We take great pleasure in presenting our diligently prepared course notes on artificial intelligence, where we have devoted our best efforts. Throughout this endeavor, we have ventured into exploring diverse AI concepts. The central concept that binds everything together in this study is that of an intelligent agent. In this context, Artificial Intelligence (AI) is defined as the exploration of agents capable of receiving information from their surroundings and taking actions accordingly. Each agent is equipped with a function that translates sequences of received information (percepts) into appropriate actions, and we delve into various methods of representing these functions.

We emphasize the significance of learning in expanding the capabilities of an agent beyond familiar environments. This expansion poses challenges for agent design, favoring explicit knowledge representation and reasoning. Throughout this course, we shed light on how learning impacts agent design and the importance of accommodating it effectively.

Primarily tailored as a fundamental resource for students studying AI courses, these course notes aim to serve as a primary reference for delving into the world of intelligent agents, their functions, and the dynamics of AI in different environments.

Course Notes Structure:
The course note has four Parts:

- Artificial intelligence
- Problem solving
- Knowledge and reasoning
- Uncertain Knowledge and reasoning

Unit1 Introduction to Artificial Intelligence: It provides a comprehensive overview of the historical development, foundational principles, and real-world applications of AI. The focal point is on intelligent agents - sophisticated systems capable of making decisions and subsequently executing actions based on their understanding.
Problem solving agents: It centers on a diverse range of search techniques aimed at tackling the complexities of decision-making, especially in navigation scenarios that demand thinking several steps ahead. In this unit various searching techniques for problem solving are uniformed search strategies, informed search strategies.

Unit II Beyond Classical Search: This unit investigates the various algorithm, which performs purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which the path cost is irrelevant and all that matters is the solution state itself.
Constraint satisfaction problems, Whose states and goal test conform to a standard,

structures and very simple representation.

Unit III Propositional logic: This unit examines about the knowledge-based agents. The representation of knowledge and the reasoning processes which brings knowledge to the entire field of AI. Here we also portray the simple logic called propositional logic.
Unit IV First-order logic: Which is sufficiently expressive to represent a good deal of our commonsense knowledge. This unit examines the representation languages in general and also covers the syntax and semantics of first – order logic.
Inference in first order logic: This unit introduces the inference rules for quantifiers and also shows how to reduce the first order inference to propositional inference.
Unit-V knowledge representation: It introduces the idea of general ontology, which organizes everything in the world into a hierarchy of categories. It also shows the representations of actions, basic categories of objects and substances and also cover the specialized reasoning systems for representing uncertain and changing knowledge.
Quantifying Uncertainty: This module focuses on the principles of reasoning and decision-making when confronted with uncertain situations in the world.

UNIT I:

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

1 Introduction:

- Artificial Intelligence is concerned with the design of intelligence in an artificial device.
- Artificial intelligence comprehends intelligent beings moreover it is capable of constructing intelligent entities.
- **The goal of AI** is to perform tasks, recognize patterns, and make decisions akin to human judgment.
- **Intelligence** is the ability to acquire, understand and apply the knowledge to achieve goals in the world.

- The **definition** of artificial intelligence encompasses two primary dimensions: thought processes and reasoning, as well as behavior. Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, able to learn and use knowledge to solve problems on their own.

  The term was coined by John McCarthy in 1956. His definition is "It is the Science and Engineering of making intelligent machines, especially intelligent computer programs."
- AI is the study of the mental faculties through the use of computational models
- AI is the study of intellectual/mental processes as computational processes.
- AI program will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task.
- AI is unique, sharing borders with Mathematics, Computer Science, Philosophy, Psychology, Biology, Cognitive Science and many others.

2. Foundations of Artificial Intelligence:

A brief history of the disciplines that contributed ideas, viewpoints, and techniques to AI are as follows:

2.1 Philosophy(the study of the fundamental nature of knowledge):

- 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?

- Aristotle (384–322 B.C.), was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which in principle allowed one to generate conclusions mechanically, given initial premises.

  Eg.

  all dogs are animals;

all animals have four legs;

therefore all dogs have four legs

- Thomas Hobbes (1588–1679) proposed that reasoning was like numerical computation that ―we add and subtract in our silent thoughts.‖
- Rene Descartes (1596–1650) gave the first clear discussion of the distinction between mind and matter and of the problems that arise.
- The empiricism movement, starting with Francis Bacon's (1561— 1626).
- The confirmation theory of Carnap and Carl Hempel (1905-1997) attempted to analyze the acquisition of knowledge from experience.
- Carnap's book The Logical Structure of the World (1928) defined an explicit computational procedure for extracting knowledge from elementary experiences. It was probably the first theory of mind as a computational process.
- The final element in the philosophical picture of the mind is the connection between knowledge and action. This question is vital to Al because intelligence requires action as well as reasoning.

2.2 Mathematics

- What are the formal rules to draw valid conclusions?
- What can be computed?

Formal science required a level of mathematical formalization in three fundamental areas: **logic, computation, and probability.**

Logic: George Boole (1815–1864), who worked out the details of propositional, or Boolean, logic.

In 1879, Gottlob Frege (1848–1925) extended Boole's logic to include objects and relations, creating the firstorder logic that is used today.

First order logic – Contains predicates, quantifiers and variables

E.g. Philosopher(a) ⇒ Scholar(a)

∀x, effect_carona(x) ⇒ quarantine(x)

∀x, King(x) ^ Greedy (x) ⇒ Evil (x)

Alfred Tarski (1902–1983) introduced a theory of reference that shows how to relate the objects in a logic to objects in the real world.

Logic and Computation: The first nontrivial algorithm is thought to be Euclid's algorithm for computing greatest common divisors(GCD).

- Beside logic and computation, the third great contribution of mathematics to AI is the **probability.** The Italian Gerolamo Cardanao (1501-1576) first framed the idea of probability, describing it in terms of the possible outcomes of gambling events.
- Thomas Bayes (1702-1761) proposed a rule for updating probabilities in the light of new evidence. Baye's rule underlies most modern approaches to uncertain reasoning in AI systems.

2.3 **Economics**

- How should we make decisions so as to maximize payoff?
- How should we do this when the payoff may be far in the future?
- The science of economics got its start in 1776, when Scottish philosopher Adam Smith treat it as a science, using the idea that economies can be thought of as consisting of individual agents maximizing their own economic well being.
- Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for decisions (economic or otherwise) made under uncertainty— that is, in cases where probabilistic descriptions appropriately capture the decision maker's environment.
- Von Neumann and Morgenstern's development of game theory included the surprising result that, for some games, a rational agent should adopt policies that are randomized. Unlike decision theory, game theory does not offer an unambiguous prescription for selecting actions.

2.4 Neuroscience:

- How do brain process information?
- Neuroscience is the study of the nervous system, particularly the brain.
- 335 B.C. Aristotle wrote, "Of all the animals, man has the largest brain in proportion to his size."
- Nicolas Rashevsky (1936, 1938) was the first to apply mathematical models to the study of the nervous system.

Fig. A neuron cell of human brain.

- The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG).
- The recent development of functional magnetic resonance imaging (fMRI) (Ogawa et al., 1990; Cabeza and Nyberg, 2001) is giving neuroscientists unprecedentedly detailed images of brain activity, enabling measurements that correspond in interesting ways to ongoing cognitive processes.

2.5 Psychology:

- How do humans and animals think and act?
- Behaviorism movement, led by John Watson(1878-1958). Behaviorists insisted on studying only objective measures of the percepts(stimulus) given to an animal and its resulting actions(or response). Behaviorism discovered a lot about rats and pigeons but had less success at understanding human.
- Cognitive psychology, views the brain as an information processing device. Common view among psychologist that a cognitive theory should be like a computer program.(Anderson 1980) i.e. It should describe a detailed information processing mechanism whereby some cognitive function might be implemented.

2.6 Computer engineering:

- How can we build an efficient computer?
- For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact (object) of choice.
- The first operational computer was the electromechanical Heath Robinson, built in 1940 by Alan Turing's team for a single purpose: deciphering German messages.
- The first operational programmable computer was the Z-3, the invention of KonradZuse in Germany in 1941.
- The first electronic computer, the ABC, was assembled by John Atanasoff and his student Clifford Berry between 1940 and 1942 at Iowa State University.
- The first programmable machine was a loom, devised in 1805 by Joseph Marie Jacquard (1752-1834) that used punched cards to store instructions for the

pattern to be woven.

2.7 Control theory and cybernetics:

- How can artifacts operate under their own control?
- Ktesibios of Alexandria (c. 250 B.C.) built the first self-controlling machine: a water clock with a regulator that maintained a constant flow rate. This invention changed the definition of what an artifact could do.
- Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an objective function over time.
- This roughly OBJECTIVE FUNCTION matches our view of Al: designing systems that behave optimally.
    - Calculus and matrix algebra- the tools of control theory
- The tools of logical inference and computation allowed AI researchers to consider problems such as language, vision, and planning that fell completely outside the control theorist's purview.

2.8 **Linguistics:**

- How does language relate to thought?
- In 1957, B. F. Skinner published Verbal Behavior. This was a comprehensive, detailed account of the behaviorist approach to language learning, written by the foremost expert in the field.
- Noam Chomsky, who had just published a book on his own theory, Syntactic Structures. Chomsky pointed out that the behaviorist theory did not address the notion of creativity in language.
- Modern linguistics and AI were ―born‖ at about the same time, and grew up together, intersecting in a hybrid field called computational linguistics or natural language processing.
- The problem of understanding language soon turned out to be considerably more complex than it seemed in 1957. Understanding language requires an understanding of the subject matter and context, not just an understanding of the structure of sentences.
- knowledge representation (the study of how to put knowledge into a form that a computer can reason with)- tied to language and informed by research in linguistics.

3 History of AI:

Important research that laid the groundwork for AI:

Maturation of Artificial Intelligence:

- ✔ 1943: The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of artificial neurons.

- ✔ 1950: The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes "Computing Machinery and Intelligence" in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a Turing test.

Birth of Artificial Intelligence:

- ✔ In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the **Dartmouth Conference**, the first conference devoted to the subject.

The golden years-Early enthusiasm:

- ✔ **1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.

- ✔ **1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1

A boom of AI:

- ✔ **1980:** After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert. In the Year 1980, the first national conference of the American Association of Artificial Intelligence was **held at Stanford University.**

The emergence of intelligent agents:

- ✔ **1997:** In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.

- ✔ **2002**: for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.

- ✔ **2006:** AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Deep learning, big data and artificial general intelligence:

- ✔ **2011:** In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.

- ✔ **2012:** Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.

- ✔ **2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."

- ✔ **2018:** The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.

4 Sub Areas of AI:

Artificial intelligence (AI) is a vast and interdisciplinary field that encompasses many different subfields or branches. These subfields focus on different aspects of creating intelligent systems, such as:

- ✔ **Machine Learning:** Machine learning is a branch of artificial intelligence that allows computers to learn without being explicitly programmed. This means that computers can learn from data and improve their performance over time, without having to be told what to do. The goal of machine learning is to develop algorithms that can automatically identify patterns and relationships in data, and use these patterns to make predictions or decisions. Machine learning can be divided into four main categories: Supervised learning, Unsupervised learning, Deep Learning, Reinforcement learning,

- ✔ **Natural language processing:** Natural language processing (NLP) is a field of artificial intelligence (AI) that deals with the interaction between computers and human language. NLP aims to enable computers to understand, interpret, and generate human language in a way that is similar to how humans do. This involves the use of algorithms, statistical models, and machine learning techniques to analyze and derive meaning from human language data. The task includes **a**)Text processing b) Text Analysis c) Speech processing d) Text Translation.

- ✔ **Computer vision:** Computer vision is a domain of artificial intelligence (AI) that deals with enabling machines to interpret and understand visual information from the world around them. It involves the use of algorithms and mathematical models to analyze, interpret, and process visual data, and has numerous applications in various industries. It involves image and video analysis, object detection, and Image recognition.

- ✔ **Robotics:** Robotics combines AI with mechanical engineering to create intelligent robots that can interact with the physical world. Robotics is a rapidly growing field with many potential applications including manufacturing, healthcare, and the military.

- ✔ **Expert systems:** Expert systems are a type of AI that has been around for decades and has proven to be effective in a variety of domains. They are designed to solve complex problems by reasoning through bodies of knowledge, represented mainly as if–then rules rather than through conventional procedural code. Expert systems can be used to improve the quality of decision-making in a variety of domains.

- ✔ **Knowledge representation and reasoning:** AI in Knowledge Representation and Reasoning encompasses the development of methods to portray and structure knowledge in ways that empower AI systems to engage in reasoning and make inferences.

- ✔ **Fuzzy logic:** Fuzzy logic is often used in artificial intelligence (AI) to model the uncertainty that is inherent in many real-world problems. Fuzzy logic is a type of logic that allows for partial truths. This means that a statement can be true to a certain degree, rather than being simply true or false.

- ✔ **Game AI:** It is a subfield of artificial intelligence (AI) that focuses on creating intelligent agents that can play and compete in games.ome of the most common techniques used in game AI include: Path finding, State space search, genetic algorithm.

5 Application of AI:

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless, for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches. Some of the key applications of AI include:

1. Business:
    - ✔ Customer service: AI-powered chatbots can answer customer questions and resolve issues 24/7.
    - ✔ Fraud detection: AI can be used to identify fraudulent transactions and prevent financial losses.
    - ✔ Marketing: AI can be used to target ads to specific customers and track the effectiveness of marketing campaigns.

2. Social media:
    - ✔ Content moderation: AI can be used to identify and remove harmful content from social media platforms.
    - ✔ Personalization: AI can be used to personalize news feeds and recommendations for users.
    - ✔ Advertising: AI can be used to target ads to specific users and track the effectiveness of advertising campaigns.

3. Engineering:
    - ✔ Design: AI can be used to help engineers design products and systems that are more efficient and reliable.
    - ✔ Construction: AI can be used to automate tasks on construction sites and improve safety.
    - ✔ Maintenance: AI can be used to monitor and diagnose equipment problems, preventing costly downtime.

4. Manufacturing:
    - ✔ Planning: AI can be used to help manufacturers plan production schedules and optimize resource allocation.
    - ✔ Quality control: AI can be used to inspect products for defects and improve quality.
    - ✔ Robotics: AI can be used to control robots that perform manufacturing tasks.
5. Medicine:
    - ✓ Diagnosis: AI can be used to help doctors diagnose diseases more accurately.
    - ✓ Treatment planning: AI can be used to help doctors develop treatment plans that are more effective and personalized.
    - ✓ Drug discovery: AI can be used to discover new drugs and improve the efficiency of drug development.

6. E-commerce:
    - ✔ Product recommendations: AI can be used to recommend products to customers based on their past purchases and browsing history.
    - ✔ Fraud detection: AI can be used to identify fraudulent transactions and prevent financial losses.
    - ✔ Pricing: AI can be used to set prices for products that are both competitive and profitable.

7. Entertainment:
    - ✔ Virtual assistants: AI-powered virtual assistants can help users with tasks such as setting reminders, making appointments, and controlling smart home devices.
    - ✔ Personalized recommendations: AI can be used to recommend movies, TV shows, music, and other entertainment content to users based on their interests.
    - ✔
8. Education:

- ✔ Personalized learning: AI can be used to personalize learning experiences for students based on their individual needs and abilities.
- ✔ Grading: AI can be used to grade student work more accurately and efficiently.
- ✔ Tutoring: AI-powered tutors can provide students with personalized help and feedback.

9. Fraud detection:
   - ✔ Credit card fraud: AI can be used to identify fraudulent credit card transactions.
   - ✔ Bank fraud: AI can be used to identify fraudulent bank transactions.
   - ✔ Insurance fraud: AI can be used to identify fraudulent insurance claims.

10. Surveillance:
    - ✔ Security: AI can be used to monitor security cameras and identify potential threats.
    - ✔ Traffic management: AI can be used to monitor traffic and optimize traffic flow.
    - ✔ Wildlife protection: AI can be used to monitor wildlife populations and identify poachers.

11. Information retrieval:

✔ Search engines: AI can be used to improve the accuracy and relevance of search results.
✔ Question answering: AI can be used to answer questions in a comprehensive and informative way.
✔ Translation: AI can be used to translate text from one language to another.

12. Space exploration:

✔ Planning: AI can be used to help plan space missions and optimize resource allocation.
✔ Navigation: AI can be used to navigate spacecraft through space.
✔ Robotics: AI can be used to control robots that perform tasks in space.

13. Gaming:

✔ Game AI: AI can be used to create more intelligent and challenging games.
✔ Virtual reality: AI can be used to create more immersive and realistic virtual reality experiences.
✔ Augmented reality: AI can be used to overlay digital information onto the real world.

6 **Building AI Systems:**

1) Perception

Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an autonomous vehicle, input might be images from a camera and range information

from a rangefinder. For a medical diagnosis system, perception is the set of symptoms and test results that have been obtained and input to the system manually.

2) **Reasoning**

Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc. Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.

3) **Action**

Biological systems interact within their environment by actuation, speech, etc. All behavior is centered around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

7 **Intelligent Systems**:

In order to design intelligent systems, it is important to categorize them into four categories:

1. Systems that think like humans
2. Systems that think rationally
3. Systems that act like humans:
4. Systems that act rationally

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

Figure.1 The definitions on the top, **(a)** and **(b)** are concerned with **reasoning**, whereas those on the bottom**, (c)** and **(d)** address **behavior.** The definitions on the left, **(a)** and **(c)** measure success in terms of human performance, and those on the right, **(b)** and **(d)** measure the ideal concept of intelligence called rationality

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

Table 1 Four categories of AI

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

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

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

✔ **Cognitive Science: Think Human-Like**

a. Requires a model for human cognition. Precise enough models allow simulation by computers.

b. Focus is not just on behavior and I/O, but looks like reasoning process.

c. Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

✔ **Laws of thought: Think Rationally**

a. The study of mental faculties through the use of computational models; that it is, the study of computations that make it possible to perceive reason and act.

b. Focus is on inference mechanisms that are probably correct and guarantee an optimal solution.

c. Goal is to formalize the reasoning process as a system of logical rules and procedures of inference.

d. Develop systems of representation to allow inferences to be like "*Socrates is a man. All men are mortal. Therefore Socrates is mortal"*

✔ **Turing Test: Act Human-Like**

a. The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which, at the moment, people do better.

b. Focus is on action, and not intelligent behavior centered around the representation of the world

c. Example: Turing Test

- 3 rooms contain: a person, a computer and an interrogator.

- The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance of voice of the person)

- The interrogator tries to determine which the person is and which the machine is.

- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.

- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

✔ **Rational agent: Act Rationally**

a. Tries to explain and emulate intelligent behavior in terms of computational process; that it is concerned with the automation of the intelligence.

b. Focus is on systems that act sufficiently if not optimally in all situations.

c. Goal is to develop systems that are rational and sufficient

Types of AI:

The difference between strong AI and weak AI:

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

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

8 INTELLIGENT AGENT'S:

8.1 AGENTS AND ENVIRONMENTS:

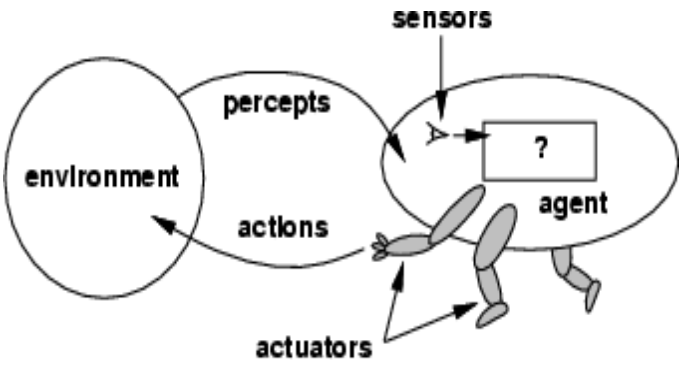


Fig 1 Agents and Environments

Agent:

