical theories connected, ultimately, to

observation sentences that correspond to sensory inputs; thus logical positivism combines rationalism and empiricism.

The **confirmation theory** attempted to analyze the acquisition of knowledge from experience by quantifying the de- gree of belief that should be assigned to logical sentences based on their connection to observations that confirm or disconfirm them.

Utilitarianism: that rational decision making based on maximizing utility should apply to all spheres of human activity, including public policy decisions made on behalf of many individuals. Utilitarianism is a specific kind of **consequen- tialism**: the idea that what is right and wrong is determined by the expected outcomes of an action.

In contrast, a theory of rule-based or **deontological ethics**, in which "doing the right thing" is determined not by outcomes but by universal social laws that govern allowable actions, such as "don't lie" or "don't kill." Many modern AI systems adopt exactly this approach.

1.2.2 Mathematics

- What are the formal rules to draw valid conclusions?
- What can be computed?
- How do we reason with uncertain information?

The theory of **probability** can be seen as generalizing logic to situations with uncertain information—a consideration of great importance for AI. The formalization of probability, combined with the availability of data, led to the emergence of **statistics** as a field.

The history of computation is as old as the history of numbers, but the first nontrivial **algorithm** is thought to be Euclid's algorithm for computing greatest common divisors. The **incompleteness theorem** showed that in any formal theory as strong as Peano arithmetic (the elementary theory of natural numbers), there are necessarily true statements that have no proof within the theory.

Alan Turing tried to characterize exactly which functions *are* **computable** —capable of being computed by an effective procedure. For example, no machine can tell *in general* whether a given program will return an answer on a given input or run forever.

Although computability is important to an understanding of computation, the notion of **tractability** has had an even greater impact on AI. A problem is called intractable if the time required to solve instances of the problem grows exponentially with the size of the instances.

The theory of **NP-completeness** provides a basis for analyzing the tractability of problems: any problem class to which the class of NP-complete problems can be reduced is likely to be intractable.

1.2.3 Economics

- How should we make decisions in accordance with our preferences?
- How should we do this when others may not go along?
- How should we do this when the payoff may be far in the future?

Decision theory, which combines probability theory with utility theory, provides a formal and

complete framework for individual decisions (economic or otherwise) made under uncertainty—that is, in cases where probabilistic descriptions appropriately capture the decision maker's environment.

Models based on **satisficing**—making decisions that are "good enough," rather than laboriously calculating an optimal decision—gave a better description of actual human behavior.

1.2.4 Neuroscience

- How do brains process information?

Neuroscience is the study of the nervous system, particularly the brain. *A collection of simple cells can lead to thought, action, and consciousness*.

Even with a computer of virtually unlimited capacity, we still require further conceptual breakthroughs in our under- standing of intelligence.

1.2.5 Psychology

- How do humans and animals think and act?

Wundt insisted on carefully controlled experiments in which his workers would perform a perceptual or associative task while introspecting on their thought processes. The **behaviorism** movement rejected any theory involving mental processes on the grounds that introspection could not provide reliable evidence.

Cognitive psychology views the brain as an information-processing device. Three key steps of a knowledge-based agent: (1) the stimulus must be translated into an internal representation, (2) the representation is manipulated by cognitive processes to derive new internal representations, and (3) these are in turn retranslated back into action.

Intelligence augmentation states that computers should augment human abilities rather than automate away human tasks.

1.2.6 Computer engineering

- How can we build an efficient computer?

Moore's law states that performance doubled every 18 months. **Quantum computing** holds out the promise of far greater accelerations for some important subclasses of AI algorithms.

1.2.7 Control theory and cybernetics

- How can artifacts operate under their own control?

Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize a **cost function** over time.

1.2.8 Linguistics

- How does language relate to thought?

Modern linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called **computational linguistics** or **natural language processing**.

1.3 The History of Artificial Intelligence

1.3.1 The inception of artificial intelligence (1943-1956)

The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943).

1.3.2 Early enthusiasm, great expectations (1952-1969)

The intellectual establishment of the 1950s, by and large, preferred to believe that "a machine can never do X". John McCarthy referred to this period as the "Look, Ma, no hands!" era.

1.3.3 A does of reality (1966-1973)

In almost all cases these early systems failed on more difficult problems. The illusion of unlimited computational power was not confined to problem-solving programs.

1.3.4 Expert systems (1969-1986)

The picture of problem solving that had arisen during the first decade of AI research was of a general-purpose search mechanism trying to string together elementary reasoning steps to find complete solutions. Such approaches have been called **weak methods**. The alternative to weak methods is to use more powerful, domain-specific knowledge that allows larger reasoning steps and can more easily handle typically occurring cases in narrow areas of expertise.

1.3.5 The return of neural networks (1986-present)

In the mid-1980s at least four different groups reinvented the **back-propagation** learning algorithm first developed in the early 1960s.

1.3.6 Probabilistic reasoning and machine learning (1987-present)

In the 1980s, approaches using **hidden Markov models** (HMMs) came to dominate the area. Pearl's development of **Bayesian networks** yielded a rigorous and efficient formalism for representing uncertain knowledge as well as practical algorithms for probabilistic reasoning.

1.3.7 Big data (2001-present)

Remarkable advances in computing power and the creation of the World Wide Web have

facilitated the creation of very large data sets—a phenomenon sometimes known as **big data**.

1.3.8 Deep learning (2011-present)

The term **deep learning** refers to machine learning using multiple layers of simple, adjustable computing elements. Experiments were carried out with such networks as far back as the 1970s, and in the form of **convolutional neural networks** they found some success in handwritten digit recognition in the 1990s.

1.4 The Sate of the Art

Robotic vehicles

legged locomotion

Autonomous planning and scheduling machine

translation

Speech recognition

recommendations

game playing

Image

understanding

medicine

Climate science

1.5 Risks and Benefits of AI

Lethal autonomous weapons

surveillance and persuasion

biased decision making

impact on employment

safety-critical applications

Cybersecurity

Intelligent Agents

2.1 Agents and Environments

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting

upon that environ- ment through **actuators**.

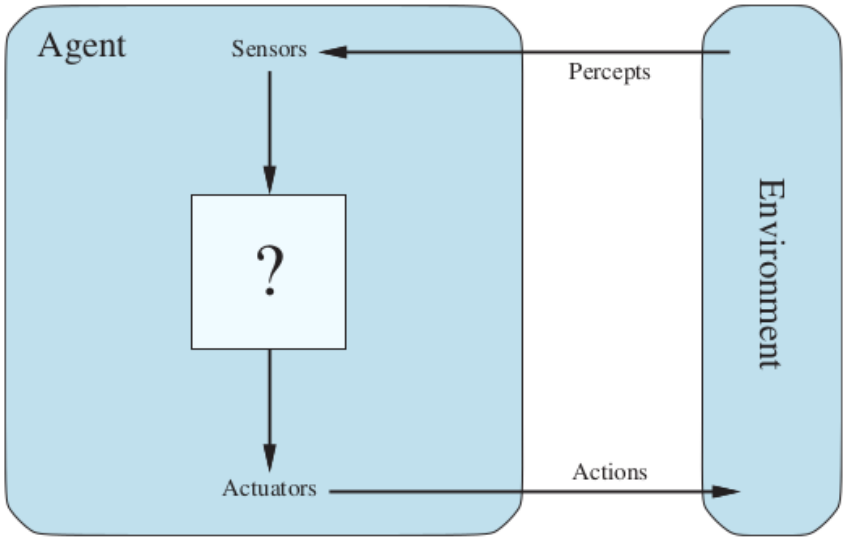


Figure 2.1 Agents interact with environments through sensors and actuators.

We use the term **percept** to refer to the content an agent's sensors are perceiving. An agent's **percept sequence** is the complete history of everything the agent has ever perceived. In general, *an agent's choice of action at any given instant can depend on its built-in knowledge and on the entire percept sequence observed to date, but not on anything it hasn't perceived*. Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

Internally, the agent function for an artificial agent will be implemented by an **agent program**. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.

2.2 Good Behavior: The Concept of Rationality

A **rational agent** is one that does the right thing.

2.2.1Performance measures

Moral philosophy has developed several different notions of the "right thing," but AI has generally stuck to one notion called **consequentialism**: we evaluate an agent's behavior by its consequences.

This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

2.2.2 Rationality

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

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

Definition of a rational agent: *For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*

2.2.3 Omniscience, learning, and autonomy

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

reality. Rationality maximizes *expected* performance, while perfection maximizes

actual performance.

Doing actions *in order to modify future percepts*—sometimes called **information gathering**. Our definition requires a rational agent not only to gather information but also to **learn** as much as possible from what it perceives.

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts and learning processes, we say that the agent lacks **autonomy**. A rational agent should be autonomous—it should learn what it can to compensate for partial or incorrect prior knowledge.

2.3 The Nature of Environments

2.3.1 Specifying the task environment

The **task environment** is composed of **PEAS** (Performance, Environment, Actuators, Sensors).

2.3.2 Properties of task environments

FULLY OBSERVABLE VS. PARTIALLY OBSERVABLE: If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data. If the agent has no sensors at all then the environment is **unobservable**.

SINGLE-AGENT VS. MULTIAGENT: The agent-design problems in multiagent environments are often quite dif- ferent from those in single-agent environments.

DETERMINISTIC VS. NONDETERMINISTIC: If the next state of the environment is completely determined by the current state and the action executed by the agent(s), then we say the environment is deterministic; otherwise, it is nondeterministic.

EPISODIC VS. SEQUENTIAL: In an episodic task environment, the agent's experience is divided into atomic episodes. Crucially, the next episode does not depend on the actions taken in previous episodes. In sequential en- vironments, on the other hand, the current decision could affect all future decisions.

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. If the environment itself 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: The discrete/continuous distinction applies to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent.

KNOWN VS. UNKNOWN: In a known environment, the outcomes (or outcome probabilities if the environment is nondeterministic) for all actions are given. Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.

The performance measure itself may be unknown, either because the designer is not sure how to write it down correctly or because the ultimate user—whose preferences matter—is not known. The hardest case is *partially observable*, *multiagent*, *nondeterministic*, *sequential*, *dynamic*, *continuous*, and *unknown*.

2.4 The Structure of Agents

The job of AI is to design an **agent program** that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the **agent architecture**:

agent = architecture + program.

2.4.1 Agent programs

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

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

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

The daunting size of these tables means that (a) no physical agent in this universe will have the space to store the table;
(b) the designer would not have time to create the table; and (c) no agent could ever learn all the right table entries from its experience.

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.

2.4.2 Simple reflex Agents

The simplest kind of agent is the **simple reflex agent**. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history.

function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Right*
else if *location* = *B* **then return** *Left*

Figure 2.8 The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure **??**.

A more general and flexible approach is first to build a general-purpose interpreter for condition–action rules and then to create rule sets for specific task environments.

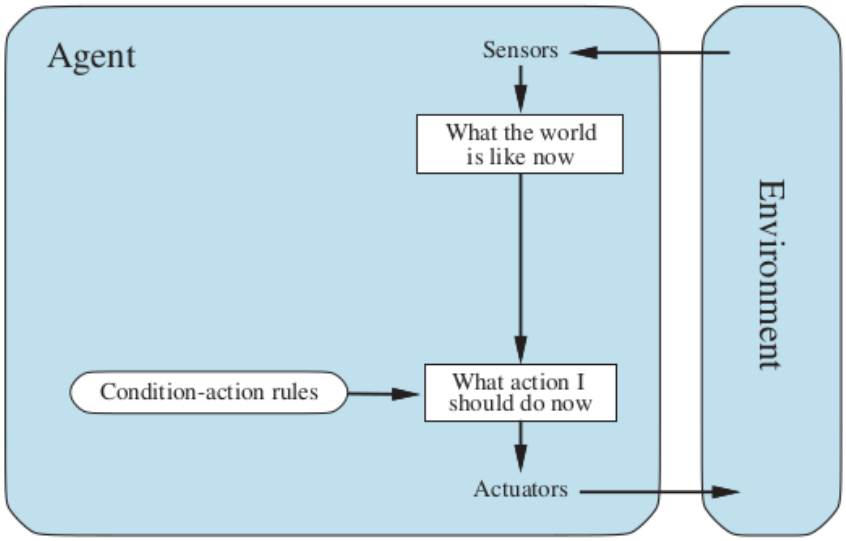


Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

The agent in Figure 2.10 will work *only if the correct decision can be made on the basis of just the current percept—that is, only if the environment is fully observable*. Even a little bit of unobservability can cause serious trouble. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments. Escape from infinite loops is possible if the agent can **randomize** its actions.

2.4.3 Model-based reflex agents

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now*. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program in some form. First, we need some information about how the world changes over time, which can be divided roughly into two parts: the effects of the agent's actions and how the world evolves independently of the agent. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a **transition model** of the world. Second, we need some information about how the state of the world is reflected in the agent's percepts. This kind of knowledge is called a **sensor model**.

An agent that uses such models is called a **model-based agent**.

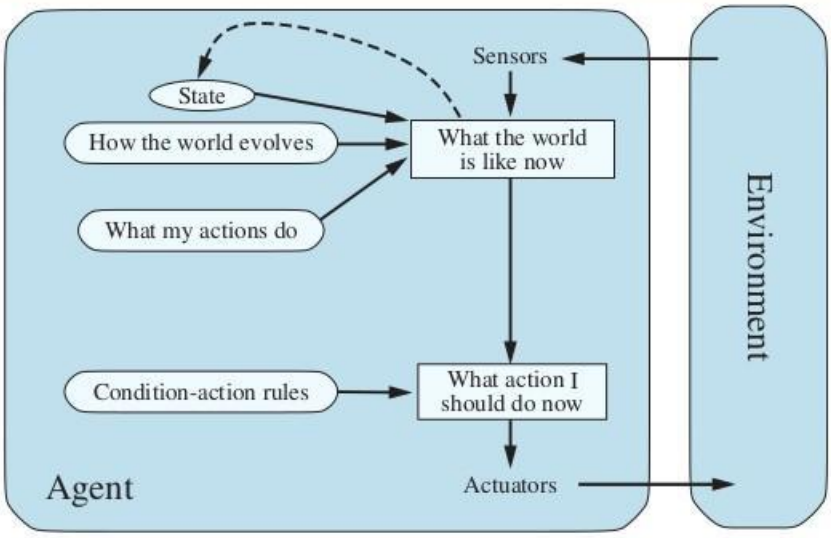


Figure 2.11 A model-based reflex agent.

function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *state*, the agent's current conception of the world state
    transition_model, a description of how the next state depends on the current state and action
    sensor_model, a description of how the current world state is reflected in the agent's percepts
    rules, a set of condition–action rules
    action, the most recent action, initially none

  state ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

2.4.4 Goal-based agents

As well as a current state description, the agent needs some sort of goal information that describes situations that are desirable. The agent program can combine this with the model (the same information as was used in the model-based reflex agent) to choose actions that achieve the goal.

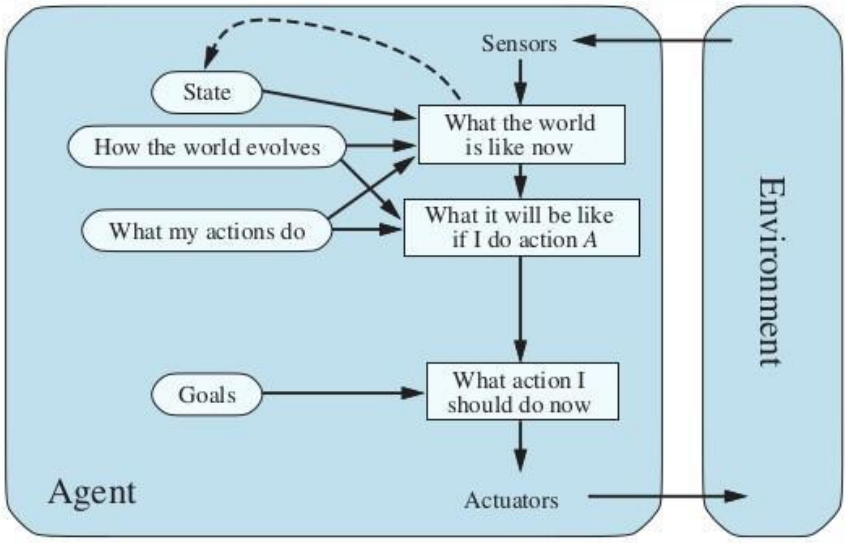


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Search and **planning** are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

2.4.5 Utility-based agents

An agent's **utility function** is essentially an internalization of the performance measure. Provided that the internal utility function and the external performance measure are in agreement, an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

A rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome.

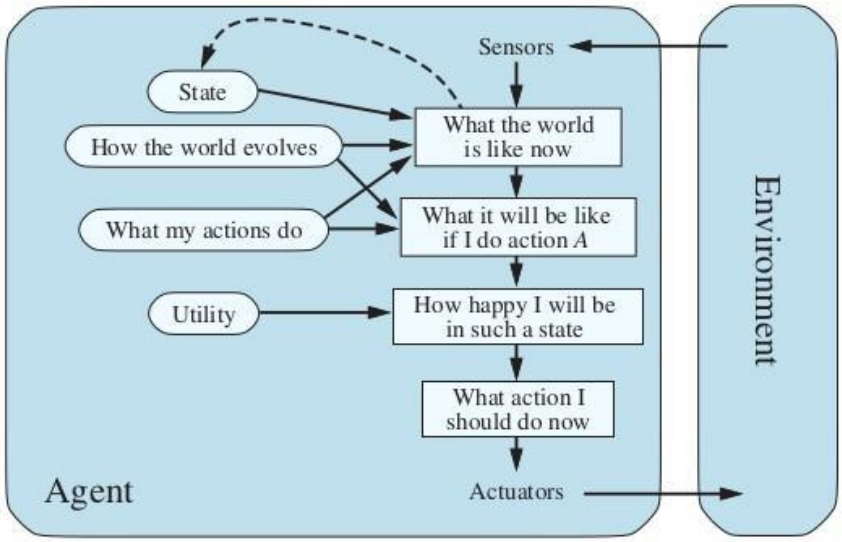


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

A **model-free agent** can learn what action is best in a particular situation without ever learning exactly how that action changes the environment.

2.4.6Learning agents

A learning agent can be divided into four conceptual components. The most important distinction is between the **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions. The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

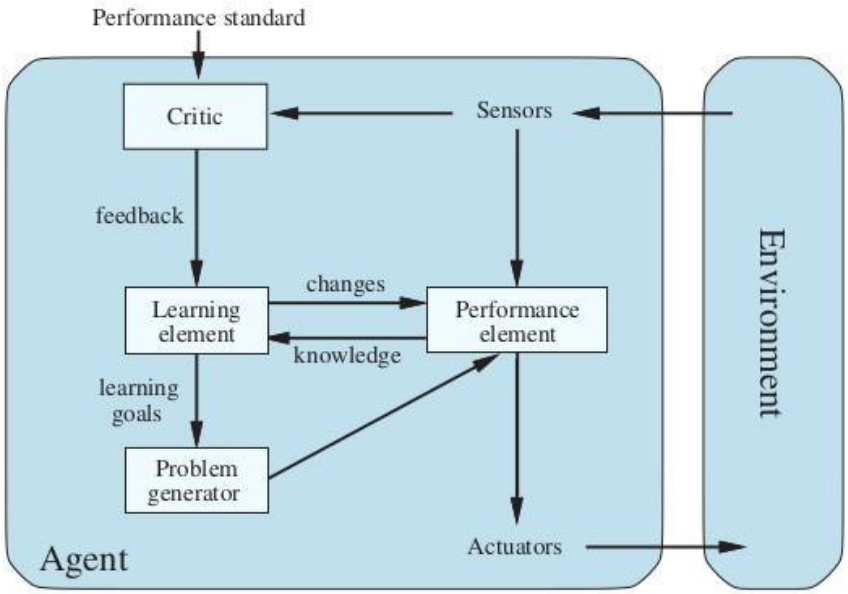


Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

The design of the learning element depends very much on the design of the performance element. Given a design for the performance element, learning mechanisms can be constructed to improve every part of the agent.

The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent's success. It is important that the performance standard be fixed.

The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and informative experiences.

Improving the model components of a model-based agent so that they conform better with reality is almost always a good idea, regardless of the external performance standard. In a sense, the performance standard distinguishes part of the incoming percept as a **reward** (or **penalty**) that provides direct feedback on the quality of the agent's behavior. More generally, *human choices* can provide information about human preferences.

Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent.

2.4.7 How the components of agent programs work

Roughly speaking, we can place the representations along an axis of increasing complexity and expressive power—atomic, factored, and structured.

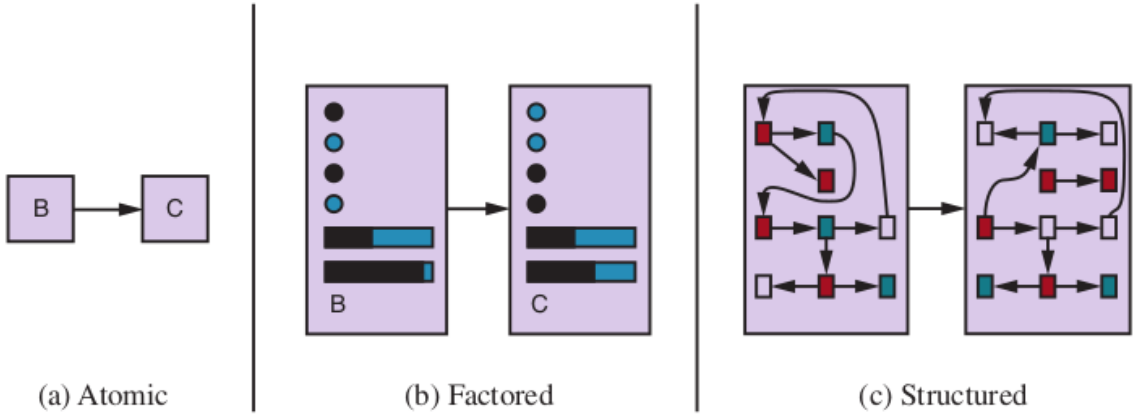


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

In an **atomic representation** each state of the world is indivisible—it has no internal structure. The standard algorithms underlying search and game-playing, hidden Markov models, and Markov decision processes all work with atomic representations.

A **factored representation** splits up each state into a fixed set of **variables** or **attributes**, each of which can have a **value**. Many important areas of AI are based on factored representations, including constraint satisfaction algorithms, propositional logic, planning, Bayesian networks, and various machine learning algorithms.

In a **Structured representation**, objects and their various and varying relationships can be described explicitly. Struc- tured representations underlie relational databases and first-order logic, first-order probability models, and much of natural language understanding.

The axis along which atomic, factored, and structured representations lie is the axis of increasing **expressiveness**. Roughly speaking, a more expressive representation can capture, at least as concisely, everything a less expressive one can capture, plus some more.

Another axis for representation involves the mapping of concepts to locations in physical memory, whether in a com- puter or in a brain. If there is a one-to-one mapping between concepts and memory locations, we call that a **localist representation**. On the other hand, if the representation of a concept is spread over many memory locations, and each memory location is employed as part of the representation of multiple different concepts, we call that a **distributed representation**. Distributed representations are more robust against noise and information loss.

PART 3
PROBLEM-SOLVING

Solving Problems by Searching

When the correct action to take is not immediately obvious, an agent may need to *plan ahead*: to consider a *sequence* of actions that form a path to a goal state. Such an agent is called a **prblem-solving agent**, and the computational process it undertakes is called **search**.

Problem-solving agents use **atomic** representations, that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms. Agents that use **factored** or **structured** representations of states are called **planning agents**.

We distinguish between **informed** algorithms, in which the agent can estimate how far it is from the goal, and **un-informed** algorithms, where no such estimate is available.

3.1 Problem-Solving Agents

If the agent has no additional information—that is, if the environment is **unknown**—then the agent can do no better than to execute one of the actions at random. For now, we assume that our agents always have access to information about the world. With that information, the agent can follow this four-phase problem-solving process:

- **GOAL FORMULATION**: Goals organize behavior by limiting the objectives and hence the actions to be con- sidered.
- **PROBLEM FORMULATION**: The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world.
- **SEARCH**: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution**.
- **EXECUTION**: The agent can now execute the actions in the solution, one at a time.

It is an important property that in a fully observable, deterministic, known environment, *the solution to any problem is a fixed sequence of actions*. The **open-loop** system means that ignoring the percepts breaks the loop between agent and environment. If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent would be safer using a **closed-loop** approach that monitors the percepts.

In partially observable or nondeterministic environments, a solution would be a branching strategy that recommends different future actions depending on what percepts arrive.

3.1.1 Search problems and solutions

A search **problem** can be defined formally as follows:

- A set of possible **states** that the environment can be in. We call this the **state space**.

- The **initial state** that the agent starts in.
- A set of one or more **goal states**. We can account for all three of these possibilities by specifying an *IsGoa()* method for a problem.
- The **actions** available to the agent. Given a state *s*, *Action*(*s*) returns a finite set of actions that can be executed in *s*. We say that each of these actions is **applicable** in *s*.
- A **transition model**, which describes what each action does. *Resul*(*s*, *a*) returns the state that results from doing action *a* in state *s*.
- An **action cost function**, denote by *ActionCos*(*s*, *a*, *s*) when we are programming or *c*(*s*, *a*, *s*) when we are doing math, that gives the numeric cost of applying action *a* in state *s* to reach state *s*.

A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state. We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs. An **optimal solution** has the lowest path cost among all solutions.

The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.

3.1.2 Formulating problems

The process of removing detail from a representation is called **abstraction**. The abstraction is *valid* if we can elaborate any abstract solution into a solution in the more detailed world. The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem.

3.2 Example Problems

A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a con- cise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms. A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

3.2.1 Standardized problems

A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.

- **Vacuum world**
- **Sokoban puzzle**
- **Sliding-tile puzzle**

3.2.2 Real-world problems

- **Route-finding problem**
- **Touring problems**

- **Traveling salesperson problem (TSP)**
- **VLSI layout problem**
- **Robot navigation**
- **Automatic assembly sequencing**

3.3 Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure. We consider algorithms that superimpose a **search tree** over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.

The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

The **frontier** separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

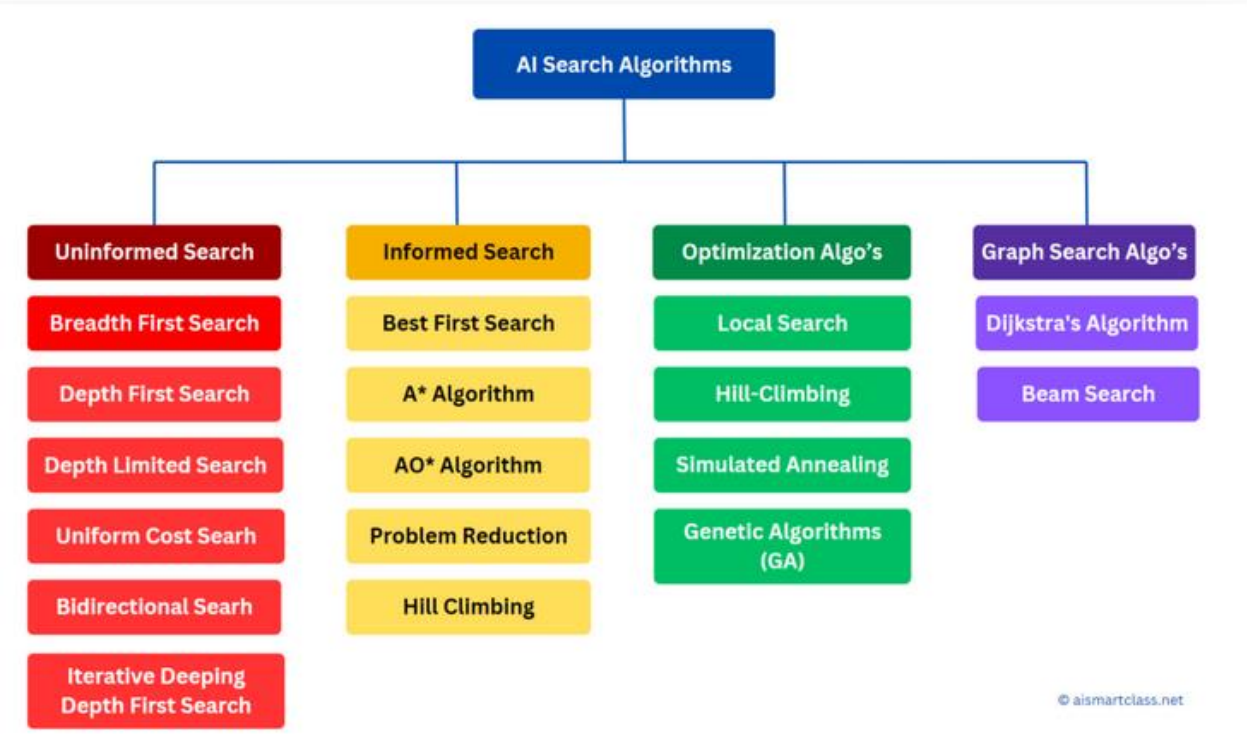


Fig: types of search and optimization algorithms

3.4 Uninformed Search Strategies

Uninformed search algorithms, also known as **Blind Search Algorithms**. These algorithms are called "uninformed" because they lack additional knowledge about the problem beyond what is directly provided by the problem definition. As a result, they may require more exploration and can be less efficient in certain scenarios compared to informed search algorithms, especially in larger or more complex problem spaces**.**
These algorithms do not have access to heuristic information or other forms of guidance that could help direct the search toward the goal state more efficiently.

3.4.1 Breadth-first search

When all actions have the same cost, an appropriate strategy is **breadth-first search**, in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth d,, it has already generated all the nodes at depth $d - 1$,, so if one of them were a solution, it would have been found.

All the nodes remain in memory, so both time and space complexity are O(bd). The memory requirements are a bigger problem for breadth-first search than the execution time. In general, exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.

3.4.2 Dijkstra's algorithm or uniform-cost search

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's algorithm by the theoretical computer science community, and **uniform-cost search** by the AI community.

The complexity of uniform-cost search is characterized in terms of C^*, the cost of the optimal solution, and ϵ, a lower bound on the cost of each action, with $\epsilon > 0$. Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d.

When all action costs are equal, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}, and uniform-cost search is similar to breadth-first search.

3.4.3 Depth-first search and the problem of memory

Depth-first search always expands the *deepest* node in the frontier first. It could be $FirstSearch$ where the evaluation function f is the negative of the depth.

For problems where a tree-like search is feasible, depth-first search has much smaller r
tree-like search takes time proportional to the number of states, and has memory com
is the branching factor and m is the maximum depth of the tree.

A variant of depth-first search called **backtracking search** uses even less memory.

3.4.4 Depth-limited and iterative deepening search

To keep depth-first search from wandering down an infinite path, we can use **depth-limited search**, a version of depth-first search in which we supply a depth limit, l, and treat all nodes at depth l as if they had no successors. The time complexity is $O(b^l)$ and the space complexity is $O(bl)$

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Figure 3.12 Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than ℓ. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

Iterative deepening search solves the problem of picking a good value for l by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth- limited search returns the *failure* value rather than the *cutoff* value.

Its memory requirements are modest: $O(bd)$ when there is a solution, or $O(bm)$ on finite state spaces with no solution. The time complexity is $O(bd)$ when there is a solution, or $O(bm)$ when there is none.

In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known.

3.4.5 Bidirectional search

An alternative approach called **bidirectional search** simultaneously searches forward from the initial state and back- wards from the goal state(s), hoping that the two searches will meet.

```
function BiBF-Search(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← Node(problem_F.Initial)                    // Node for a start state
  node_B ← Node(problem_B.Initial)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.State and value node_F
  reached_B ← a lookup table, with one key node_B.State and value node_B
  solution ← failure
  while not Terminated(solution, frontier_F, frontier_B) do
    if f_F(Top(frontier_F)) < f_B(Top(frontier_B)) then
      solution ← Proceed(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← Proceed(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function Proceed(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← Pop(frontier)
  for each child in Expand(problem, node) do
    s ← child.State
    if s not in reached or Path-Cost(child) < Path-Cost(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← Join-Nodes(dir, child, reached_2[s]))
        if Path-Cost(solution_2) < Path-Cost(solution) then
          solution ← solution_2
  return solution
```

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function Join-Nodes) to form a solution. The first solution we get is not guaranteed to be the best; the function Terminated determines when to stop looking for new solutions.

3.4.6 Comparing uninformed search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

3.5 Informed (Heuristic) Search Strategies

An **informed search** strategy uses domain–specific hints about the location of goals to find colutions more efficiently than an uninformed strategy. The hints come in the form of a **heuristic function**, denoted $h(n)$:

$h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

3.5.1 Greedy best-first search/ Best-first search

Greedy best-first search is a form of best-first search that expands first the node with the lowest $h(n)$ value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function $f(n) = h(n)$.

In **best-first search** we choose a node, *n*, with minimum value of some **evaluation function**,

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section **??**. See Appendix B for **yield**.

Search data structures

A node in the tree is represented by a data structure with four components
• **node.State**: the state to which the node corresponds;
• **node.P arent:** the node in the tree that generated this node;
• **node.Action**: the action that was applied to the parent's state to generate this node;
• **node.P athCost:** the total cost of the path from the initial state to this node. In mathematical formulas, we use g(node) as a synonym for P athCost.
Following the P ARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution.
We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a frontier are:
• **IsEmpty(frontier)** returns true only if there are no nodes in the frontier.
• **P op(frontier)** removes the top node from the frontier and returns it.
• **Top(frontier**) returns (but does not remove) the top node of the frontier.
• **Add(node, frontier)** inserts node into its proper place in the queue.
Three kinds of queues are used in search algorithms: A **node** in the tree is represented by a data structure with four components

Three kinds of queues are used in search algorithms:

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, . It is used in best-first search.
- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
- A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node; we shall see it is used in depth-first search.

Redundant paths

A cycle is a special case of a **redundant path**.