An *Agent* is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- ✔ A *human agent* has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- ✔ A *robotic agent* might have cameras and infrared range finders for sensors and various motors for actuators.
- ✔ A *software agent* receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← RECOMMENDATION(seq, state)
  seq ← REMAINDER(seq, state)
  return action
```

Figure 2: Agent program

The Structure of Intelligent Agents

Agent = Architecture + Agent Program

Percept:

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

Percept Sequence:

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

Agent function:

Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any **given percept sequence to an action**.

Agent program

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

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in **Fig 3**. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in **Table.**

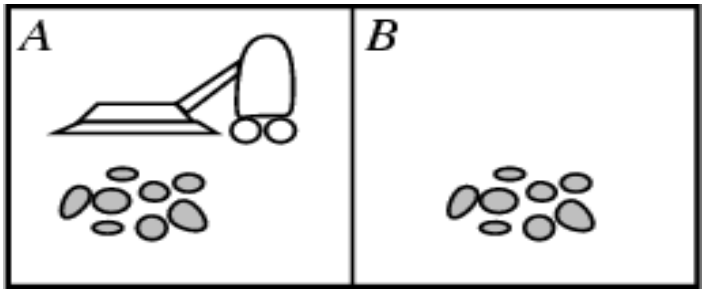


Fig 3: A vacuum-cleaner world with just two locations.

Agent function

Percept Sequence	**Action**
[A, Clean]	Right

[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
…	

Table 3 Partial tabulation of a simple agent function for the example: vacuum-cleaner world shown in the **Fig 3**

Fig 1.4 The REFLEX-VACCUM-AGENT program is invoked for each new percept (location, status) and returns **an** action each time

9 THE CONCEPT OF RATIONALITY :

- The **rational agent i**s one that does the right thing- conceptually speaking, every entry in the table for the agent function is filled out correctly.
- A **Performance measure** embodies the criterion for success of an agent's behavior.
- As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according, to how one think the agent should behave.

- **Rationality** 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.
- **Define Rationality:**

    For each possible percept sequence a rational agent should select an action that is expected to maximize its performance measure, give the evidence provided by the perfect sequence and whatever build in knowledge the agent has.

10 TASK ENVIRONMENT:

The task environments are essentially the 'problems' for which rational agents are the 'solutions'. For solving a problem rationally, we need to specify the **p**erformance measure, the **e**nvironment, and the agent's **a**ctuators and **s**ensors, which is called **PEAS**.

Agent Type	Performance measure	Environment	Actuators	Sensors
Taxi driver	Safe, Fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Table 2 PEAS description of the task environment for an automated taxi

Properties of agent:

The range of task environment arise and categories based on the following properties

i) **Fully observable vs partially observable**

If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.

ii) Deterministic vs stochastic:

If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.

A stochastic environment is random in nature and cannot be determined completely by an agent.

iii) Episodic vs sequential

In an episodic setting, a series of single, isolated actions occurs, with the present percept being sufficient for guiding each action.

In contrast, within a sequential context, the agent's decision-making necessitates the recollection of prior actions to ascertain the optimal course of subsequent actions.

iv) Static vs dynamic

If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment. Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.

v) Discrete vs continuous

An environment is classified as discrete if it offers a finite set of percepts and actions that can be executed within it; otherwise, it is termed a continuous environment.

vi) Single agent vs multi agent

When a solitary agent functions independently within an environment, it is referred to as a single-agent environment.

Conversely, if numerous agents operate within an environment simultaneously, it is termed a multi-agent environment.

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Figure: 4 Examples of task environment and their environments

11 TYPES OF AGENTS

Agents can be grouped into four classes based on their degree of perceived intelligence and capability:

- ✔ Simple Reflex Agents
- ✔ Model-Based Reflex Agents
- ✔ Goal-Based Agents
- ✔ Utility-Based Agents
- ✔ Learning Agent

i) The Simple reflex agents

• The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history (past State).
• These agents only succeed in the fully observable environment.
• The Simple reflex agent does not consider any part of percepts history during their decision and action process.
• The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.

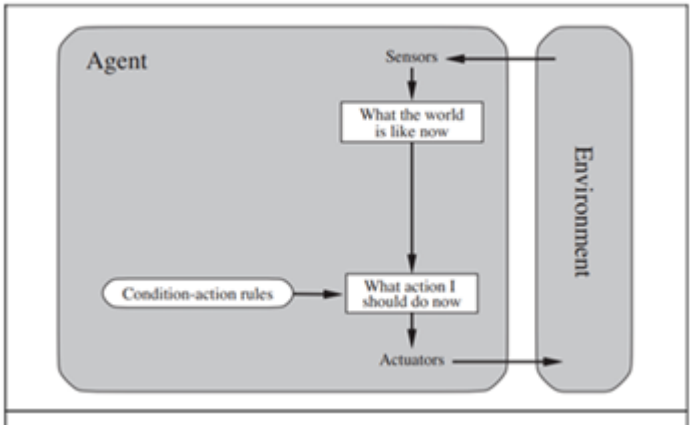


Figure 5: The Simple reflex agents

ii) Model Based Reflex Agents

The Model-based agent can work in a partially observable environment, and track the situation.

• A model-based agent has two important factors:

o Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.

o Internal State: It is a representation of the current state based on percept history.

• These agents have the model, "which is knowledge of the world" and based on the model they perform actions.

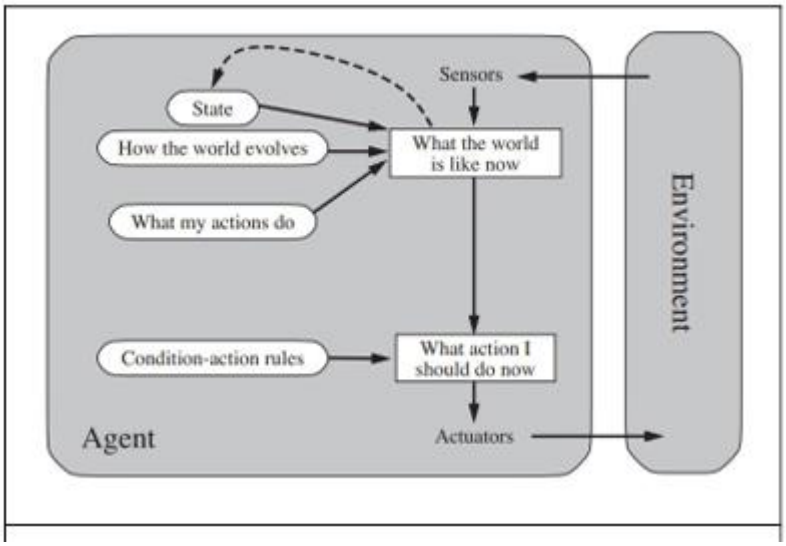


Figure 6: A model based reflex agent

iii) Goal Based Agents:

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

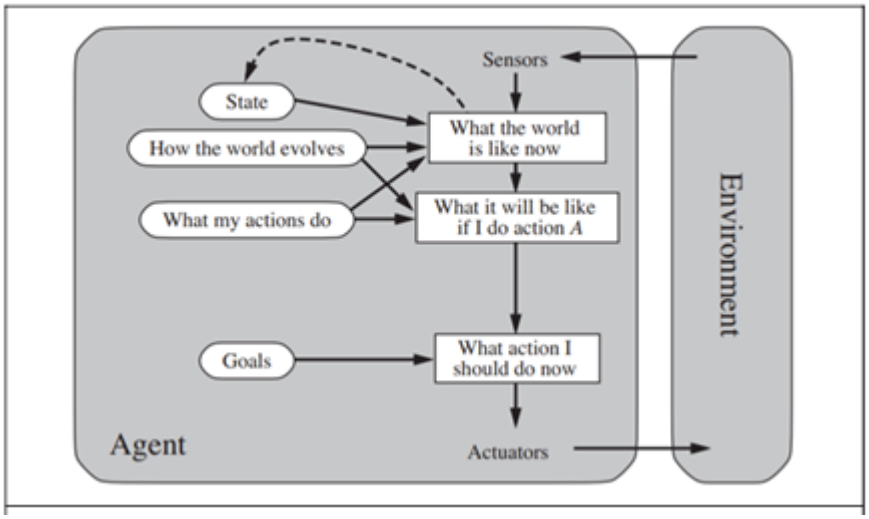


Figure7: Goal based agents

iv) Utility Based Agents

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

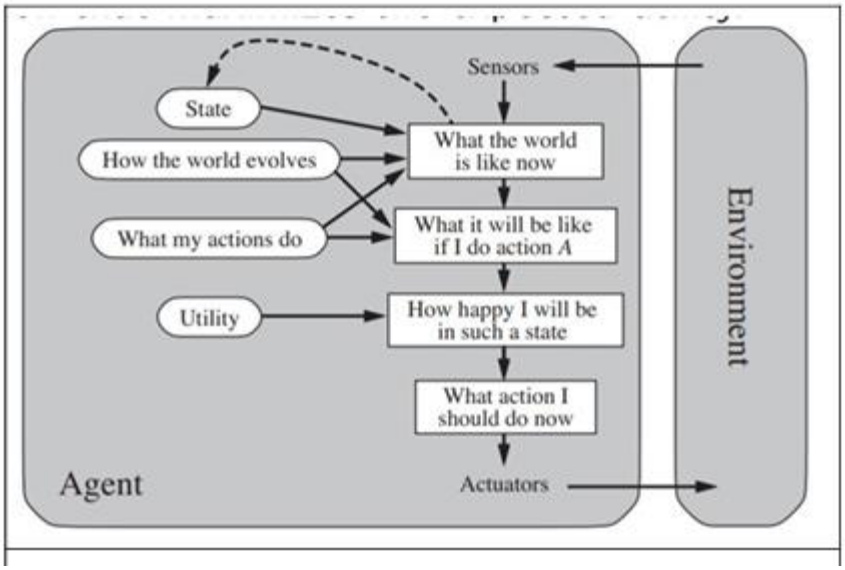


Figure 8: Utility Based Agents

v) Learning agents :

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
    - A learning agent has mainly four conceptual components, which are:
    - Learning element: It is responsible for making improvements by learning from environment
    - Critic: Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
    - Performance element: It is responsible for selecting external action
    - Problem generator: This component is responsible for suggesting actions that will lead to new and informative experiences.
- Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.

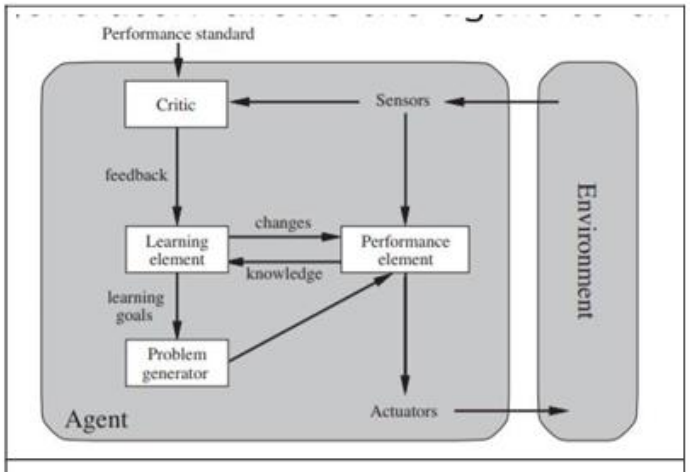


Figure 9: Learning agents

12 SEARCHING FOR SOLUTIONS

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem- solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

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

−a problem is characterized by a set of goals,

−a set of objects, and

−a set of operations.

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

Searching Solutions:

To build a system to solve a problem:

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

Defining the problem as State Space Search:

The state space representation forms the basis of most of the AI methods.

- Formulate a problem as a **state space search** by showing the legal problem states, the legal operators, and the initial and goal states.
- A **state** is defined by the specification of the values of all attributes of interest in the world
- An **operator** changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The **initial state** is where you start
- The **goal state** is the partial description of the solution

State Spaces versus Search Trees:

- State Space
    - Set of valid states for a problem
    - Linked by operators
    - e.g., 20 valid states (cities) in the Romanian travel problem
- Search Tree
    - Root node = initial state
    - Child nodes = states that can be visited from parent
    - Note that the depth of the tree can be infinite
        - E.g., via repeated states
    - Partial search tree
        - Portion of tree that has been expanded so far
    - Fringe
        - Leaves of partial search tree,

candidates for expansion Search trees = data structure

to search state-space

Properties of Search Algorithms

Which search algorithm one should use will generally depend on the problem domain.

There are four important factors to consider:

1. ***Completeness*** – Is a solution guaranteed to be found if at least one solution exists?

2. ***Optimality*** – Is the solution found guaranteed to be the best (or lowest cost)

solution if there exists more than one solution?

3. ***Time Complexity*** – The upper bound on the time required to find a solution,

as a function of the complexity of the problem.

4. ***Space Complexity*** – The upper bound on the storage space (memory)
required at any point during the search, as a function of the complexity of the
problem.

Formal Description of the problem:

1. Define a state space that contains all the possible configurations of the relevant objects.
2. Specify one or more states within that space that describe possible situations from which the problem solving process may start **(initial state).**

3. Specify one or more states that would be acceptable as solutions to the problem **(goal states).**
4. Specify a set of rules that describe the actions **(operations)** available.

Some of the most popularly used problem solving with the help of artificial intelligence are:

1. Chess.
2. Travelling Salesman Problem.
3. Tower of Hanoi Problem.
4. Water-Jug Problem.
5. N-Queen Problem.

Example 1: Route Estimation problem

A problem is defined by four items:

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

   e.g., *x* = "at Bucharest‖, *Checkmate(x)*
4. **path cost (additive)**

   e.g., sum of distances, number of actions executed, etc.

   c(x,a,y) is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

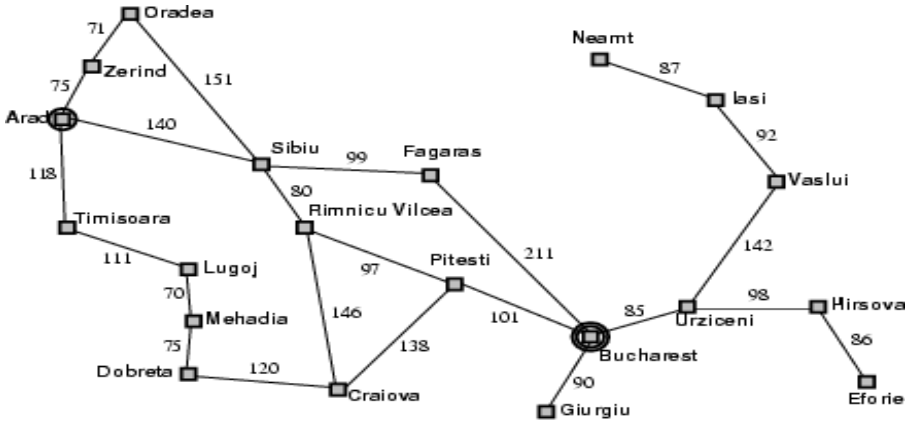


Figure 10 A simplified road map of part of Romania, with road distance in miles

Example 2: 8-**queens problem**

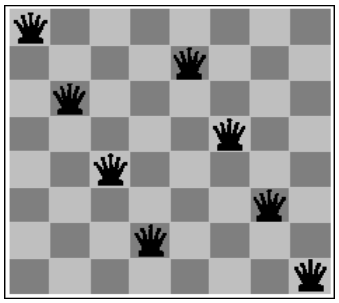


Figure 11: 8-queen problem

1. **Initial State**: Any arrangement of 0 to 8 queens on board.
2. **Operators**: add a queen to any square.
3. **Goal Test**: 8 queens on board, none attacked.
4. **Path cost**: not applicable or Zero (because only the final state counts, search cost might be of interest).

Example 3: Water Jug Problem

Consider the following problem:

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

State Representation and Initial State :

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

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

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

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

Through Graph Search, the following solution is found :

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

Example 4: Vacuum World States

Vacuum World States: The state is determined by both the agent location and the dirt locations.
The agent is in one of the 2 locations, each of which might or might not contain dirt. Thus there are 2*2^2=8 possible world states.
Initial state: Any state can be designated as the initial state.
Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure.
Goal test: This checks whether all the squares are clean. Path cost: Each step costs 1, so the path cost is the number of steps in the path.

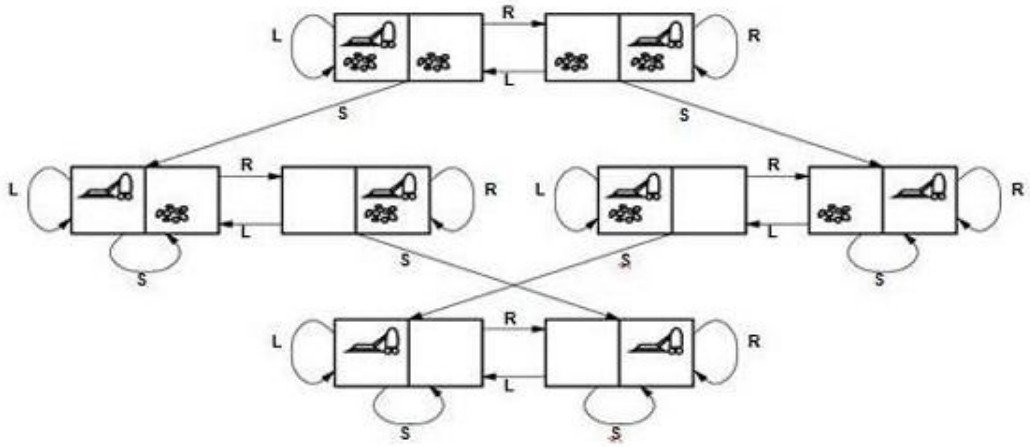


Figure 12: Vacuum World States

Example 5: 8-Puzzle Problem

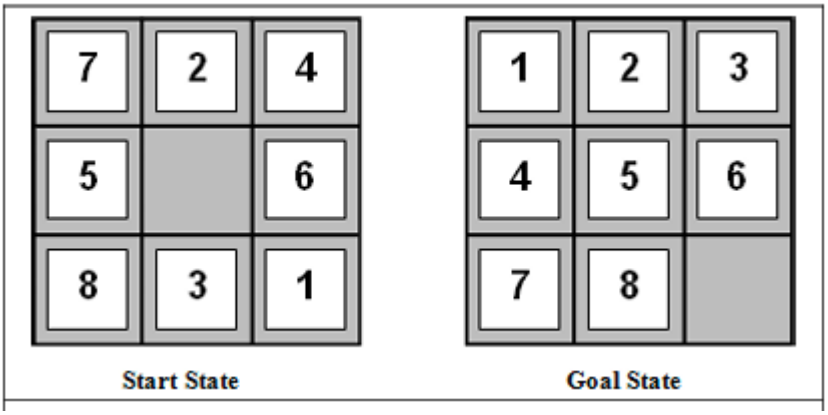


Figure 13: **8-Puzzle Problem**

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.

The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.

Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure the resulting state has the 5 and the blank switched.

Goal test: This checks whether the state matches the goal configuration shown in Figure.
Path cost: Each step costs 1, so the path cost is the number of steps in the path.

13 TYPES OF SEARCHING STRATEGIES

A **search algorithm takes** a search problem as input and returns a solution, or an indication of failure. The 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.

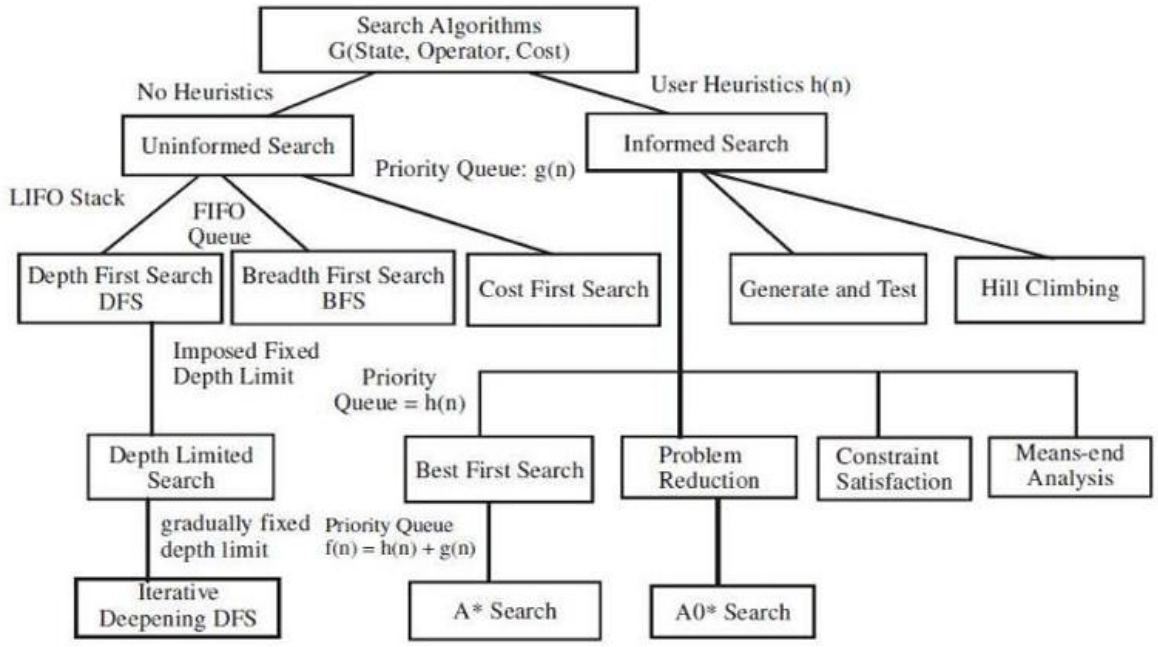


Figure 14: types of search strategies

13.1 UNINFORMED SEARCH STRATEGIES

- Also called blind, exhaustive or brute-force search, uses no information about the problem to guide the search and therefore may not be very efficient.
- Strategies that know whether one non goal state is "more promising" than another are called Informed search or heuristic search strategies.

- There are five uninformed search strategies as given below.
    - o Breadth-first search
    - o Uniform-cost search
    - o Depth-first search
    - o Depth-limited search
    - o Iterative deepening search

i) **Breadth-first search,** also known as BFS, is a straightforward technique where exploration begins from the root node, followed by expansion of all successors of the root node, and then their respective successors, and so forth. Essentially, at a particular depth level in the search tree, all nodes are expanded prior to progressing to the subsequent depth level.

The mechanism of breadth-first search is achieved by invoking TREE-SEARCH with an empty fringe designed as a first-in-first-out (FIFO) queue. This ensures that the nodes visited earliest will be the ones expanded first. In simpler terms, utilizing TREE-SEARCH (problem, FIFO-QUEUE()) leads to the execution of breadth-first search. The FIFO queue appends newly generated successor nodes at the queue's tail, thus prioritizing the expansion of shallower nodes over deeper ones.

Algorithm:

1. Create a variable called NODE-LIST and set it to initial state
2. Until a goal state is found or NODE-LIST is empty do
    a. Remove the first element from NODE-LIST and call it E.

       If NODE-LIST was empty, quit
    b. For each way that each rule can match the state described in E do:
        i. Apply the rule to generate a new state
        ii. If the new state is a goal state, quit and return this state
        iii. Otherwise, add the new state to the end of NODE-LIST

BFS illustration:

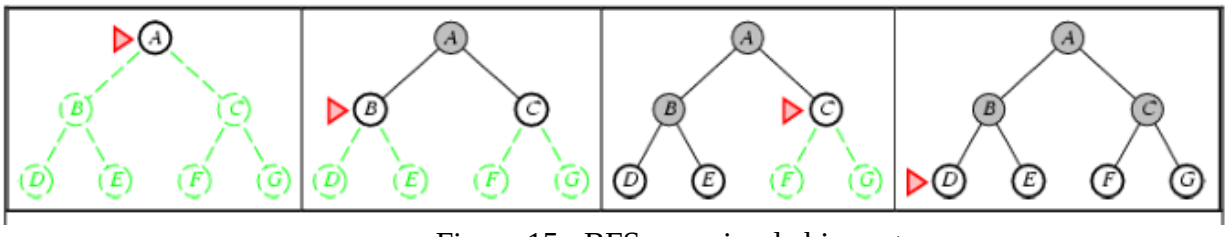


Figure 15 : BFS on a simple binary tree

Properties of BFS

- Complete? Yes (if *b* is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d = b(b^d-1) = O(b^{d+1})$

- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

Space is the bigger problem (more than time)

Maximum length of any path (m) in search space

Advantages: Finds the path of minimal length to the goal.

Disadvantages:

- Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node.
- The breadth first search algorithm cannot be effectively used unless the search space is quite small.

ii) Uniform cost search

Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost. Uniform-cost search does not care about the number of steps a path has, but only about their total cost.

Properties:

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

iii) Depth first Search

Depth-first search (DFS) represents a recursive technique employed to traverse tree or graph data structures. Its name derives from the fact that it commences exploration from the root node, pursuing each route to its utmost depth node prior to progressing to the subsequent route. The implementation of DFS utilizes a stack data structure. The operational process of the DFS algorithm exhibits resemblances to the BFS algorithm.

- **Algorithm:**
  1. Create a variable called NODE-LIST and set it to initial state
  2. Until a goal state is found or NODE-LIST is empty do
     a. Remove the first element from NODE-LIST and call it E.

If NODE-LIST was empty, quit

b. For each way that each rule can match the state described in E do:
   i. Apply the rule to generate a new state
   ii. If the new state is a goal state, quit and return this state
   iii. Otherwise, add the new state in front of NODE-LIST

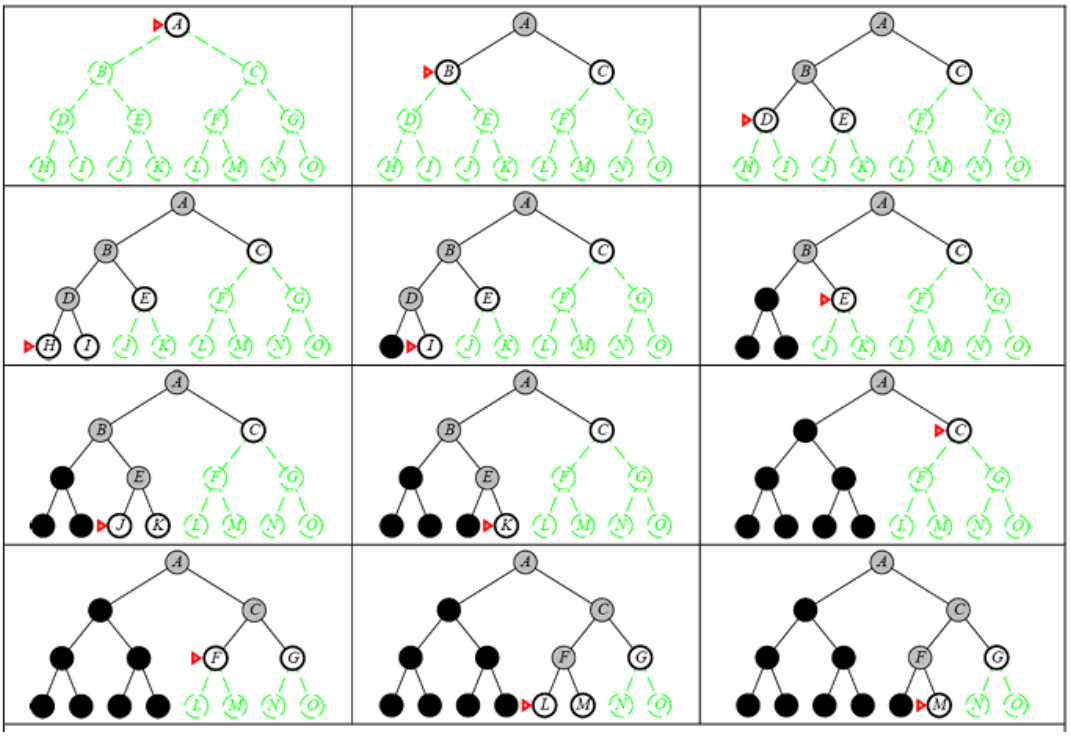


Figure DFS on a binary tree

Properties :

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Drawback of Depth-first-search:

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

iv) Depth limited search

A variant of depth-first search called backtracking search uses less memory and only one

successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm). Depth limited search is backtracking search.

- It is depth-first search
  - with a predefined maximum depth
  - However, it is usually not easy to define the suitable maximum depth
  - too small no solution can be found
  - too large the same problems are suffered from
- Anyway the search is
  - complete
  - but still not optimal

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l. That is, nodes at depth l are treated as if they have no successors. This approach is called depth-limited-search. The depth limit solves the infinite path problem.

Depth-limited-search can be implemented as a simple modification to the general treesearch algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth-limited-search is shown in Figure.

It can be noted that the above algorithm can terminate with two kinds of failure : the standard failure value indicates no solution; the cutoffvalue indicates no solution within the depth limit. Depth-limited search = depth-first search with depth limit l,returns cut off if any path is cut off by depth limit

Algorithm:

Depth limited search Illustration

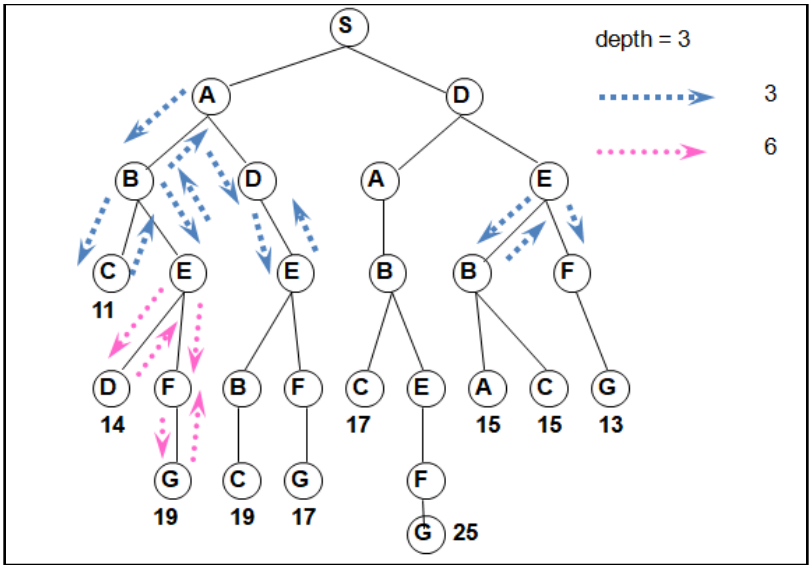


Figure 17: Illustration of Depth limited search

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

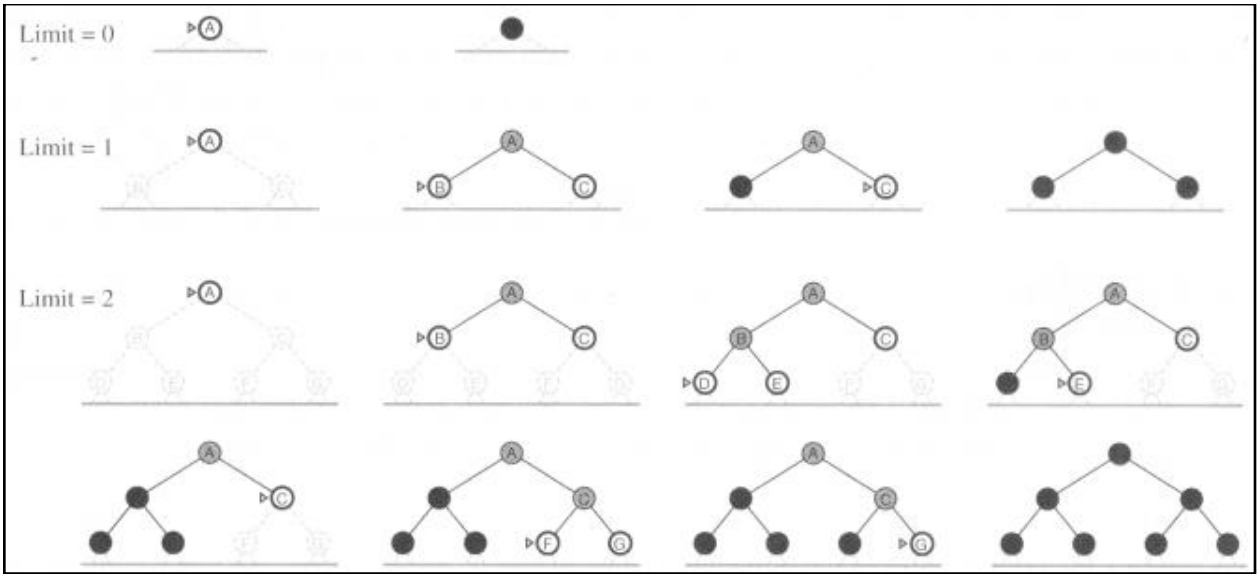
v) Exhaustive searches - Iterative deepening depth first search

Iterative deepening search, also referred to as iterative-deepening-depth-first search, is a widely used approach that is often paired with depth-first search to identify an optimal depth limit. This method gradually elevates the limit—starting from 0, then progressing to 1, followed by 2, and so forth—until a goal is successfully located. This termination takes place once the depth limit aligns with 'd,' representing the depth of the shallowest goal node. The algorithm's schematic is depicted in the figure.

Iterative deepening search amalgamates the advantages of both depth-first and breadth-first search techniques. Similar to depth-first search, it maintains a modest memory requirement, specifically O(bd). Simultaneously, akin to breadth-first search, it guarantees completeness under the condition of finite branching factors and optimality when the path cost corresponds to a non-decreasing function of the node's depth.

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

Figure 18: Iterative deepening depth first search



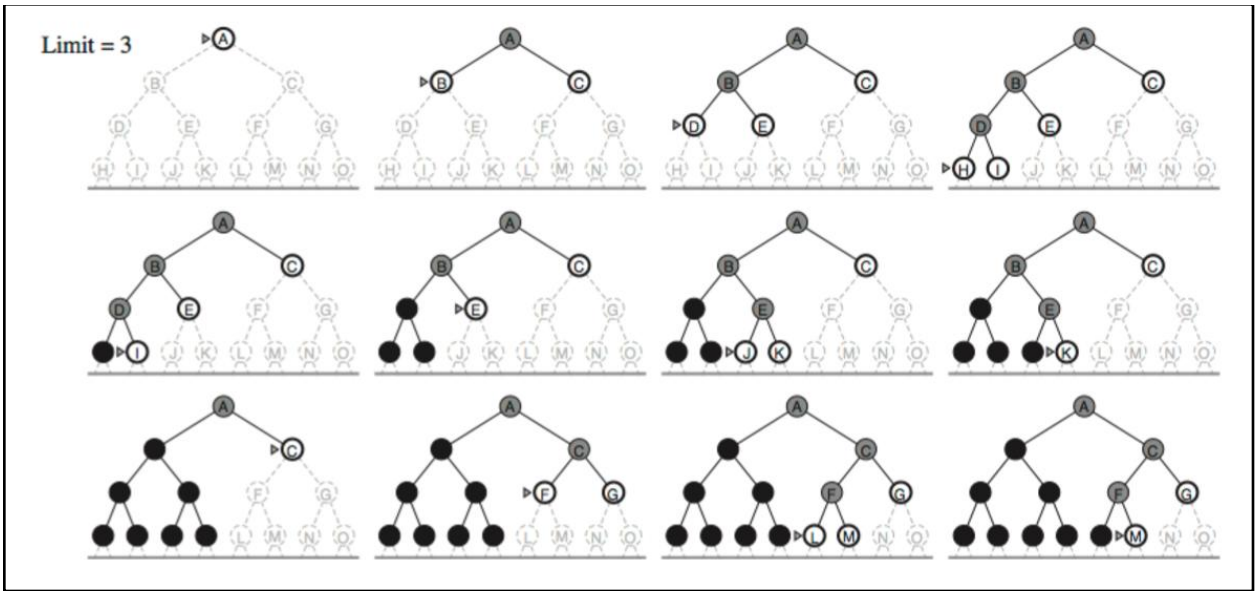


Figure 19: **Iterative deepening depth first search Illustration:**

Properties:

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $\mathbf{O(b^d)}$.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

vi) Bidirectional search:

- The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ much less than, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

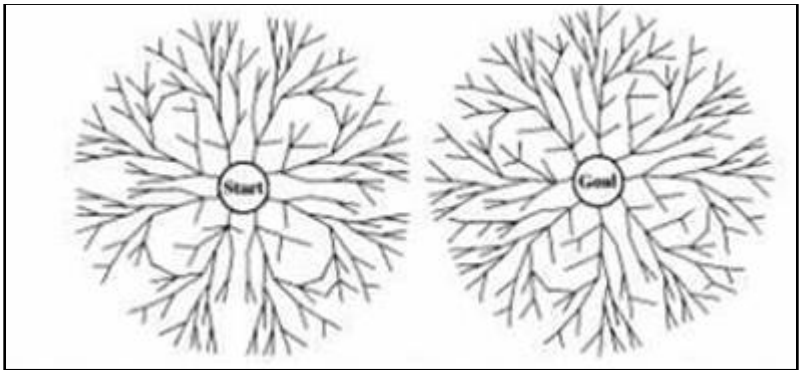


Figure:20 A schematic view of bidirectional search

Before moving into bidirectional search let's first understand a few terms.

• Forward Search: Looking in-front of the end from start.
• Backward Search: Looking from end to the start back-wards.

• So Bidirectional Search as the name suggests is a combination of forwarding and backward search. Basically, if the average branching factor going out of node / fan-out, if fan-out is less, prefer forward search. Else if the average branching factor is going into a node/fan in is less (i.e. fan-out is more), prefer backward search.

• We must traverse the tree from the start node and the goal node and wherever they meet the path from the start node to the goal through the intersection is the optimal solution. The BS Algorithm is applicable when generating predecessors is easy in both forward and backward directions and there exist only 1 or fewer goal states.

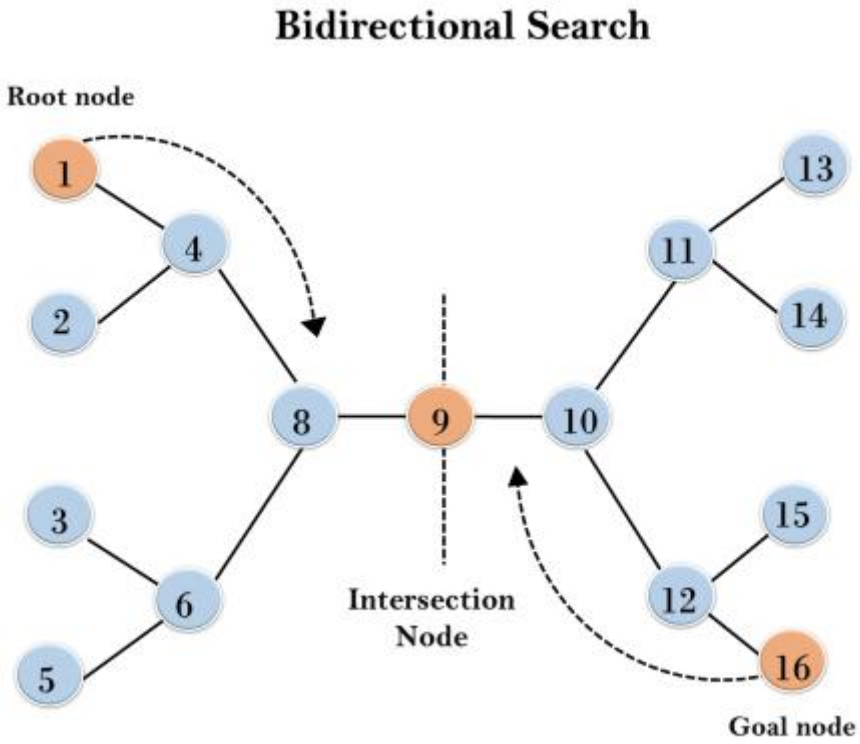


Figure21 : Bidirectional search illustration

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

COMPARING UNINFORMED SEARCH STRATEGIES

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

Figurre 22:The evaluation of search strategies. B is the branching factor; d is the depth of the shallowet solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= Ɛ for positive Ɛ; [c] optimal if step costs are all identical; [d] if both directions use breadth- first search.

13.2 INFORMED (HEURISTIC) SEARCH STRATEGIES

The informed search strategy is one that uses problem- specific knowledge beyond the definition of the problem itself. And it can find solutions more efficiently than an uninformed strategy.

Best-first search: Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** f(n). The node with lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal. This can be implemented using a priority-queue, a data structure that will maintain the fringe in ascending order of f-values.

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

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

i) Greedy Best-first search

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

function:

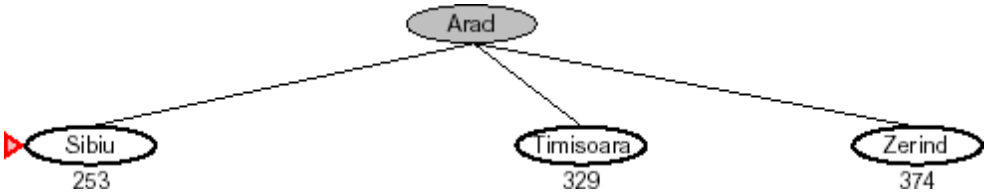
Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in **Figure**. For example, the initial state is In (Arad), and the straight line distance heuristic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

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

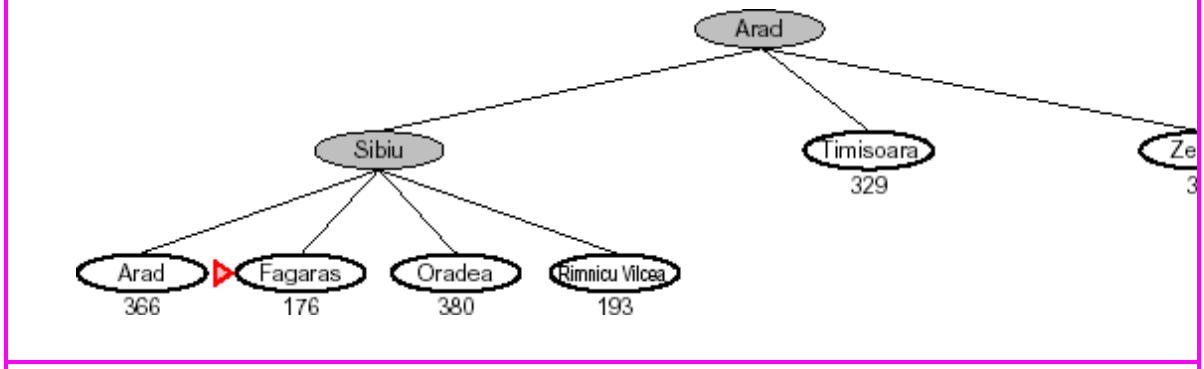
Figure: Values of h_{SLD}-straight-line distances to B u c h a r e s t.

The Initial State

After Expanding Arad



After Expanding Sibiu



After Expanding Fagaras

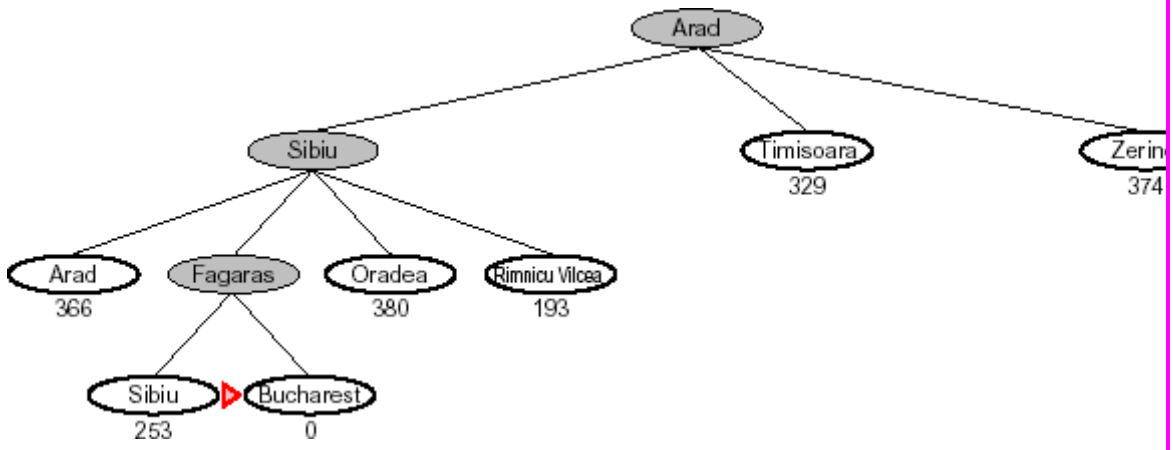


Figure 23 : Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Properties of greedy search to Complete:

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

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

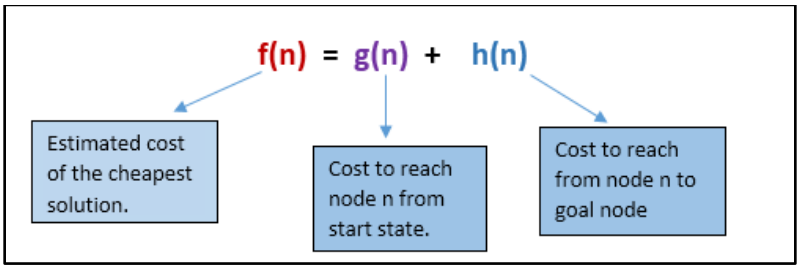
ii) A* SEARCH

A* Search is the most widely used form of best-first search. The evaluation function f(n) is

obtained by combining

(1) g(n) = the cost to reach the node, and

(2) h(n) = the cost to get from the node to the goal : f(n) = g(n) + h(n).



A* Search is both optimal and complete. A* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance hSLD. It cannot be an overestimate. A* Search is optimal if h(n) is an admissible heuristic – that is, provided that h(n) never overestimates the cost to reach the goal. An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest. The progress of an A* tree search for Bucharest is shown in Figure The values of 'g ' are computed from the step costs shown in the Romania map(figure). Also the values of hSLD are given in Figure

Algorithm:

Algorithm:

1. Initialize	**:**	**Set OPEN = (S); CLOSE D = ( ) g(s)= 0, f(s)=h(s)**
2. Fail	**:**	**If OPEN = ( ), Terminate and fail.**
3. Select	**:**	**select the minimum cost state, n, from OPEN, save n in CLOSED**
4. Terminate	**:**	**If n €G, Terminate with success and return f(n)**

5. Expand : for each successor, m, of n

a) If m € *OPEN U CLOSED+ Set g(m) = g(n) + c(n , m) Set f(m) = g(m) + h(m)

Insert m in OPEN

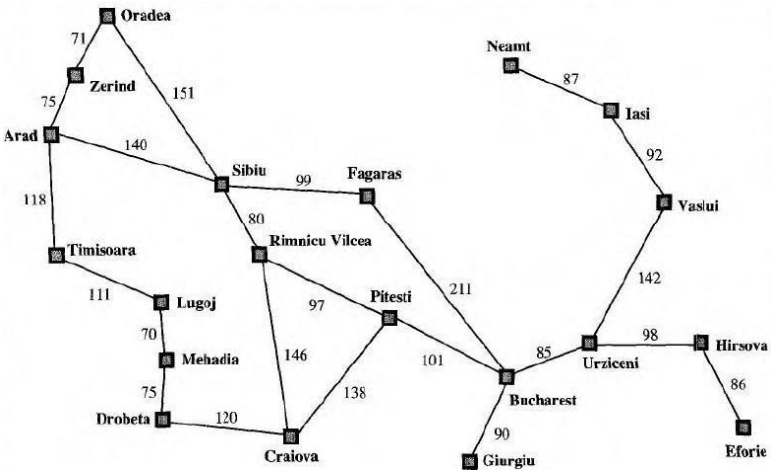
b) If m € *OPEN U CLOSED+

Set g(m) = min { g(m) , g(n) + c(n , m)} Set f(m) = g(m) + h(m)

If f(m) has decreased and m € CLOSED Move m to OPEN.

A* Illustration:

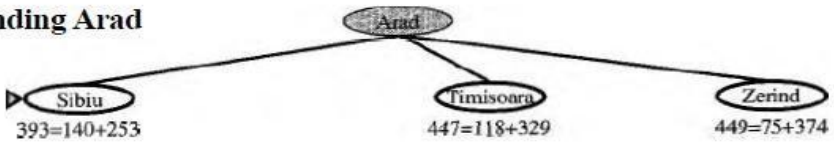
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



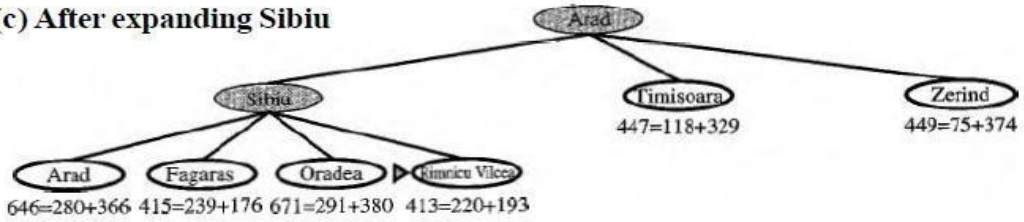
(a) The initial state

366=0+366

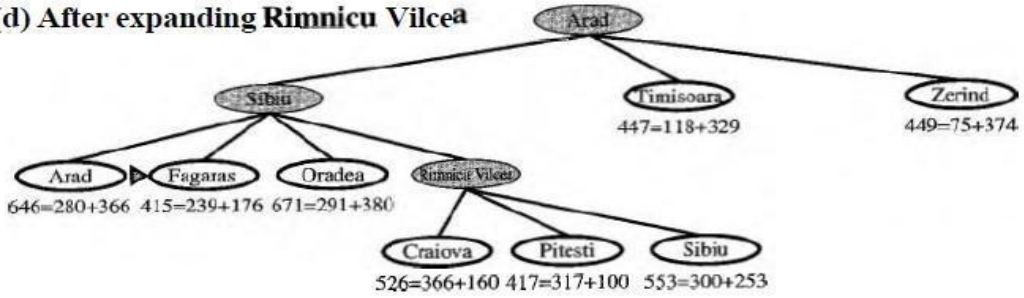
(b) After expanding Arad



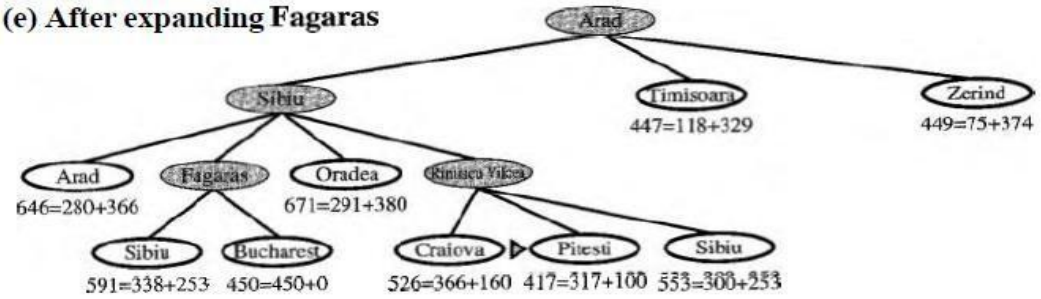
(c) After expanding Sibiu



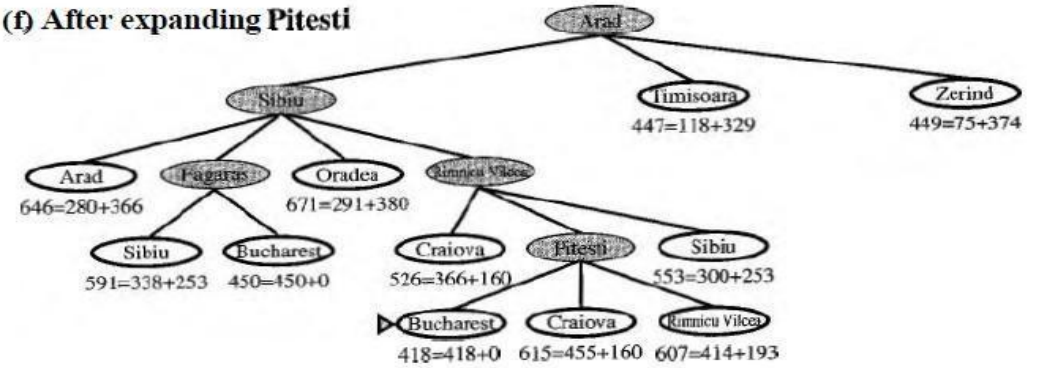
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



A* search properties:

- The algorithm A* is admissible. This means that provided a solution exists, the

first solution found by A* is an optimal solution. A* is admissible under the following conditions:

- Heuristic function: for every node n , $h(n) \leq h^*(n)$.
- A* is also complete.
- A* is optimally efficient for a given heuristic.
- A* is much more efficient that uninformed search.

UNIT 1: QUESTION BANK

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

S.No	Questions	BT Level	Competence
1	Define Artificial intelligence(AI)? Explain the techniques of AI and describe the characteristics of AI.	**BT2**	**Understand**
2	Examine the PEAS specification of the task environment of an agent?	**BT4**	**Analyse**
3	Summarize the following uninformed search i)Depth First Search ii)Iterative Deepening Depth First Search. iii) Bidirectional Search.	**BT2**	
4	Identify and discuss any two informed search methods with examples.		
5	Compare and contrast uninformed search strategies (e.g., breadth-first, depth-first) with informed search strategies (e.g., A* search) in terms of their efficiency and optimality.	**BT4**	
6	Evaluate the performance measure of various informed search algorithms?	**BT5**	
7	What is task environment and its characteristics? And indentify the task environment for the following problems i) Travellingsaleman problem ii) 8-puzzle iii) Twoers of Hanoi iv) Chess	**BT1**	
8	What is agent programs and agent function? Illustrate with example?	**BT3**	

9	Explain in details about the various agents with schematic diagram or pseudocode?	**BT2**	
10	Define Heuristics. Demonstrate the significance of heuristic function in the informed search with suitable example.	**BT2**	
11	Consider the given problem. Formulate the operator involved in it. Consider the water jug problem: You are given two jugs, a 4gallon one and 3-gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water from the 4gallon jug? Explicit Assumptions: A jug can be filled from the pump, water can be poured out of a jug on to the ground, water can be poured from one jug to another and that there are no other measuring devices available.	**BT6**	

Artificial Intelligence

Unit-2

- **Beyond Classical Search:** Hill-climbing search, simulated annealing search, Local Search in Continuous Spaces, Searching with Non-Deterministic Actions, Searching with Partial Observations, Online Search Agents and Unknown Environment.
- **Constraint Satisfaction Problems:** Defining Constraint Satisfaction Problems, Constraint Propagation, Backtracking Search for CSPs, Local Search for CSPs, the Structure of Problems.

2.1) Beyond Classical Search

Local Search

2.1.1: Hill Climbing Search

Hill climbing is a simple optimization algorithm used in Artificial Intelligence (AI) to find the best possible solution for a given problem. It belongs to the family of local search algorithms and is often used in optimization problems where the goal is to find the best solution from a set of possible solutions.

- In Hill Climbing, the algorithm starts with an initial solution and then iteratively makes small changes to it in order to improve the solution. These changes are based on a heuristic function that evaluates the quality of the solution. The algorithm continues to make these small changes until it reaches a local maximum, meaning that no further improvement can be made with the current set of moves.
- There are several variations of Hill Climbing, including steepest ascent Hill Climbing, first-choice Hill Climbing, and simulated annealing. In steepest ascent Hill Climbing, the algorithm evaluates all the possible moves from the current solution and selects the one that leads to the best improvement. In first-choice Hill Climbing, the algorithm randomly selects a move and accepts it if it leads to an improvement, regardless of whether it is the best move. Simulated annealing is a probabilistic variation of Hill Climbing that allows the algorithm to occasionally accept worse moves in order to avoid getting stuck in local maxima.

Hill Climbing can be useful in a variety of optimization problems, such as scheduling, route planning, and resource allocation. However, it has some limitations, such as the tendency to get stuck in local maxima and the lack of diversity in the search space. Therefore, it is often combined with other optimization techniques, such as genetic algorithms or simulated annealing, to overcome these limitations and improve the search results.