As the saying goes, *algorithms that cannot remember the past are doomed to repeat it*. There are three approaches to this issue.

First, we can remember all previously reached states (as best-first search does), allowing us to detect all redundant paths, and keep only the best path to each state.

Second, we can not worry about repeating the past. We call a search algorithm a **graph search** if it checks for redundant paths and a **tree-like search** if it does not check.

Third, we can compromise and check for cycles, but not for redundant paths in general.

Measuring problem-solving performance

- **COMPLETENESS**: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- **COST OPTIMALITY**: Does it find a solution with the lowest path cost of all solutions?
- **TIME COMPLEXITY**: How long does it take to find a solution?
- **SPACE COMPLEXITY**: How much memory is needed to perform the search?

To be complete, a search algorithm must be **systematic** in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state.

In theoretical computer science, the typical measure of time and space complexity is the size of the state-space graph,|V | + |E|, where |V | is the number of vertices (state nodes) of the graph and |E| is the number of edges (distinct state/action pairs). For an implicit state space, complexity can be measured in terms of d, the depth or number of actions in an optimal solution; m, the maximum number of actions in any path; and b, the branching factor or number of successors of a node that need to be considered.

3.5.2 A* Search:

The A* (pronounced “A-star”) algorithm is a highly versatile and widely used graph traversal and pathfinding algorithm. It’s a popular choice in various applications, including game development, robotics, and navigation systems, due to its ability to intelligently find optimal paths within

complex environments.

What Is the A-Star Algorithm?
At its core, A* is a best-first search algorithm. It aims to find the least-cost path (shortest path) from a given starting node to a goal node in a weighted graph. The weights on the graph's edges can represent distances, travel times, costs, or other factors relevant to the problem domain.
The key to A*'s efficiency lies in its evaluation function:
f(n) = g(n) + h(n)
where:

- **n:** The current node being evaluated
- **f(n):** The estimated total cost of the cheapest solution path from the start to the goal, passing through node 'n'.
- **g(n):** The actual cost to reach node 'n' from the start node.
- **h(n):** A heuristic estimate of the cost from node 'n' to the goal node.

Algorithm:

Initialization:

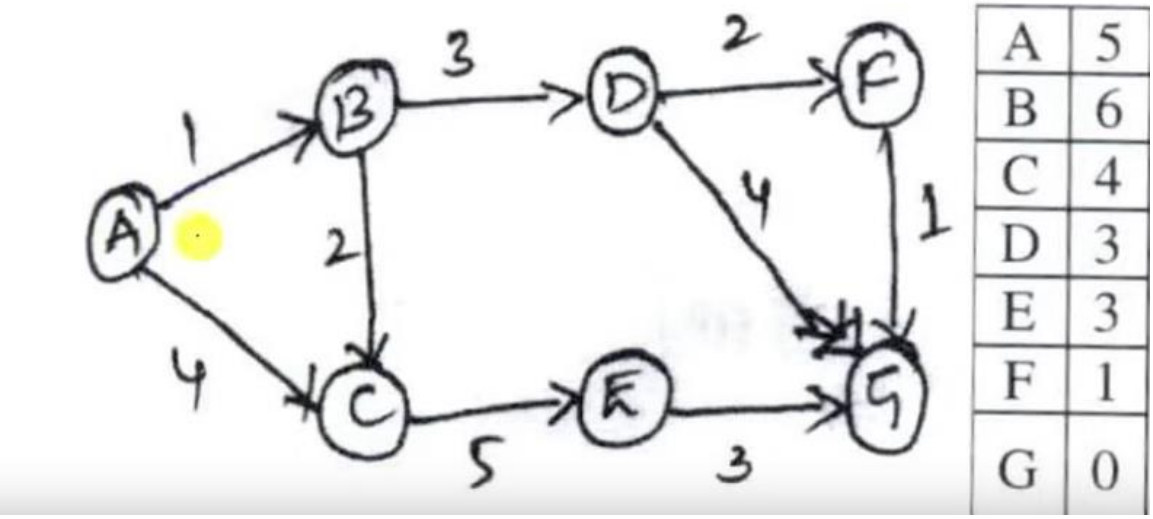
1. **Define the start and goal nodes:** Identify the starting point and the desired destination in the graph.
2. **Initialize the open and closed lists:** Create two lists: the open list (nodes to be evaluated) and the closed list (nodes already evaluated).
3. **Add the start node to the open list:** Initialize the open list with the starting node.

Main Loop:

1. **Select the best node:** Choose the node from the open list with the lowest f-score (f(n) = g(n) + h(n)).
   - `g(n)` is the cost from the start node to the current node.
   - `h(n)` is the heuristic estimate of the cost from the current node to the goal.
2. **Move the node to the closed list:** Mark the selected node as evaluated.
3. **Check for the goal:** If the selected node is the goal node, reconstruct the path and terminate.
4. **Generate successors:** Identify all neighboring nodes connected to the current node.

5. **Evaluate successors:**
   - For each successor:
     - Calculate its g, h, and f scores.
     - If the successor is not in the open or closed list, add it to the open list.
     - If the successor is already in the open or closed list, compare the new f-score with the existing one. If the new f-score is lower, update the successor's parent and f-score.
6. **Repeat:** Go back to step 1 until the open list is empty (no path found) or the goal is reached.

A* Algorithm Example:



A	5
B	6
C	4
D	3
E	3
F	1
G	0

A* Algorithm Execution:

Initial:

- **Start at A**
- g(A)=0, h(A)=5, f(A)=0+5=5
- OPEN = [(A, 5)], CLOSED = []

Step-by-step Execution:

1. Expand A:

- Move A to CLOSED
- Neighbors: B (g=1, h=6, f=7), C (g=4, h=4, f=8)
- OPEN = [(B,7), (C,8)], CLOSED = [A]

2. Expand B (lowest f):

- Move B to CLOSED
- Neighbors:
  - D: g=1+3=4, h=3, f=7

    - C already in OPEN with g=4, new g=1+2=3 (better), update C to f=3+4=7
- OPEN = [(D,7), (C,7)], CLOSED = [A, B]

3. Expand D (tie, pick D):

- Move D to CLOSED
- Neighbors:
    - F: g=4+2=6, h=1, f=7
    - G: g=4+4=8, h=0, f=8
- OPEN = [(C,7), (F,7), (G,8)], CLOSED = [A, B, D]

4. Expand C:

- Move C to CLOSED
- Neighbor:
    - E: g=3+5=8, h=3, f=11
- OPEN = [(F,7), (G,8), (E,11)], CLOSED = [A, B, D, C]

5. Expand F:

- Move F to CLOSED
- Neighbor:
    - G: g=6+1=7, h=0, f=7 (better than existing f=8), update G
- OPEN = [(G,7), (E,11)], CLOSED = [A, B, D, C, F]

6. Expand G:

- Goal reached!
- Path found

Final Path:

A → B → D → F → G

Total cost:

A→B (1) + B→D (3) + D→F (2) + F→G (1) = **7**

3.6 Heuristic Function :

A **heuristic function** in AI is a technique or method used to **guide search algorithms** and decision-making by **estimating** the "goodness" or cost of possible solutions, without fully evaluating them. Heuristics are especially used in **informed search algorithms**, like:

- A* Search
- Greedy Best-First Search
- Game-playing algorithms (Minimax with Heuristics)

Informed Search Algorithms

- Informed search algorithms are a class of algorithms used in artificial intelligence that leverage additional information about the problem to guide the search process more efficiently.
- Also known as heuristic search algorithms.
- Incorporate heuristic information.
- Utilize additional knowledge, represented by heuristic functions, to make more informed decisions during the search process, improving the efficiency of finding solutions in complex problem spaces.

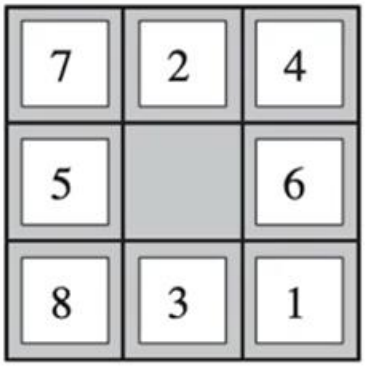
Need for Heuristic Functions

- In the vast landscapes of problem spaces, blind exploration is often impractical. This is where heuristic functions come into play.
- They provide a way for AI systems to make informed decisions about which paths to explore based on heuristic evaluations.

What Are Heuristic Functions?

- Heuristic functions estimate the cost from a given state to the goal.
- In informed search algorithms, these functions help prioritize paths, focusing on those likely to lead to optimal solutions more efficiently.
- They provide a measure of optimism about a state and aid in avoiding unnecessary exploration of unpromising paths. This characteristic ensures that the search process is not only targeted but also efficient.

Example - The 8-Puzzle Problem



- In the context of the 8-puzzle, a common heuristic is the Manhattan Distance.
- This heuristic estimates the distance of each tile from its goal position. The sum of these distances across all tiles provides a heuristic value for a given state.
- Informed search algorithms utilize this heuristic to guide the exploration toward the goal state efficiently.

Advantages of Heuristic Functions

- Provide Efficiency without sacrificing optimality.
- Empower AI systems to navigate complex problem spaces, finding optimal solutions in a fraction of the time that a blind search would require.

How to calculate heuristics?

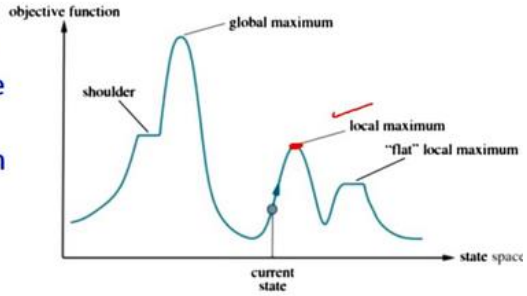
1. Define the Problem
2. Identify State Representations
3. Define the Goal
4. Identify Relevant Metrics
5. Design the Heuristic Function
6. Test and Refine

3.7 Beyond classical search

3.7.1 Hill climbing search:

Hill-climbing Search Algorithm

- Different regions in the state space landscape:
- **Local Maximum** is a state which is better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum** is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state** is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum** is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder** is a plateau region which has an uphill edge.

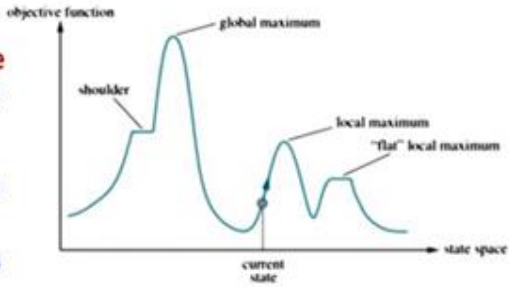


Activ

- The hill-climbing search algorithm, which is the most basic local search technique.
- At each step the current node is replaced by the best neighbor.

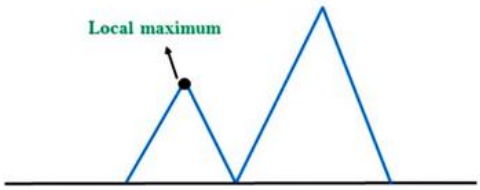
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

- Hill climbing algorithm is a Heuristic search algorithm which continuously moves in the direction of increasing value to **find the peak of the mountain or best solution to the problem.**
- It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent.**

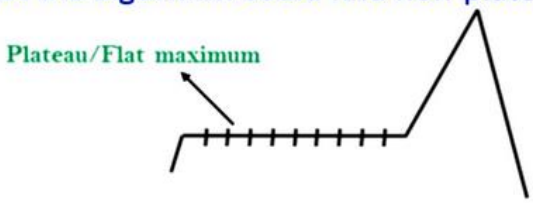


Problems in Hill Climbing Algorithm

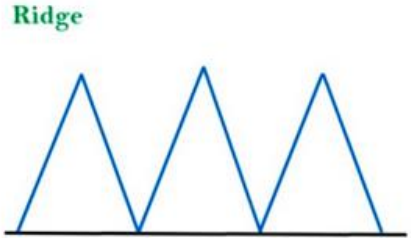
- **1. Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape.
- Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



- **2. Plateau:** A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move.
- A hill-climbing search might be lost in the plateau area.
- **Solution:** The solution for the plateau is to take big steps while searching, to solve the problem.
- Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



- **3. Ridges:** A ridge is a special form of the local maximum.
- It has an area, which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.
- **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.



3.7.2 Simulated Annealing:

Global Search Algorithms
Simulated Annealing Algorithm

Physical Annealing





- The Simulated Annealing algorithm is based upon Physical Annealing in real life.
- Physical Annealing is the process of heating up a material until it reaches an **annealing temperature** and then
- it will be **cooled down** slowly in order to change the material to a desired structure.
- When the material is hot, the molecular structure is weaker and is more susceptible to change.
- When the material cools down, the molecular structure is harder and is less susceptible to change.

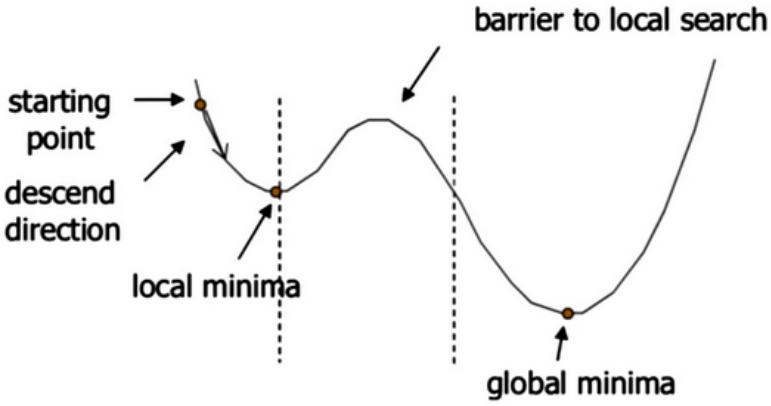
- Thermal Dynamics Equation calculates the probability that the Energy Magnitude will increase.

$$P(\Delta E) = e^{-\frac{\Delta E}{k*t}}$$

- Where ΔE - Energy Magnitude
- t - temperature
- k - Boltzmann constant.

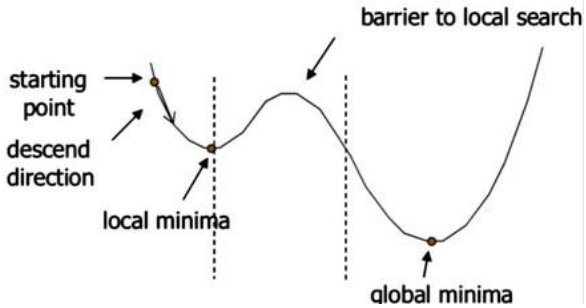
Simulated Annealing

- **Simulated Annealing** is a stochastic global search optimization algorithm and it is modified version of stochastic hill climbing.
- This algorithm appropriate for nonlinear objective functions where other local search algorithms do not operate well.
- **The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and**
- **then gradually reduce the intensity of the shaking (i.e., lower the temperature).**
- Simulated Annealing (SA) is very useful for situations where there are a lot of local minima.



Simulated Annealing- Example- ping-pong ball

- Imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum.
- The trick is to shake just **hard** enough, to bounce the ball out of local minima then the ball will reach the global minimum.



- Simulated annealing improves this strategy through the introduction of two tricks.
- This algorithm picks a *random* move instead of picking the best move.
- If the move improves the result then it accepts this random move, otherwise it accepts the move with some probability less than 1.

- Based on annealing, the algorithm uses the equation

$$e^{-\Delta D/T} > R(0, 1),$$

- Where ΔD is Change of distance,
- T is synthetic temperature (control Variable) and
- R(0,1) is a random number in the interval [0,1].

Simulated Annealing Algorithm

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  current ← *problem*.INITIAL
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE(*current*) – VALUE(*next*)
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{-\Delta E/T}$

Simulated annealing...

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

3.8 Local Search in Continuous State Spaces:

Definition:

Local search in **continuous state spaces** refers to optimization or search techniques where the variables or parameters to be optimized are **real-valued** (not discrete) and the search space is **continuous**. Instead of moving between discrete states, we navigate through a continuum of possibilities using gradient-based or stochastic methods.

Why is it Important?

- Classical search algorithms like BFS, DFS, A* are limited to **discrete** spaces.
- Many real-world problems involve continuous parameters, e.g., tuning neural network weights, robot motion planning, and engineering design.
- Local search provides efficient methods for exploring large or infinite continuous spaces.

Core Concepts:

Concept	Description
State Space	Set of all possible configurations (with real values).
Objective Function	Function to optimize (minimize or maximize).
Local Search	Start from a state, iteratively move to neighboring states based on the objective function.
Gradient	A vector of partial derivatives indicating the direction of steepest ascent.
Local Optimum	A solution that is better than all nearby solutions but not necessarily the best overall.
Global Optimum	The absolute best solution in the entire space.

Techniques Used:

1. Gradient Descent

- **Goal:** Minimize a real-valued function.
- **Method:** Move in the direction of the negative gradient.
- **Formula:**
  - Formula:

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

where:

- x_t: current state
- α: learning rate
- $\nabla f(x_t)$: gradient of the function at x_t

2. Stochastic Gradient Descent (SGD)

- Uses random samples to compute an approximate gradient.
- Common in machine learning due to efficiency on large datasets.

3. Newton's Method

- Uses second-order derivative (Hessian) for faster convergence.
- Update rule:

$$x_{t+1} = x_t - H^{-1} \nabla f(x_t)$$

4. Random Restart Hill Climbing

- Hill climbing with multiple random initial states to escape local optima.
- Suitable for non-convex landscapes.

5. Simulated Annealing

- Allows occasional "bad" moves to escape local optima.

- Gradually reduces randomness (temperature).

Applications:

- **Robotics** – trajectory optimization
- **Computer Vision** – image registration
- **Machine Learning** – training neural networks
- **Operations Research** – scheduling, layout design

Discrete vs. Continuous Local Search:

Feature	Discrete Local Search	Continuous Local Search
Variables	Finite	Infinite (real numbers)
Neighborhood	Defined by state adjacency	Defined by proximity in continuous space
Examples	8-queens, CSPs	Gradient descent, neural net training

Local search in continuous state spaces is crucial for solving optimization problems involving real-valued parameters. Techniques like gradient descent and simulated annealing help efficiently explore these spaces but come with challenges like local optima and step size tuning.

Constrain Satisfaction Problem (CSP)

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables,**X1,X2, ….Xn, and a set of constraints C1,C2,…,Cm. Each variable Xi has a nonempty **domain** D,of possible **values**.

Each constraint Ci involves some subset of variables and specifies the allowable combinations of values for that subset.

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

Some CSPs also require a solution that maximizes an **objective function**.

Example for Constraint Satisfaction Problem

Figure shows the map of Australia showing each of its states and territories. We are given the task of coloring each region either red, green, or blue in such a way that the neighboring regions have the same color. To formulate this as CSP, we define the variable to be the regions

:WA,NT,Q,NSW,V,SA, and T. The domain of each variable is the set {red,green,blue}.The constraints require neighboring regions to have distinct colors; for example, the allowable combinations for WA and NT are the pairs

{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.

The constraint can also be represented more succinctly as the inequality WA not = NT, provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as

{ WA = red, NT = green, Q = red, NSW = green, V = red,SA = blue,T = red}.

It is helpful to visualize a CSP as a constraint graph, as shown in Figure 2.29. The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.



Figure 2.29 Principle states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

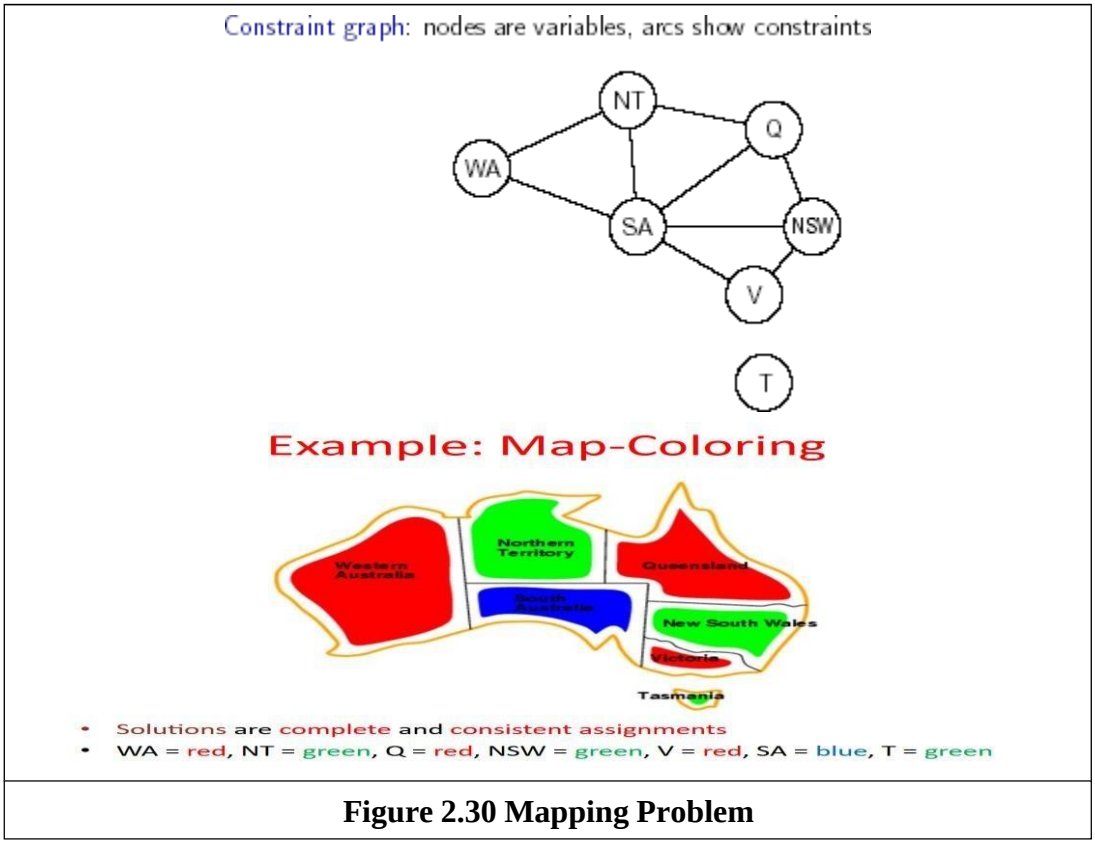


Figure 2.30 Mapping Problem

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

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

Every solution must be a complete assignment and therefore appears at depth n if there are n variables.

Depth first search algorithms are popular for CSPs

Varieties of CSPs

(i) Discrete variables Finite domains

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain

CSP, where the variables Q1,Q2,…..Q8 are the positions each queen in columns 1, ….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any

variable in a CSP is d, then the number of possible complete assignments is $O(d^n)$ – that is, exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**, whose variables can be either *true* or *false*. **Infinite domains**

Discrete variables can also have **infinite domains** – for example, the set of integers or the set of strings. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as Startjob1 + 5 <= Startjob3.

(ii) CSPs with continuous domains

CSPs with continuous domains are very common in real world. For example in operation research field, the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and manoeuvre are continuous-valued variables that must obey a variety of astronomical, precedence and power constraints. The best known category of continuous-domain CSPs is that of **linear programming** problems, where the constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables.

Varieties of constraints

(i) **unary constraints** involve a single variable.

Example : SA # green

(ii) Binary constraints involve paris of variables.

Example : SA # WA

(iii) Higher order constraints involve 3 or more variables. Example :cryptarithmetic puzzles.

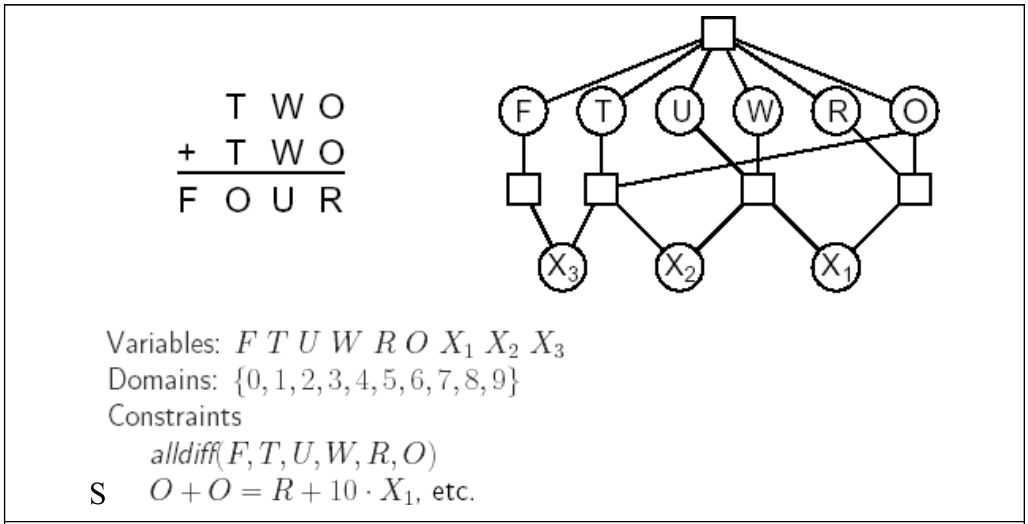


Figure 2.31 cryptarithmetic puzzles.

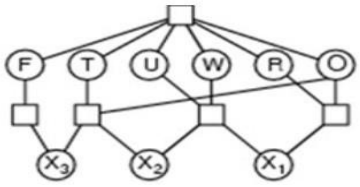
S

Example: Cryptarithmetic

Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed

```
  T W O
+ T W O
F O U R
```

- Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
- Domains: {0,1,2,3,4,5,6,7,8,9}
- Constraints: *Alldiff (F,T,U,W,R,O)*
- where XI, X2, and X3 are auxiliary variables representing the digit (0 or 1) carried over into the next column
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$



The constraint hyper graph for the cryptarithmetic problem, showing the *Alldzff constraint as well as the column addition* constraints

Figure 2.32 Cryptarithmetic puzzles-Solution

- **Each puzzle may has one or many solutions or no solution**

```
  T W O
+ T W O
F O U R
```

has 7 solutions:

1. 734 F=1 O=4 R=8 T=7 U=6 W=3
   +734

   1468

2. 765 F=1 O=5 R=0 T=7 U=3 W=6
   +765

   1530

3. 836 F=1 O=6 R=2 T=8 U=7 W=3
   +836

   1672

4. 846 F=1 O=6 R=2 T=8 U=9 W=4
   +846

   1692

5. 867 F=1 O=7 R=4 T=8 U=3 W=6
   +867

   1734

6. 928 F=1 O=8 R=6 T=9 U=5 W=2
   +928

   1856

7. 938 F=1 O=8 R=6 T=9 U=7 W=3
   +938

   1876

Figure 2.33 Numerical Solution

Backtracking Search for CSPs

The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure

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

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given 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 2.34 A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search

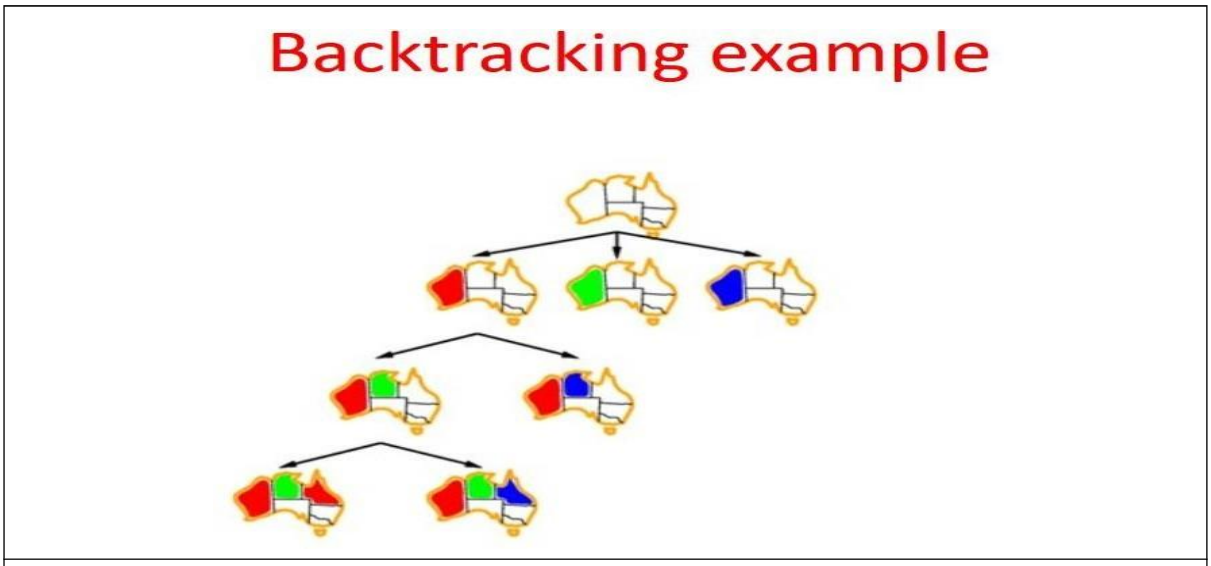


Figure 2.34 Part of the search tree generated by simple backtracking for the map-coloring problem

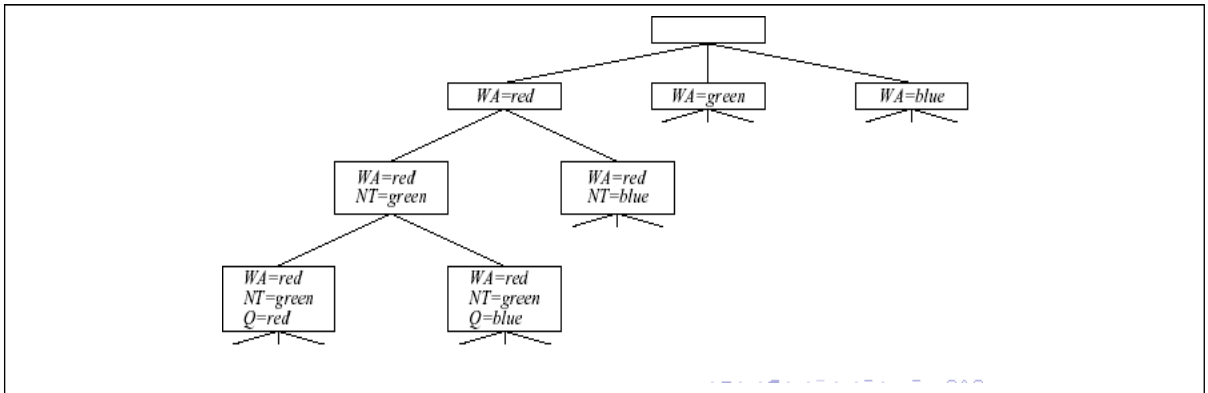


Figure 2.35 Part of search tree generated by simple backtracking for the map coloring problem.

Forward checking

One way to make better use of constraints during search is called forward checking. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking.

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

Figure 2.36 The progress of a map-coloring search with forward checking. WA = *red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q* = *green*, *green* is deleted from the domain of *NT*, *SA*, and *NSW*. After *V* = *blue*, *blue*, is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them.

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

Arc Consistency

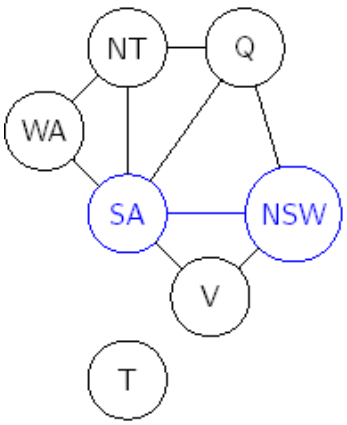


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value x of *SA*
  - There is some value y of *NSW* that is consistent with x
- Examine arcs for consistency in *both* directions

Figure 2.37 Arc Consistency

- Can define stronger forms of consistency

k-Consistency

A CSP is *k*-**consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the k-th variable

- **1-consistency (node consistency)**
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency (arc consistency)**
- **3-consistency (path consistency)**
  - Any pair of adjacent variables can be extended to a third

Figure 2.38 Arc Consistency –CSP

k-Consistency

Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

The Structure of Problems Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Subproblems

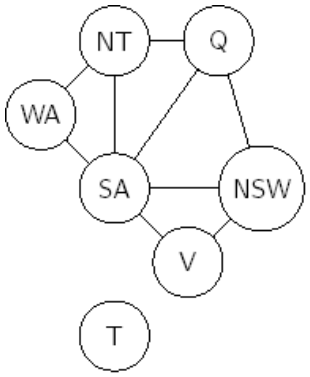


Figure: Australian Territories

- T is not connected
- Coloring T and coloring remaining nodes are **independent subproblems**
- *Any* solution for T combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Figure 2.39 Independent Subproblems

Tree-Structured CSPs

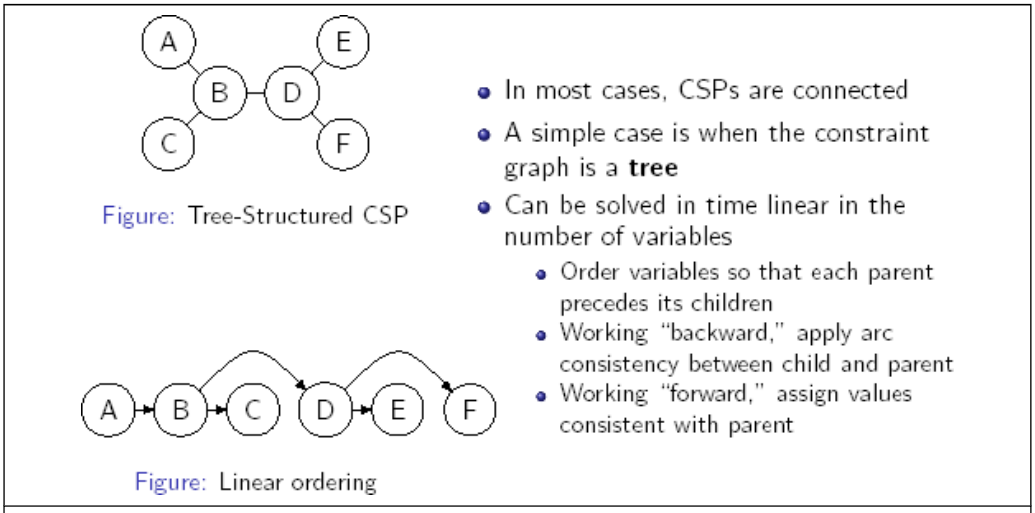


Figure 2.40 Tree-Structured CSPs

2.30 ADVERSARIAL SEARCH

Competitive environments, in which the agent's goals are in conflict, give rise to **adversarial search** problems – often known as **games.**

Games

Mathematical **Game Theory**, a branch of economics, views any **multiagent environment** as a **game** provided that the impact of each agent on the other is "significant", regardless of whether the agents are cooperative or competitive. In, **AI**, "games" are deterministic, turn-taking, two-player, zero-sum games of perfect information. This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the **utility values** at the end of the game are always equal and opposite. For example, if one player wins the game of chess(+1),the other player necessarily loses(-1). It is this opposition between the agents' utility functions that makes the situation **adversarial.**

Formal Definition of Game

We will consider games with two players, whom we will call **MAX** and **MIN**. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A **game** can be formally defined as a **search problem** with the following components:

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (*move*, *state*) pairs, each indicating a legal move and the resulting state.

- A **terminal test**, which describes when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function), which give a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values+1,-1, or 0. he payoffs in backgammon range from +192 to -192.

Game Tree

The **initial state** and **legal moves** for each side define the **game tree** for the game. Figure 2.18 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree (particularly the utility of terminal states) to determine the best move.

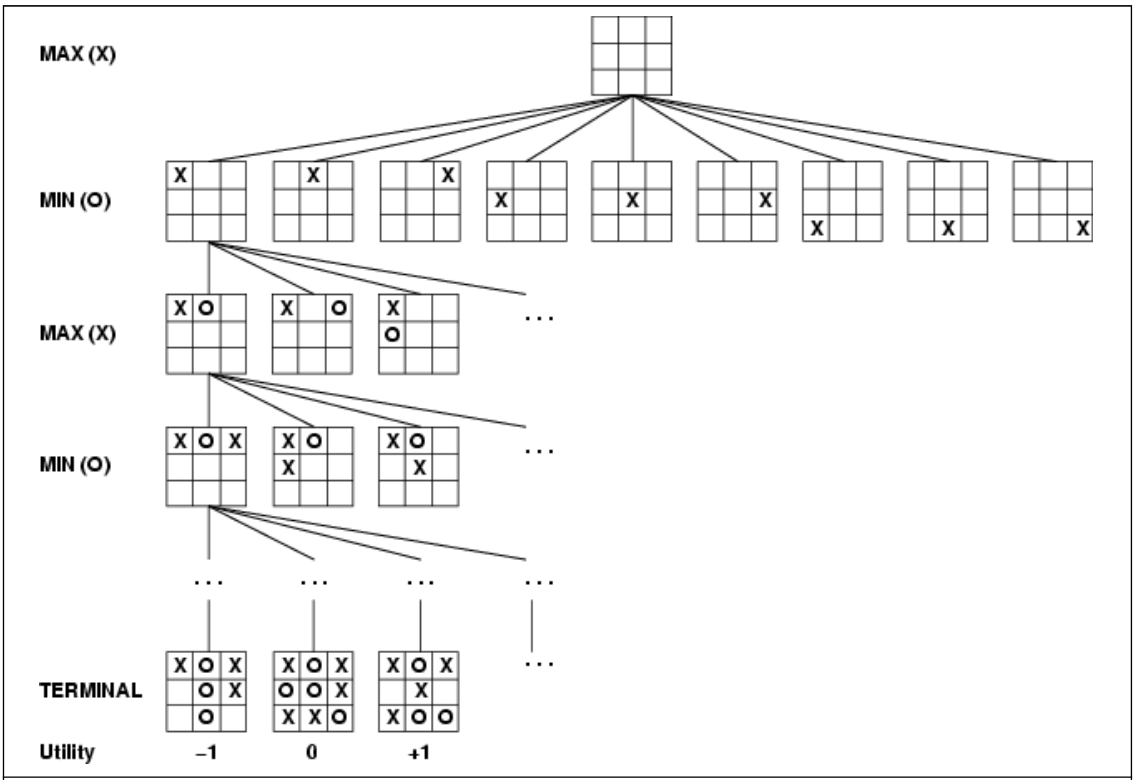


Figure 2.41 A partial search tree. The top node is the initial state, and MAX move first, placing an X in an empty square.

Optimal Decisions in Games

In normal search problem, the **optimal solution** would be a sequence of move leading to a **goal state** – a terminal state that is a win. In a game, on the other hand, MIN has something

to say about it, MAX therefore must find a contingent **strategy**, which specifies MAX's move in the **initial state**, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN those moves, and so on. An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

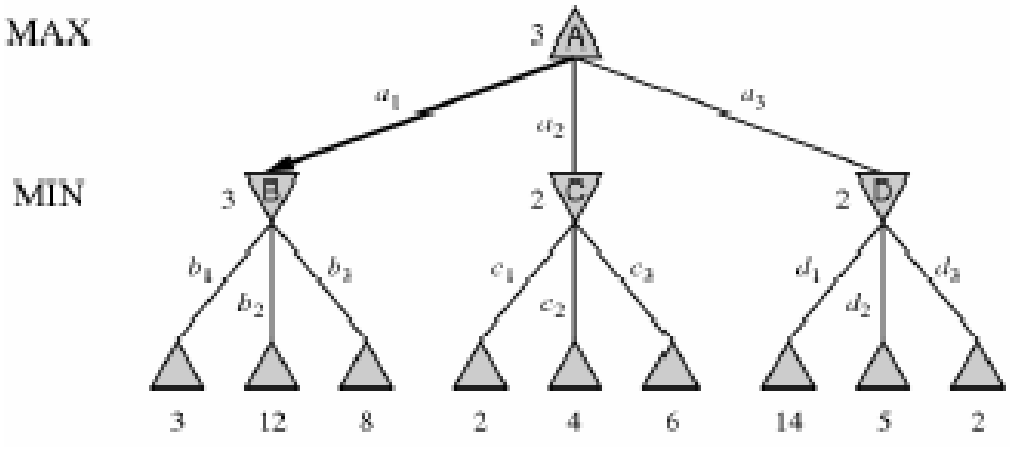


Figure 2.42 Optimal Decisions in Games

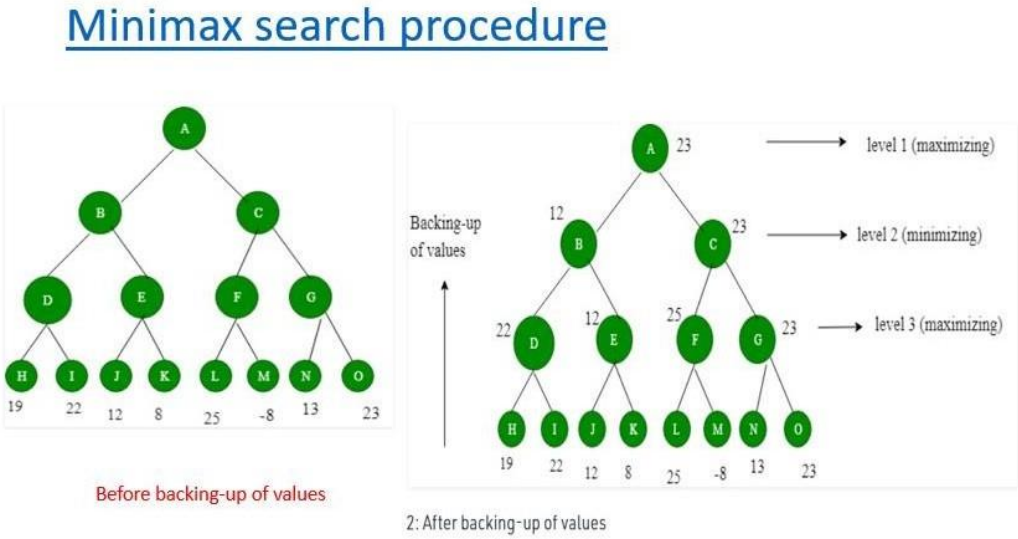


Figure 2.43 MAX-VALUE and MIN-VALUE

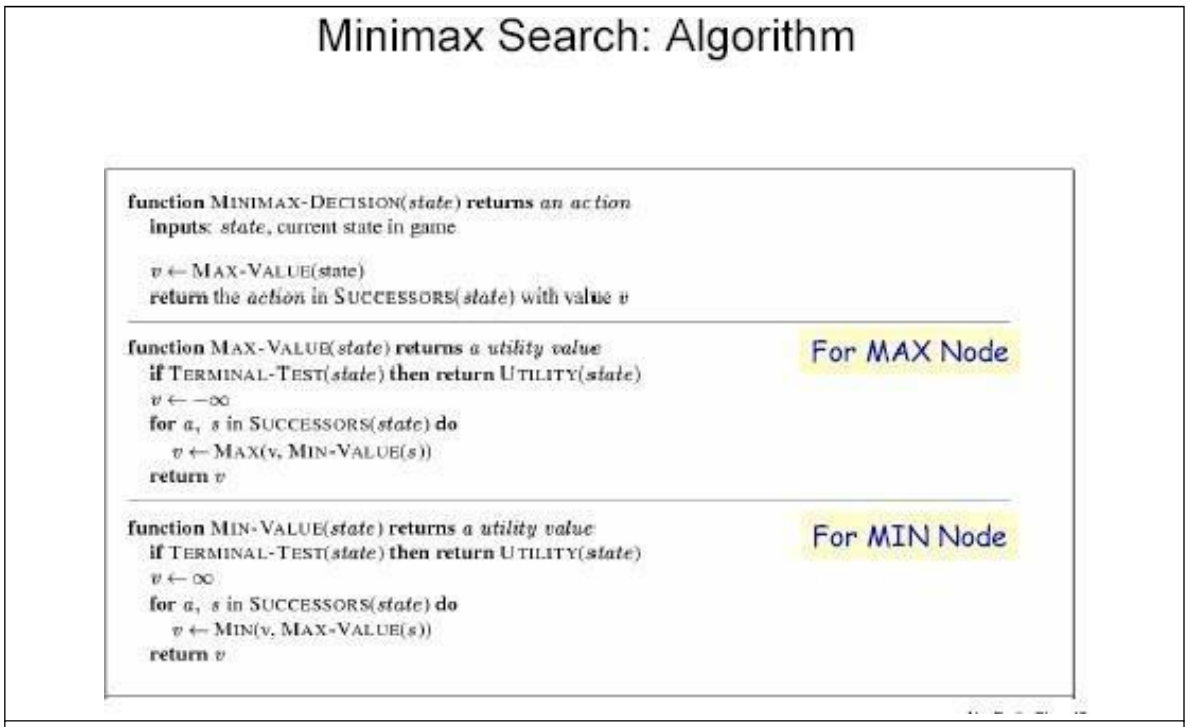


Figure 2.44 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

The minimax Algorithm

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds**.** For example in Figure 2.19,the algorithm first recourses down to the three bottom left nodes, and uses the utility function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 at the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*, and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for analgorithm that generates successors at once.

Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is **exponential** in the number of moves. Unfortunately, we can't eliminate the exponent, but

we can effectively cut it in half. By performing **pruning,** we can eliminate large part of the tree from consideration. We can apply the technique known as **alpha beta pruning**, when applied to a minimax tree, it returns the same move as **minimax** would, but **prunes away** branches that cannot possibly influence the final decision.

Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- α : the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path of MAX.
- β: the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at anode(i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN, respectively. The complete algorithm is given in Figure. The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case, it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put an other way alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.

```
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(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2.45 The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20,except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β

Key points in Alpha-beta Pruning

- Alpha: Alpha is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is – ∞.
- Beta: Beta is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is + ∞.
- The condition for Alpha-beta Pruning is that α >= β.
- Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.
- MAX will update only alpha values and MIN player will update only beta values.
- The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.
- Alpha and Beta values only be passed to child nodes.

Working of Alpha-beta Pruning

1. We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. α = -∞ and β= +∞. We will prune the node only when alpha becomes greater than or equal to beta.

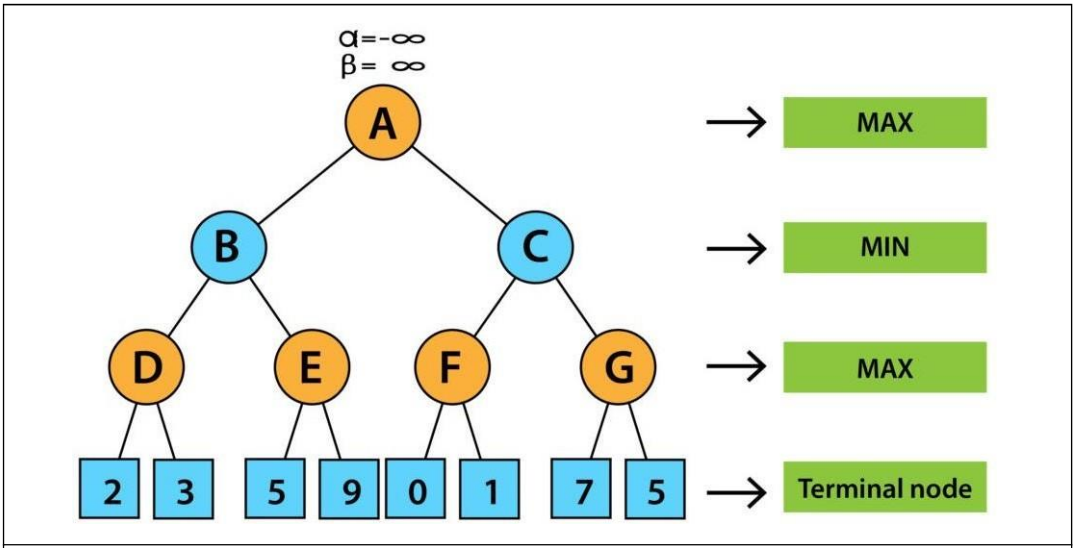


Figure 2.46 Step 1 Alpha-beta Pruning

2. Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

3. Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.

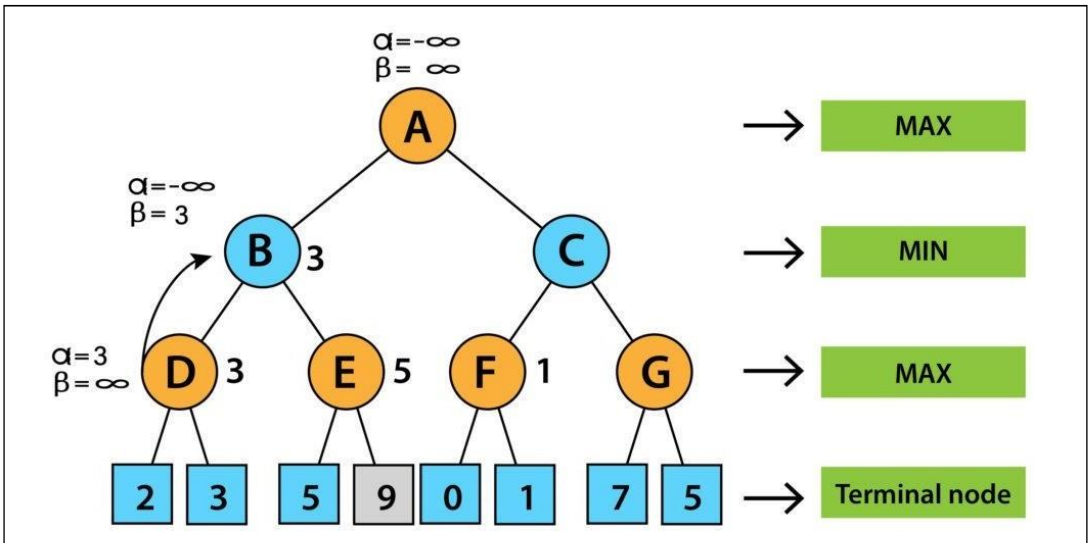


Figure 2.47 Step 2 Alpha-beta Pruning

In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

4. Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.

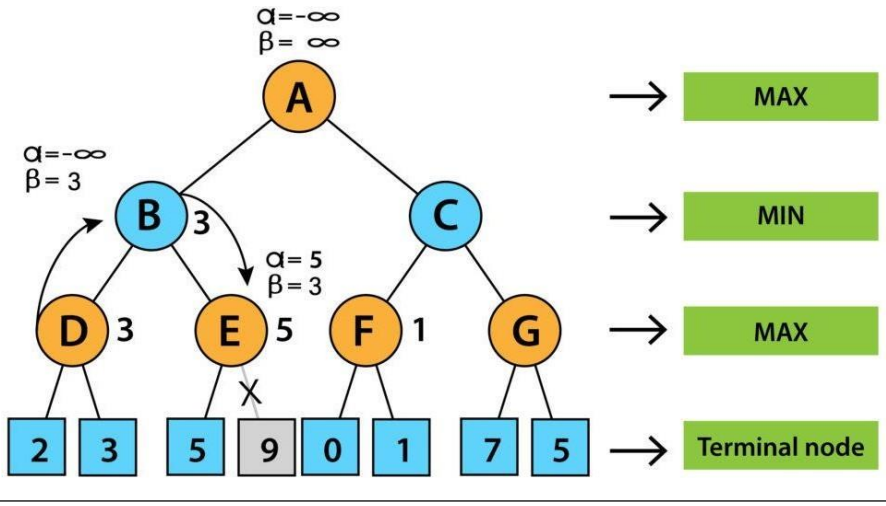


Figure 2.48 Step 3 Alpha-beta Pruning

6. In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

7. At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.

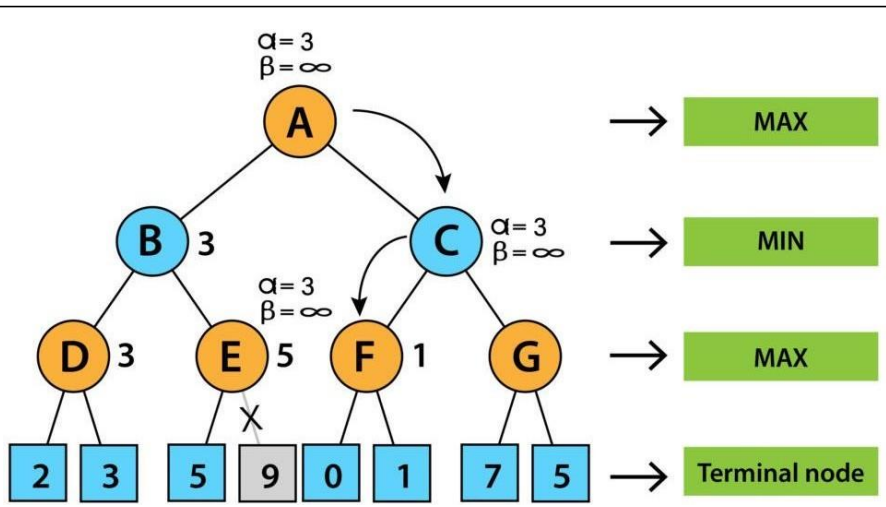


Figure 2.49 Step 4 Alpha-beta Pruning

8. Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.

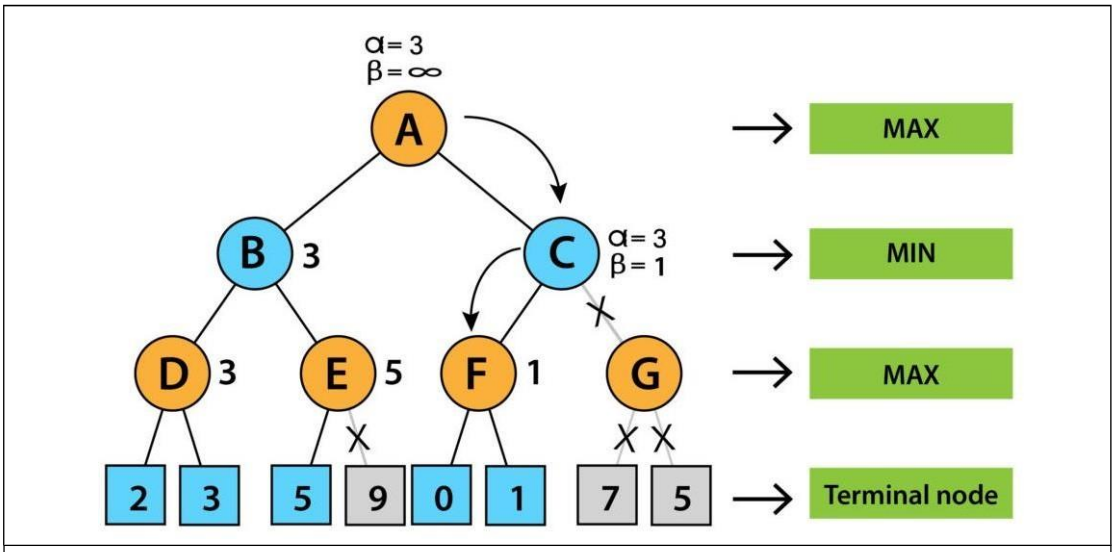


Figure 2.50 Step 5 Alpha-beta Pruning

Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.

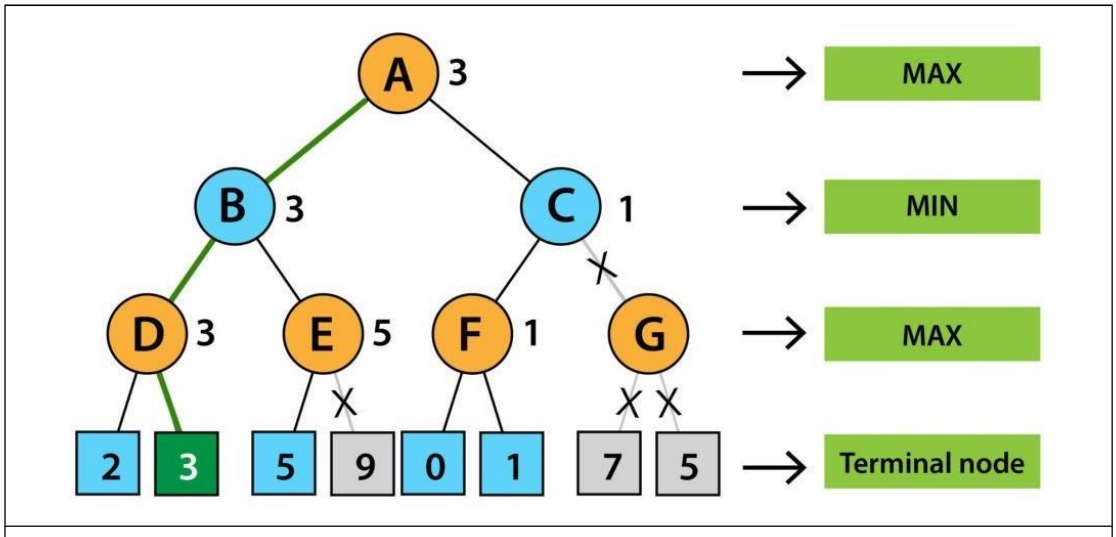


Figure 2.51 Step 6 Alpha-beta Pruning

The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.

Logic Agents and Propositional Logic



CHAPTER 7

HASSAN KHOSRAVI

SPRING2011

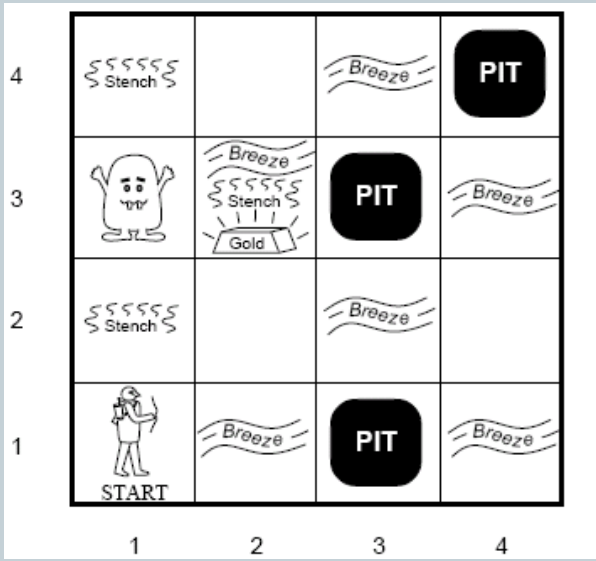
Knowledge-Based Agents



- KB = knowledge base
  - A set of sentences or facts
  - e.g., a set of statements in a logic language
- Inference
  - Deriving new sentences from old
  - e.g., using a set of logical statements to infer new ones
- A simple model for reasoning
  - Agent is told or perceives new evidence
    - E.g., A is true
  - Agent then infers new facts to add to the KB
    - E.g., KB = { A -> (B OR C) }, then given A and not C we can infer that B is true
    - B is now added to the KB even though it was not explicitly asserted, i.e., the agent inferred B

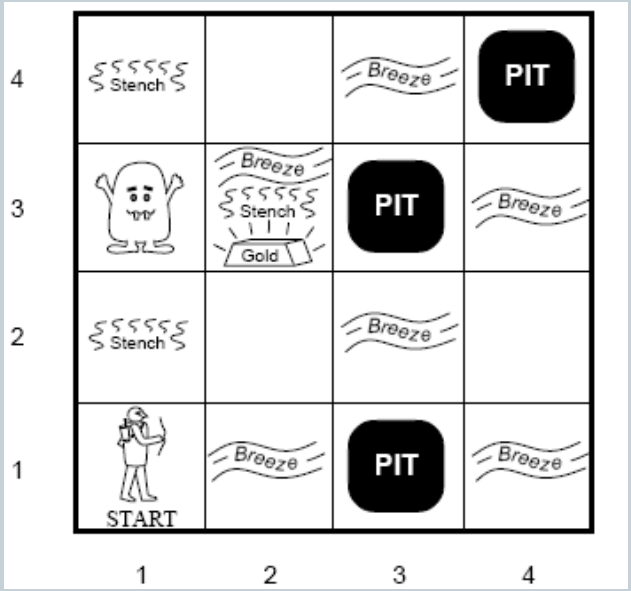
Wumpus World

- Environment
  - Cave of 4×4
  - Agent enters in [1,1]
  - 16 rooms
    - Wumpus: A deadly beast who kills anyone entering his room.
    - Pits: Bottomless pits that will trap you forever.
    - Gold



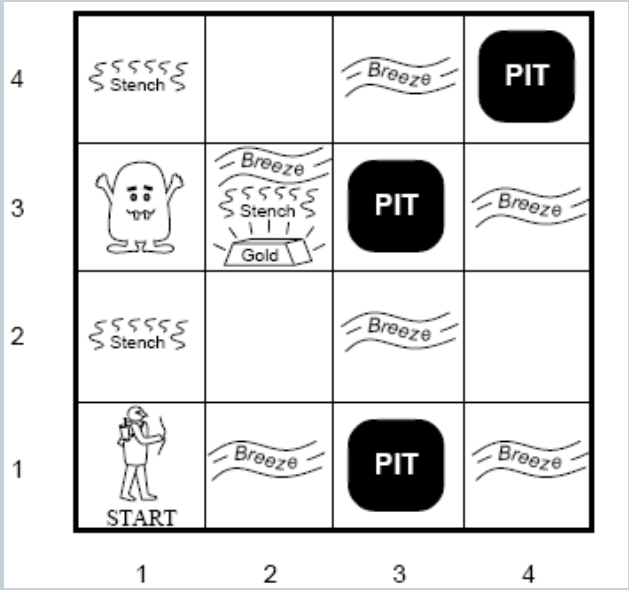
Wumpus World

- Agents Sensors:
  - Stench next to Wumpus
  - Breeze next to pit
  - Glitter in square with gold
  - Bump when agent moves into a wall
  - Scream from wumpus when killed
- Agents actions
  - Agent can move forward, turn left or turn right
  - Shoot, one shot



Wumpus World

- Performance measure
  - +1000 for picking up gold
  - -1000 got falling into pit
  - -1 for each move
  - -10 for using arrow



Reasoning in the Wumpus World

- Agent has initial ignorance about the configuration
  - Agent knows his/her initial location
  - Agent knows the rules of the environment
- Goal is to explore environment, make inferences (reasoning) to try to find the gold.
- Random instantiations of this problem used to test agent reasoning and decision algorithms

(applications? "intelligent agents" in computer games)

Exploring the Wumpus World

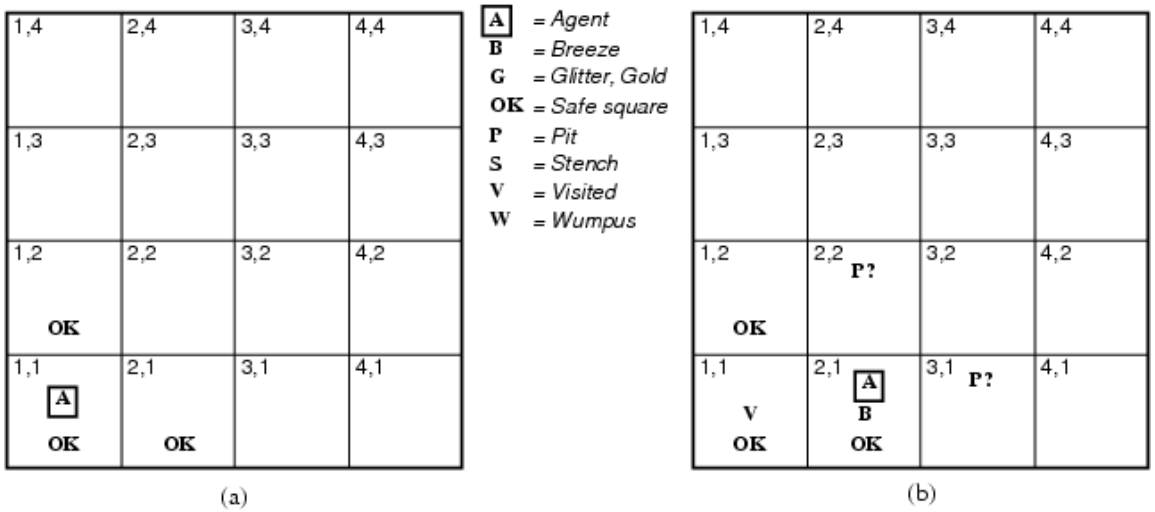


[1,1] The KB initially contains the rules of the environment.

The first percept is [*none, none,none,none,none*],

move to safe cell e.g. 2,1

Exploring the Wumpus World

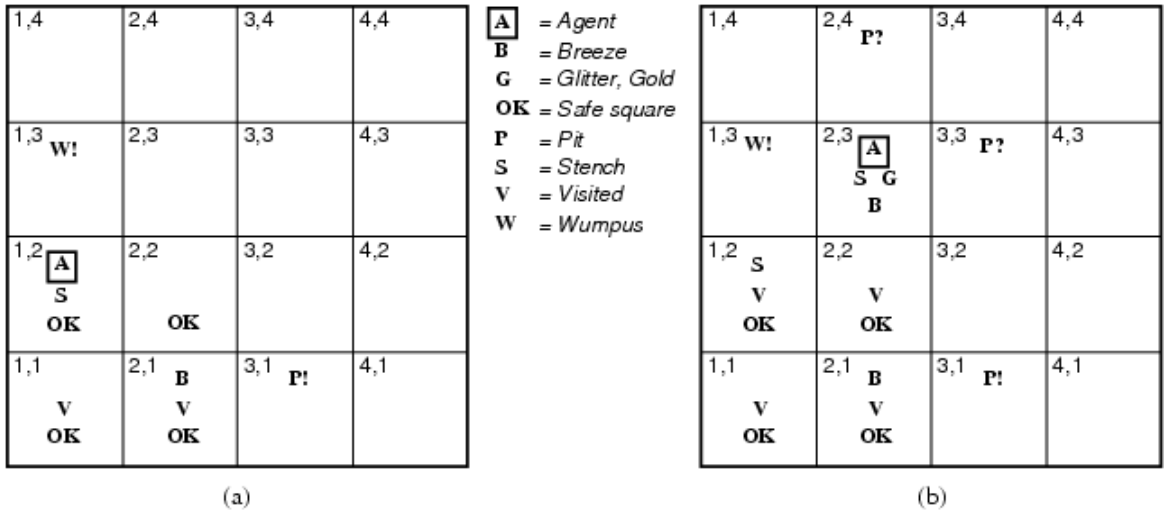


[2,1] = breeze

indicates that there is a pit in [2,2] or [3,1],

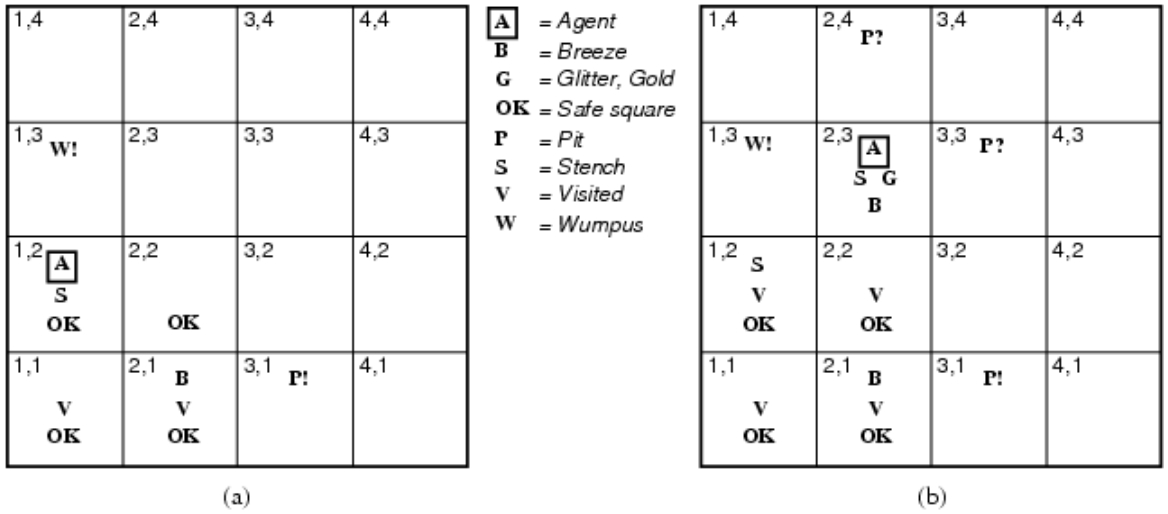
return to [1,1] to try next safe cell

Exploring the Wumpus World



[1,2] Stench in cell which means that wumpus is in [1,3] or [2,2]
YET … not in [1,1]
YET … not in [2,2] or stench would have been detected in [2,1]
(this is relatively sophisticated reasoning!)

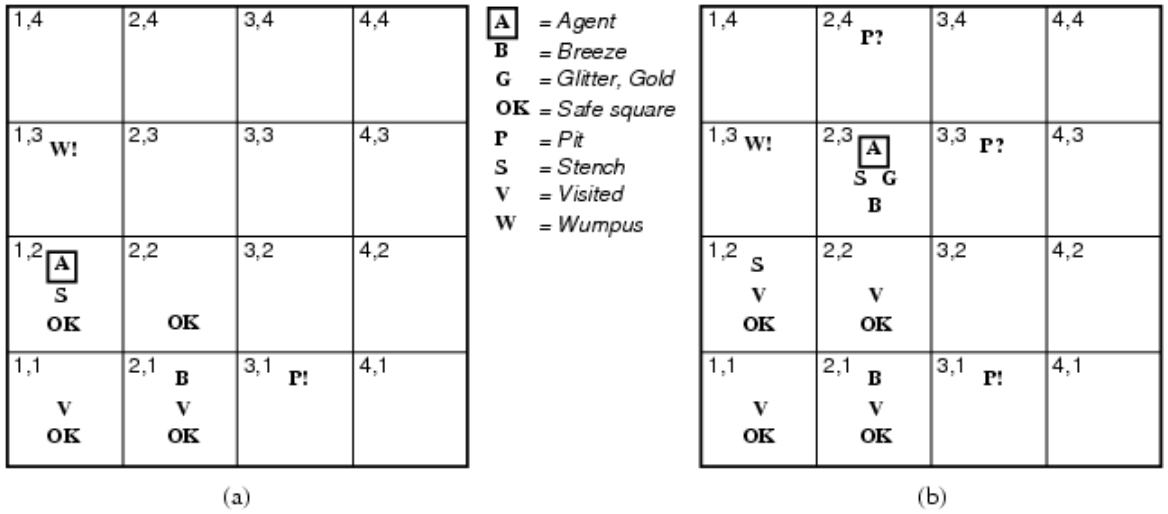
Exploring the Wumpus World



[1,2] Stench in cell which means that wumpus is in [1,3] or [2,2]
YET … not in [1,1]
YET … not in [2,2] or stench would have been detected in [2,1]
(this is relatively sophisticated reasoning!)

THUS … wumpus is in [1,3]
THUS [2,2] is safe because of lack of breeze in [1,2]
THUS pit in [1,3] (again a clever inference)
move to next safe cell [2,2]

Exploring the Wumpus World



[2,2] move to [2,3]

[2,3] detect glitter , smell, breeze
THUS pick up gold
THUS pit in [3,3] or [2,4]

What our example has shown us



- Can represent general knowledge about an environment by a set of rules and facts
- Can gather evidence and then infer new facts by combining evidence with the rules
- The conclusions are guaranteed to be correct if
  - The evidence is correct
  - The rules are correct
  - The inference procedure is correct

    -> logical reasoning
- The inference may be quite complex
  - E.g., evidence at different times, combined with different rules, etc

What is a Logic?



- A formal language
  - KB = set of sentences
- Syntax
  - what sentences are legal (well-formed)
  - E.g., arithmetic
    - X+2 >= y is a wf sentence, +x2y is not a wf sentence
- Semantics
  - loose meaning: the interpretation of each sentence
  - More precisely:
    - Defines the truth of each sentence wrt to each possible world
  - e.g,
    - X+2 = y is true in a world where x=7 and y =9
    - X+2 = y is false in a world where x=7 and y =1
  - Note: standard logic – each sentence is T of F wrt eachworld
    - Fuzzy logic – allows for degrees of truth.

Models and possible worlds

- Logicians typically think in terms of models, which are formally structured worlds with respect to which truth can be evaluated.
- m is a model of a sentence α if α is true in m
- $M(\alpha)$ is the set of all models of α
- Possible worlds ~ models
  - Possible worlds: potentially real environments
  - Models: mathematical abstractions that establish the truth or falsity of every sentence
- Example:
  - x + y = 4, where x = #men, y = #women
  - Possible models = all possible assignments of integers to x and y

Entailment



- One sentence follows logically from another

  $\alpha \models \beta$

  α entails sentence β *if and only if* β is true in all worlds where α is true.

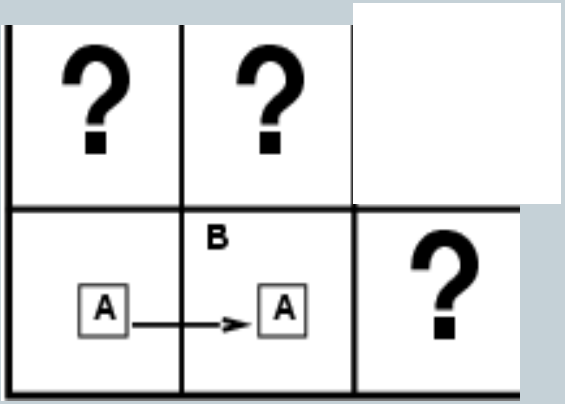
  e.g., x+y=4 |= 4=x+y

- Entailment is a relationship between sentences that is based on semantics.

Entailment in the wumpus world

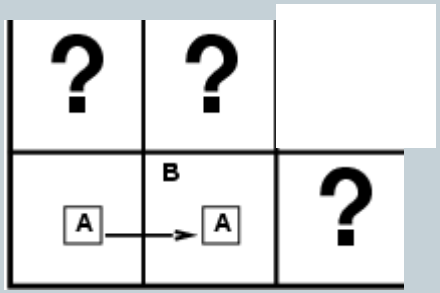
- Consider possible models for *KB* assuming only pits and a reduced Wumpus world

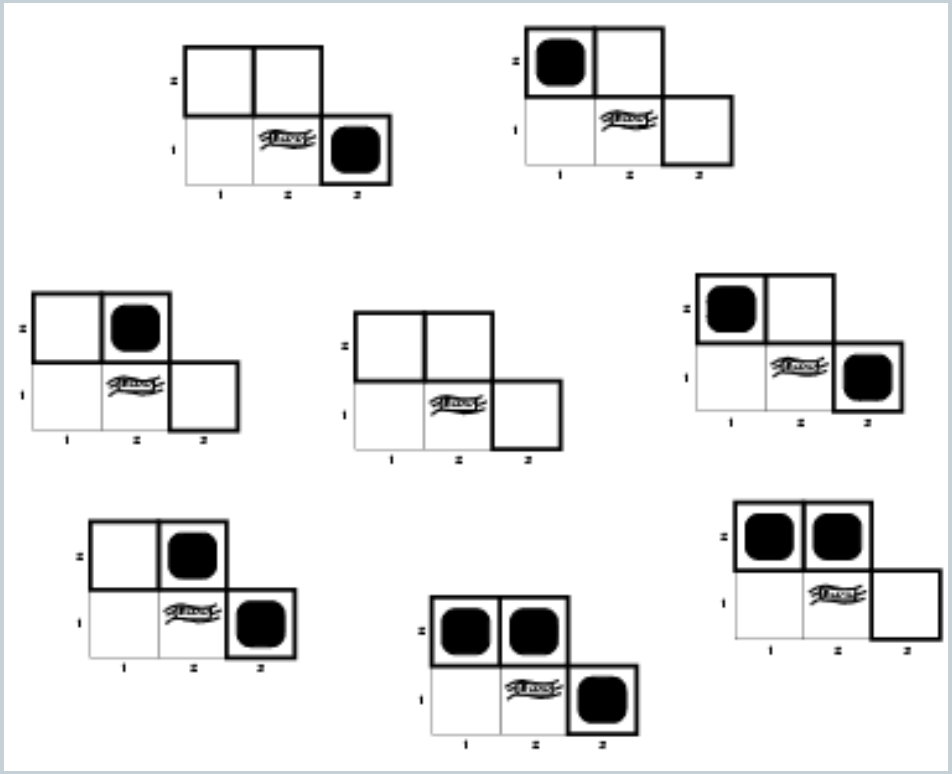
- Situation after detecting nothing in [1,1], moving right, detecting breeze in [2,1]



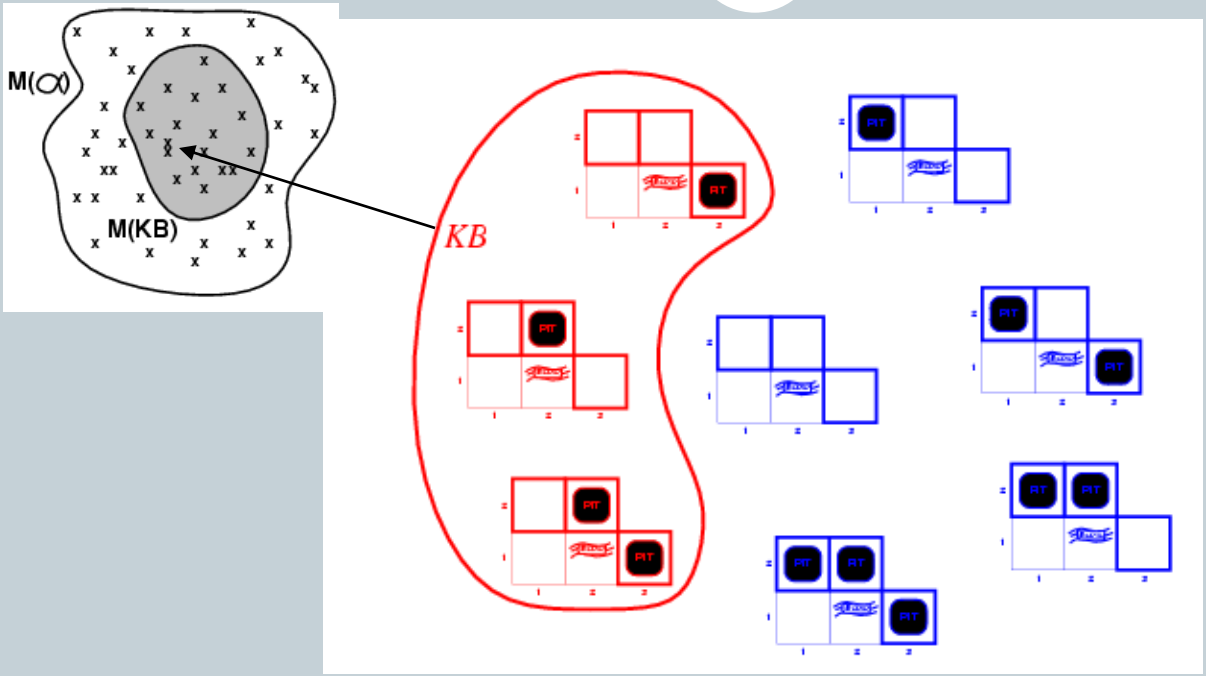
Wumpus models

All possible models in this reduced Wumpus world.





Wumpus models



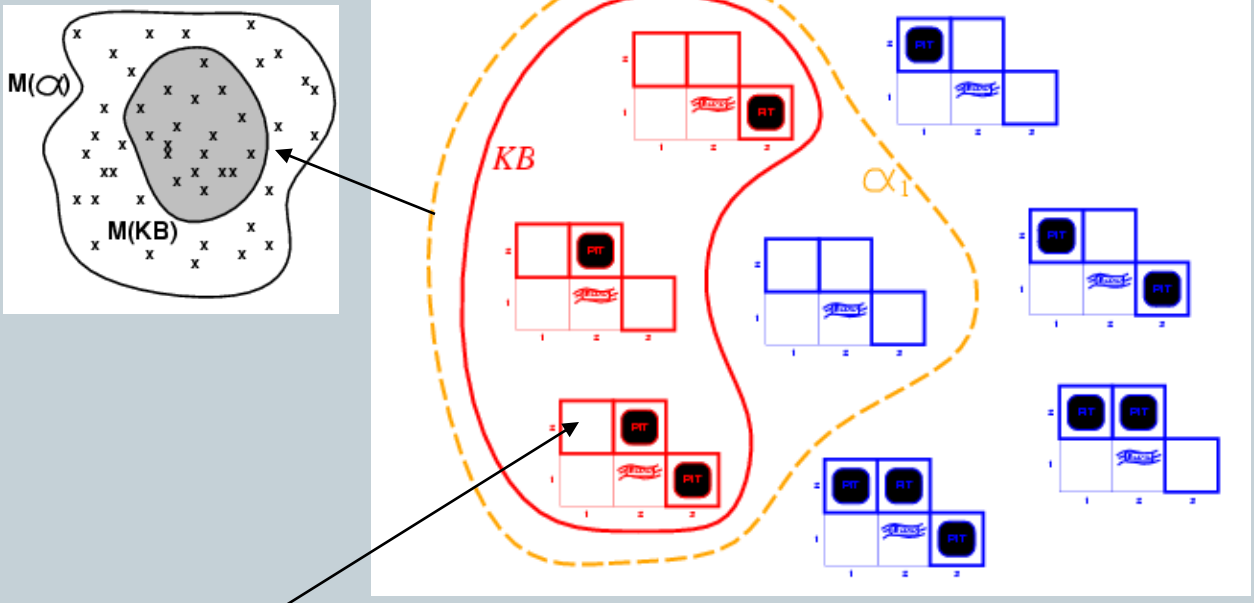
- *KB* = all possible wumpus-worlds consistent with the observations and the "physics" of the Wumpus world.

Inferring conclusions



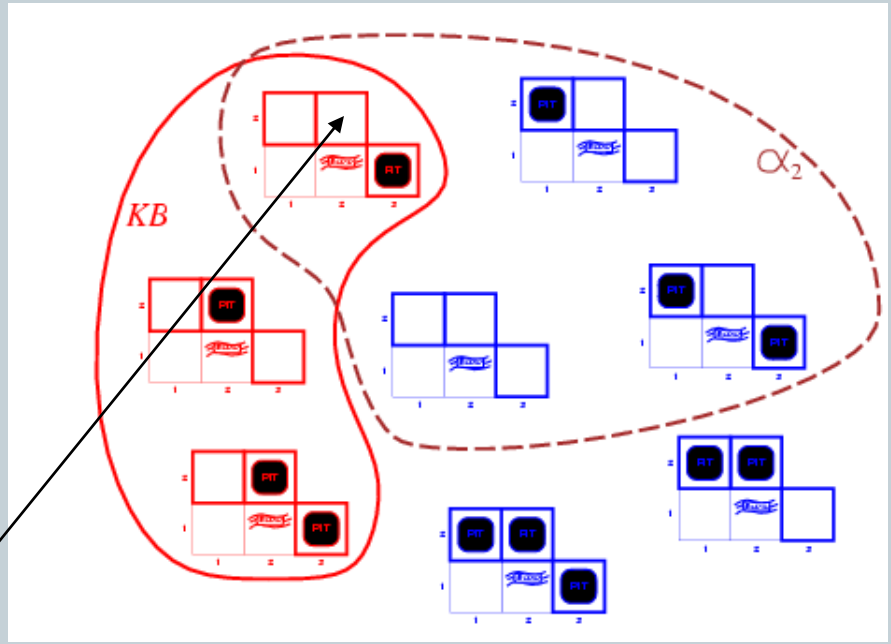
- Consider 2 possible conclusions given a KB
  - α_1 = "[1,2] is safe"
  - α_2 = "[2,2] is safe“
- One possible inference procedure
  - Start with KB
  - Model-checking
    - Check if KB $\models \alpha$ by checking if in all possible models where KB is true that α is also true
- Comments:
  - Model-checking enumerates all possible worlds
    - Only works on finite domains, will suffer from exponential growth of possible models

Wumpus models



α_1 = "[1,2] is safe", $KB \models \alpha_1$, proved by model checking

Wumpus models



α_2 = "[2,2] is safe", $KB \not\models \alpha_2$

There are some models entailed by KB where α_2 is false

Logical inference



- The notion of entailment can be used for logic inference.
  - Model checking (see wumpus example): enumerate all possible models and check whether α is true.

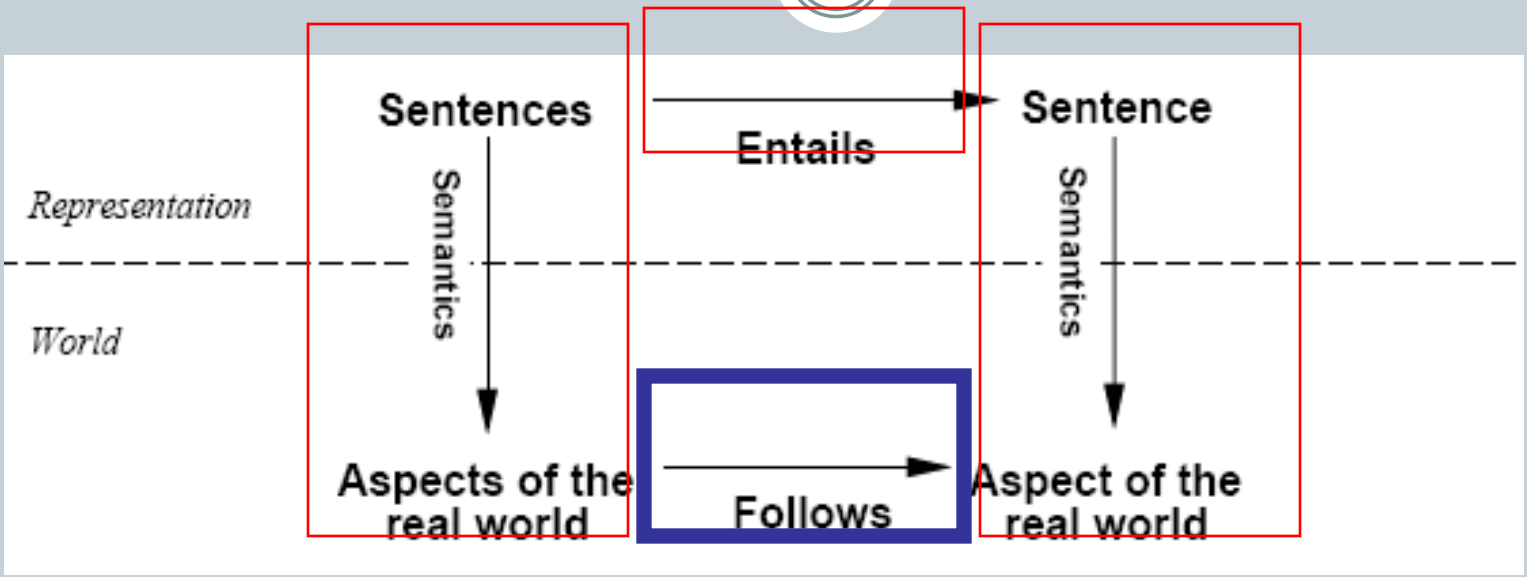
- If an algorithm only derives entailed sentences it is called *sound* or *truth preserving*.
  - Otherwise it just makes things up.

    i is sound if whenever KB $|-_i \alpha$ *it is also true that KB* $|= \alpha$
  - *E.g., model-checking is sound*

- Completeness : the algorithm can derive any sentence that is entailed.

  i is complete if whenever KB $|= \alpha$ *it is also true that KB* $|-_i \alpha$

Schematic perspective



If KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world.

Propositional logic: Syntax



- Propositional logic is the simplest logic – illustrates basic ideas
- Atomic sentences = single proposition symbols
  - E.g., P, Q, R
  - Special cases: True = always true, False = always false
- Complex sentences:
  - If S is a sentence, $\neg S$ is a sentence (negation)
  - If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction)
  - If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction)
  - If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
  - If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

Propositional logic: Semantics

Each model/world specifies true or false for each proposition symbol

E.g. $P_{1,2}$ false $P_{2,2}$ true $P_{3,1}$ false

With these symbols, 8 possible models, can be enumerated automatically.

Rules for evaluating truth with respect to a model *m*:

$\neg S$ is true iff S is false

$S_1 \wedge S_2$ is true iff S_1 is true and S_2 is true

$S_1 \vee S_2$ is true iff $S_1$is true or S_2 is true

$S_1 \Rightarrow S_2$ is true iff S_1 is false or S_2 is true
i.e., is false iff S_1 is true and S_2 is false

$S_1 \Leftrightarrow S_2$ is true iff $S_1 \Rightarrow S_2$ is true and $S_2 \Rightarrow S_1$ is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = \mathit{true} \wedge (\mathit{true} \vee \mathit{false}) = \mathit{true} \wedge \mathit{true} = \mathit{true}$$

Truth tables for connectives



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

Truth tables for connectives

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

Implication is always true when the premise is false

Why? P=>Q means "if P is true then I am claiming that Q is true otherwise no claim"

Only way for this to be false is if P is true and Q is false

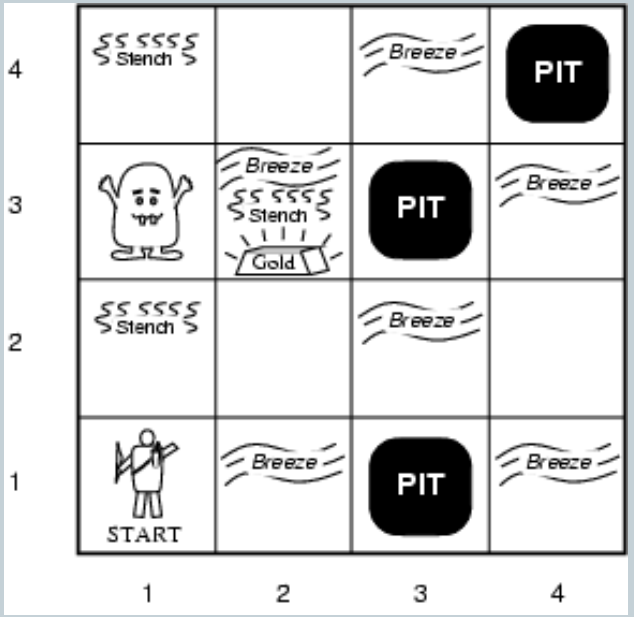
Wumpus world sentences

Let $P_{i,j}$ be true if there is a pit in [i, j].
Let $B_{i,j}$ be true if there is a breeze in [i, j].

start: $\neg P_{1,1}$

$\neg B_{1,1}$

$B_{2,1}$



- "Pits cause breezes in adjacent squares"

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

  $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

- KB can be expressed as the conjunction of all of these sentences

- Note that these sentences are rather long-winded!
  - E.g., breese "rule" must be stated explicitly for each square
  - First-order logic will allow us to define more general relations (later)

Truth tables for the Wumpus KB

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
false	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*
false	*false*	*false*	*false*	*false*	*false*	*true*	*false*	*true*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*true*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*false*	*true*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*false*

Inference by enumeration



- We want to see if α is entailed by KB
- Enumeration of all models is sound and complete.
- But…for *n* symbols, time complexity is $O(2^n)$…
- We need a more efficient way to do inference
  - But worst-case complexity will remain exponential for propositional logic

Logical equivalence



- To manipulate logical sentences we need some rewrite rules.
- Two sentences are logically equivalent iff they are true in same models: $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$

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

- Modus Ponens

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

- And-Elimination

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

- Bi-conditional Elimination

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

Validity and satisfiability

A sentence is valid if it is true in all models,
e.g., *True*, $A \vee \neg A$, $A \Rightarrow A$, $(A \wedge (A \Rightarrow B)) \Rightarrow B$
(tautologies)

Validity is connected to inference via the Deduction Theorem:
$KB \models \alpha$ if and only if $(KB \Rightarrow \alpha)$ is valid

A sentence is satisfiable if it is true in some model
e.g., $A \vee B$, C
(determining satisfiability of sentences is NP-complete)

A sentence is unsatisfiable if it is false in all models
e.g., $A \wedge \neg A$

Satisfiability is connected to inference via the following:
$KB \models \alpha$ if and only if $(KB \wedge \neg \alpha)$ is unsatisfiable
(there is no model for which KB=true and α is false)
(aka proof by contradiction: assume α to be false and this leads to contraditions in KB)

Proof methods



- Proof methods divide into (roughly) two kinds:

Application of inference rules:

Legitimate (sound) generation of new sentences from old.

- Resolution
- Forward & Backward chaining

Model checking

Searching through truth assignments.

- Improved backtracking: Davis--Putnam-Logemann-Loveland (DPLL)
- Heuristic search in model space: Walksat.

Normal Form



We want to prove: $KB \models \alpha$

equivalent to : $KB \wedge \neg\alpha$ *unsatifiable*

We first rewrite $KB \wedge \neg\alpha$ into conjunctive normal form (CNF).

A "conjunction of disjunctions"

literals

$(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$

Clause Clause

- Any KB can be converted into CNF
- k-CNF: exactly k literals per clause

Example: Conversion to CNF



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

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
   $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$

3. Move $\neg$ inwards using de Morgan's rules and double-negation:
   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$

4. Apply distributive law ($\wedge$ over $\vee$) and flatten:
   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

Resolution Inference Rule for CNF

$(A \vee B \vee C)$

$(\neg A)$

$\therefore (B \vee C)$

"If A or B or C is true, but not A, then B or C must be true."

$(A \vee B \vee C)$

$(\neg A \vee D \vee E)$

$\therefore (B \vee C \vee D \vee E)$

"If A is false then B or C must be true,
or if A is true then D or E must be true,
hence since A is either true or false, B or C
or D or E must be true."

$(A \vee B)$

$(\neg A \vee B)$

$\therefore (B \vee B) \equiv B$



Simplification

Resolution Algorithm



- The resolution algorithm tries to prove: $KB \models \alpha$ *equivalent to* $KB \wedge \neg\alpha$ *unsatisfiable*

- Generate all new sentences from KB and the query.
- One of two things can happen:

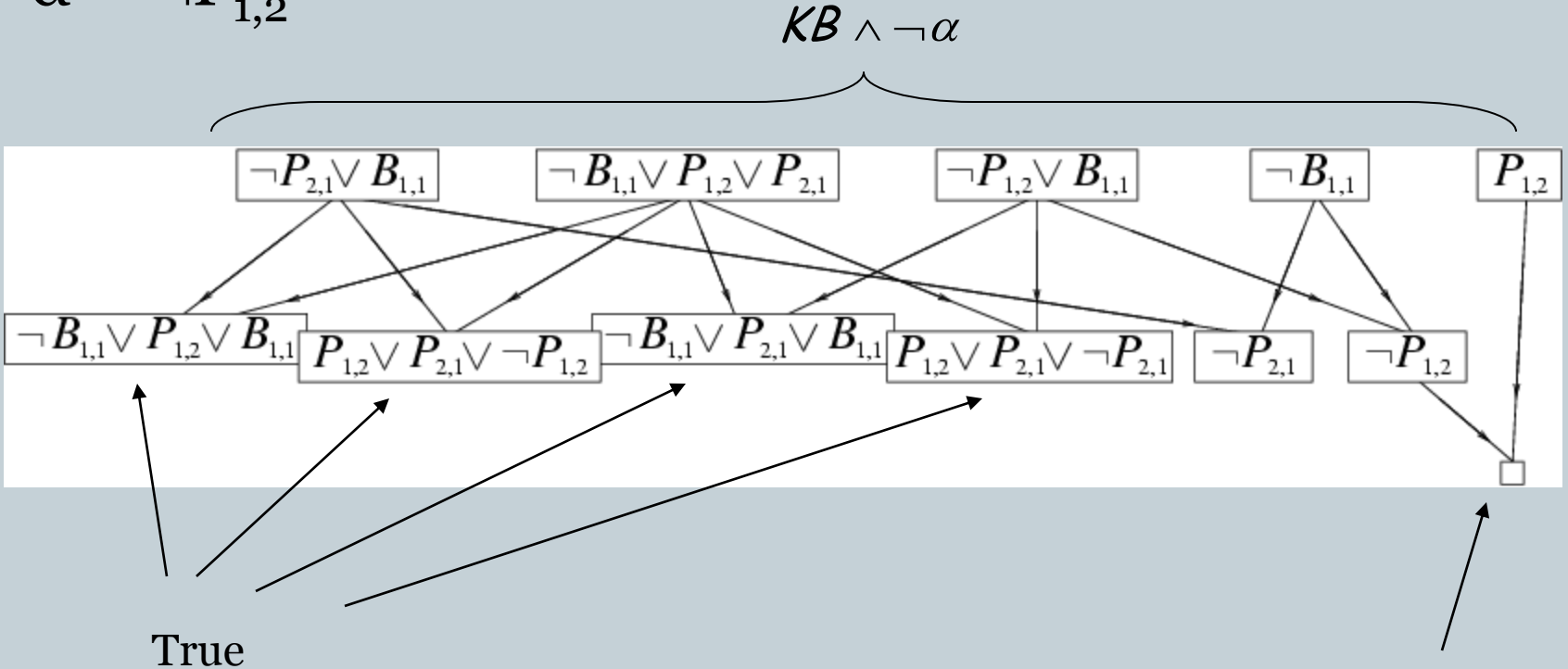
1. We find $P \wedge \neg P$ which is unsatisfiable,
   i.e. we can entail the query.

2. We find no contradiction: there is a model that satisfies the Sentence (non-trivial) and hence we cannot entail the query.

$KB \wedge \neg\alpha$

Resolution example

- $KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$
- $\alpha = \neg P_{1,2}$



Horn Clauses



- Resolution in general can be exponential in space and time.
- If we can reduce all clauses to "Horn clauses" resolution is linear in space and time

A clause with at most 1 positive literal.

e.g. $A \vee \neg B \vee \neg C$

- Every Horn clause can be rewritten as an implication with a conjunction of positive literals in the premises and a single positive literal as a conclusion.

e.g. $B \wedge C \Rightarrow A$

- 1 positive literal: definite clause
- 0 positive literals: Fact or integrity constraint:
  e.g. $(\neg A \vee \neg B) \equiv (A \wedge B \Rightarrow False)$

Forward-chaining pseudocode



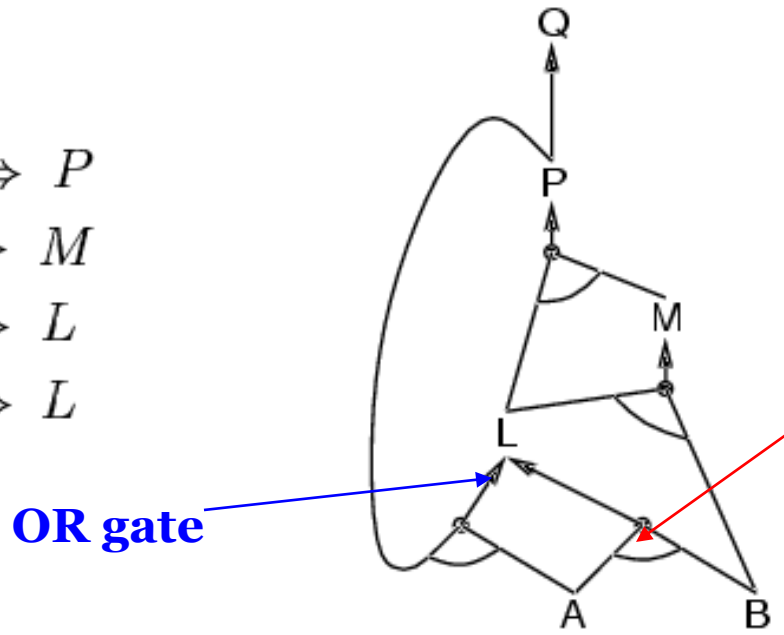
```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known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

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

Forward chaining: graph representation

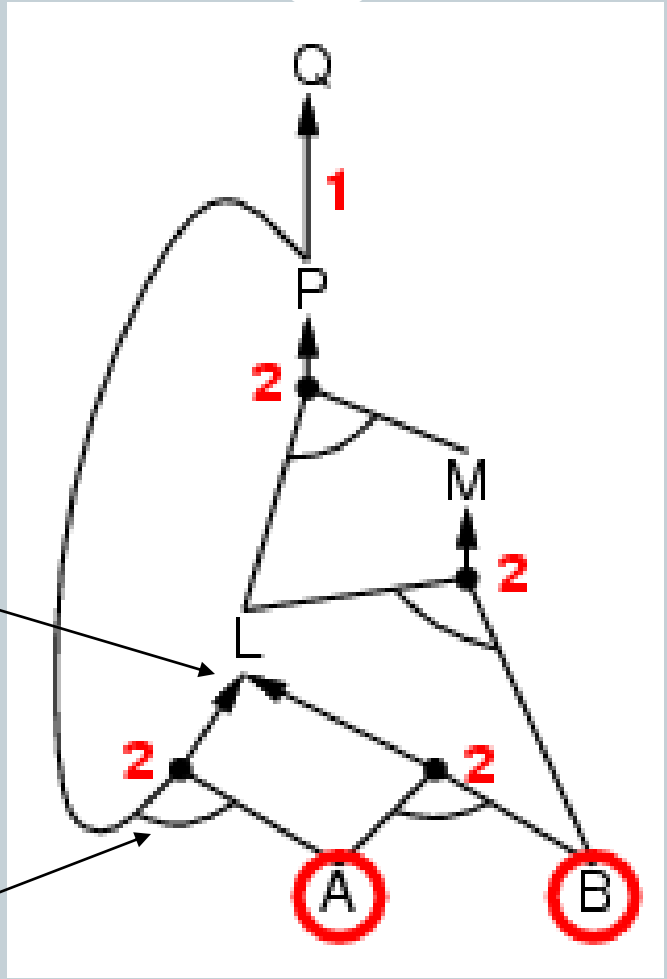
- Idea: fire any rule whose premises are satisfied in the *KB*,
  - add its conclusion to the *KB*, until query is found