```
function HILL-CLIMBING( problem) return a state that is a local maximum
        input: problem, a problem
 local variables: current, a
                          node.
            neighbor, a node.

        current ←MAKE-NODE(INITIAL-STATE[problem])
        loop do
                neighbor ← a highest valued successor of current
                if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
                current ←neighbor
```

Advantages of Hill Climbing algorithm:

1. Hill Climbing is a simple and intuitive algorithm that is easy to understand and implement.
2. It can be used in a wide variety of optimization problems, including those with a large search space and complex constraints.
3. Hill Climbing is often very efficient in finding local optima, making it a good choice for problems where a good solution is needed quickly.
4. The algorithm can be easily modified and extended to include additional heuristics or constraints.

Disadvantages of Hill Climbing algorithm:

1. Hill Climbing can get stuck in local optima, meaning that it may not find the global optimum of the problem.
2. The algorithm is sensitive to the choice of initial solution, and a poor initial solution may result in a poor final solution.
3. Hill Climbing does not explore the search space very thoroughly, which can limit its ability to find better solutions.
4. It may be less effective than other optimization algorithms, such as genetic algorithms or simulated annealing, for certain types of problems.

Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

- In the above definition, mathematical optimization problems imply that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by the salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in a reasonable time.

- A heuristic function is a function that will rank all the possible alternatives at any branching step in the search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

Features of Hill Climbing
1. Variant of generating and test algorithm:
It is a variant of generating and testing algorithms. The generate and test algorithm is as follows :

- Generate possible solutions.
- Test to see if this is the expected solution.
- If the solution has been found quit else go to step 1.

Hence we call Hill climbing a variant of generating and test algorithm as it takes the feedback from the test procedure. Then this feedback is utilized by the generator in deciding the next move in the search space.

2. Uses the Greedy approach:
At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.

2.1.1.1: Types of Hill Climbing and Algorithms

1. Simple Hill climbing:
It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as the next node.

Algorithm for Simple Hill climbing :

- Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state as the current state.
- Loop until the solution state is found or there are no new operators present which can be applied to the current state.
    - Select a state that has not been yet applied to the current state and apply it to produce a new state.
    - Perform these to evaluate the new state.
        - If the current state is a goal state, then stop and return success.
        - If it is better than the current state, then make it the current state and proceed further.
        - If it is not better than the current state, then continue in the loop until a solution is found.
- Exit from the function.

2. Steepest-Ascent Hill climbing:
It first examines all the neighboring nodes and then selects the node closest to the solution state as of the next node.

Algorithm for Steepest Ascent Hill climbing :

- Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state as the current state.
- Repeat these steps until a solution is found or the current state does not change
    - Select a state that has not been yet applied to the current state.
    - Initialize a new 'best state' equal to the current state and apply it to produce a new state.
    - Perform these to evaluate the new state
        - If the current state is a goal state, then stop and return success.
        - If it is better than the best state, then make it the best state else continue the loop with another new state.
    - Make the best state as the current state and go to Step 2 of the second point.
- Exit from the function.

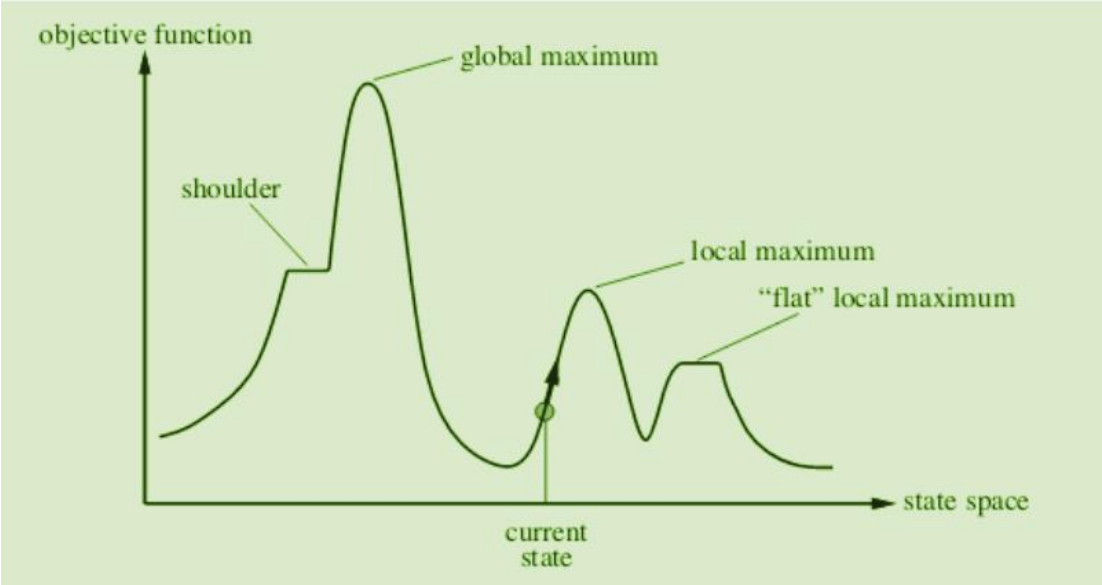
3. Stochastic hill climbing:
It does not examine all the neighboring nodes before deciding which node to select. It just selects a neighboring node at random and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

- Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state the current state.
- Repeat these steps until a solution is found or the current state does not change.
    - Select a state that has not been yet applied to the current state.
    - Apply the successor function to the current state and generate all the neighbor states.
    - Among the generated neighbor states which are better than the current state choose a state randomly (or based on some probability function).
    - If the chosen state is the goal state, then return success, else make it the current state and repeat step 2 of the second point.
- Exit from the function.

2.1.1.2 :State Space diagram for Hill Climbing
The state-space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).

- X-axis: denotes the state space ie states or configuration our algorithm may reach.
- Y-axis: denotes the values of objective function corresponding to a particular state.

The best solution will be a state space where the objective function has a maximum value(global maximum).



Different regions in the State Space Diagram:

- Local maximum: It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here the value of the objective function is higher than its neighbors.

- Global maximum: It is the best possible state in the state space diagram. This is because, at this stage, the objective function has the highest value.
- Plateau/flat local maximum: It is a flat region of state space where neighboring states have the same value.
- Ridge: It is a region that is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
- Current state: The region of the state space diagram where we are currently present during the search.
- Shoulder: It is a plateau that has an uphill edge.

2.1.1.3 :Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

- **Local maximum:** At a local maximum all neighboring states have a value that is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.

  To overcome the local maximum problem: Utilize the backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
- **Plateau:** On the plateau, all neighbors have the same value. Hence, it is not possible to select the best direction .
- To overcome plateaus: Make a big jump. Randomly select a state far away from the current state. Chances are that we will land in a non-plateau region.

- **Ridge:** Any point on a ridge can look like a peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.

  To overcome Ridge: In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

2.1.2 SIMULATED ANNEALING

2.1.2.1 : The Concept

SA algorithm is one of the most preferred heuristic methods for solving the optimization problems. Kirkpatrick et al. introduced SA by inspiring the annealing procedure of the metal working . Annealing procedure defines the optimal molecular arrangements of metal particles where the potential energy of the mass is minimized and refers cooling the metals gradually after subjected to high heat. In general manner, SA algorithm adopts an iterative movement according to the variable temperature parameter which imitates the annealing transaction of the metals .

A simple optimization algorithm compares iteratively the outputs of the objective functions running with current and neighboring point in the domain so that, if the neighboring point generates better result than the current one, then it is saved as base solution for the next iteration. Otherwise, the algorithm terminates the procedure without searching the wider domain for better results. So that, the algorithm is prone to be getting trapped in local minima or maxima. Instead, SA algorithm proposes an effective solution to this problem as incorporating two iterative loops which are the cooling procedure for the annealing process and Metropolis criterion . Basic idea behind the Metropolis criterion is to be executed randomly to extra search the neighborhood of the candidate solution to avoid being trapped to local extreme points.

To be more precise, let us consider the minimization problem and define an objective function f(xi) corresponding to the argument set of $xi=\{x1,x2,\cdots,xn\}$ with $n\in R$.

Therefore, if $f(xi+1)<f(xi)$, then take $xi+1$ as a new candidate extreme point to check.

Otherwise, define $w=\exp[-\{f(xi+1)-f(xi)\}/Tc]$

where Tc is the current temperature parameter and generate a random number s, such that $0<s<1$. Then, if the relation of $w>s$ is true, you also accept the $xi+1$ as a new candidate, else reject and go back to previous step and generate another s. Hence, Metropolis criterion allows for the motion of the current step to a certain extent even the objective function's trajectory is getting convergent through the potential local minimum point.

On the other hand, Metropolis algorithm proposes solution for a constant temperature. So that, for larger values of Tc, the algorithm requires wider search area. Therefore, this raises the probability for reaching the global minimum. Besides that, as the iterative motion of algorithm is

initialized randomly, it may skips the global minima or adopts non precision approach through the extreme point. On the contrary, for smaller values of Tc, it requires smaller search area and this case may give rise of being trapped in local minima. For removing this handicap, SA offers an iterative solution as incorporating nested loops for changing the temperature parameter and the solution point. The motion of SA algorithm begins with a larger value of temperature to execute the iteration for inner loop and immediately, the point leads the best objective function in the current inner loop is assigned as new candidate solution. Then, the outer loop is run by incrementing the temperature and updating the starting point. This iterative process continues until reaching the lowest limit of temperature or realizing the predetermined number of iterations.

2.1.2.2: The Algorithm / Pesudocode

Here are the steps of the algorithm:

1. Initialization: Start with an initial solution to the problem. This can be a random solution or a solution obtained using some method.

2. Define parameters: Set the initial temperature and cooling schedule. Typically, initial temperature T_0=100T0=100 and cooling are done by formula $Tnew=\alpha Tcurrent$, where $\alpha\in[0.7,0.99]$ is a cooling factor. Set number of iterations per temperature: typically, this number is between 100100 and 10001000.

3. Iterative process: Perform iterations until a stopping criterion is met. This can be a maximum number of iterations or reaching a final temperature.

4. Generate a neighboring solution: Modify the current solution to generate a neighboring solution. The modification can be swapping variables or changing values.

5. Evaluate the neighboring solution: Calculate the cost or objective function value for the neighboring solution, indicating its quality. The cost value f(S_{neighbor})should be evaluated for neighbor solution S_{neighbor} .

6. Acceptance criterion: Determine whether to accept the neighboring solution as the new current solution. If the neighboring solution is better than the current solution, accept it:

S_{new}= S_{neighbor} if f(S_{current}) > f(S_{neighbor}).

As in this case, Cost Minimization problem is assumed .Thus , here the neighbor is considered to be better if the cost function of neighbor is lower (f(S_{current}) > f(S_{neighbor}) and considered to be worse if the cost function of neighbor is higher (f(S_{current}) < f(S_{neighbor})

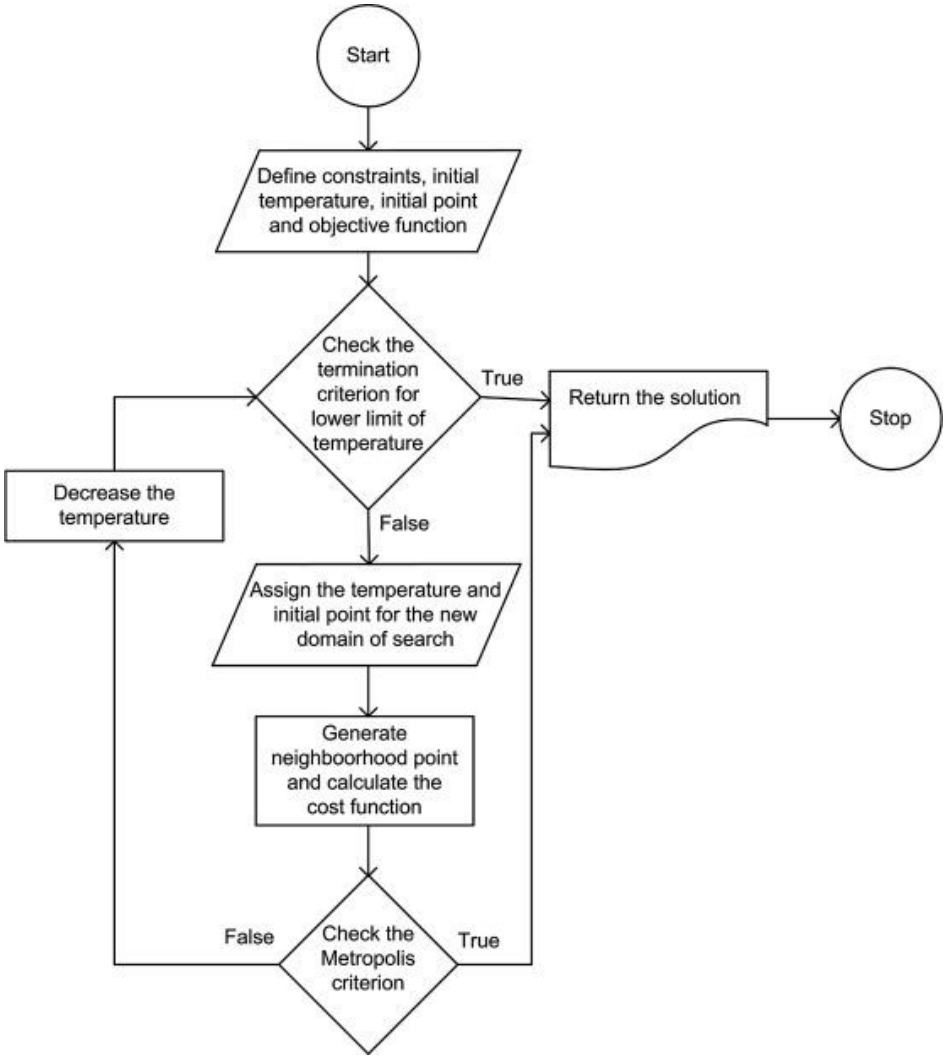
If the neighboring solution is worse, accept it with a probability determined by the acceptance probability formula: $Snew=Sneighbor$

if P= $\exp\{-(f(Sneighbor)-f(Scurrent))\}>r$, where $r\in[0,1]$ is some random number.

The neighbor will be chosen with this probability P.

7. Update the temperature: Adjust the temperature according to the cooling schedule: *Tnew* $=\alpha Tcurrent$. T (new)= T (current)/{(No of iterations)+1}
8. Repeat: Go back to step four and continue the iterative process.
9. Termination: Stop the algorithm when the stopping criterion is met.
10. Output: Return the best solution (with the lowest cost value) obtained during the iterations. This solution approximates the global optimum solution.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps
  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```



Flowchart of SA technique.

Note that, it is very critical to choose the temperature changing steps in the outer loop and randomly step sizes of inner loop. If one chooses Tc very large, then w>r is satisfied at any iteration so that the algorithm will be terminated most likely at a random minimum in the domain of the function to be optimized. Besides, if one chooses Tc very small the condition w>s never be satisfied and the motion will be terminated at the first minima. Moreover, selecting rating of temperature change has a crucial effect on the quality of the solution . For preventing of those handicaps, SA algorithm deals with the optimization problem with large value of temperature and travels a large number of points in the domain of the objective function. For the further steps, it decreases the temperature gradually and narrows the search sector by adopting the local minimum point at the previous inner loop. By this way, it eliminates the local minimum points from the search space, and also, it converges to the global minimum point sensitively. Consequently, it can be commented that SA is very preferable technique among the other heuristic approaches as providing practical randomness into the search to avoid the local extreme

points. On the other hand, SA contains a trade-off between computational time and the sensitivity of the solution.

2.1.3 Local Search in Continuous Space :

A continuous action space has an infinite branching factor, and thus can't be handled by most of the algorithms what we discussed previously (with the exception of first-choice hill climbing and simulated annealing).

One way to deal with a continuous state space is to discretize it. Instead of allowing any point in continuous two-dimensional space, we could limit them to fixed points with spacing of size.We can then apply any of our local search algorithms to this discrete space. The states are defined by n variables (defined by an n-dimensional vector of variables **x**.)

Often we have an objective function expressed in a mathematical form such that we can use calculus to solve the problem analytically rather than empirically. Many methods attempt to use the gradient of the landscape to find amaximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope.

$$\frac{\partial f}{\partial x_i}$$

Can perform steepest-ascent hill climbing by updating the current state according to the formula $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$, αis the step size (a small constant). If the objection function *f* is not available in a differentiable form, use empirical gradient search.

- Often resists a closed-form solution
  - ✓ Fake up an empirical gradient
  - ✓ Amounts to greedy hill climbing in discretized statespace
- Can employ Newton-Raphson Method to findmaxima.
- Continuous problems have similar problems: plateaus, ridges, local maxima,etc.

Line search: tries to overcome the dilemma of adjusting α by extending the current gradient direction—repeatedly doubling αuntil f starts to decrease again. The point at which this occurs becomes the new current state.

Newton-Raphson method: Another most effective algorithm is the venerable **Newton-Raphson method.** This is a general technique for finding roots of functions- that is, solving equations of the form g(x)=0. It works by computing a new estimate for the root x according to newton's formula

$\boldsymbol{x \leftarrow x - g(x)/g'(x)}$**.**

To find a maximum or minimum of *f*,we need to find **x** such that the gradient is a zero vector (i.e.,$\nabla f(x) = 0$). Thus, g(x)is Newton's formula becomes .,$\nabla f(x)$, and the update equation can be written in matrix-vector form as

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla \mathbf{f}(\mathbf{x}) .$$

$\mathbf{H}_f(\mathbf{x})$ is the Hessian matrix of second derivatives, $H_{ij} = \partial^2 f / \partial x_i \partial x_j$

Constrained optimization: An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. (e.g. : linear programming problems).

Many local search methods apply also to problems in continuous spaces. **Linear programming** and **convex optimization problems** obey certain restrictions on the shape of the state space and the nature of the objective function, and admit polynomial-time algorithms that are oftenly extremely efficient in practice.

2.1.4 Searching with nondeterministic actions

- When the environment is either partially observable or nondeterministic (or both), the future percepts cannot be determined in advance, and the agent's future actions will depend on those future percepts.
- In partially observable and nondeterministic environments, the solution to a problem is no longer a sequence, but rather a conditional plan (sometimes called a contingency plan or a strategy).

Nondeterministic problems:

Transition model is defined by RESULTS function that returns a set of possible outcome states; Instead of defining the transition model by a RESULT function that returns a single outcome state, we use a RESULT function that returns a set of possible outcome states.

RESULT(1,SUCK)={5,7}

Solution is not a sequence but a contingency plan (strategy),

e.g. [Suck, if State = 5 then [Right, Suck] else []];

In nondeterministic environments, agents can apply AND-OR search to generate contingent plans that reach the goal regardless of which outcomes occur during execution.

AND-OR search trees

OR nodes: In a deterministic environment, the only branching is introduced by the agent's own choices in each state, we call these nodes OR nodes.

AND nodes: In a nondeterministic environment, branching is also introduced by the environment's choice of outcome for each action, we call these nodes AND nodes.

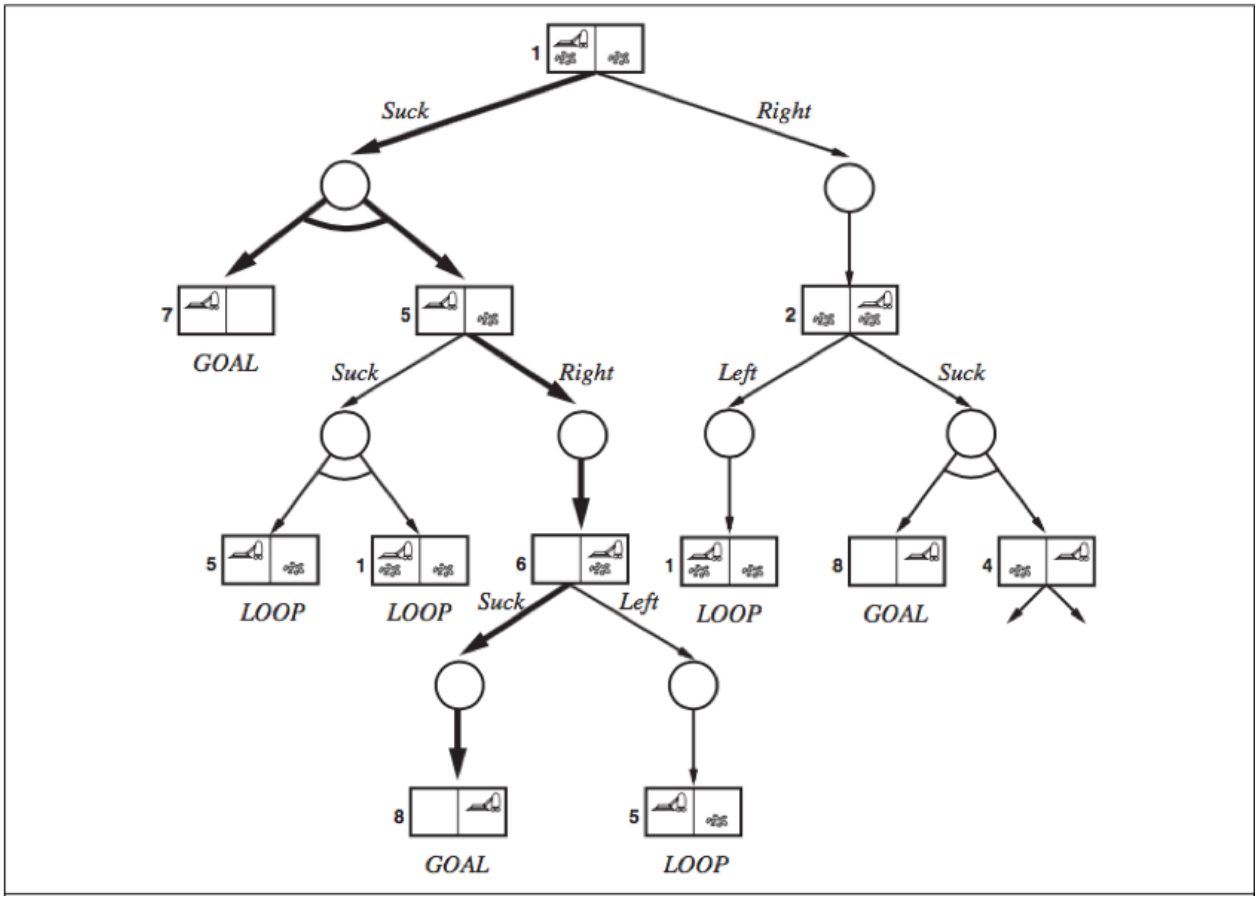
AND-OR tree: OR nodes and AND nodes alternate. States nodes are OR nodes where some action must be chosen. At the AND nodes (shown as circles), every outcome must be handled.

A solution (shown in bold lines) for an AND-OR search problem is a subtree that

1) has a goal node at every leaf;

2) specifies one action at each of its OR nodes;

3) includes every outcome branch at each of its AND nodes.



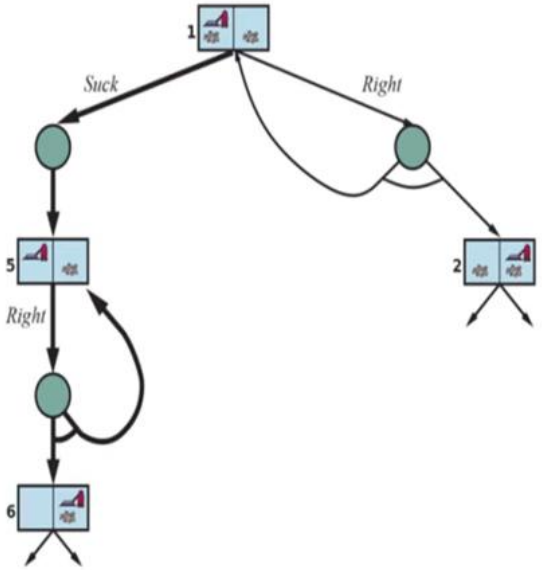
A recursive, depth-first algorithm for AND-OR graph search

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

Cyclic solution

Cyclic solution: keep trying until it works. We can express this solution by adding a label to denote some portion of the plan and using the label later instead of repeating the plan itself. E.g.: [Suck, L1: Right, if State = 5 then L1 else Suck]. (or "while State = 5 do Right").



2.1.5 Searching with partial observatioins

Belief state: The agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.

Standard search algorithms can be applied directly to belief-state space to solve sensorless problems, and belief-state AND-OR search can solve general partially observable problems. Incremental algorithms that construct solutions state-by-state within a belief state are often more efficient.

1. Searching with no observation

When the agent's percepts provide no information at all, we have a sensorless problem.

To solve sensorless problems, we search in the space of belief states rather than physical states. In belief-state space, the problem is fully observable, the solution is always a sequence of actions.

Belief-state problem can be defined by: (The underlying physical problem P is defined by $ACTIONS_P$, $RESULT_P$, $GOAL\text{-}TEST_P$ and $STEP\text{-}COST_P$)

·Belief states: Contains every possible set of physical states. If P has N states, the sensorless problem has up to 2^N states (although many may be unreachable from the initial state).

·Initial states: Typically the set of all states in P.

·Actions:

a. If illegal actions have no effect on the environment, take the union of all the actions in any of the physical states in the current belief b:

ACTIONS(b) = $\bigcup_{s \in b} RESULTS_P(s, a)$

b. If illegal actions are extremely dangerous, take the intersection.

·Transition model:

a. For deterministic actions,

b' = RESULT(b,a) = {s' : s' = $RESULT_P$(s,a) and s∈b}. (b' is never larger than b).

b. For nondeterministic actions,

b' = RESULT(b,a) = {s' : s' = $RESULT_P$(s,a) and s∈b} = (b' may be larger than b)

The process of generating the new belief state after the action is called the prediction step.

·Goal test: A belief state satisfies the goal only if all the physical states in it satisfy GOAL-TEST$_P$.

·Path cost

If an action sequence is a solution for a belief state b, it is also a solution for any subset of b. Hence, we can discard a path reaching the superset if the subset has already been generated. Conversely, if the superset has already been generated and found to be solvable, then any subset is guaranteed to be solvable.

Main difficulty: The size of each belief state.

Solution:

a. Represent the belief state by some more compact description;

b. Avoid the standard search algorithm, develop incremental belief state search algorithms instead.

Incremental belief-state search: Find one solution that works for all the states, typically able to detect failure quickly.

2. Searching with observations

When observations are partial, The ACTIONS, STEP-COST, and GOAL-TEST are constructed form the underlying physical problem just as for sensorless problems.

·Transition model: We can think of transitions from one belief state to the next for a particular action as occurring in 3 stages.

The prediction stage is the same as for sensorless problem, given the action a in belief state b, the predicted belief state is $\hat{b}$ = PREDICT(b,a);

The observation prediction stage determines the set of percepts o that could be observed in the predicted belief state:

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\};$$

The update stage determines, for each possible percept, the belief state that would result from the percept. The new belief state b_o is the set of states in $\hat{b}$ that could have produced the percept:

$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}$

(b_o can be no larger than the predicted belief state $\hat{b}$)

In conclusion:

RESULTS(b,a) = { b_o : b_o = UPDATE(PREDICT(b,a),o) and o∈POSSIBLE-PERCEPTS(PREDICT(b,a))}

Search algorithm return a conditional plan that test the belief state rather than the actual state.

Agent for partially observable environments is similar to the simple problem-solving agent (formulates a problem, calls a search algorithm, executes the solution).

Main difference:

1) The solution will be a conditional plan rather than a sequence.

2) The agent will need to maintain its belief state as it performs actions and receives percepts.

Given an initial state b, an action a, and a percept o, the new belief state is

b' = UPDATE(PREDICT(b, a), o). //recursive state estimator

Sate estimation: a.k.a. monitoring or filtering, a core function of intelligent system in partially observable environments—maintaining one's belief state.

2.1.6 Online search Agents

Online search is a necessary idea for unknown environments. Online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.

1. Online search problem

Assume a deterministic and fully observable environment, the agent only knows:

·ACTION(s): returns a list of actions allowed in state s;

·c(s, a, s'): The step-cost function, cannot be used until the agent knows that s' is the outcome;

·GOAL-TEST(s).

·The agent cannot determine RESULT(s, a) except by actually being in s and doing a.

·The agent might have access to an admissible heuristic function h(s) that estimates the distance from the current state to a goal state.

Competitive ratio: The cost (the total path cost of the path that the agent actually travels) / the actual shortest path (the path cost of the path the agent would follow if it knew the search space in advance). The competitive ratio is expected to be as small as possible.

In some case the best achievable competitive ratio is infinite, e.g. some actions are irreversible and might reach a dead-end state. No algorithm can avoid dead ends in all state spaces.

Safely explorable: some goal state is reachable from every reachable state. E.g. state spaces with reversible actions such as mazes and 8-puzzles.

No bounded competitive ratio can be guaranteed even in safely explorable environments if there are paths of unbounded cost.

2. Online search agents

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  if s is not null then
      result[s, a] ← s'
      add s to the front of unbacktracked[s']
  if untried[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

ONLINE-DFS-AGENT works only in state spaces where the actions are reversible.

RESULT: a table the agent stores its map, RESULT[s, a] records the state resulting from executing action a in state s.

Whenever an action from the current sate has not been explored, the agent tries that action.

When the agent has tried all the actions in a state, the agent in an online search backtracks physically (in a depth-first search, means going back to the state from which the agent most recently entered the current sate).

3. Online local search

Exploration problems arise when the agent has no idea about the satate and actions of its environment. For safely explorable environments, online search agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

Random walk: Because online hill-climbing search cannot use restart(because the agent cannot transport itself to a new state), can use random walk instead. A random walk simply selects at random one of the available actions from the current state, preference can be given to actions that have not yet been tried.

Basic idea of online agent: Random walk will eventually find a goal or complete its exploration if the space is finite, but can be very slow. A more effective approach is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited. H(s) starts out being the heuristic estimate h(s) and is updated as the agent gains experience in the state space.

If the agent is stuck in a flat local minimum, the agent will follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimate cost to reach the goal through a neighbor s' is the cost to get to s' plus the estimated cost to get to a goal from there, that is, c(s, a, s') + H(s').

LRTA*: learning real-time A*. It builds a map of the environment in the result table, update the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

Actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost (a.k.a.h(s)), this optimism under uncertainty encourages the agent to explore new, possible promising paths.

2.2 Constraint Satisfaction Problem (CSP)

Sometimes a problem is not embedded in a long set of action sequences but requires picking the best option from available choices. A good general-purpose problem solving technique is to list the constraints of a situation (either negative constraints, like limitations, or positive elements that you want in the final solution). Then pick the choice that satisfies most of the constraints.

Formally speaking, a **constraint satisfaction problem(orCSP**)is defined by

a set of variables,X1;X2;:::;Xn, and

a set of constraints, C1;C2; : : : ;Cm.

Domain D {D1,D2,……Dn}

Each variable Xi has an Nonempty domain Di of possible values. Each constraint Ci involves some subset of t variables and specifies the allowable combinations of values for

that subset. A state of the problem is defined by an assignment of values to some or all of the variables, {Xi = vi; Xj =vj ; : : :}

- An assignment that does not violate any constraints is called a **consistent or legal assignment.**
- A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a **complete assignment** that satisfies all the constraints.
- **Partial assignment** is one that leaves **some variables unassigned** and a **partial solution** is a partial assignment that is consistent.
- Some CSPs also require a solution that maximizes an objective function.

CSP FORMULATION

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

1. **Initial state**: the empty assignment fg, in which all variables are unassigned.
2. **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assignedvariables.
3. **Goal test**: the current assignment iscomplete.
4. **Path cost**: a constant cost for everystep

Examples:

The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a *convex* region.

Example: Crypt arithmetic puzzles.

- A cryptarithmetic problem. 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.
- The constraint hypergraph for the cryptarithmetic problem, showing the *ALLdiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C1, C2, and C3 represent the carry digits for the three columns from right to left.

Example: Cryptarithmetic problems

- Find numeric substitutions that make an equation hold:

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

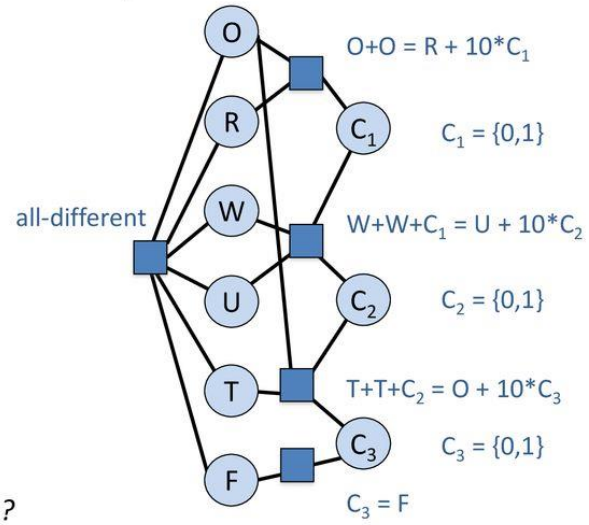
For example:

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