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



- Forward chaining is sound and complete for Horn KB

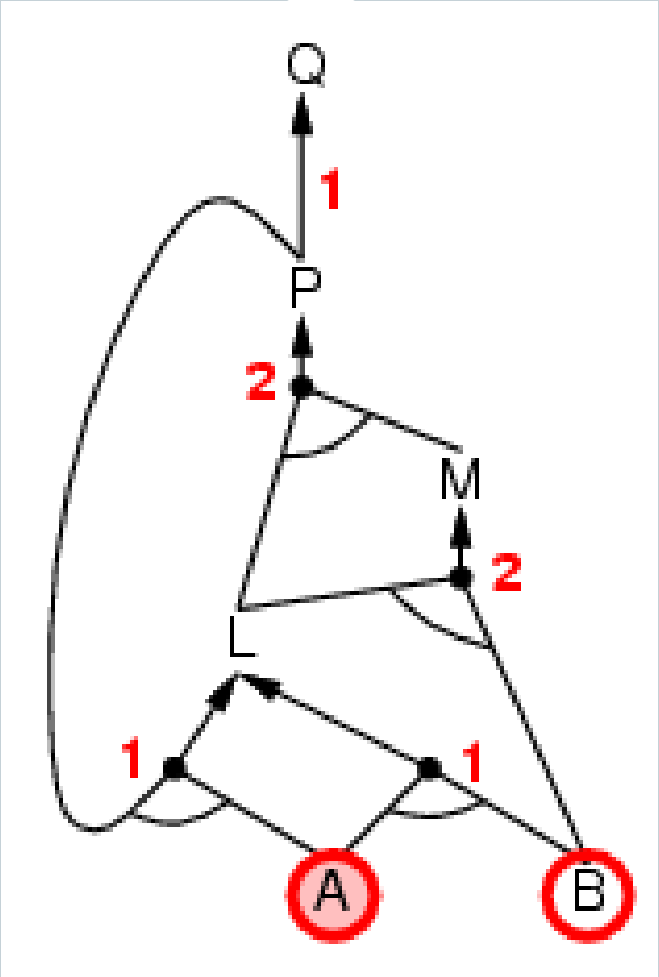
Forward chaining example



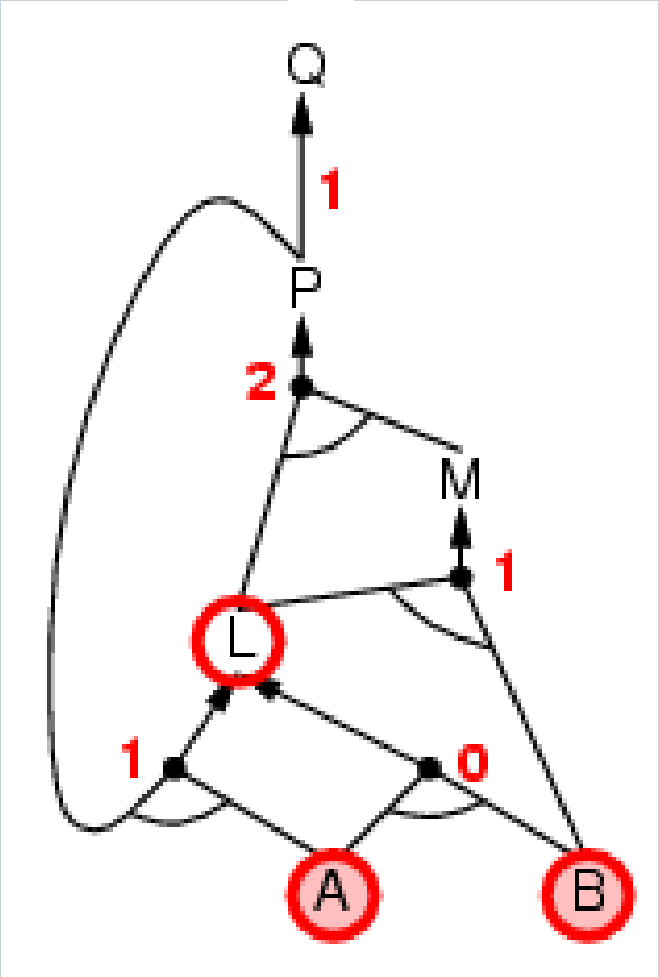
“OR” Gate

“AND” gate

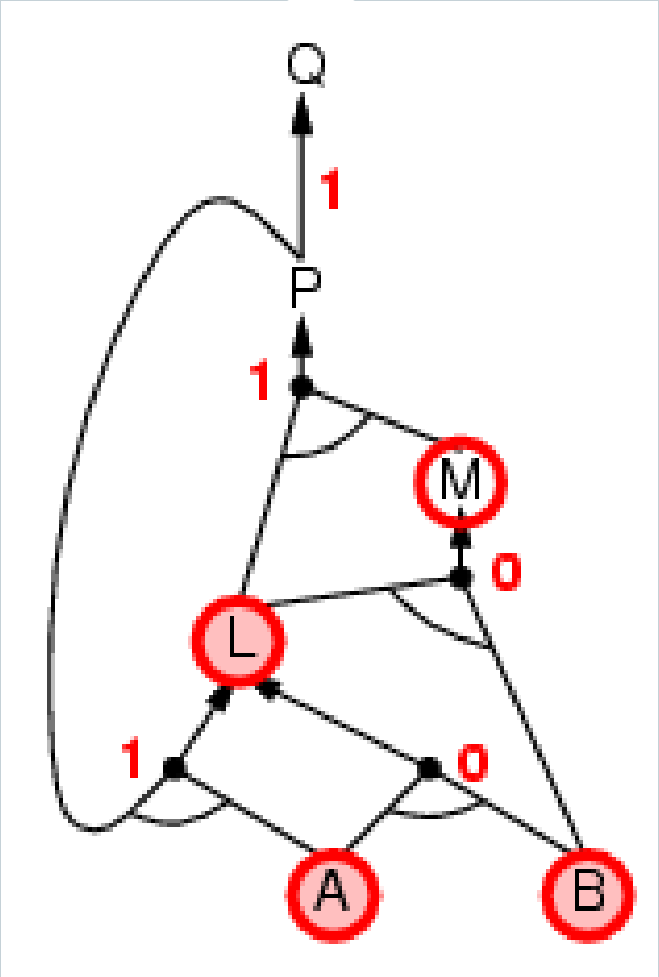
Forward chaining example



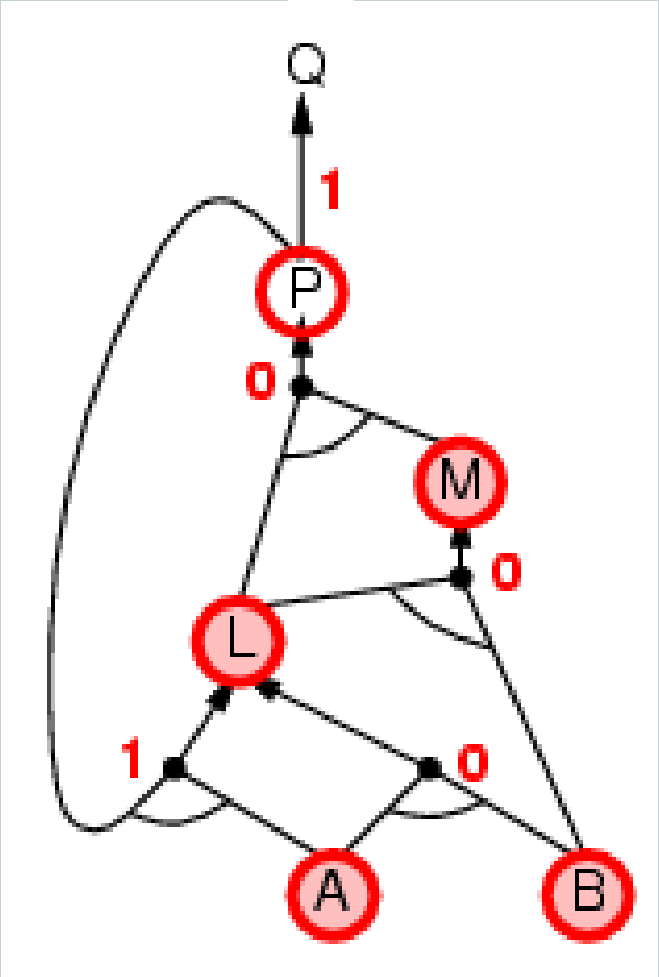
Forward chaining example



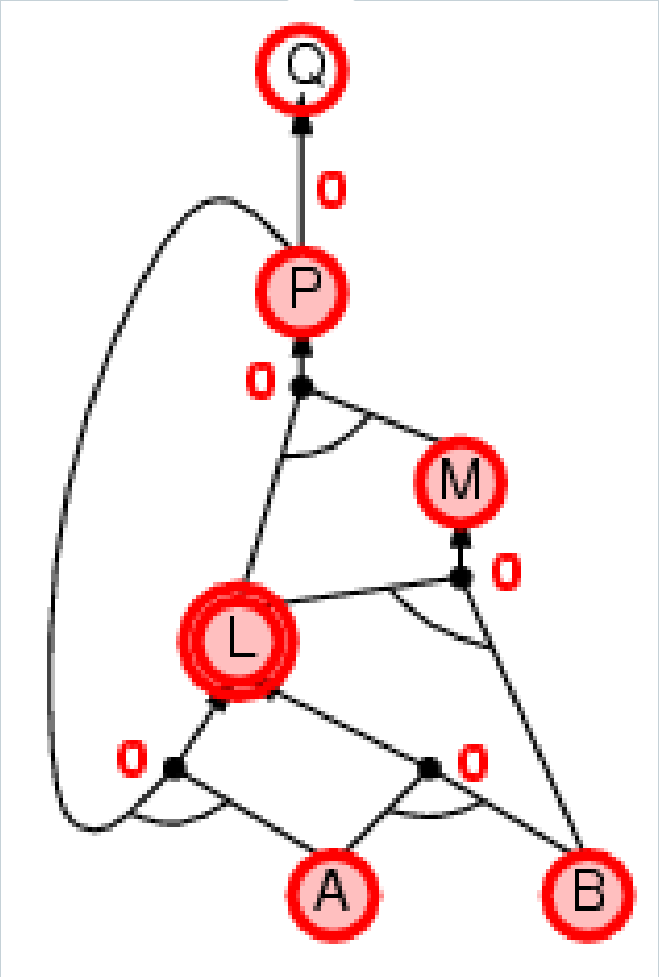
Forward chaining example



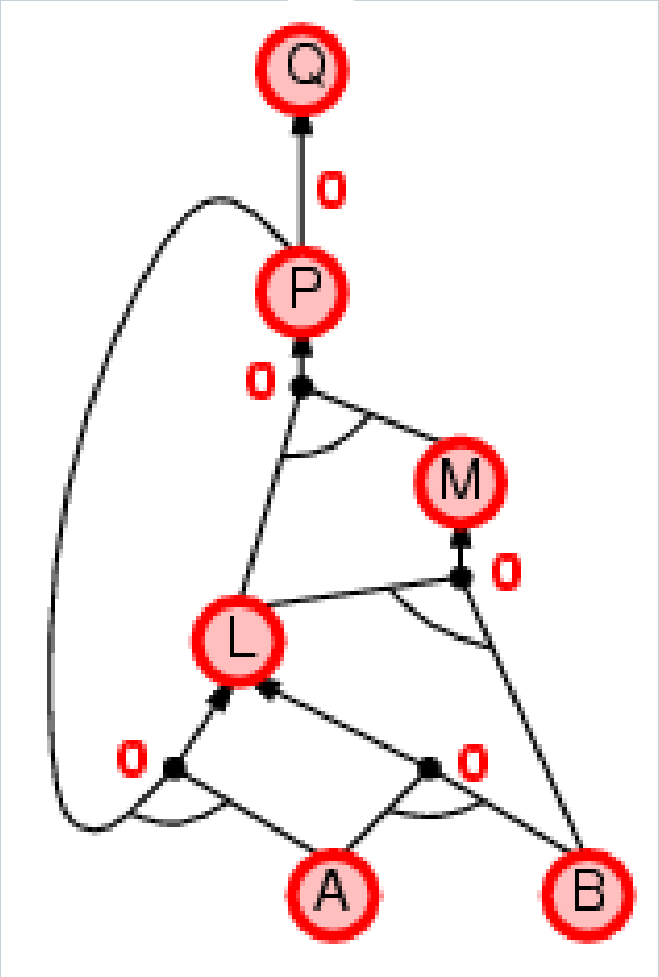
Forward chaining example



Forward chaining example



Forward chaining example



Forward chaining



- FC is data-driven, automatic, unconscious processing,
  - e.g., object recognition, routine decisions
- May do lots of work that is irrelevant to the goal

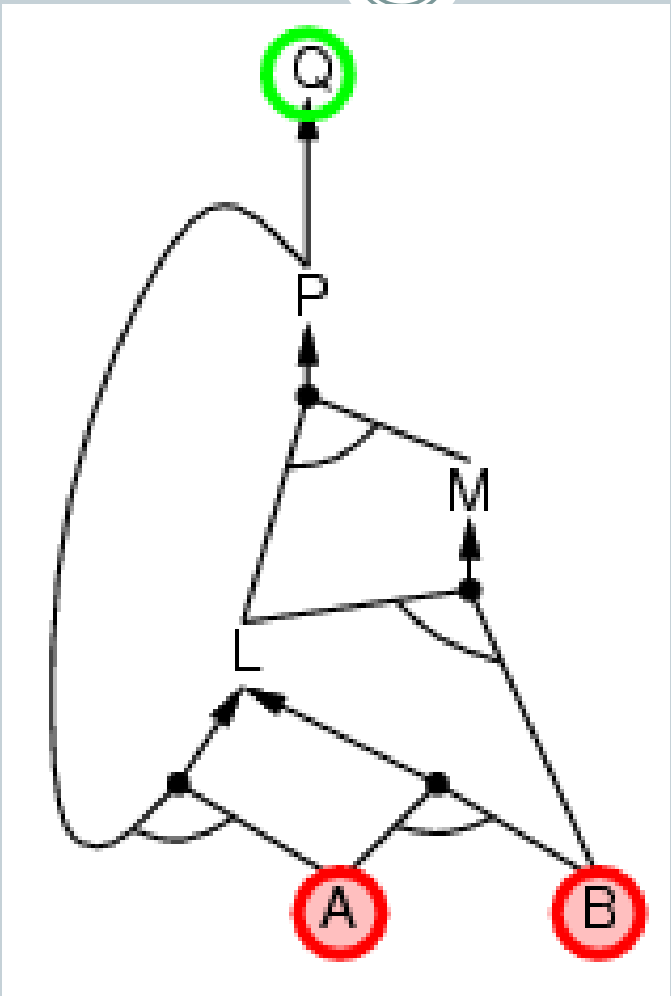
Backward chaining



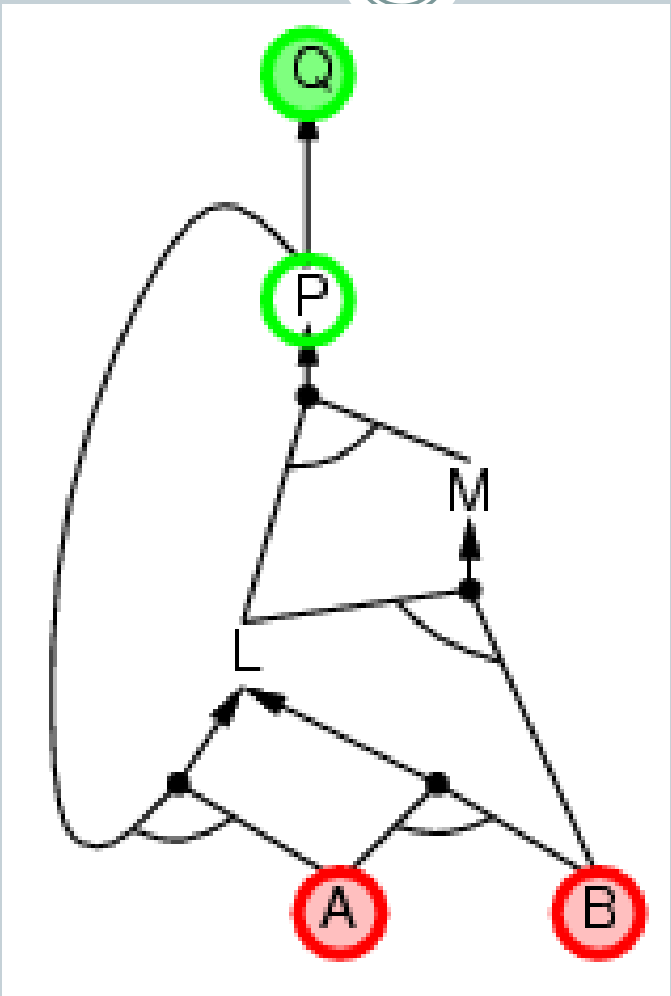
Idea: work backwards from the query *q*

- check if *q* is known already, or
- prove by BC all premises of some rule concluding *q*
- Hence BC maintains a stack of sub-goals that need to be proved to get to q.

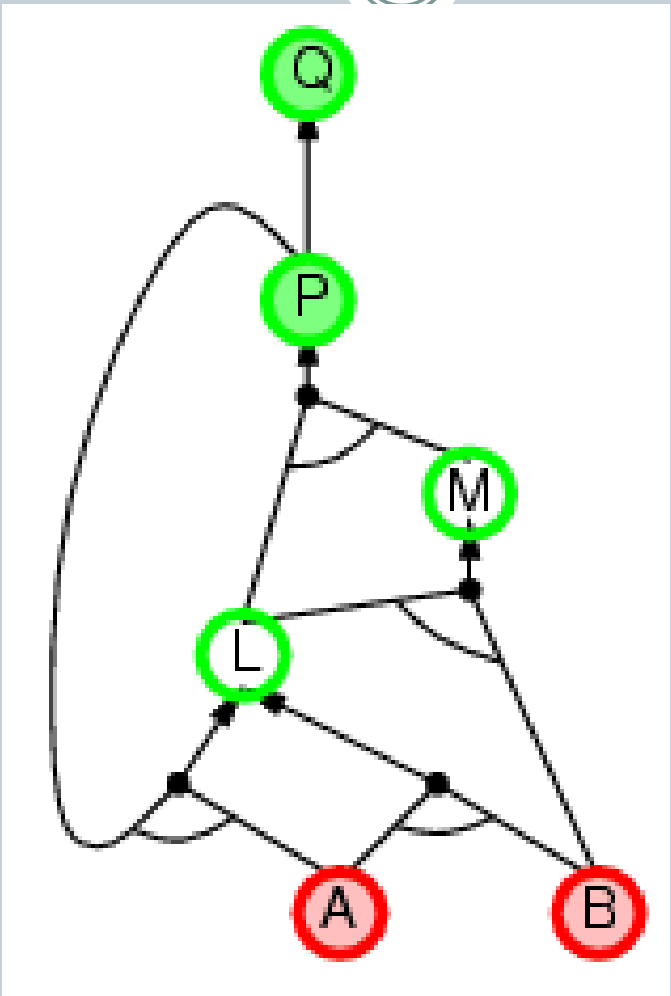
Backward chaining example



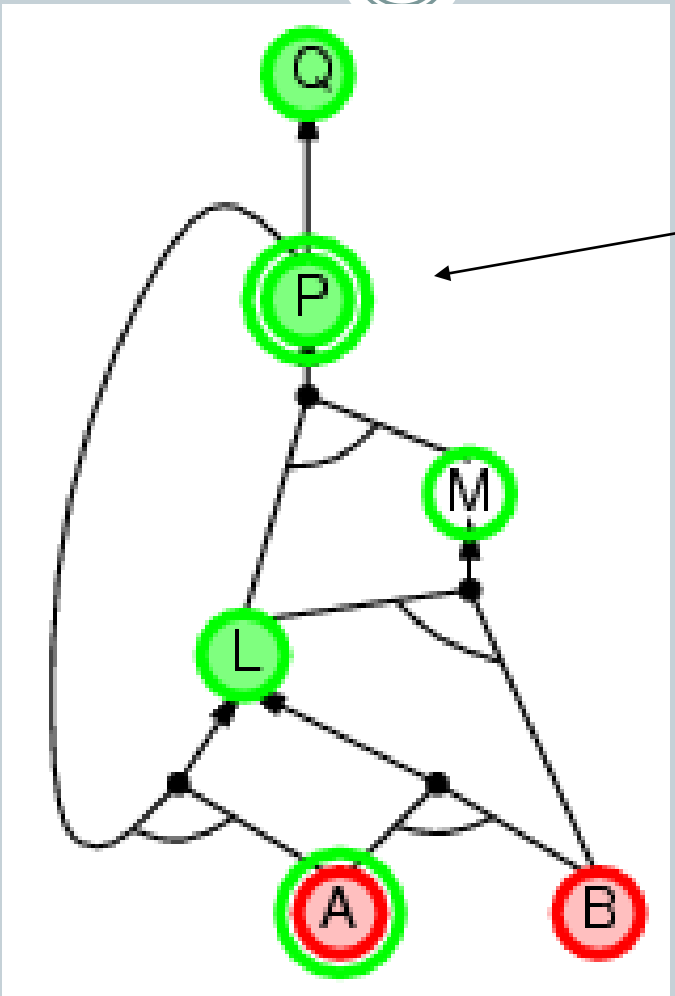
Backward chaining example



Backward chaining example

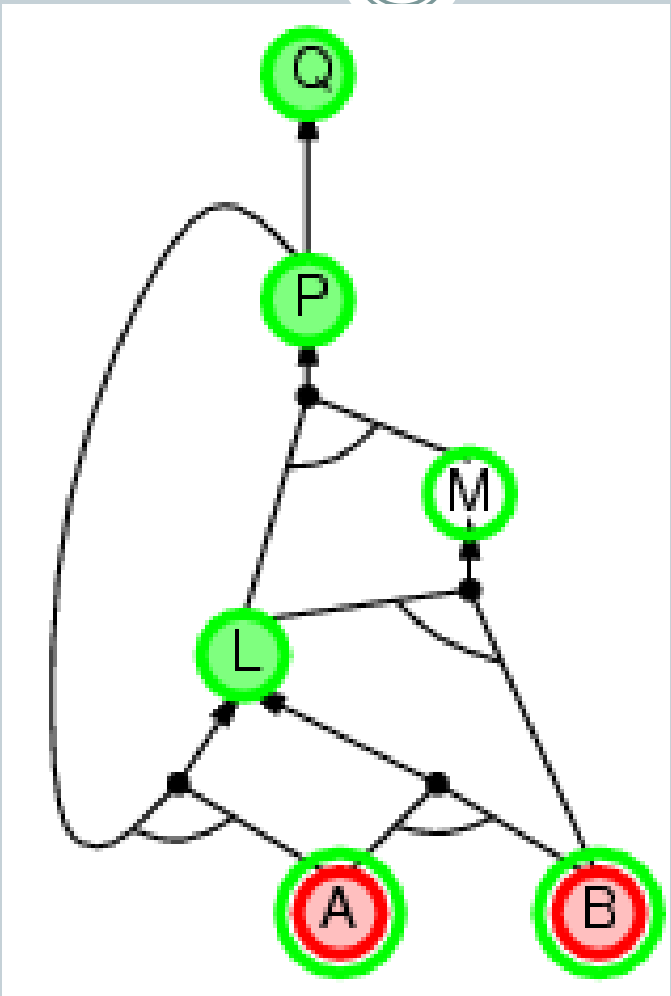


Backward chaining example

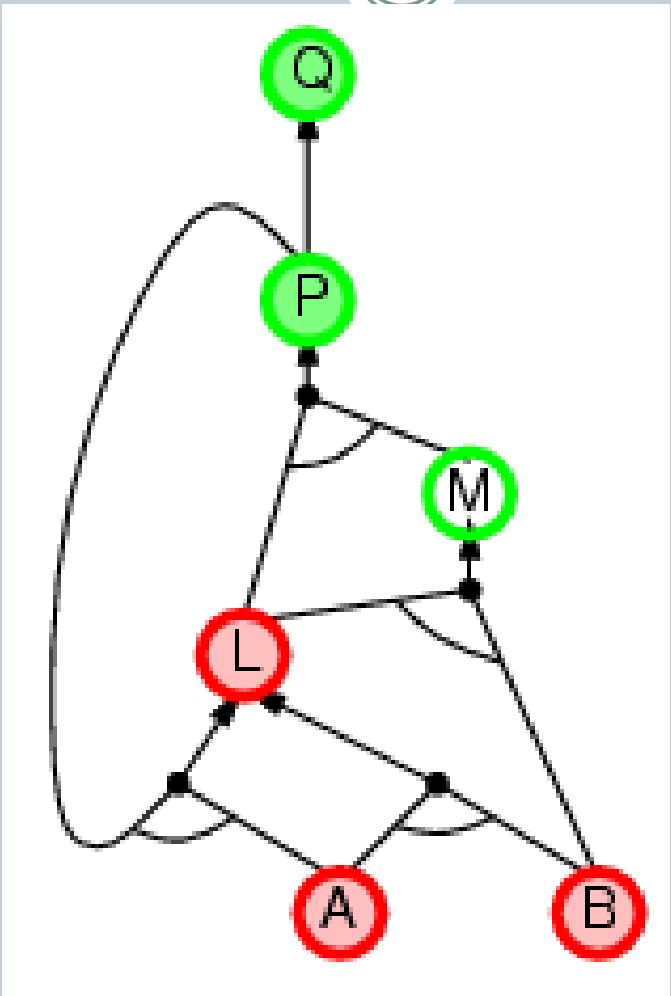


we need P to prove
L and L to prove P.

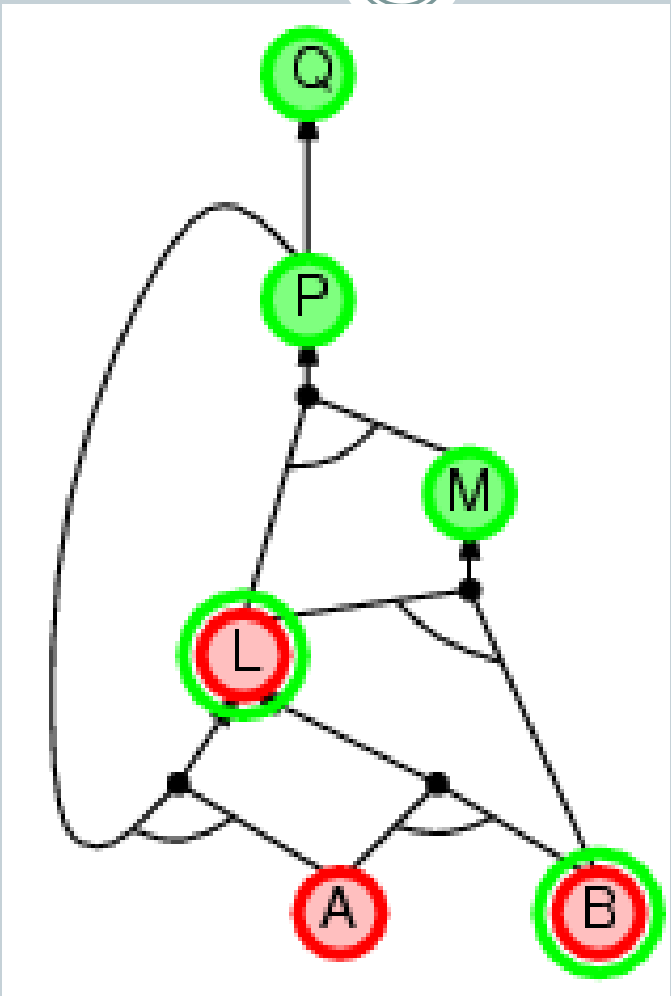
Backward chaining example



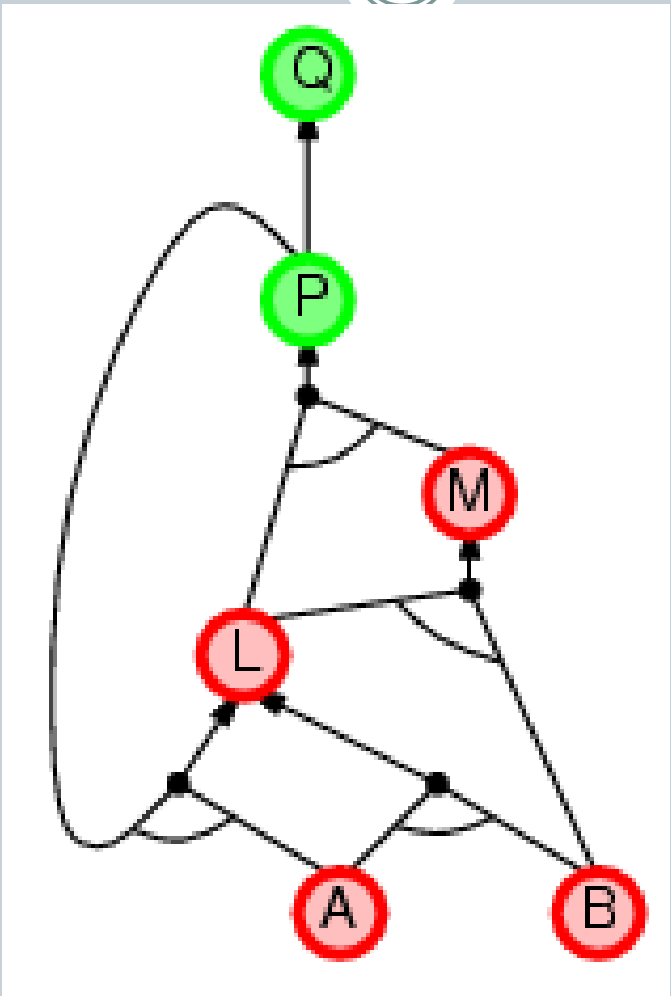
Backward chaining example



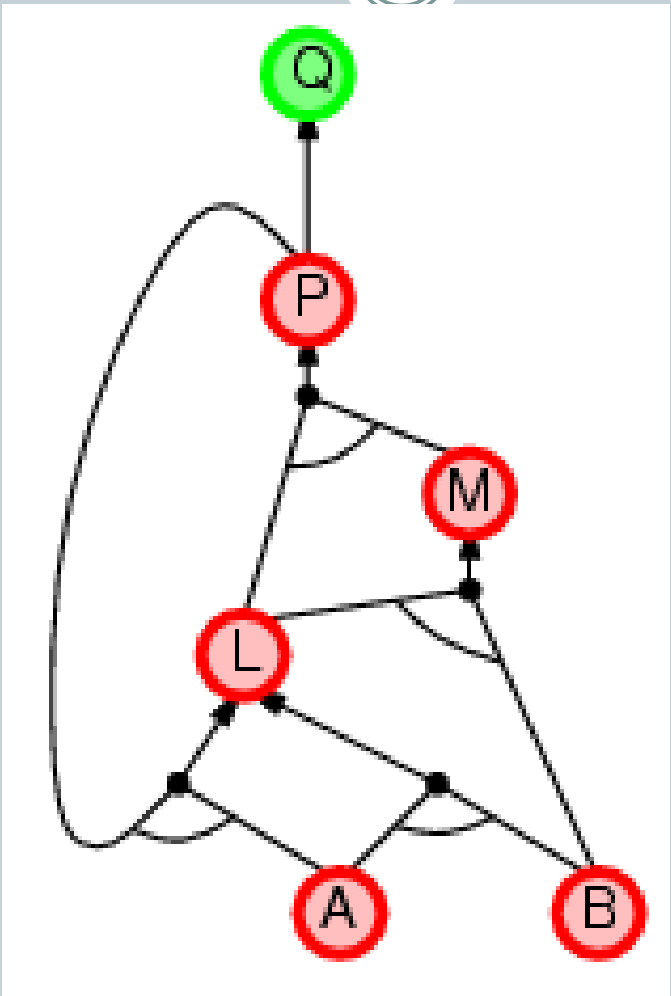
Backward chaining example



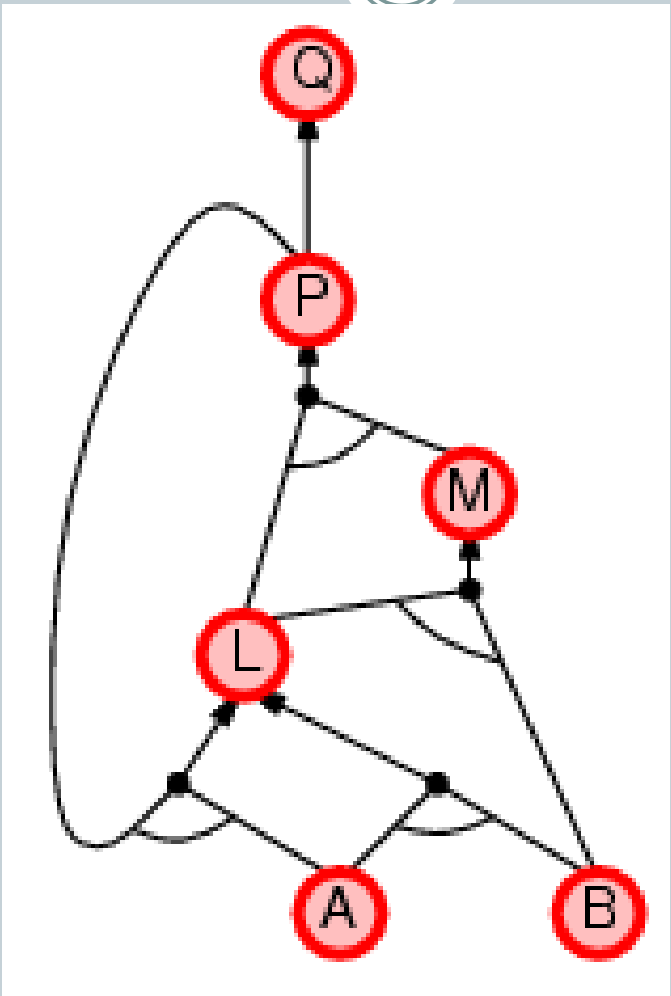
Backward chaining example



Backward chaining example



Backward chaining example



Backward chaining



- BC is goal-driven, appropriate for problem-solving,
  - e.g., Where are my keys? How do I get into a PhD program?
- Complexity of BC can be much less than linear in size of KB

Avoid loops: check if new sub-goal is already on the goal stack

Avoid repeated work: check if new sub-goal

1. has already been proved true, or
2. has already failed

Like FC, is linear and is also sound and complete (for Horn KB)

Model Checking



Two families of efficient algorithms:

- Complete backtracking search algorithms: DPLL algorithm
- Incomplete local search algorithms
  - `WalkSAT` algorithm

Satisfiability problems



- Consider *a* CNF sentence, e.g.,

  $(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$

  Satisfiability: Is there a model consistent with this sentence?