```
    7 3 4
+   7 3 4
-------
= 1 4 6 8
```

Note: not unique – how many solutions?

Non-pairwise CSP:



Example: The map coloring problem.

The task of coloring each region red, green or blue in such a way that no neighboring regions have the same color.

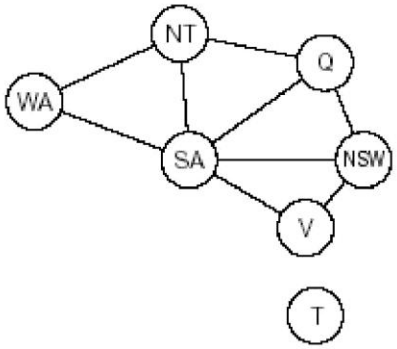
We are given the task of coloring each region red, green, or blue in such a way that the neighboring regions must not have the same color.

- To formulate this as CSP, we define the variable to be the regions: WA, NT, Q, NSW, V, SA, and T.
- The domain of each variable is the set {red, green, blue}.
- The constraints require neighboring regions to have distinct colors: for example, the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}. (The constraint can also be represented as the in equalityWA≠NT).
- There are many possible solutions, such as {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}.
- Map of Australia showing each of its states and territories

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

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

Constraint graph: nodes are variables, arcs show constraints



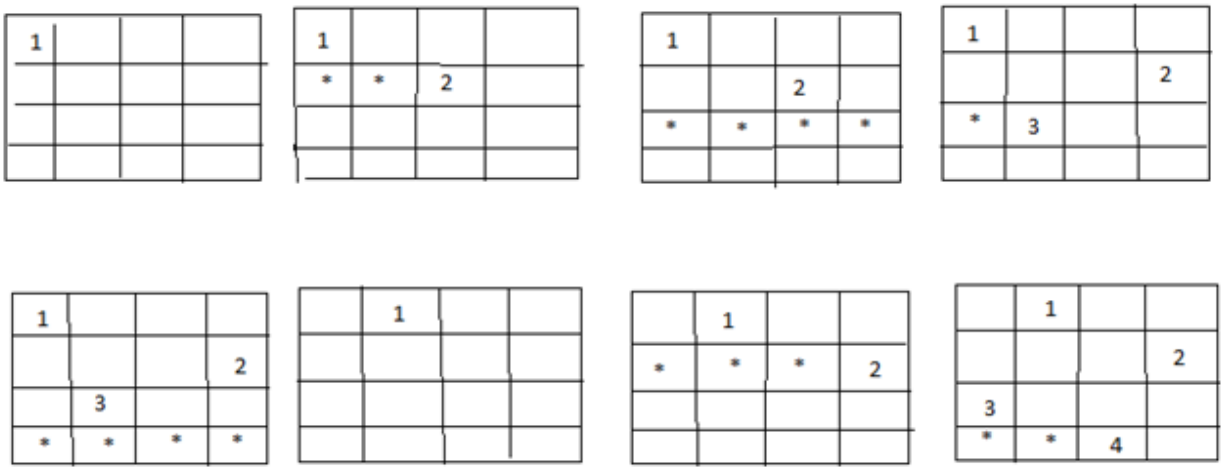
The map-coloring problem represented as a constraint graph. CSP can be viewed as a standard search problem as follows:

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

Example : **4- queens problem**

In **4- queens problem**, we have 4 queens to be placed on a 4*4 chessboard, satisfying the constraint that no two queens should be in the same row, same column, or in same diagonal.

1. Define the Variables:
    - Create four variables, one for each column of the chessboard. Each variable represents the row position of the queen in its respective column. Let's call these variables Q1, Q2, Q3, and Q4.
2. Define the Domain:
    - The domain of each variable should be the set {1, 2, 3, 4}, representing the four rows of the chessboard.
3. Define the Constraints:
    - Ensure that no two queens can share the same row by adding the constraint:
        - Q1 ≠ Q2, Q1 ≠ Q3, Q1 ≠ Q4, Q2 ≠ Q3, Q2 ≠ Q4, Q3 ≠ Q4
    4.
        - Ensure that no two queens can share the same column:
            - This constraint is automatically satisfied because we have distinct variables for each column.
        - Ensure that no two queens can share the same diagonal:
            - Add constraints to handle the diagonal threats:
                1. |Q1 - Q2| ≠ 1
                2. |Q1 - Q3| ≠ 2
                3. |Q1 - Q4| ≠ 3
                4. |Q2 - Q3| ≠ 1
                5. |Q2 - Q4| ≠ 2
                6. |Q3 - Q4| ≠ 1
    5. Search for a Solution:
        - You can use a CSP solver, such as backtracking or constraint propagation, to search for a solution to this CSP problem.



Now, we have successfully placed four queens on the chessboard without any of them threatening each other. This is a valid solution to the 4-Queens problem using CSP. You can explore different combinations to find all possible solutions or use a CSP solver to automate the process for larger N-Queens problems.

Example Job Scheduling Problem

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

Next, we represent precedence constraints between individual tasks. Whenever a task T1 must occur before task T2, and task T1 takes duration D1 to complete, we add an arithmetic constraint of the form

$$T_1 + d_1 \leq T_2.$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF};$$
$$Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}.$$

Next we say that for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF};$$
$$Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF};$$
$$Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB};$$
$$Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB}.$$

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a disjunctive constraint to say that *AxleF* and *AxleB* must not overlap in time; either one comes first or the other does:

$$(Axle_F + 10 \leq Axle_B) \quad \text{or} \quad (Axle_B + 10 \leq Axle_F).$$

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except ***Inspect*** we add a constraint of the form

$$X + d_X \leq Inspect.$$

Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

D_i={0,1,2,3,...,30}.

This particular problem is trivial to solve, but CSPs have been successfully applied to jobshop scheduling problems like this with thousands of variables.

Variation of CPS

Discrete variables :

- Finite domains: .⇒ size d O(dn) complete assignments E.g. Boolean CSPs, include. Boolean satisfiability (NP-complete).
- Infinite domains (integers, strings, etc.)
    - E.g. job scheduling, variables are start/end days for each job
    - Need a constraint language e.g $StartJob_1 + 5 \leq StartJob_3$.
    - Linear constraints solvable, nonlinear undecidable.

Continuous variables:
e.g. start/end times for Hubble Telescope observations.
Linear constraints solvable in poly time by LP methods.

Varieties of constraints:

- **Unary constraints** involve a single variable. green≠e.g. SA
- **Binary constraints** involve pairs of variables. WA≠e.g. SA
- **Higher-order constraints** involve 3 or more variables. e.g. cryptharithmetic column constraints.
- **Preference (soft constraints)** e.g. red is better than green often representable by a cost for each variable assignment →constrained optimization problems.

CONSTRAINT PROPOGATION

constraint propagation: Using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
local consistency: If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph.
There are different types of local consistency:

Node consistency

A single variable (a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraint.
We say that a network is node-consistent if every variable in the network is node-consistent.

Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints.
X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j).
A network is arc-consistent if every variable is arc-consistent with every other variable.
Arc consistency tightens down the domains (unary constraint) using the arcs (binary constraints).

Path consistency

Path consistency: A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraint on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.
Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

K-consistency

K-consistency: A CSP is k-consistent if, for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.
1-consistency = node consistency; 2-consisency = arc consistency; 3-consistensy = path consistency.
A CSP is **strongly k-consistent** if it is k-consistent and is also (k - 1)-consistent, (k – 2)-consistent, … all the way down to 1-consistent.

A CSP with n nodes and make it strongly n-consistent, we are guaranteed to find a solution in time $O(n^2d)$. But algorithm for establishing n-consitentcy must take time exponential in n in the worse case, also requires space that is exponential in n.

Global constraints

A global constraint is one involving an arbitrary number of variables (but not necessarily all variables).

Backtracking search for CSPs

Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.

Commutativity: CSPs are all commutative. A problem is commutative if the order of application of any given set of actions has no effect on the outcome.

Backtracking search: A depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

There is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

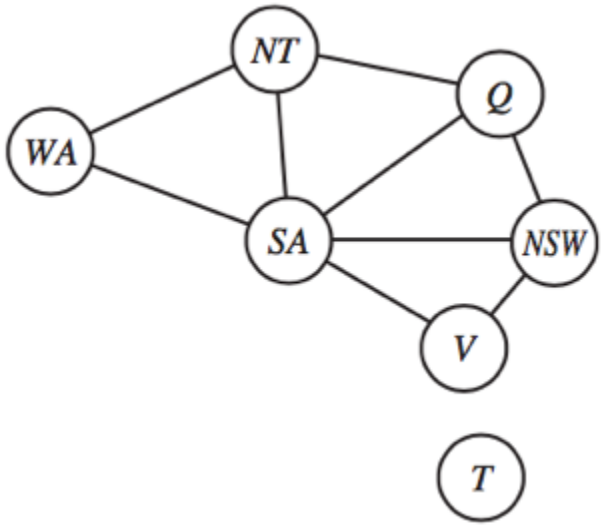
BACKTRACKING-SARCH keeps only a single representation of a state and alters that representation rather than creating a new ones.

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

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction probl gorithm is modeled on the recursive depth-first search of Chapter 3. By varying SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can in general-purpose heuristics discussed in the text. The function INFERENCE can used to impose arc-, path-, or *k*-consistency, as desired. If a value choice lea (noticed either by INFERENCE or by BACKTRACK), then value assignments (inc made by INFERENCE) are removed from the current assignment and a new valu

To solve CSPs efficiently without domain-specific knowledge, address following questions:
1)function SELECT-UNASSIGNED-VARIABLE: which variable should be assigned next?
function ORDER-DOMAIN-VALUES: in what order should its values be tried?
2)function INFERENCE: what inferences should be performed at each step in the search?
3)When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

1. Variable and value ordering



SELECT-UNASSIGNED-VARIABLE

Variable selection—fail-first

Minimum-remaining-values (MRV) heuristic: The idea of choosing the variable with the fewest "legal" value. A.k.a. "most constrained variable" or "fail-first" heuristic, it picks a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.

E.g. After the assignment for WA=red and NT=green, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q.

[Powerful guide]

Degree heuristic: The degree heuristic attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. **[useful tie-breaker]**

e.g. SA is the variable with highest degree 5; the other variables have degree 2 or 3; T has degree 0.

ORDER-DOMAIN-VALUES

Value selection—fail-last

If we are trying to find all the solution to a problem (not just the first one), then the ordering does not matter.

Least-constraining-value heuristic: prefers the value that rules out the fewest choice for the neighboring variables in the constraint graph. **(Try to leave the maximum flexibility for subsequent variable assignments.)**

e.g. We have generated the partial assignment with WA=red and NT=green and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA, therefore prefers red to blue.

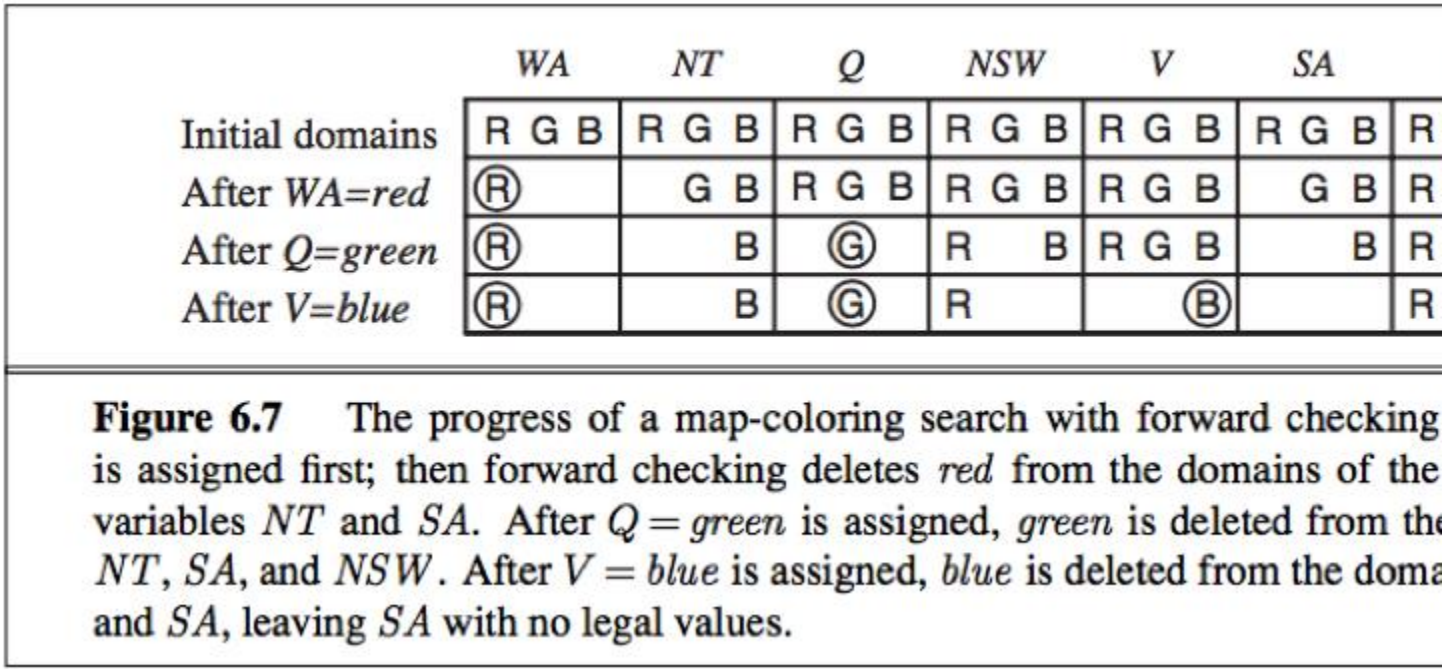
The **minimum-remaining-values** and **degree** heuristic are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining-value** heuristic helps in deciding which value to try first for a given variable.

2. Interleaving search and inference

INFERENCE

forward checking: [One of the simplest forms of inference.] Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

There is no reason to do forward checking if we have already done arc consistency as a preprocessing step.



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

Figure 6.7 The progress of a map-coloring search with forward checking is assigned first; then forward checking deletes *red* from the domains of the variables *NT* and *SA*. After $Q = green$ is assigned, *green* is deleted from the *NT*, *SA*, and *NSW*. After $V = blue$ is assigned, *blue* is deleted from the doma and *SA*, leaving *SA* with no legal values.

Advantage: For many problems the search will be more effective if we combine the MRV heuristic with forward checking.

Disadvantage: Forward checking only makes the current variable arc-consistent, but doesn't look ahead and make all the other variables arc-consistent.

MAC (Maintaining Arc Consistency) algorithm: [More powerful than forward checking, detect this inconsistency.] After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs(X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately.

3. Intelligent backtracking

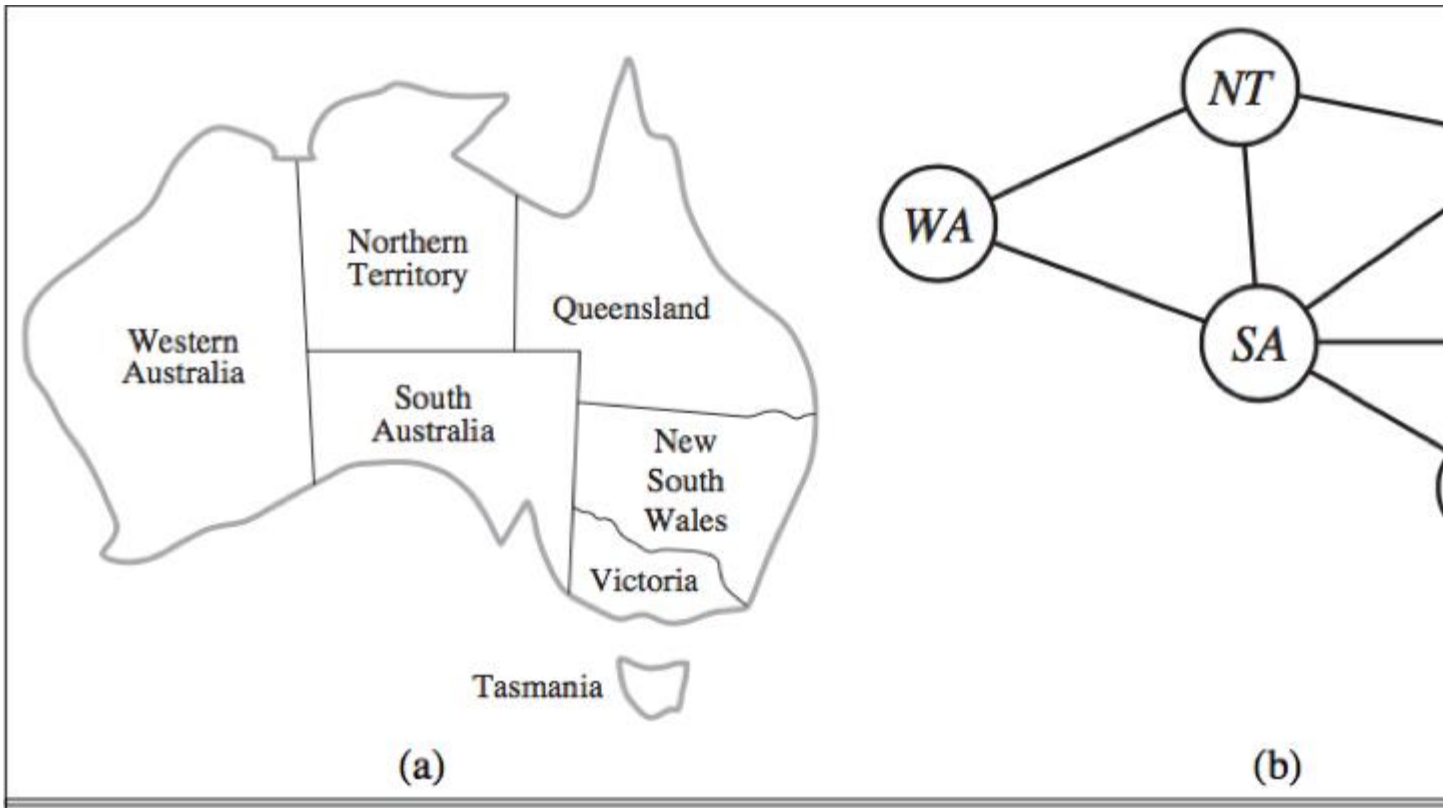


Figure 6.1 (a) The principal states and territories of Australia. Coloring th
be viewed as a constraint satisfaction problem (CSP). The goal is to assign col
region so that no neighboring regions have the same color. (b) The map-colori
represented as a constraint graph.

chronological backtracking: The BACKGRACKING-SEARCH in Fig 6.5. When a branch of the search fails, back up to the preceding variable and try a different value for it. (The most recent decision point is revisited.)
e.g.
Suppose we have generated the partial assignment {Q=red, NSW=green, V=blue, T=red}.
When we try the next variable SA, we see every value violates a constraint.
We back up to T and try a new color, it cannot resolve the problem.
Intelligent backtracking: Backtrack to a variable that was responsible for making one of the possible values of the next variable (e.g. SA) impossible.
Conflict set for a variable: A set of assignments that are in conflict with some value for that variable.
(e.g. The set {Q=red, NSW=green, V=blue} is the conflict set for SA.)
backjumping method: Backtracks to the most recent assignment in the conflict set.
(e.g. backjumping would jump over T and try a new value for V.)

Forward checking can supply the conflict set with no extra work.

Whenever forward checking based on an assignment X=x deletes a value from Y's domain, add X=x to Y's conflict set;
If the last value is deleted from Y's domain, the assignment in the conflict set of Y are added to the conflict set of X.
In fact,every branch pruned by backjumping is also pruned by forward checking. Hence simple backjumping is redundant in a forward-checking search or in a search that uses stronger consistency checking (such as MAC).

Conflict-directed backjumping:
e.g.
consider the partial assignment which is proved to be inconsistent: {WA=red, NSW=red}.
We try T=red next and then assign NT, Q, V, SA, no assignment can work for these last 4 variables.
Eventually we run out of value to try at NT, but simple backjumping cannot work because NT doesn't have a complete conflict set of preceding variables that caused to fail.
The set {WA, NSW} is a deeper notion of the conflict set for NT, caused NT together with any subsequent variables to have no consistent solution. So the algorithm should backtrack to NSW and skip over T.
A backjumping algorithm that uses conflict sets defined in this way is called conflict-direct backjumping.
How to Compute:
When a variable's domain becomes empty, the "terminal" failure occurs, that variable has a standard conflict set.
Let X_j be the current variable, let $conf(X_j)$ be its conflict set. If every possible value for X_j fails, backjump to the most recent variable X_i in $conf(X_j)$, and set
$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$.
The conflict set for an variable means, there is no solution from that variable onward, given the preceding assignment to the conflict set.
e.g.
assign WA, NSW, T, NT, Q, V, SA.
SA fails, and its conflict set is {WA, NT, Q}. (standard conflict set)
Backjump to Q, its conflict set is {NT, NSW}∪{WA,NT,Q}-{Q} = {WA, NT, NSW}.
Backtrack to NT, its conflict set is {WA}∪{WA,NT,NSW}-{NT} = {WA, NSW}.
Hence the algorithm backjump to NSW. (over T)

After backjumping from a contradiction, how to avoid running into the same problem again:
Constraint learning: The idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**. We then record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.

2.5 Local Search for CSPs

Local search algorithms for CSPs use a complete-state formulation: the initial state assigns a value to every variable, and the search change the value of one variable at a time.

Local search algorithms are known to be highly effective in solving a wide range of Constraint Satisfaction Problems (CSPs). These algorithms operate with a complete-state formulation, meaning that they assign values to all variables within the problem, and their search process involves making incremental changes to the value of one variable at a time. This approach is illustrated using the 8-queens problem.

In Figure 6.8 we start on the left with a complete assignment to the 8 variables; typically this will violate several constraints. We then randomly choose a conflicted variable, which turns out to be Q8, the rightmost column. We'd like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables—the min-conflicts heuristic.

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

In the figure we see there are two rows that only violate one constraint; we pick Q8=3 (that is, we move the queen to the 8th column, 3rd row). On the next iteration, in the middle board of the figure, we select Q6 as the variable to change, and note that moving the queen to the 8th row results in no conflicts. At this point there are no more conflicted variables, so we have a solution.

The min-conflicts heuristic: In choosing a new value for a variable, select the value that results in the minimum number of conflicts with other variables

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

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

The landscape of a CSP under the mini-conflicts heuristic usually has a series of plateau. Simulated annealing and Plateau search (i.e. allowing sideways moves to another state with the same score) can help local search find its way off the plateau. This wandering on the plateau can be directed with **tabu search**: keeping a small list of recently visited states and forbidding the algorithm to return to those tates.

Constraint weighting: a technique that can help concentrate the search on the important constraints.
Each constraint is given a numeric weight W_i, initially all 1.
At each step, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.
The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

Local search can be used in an online setting when the problem changes, this is particularly important in scheduling problems.

2.6 The Structure of Problems

The structure of the problem is represented by the constraint graph, can be used to find solutions quickly. Most of the approaches here also apply to other problems besides CSPs, such as probabilistic reasoning.

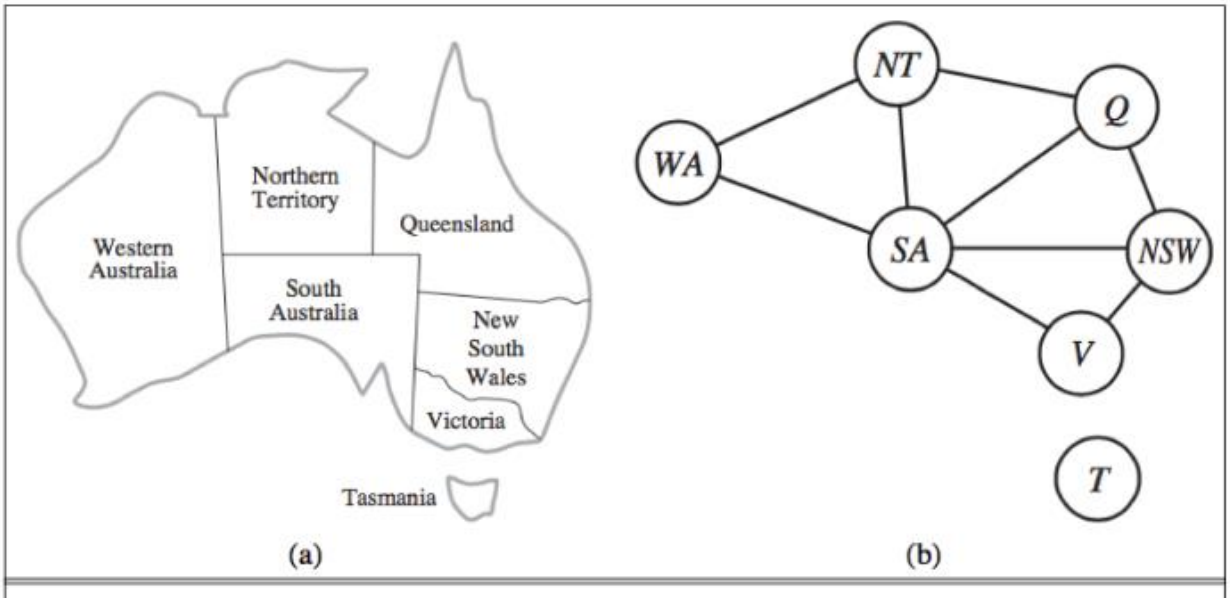


Figure:2.6

1. The structure of constraint graph

The structure of the problem as represented by the constraint graph can be used to find solution quickly.

e.g. The problem can be decomposed into 2 **independent subproblems**: Coloring T and coloring the mainland.

Tree: A constraint graph is a tree when any two varyiable are connected by only one path.

Directed arc consistency (DAC): A CSP is defined to be directed arc-consistent under an ordering of variables $X_1, X_2, \ldots, X_n$ if and only if every X_i is arc-consistent with each X_j for j>i.

By using DAC, any tree-structured CSP can be solved in time linear in the number of variables.

How to solve a tree-structure CSP:

Pick any variable to be the root of the tree;

Choose an ordering of the variable such that each variable appears after its parent in the tree. (**topological sort**)

Any tree with n nodes has n-1 arcs, so we can make this graph directed arc-consistent in O(n) steps, each of which must compare up to d possible domain values for 2 variables, for a total time of $O(nd^2)$.

Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value.

Since each link from a parent to its child is arc consistent, we won't have to backtrack, and can move linearly through the variables.

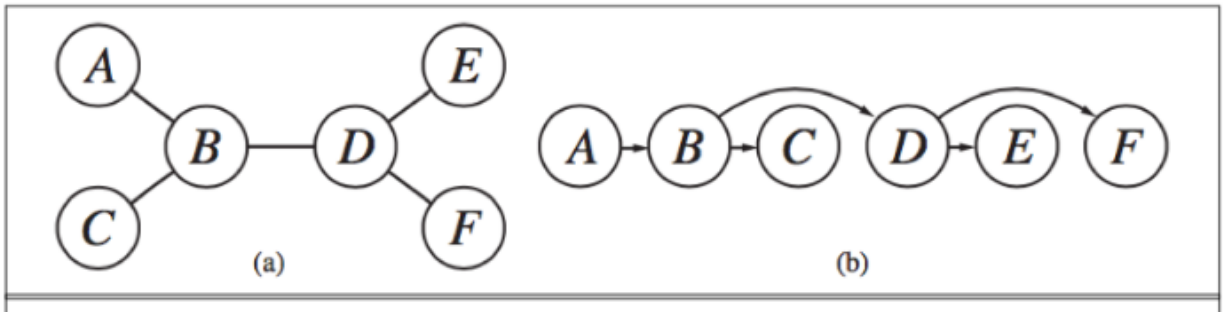


Figure 6.1: The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
    inputs: csp, a CSP with components X, D, C

    n ← number of variables in X
    assignment ← an empty assignment
    root ← any variable in X
    X ← TOPOLOGICALSORT(X, root)
    for j = n down to 2 do
        MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
        if it cannot be made consistent then return failure
    for i = 1 to n do
        assignment[X_i] ← any consistent value from D_i
        if there is no consistent value then return failure
    return assignment
```

Figure 6.2 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

The general algorithm:

Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a cycle cutset.

For each possible assignment to the variables in S that satisfies all constraints on S,

(a) remove from the domain of the remaining variables any values that are inconsistent with the assignment for S, and

(b) If the remaining CSP has a solution, return it together with the assignment for S.

Time complexity: $O(d^c \cdot (n-c)d^2)$, c is the size of the cycle cut set.

Cutset conditioning: The overall algorithmic approach of efficient approximation algorithms to find the smallest cycle cutset.

The first way to reduce a constraint graph to a tree involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure a) Without South Australia, the graph would become a tree, as in (b). Fortunately, we can delete South Australia (in the graph, not the country) by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

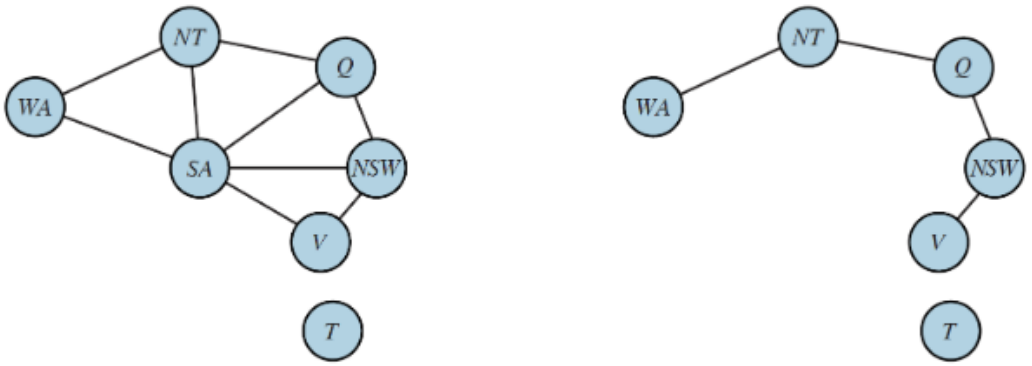


Figure6.3 (a) The original constraint graph from Figure 6.1. (b) After the removal of SA, the constraint graph becomes a forest of two trees.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. (This works for binary CSPs; the situation is more complicated with higher-order constraints.) Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem. Of course, in the general case (as opposed to map coloring), the value chosen for SA could be the wrong one, so we would need to try each possible value. The general algorithm is as follows:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a cycle cutset.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S, and
   (b) if the remaining CSP has a solution, return it together with the assignment for S.

Tree decomposition

The second way to reduce a constraint graph to a tree is based on constructing a tree decomposition of the constraint graph: a transformation of the original graph into a tree where each node in the tree consists of a set of variables, as in Figure. A tree decomposition must satisfy these three requirements:

- Every variable in the original problem appears in at least one of the tree nodes.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
- If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes.

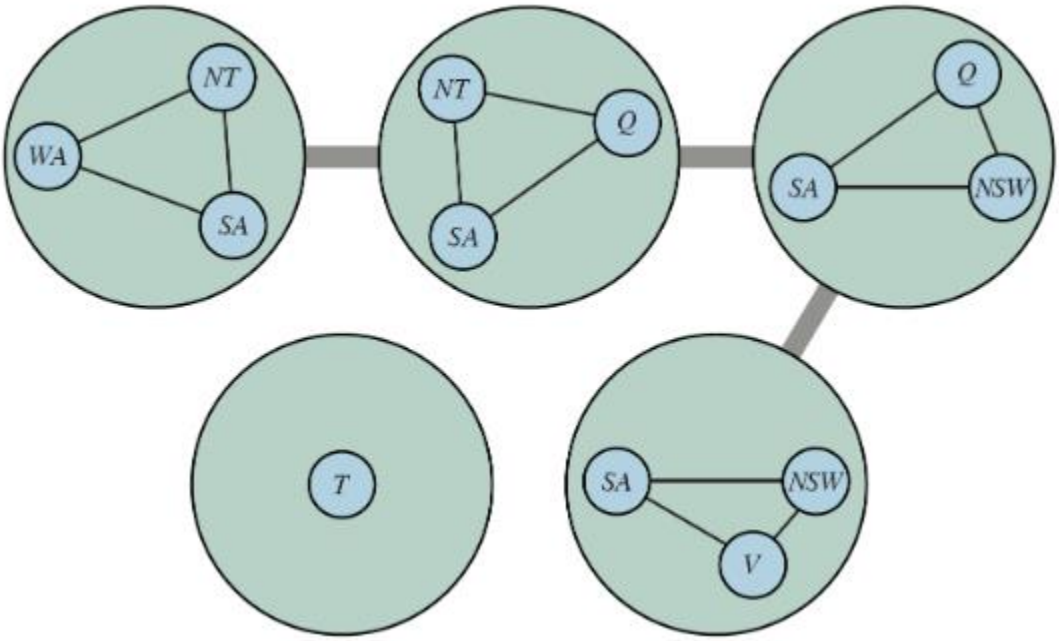


Figure 6.4 A tree decomposition of the constraint graph in Figure 6.3(a)

Once we have a tree-structured graph, we can apply TREE−CSP−SOLVER to get a solution in $O(nd^2)$ time, where n is the number of tree nodes and d is the size of the largest domain. But note that in the tree, a domain is a set of tuples of values, not just individual values.

For example, the top left node in Figure 6.13 represents, at the level of the original problem, a subproblem with variables {WA,NT,SA}, domain {red,green,blue}, and constraints WA \neq NT,SA≠NT,WA≠SA. At the level of the tree, the node represents a single variable, which we can call SANTW A, whose value must be a three-tuple of colors, such as (red,green,blue), but not (red, red, blue), because that would violate the constraint SA≠NT from the original problem. We can then move from that node to the adjacent one, with the variable we can call SANTQ, and find that there is only one tuple, (red, green, blue), that is consistent with the

choice for SANTWA. The exact same process is repeated for the next two nodes, and independently we can make any choice for T.

Value symmetry

So far, we have looked at the structure of the constraint graph. There can also be important structure in the values of variables, or in the structure of the constraint relations themselves. Consider the map-coloring problem with d colors. For every consistent solution, there is actually a set of $d!$ solutions formed by permuting the color names. For example, on the Australia map we know that WA, NT, and SA must all have different colors, but there are 3!=6 ways to assign three colors to three regions. This is called value symmetry. We would like to reduce the search space by a factor of $d!$ by breaking the symmetry in assignments. We do this by introducing a symmetry-breaking constraint. For our example, we might impose an arbitrary ordering constraint, NT<SA<WA, that requires the three values to be in alphabetical order. This constraint ensures that only one of the $d!$ solutions is possible: {NT=blue,SA=green,WA=red}.

For map coloring, it was easy to find a constraint that eliminates the symmetry. In general it is NP-hard to eliminate all symmetry, but breaking value symmetry has proved to be important and effective on a wide range of problems.

UNIT III: Propositional Logic: Knowledge-Based Agents, The Wumpus World, Logic, Propositional Logic, Propositional Theorem Proving: Inference and proofs, Proof by resolution, Horn clauses and definite clauses, Forward and backward chaining, Effective Propositional Model Checking, Agents Based on Propositional Logic.

KNOWLEDGE-BASED AGENTS

The central component of a knowledge-based agent is its knowledge base, or KB. A knowledge base is a set of sentences. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world. Sometimes we dignify a sentence with the name axiom, when the sentence is taken as given without being derived from other sentences.

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively. Both operations may involve inference—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously. Later in this chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the inference process should not make things up as it goes along.

Algorithm shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, KB, which may initially contain some background knowledge.

Each time the agent program is called, it does three things. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

function *KB-AGENT(percept)* ***returns*** *an action*

persistent: *KB, a knowledge base t, a counter, initially 0, indicating time*

TELL(KB, MAKE-PERCEPT-SENTENCE(percept,t))

action ← ASK(KB, MAKE-ACTION-QUERY(t))

TELL(KB, MAKE-ACTION-SENTENCE(action,t))

t ← t + 1

return *action*

Algorithm 3.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action

The agent in algorithm appears quite similar to the agents with internal state. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at the knowledge level, where we need specify only what the agent knows and what its goals are, in order to fix its behavior. For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge because it knows that that will achieve its goal. Notice that this analysis is independent of how the taxi works at the implementation level. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the declarative approach to system building. In contrast, the procedural approach encodes desired behaviors directly as program code. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.

We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. Create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

THE WUMPUS WORLD

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

of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure.

• **Performance measure:** +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.

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

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

• **Sensors:** The agent has five sensors, each of which gives a single bit of information:

– In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a Stench.

– In the squares directly adjacent to a pit, the agent will perceive a Breeze. – In the square where the gold is, the agent will perceive a Glitter.

– When an agent walks into a wall, it will perceive a Bump.

– When the wumpus is killed, it emits a woeful Scream that can be perceived anywhere in the cave.

The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench, Breeze, None, None, None*].

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

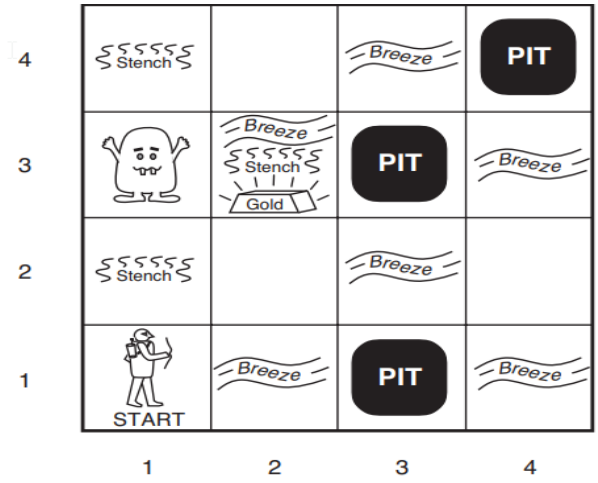


Figure3.1 *A typical wumpus world. The agent is in the bottom left corner, facing right.*

For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 3.1. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 3.2 and 3.3).

The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1,1].

The first percept is [None, None, None, None, None], from which the agent can conclude that its neighbouring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 3.2(a) shows the agent's state of knowledge at this point.

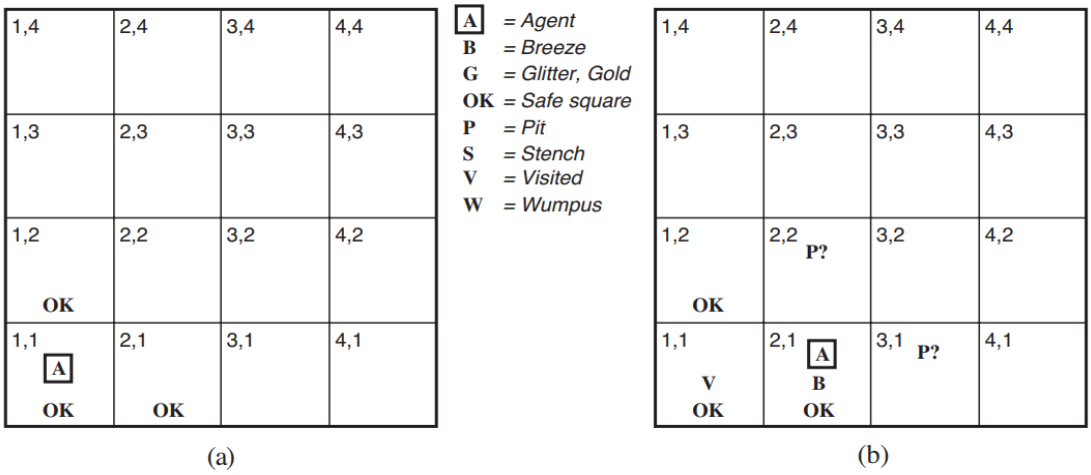


Figure 3.2*The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [None, None, None, None,*

None]. (b) After one move, with percept [None, Breeze, None, None, None].

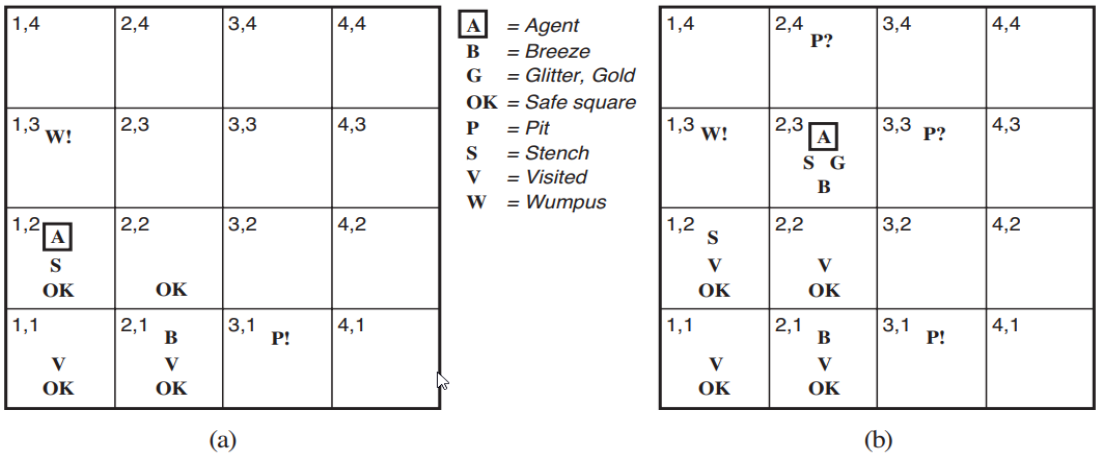


Figure 3.3*Two later stages in the progress of the agent. (a) After the third move, with percept [Stench, None, None, None, None]. (b) After the fifth move, with percept [Stench, Breeze, Glitter , None, None].*

A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighbouring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure 3.2(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But thewumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 3.3(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home.

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct. This is a fundamental property of logical reasoning.

LOGIC

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

A logic must also define the semantics or meaning of sentences. The semantics defines the truth of each sentence with respect to each possible world. For example, the semantics for arithmetic specifies that the sentence "x + y = 4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between."

When we need to be precise, we use the term model in place of "possible world." Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentence. Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence x + y = 4 is true when there are four people in total. Formally, the possible models are just all possible assignments of real numbers to the variables x and y. Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y. If a sentence α is true in model m, we say that m satisfies α or sometimes m is a model of α. We use the notation M(α) to mean the set of all models of α.

Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical entailment between sentences—the idea that a sentence follows logically from another sentence. In mathematical notation, we writeto mean that the sentence α entails the sentence β. The formal definition of entailment is this: α |= β if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$\alpha \models \beta$ *if and only if* $M(\alpha) \subseteq M(\beta)$

The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence x = 0 entails the sentence xy = 0. Obviously, in any model where x is zero, it is the case that xy is zero (regardless of the value of y).

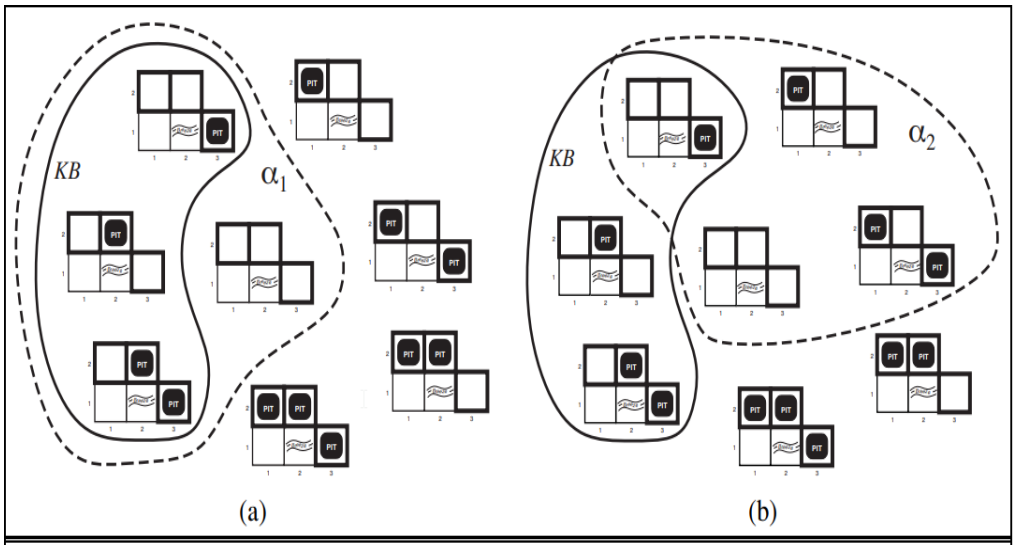


Figure 3.4 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α1 (no pit in [1,2]). (b) Dotted line shows models of α2 (no pit in [2,2]).

We can apply the same kind of analysis to the wumpus-world reasoning exampleConsider the situation in Figure 3.2(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are 23 = 8 possible models. These eight models are shown in Figure 3.4.

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows— for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these areshown surrounded by a solid line in Figure 3.4. Now let us consider two possible conclusions:

α1 = "There is no pit in [1,2]."

α2 = "There is no pit in [2,2]."

We have surrounded the models of α1 and α2 with dotted lines in Figures 73.4(a) and 3.4(b), respectively. By inspection, we see the following:

in every model in which KB is true, α1 is also true.

Hence, KB |= α1: there is no pit in [1,2]. We can also see that

in some models in which KB is true, α2 is false.

Hence, KB |= α2: the agent cannot conclude that there is no pit in [2,2]. (Nor can it conclude that there is a pit in [2,2].)

The preceding example not only illustrates entailment but also shows how the definition LOGICAL INFERENCE of entailment can be applied to derive conclusions—that is, to carry out logical inference. MODEL CHECKING The inference algorithm illustrated in Figure 3.4 is called model checking, because it enumerates all possible models to check that α is true in all models in which KB is true, that is, that M(KB) ⊆ M(α).

In understanding entailment and inference, it might help to think of the set of all consequences of KB as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm i can derive α from KB, we write

KB $\vdash_i$ α , which is pronounced "α is derived from KB by i" or "i derives α from *KB*."

An inference algorithm that derives only entailed sentences is called sound or truth preserving. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable,4 is a sound procedure.

The property of completeness is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue. Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, 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. So, while an inference process operates on "syntax"—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process corresponds to the real-world relationship whereby some aspect of the real world is the case6 by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 3.5.