  $[A \vee B] \wedge [\neg B \vee \neg C] \wedge [A \vee C] \wedge [\neg D] \wedge [\neg D \vee \neg A]$

The `WalkSAT` algorithm



- Incomplete, local search algorithm
  - Begin with a random assignment of values to symbols
  - Each iteration: pick an unsatisfied clause
    - Flip the symbol that maximizes number of satisfied clauses, OR
    - Flip a symbol in the clause randomly
- Trades-off greediness and randomness
- Many variations of this idea
- If it returns failure (after some number of tries) we cannot tell whether the sentence is unsatisfiable or whether we have not searched long enough
  - If max-flips = infinity, and sentence is unsatisfiable, algorithm never terminates!
- Typically most useful when we expect a solution to exist

Pseudocode for WalkSAT



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

    model ← a random assignment of true/false to the symbols in clauses
    for i = 1 to max-flips do
        if model satisfies clauses then return model
        clause ← a randomly selected clause from clauses that is false in model
        with probability p flip the value in model of a randomly selected symbol
                from clause
        else flip whichever symbol in clause maximizes the number of satisfied clauses
    return failure
```

Hard satisfiability problems



- Consider *random* 3-CNF sentences. e.g.,

  $(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$

  m = number of clauses (5)

  n = number of symbols (5)

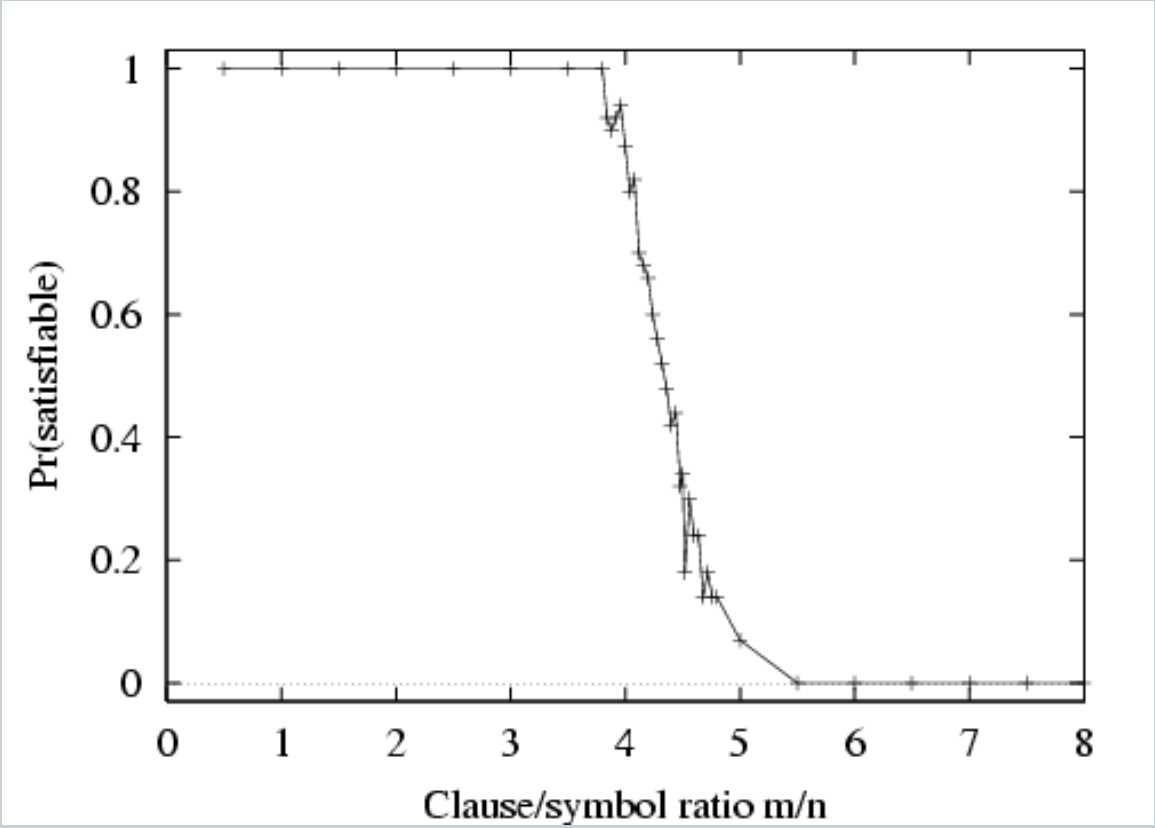
  - Underconstrained problems:
    - Relatively few clauses constraining the variables
    - Tend to be easy
    - 16 of 32 possible assignments above are solutions
      - (so 2 random guesses will work on average)

Hard satisfiability problems

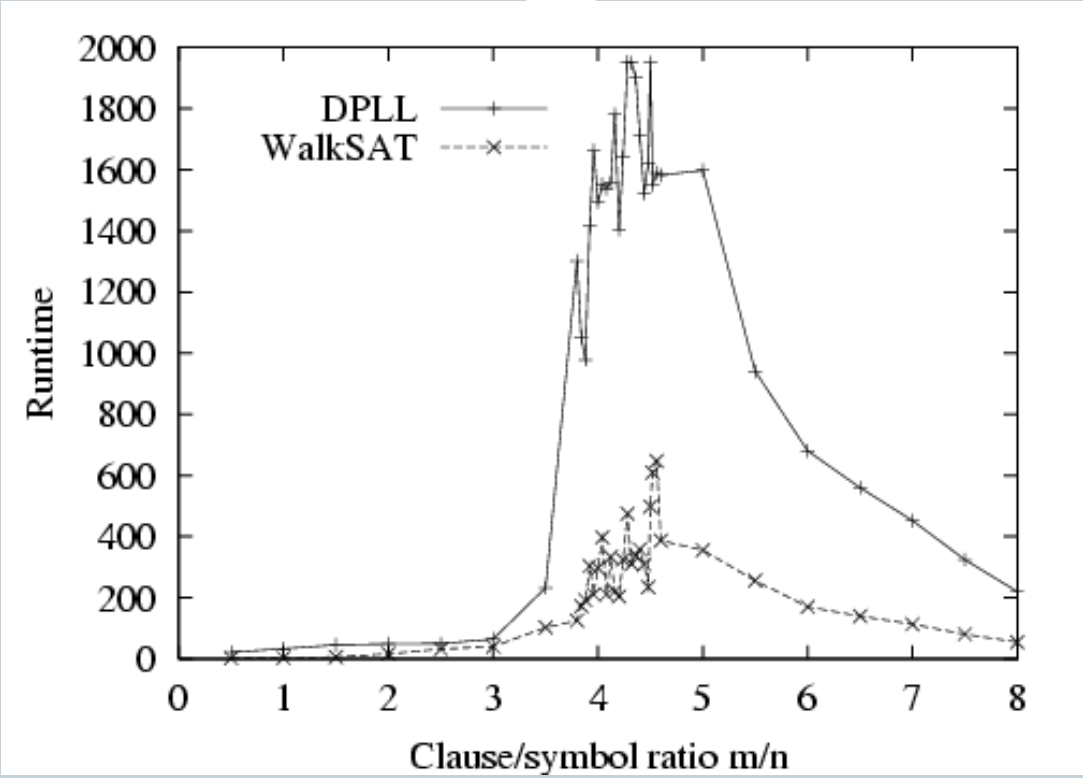


- What makes a problem hard?
  - Increase the number of clauses while keeping the number of symbols fixed
  - Problem is more constrained, fewer solutions
  - Investigate experimentally….

P(satisfiable) for random 3-CNF sentences, n = 50



Run-time for DPLL and WalkSAT



- Median runtime for 100 satisfiable random 3-CNF sentences, $n = 50$

Inference-based agents in the wumpus world

A wumpus-world agent using propositional logic:

$\neg P_{1,1}$ (no pit in square [1,1])

$\neg W_{1,1}$ (no Wumpus in square [1,1])

$B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y})$ (Breeze next to Pit)

$S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y})$ (stench next to Wumpus)

$W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,4}$ (at least 1 Wumpus)

$\neg W_{1,1} \vee \neg W_{1,2}$ (at most 1 Wumpus)

$\neg W_{1,1} \vee \neg W_{8,9}$

…

$\Rightarrow$ 64 distinct proposition symbols, 155 sentences

Limited expressiveness of propositional logic



- KB contains "physics" sentences for every single square
- For every time *t* and every location [*x*,*y*],
  $L_{x,y} \wedge FacingRight^t \wedge Forward^t \Rightarrow L_{x+1,y}$
- Rapid proliferation of clauses.

  First order logic is designed to deal with this through the introduction of variables.

Summary



- Logical agents apply inference to a knowledge base to derive new information and make decisions
- Basic concepts of logic:
  - syntax: formal structure of sentences
  - semantics: truth of sentences wrt models
  - entailment: necessary truth of one sentence given another
  - inference: deriving sentences from other sentences
  - soundness: derivations produce only entailed sentences
  - completeness: derivations can produce all entailed sentences
- Resolution is complete for propositional logic
- Forward, backward chaining are linear-time, complete for Horn clauses
- Propositional logic lacks expressive power

UNIT III

Logic and Knowledge Representation First-Order Logic:

Representation, Syntax and Semantics of First-Order Logic, Using First- Order Logic,

Knowledge Engineering in First-Order Logic:

Inference in First-Order Logic, Propositional vs. First-Order Inference, Unification and Lifting, Forward Chaining, Backward Chaining, Resolution

First-Order Logic

n the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic.

- "Some humans are intelligent", or
- "Sachin likes cricket."

To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

First-Order logic:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as Predicate logic or First-order predicate logic. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
    a. Objects: A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

b. Relations: It can be unary relation such as: red, round, is adjacent, or n-any relation such as: the sister of, brother of, has color, comes between

c. Function: Father of, best friend, third inning of, end of, ...

- As a natural language, first-order logic also has two main parts:
  a. Syntax
  b. Semantics

Syntax of First-Order logic:

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

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

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

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as Predicate (term1, term2, ..., term n).

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).

Chinky is a cat: => cat (Chinky).

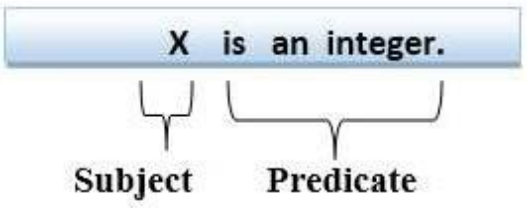
Complex Sentences:

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

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

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

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

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

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

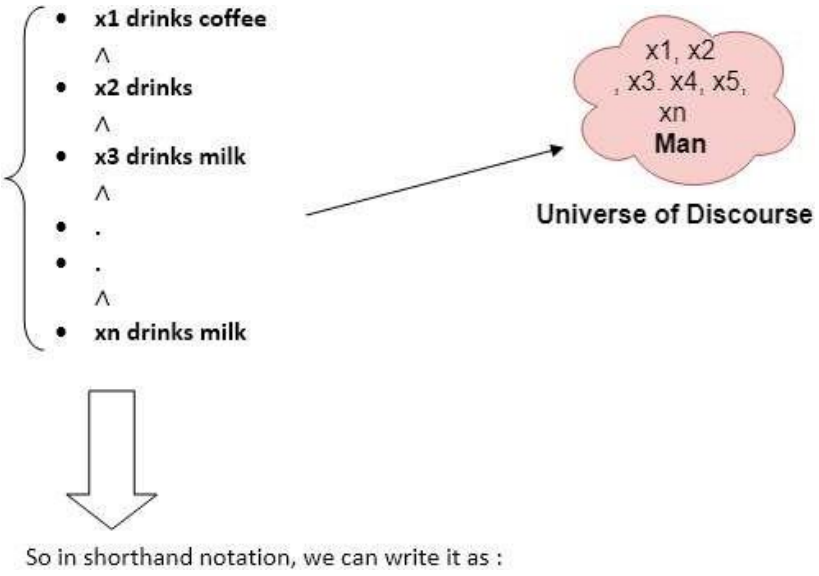
If x is a variable, then ∀x is read as:

- For all x
- For each x
- For every x.

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



$\forall x$ man(x) → drink (x, coffee).

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

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator $\exists$, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

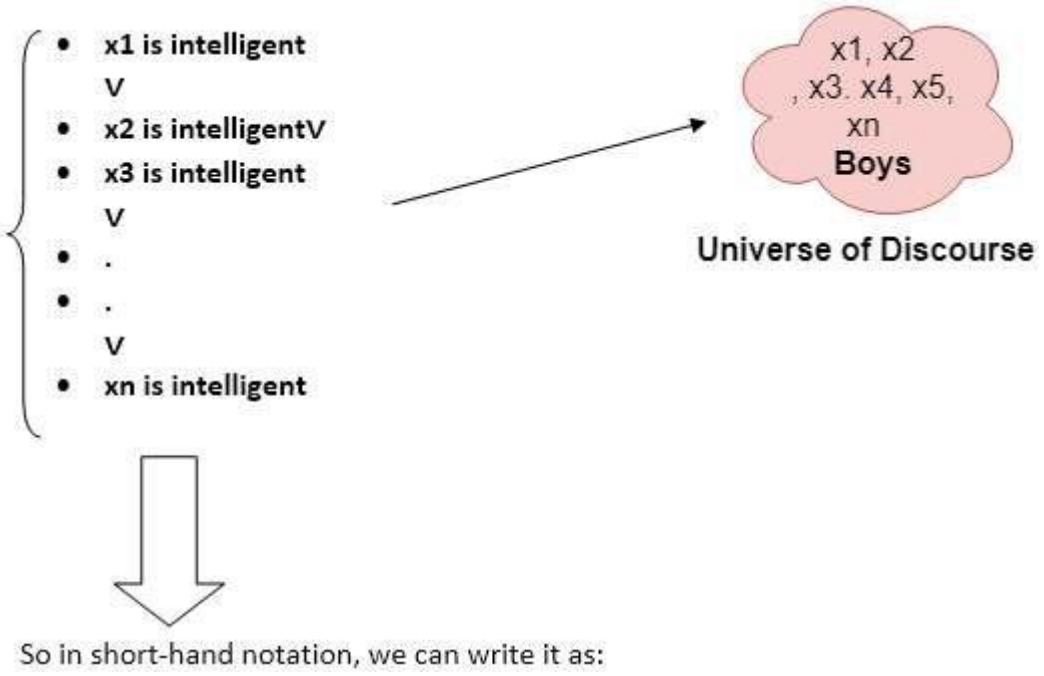
If x is a variable, then existential quantifier will be $\exists x$ or $\exists (x)$. And it will be read as:

- There exists a 'x.'
- For some 'x.'
- For at least one 'x.'

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

Example:

Some boys are intelligent.



$\exists$ x: boys(x) $\wedge$ intelligent(x)

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

Points to remember:

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

Properties of Quantifiers:

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

Some Examples of FOL using quantifier:

1. All birds fly.

In this question the predicate is "fly(bird)."

And since there are all birds who fly so it will be represented as follows.

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

2. Every man respects his parent.

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

Since there is every man so will use $\forall$, and it will be represented as follows:

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

3. Some boys play cricket.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use $\exists$, and it will be represented as:

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

4. Not all students like both Mathematics and Science.

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

Since there are not all students, so we will use $\forall$ with negation, so following representation for this:

$\neg\forall$ (x) [student(x) $\rightarrow$ like(x, Mathematics) $\wedge$ like(x, Science)].

5. Only one student failed in Mathematics.

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

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$\exists$ (x) [student(x) $\rightarrow$ failed (x, Mathematics) $\wedge\forall$ (y) [$\neg$(x==y) $\wedge$ student(y) $\rightarrow$ $\neg$failed (x, Mathematics)].

Free and Bound Variables:

The quantifiers interact with variables which appear in a suitable way. There are two types of variables in First-order logic which are given below:

Free Variable: A variable is said to be a free variable in a formula if it occurs outside the scope of the quantifier.

Example: ∀x ∃(y)[P (x, y, z)], where z is a free variable.

Bound Variable: A variable is said to be a bound variable in a formula if it occurs within the scope of the quantifier.

Example: ∀x [A (x) B(y)], here x and y are the bound variables.

Knowledge Engineering in First-order logic

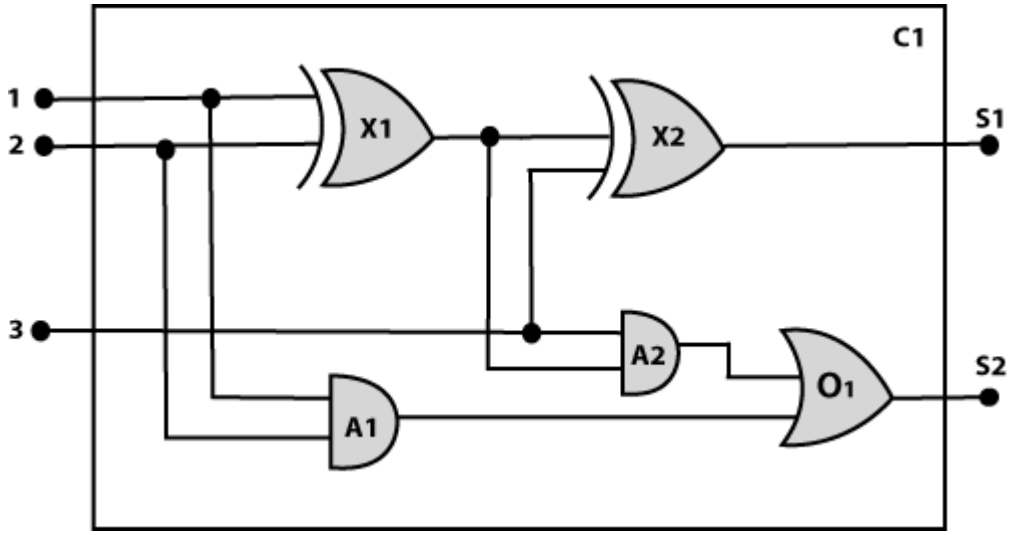
What is knowledge-engineering?

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

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

The knowledge-engineering process:

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



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks.

At the first level or highest level, we will examine the functionality of the circuit:

- Does the circuit add properly?
- What will be the output of gate A2, if all the inputs are high?

At the second level, we will examine the circuit structure details such as:

- Which gate is connected to the first input terminal?
- Does the circuit have feedback loops?

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: AND, OR, XOR, and NOT.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, Gate(X1). The functionality of each gate is determined by its type, which is taken as constants such as AND, OR, XOR, or NOT. Circuits will be identified by a predicate: Circuit (C1).

For the terminal, we will use predicate: Terminal(x).

For gate input, we will use the function In(1, X1) for denoting the first input terminal of the gate, and for output terminal we will use Out (1, X1).

The function Arity(c, i, j) is used to denote that circuit c has i input, j output.

The connectivity between gates can be represented by predicate Connect(Out(1, X1), In(1, X1)).

We use a unary predicate On (t), which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

- If two terminals are connected then they have the same input signal, it can be represented as:

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

1. ∀ t1, t2 Terminal (t1) ∧ Terminal (t2) ∧ Connect (t1, t2) → Signal (t1) = Signal (2).

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

1. ∀ t Terminal (t) →Signal (t) = 1 ∨Signal (t) = 0.

- Connect predicates are commutative:

1. ∀ t1, t2 Connect(t1, t2) → Connect (t2, t1).

- Representation of types of gates:

1. ∀ g Gate(g) ∧ r = Type(g) → r = OR ∨r = AND ∨r = XOR ∨r = NOT.

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

1. ∀ g Gate(g) ∧ Type(g) = AND →Signal (Out(1, g))= 0 ⇔ ∃n Signal (In(n, g))= 0.

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

1. ∀ g Gate(g) ∧ Type(g) = OR → Signal (Out(1, g))= 1 ⇔ ∃n Signal (In(n, g))= 1

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

1. ∀ g Gate(g) ∧ Type(g) = XOR → Signal (Out(1, g)) = 1 ⇔ Signal (In(1, g)) ≠ Signal (In(2, g)).

- Output of NOT gate is invert of its input:

1. ∀ g Gate(g) ∧ Type(g) = NOT → Signal (In(1, g)) ≠ Signal (Out(1, g)).

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

1. ∀ g Gate(g) ∧ Type(g) = NOT → Arity(g, 1, 1)

2. ∀ g Gate(g) ∧ r =Type(g) ∧ (r= AND ∨r= OR ∨r= XOR) → Arity (g, 2, 1).

- All gates are logic circuits:

1. ∀ g Gate(g) → Circuit (g).

5. Encode a description of the problem instance:

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below:

Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

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

And then represent the connections between all the gates.

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

In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

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

7. Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.

In the knowledge base, we may have omitted assertions like $1 \neq 0$

Inference in First-Order Logic

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

Substitution:

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

Equality:

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

Example: Brother (John) = Smith.

As in the above example, the object referred by the Brother (John) is similar to the object referred by Smith. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: ¬(x=y) which is equivalent to x ≠y.

FOL inference rules for quantifier:

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

- Universal Generalization
- Universal Instantiation
- Existential Instantiation
- Existential introduction

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "A byte contains 8 bits", so for ∀ x P(x) "All bytes contain 8 bits.", it will also be true.

2. Universal Instantiation:

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

Example:1.

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

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

Example: 2.

Let's take a famous example,

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

∀x king(x) ∧ greedy (x) → Evil (x),

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

- King(John) ∧ Greedy (John) → Evil (John),
- King(Richard) ∧ Greedy (Richard) → Evil (Richard),
- King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John)),

3. Existential Instantiation:

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

- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

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

So we can infer: Crown(K) ∧ OnHead(K, John), as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is called Skolem constant.
- The Existential instantiation is a special case of Skolemization process.

4. Existential introduction

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

  "Priyanka got good marks in English."

  "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences pi, pi', q. Where there is a substitution θ such that SUBST (θ, pi',) = SUBST(θ, pi), it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

1. Here let say, p1' is king(John) p1 is king(x)
2. p2' is Greedy(y) p2 is Greedy(x)
3. θ is {x/John, y/John} q is evil(x)
4. SUBST(θ,q).

What is Unification?

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\Psi_1\sigma = \Psi_2\sigma$, then it can be expressed as UNIFY(Ψ_1, Ψ_2).
- Example: Find the MGU for Unify{King(x), King(John)}

Let Ψ_1 = King(x), Ψ_2 = King(John),

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

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

- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, P(x, y), and P(a, f(z)).

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

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

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

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

Conditions for Unification:

Following are some basic conditions for unification:

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

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if $\Psi_1$is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return { (Ψ_2/ Ψ_1)}.

c) Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE,

b. Else return {(Ψ_1/ Ψ_2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

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

For each pair of the following atomic sentences find the most general unifier (If exist).

1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S_0 => Here, Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(X, X)

SUBST θ= {f(a) / X}

S1 => Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(f(a), f(a))

SUBST θ= {f(a) / g(y)}, Unification failed.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ_1 = p(b, X, f(g(Z))) , and Ψ_2 = p(Z, f(Y), f(Y))

S_0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}

SUBST θ={b/Z}

S_1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}

SUBST θ={f(Y) /X}

S_2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}

SUBST θ= {g(b) /Y}

S_2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} Unified Successfully.

And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ_1 = {p (X, X), and Ψ_2 = p (Z, f(Z))

S_0 => {p (X, X), p (Z, f(Z))}

SUBST θ= {X/Z}

S1 => {p (Z, Z), p (Z, f(Z))}

SUBST θ= {f(Z) / Z}, Unification Failed.

Hence, unification is not possible for these expressions.

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}

S_0 => {prime(11) , prime(y)}

SUBST θ= {11/y}

S_1 => {prime(11) , prime(11)} , Successfully unified.

Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ_1 = Q(a, g(x, a), f(y)), and Ψ_2 = Q(a, g(f(b), a), x)

S_0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}

SUBST θ= {f(b)/x}

S_1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}

S_1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, Successfully Unified.

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)

S_0 => { knows(Richard, x); knows(Richard, John)}

SUBST θ= {John/x}

S_1 => { knows(Richard, John); knows(Richard, John)}, Successfully Unified.

Unifier: {John/x}.

Resolution in FOL

Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the conjunctive normal form or clausal form.

Clause: Disjunction of literals (an atomic sentence) is called a clause. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be conjunctive normal form or CNF.

The resolution inference rule:

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

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

Where l_i and m_j are complementary literals.

This rule is also called the binary resolution rule because it only resolves exactly two literals.

Example:

We can resolve two clauses which are given below:

[Animal (g(x) V Loves (f(x), x)] and [¬ Loves(a, b) V ¬Kills(a, b)]

Where two complimentary literals are: Loves (f(x), x) and ¬ Loves (a, b)

These literals can be unified with unifier θ= [a/f(x), and b/x] , and it will generate a resolvent clause:

[Animal (g(x) V ¬ Kills(f(x), x)].

Steps for Resolution:

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

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

a. John likes all kind of food.
b. Apple and vegetable are food
c. Anything anyone eats and not killed is food.
d. Anil eats peanuts and still alive

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

e. Harry eats everything that Anil eats.

Prove by resolution that:

f. John likes peanuts.

Step-1: Conversion of Facts into FOL

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

a. ∀x: food(x) → likes(John, x)
b. food(Apple) ∧ food(vegetables)
c. ∀x ∀y: eats(x, y) ∧ ¬ killed(x) → food(y)
d. eats (Anil, Peanuts) ∧ alive(Anil).
e. ∀x : eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬ killed(x) → alive(x) } **added predicates.**
g. ∀x: alive(x) →¬ killed(x) }
h. likes(John, Peanuts)

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- Eliminate all implication (→) and rewrite
    a. ∀x ¬ food(x) V likes(John, x)
    b. food(Apple) ∧ food(vegetables)
    c. ∀x ∀y ¬ [eats(x, y) ∧ ¬ killed(x)] V food(y)
    d. eats (Anil, Peanuts) ∧ alive(Anil)
    e. ∀x ¬ eats(Anil, x) V eats(Harry, x)
    f. ∀x¬ [¬ killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts).

- Move negation (¬)inwards and rewrite

a. ∀x ¬ food(x) V likes(John, x)

b. food(Apple) Λ food(vegetables)

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

d. eats (Anil, Peanuts) Λ alive(Anil)

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

f. ∀x ¬killed(x)] V alive(x)

g. ∀x ¬ alive(x) V ¬ killed(x)

h. likes(John, Peanuts).

- Rename variables or standardize variables

a. ∀x ¬ food(x) V likes(John, x)

b. food(Apple) Λ food(vegetables)

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

d. eats (Anil, Peanuts) Λ alive(Anil)

e. ∀w¬ eats(Anil, w) V eats(Harry, w)

f. ∀g ¬killed(g)] V alive(g)

g. ∀k ¬ alive(k) V ¬ killed(k)

h. likes(John, Peanuts).

- Eliminate existential instantiation quantifier by elimination.

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

- Drop Universal quantifiers.

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

  a. ¬ food(x) V likes(John, x)
  b. food(Apple)
  c. food(vegetables)
  d. ¬ eats(y, z) V killed(y) V food(z)
  e. eats (Anil, Peanuts)
  f. alive(Anil)
  g. ¬ eats(Anil, w) V eats(Harry, w)