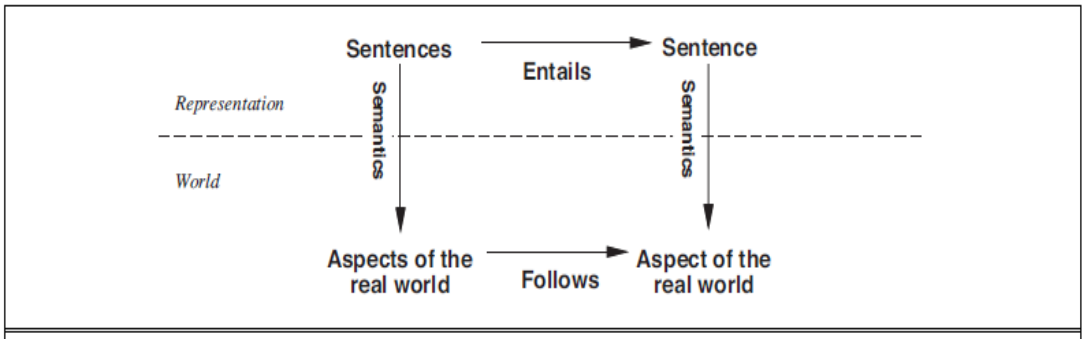


Figure 3.5 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

The final issue to consider is grounding—the connection between logical reasoning processes and the real environment in which the agent exists. In particular, how do we know that KB is true in the real world? (After all, KB is just "syntax" inside the agent's head.) This is a philosophical question about which many, many books have been written. A simple answer is that the agent's sensors create the connection. For example, our wumpus-world agent has a smell sensor. The agent program creates a suitable sentence whenever there is a smell. Then, whenever that sentence is in the knowledge base, it is true in the real world. Thus, the meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them. What about the rest of the agent's knowledge, such as its belief that wumpuses cause smells in adjacent squares? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from perceptual experience but not identical to a statement of that experience. General rules like this are produced by a sentence construction process called learning, which is the subject of Part V. Learning is fallible. It could be the case that wumpuses cause smells except on February 29 in leap years, which is when they take their baths. Thus, KB may not be true in the real world, but with good learning procedures, there is reason for optimism.

PROPOSOTIONAL LOGIC

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

Syntax

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

¬ (not). A sentence such as ¬W1,3 is called the negation of W1,3. A literal is either an atomic sentence (a positive literal) or a negated atomic sentence (a negative literal).

∧ (and). A sentence whose main connective is ∧, such as W1,3 ∧ P3,1, is called a con junction; its parts are the conjuncts. (The ∧ looks like an “A” for “And.”)

∨ (or). A sentence using ∨, such as (W1,3∧P3,1)∨W2,2, is a disjunction of the disjuncts (W1,3 ∧ P3,1) and W2,2. (Historically, the ∨ comes from the Latin “vel,” which means “or.” For most people, it is easier to remember ∨ as an upside-down ∧.)

⇒ (implies). A sentence such as (W1,3∧P3,1) ⇒ ¬W2,2 is called an implication (or conditional). Its premise or antecedent is (W1,3 ∧P3,1), and its conclusion or consequent is ¬W2,2. Implications are also known as rules or if–then statements. The implication RULES symbol is sometimes written in other books as ⊃ or →.

⇔ (if and only if). The sentence W1,3 ⇔ ¬W2,2 is a biconditional. Some other books write this as ≡.

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

Figure 3.6 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 3.6 gives a formal grammar of propositional logic; see page 1060 if you are not familiar with the BNF notation. The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways. To eliminate the ambiguity we define a precedence for each operator. The "not" operator (¬) has the highest precedence, which means that in the sentence ¬A ∧ B the ¬ binds most tightly, giving us the equivalent of (¬A)∧B rather than ¬(A∧B). (The notation for ordinary arithmetic is the same: −2+4 is 2, not –6.) When in doubt, use parentheses to make sure of the right interpretation. Square brackets mean the same thing as parentheses; the choice of square brackets or parentheses is solely to make it easier for a human to read a sentence.

Semantics

Having specified the syntax of propositional logic, we now specify its semantics. The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the truth value—true or false—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols P1,2, P2,2, and P3,1, then one possible model is

m1 = {P1,2 = false, P2,2 = false, P3,1 = true} .

With three proposition symbols, there are 23 = 8 possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. P1,2 is just a symbol; it might mean "there is a pit in [1,2]" or "I'm in Paris today and tomorrow."

The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:

• True is true in every model and False is false in every model.
• The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m1 given earlier, P1,2 is false.

For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):

• ¬P is true iff P is false in m.
• P ∧ Q is true iff both P and Q are true in m.
• P ∨ Q is true iff either P or Q is true in m.
• P ⇒ Q is true unless P is true and Q is false in m.
• P ⇔ Q is true iff P and Q are both true or both false in m.

The rules can also be expressed with truth tables that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be

computed with respect to any model m by a simple recursive evaluation. For example, the sentence ¬P1,2 ∧ (P2,2 ∨ P3,1), evaluated in m1, gives true ∧ (false ∨ true) = true ∧ true = true.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that P ∨ Q is true when P is true or Q is true or both. A different connective, called "exclusive or" ("xor" for short), yields false when both disjuncts are true.7 There is no consensus on the symbol for exclusive or; some choices are ∨˙ or = or ⊕.

The truth table for ⇒ may not quite fit one's intuitive understanding of "P implies Q" or "if P then Q." For one thing, propositional logic does not require any relation of causation or relevance between P and Q. The sentence "5 is odd implies Tokyo is the capital of Japan" is a true sentence of propositional logic (under the normal interpretation), even though it is a decidedly odd sentence of English. Another point of confusion is that any implication is true whenever its antecedent is false. For example, "5 is even implies Sam is smart" is true, regardless of whether Sam is smart. This seems bizarre, but it makes sense if you think of "P ⇒ Q" as saying, "If P is true, then I am claiming that Q is true. Otherwise I am making no claim." The only way for this sentence to be false is if P is true but Q is false.

The biconditional, P ⇔ Q, is true whenever both P ⇒ Q and Q ⇒ P are true. In English, this is often written as "P if and only if Q." Many of the rules of the wumpus world are best written using ⇔. For example, a square is breezy if a neighboring square has a pit, and a square is breezy only if a neighboring square has a pit. So we need a biconditional,

B1,1 ⇔ (P1,2 ∨ P2,1) ,

where B1,1 means that there is a breeze in [1,1].

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

Figure 3.7 Truth tables for the five logical connectives. To use the table to compute, for example, the value of P ∨ Q when P is true and Q is false, first look on the left for the row where P is true and Q is false (the third row). Then look in that row under the P ∨Q column to see the result: true.

A Simple Knowledge Base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the immutable aspects of the wumpus world, leaving the mutable aspects for a later section. For now, we need the following symbols for each [x, y] location:

Px,y is true if there is a pit in [x, y].

Wx,y is true if there is a wumpus in [x, y], dead or alive.
Bx,y is true if the agent perceives a breeze in [x, y].
Sx,y is true if the agent perceives a stench in [x, y].
The sentences we write will suffice to derive ¬P1,2 (there is no pit in [1,2]) We label each sentence Ri so that we can refer to them:

- There is no pit in [1,1]:

  R1 : ¬P1,1 .

- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

  R2 : B1,1 ⇔ (P1,2 ∨ P2,1) .

  R3 : B2,1 ⇔ (P1,1 ∨ P2,2 ∨ P3,1) .

  R4 : ¬B1,1 . R5 : B2,1 .

A Simple Inference Procedure

Our goal now is to decide whether KB |= α for some sentence α. For example, is ¬P1,2 entailed by our KB? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of true or false to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are B1,1, B2,1, P1,1, P1,2, P2,1, P2,2, and P3,1. With seven symbols, there are 27 = 128 possible models; in three of these, KB is true (Figure 3.8). In those three models, ¬P1,2 is true, hence there is no pit in [1,2]. On the other hand, P2,2 is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2].

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

Figure 3.8 A truth table constructed for the knowledge base given in the text. KB is true if R1 through R5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, P1,2 is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

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

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

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

Figure 3.9 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns true if a sentence holds within a model. The variable model represents a partial model—an assignment to some of the symbols. The keyword "and" is used here as a logical operation on its two arguments, returning true or false.

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

Figure 3.10 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

PROPOSITIONAL THEOREM PROVING

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

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 3.10. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

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

The second concept we will need is validity. A sentence is valid if it is true in all models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as tautologies—they are necessarily true. Because the sentence True is true in all models, every valid sentence is logically equivalent to True. What good are valid sentences? From our definition of entailment, we can derive the deduction theorem, which was known to the ancient Greeks:

For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 3.9 does or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to True. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference. The final concept we will need is satisfiability. A sentence is satisfiable if it is true in, or satisfied by, some model. For example, the knowledge base given earlier, $(R1 \wedge R2 \wedge R3 \wedge R4 \wedge R5)$, is satisfiable because there are three models in which it is true, as shown in Figure 3.8. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences in propositional logic—the SAT problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 6 ask whether the constraints are satisfiable by some assignment.

Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result: $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable. Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds exactly to the standard mathematical proof technique of reductio ad absurdum (literally, “reduction to an absurd thing”). It is also called proof by refutation or proof by contradiction. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α. This contradiction is exactly what is meant by saying that the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

Inference and Proofs

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

$$\alpha \Rightarrow \beta, \alpha / \beta .$$

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

Another useful inference rule is And-Elimination, which says that, from a conjunction, any of the conjuncts can be inferred: $\alpha \wedge \beta / \alpha$. For example, from (WumpusAhead ∧ WumpusAlive), WumpusAlive can be inferred. By considering the possible truth values of α and β, one can show easily that Modus Ponens and And-Elimination are sound once and for all. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models. All of the logical equivalences in Figure 3.10 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$\alpha \Leftrightarrow \beta / (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$ and $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha) / \alpha \Leftrightarrow \beta$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain $\alpha \Rightarrow \beta$ and α from β.

Let us see how these inference rules and equivalences can be used in the wumpus world. We start with the knowledge base containing R1 through R5 and show how to prove ¬P1,2, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R2 to obtain

R6 : (B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

Then we apply And-Elimination to R6 to obtain

R7 : ((P1,2 ∨ P2,1) ⇒ B1,1) .

Logical equivalence for contrapositives gives

R8 : (¬B1,1 ⇒ ¬(P1,2 ∨ P2,1)) .

Now we can apply Modus Ponens with R8 and the percept R4 (i.e., ¬B1,1), to obtain

R9 : ¬(P1,2 ∨ P2,1) .

Finally, we apply De Morgan's rule, giving the conclusion

R10 : ¬P1,2 ∧ ¬P2,1 . That is, neither [1,2] nor [2,1] contains a pit.

We found this proof by hand, but we can apply any of the search algorithms in Chapter 3 to find a sequence of steps that constitutes a proof. We just need to define a proof problem as follows:

• INITIAL STATE: the initial knowledge base.
• ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
• RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
• GOAL: the goal is a state that contains the sentence we are trying to prove.

Thus, searching for proofs is an alternative to enumerating models. In many practical cases finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are. For example, the proof given earlier leading to ¬P1,2 ∧ ¬P2,1 does not mention the propositions B2,1, P1,1, P2,2, or P3,1. They can be ignored because the goal proposition, P1,2, appears only in sentence R2; the other propositions in R2 appear only in R4 and R2; so R1, R3, and R5 have no bearing on the proof. The same would hold even if we added a million more sentences to the knowledge base; the simple truth-table algorithm, on the other hand, would be overwhelmed by the exponential explosion of models.

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

if KB |= α then KB ∧ β |= α .

For example, suppose the knowledge base contains the additional assertion β stating that there are exactly eight pits in the world. This knowledge might help the agent draw additional conclusions, but it cannot invalidate any conclusion α already inferred—such as the conclusion that there is no pit in [1,2]. Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base—the conclusion of the rule must follow regardless of what else is in the knowledge base.

Proof by Resolution

We have argued that the inference rules covered so far are sound, but we have not discussed the question of completeness for the inference algorithms that use them. Search algorithms such as iterative deepening search (page 89) are complete in the sense that they will find any reachable goal, but if the available inference rules are inadequate, then the goal is not reachable—no proof exists that uses only those inference rules. For example, if we removed the biconditional elimination rule, the proof in the preceding section would not go through. The current section introduces a single inference rule, resolution, that yields a complete inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the wumpus world. Let us consider the steps leading up to Figure 3.3(a): the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

R11 : ¬B1,2 .

R12 : B1,2 ⇔ (P1,1 ∨ P2,2 ∨ P1,3) .

By the same process that led to R10 earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pitless):

R13 : ¬P2,2 .

R14 : ¬P1,3 .

We can also apply biconditional elimination to R3, followed by Modus Ponens with R5, to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

R15 : P1,1 ∨ P2,2 ∨ P3,1 .

Now comes the first application of the resolution rule: the literal ¬P2,2 in R13 resolves with the literal P2,2 in R15 to give the resolvent

R16 : P1,1 ∨ P3,1 . In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal ¬P1,1 in R1 resolves with the literal P1,1 in R16 to give

R17 : P3,1 .

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

$$1 \vee \cdots \vee lk, \; m \; / l1 \vee \cdots \vee li-1 \vee li+1 \vee \cdots \vee lk \; ,$$

where each l is a literal and ————————————i and m are complementary literals (i.e., one is the negation of the other). Thus, the unit resolution rule takes a clause—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a unit clause.

The unit resolution rule can be generalized to the full resolution rule,

$$l1 \vee \cdots \vee lk, \; m1 \vee \cdots \vee mn \; / \; l1 \vee \cdots \vee li-1 \vee li+1 \vee \cdots \vee lk \vee m1 \vee \cdots \vee mj-1 \vee mj+1 \vee \cdots \vee mn \; ,$$

where li and mj are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses except the two complementary literals. For example, we have

$$P1,1 \vee P3,1, \; \neg P1,1 \vee \neg P2,2 \; / \; P3,1 \vee \neg P2,2 \; .$$

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal. The removal of multiple copies of literals is called factoring. For example, if we resolve (A ∨ B) with (A ∨ ¬B), we obtain (A ∨ A), which is reduced to just A.

The soundness of the resolution rule can be seen easily by considering the literal li that is complementary to literal mj in the other clause. If li is true, then mj is false, and hence $m1 \vee \cdots \vee mj-1 \vee mj+1 \vee \cdots \vee mn$ must be true, because $m1 \vee \cdots \vee mn$ is given. If li is false, then $l1 \vee \cdots \vee li-1 \vee li+1 \vee \cdots \vee lk$ must be true because $l1 \vee \cdots \vee lk$ is given. Now li is either true or false, so one or other of these conclusions holds—exactly as the resolution rule states.

What is more surprising about the resolution rule is that it forms the basis for a family of complete inference procedures. A resolution-based theorem prover can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$. The next two subsections explain how resolution accomplishes this.

Conjunctive Normal Form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that every sentence of propositional logic is logically equivalent to a conjunction of clauses. A sentence expressed as a conjunction of clauses is said to be in conjunctive normal form or CNF (see Figure 3.13). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B1,1 \Leftrightarrow (P1,2 \vee P2,1)$ into CNF. The steps are as follows:

1. Eliminate ⇔, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
$$(B1,1 \Rightarrow (P1,2 \vee P2,1)) \wedge ((P1,2 \vee P2,1) \Rightarrow B1,1) \; .$$
2. Eliminate ⇒, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

$$(\neg B1,1 \vee P1,2 \vee P2,1) \wedge (\neg(P1,2 \vee P2,1) \vee B1,1) .$$

3. CNF requires ¬ to appear only in literals, so we "move ¬ inwards" by repeated application of the following equivalences:

   $\neg(\neg\alpha) \equiv \alpha$ (double-negation elimination)

   $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ (De Morgan)

   $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ (De Morgan)

   In the example, we require just one application of the last rule:

   $$(\neg B1,1 \vee P1,2 \vee P2,1) \wedge ((\neg P1,2 \wedge \neg P2,1) \vee B1,1) .$$

4. Now we have a sentence containing nested ∧ and ∨ operators applied to literals. We apply the distributivity law, distributing ∨ over ∧ wherever possible.

   $$(\neg B1,1 \vee P1,2 \vee P2,1) \wedge (\neg P1,2 \vee B1,1) \wedge (\neg P2,1 \vee B1,1) .$$

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

A Resolution Algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction introduced on page 250. That is, to show that KB |= α, we show that (KB ∧ ¬α) is unsatisfiable. We do this by proving a contradiction.

In resolution algorithm, First (KB ∧ ¬α) is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- there are no new clauses that can be added, in which case KB does not entail α; or,
- two clauses resolve to yield the empty clause, in which case KB entails α.

The empty clause—a disjunction of no disjuncts—is equivalent to False because a disjunction is true only if at least one of its disjuncts is true. Another way to see that an empty clause represents a contradiction is to observe that it arises only from resolving two complementary unit clauses such as P and ¬P.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is

$$KB = R2 \wedge R4 = (B1,1 \Leftrightarrow (P1,2 \vee P2,1)) \wedge \neg B1,1$$

and we wish to prove α which is, say, ¬P1,2. When we convert (KB ∧ ¬α) into CNF, we obtain the clauses shown at the top of Figure 3.12. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when P1,2 is resolved with ¬P1,2, we obtain the empty clause, shown as a small square. Inspection of Figure 3.12 reveals that many resolution steps are pointless. For example, the clause $B1,1 \vee \neg B1,1 \vee P1,2$ is equivalent to True ∨ P1,2 which is equivalent to True. Deducing that True is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.

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

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

Figure 3.11 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

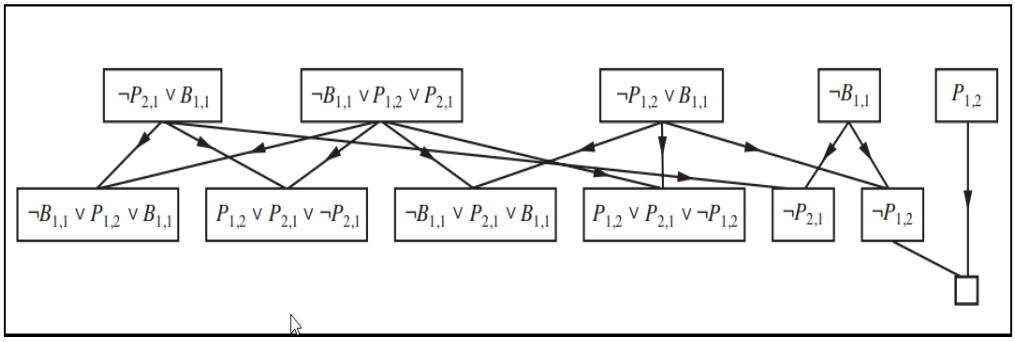


Figure 3.12 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. ¬P1,2 is shown to follow from the first four clauses in the top row. Partial application of PL-RESOLUTION to a simple inference in the wumpus world. ¬P1,2 is shown to follow from the first four clauses in the top row.

Completeness of resolution

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the resolution closure RC (S) of a set of clauses S, which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable clauses. It is easy to see that RC (S) must be finite, because there are only finitely many distinct clauses that can be constructed out of the symbols P1,...,Pk that appear in S. (Notice that this would not be true without the factoring step that removes multiple copies of literals.) Hence, PL-RESOLUTION always terminates.

The completeness theorem for resolution in propositional logic is called the ground resolution theorem:

If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

This theorem is proved by demonstrating its contrapositive: if the closure RC(S) does not contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for P1,...,Pk. The construction procedure is as follows:

For i from 1 to k,
– If a clause in RC(S) contains the literal ¬Pi and all its other literals are false under the assignment chosen for P1,...,Pi−1, then assign false to Pi.
– Otherwise, assign true to Pi.

This assignment to P1,...,Pk is a model of S. To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol Pi causes some clause C to become false. For this to happen, it must be the case that all the other literals in C must already have been falsified by assignments to P1,...,Pi−1. Thus, C must now look like either (false ∨ false ∨ ⋯ false∨Pi) or like (false∨false∨⋯ false∨¬Pi). If just one of these two is in RC(S), then the algorithm will assign the appropriate truth value to Pi to make C true, so C can only be falsified if both of these clauses are in RC(S). Now, since RC(S) is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to P1,...,Pi−1. This contradicts our assumption that the first falsified clause appears at stage i. Hence, we have proved that the construction never falsifies a clause in RC(S); that is, it produces a model of RC(S) and thus a model of S itself (since S is contained in RC(S)).

Horn Clauses & Definite Clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

One such restricted form is the definite clause, which is a disjunction of literals of which exactly one is positive. For example, the clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) is a definite clause, whereas (¬B1,1 ∨ P1,2 ∨ P2,1) is not.

Slightly more general is the Horn clause, which is a disjunction of literals of which at most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called goal clauses. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

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

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. For example, the definite clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) can be written as the implication (L1,1 ∧ Breeze) ⇒ B1,1. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy. BODY In Horn form, the premise is called the body and the

conclusion is called the head. A HEAD sentence consisting of a single positive literal, such as L1,1, is called a fact. It too can FACT be written in implication form as True ⇒ L1,1, but it is simpler to write just L1,1.

2. Inference with Horn clauses can be done through the forward-chaining and backwardchaining algorithms, which we explain next. Both of these algorithms are natural, BACKWARDCHAINING in that the inference steps are obvious and easy for humans to follow.
3. Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base—a pleasant surprise.

$$
\begin{aligned}
CNFSentence &\rightarrow Clause_1 \wedge \cdots \wedge Clause_n \\
Clause &\rightarrow Literal_1 \vee \cdots \vee Literal_m \\
Literal &\rightarrow Symbol \mid \neg Symbol \\
Symbol &\rightarrow P \mid Q \mid R \mid \ldots \\
HornClauseForm &\rightarrow DefiniteClauseForm \mid GoalClauseForm \\
DefiniteClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow Symbol \\
GoalClauseForm &\rightarrow (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow False
\end{aligned}
$$

Figure 3.13 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as A ∧ B ⇒ C is still a definite clause when it is written as ¬A ∨ ¬B ∨ C, but only the former is considered the canonical form for definite clauses. One more class is the k-CNF sentence, which is a CNF sentence where each clause has at most k literals.

Forward and Backward Chaining

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if L1,1 and Breeze are known and (L1,1 ∧ Breeze) ⇒ B1,1 is in the knowledge base, then B1,1 can be added. This process continues until the query q is added or until no further inferences can be made. The detailed algorithm is shown in Figure 3.14; the main point to remember is that it runs in linear time.

The best way to understand the algorithm is through an example and a picture. Figure 3.15(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 3.15(b) shows the same knowledge base drawn as an AND–OR graph (see Chapter 4). In AND–OR graphs, multiple links joined by an arc indicate a conjunction—every link must be proved—while multiple links without an arc indicate a disjunction—any link can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.

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

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

Figure 3.14 The forward-chaining algorithm for propositional logic

It is easy to see that forward chaining is sound: every inference is essentially an application of Modus Ponens. Forward chaining is also complete: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the inferred table (after the algorithm reaches a fixed point where no new inferences are possible). The table contains true for each symbol inferred during the process, and false for all other symbols. We can view the table as a logical model; moreover, every definite clause in the original KB is true in this model. To see this, assume the opposite, namely that some clause a1∧...∧ak ⇒ b is false in the model. Then a1 ∧ ... ∧ ak must be true in the model and b must be false in the model. But this contradicts our assumption that the algorithm has reached a fixed point! We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

Forward chaining is an example of the general concept of data-driven reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be cancelled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor's garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 3.15, it works back down the graph until it reaches a set of known facts, A and B, that

forms the basis for a proof. As with forward chaining, an efficient implementation runs in linear time.

Backward chaining is a form of goal-directed reasoning. It is useful for answering specific questions such as “What shall I do now?” and “Where are my keys?” Often, the cost of backward chaining is much less than linear in the size of the knowledge base, because the process touches only relevant facts.

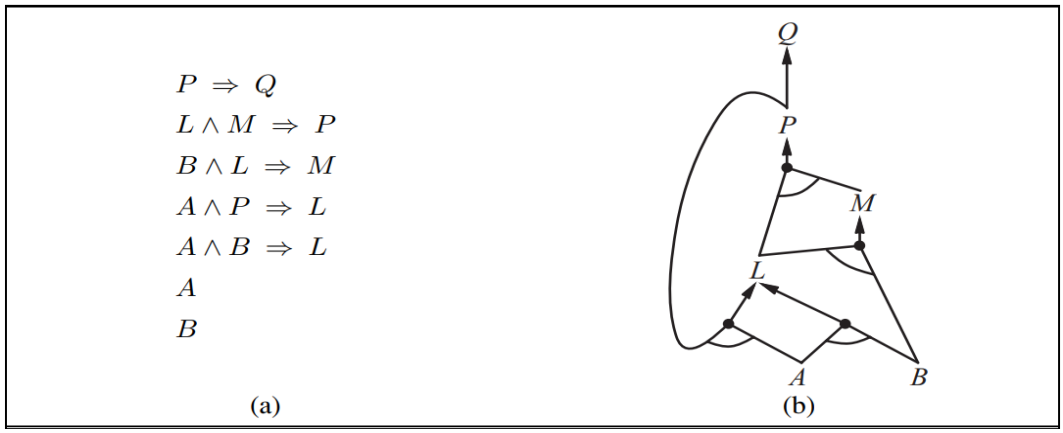


Figure 3.15 (a) A set of Horn clauses. (b) The corresponding AND–OR graph

EFFECTIVE PROPOSITIONAL MODEL CHECKING

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the “technology” of propositional logic.

The algorithms we describe are for checking satisfiability: the SAT problem. (As noted earlier, testing entailment, $\alpha \models \beta$, can be done by testing unsatisfiability of $\alpha \wedge \neg\beta$.) We have already noted the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

A complete backtracking algorithm

The first algorithm we consider is often called the Davis–Putnam algorithm, after the seminal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses. Like BACKTRACKING-SEARCH and TT-ENTAILS?, it is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of TT-ENTAILS?:

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

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

• Unit clause heuristic: A unit clause was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model. For example, if the model contains B = true, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to false. The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately (Exercise 7.23). Notice also that assigning one unit clause can create another unit clause—for example, when C is set to false, $(C \vee A)$ becomes a unit clause, causing true to be assigned to A. This "cascade" of forced assignments is called unit propagation. It resembles the process of forward chaining with definite clauses, and indeed, if the CNF expression contains only definite clauses then DPLL essentially replicates forward chaining.

UNIT IV: First-Order Logic: Representation, Syntax and Semantics of First-Order Logic, Uses of 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

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.

First-Order Logic is a logic which is sufficiently expressive to represent a good deal of our common sense knowledge.

• It is also either includes or forms the foundation of many other representation languages.

• It is also called as First-Order Predicate calculus.

• It is abbreviated as FOL or FOPC

FOL adopts the foundation of propositional logic with all its advantages to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks.

The Syntax of natural language contains elements such as,

1. Nouns and noun phrases that refer to objects (Squares, pits, rumpuses)

2. Verbs and verb phrases that refer to among objects (is breezy, is adjacent to)

Some of these relations are functions-relations in which there is only one “Value” for a given “input”. Whereas propositional logic assumes the world contains facts, first-order logic (like natural language) assumes the world contains Objects: people, houses, numbers, colors, baseball games, wars, …

Relations: red, round, prime, brother of, bigger than, part of, comes between,… Functions: father of, best friend, one more than, plus,…….

3.1 SPECIFY THE SYNTAX OF FIRST-ORDER LOGIC IN BNF FORM

The domain of a model is DOMAIN the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object. Figure 8.2 shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown. The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation TUPLE is just the set of tuples of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.) Thus, the brotherhood relation in this model is the set

$$\{ \langle \textit{Richard the Lionheart}, \textit{King John} \rangle, \langle \textit{King John}, \textit{Richard the Lionheart} \rangle \}$$

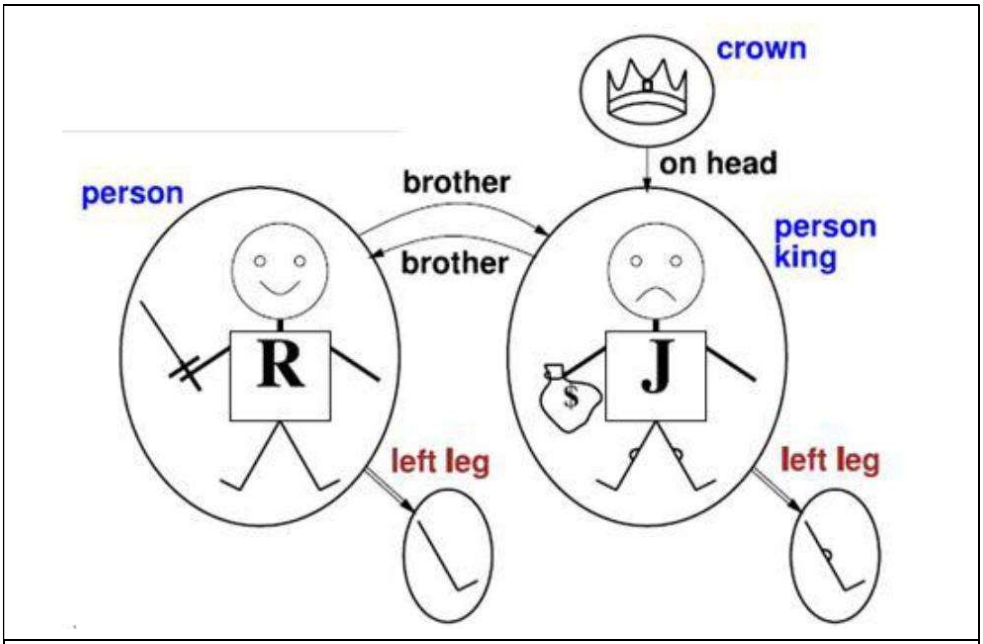


Figure 3.1 First-Order Logic in BNF Form

The crown is on King John's head, so the "on head" relation contains just one tuple, _ the crown, King John_. The "brother" and "on head" relations are binary relations — that is, they relate pairs of objects. Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

$$\langle \textit{Richard the Lionheart} \rangle \rightarrow \textit{Richard's left leg}$$
$$\langle \textit{King John} \rangle \rightarrow \textit{John's left leg}$$

The five objects are,

- Richard the Lionheart
- His younger brother
- The evil King John
- The left legs of Richard and John
- A crown

- The objects in the model may be related in various ways, In the figure Richard and John are brothers.
- Formally speaking, a relation is just the set of tuples of objects that are related.
- A tuple is a collection of Objects arranged in a fixed order and is written with angle brackets surrounding the objects.
- Thus, the brotherhood relation in this model is the set **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- The crown is on King John's head, so the "on head" relation contains just one tuple, (the crown, King John).
    - The relation can be binary relation relating pairs of objects (Ex:- "Brother") or unary relation representing a common object (Ex:- "Person" representing both Richard and John)

Certain kinds of relationships are best considered as functions that relates an object to exactly one object.

- For Example:- each person has one left leg, so the model has a unary "left leg" function that includes the following mappings (Richard the Lionheart) ----> Richard's left leg (King John) ----> John's left leg
- **Symbols and Interpretations:**
- The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations** and **functions**
- **Kinds of Symbols**
- The symbols come in three kinds namely,
- Constant Symbols standing for **Objects** (Ex:- Richard)
- Predicate Symbols standing for **Relations** (Ex:- King)
- Function Symbols stands for **functions** (Ex:-Left Leg)

o Symbols will begin with uppercase letters

o The choice of names is entirely up to the user

o Each predicate and function symbol comes with an arity

o Arity fixes the number of arguments.