  h. killed(g) V alive(g)
  i. ¬ alive(k) V ¬ killed(k)
  j. likes(John, Peanuts).

Distribute conjunction ∧ over disjunction ¬.

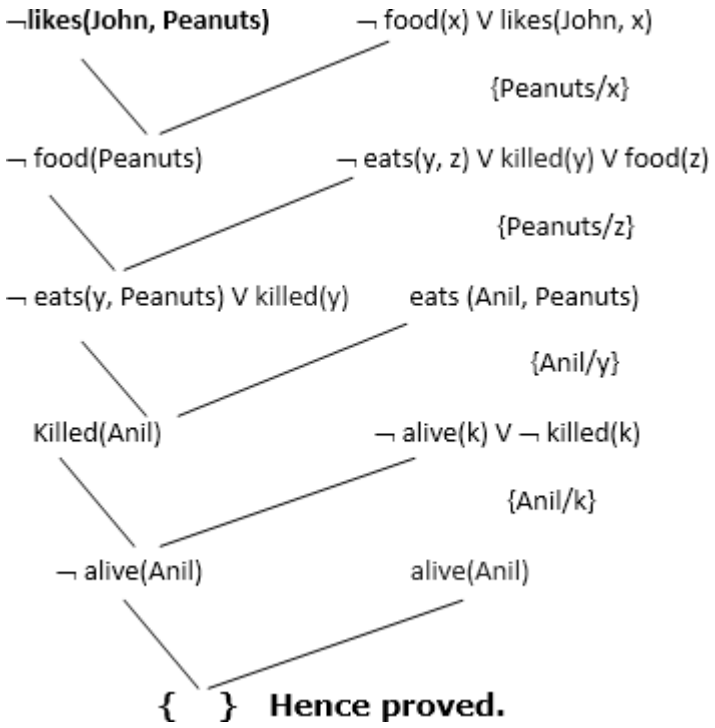
This step will not make any change in this problem.

Step-3: Negate the statement to be proved

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

Step-4: Draw Resolution graph:

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



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

Explanation of Resolution graph:

- In the first step of resolution graph, ¬likes(John, Peanuts) , and likes(John, x) get resolved(canceled) by substitution of {Peanuts/x}, and we are left with ¬ food(Peanuts)
- In the second step of the resolution graph, ¬ food(Peanuts) , and food(z) get resolved (canceled) by substitution of { Peanuts/z}, and we are left with ¬ eats(y, Peanuts) V killed(y) .
- In the third step of the resolution graph, ¬ eats(y, Peanuts) and eats (Anil, Peanuts) get resolved by substitution {Anil/y}, and we are left with Killed(Anil) .

- In the fourth step of the resolution graph, Killed(Anil) and ¬ killed(k) get resolve by substitution {Anil/k}, and we are left with ¬ alive(Anil) .
- In the last step of the resolution graph ¬ alive(Anil) and alive(Anil) get resolved.

Forward Chaining and backward chaining in AI

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

Inference engine:

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

a. Forward chaining
b. Backward chaining

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the first-order definite clause.Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

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 ∧ q → k.

A. Forward Chaining

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

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

Properties of Forward-Chaining:

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

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

Example:

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

Prove that "Robert is criminal."

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

Facts Conversion into FOL:

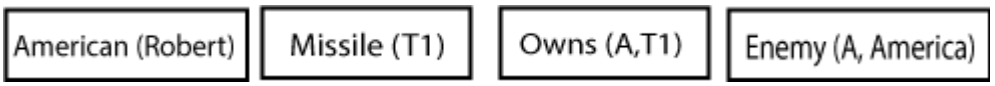
- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. ?p Owns(A, p) ∧ Missile(p). It can be written in two definite clauses by using Existential Instantiation, 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.



Step-2:

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

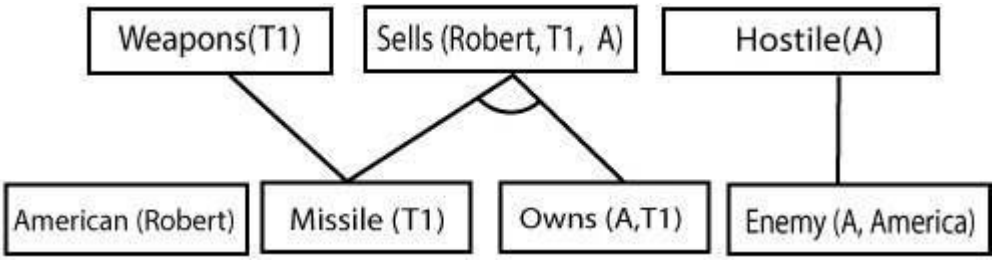
Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

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).

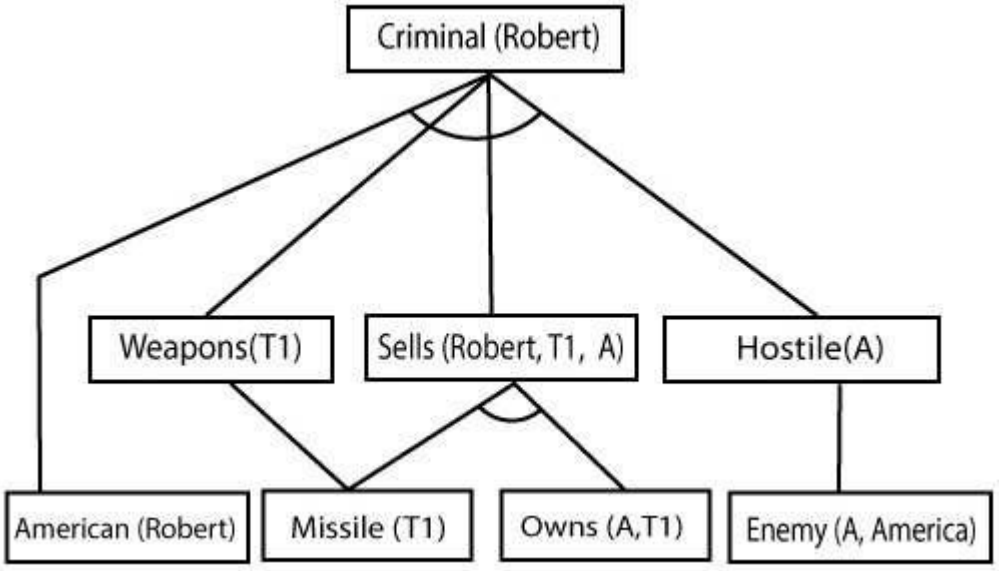
Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh



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.



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

B. Backward Chaining:

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

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

Properties of backward chaining:

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

Example:

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

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

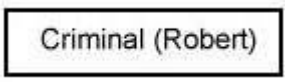
- American(Robert) ..... (8)

Backward-Chaining proof:

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

Step-1:

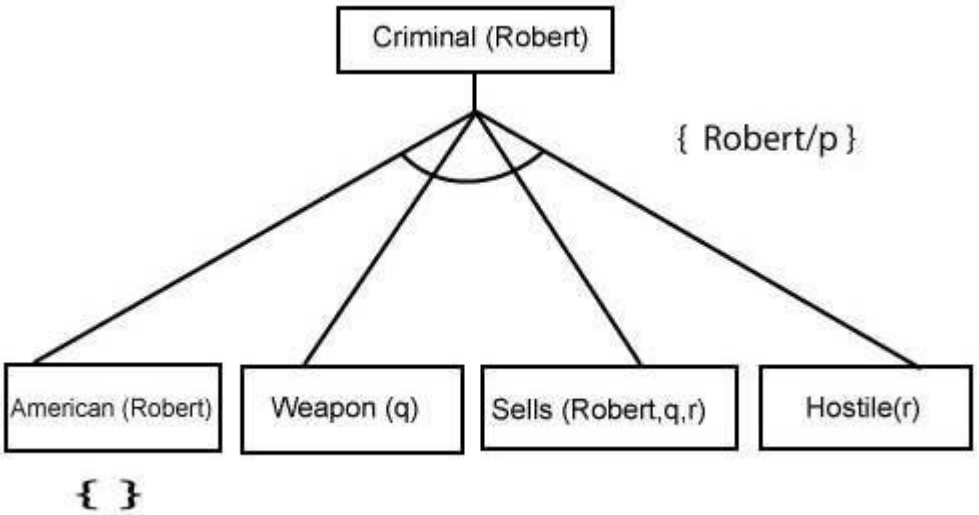
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.



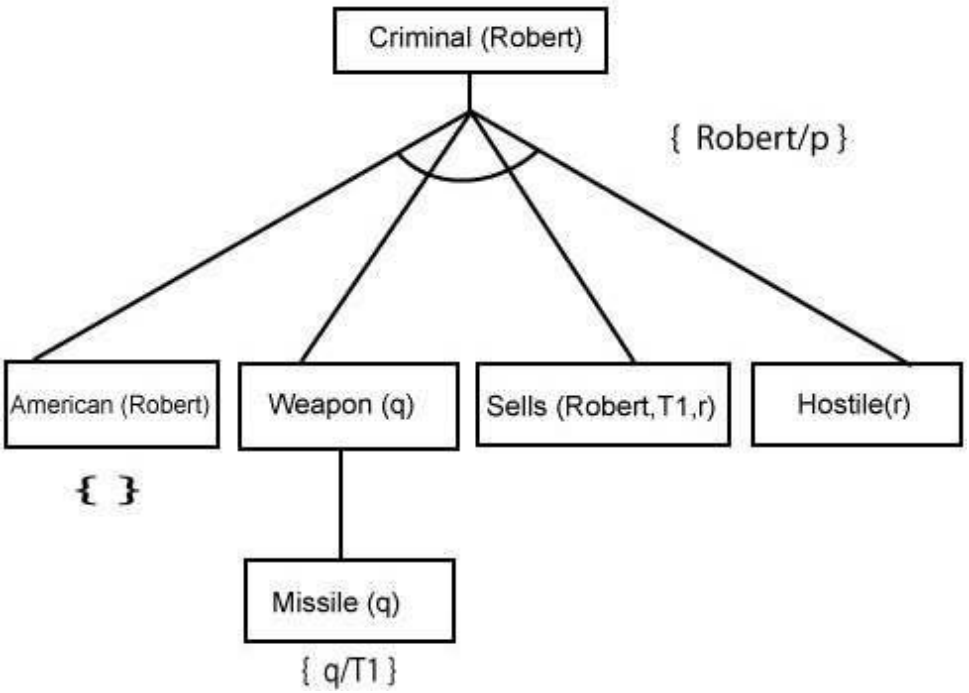
Step-2:

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

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

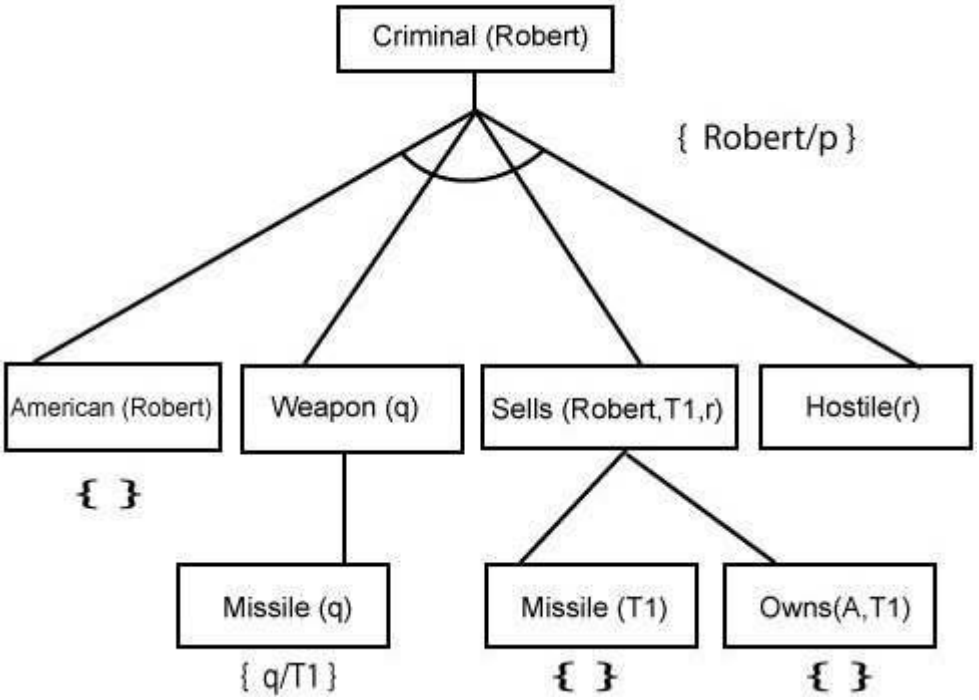


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



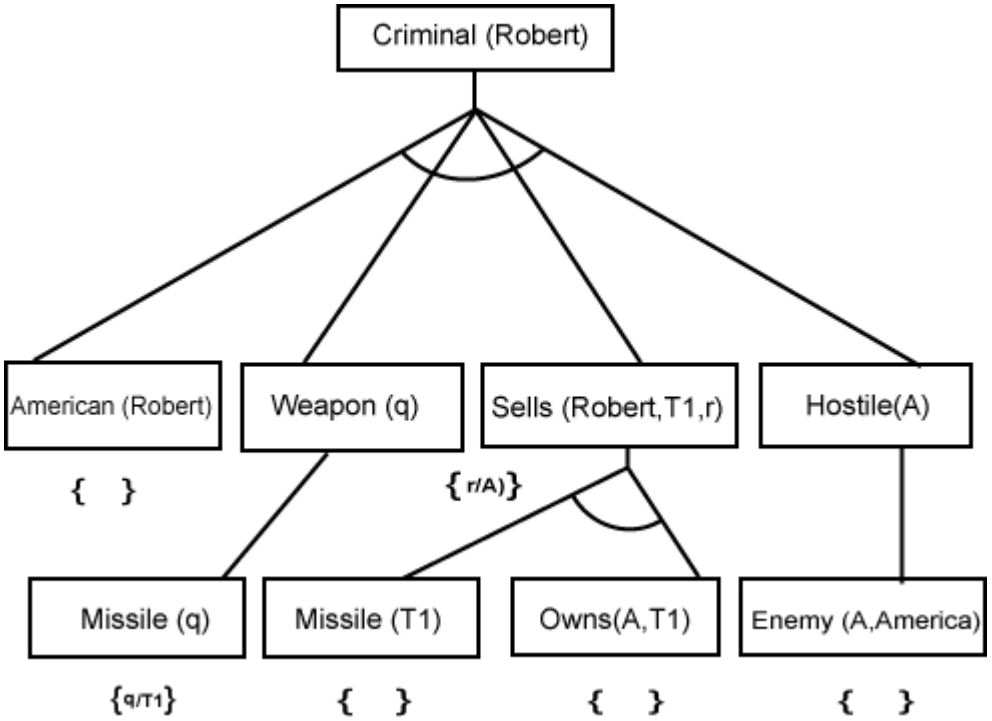
Step-4:

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



Step-5:

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



Difference between backward chaining and forward chaining

Following is the difference between the forward chaining and backward chaining:

- Forward chaining as the name suggests, start from the known facts and move forward by applying inference rules to extract more data, and it continues until it reaches to the goal, whereas backward chaining starts from the goal, move backward by using inference rules to determine the facts that satisfy the goal.
- Forward chaining is called a data-driven inference technique, whereas backward chaining is called a goal-driven inference technique.
- Forward chaining is known as the down-up approach, whereas backward chaining is known as a top-down approach.

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

- Forward chaining uses breadth-first search strategy, whereas backward chaining uses depth-first search strategy.
- Forward and backward chaining both applies Modus ponens inference rule.
- Forward chaining can be used for tasks such as planning, design process monitoring, diagnosis, and classification, whereas backward chaining can be used for classification and diagnosis tasks.
- Forward chaining can be like an exhaustive search, whereas backward chaining tries to avoid the unnecessary path of reasoning.
- In forward-chaining there can be various ASK questions from the knowledge base, whereas in backward chaining there can be fewer ASK questions.
- Forward chaining is slow as it checks for all the rules, whereas backward chaining is fast as it checks few required rules only.

www.android.previousquestionpapers.com | www.previousquestionpapers.com | https://telegram.me/jntuh

S. No.	Forward Chaining	Backward Chaining
1.	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2.	It is a bottom-up approach	It is a top-down approach
3.	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4.	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5.	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.

6.	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7.	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8.	It operates in the forward direction.	It operates in the backward direction.
9.	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

UNIT-IV

Syllabus: Knowledge Representation:
Ontological Engineering, Categories and Objects, Events, Mental Events and Mental Objects, Reasoning Systems for Categories, Reasoning with Default Information.
Classical Planning:
Definition of Classical Planning, Algorithms for Planning with State- Space Search, Planning Graphs, other Classical Planning Approaches, Analysis of Planning approaches

Ontology

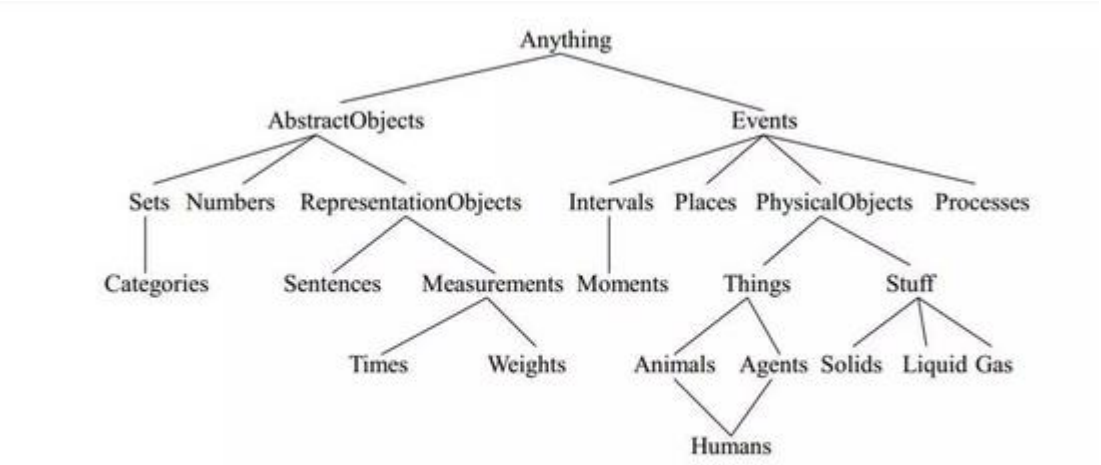
- An ontology consists of a set of concepts, axioms, and relationships that describe a domain of interest.

Ontological Engineering

Create more general and flexible representations

- Concepts like actions, time, physical object and beliefs
- Operates on a bigger scale than K.E.

- Representing a general-purpose ontology is a difficult task called *ontology engineering*
- Existing GP ontologies have been created in different ways:
  - By team of trained ontologists
  - By importing concepts from database(s)
  - By extracting information from text documents
  - By inviting anybody to enter commonsense knowledge
- Ontological engineering has only been partially successful, and few large AI systems are based on GP ontologies (use special purpose ontologies)



Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example.

Upper Ontology:

Limited to Meta, generic, abstract and philosophical and therefore are general enough to address (at a high level) a broad range of domain areas.

Provides a structure and a set of general concepts upon which domain specific ontologies (e.g. medical, financial engineering, etc.) could be constructed.

Categories and Objects

The organization of objects into categories 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

Categories also serve to make predictions about objects once they are classified.

Representation of categories in first-order logic:

Predicates

eg. Basketball(b)

Objects

eg. Basketball

Categories serve to organize and simplify the knowledge base through inheritance

Subclass relations organize categories into a taxonomy, or taxonomic hierarchy

Subset(Basketballs, Balls)

Examples

-An object is a member of a category. BB9 Basketballs

-A category is a subclass of another category.

Basketballs C Balls

-All members of a category have some properties.

$$\forall x\ (\text{MemberOf}(x,\text{Basketballs}) \Rightarrow \text{Round}(x))$$

or

(x Basketballs) Spherical (x)

- Members of a category can be recognized by some properties. Orange(x)

$$\forall x\ (\text{Orange}(x) \wedge \text{Round}(x) \wedge \text{Diameter}(x)=9.5\text{in} \wedge \text{MemberOf}(x,\text{Balls}) \Rightarrow \text{MemberOf}(x,\text{BasketBalls}))$$

- A category as a whole has some properties.

MemberOf(Dogs,DomesticatedSpecies)

- Two or more categories are ***disjoint*** if they have no members in common:
  - **Disjoint(s)⇔(∀ c_1,c_2 $c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow$ Intersection(c_1,c_2) ={})**
    Example:
    Disjoint({animals, vegetables})
- A set of *s* categories constitutes an ***exhaustive decomposition*** of a category *c* if all members of the set *c* are covered by categories in *s*:
  - **E.D.(s,c) ⇔ (∀ i $i \in c \Rightarrow \exists\ c_2$ $c_2 \in s \wedge i \in c_2$)**
    Example:
    ExhaustiveDecomposition({Americans, Canadians, Mexicans},NorthAmericans).

- A ***partition*** is a disjoint exhaustive decomposition:
  Partition(s,c) ⇔ Disjoint(s) ∧ E.D.(s,c)
  Example:
  Partition({Males,Females},Persons)
- Is ({Americans,Canadians, Mexicans},NorthAmericans) a partition?
- Categories can be defined by providing necessary and sufficient conditions for membership
  ∀ x Bachelor(x) ⇔ Male(x) ∧ Adult(x) ∧ Unmarried(x)

- Many categories have no clear-cut definitions (chair, bush, book).
  Tomatoes: sometimes green, red, yellow, black, mostly round.
  One solution:
  category *Typical(Tomatoes)*

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

  We can write down useful facts about categories without providing exact definitions

Objects:

The real world can be seen as consisting of primitive objects (e.g., atomic particles) and composite objects built from them.

- Stuff
- things

some properties are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole

Extrinsic properties-weight, length, shape, and so qn-are not retained under subdivision.

A category of objects that includes in its definition only intrinsic properties is then a substance, or mass noun; a class that includes any extrinsic properties in its definition is a count noun.

Events

how situation calculus represents actions and their effects.

- 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.
- Event calculus reifies fluents and events.

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.

- To assert that a fluent is actually true at some point in time we use the predicate T, as in

T(At(Shankar,Berkeley),t).

- Events are described as instances of event categories

- The event E1 of Shankar flying from San Francisco to Washington, D.C. is

(E1,SF) A Destination(E1,DC).

- The complete set of predicates for one version of the event calculus is T(f,t)

Fluent f is true at time t

- Happens (e, i)

Event e happens over the time interval i

-Initiates (e, f,t)

Event e causes fluent f to start to hold at time t

- Terminates (e, f,t)

Event e causes fluent tocease to hold at time t

- Clipped (f, i)

Fluent ceases to be true at some point during time inter

-Restored (f, i) Fluent becomes true sometime during time interval i

Processes

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

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

(e∈Processes)∧Happens(e,(t1,t4))^(t1<t2<t3<t4) Happens(e,(t2,t3))

Mental Events and Mental Objects

- Knowledge about one's own knowledge and reasoning processes is useful for controlling the inference

-The agent should know what are in its knowledge base and what are not

- What weneed is a model ofthe mental objects that are in someone's head (or some-thing's knowledge base) and of the mental processes that manipulate those mental objects.

- propositional attitudes that an agent can have toward mental objects: attitudes such as Believes, Knows, Wants, Intends, and Informs.

Reasoning in Artificial intelligence

In previous topics, we have learned various ways of knowledge representation in artificial intelligence. Now we will learn the various ways to reason on this knowledge using different logical schemes.

Reasoning:

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

Reasoning Systems for Categories

2 families of systems designed for organizing and reasoning with categories:

- **semantic networks** – utilizing graphs and algorithms for inferring properties of an object on the basis of its category membership
- **description logics** – provides a formal language for constructing and combining category definitions and 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.

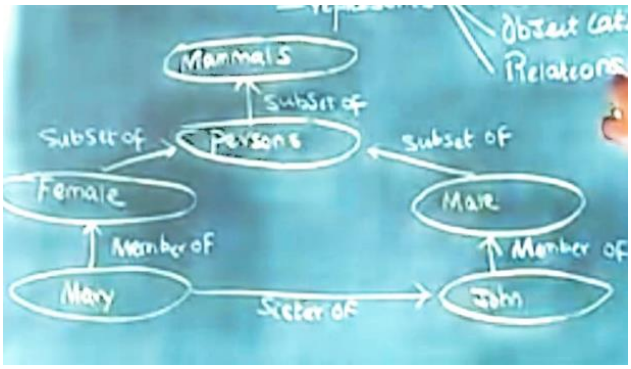


Figure 12.4.

For example, Figure 12.4 has a MemberOf link between Mary and FemalePersons, corresponding to the logical assertion Mary ∈ FemalePersons;

similarly, the SisterOf link between 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.

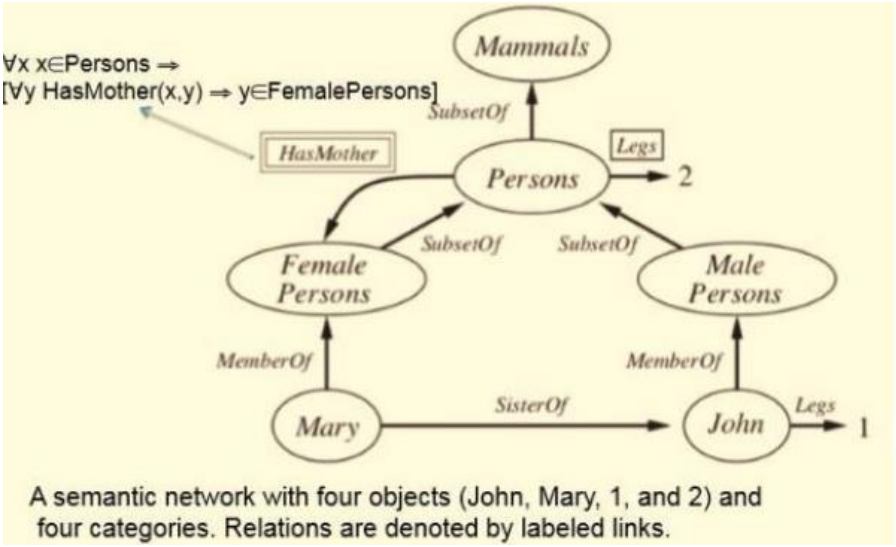


Figure 12.5

For this reason, we have used a special notation (the double-boxed link) in Figure 12.5. This link asserts that

∀x x∈Persons ⇒ [∀y HasMother(x,y) ⇒ y∈FemalePersons]

We might also want to assert that persons have two legs (the singly-boxed link) in Figure 12.5

∀x x∈Persons ⇒ Legs(x,2)

As before, we need to be careful not to assert that a category has legs; the single-boxed link in Figure 12.5 is used to assert properties of every member of a category.

semantic network notation makes it convenient to perform inheritance reasoning. However, inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called multiple inheritance Semantic Networks – Default Values

semantic network also has the ability to represent default values for categories. We say the default value is overridden by the more specific value (e.g. figure 12.5 John has 1 leg even though he is a person and all persons have 2 legs)

Drawbacks of Semantic Networks

- one drawback of semantic network notation (compared to first-order logic) is that links between nodes/bubbles represent only binary relations.

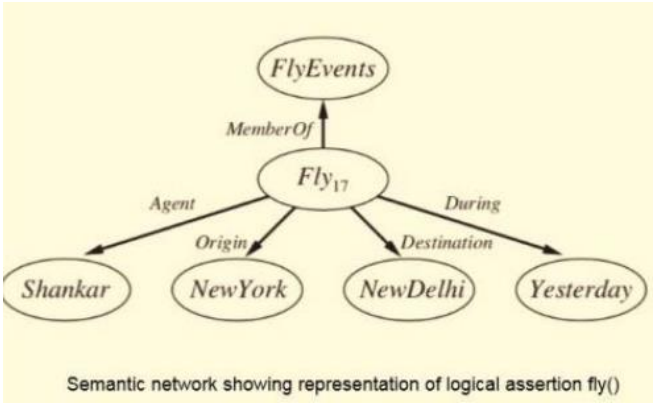


Figure 12.6

for example, to represent the following sentence in semantic networks is shown in figure 12.6 by reifying the proposition itself as an event belonging to an appropriate event category

Fly(Shankar, NewYork, NewDelhi, Yesterday)

- another drawback for this simple version of semantic networks, is that it still leaves out the full first-order logic such as representing negation, disjunction, nested function symbols, and existential quantification.

Description Logics: are designed to describe definitions and properties of categories.

- Formalization of semantic networks .

The principle **inference tasks** of 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
- **consistency** – some description logic system includes consistency of a category definition (whether the membership criteria are logically satisfiable)

-The CLASSIC language (Borgida et al., 1989) is a typical description logic

- Syntax of CLASSIC descriptions shown in figure 12.7

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

Figure 12.7:Syntax of Classic language

For example, to say “bachelors are unmarried adult males”:

Bachelor = And(Unmarried, Adult, Male).

The equivalent in **first-order logic**:

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

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

For example, to **describe the set of men with at least 3 sons who are all unemployed and married to doctors, and at most 2 daughters who are all professors in physics or math departments:**

$$\begin{aligned}&And(Man, AtLeast(3, Son), AtMost(2, Daughter),\\ &\quad All(Son, And(Unemployed, Married, All(Spouse, Doctor))),\\ &\quad All(Daughter, And(Professor, Fills(Department, Physics, Math)))) .\end{aligned}$$

Description Logic – Its Tractability and its Drawbacks the most important aspect of description logics is their emphasis on tractability of inference.

A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. Description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions

This sounds wonderful in principle, until one realizes that it can only have one of 2 consequences:

- either hard problems cannot be stated at all
- or they require exponentially large descriptions

However, the tractability results do shed light on what sorts of constructs cause problems and thus help the user to understand how different representations behave. For example, description logics usually lack negation and disjunction. Each forces first- order logical systems to go through a potentially exponential case analysis in order to ensure completeness. CLASSIC allows only a limited form of disjunction in the Fills and One Of constructs, which permit disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

Reasoning With Default Information

2 families of systems designed for organizing and reasoning with default information:

- circumscription & default logic
- Truth Maintenance Systems

Circumscription & Default logic

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.

2 examples that violate the monotonicity property of logic (and thus are non-monotonic):

- semantic networks we saw that property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory
- in closed-world assumption:

if a proposition α is not mentioned in KB then:
$KB \models \neg\alpha$

but when α is mentioned then:
$KB \wedge \alpha \models \alpha$

failures of monotonicity are widespread in commonsense reasoning

2 types of non-monotonic logics:

- circumscription
- default logic

Circumscription

A more powerful and precise version of the **closed world assumption**

Circumscription can be viewed as an example of a model preference logic. In such logics, 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. For circumscription, one model is preferred to another if it has fewer abnormal objects

for example, to assert the default rule that birds fly, we introduce a
predicate Abnormal1(x) and write

$$Bird(x) \wedge \neg Abnormal1(x) \Rightarrow Flies(x)$$

or in prolog syntax

flies(X) :- bird(X), not abnormal(X).

if we say that Abnormal1 is to be circumscribed, a circumscriptive reasoner is entitled to assume ¬Abnormal1(x) unless Abnormal1(x) is known to be true. This allows the conclusion Flies(Tweety) to be drawn from the premise Bird(Tweety), but the conclusion no longer holds if Abnormal1(Tweety) is asserted

Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the "Nixon diamond." It arises from the observation that 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) ∧ Quaker(Nixon)

Republican(x) ∧ ¬Abnormal2(x) ⇒ ¬Pacifist(x) Quaker(x) ∧ ¬Abnormal3(x) ⇒ Pacifist(x)

If we circumscribe Abnormal2 and Abnormal3, there are two preferred models: one in which Abnormal2(Nixon) and Pacifist(Nixon) hold and one in which Abnormal3(Nixon) and ¬Pacifist(Nixon) hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon was a pacifist. 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 Abnormal3 is minimized.

Default Logic

Default logic is a formalism in which default rules can be written to generate contingent, nonmonotonic conclusions

A default rule looks like this:

Bird(x) : Flies(x) / Flies(x)

This rules means that if Bird(x) is true, and if Flies(x) is consistent (causing no contradiction) with knowledge base, then Flies(x) may be concluded by default.

Default rule has the form:

P: J_1, …, J_n / C

where:

- P is called the prerequisite
- C is the conclusion

- J_i are the justifications (if any one of them is proven false, then conclusion cannot be drawn).

Any variable that appears in J_i and C must also appear in P

The Nixon-diamond example can be represented in default logic with 1 fact and 2 default rules:

```
 Republican(Nixon) ∧ Quaker(Nixon)          # fact
Republican(x) : ¬Pacifist(x) / ¬Pacifist(x)  # default rule
Quaker(x) : Pacifist(x) / Pacifist(x)        # default rule
```

Truth Maintenance Systems

Inferences drawn by the 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

The Difficulty of Belief Revisioning

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). This sounds easy enough. 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.

Truth Maintenance Systems (TMSs): are designed to handle the difficulty of belief revisioning

Types of TMSs:

- justification-based truth maintenance system (JTMS)
- assumption-based truth maintenance system (ATMS)

Justification-based truth maintenance system (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}

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

§ but if it also had the justification {R, P ∨ R ⇒ Q}, then it would be spared

§ In this way, the time required for retraction of P depends only on the number of sentences derived from P rather than on the number of other sentences added since P entered the knowledge base

Assumption-based truth maintenance system (ATMS): are efficient in context-switching between hypothetical worlds

JTMS vs ATMS

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. An ATMS represents all the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being in or out, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, 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

TMSs also provide a mechanism for generating explanations – an explanation of a sentence P is a set of sentences E such that E entails P

Unit-4

Planning:-

→ Task of coming up with a sequence of actions that will achieve a goal is called planning.

Scope of Environments:-

→ Fully Observable: we see everything that matters.

→ deterministic: The effect of actions known exactly.

→ static:- no changes to environment other than those caused by agent actions.

→ discrete:- changes in time & space occurs in quantum. (finite set of values/time)

→ single-agent:- No competition for a co-operation to account for.

Classical-Planning:-

→ Classical Planning considers only the environments that are fully observable, deterministic, static, discrete for coming up with sequence of actions to achieve a goal.

→ The foundation of everything from robotics to manufacturing, logistics to space exploration is classical planning, which offers an organized method for accomplishing objectives.

→ It can be used to solve problems in different domains

→ It's main feature is that it is domain independent and can be used to solve problems without having to learn domain knowledge or heuristics.

→ Planning languages used in classical planning are :

i) PDDL (Planning Domain Definition Lang.)
ii) STRIPS (Standford Research Institute Problem Solver)
iii) ADL (Action Description Lang)

→ classical Planning Alg. can be classified into 2 main approaches:

i) State-Space Search :-

These alg. are state space by producing successive state until reaching goal state.
Ex:- DFS, BFS, A*, etc.

ii) Plan-Space Search :-

These alg. are applied in the operational plan space, where the emphasis is on developing & fine tuning partial maps.

Non-classical Planning:-

→ It is used for partially observable & stochastic environments and involves diff. sets of algorithms & agents.

Planning Algorithms:-

→ Planning Algorithms are basically machine codes that are used to generate a goal to achieve.

→ They help construct a plan from current state to goal state.

→ The execute plans until they are finished and then begin again with new goal.

Algorithms for Planning with State Space Search:-

→ There are three algorithms for planning with state space search. They are:

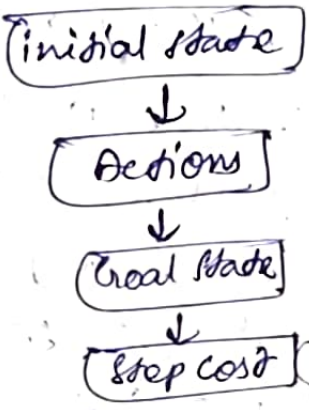
i) Forward State Space Search:-

→ It is also called as Progression planning.

→ Planning with Forward State Space search is similar to the problem solving approach.

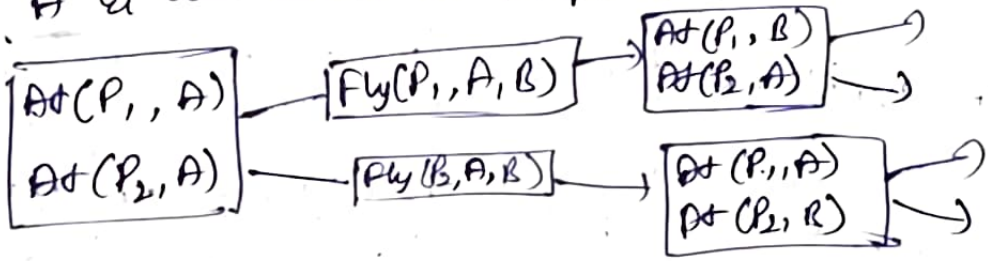
→ It is an alg. that searches forward from initial state of the world to find a state that satisfies the goal formula.

→ Steps involved in FSSS Alg. are



→ Ex:- Air Cargo Problem

There are 2 planes P_1 & P_2 located at Airport A & are moved to Airport B.



ii) Backward State Space Search :-

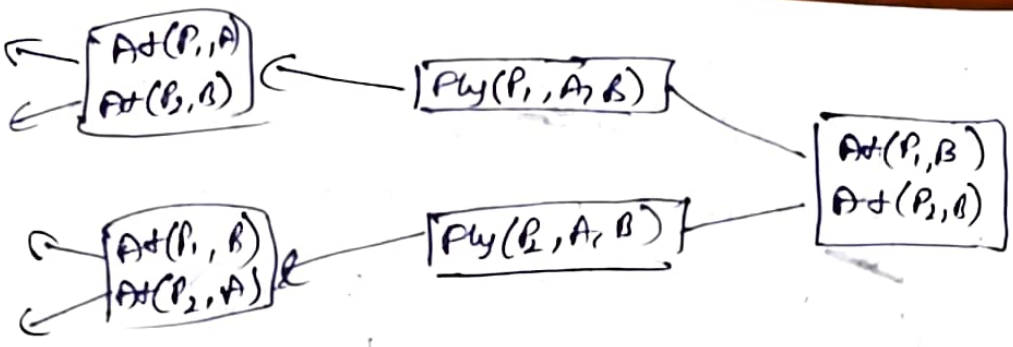
→ Also called regression planning.

→ It starts at the goal state and works backwards towards the initial state.

→ BSSS is also known as Backward Reasoning or Back propogation.

→ It moves from the goal state to a sub-goal, which determined the action needed to achieve the goal.

→ It uses inverse of actions to search backward to initial state.



iii) Heuristic State Space Search:-

→ The basic idea of this is to guess how many actions are needed to achieve all the goals.

→ Finding the exact number is hard, so it might derive an admissible heuristic - one that doesn't overestimate.
(estimate / guess the no. of actions)

→ There are 2 approaches for deriving: (no. of actions)

i) Derive a relaxed problem from the given problem specification. (doesn't consider obstacles goes in straight line) (perform learning on specific)

ii) Use subgoal independence that includes divide & conquer Alg. (dividing problem into subgoals & solve to find actions)

iv) Planning Graph:-

→ It is based on idea of reachability analysis in which it is a process of computing whether a set of states is reachable or not from an initial set of states.

→ We can extract a solution directly from planning graph using a specialized alg. called GRAPHPLAN which was developed in 1995 by "Avrim Blum" & "Merrick Furst".

→ It is an efficient way to create a representation of planning problem that can be used to achieve better heuristics

→ A planning graph consists of sequence of levels ~~that correspond to time steps in the plan~~, where level 0 is initial state. and each level contains literals & actions

→ Literals are all those that could be true at that time step (interval of time), depending on the actions executed at preceding time steps.

→ Actions are all those that could have their preconditions satisfied at that time step, depending on which of the literals actually hold.

→ Planning graphs have proved to be effective tools for solving hard planning problems.

→ Ex:- Have cake & Eat cake too problem.

Init(Have(Cake))
Goal(Have(Cake) ∧ Eaten(Cake))
Action(Eat(Cake))
(Precondition) Precond: Have(Cake)
Effect: ¬Have(Cake) ∧ Eaten(Cake)
Action(Bake(Cake))
Precond: ¬Have(Cake)
Effect: Have(Cake)

④ Other Classical Approaches:-

→ Let us see all approaches:

i) State-space search (Types)

2) Graph-Based Planning:- It involves creating a planning graph that captures relationship b/w actions & states over time. It is used to explore the possible actions & their effects to find a plan. Ex: Planning Graph

→ Other Classical Approaches are:

i) SAT Based Planning:- (Satisfiability)

→ This approach converts planning problem into a Boolean satisfiability problem (SAT).

→ The planner then uses SAT solvers to find a sequence of actions that satisfies goal conditions.

→ The process in this begins by creating a formula that represents the possible actions that can lead from initial to goal state.

→ The planner then utilizes a SAT solver, a specialized alg., to search for a sequence of action that satisfies all encoded constraints.

ii) Partial Order Planning (POP):-

→ It works on important decisions first rather than working on steps in chronological order.

→ It takes advantage of problem decomposition that works on several (divides) subgoals independently, solves them with several subplans & combines the subplans.

→ It takes advantage of flexibility in the order in which it constructs the plan.

→ Any planning algorithm that can place two actions in a plan without specifying which comes first is called a partial order planner.

→ It is totally opposite to totally order plans which work in chronological order like Forward & Backward SSS.

→ Delaying a choice during the search is called "least commit strategy". (choice = choosing a decision).

④ Analysis of Planning Approaches:-

→ Planning combines two major areas of AI.

① Search (exploring diff solutions to a problem)

② Logic. (focuses on formal reasoning & rules to make conclusions)

→ These lead to improvement in performance ~~according to several order of magnitude~~ significantly in last decade and to increase the use of planning in industrial applications.

→ Planning is managing the major challenge of "combinatorial explosion". This means that as the complexity of a problem grows, the number of possible solutions increase rapidly making it difficult to find optimal one.

→ Planning tools like GraphPlan and SATPlan have pushed the field forward by enhancing the performance of planning systems.

Planning & Acting in Real World:

→ In real-world scenarios, planning must consider various constraints like time, resources & scheduling.

→ Time in Planning:-

i) Temporal Constraints:- Time is critical in planning. Plans must be executed within specific time frames or deadlines.

ii) Time Representation:- It can be represented in discrete or continuous intervals.

iii) Temporal Planning Alg:- Alg like Temporal Planning N/ws (TPNs) and Simple Temporal N/w (STNs) are used.

→ Scheduling in Planning:- It involves assigning tasks to time slots while considering resource availability & constraints. Two types

- before execution (fixed) Static & Dynamic

→ Resources in Planning:-

i) Resource Constraints (Limited)
ii) Resource Management (optimal using)
iii) Multi Resource Planning: Planning according to use multiple resources simultaneously..

⑤ Heirarchical Planning:-

→ It is a technique used to manage the complexity of planning problems by breaking them into more managable sub problems.
→ The decomposition of tasks into subtasks can be solved individually before being integrated to a global solution.
→ It uses Heirarchical Task Network (HTN) Planning.
→ This planning allows the agent to take advice from domain designer in a form of decomposition.
→ This make it easy to create large plans required by many real world applications.
→ This alg. assumes complete & correct info and deterministic, fully observable environments.

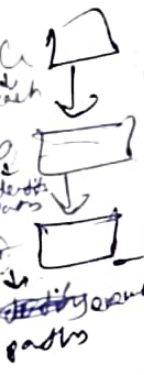
(iv) Planning & Acting in Non Deterministic Domains:-

→ Incomplete info can be dealt with by planning to use sensing actions to obtain the information needed.

→ Conditional plans allow the agents to sense the world during execution to decide what branch of plan to follow.

→ Here we deal with environment where outcome are always predictable...

→ Unlike deterministic domains, where a specific action leads to a ~~deterministic~~ specific outcome, these domains have multiple outcomes for single action.



→ This uncertainty makes this planning more complex. (unknown terrain)

→ It is used in Robot Navigation, Decision making systems. (Control over system doesn't have complete)

→ components are High level goals, Mid level plans low level actions.

(v) Multi-Agent Planning:-

→ It refers to the process of planning & decision making in systems where multiple autonomous agents interact and collaborate to achieve their goals.

→ Each agent in this system has its own objectives & abilities, and they must work together (or sometimes against each other) to accomplish tasks.

→ Sometimes there can be cooperative planning or competitive planning (game) in these systems.

→ There are 2 types of planning.

i) Centralized – A single entity makes decision for all agents.

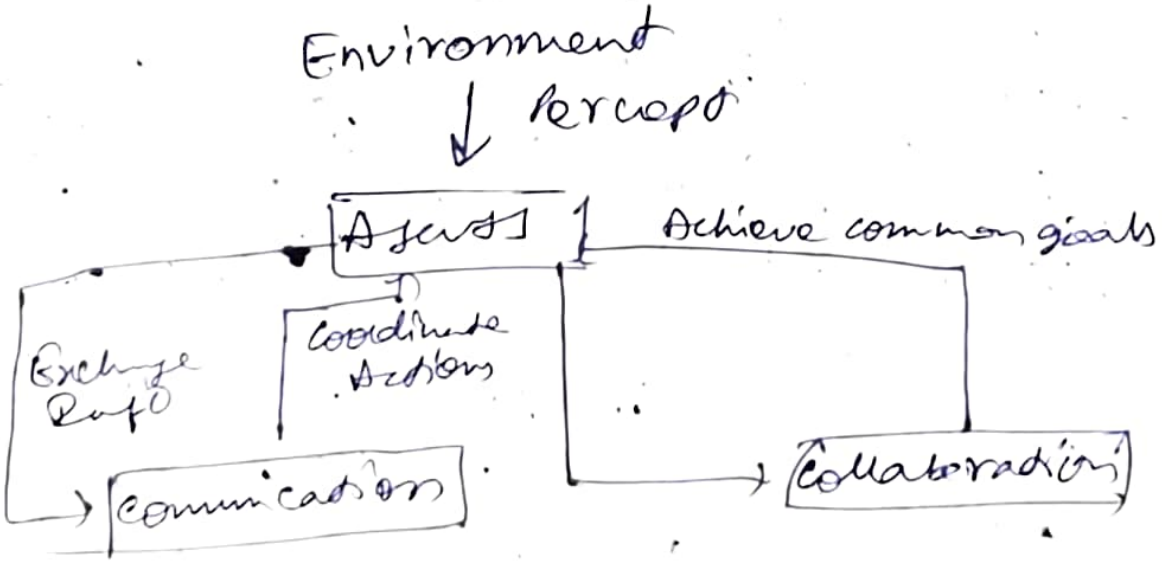
ii) Decentralized – Each agents makes their own decisions

→ Communication & Coordination is most important in these systems.

→ Ex :- Multiple robots working together.

→ Challenges :—

i) Communication overhead
ii) Conflict resolution
iii) Dynamic Environment



Classical Planning Overview

- Classical planning assumes a deterministic, fully observable environment
- States are represented as logical propositions
- Goal is to find a sequence of actions to reach a desired state
- Search is performed in a state space of possible worlds

1. Forward State-Space Search (Progression Planning)

- Start from initial state and apply applicable actions
- Generates new states by simulating action effects
- Goal test checks if current state satisfies goal conditions

2. Backward State-Space Search (Regression Planning)

- Start from goal and regress over actions to find predecessors
- Each node represents a goal condition that must be achieved
- Search continues until a goal is regressed to initial state
- Works well when goals are specific and few

3. Planning Graph Analysis

level

- Graph contains alternating layers of literals and actions
- Used to compute heuristics for forward search
- Can extract relaxed plans or check plan feasibility
- Used in GraphPlan and FF heuristics

4. Partial Order Planning (POP)

- Plans are partially ordered rather than totally ordered
- Handles concurrency and subgoal interactions
- Steps added incrementally with causal links and ordering constraints
- More flexible than linear planning algorithms

5. Heuristic State-Space Planning

- Enhance forward or backward search with heuristics
- Heuristics derived from relaxed problems or planning graphs
- Ignore delete lists to simplify problem
- Used in FastForward (FF) planner for efficient search

6. Hierarchical Task Network (HTN) Planning

- Break down high-level tasks into smaller subtasks
- Focuses on task decomposition rather than state transitions
- Used in real-world planning: robotics, military, games
- Not strictly classical—extends planning with structure

7. SATPlan (Satisfiability Planning)

- Translate planning problem into SAT (Boolean satisfiability)
- Incrementally increase plan length and solve SAT
- Check satisfiability at each length to find plan
- Leverages powerful SAT solvers for optimal planning

Summary of Classical Planning Algorithms

Algorithm	Search Space	Strategy	Best Use Case
Forward Search	State-space	Start from initial state and apply actions	General use with heuristic support
Backward Search	State-space	Regress goals through actions	When goals are small and focused
GraphPlan	Planning Graph	Use a layered graph to extract plans	Fast planning, cost estimation
Partial-Order Planning	Plan-space	Create flexible plans with minimal ordering	Complex domains with reusable subplans
Heuristic Planning	State-space + heuristics	Use relaxed problem for guidance	Scalable performance
SATPlan	Logic/SAT	Encode planning as satisfiability	Finding optimal-length plans

UNIT V

Uncertain knowledge and Learning Uncertainty:

Acting under Uncertainty, Basic Probability Notation, Inference Using Full Joint Distributions, Independence, Bayes' Rule and its use.

Probabilistic Reasoning:

Representing Knowledge in an Uncertain Domain, The Semantics of Bayesian Networks, Efficient

Representation of Conditional Distributions, Approximate Inference in Bayesian Networks,Relational and First-Order Probability, Other Approaches to Uncertain Reasoning; Dempster-

Shafer theory.

UNCERTAIN KNOWLEDGE AND REASONING

- Uncertainty
- Review of probability
- Probabilistic Reasoning
- Bayesian networks
- Inferences in Bayesian networks
- Temporal models
- Hidden Markov models

Uncertainty

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

CAR





Nature of Uncertain Knowledge

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

Nature of Uncertain Knowledge…

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

Nature of Uncertain Knowledge…

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

Probability

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

Probability

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

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

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

- All probability statements must indicate the evidence with respect to that probability is being assessed.
- If an agent receives new percepts, its probability assessments are updated to reflect the new evidence.

Random Variable

- Referring to a "part" of the world, whose "status" is initially unknown
- We will use lowercase for the names of values
  - P(a) = 1 - P(¬a)
- Tossing coin : P(h) = 1 - P(¬ h) : (0.5 = 1 - 0.5)
- Rolling dice : P(n) = 1 – P(¬n) : (0.16 = 1 - 0.84)

Types of random variables

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

Atomic events

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

Atomic events...

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

Prior Probability

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

Prior Probability...

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

Conditional Probability

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

Conditional probabilities...

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

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

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

Basic Axioms of Probability

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

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

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

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

- The probability of a disjunction is given by

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

Bayes Rule

- One of the most important rules for AI usage.
- Derived from the chain rule:
- P(A,B) = P(A | B)P(B) = P(B | A)P(A)
- With these powerful rule, the Bayes theorem constructed

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



Thomas Bayes was an English clergyman who set out his theory of probability in 1764.

Conditional probability…

- In some cases, given knowledge of one or more random variables, we can improve upon our prior belief of another random variable
- For example:
  - p(slept in movie) = 0.5
  - p(slept in movie | liked movie) = 1/3
  - p(didn't sleep in movie | liked movie) = 2/3

Liked movie	Slept	P
1	1	0.2
1	0	0.4
0	0	0.1
0	1	0.3

Bayes Rule

- One of the most important rules for AI usage.
- Derived from the chain rule:
- P(A,B) = P(A | B)P(B) = P(B | A)P(A)
- With these powerful rule, the Bayes theorem constructed

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



Thomas Bayes was an English clergyman who set out his theory of probability in 1764.

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

- Here P(B/A) is Probability of evidence given the hypothesis
- P(A) is the Prior probability of hypothesis
- P(B) is the Prior probability of evidence
- P(A/B) is the probability of hypothesis A conditional on a new piece of evidence B

$$P(Cause \mid Effect) = \frac{P(Effect \mid Cause)P(Cause)}{P(Effect)}$$

Bayes' Rule

- **Bayes' rule** allows unknown probabilities, to be computed from known conditional probabilities, usually in the very causal direction.

Example – Hypothesis for Flu based on Symptoms

- Given :
  - P(A) = Symptom of Flu = 0.00001
  - P(B/A) = Probability of Symptoms gives Flu = 0.95
  - P(B) = Your symptom of flu = 0.01 (head ache or running nose, 1 in 100)

- Compute P(A/B)
  - P(A/B) = 0.95 * 0.00001 / 0.01
  - = 0.00095 (less than one in thousand)

- Important to note the basic mistake
  - P(A/B) is not equal to P(B/A)
  - 0.00095 is not equal to 0.95

Bayesian Networks

- Joint probability distribution
- Bayesian Networks with examples
- Semantics of Bayesian Networks
  - Representing the full joint distribution
    - A method for constructing Bayesian Network
    - Compactness and node ordering
  - Conditional independence relation in Bayesian networks.

Joint probability distribution

- The **full joint probability distribution** specifies the probability of values to random variables.
- It is usually too large to create or use in its explicit form.
- Joint probability distribution of two variables X and Y are

Joint probabilities	X	X'
Y	0.20	0.12
Y'	0.65	0.03

- Joint probability distribution for n variables require 2^n entries with all possible combination.

Drawbacks of joint Probability Distribution

- Large number of variables and grows rapidly
- Time and space complexity are huge
- Statistical estimation with probability is difficult
- Human tends signal out few propositions
- The alternative to this is Bayesian Networks.

Bayesian Networks

- Bayesian Network is to represent the dependencies among variables and to give a brief specification of any full joint probability distribution.
- A Bayesian network is a directed graph in which each nodes are variables and edges are relations.
- The full specification is as follows:
  - 1. A set of random variables makes up the nodes of the network. Variables may be discrete or continuous.
  - 2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node *X* to node *Y,* X is said to be a parent of *Y.*
  - 3. Each node *X,* has a conditional probability distribution *P(X,(Parents(X,))* that quantifies the effect of the parents on the node. (X is parent of Y)
  - 4. The graph has no directed cycles (and hence is a directed, acyclic graph, or DAG).



Example



Bayesian Belief Network

- A & B are unconditional, independent, evidence and parent nodes
- C & D are conditional, dependent, hypothesis and child nodes.

Conditional Probability Table



P(A)	=	0.3
P(B)	=	0.6
P(C|A)	=	0.4
P(C|~A)	=	0.2
P(D|A, B)	=	0.7
P(D|A, ~B)	=	0.4
P(D|~A, B)	=	0.2
P(D|~A, ~B)	=	0.01

Conditional Probability Tables

P(A)	P(B)
0.3	0.6

A	P(C)
T	0.4
F	0.2

A	B	P(D)
T	T	0.7
T	F	0.4
F	T	0.2
F	F	0.01

- P(A,B,C,D) = P(D|A,B)*P(C|A)*P(B)*P(A)

Example 2

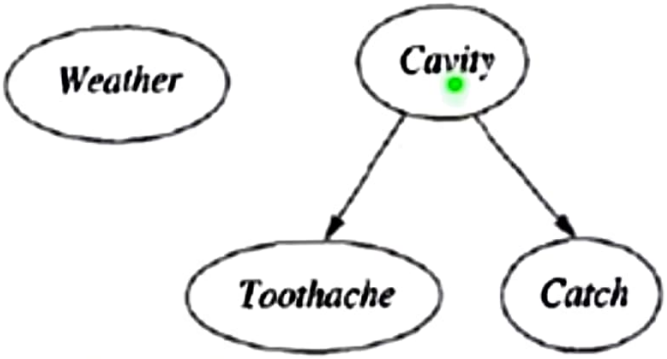


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given **Cavity**.

Example -3 - Burglar alarm

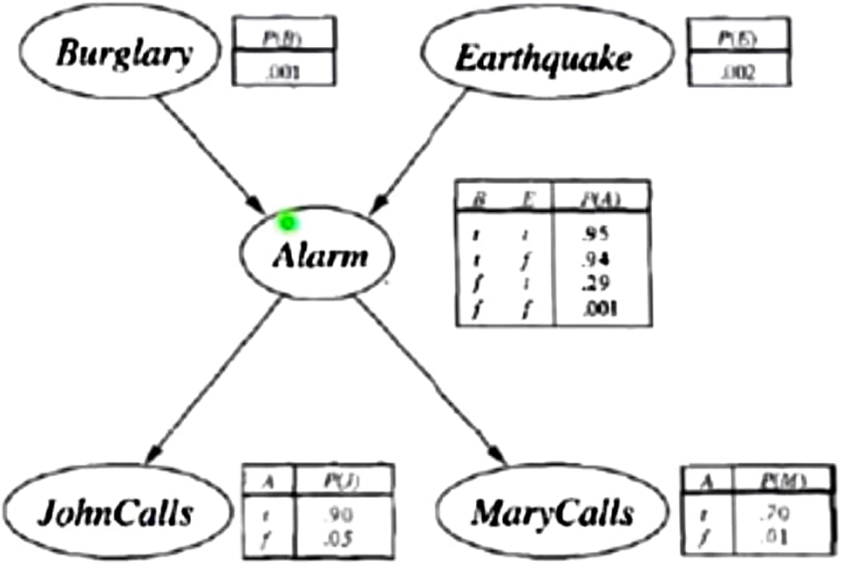
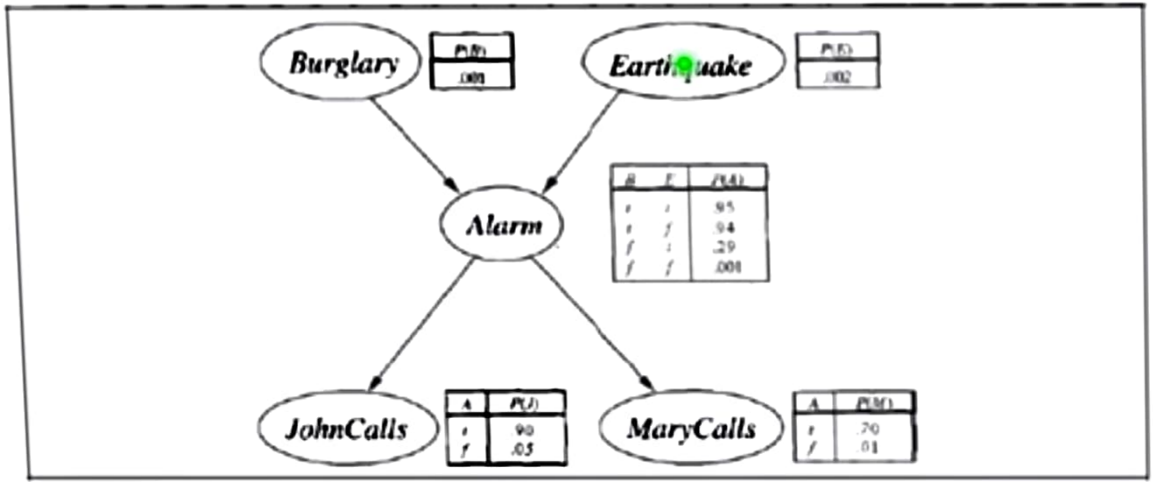


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and *M* stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

Example - Burglar alarm

- You have a new burglar alarm installed at home.
- It also responds on occasion to minor earthquakes.
- You also have two neighbors, John and Mary, they promised to call you when they hear the alarm.
- John always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm, and calls then, too.
- Mary likes rather loud music, and sometimes, she misses the alarm altogether.
- Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

- The burglary and earthquakes directly affect the probability of the alarm's going off,
- But, John and Mary call depends only on the alarm.
- The network does not have nodes for Mary's currently listening to loud music or the telephone ringing and confusing John.



Example

- We can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call.

$$
\begin{aligned}
&P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) \\
&\quad = P(j|a)P(m|a)P(a|\neg b\, A\, \neg e)P(\neg b)P(\neg e) \\
&\quad = 0.90\ x\ 0.70\ \mathrm{x}\ 0.001\ \mathrm{x}\ 0.999\ \mathrm{x}\ 0.998 = 0.0006'2.
\end{aligned}
$$

Semantics of Bayesian Network

- An entry in joint distribution is the probability of **conjunction** of particular assignment to each variable, such as
- $P(X1=x_1 \Lambda X2=x_2 \Lambda \ldots \Lambda..Xn=x_n)$ is equal to
- $P(x_1,x_2,\ldots,x_n) = \prod_{i=1}^{n} P(x_i \mid Parent(X_i))$

Method for Constructing Bayesian Network

- Rewrite the joint distribution in terms of a conditional probability, using the **product rule**

$$P(x_1, \ldots, x_n) = P(x_n|x_{n-1}, \ldots, x_1)P(x_{n-1}, \ldots, x_1)$$

- Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$\begin{aligned} P(x_1, \ldots, x_n) &= P(x_n|x_{n-1}, \ldots, x_1)P(x_{n-1}|x_{n-2}, \ldots, x_1) \cdots P(x_2|x_1)P(x_1) \\ &= \prod_{i=1}^{n} P(x_i|x_{i-1}, \ldots, x_1) . \end{aligned}$$

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

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

$$\mathbf{P}(MaryCalls|JohnCalls, Alarm, Earthquake, Burglary) = \mathbf{P}(MaryCalls|Alarm) .$$

Compactness and Node Ordering

- The compactness of Bayesian network is an example of general property of locally constructed systems. (also called as spare systems, inside some components there, and those are communicated)
- In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components.
- Therefore the correct order in which to add node is to add the 'root causes' first, then the variables they influenced and so on until we reach the leaves.



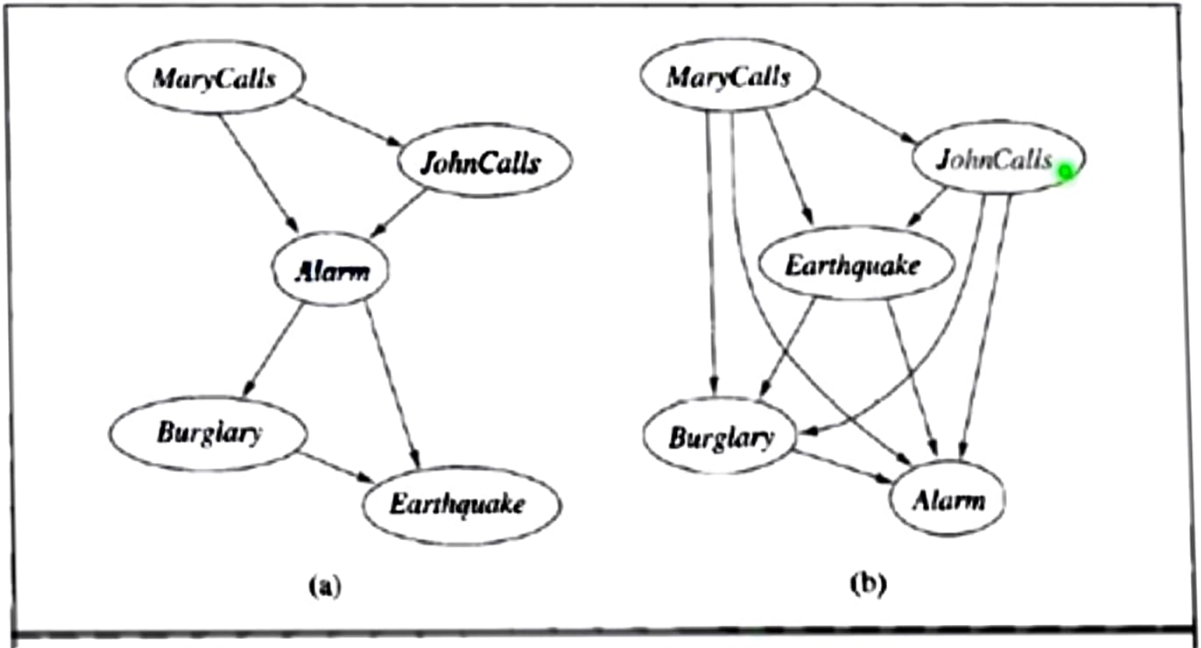


Figure 14.3 Network structure depends on order of introduction. In each network, we have introduced nodes in top-to-bottom order.

Conditional independence relations in Bayesian networks

- 1. A node is conditionally independent of its non-descendants, given its parents.
  - For example, *JohnCalls* is independent of .*Burglary* and *Earthquake*, given the value of *Alarm*.



Conditional independence relations in Bayesian networks...

- 2. A node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents-that is, given its Markov blanket.
  - For example, *Burglary* is independent of *JohnCalls* and *MaryCalls*, given *Alarm* and *Earthquake*.



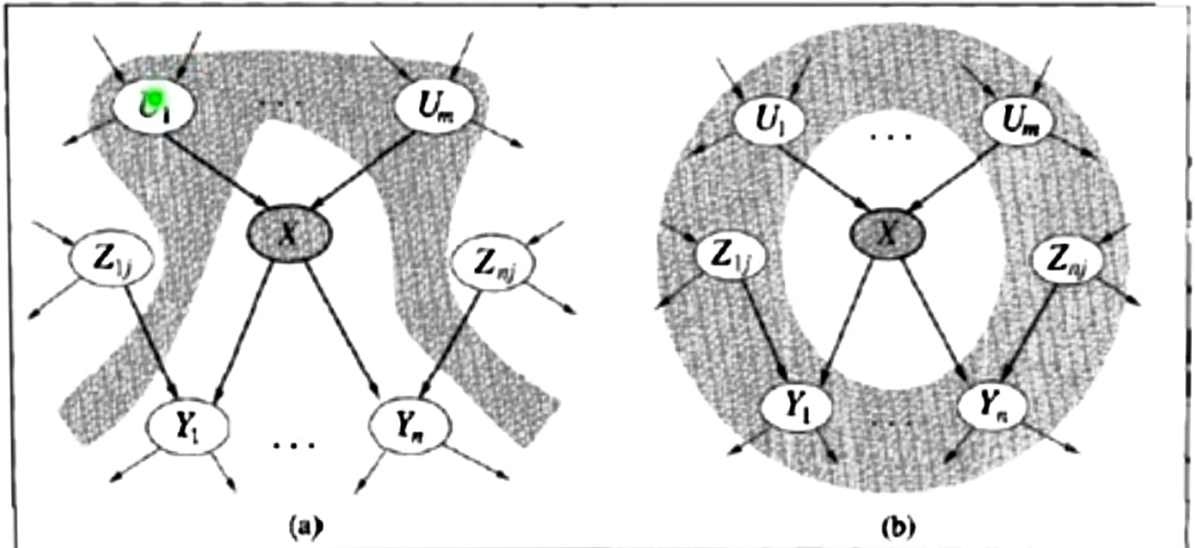


Figure 14.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

00:00:38

Basic probability notation

6. Inference using Full joint Distribution

- Probability inference means, computation from observed evidence of posterior probabilities, for query propositions. The knowledge based answering the query is represented as full joint distribution.

	Toothache		~Toothache	
	Catch	~Catch	Catch	~Catch
Cavity	0.108	0.012	0.072	0.008
~Cavity	0.016	0.064	0.144	0.576

- One particular common task in inferencing is to extract the distribution over some subset of variables or a single variable. This distribution over some variables or single variable is called as **marginal probability (Marginalization/ Summing).**

- **P(Cavity)** = 0.108+ 0.012 + 0.072 + 0.00 8
  - = 0.2

- Computing probability of a cavity, given evidence of a toothache is as follow:
- $\mathbf{P(Cavity \mid Toothache)} = P(Cavity \wedge Toothache) / P(Toothache)$

$= 0.108 + 0.012 / 0.108 + 0.012 + 0.016 + 0.064$

$= 0.6$

- Just to check also compute the probability that there is no cavity goven toothache is as follow:
- $\mathbf{P(\sim Cavity \mid Toothache)} = P(\sim Cavity \wedge Toothache) / P(Toothache)$

$= 0.016 + 0.064 / 0.108 + 0.012 + 0.016 + 0.064$

$= 0.4$

- Notice that in these 2 calculations the term 1/P (toothache) remains constant, no matter which value of cavity we calculate. With this notation we can write above two questions in one.
- **P**(Cavity | Toothache)

  = ∞ **P** (Cavity, Toothache)

  = ∞ [P(Cavity, Toothache, Catch) + P(~Cavity, Toothache, ~Catch)]

  = ∞ [<0.108 , 0.016> + <0.012 , 0.064>]

  = ∞ [<0.12, 0.08>] = [<0.6, 0.4>]

Dempster-Shafer Theory

A powerful framework for reasoning under uncertainty.

Problem: Identifying a Suspicious Email

- A system is trying to determine whether an email is spam or not spam based on two pieces of evidence:

1. **Evidence from content analysis:** It suggests the email is spam with 60% certainty.
2. **Evidence from sender reputation:** It suggests the email is spam with 70% certainty.

Contents

- Introduction
- Key Concepts: Belief, Plausibility, and Evidence
- Comparison with Bayesian Reasoning
- Steps in Applying Dempster-Shafer Theory
- Solved Example
- Applications in AI and Decision-Making Systems
- Advantages and Challenges

Introduction

- Uncertainty arises when exact knowledge is unavailable.
- Traditional probabilistic methods like Bayesian reasoning rely on prior probabilities.
- Dempster-Shafer theory (DST) extends probability theory by separating belief and plausibility.

Key Concepts

1. **Belief (Bel):** The degree of evidence supporting a hypothesis.
2. **Plausibility (Pl):** The degree to which a hypothesis could be true, given the evidence.
3. **Evidence:** Information that supports one or more hypotheses.

Mathematical Representation

- **Frame of Discernment (Θ):** All possible hypotheses.
- **Belief Function:**

$$Bel(A) = \sum_{B \subseteq A} m(B)$$

- **Plausibility Function:**

$$Pl(A) = 1 - Bel(\neg A)$$

Comparison with Bayesian Reasoning

Aspect	Bayesian Reasoning	Dempster-Shafer Theory
Evidence	Requires precise probabilities	Works with partial evidence
Prior Knowledge	Requires priors	Does not require priors
Hypothesis Support	Exact probability	Belief and plausibility

Steps in Applying DST

1. Define the frame of discernment.
2. Assign basic probability masses (m(A)) to subsets.
3. Combine evidence using Dempster's rule of combination

$$m(C) = \frac{\sum_{A \cap B = C} m_1(A) \cdot m_2(B)}{1 - \sum_{A \cap B = \emptyset} m_1(A) \cdot m_2(B)}$$

Problem: Identifying a Suspicious Email

- A system is trying to determine whether an email is spam or not spam based on two pieces of evidence:

1. **Evidence from content analysis:** It suggests the email is spam with 60% certainty.
2. **Evidence from sender reputation:** It suggests the email is spam with 70% certainty.

Step 1: Define the Frame of Discernment

- The frame of discernment is the set of all hypotheses:

$$\Theta = \{\text{Spam, Not Spam}\}$$

Step 2: Assign Basic Probability Masses

- **For content analysis:**
  - m1(Spam)=0.6
  - m1(Not Spam)=0.3
  - m1(Uncertainty)=0.1
- **For sender reputation:**
  - m2(Spam)=0.7
  - m2(Not Spam)=0.2
  - m2(Uncertainty)=0.1

Step 3: Apply Dempster's Rule of Combination

$$m(C) = \frac{\sum_{A \cap B = C} m_1(A) \cdot m_2(B)}{1 - \sum_{A \cap B = \emptyset} m_1(A) \cdot m_2(B)}$$

Compute the Intersection of Masses

A (from m_1)	B (from m_2)	Intersection $A \cap B$	$m_1(A) \cdot m_2(B)$
Spam	Spam	Spam	$0.6 \cdot 0.7 = 0.42$
Spam	Not Spam	Empty (∅)	$0.6 \cdot 0.2 = 0.12$
Spam	Uncertainty	Spam	$0.6 \cdot 0.1 = 0.06$
Not Spam	Spam	Empty (∅)	$0.3 \cdot 0.7 = 0.21$
Not Spam	Not Spam	Not Spam	$0.3 \cdot 0.2 = 0.06$
Not Spam	Uncertainty	Not Spam	$0.3 \cdot 0.1 = 0.03$
Uncertainty	Spam	Spam	$0.1 \cdot 0.7 = 0.07$
Uncertainty	Not Spam	Not Spam	$0.1 \cdot 0.2 = 0.02$
Uncertainty	Uncertainty	Uncertainty	$0.1 \cdot 0.1 = 0.01$

Normalize the Masses

- Conflict: Sum of masses where A∩B=∅:
  K=0.12+0.21=0.33
- Normalization factor:
  1−K=1−0.33=0.67
- Normalize masses:

$$m(\text{Spam}) = \frac{0.42 + 0.06 + 0.07}{0.67} = \frac{0.55}{0.67} \approx 0.82$$

$$m(\text{Not Spam}) = \frac{0.06 + 0.03 + 0.02}{0.67} = \frac{0.11}{0.67} \approx 0.16$$

$$m(\text{Uncertainty}) = \frac{0.01}{0.67} \approx 0.015$$

Step 4: Interpret the Results

- **Spam:** 82% belief.
- **Not Spam:** 16% belief.
- **Uncertainty:** 1.5%.

Applications in AI and Decision-Making

1. Expert Systems: Medical diagnosis, fault diagnosis.
2. Sensor Fusion: Combining data from multiple sensors in robotics.
3. Decision Support Systems: Risk analysis and financial modeling.

Advantages and Challenges

Advantages:

- Handles partial and conflicting evidence.
- Does not require prior probabilities.

Challenges:

- Computationally expensive.
- Difficulty in assigning mass functions.