- The semantics must relate sentences to models in order to determine truth.
- To do this, an interpretation is needed specifying exactly which **objects, relations** and **functions** are referred to by the **constant, predicate and function symbols**.
- One possible interpretation called as the intended interpretation- is as follows;
- **Richard** refers to **Richard the Lion heart** and **John** refers to the **evil King John.**
- **Brother** refers to the brotherhood relation, that is the set of tuples of objects given in equation **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- **On Head** refers to the "on head" relation that holds between the crown and King John; **Person, King** and **Crown** refer to the set of objects that are persons, kings and crowns.
- **Left leg** refers to the "left leg" function, that is, the mapping given in **{(Richard the Lion heart, King John), (King John, Richard the Lionheart)}**

A complete description from the formal grammar is as follows

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \\
&\mid (\ \textit{Sentence Connective Sentence}\ ) \\
&\mid \textit{Quantifier Variable},\ldots\ \textit{Sentence} \\
&\mid \neg\, \textit{Sentence} \\
\\
\textit{AtomicSentence} &\rightarrow \textit{Predicate}(\textit{Term},\ldots) \mid \textit{Term} = \textit{Term} \\
\\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term},\ldots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\\
\textit{Connective} &\rightarrow \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \ldots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \ldots \\
\textit{Predicate} &\rightarrow \textit{Before} \mid \textit{HasColor} \mid \textit{Raining} \mid \ldots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \ldots
\end{aligned}
$$

Term A term is a logical expression that refers TERM to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use Left Leg (John). The formal semantics of terms is straightforward. Consider a term f(t1, . . . , tn). The function symbol f refers to some function in the model. Atomic sentences Atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as Brother (Richard, John). Atomic sentences can have complex terms as arguments. Thus, Married (Father (Richard),Mother (John)) states that Richard the Lionheart's father is married to King John's mother.

Complex Sentences

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

¬Brother (LeftLeg (Richard), John)

Brother (Richard, John) ∧ Brother (John, Richard)

King(Richard) ∨ King(John)

¬King(Richard) ⇒ King(John).

Quantifiers

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name. First-order logic contains two standard quantifiers, called universal and existential

Universal quantification (∀)

"All kings are persons," is written in first-order logic as

∀x King(x) ⇒ Person(x)

∀ is usually pronounced "For all. . ." Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a variable. A term with no variables is called a **ground term.**

Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

x → Richard the Lionheart,

x → King John, x → Richard's left leg,

x → John's left leg,

x → the crown.

The universally quantified sentence ∀ x King(x) ⇒ Person(x) is true in the original model if the sentence King(x) ⇒ Person(x) is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

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

Existential quantification (∃)

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier. To say, for example, that King John has a crown on his head, we write

∃x Crown(x) ∧OnHead(x, John)

∃x is pronounced "There exists an x such that . . ." or "For some x . . ." More precisely, ∃x P is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

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

The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Just as ⇒ appears to be the natural connective to use with ∀, ∧ is the natural connective to use with ∃.

Using ∧ as the main connective with ∀ led to an overly strong statement in the example in the previous section; using ⇒ with ∃ usually leads to a very weak statement, indeed. Consider the following sentence:

∃x Crown(x) ⇒OnHead(x, John)

Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

Richard the Lionheart is a crown ⇒ Richard the Lionheart is on John's head;
King John is a crown ⇒ King John is on John's head;
Richard's left leg is a crown ⇒ Richard's left leg is on John's head;

and so on. Now an implication is true if both premise and conclusion are true, or if its premise is false. So if Richard the Lionheart is not a crown, then the first assertion is true and the existential is satisfied. So, an existentially quantified implication sentence is true whenever any object fails to satisfy the premise

Nested quantifiers

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

∀x ∀y Brother (x, y) ⇒ Sibling(x, y)

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write

∀x, y Sibling(x, y) ⇔ Sibling(y, x). In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

∀x ∃ y Loves(x, y).

On the other hand, to say "There is someone who is loved by everyone," we write

∃y ∀ x Loves(x, y).

The order of quantification is therefore very important. It becomes clearer if we insert parentheses.

∀x (∃ y Loves(x, y)) says that everyone has a particular property, namely, the property that they love someone. On the other hand,

∃y (∀ x Loves(x, y)) says that someone in the world has a particular property, namely the property of being loved by everybody.

Connections between ∀ and ∃

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

∀x ¬Likes(x, Parsnips) is equivalent to ¬

∀x ¬Likes(x, Parsnips) is equivalent to ¬∃ x Likes(x, Parsnips). We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

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

Equality

We can use the equality symbol to signify that two terms refer to the same object. For example,

Father (John)=Henry

says that the object referred to by Father (John) and the object referred to by Henry are the same.

The equality symbol can be used to state facts about a given function, as we just did for the Father symbol. It can also be used with negation to insist that two terms are not the same object. To say that Richard has at least two brothers, we would write

∃x, y Brother (x,Richard) ∧ Brother (y,Richard) ∧ ¬(x=y).

Compare different knowledge representation languages

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief ∈ [0, 1]
Fuzzy logic	facts with degree of truth ∈ [0, 1]	known interval value

Figure 3.2 Formal languages and their ontological and epistemological commitments

What are the syntactic elements of First Order Logic?

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, come in three kinds:

a) constant symbols, which stand for objects;
b) predicate symbols, which stand for relations;
c) and function symbols, which stand for functions.

We adopt the convention that these symbols will begin with uppercase letters. Example:

Constant symbols :
Richard and John;
predicate symbols :
Brother, On Head, Person, King, and Crown; function symbol : LeftLeg.

Quantifiers

Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name if a logic that allows object is found.

It has two type,

The following are the types of standard quantifiers,

- Universal
- Existential

Universal quantification

Explain Universal Quantifiers with an example.

Rules such as "All kings are persons," is written in first-order logic as

$\forall x$ King(x) => Person(x)

where $\forall$ is pronounced as " For all.."

Thus, the sentence says, "For all *x,* if *x* is a king, then is a person." The symbol x is called a variable(lower case letters)

The sentence $\forall$**x** P, where P is a logical expression says that P is true for every object x.

- Universal Quantification make statement about every object.
- "All Kings are persons", is written in first-order logic as

  $\forall_x$ king (x) ⇒ Person (x)

- $\forall$ is usually pronounced "For all….", Thus the sentences says , "For all x, if x is a king, then x is a person".
- The symbol x is called a variable.
- A variable is a term all by itself, and as such can also serve as the argument of a function-for example, **LeftLeg(x)**.
- A term with no variables is called a **ground term.**
- Based on our model, we can extend the interpretation in five ways,

  x --------- Richard the Lionheart

  x --------- King John

  x --------- Richard's Left leg

  x --------- John's Left leg

  x --------- the crown

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

Richard the Lionheart ------ Richard the Lionheart is a person

King John is a King ------ King John is a Person

Richard's left leg is King -------- Richard's left leg is a person

John's left leg is a King -------- John's left leg is a person

The crown is a King -------- The crown is a Person

Existential quantification

Universal quantification makes statements about every object.

It is possible to make a statement about some object in the universe without naming it,by using an existential quantifier.

Example

"King John has a crown on his head"

□x Crown(x) ^ OnHead(x,John)

□x is pronounced "There exists an x such that.." or " For some x .."

Existential Quantification (∃):-

- An existential quantifier is used make a statement about some object in the universe without naming it.
- To say, for example :- that King John has a crown on his head, write ∃x crown (x) Λ OnHead (x, John).
- ∃x is pronounced " There exists an x such that.," or "For some x.."
- Consider the following sentence,

  $\exists_x$ crown (x) ⇒ OnHead (x,John)

- Applying the semantics says that at least one of the following assertion is true,

Richard the Lionheart is a crown	Λ Richard the Lionheart is on John's head
King John is Crown	Λ King John is on John's head
Richard's left leg is a crown	Λ Richard's left leg is on John's head
John's left leg is a crown	Λ John's left leg is on John's head
The crown is a crown	Λ The crown is on John's head

- Now an implication is true if both premise and conclusion are true, or if its premise is false.

Nested Quantifiers

More complex sentences are expressed using multiple quantifiers.

The following are the some cases of multiple quantifiers,

The simplest case where the quantifiers are of the same type.

For Example:- “Brothers are Siblings” can be written as

$$\forall_x \forall_y \text{ Brother } (x,y) \Rightarrow \text{sibling } (x,y)$$

Consecutive quantifiers of the same type can be written as one quantifier with several variables. For Example:- to say that siblinghood is a symmetric relationship as

$$\forall_{x,y} \text{ sibling } (x,y) \Leftrightarrow \text{sibling } (y,z)$$

3.2 THE WUMPUS WORLD

The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure

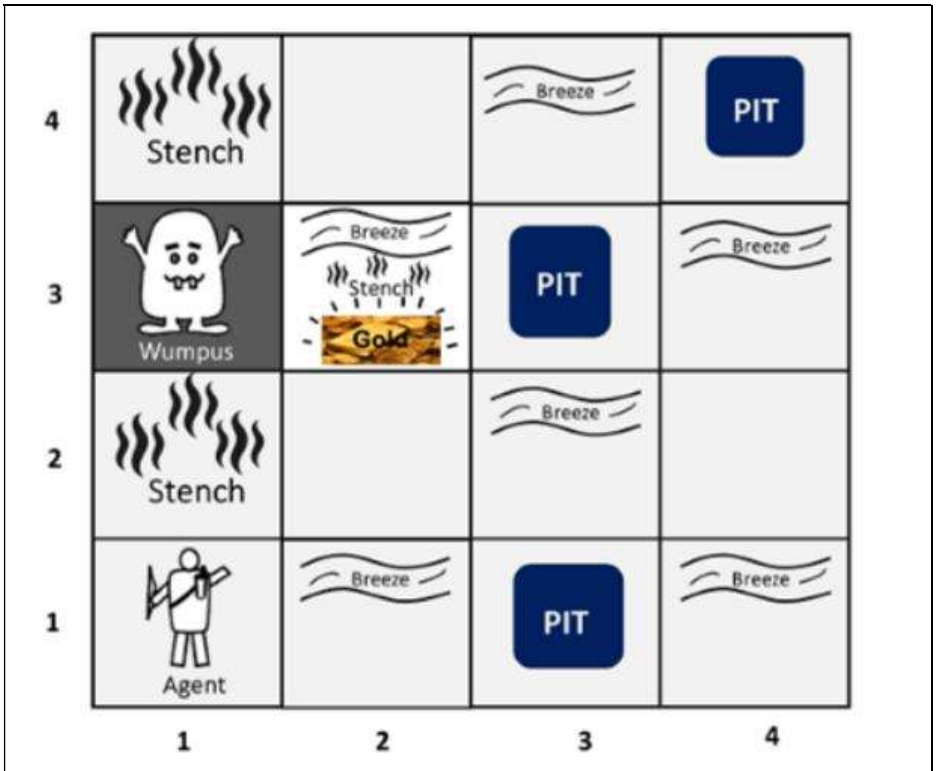


Figure 3.3 wumpus world

To specify the agent's task, we specify its percepts, actions, and goals. In the wumpus world, these are as follows:

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
- In the squares directly adjacent to a pit, the agent will perceive a breeze.
- In the square where the gold is, the agent will perceive a glitter.
- When an agent walks into a wall, it will perceive a bump.
- When the wumpus is killed, it gives out a woeful scream that can be perceived anywhere in the cave.
- The percepts will be given to the agent in the form of a list of five symbols; for example, if there is a stench, a breeze, and a glitter but no bump and no scream, the agent will receive the percept [Stench, Breeze, Glitter, None, None]. The agent cannot perceive its own location.
- Just as in the vacuum world, there are actions to go forward, turn right by 90°, and turn left by 90°. In addition, the action Grab can be used to pick up an object that is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits and kills the wumpus or hits the wall. The agent only has one arrow, so only the first Shoot action has any effect.

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

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

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

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

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

ASKVARS(∃ a BestAction(a, 5)),

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

For example:

∀t,s,g,m,c Percept ([s,Breeze,g,m,c],t) ⇒ Breeze(t),

∀t,s,b,m,c Percept ***([s,b,Glitter,m,c],t) ⇒ Glitter (t)***

These rules exhibit a trivial form of the reasoning process called perception. Simple "reflex" behavior can also be implemented by quantified implication sentences.

For example, we have $\forall t$ *Glitter (t)* $\Rightarrow$*BestAction(Grab, t)*.

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion

BestAction*(Grab, 5)*—that is, Grab is the right thing to do. For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$\forall s, t$ *At(Agent, s, t)* $\wedge$ *Breeze(t)* $\Rightarrow$ *Breezy(s).*

It is useful to know that a square is breezy because we know that the pits cannot move about. Notice that Breezy has no time argument. Having discovered which places are breezy (or smelly) and, very important, not breezy (or not smelly), the agent can deduce where the pits are (and where the wumpus is). first-order logic just needs one axiom:

$\forall s$ *Breezy(s)* $\Leftrightarrow \exists r$ *Adjacent (r, s)* $\wedge$ *Pit(r).*

3.3 SUBSTITUTION

Let us begin with universal quantifiers

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

Then it seems quite permissible to infer any of the following sentences:

King(John) $\wedge$ *Greedy(John)* $\Rightarrow$ *Evil(John)*
King(Richard) $\wedge$ *Greedy(Richard)* $\Rightarrow$ *Evil(Richard)*
King(Father (John)) $\wedge$ *Greedy(Father (John))* $\Rightarrow$ *Evil(Father (John)).*

The rule of **Universal Instantiation (UI for short)** says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.

Let *SUBST(θ,α)* denote the result of applying the substitution θ to the sentence α. Then the rule is written

$\forall v$ *α SUBST({v/g}, α)* for any variable v and ground term g.
For example, the three sentences given earlier are obtained with the substitutions *{x/John}, {x/Richard },* and *{x/Father (John)}.*

In the rule for Existential Instantiation, the variable is replaced by a single new constant symbol. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$\exists v$ *α SUBST({v/k}, α)* For example, from the sentence

∃x Crown(x) ∧OnHead(x, John) we can infer the sentence

Crown(C1) ∧OnHead(C1, John)

EXAMPLE

Suppose our knowledge base contains just the sentences

∀x King(x) ∧ Greedy(x) ⇒ Evil(x)

King(John)

Greedy(John) Brother (Richard, John)

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case,

{x/John} and {x/Richard }. We obtain

King(John) ∧ Greedy(John) ⇒ Evil(John)

King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard),

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences

King(John),

Greedy(John), and so on—as proposition symbols.

3.4 UNIFICATION

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called unification and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists: ***UNIFY(p, q)=θ*** where ***SUBST(θ, p)= SUBST(θ, q)***.

Suppose we have a query ***AskVars(Knows(John, x)):*** whom does John know? Answers can be found by finding all sentences in the knowledge base that unify with ***Knows(John, x).***

Here are the results of unification with four different sentences that might be in the knowledge base: ***UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}***

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

UNIFY(Knows(John, x), Knows(y, Mother (y))) = {y/John, x/Mother (John)}
UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail.

The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that ***Knows(x, Elizabeth)*** means “Everyone knows Elizabeth,” so we should be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by

standardizing apart one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in

Knows(x, Elizabeth) to x17 (a new variable name) without changing its meaning.

Now the unification will work

UNIFY(Knows(John, x), Knows(x17, Elizabeth)) = {x/Elizabeth, x17/John} UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier.

For example,

UNIFY(Knows(John, x), Knows(y, z)) could return
{y/John, x/z} or {y/John, x/John, z/John}.

The first unifier gives ***Knows(John, z)*** as the result of unification, whereas the second gives ***Knows(John, John).*** The second result could be obtained from the first by an additional substitution ***{z/John};*** we say that the first unifier is more general than the second, because it places fewer restrictions on the values of the variables. An algorithm for computing most general unifiers is shown in Figure.

The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed.

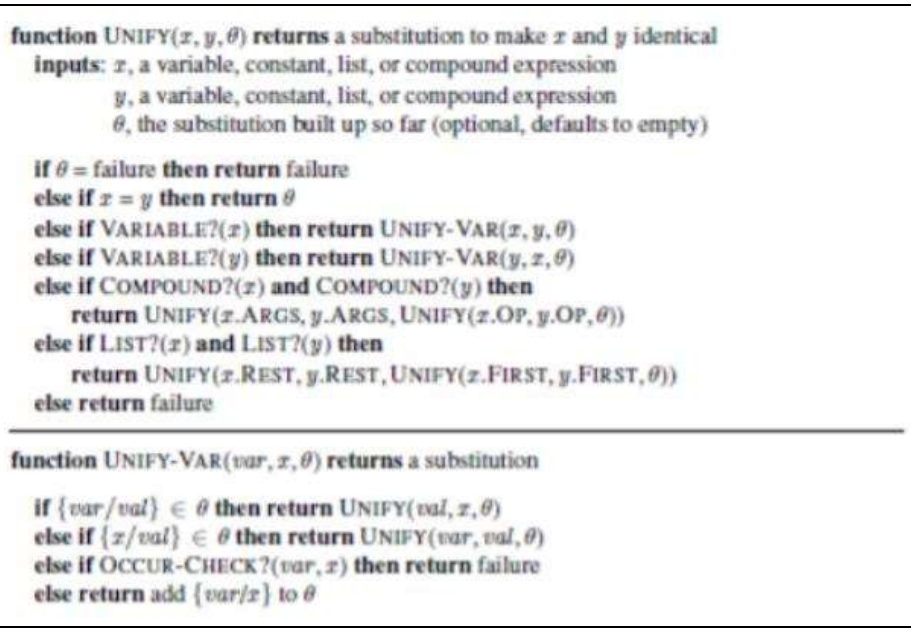


Figure 3.4 Recursively explore

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.nt

Definite clause: A clause which is a disjunction of literals with exactly one positive literal is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with at most one positive literal is known as horn clause. Hence all the definite clauses are horn clauses.

Example: (¬ p V ¬ q V k). It has only one positive literal k.

It is equivalent to p ∧ 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.

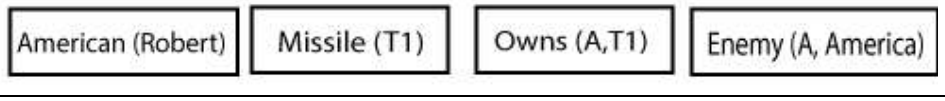


Figure 3.5

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).

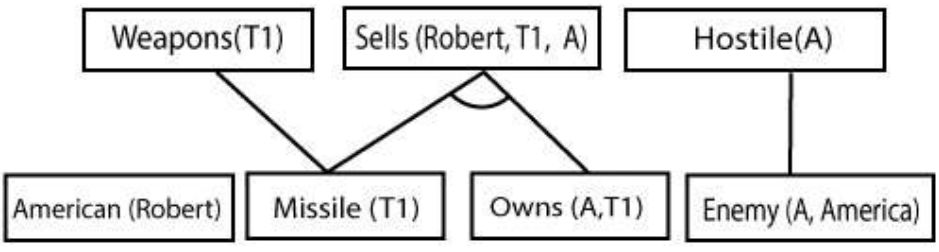


Figure 3.6

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.

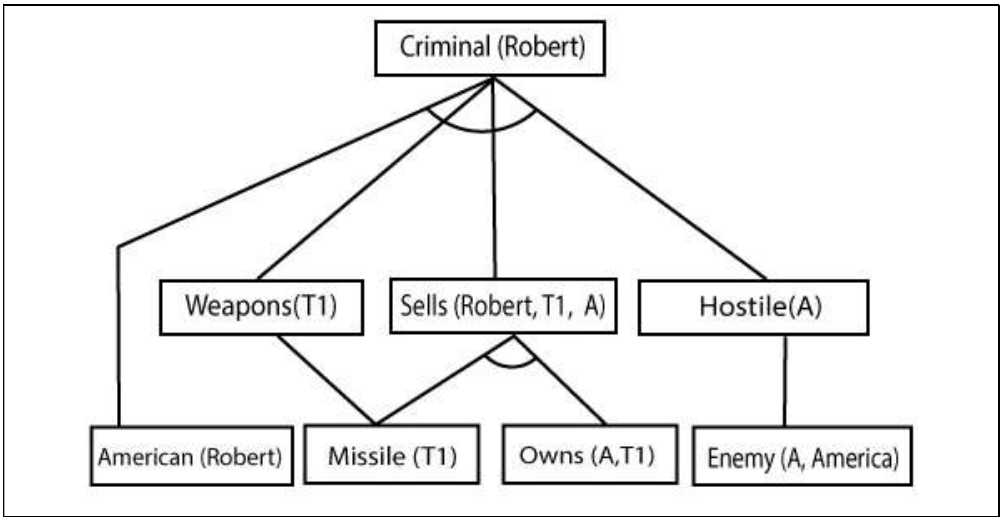


Figure 3.7

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.

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)

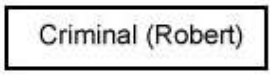
- Missile(T1)
- ?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) …(4)
- Missile(p) → Weapons (p) …(5)
- Enemy(p, America) →Hostile(p) …(6)
- Enemy (A, America) …(7)
- American (Robert). …(8)

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.

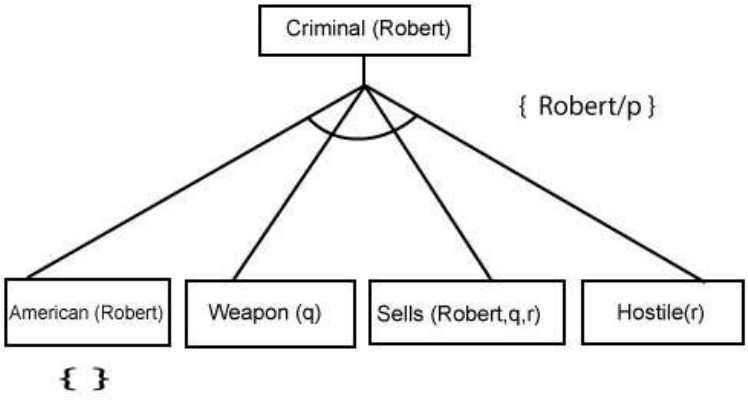


Figure 3.8

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.

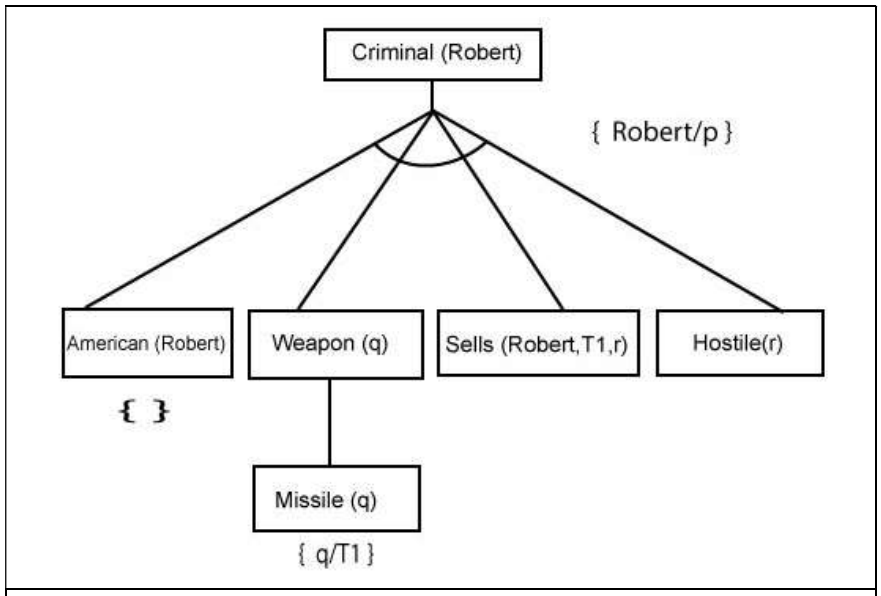


Figure 3.9

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.

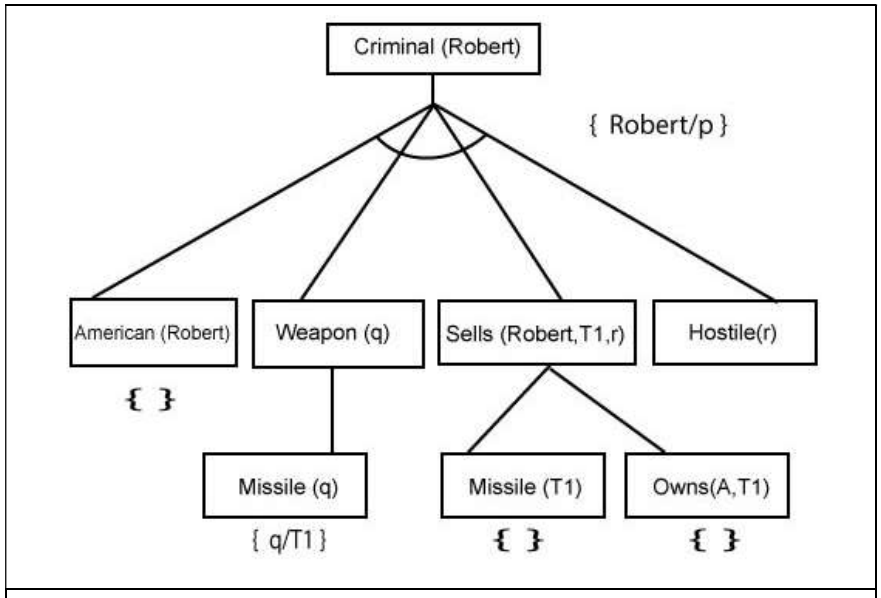


Figure 3.10

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.

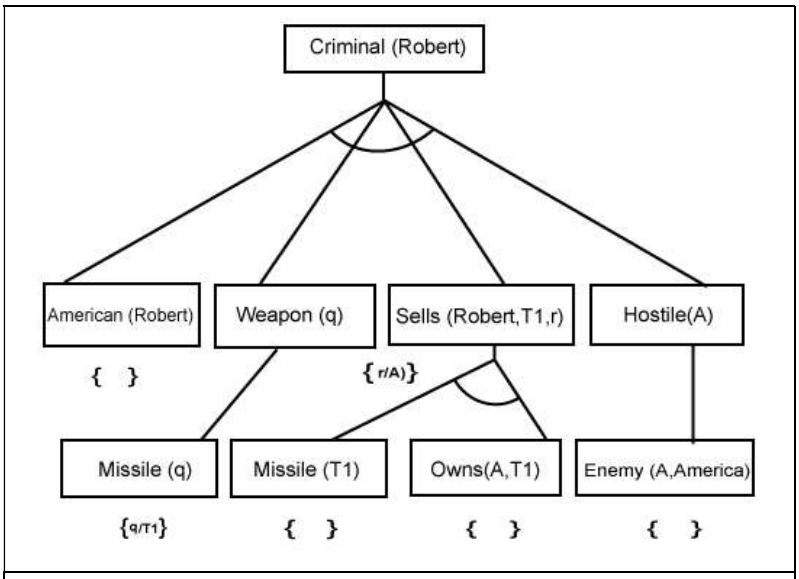


Figure 3.11

Suppose you have a production system with the FOUR rules: R1: IF A AND C then F R2: IF A AND E, THEN G R3: IF B, THEN E R4: R3: IF G, THEN D and you have four initial facts: A, B, C, D. PROVE A&B TRUE THEN D IS TRUE. Explain what is meant by "forward chaining", and show explicitly how it can be used in this case to determine new facts.

3.5 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.

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
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)

- **Eliminate all implication (→) and rewrite**

1. ∀x ¬ food(x) V likes(John, x)
2. food(Apple) ∧ food(vegetables)
3. ∀x ∀y ¬ [eats(x, y) ∧ ¬ killed(x)] V food(y)
4. eats (Anil, Peanuts) ∧ alive(Anil)
5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
6. ∀x¬ [¬ killed(x)] V alive(x)
7. ∀x ¬ alive(x) V ¬ killed(x)
8. likes(John, Peanuts).

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

  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀x ¬ eats(Anil, x) V eats(Harry, x)
  6. ∀x ¬killed(x)] V alive(x)
  7. ∀x ¬ alive(x) V ¬ killed(x)
  8. likes(John, Peanuts).

- **Rename variables or standardize variables**

  1. ∀x ¬ food(x) V likes(John, x)
  2. food(Apple) Λ food(vegetables)
  3. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
  4. eats (Anil, Peanuts) Λ alive(Anil)
  5. ∀w¬ eats(Anil, w) V eats(Harry, w)
  6. ∀g ¬killed(g)] V alive(g)
  7. ∀k ¬ alive(k) V ¬ killed(k)
  8. 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.

  1. ¬ food(x) V likes(John, x)
  2. food(Apple)
  3. food(vegetables)
  4. ¬ eats(y, z) V killed(y) V food(z)
  5. eats (Anil, Peanuts)
  6. alive(Anil)
  7. ¬ eats(Anil, w) V eats(Harry, w)
  8. killed(g) V alive(g)
  9. ¬ alive(k) V ¬ killed(k)
  10. likes(John, Peanuts).

- **Distribute conjunction Λ 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:

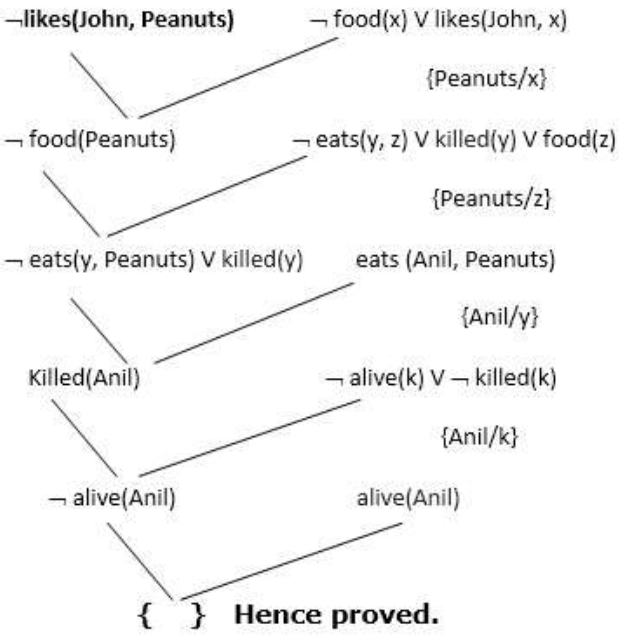


Figure 3.12

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.

Difference between Predicate Logic and Propositional Logic

Table 3.1

S. No.	Predicate logic	Propositional logic
1.	Predicate logic is a generalization of propositional logic that allows us to express and infer arguments in infinite models.	A preposition is a declarative statement that's either TRUE or FALSE (but not both).
2.	Predicate logic (also called predicate calculus and first-order logic) is an extension of propositional logic to formulas involving terms and predicates. The full predicate logic is undecidable	Propositional logic is an axiomatization of Boolean logic. Propositional logic is decidable, for example by the method of truth table.
3.	Predicate logic have variables	Propositional logic has variables. Parameters are all constant.
4.	A predicate is a logical statement that depends on one or more variables (not necessarily Boolean variables)	Propositional logic deals solely with propositions and logical connectives.
5.	Predicate logic there are objects, properties, functions (relations) are involved.	Proposition logic is represented in terms of Boolean variables and logical connectives.
6.	In predicate logic, we symbolize subject and predicate separately. Logicians often use lowercase letters to symbolize subjects (or objects) and upper case letter to symbolize predicates.	In propositional logic, we use letters to symbolize entire propositions. Propositions are statements of the form "x is y" where x is a subject and y is a predicate.
7.	Predicate logic uses quantifies such as universal quantifier ("∀"), the existential quantifier ("∃").	Prepositional logic has not qualifiers.
8.	**Example** Everything is a green as "∀x Green(x)" or "Something is blue as "∃ x Blue(x)".	**Example** Everything is a green as "G" or "Something is blue as "B(x)".

3.6 KNOWLEDGE REPRESENTATION ONTOLOGICAL ENGINEERING

Concepts such as Events, Time, Physical Objects, and Beliefs— that occur in many different domains. Representing these abstract concepts is sometimes called ontological engineering.

UNIT V: Knowledge Representation: Ontological Engineering, Categories and Objects, Events, Mental Objects and Model Logic, Reasoning Systems for Categories, Reasoning with Default Information. Quantifying Uncertainty: Acting under Uncertainty, Basic Probability Notation, and Inference Using Full Joint Distributions, Independence, Bayes' Rule and Its Use.

KNOWLEDGE REPRESENTATION ONTOLOGICAL ENGINEERING

Concepts such as Events, Time, Physical Objects, and Beliefs— that occur in many different domains. Representing these abstract concepts is sometimes called ontological engineering.

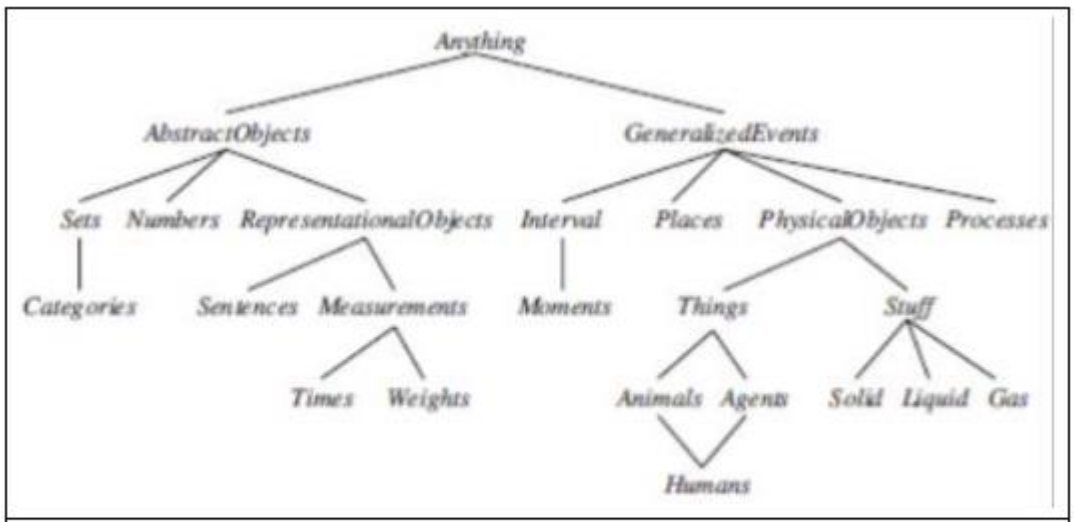


Figure 3.13 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint; a human is both an animal and an agent, for example.

The general framework of concepts is called an upper ontology because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure .

Categories and Objects

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, much reasoning takes place at the level of categories. For example, a shopper would normally have the goal of buying a basketball, rather than a particular basketball such as BB9 There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate **Basketball (b),** or we can reify1 the category as an object, Basketballs.

We could then say ***Member(b, Basketballs),*** which we will abbreviate ***as b∈Basketballs,*** to say that b is a member of the category of basketballs. We say Subset(Basketballs, Balls), abbreviated as Basketballs ⊂ Balls, to say that Basketballs is a subcategory of Balls. Categories serve to organize and simplify the knowledge base through inheritance. If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit, then we can infer that every apple is edible. We say that the individual apples inherit the property of edibility, in this case from their membership in the Food category. First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members. Here are some types of facts, with examples of each:

• An object is a member of a category.

BB9 ∈ Basketballs

- A category is a subclass of another category. ***Basketballs ⊂ Balls***
- All members of a category have some properties.

(x∈ Basketballs) ⇒ Spherical (x)

- Members of a category can be recognized by some properties.
  Orange(x) ∧ Round (x) ∧ Diameter(x)=9.5 ∧ x∈ Balls ⇒ x∈ Basketballs
- A category as a whole has some properties.

Dogs ∈ Domesticated Species

Notice that because Dogs is a category and is a member of Domesticated Species, the latter must be a category of categories. Categories can also be defined by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:

x∈ Bachelors ⇔ Unmarried(x) ∧ x∈ Adults ∧ x∈ Males

Physical Composition

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

PartOf (Bucharest, Romania)
PartOf (Romania, EasternEurope)
PartOf(EasternEurope, Europe)
PartOf (Europe, Earth)
The PartOf relation is transitive and reflexive; that is,
PartOf (x, y) ∧PartOf (y, z) ⇒PartOf (x, z)
PartOf (x, x)
Therefore, we can conclude PartOf (Bucharest, Earth).
For example, if the apples are Apple1, Apple2, and Apple3, then
BunchOf ({Apple1,Apple2,Apple3})
denotes the composite object with the three apples as parts (not elements). We can define ***BunchOf in terms of the PartOf relation. Obviously, each element of s is part of***

BunchOf (s): ∀x x∈ s ⇒PartOf (x, BunchOf (s)) Furthermore, BunchOf (s) is the smallest object satisfying this condition. In other words, BunchOf (s) must be part of any object that has all the elements of s as parts:

∀y [∀x x∈ s ⇒PartOf (x, y)] ⇒PartOf (BunchOf (s), y)

Measurements

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called measures. ***Length(L1)=Inches(1.5)=Centimeters(3.81)***

Conversion between units is done by equating multiples of one unit to another: ***Centimeters(2.54 ×d)=Inches(d)***

Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents. Measures can be used to describe objects as follows:

Diameter (Basketball12)=Inches(9.5)

ListPrice(Basketball12)=$(19)

d∈ Days ⇒ Duration(d)=Hours(24)

Time Intervals

Event calculus opens us up to the possibility of talking about time, and time intervals. We will consider two kinds of time intervals: moments and extended intervals. The distinction is that only moments have zero duration:

Partition({Moments,ExtendedIntervals}, Intervals)

i∈Moments⇔Duration(i)=Seconds(0)

The functions Begin and End pick out the earliest and latest moments in an interval, and the function Time delivers the point on the time scale for a moment.

The function Duration gives the difference between the end time and the start time.

Interval (i) ⇒Duration(i)=(Time(End(i)) − Time(Begin(i)))
Time(Begin(AD1900))=Seconds(0)

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

Time(End(AD2001))=Seconds(3218860800)
Duration(AD2001)=Seconds(31536000)

Two intervals Meet if the end time of the first equals the start time of the second. The complete set of interval relations, as proposed by Allen (1983), is shown graphically in Figure 12.2 and logically below:

Meet(i,j) ⇔ End(i)=Begin(j)

Before(i,j) ⇔ End(i) < Begin(j)

After (j,i) ⇔ Before(i, j)

During(i,j) ⇔ Begin(j) < Begin(i) < End(i) < End(j)

Overlap(i,j) ⇔ Begin(i) < Begin(j) < End(i) < End(j)

Begins(i,j) ⇔ Begin(i) = Begin(j)

Finishes(i,j) ⇔ End(i) = End(j)

Equals(i,j) ⇔ Begin(i) = Begin(j) ∧ End(i) = End(j)

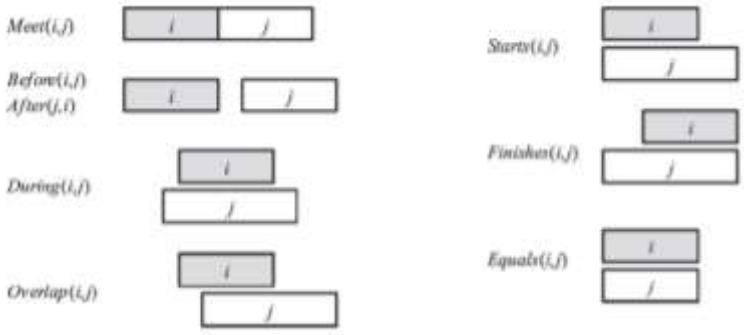


Figure 3.14 Predicates on time intervals

3.7 EVENTS

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 described as **E1 ∈Flyings∧ Flyer (E1, Shankar) ∧ Origin(E1, SF) ∧ Destination (E1,DC)** we can define an alternative three-argument version of the category of flying events and say ***E1 ∈Flyings(Shankar, SF,DC)*** We then use ***Happens(E1, i)*** to say that the event E1 took place over the time interval i, and we say the same thing in functional form with Extent(***E1)=i.*** We represent time intervals by a (start, end) pair of times; that is, i = (t1, t2) is the time interval that starts at t1 and ends at t2. The complete set of predicates for one version of the event calculus is T(f, t) Fluent f is true at time t Happens(e, i) Event e happens over the time interval i Initiates(e, f, t) Event e causes fluent f to start to hold at time t Terminates(e, f, t) Event e causes fluent f to cease to hold at time t Clipped(f, i) Fluent f ceases to be true at some point during time interval i Restored (f, i) Fluent f becomes true sometime during time interval i We assume a distinguished event, Start, that describes the initial state by saying which fluents are initiated or terminated at the start time. We define T by saying that a fluent holds at a point in time if the fluent was initiated by an event at some time in the past and was not made false (clipped) by an intervening event. A fluent does not hold if it was terminated by an event and not made true (restored) by another event. Formally, the axioms are:

Happens(e, (t1, t2)) ∧Initiates(e, f, t1) ∧¬Clipped(f, (t1, t)) ∧ t1 < t ⇒T(f, t)Happens(e, (t1, t2)) ∧ Terminates(e, f, t1)∧¬Restored (f, (t1, t)) ∧t1 < t ⇒¬T(f, t)

where Clipped and Restored are defined by Clipped(f, (t1, t2)) ⇔∃ e, t, t3 Happens(e, (t, t3))∧ t1 ≤ t < t2 ∧ Terminates(e, f, t) Restored (f, (t1, t2)) ⇔∃ e, t, t3 Happens(e, (t, t3)) ∧ t1 ≤ t < t2 ∧ Initiates(e, f, t)

3.8 MENTAL EVENTS AND MENTAL OBJECTS

What we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction. We will be happy just to be able to conclude that mother knows whether or not she is sitting.

We begin with the propositional attitudes that an agent can have toward mental objects: attitudes such as Believes, Knows, Wants, Intends, and Informs. The difficulty is that these attitudes do not behave like "normal" predicates.

For example, suppose we try to assert that Lois knows that Superman can fly: **Knows (Lois, CanFly(Superman))** One minor issue with this is that we normally think of CanFly(Superman) as a sentence, but here it appears as a term. That issue can be patched up just be reifying **CanFly(Superman);** making it a fluent. A more serious problem isthat, if it is true that Superman is Clark Kent, then we must conclude that Lois knows that Clark can fly: **(Superman = Clark) ∧Knows(Lois, CanFly(Superman)) |= Knows(Lois, CanFly (Clark))** Modal logic is designed to address this problem. Regular logic is concerned with a single modality, the modality of truth, allowing us to express "P is true." Modal logic includes special modal operators that take sentences (rather than terms) as arguments.

For example, "A knows P" is represented with the notation KAP, where K is the modal operator for knowledge. It takes two arguments, an agent (written as the subscript) and a sentence. The syntax of modal logic is the same as first-order logic, except that sentences can also be formed with modal operators. In first-order logic a model contains a set of objects and an interpretation that maps each name to the appropriate object, relation, or function. In modal logic we want to be able to consider both the possibility that Superman's secret identity is Clark and that it isn't. Therefore, we will need a more complicated model, one that consists of a collection of possible worlds rather than just one true world. The worlds are connected in a graph by accessibility relations, one relation for each modal operator. We say that world w1 is accessible from world w0 with respect to the modal operator KA if everything in w1 is consistent with what A knows in w0, and we write this as **Acc(KA,w0,w1).** In diagrams such as Figure 12.4 we show accessibility as an arrow between possible worlds. In general, a knowledge atom KAP is true in world w if and only if P is true in every world accessible from w. The truth of more complex sentences is derived by recursive application of this rule and the normal rules of first-order logic. That means that modal logic can be used to reason about nested knowledge sentences: what one agent knows about another agent's knowledge. For example, we can say that, even though Lois doesn't know whether Superman's secret identity is Clark Kent, she does know that Clark knows: **KLois [KClark Identity(Superman, Clark)**

∨KClark¬Identity(Superman, Clark)] Figure 3.15 shows some possible worlds for this domain, with accessibility relations for Lois and Superman

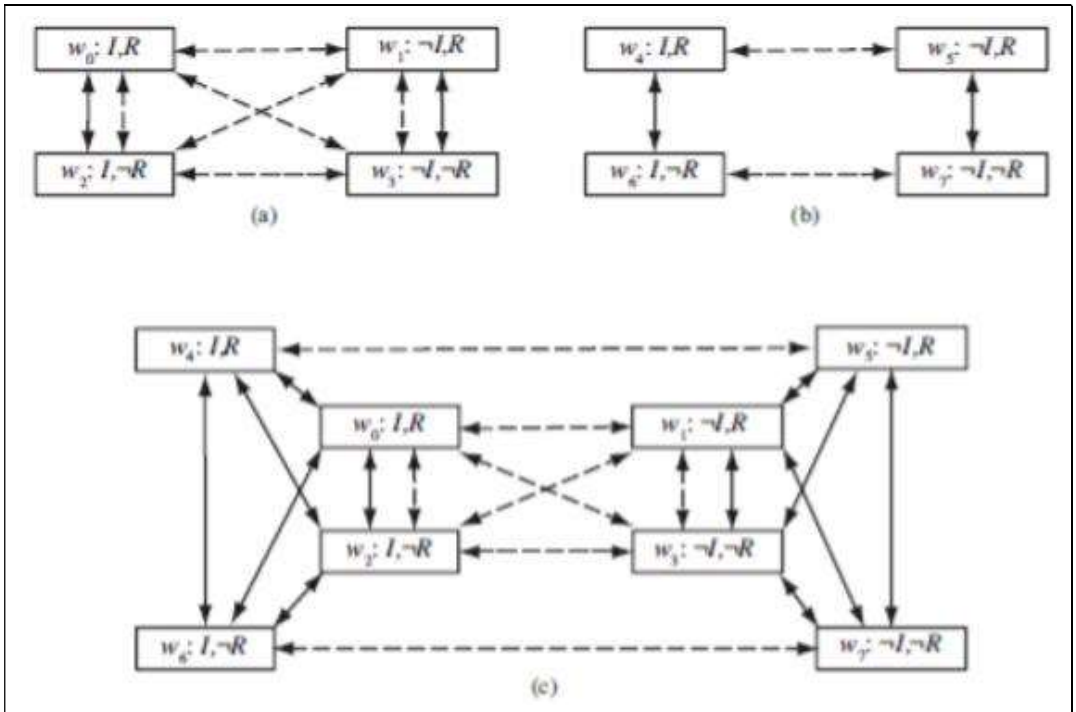


Figure 3.15

In the TOP-LEFT diagram, it is common knowledge that Superman knows his own identity, and neither he nor Lois has seen the weather report. So in w0 the worlds w0 and w2 are accessible to Superman; maybe rain is predicted, maybe not. For Lois all four worlds are accessible from each other; she doesn't know anything about the report or if Clark is Superman. But she does know that Superman knows whether he is Clark, because in every world that is accessible to Lois, either Superman knows I, or he knows ¬I. Lois does not know which is the case, but either way she knows Superman knows. In the TOP-RIGHT diagram it is common knowledge that Lois has seen the weather report. So in w4 she knows rain is predicted and in w6 she knows rain is not predicted. Superman does not know the report, but he knows that Lois knows, because in every world that is accessible to him, either she knows R or she knows ¬ R. In the BOTTOM diagram we represent the scenario where it is common knowledge that Superman knows his identity, and Lois might or might not have seen the weather report. We represent this by combining the two top scenarios, and adding arrows to show that Superman does not know which scenario actually holds. Lois does know, so we don't need to add any arrows for her. In w0 Superman still knows I but not R, and now he does not know whether Lois knows R. From what Superman knows, he might be in w0 or w2, in which case Lois does not know whether R is true, or he could be in w4, in which case she knows R, or w6, in which case she knows ¬R.

3.9 REASONING SYSTEMS FOR CATEGORIES

This section describes systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: semantic networks provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership; and description logics provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

3.10 SEMANTIC NETWORKS

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled links. For example, Figure 12.5 has a Member Of link between Mary and Female Persons, 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. 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. 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—that is, ∀x x∈ Persons ⇒ Legs(x, 2) The semantic network notation makes it convenient to perform inheritance reasoning. For example, by virtue of being a person, Mary inherits the property of having two legs. Thus, to find out how many legs Mary has, the inheritance algorithm followsthe MemberOf link from Mary to the category she belongs to, and then follows SubsetOf links up the hierarchy until it finds a category for which there is a boxed Legs link—in this case, the Persons category.

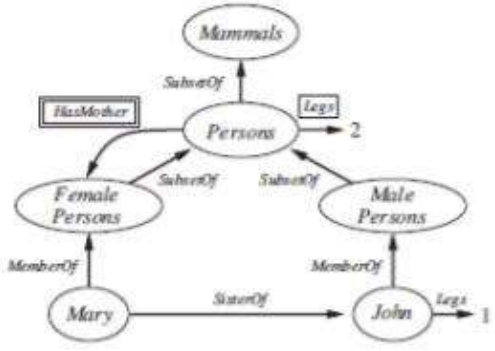


Figure 3.16 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

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. The drawback of semantic network notation, compared to first-order logic: the fact

that links between bubbles represent only binary relations. For example, the sentence Fly(Shankar, NewYork, NewDelhi, Yesterday) cannot be asserted directly in a semantic network. Nonetheless, we can obtain the effect of n-ary assertions by reifying the proposition itself as an event belonging to an appropriate event category. Figure 12.6 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts. One of the most important aspects of semantic networks is their ability to represent.

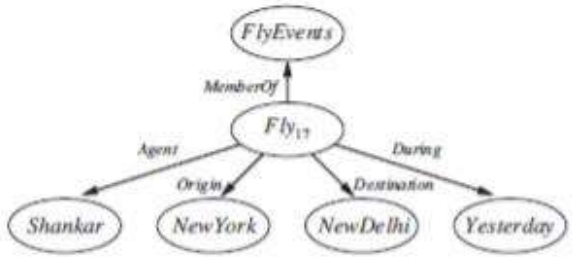


Figure 3.17 A fragment of a semantic network showing the representation of the logical assertion *Fly(Shankar, NewYork, NewDelhi, Yesterday)*

One of the most important aspects of semantic networks is their ability to represent default values for categories. Examining Figure 3.6 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information

EXAMPLE 1

Consider the crime example. The sentences in CNF are

¬American(x) V ¬Weapon(y) V ¬Sells(x, y, z) V ¬Hostile(z) V Criminal (x)

¬Missile(x) V ¬Owns(Nono, x) V Sells(West, x, Nono)

¬Enemy(x, America) V Hostile(x)

¬Missile(x) VWeapon(x)

Owns(Nono, M1) Missile(M1)

American(West) Enemy(Nono, America)

We also include the negated goal ¬ Criminal (West). The resolution proof is shown in Figure 3.18.

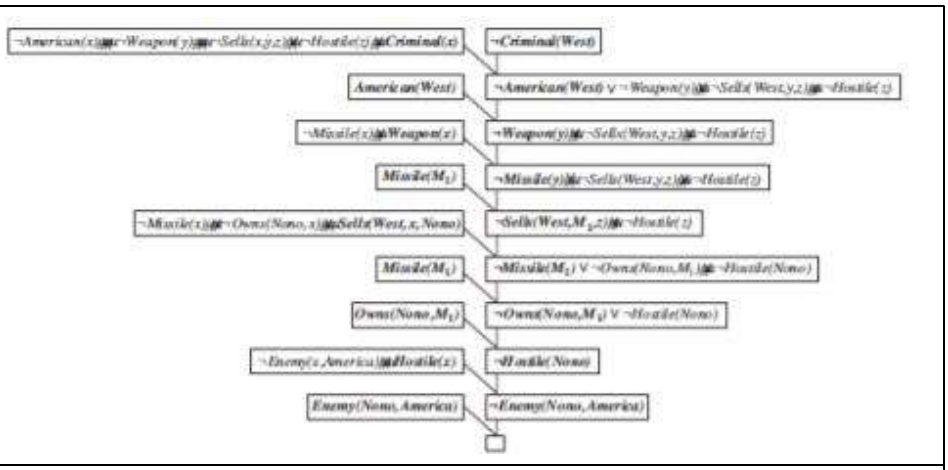


Figure 3.18 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond exactly to the consecutive values of the goals variable in the backward-chaining algorithm of Figure. This is because we always choose to resolve with a clause whose positive literal unified with the left most literal of the "current" clause on the spine; this is exactly what happens in backward chaining. Thus, backward chaining is just a special case of resolution with a particular control strategy to decide which resolution to perform next.

EXAMPLE 2

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is a follows:

Everyone who love all animals is loved by someone.
Anyone who kills an animals is loved by no one.
Jack loves all animals.

Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat? First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x,y)] \Rightarrow [\exists y \text{ Loves}(y,x)]$
B. $\forall x [\exists x \text{ Animal}(z) \wedge \text{Kills}(x,z)] \Rightarrow [\forall y \text{ Loves}(y,x)]$
C. $\forall x \text{ Animals}(x) \Rightarrow \text{Loves}(\text{Jack}, x)$
D. Kills (Jack, Tuna) V Kills (Curiosity, Tuna)
E. Cat (Tuna)
F. $\forall x \text{ Cat}(x) \Rightarrow \text{Animal}(x)$
$\neg$G. $\neg$Kills(Curiosity, Tuna)

Now we apply the conversion procedure to convert each sentence to CNF:

A1. Animal(F(x)) v Loves (G(x),x)

A2. ¬Loves(x,F(x)) v Loves (G(x),x)

B. ¬Loves(y,x) v ¬ Animal(z) V ¬ Kills(x,z)

C. ¬Animal(x) v Loves(Jack, x)

D. Kills(Jack, Tuna) v Kills (Curiosity, Tuna)

E. Cat (Tuna)

F. ¬Cat(x) v Animal (x)

¬G. ¬Kills (Curiosity, Tuna)

The resolution proof that Curiosity kills the cat is given in Figure. In English, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore Curiosity killed the cat.

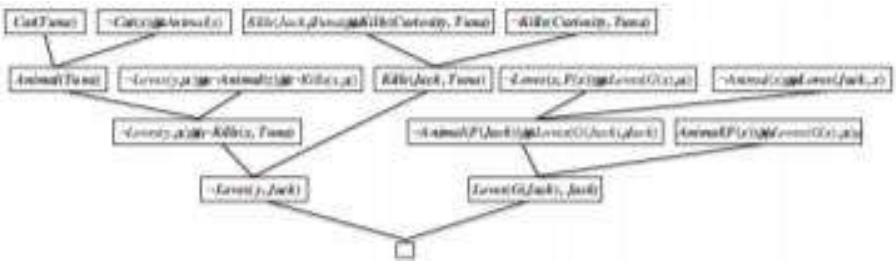


Figure 3.19 A resolution proof that Curiosity killed that Cat. Notice then use of factoring in the derivation of the clause Loves(G(Jack), Jack). Notice also in the upper right, the unification of Loves(x,F(x)) and Loves(Jack,x) can only succeed after the variables have been standardized apart

The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, such as "Who killed the cat?" Resolution can do this, but it takes a little more work to obtain the answer. The goal is ∃w Kills (w, Tuna), which, when negated become ¬Kills (w, Tuna) in CNF, Repeating the proof in Figure with the new negated goal, we obtain a similar proof tree, but with the substitution {w/Curiosity} in one of the steps. So, in this case, finding out who killed the cat is just a matter of keeping track of the bindings for the query variables in the proof.

EXAMPLE 3

1. All people who are graduating are happy.

2. All happy people smile.
3. Someone is graduating.
4. Conclusion: Is someone smiling?

Solution

Convert the sentences into predicate Logic

1. ∀x graduating(x) -> happy(x)
2. ∀x happy(x) -> smile(x)
3. ∃x graduating(x)
4. ∃x smile(x)

Convert to clausal form

(i) Eliminate the → sign

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. -∃x smile(x)

(ii) Reduce the scope of negation

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(x) vsmile(x)
3. ∃x graduating(x)
4. ∀x¬ smile(x)

(iii) Standardize variable apart

1. ∀x – graduating(x) vhappy(x)
2. ∀x – happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀x¬ smile(w)

(iv) Move all quantifiers to the left

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)
3. ∃x graduating(z)
4. ∀w¬ smile(w)

(v) Eliminate ∃

1. ∀x¬ graduating(x) vhappy(x)
2. ∀x¬ happy(y) vsmile(y)

3. graduating(name1)

4. ∀w¬ smile(w)

(vi) Eliminate∀

1. ¬graduating(x) vhappy(x)

2. ¬happy(y) vsmile(y)

3. graduating(name1)

4. ¬smile(w)

(vii) Convert to conjunct of disjuncts form

(viii) Make each conjunct a separate clause.

(ix) Standardize variables apart again.

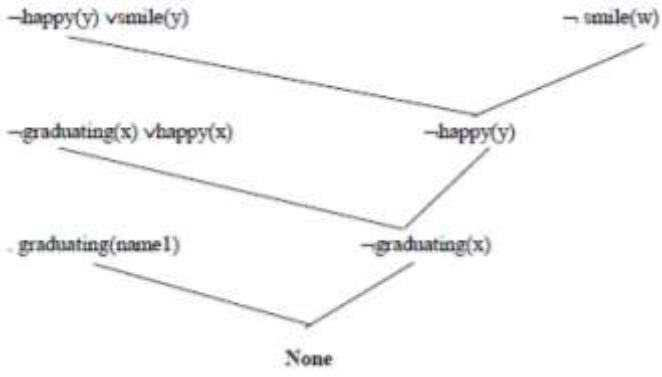


Figure 3.20 Standardize variables

Thus, we proved someone is smiling.

EXAMPLE 4

Explain the unification algorithm used for reasoning under predicate logic with an example. Consider the following facts

a. Team India

b. Team Australia

c. Final match between India and Australia

d. India scored 350 runs, Australia scored 350 runs, India lost 5 wickets, Australia lost 7 wickets.

f. If the scores are same the team which lost minimum wickets wins the match.

Represent the facts in predicate, convert to clause form and prove by resolution “India wins the match”.

Solution

Convert into predicate Logic

(a) team(India)

(b) team(Australia)

(c) team(India) ^ team(Australia) → final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x team(x) ^ wins(x) → score(x,mat_runs)

(f) ∃xy score(x,equal(y)) ^ wicket(x,min) ^ final_match(x,y) → win(x)

Convert to clausal form

(i) Eliminate the → sign

(a) team(India)

(b) team(Australia)

(c) ¬(team(India) ^ team(Australia) v final_match(India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ (team(x) ^ wins(x)) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) ^ wicket(x,min) ^final_match(x,y)) vwin(x)

(ii) Reduce the scope of negation

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match(India, Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ∃x¬ team(x) v ¬wins(x) vscore(x,max_runs))

(f) ∃xy¬ (score(x,equal(y)) v ¬ wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

(iii) Standardize variables apart

(iv) Move all quantifiers to the left

(v) Eliminate ∃

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350) ^ score(Australia,350) ^ wicket(India,5) ^ wicket(Australia,7)

(e) ¬team(x) v ¬wins(x) vscore(x, max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v-final_match(x,y)) vwin(x)

(vi) Eliminate∀

(vii) Convert to conjunct of disjuncts form.

(viii) Make each conjunct a separate clause.

(a) team(India)

(b) team(Australia)

(c) ¬team(India) v ¬team(Australia) v final_match (India,Australia)

(d) score(India,350)
Score(Australia,350)
Wicket(India,5)
Wicket(Austrialia,7)

(e) ¬team(x) v ¬wins(x) vscore(x,max_runs))

(f) ¬score(x,equal(y)) v¬wicket(x,min_wicket) v¬final_match(x,y)) vwin(x)

(ix) Standardize variables apart again

To prove:win(India)

Disprove:¬win(India)

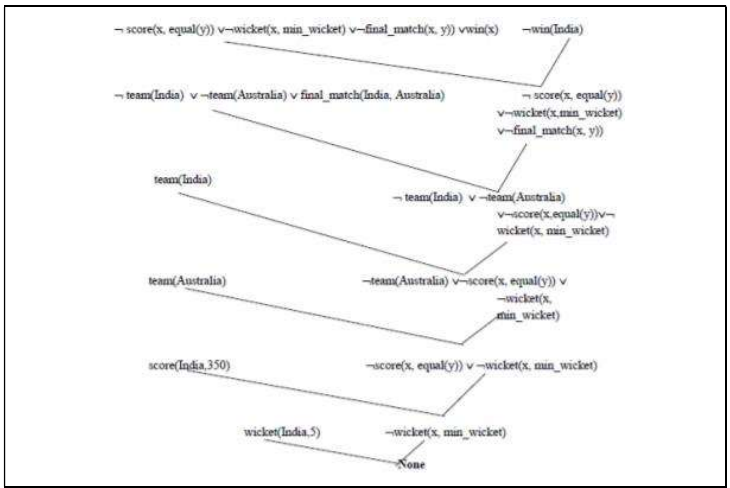


Figure 3.21 Standardize variables

Thus, proved India wins match.

EXAMPLE 5

Problem 3

Consider the following facts and represent them in predicate form:

F1. There are 500 employees in ABC company.

F2. Employees earning more than Rs. 5000 per tax.

F3. John is a manger in ABC company.

F4. Manger earns Rs. 10,000.

Convert the facts in predicate form to clauses and then prove by resolution: “John pays tax”.

Solution

Convert into predicate Logic

1. company(ABC) ^employee(500,ABC)
2. ∃x company(ABC) ^employee(x,ABC) ^ earns(x,5000)→pays(x,tax)
3. manager(John,ABC)
4. ∃x manager(x, ABC)→earns(x,10000)

Convert to clausal form

(i) Eliminate the → sign

1. company(ABC) ^employee(500,ABC)
2. ∃x¬(company(ABC) ^employee(x,ABC) ^ earns(x,5000)) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(ii) Reduce the scope of negation

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(iii) Standardize variables apart

1. company(ABC) ^employee(500,ABC)
2. ∃x¬ company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ∃x¬ manager(x, ABC) v earns(x,10000)

(iv) Move all quantifiers to the left

(v) Eliminate ∃

1. company(ABC) ^employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

(vi) Eliminate∀

(vii) Convert to conjunct of disjuncts form

(viii) Make each conjunct a separate clause.

1. (a) company(ABC)
   (b) employee(500,ABC)
2. ¬company(ABC) v ¬employee(x,ABC) v¬earns(x,5000) v pays(x,tax)
3. manager(John,ABC)
4. ¬manager(x, ABC) v earns(x,10000)

(ix) Standardize variables apart again.

Prove:pays(John,tax)

Disprove:¬pays(John,tax)

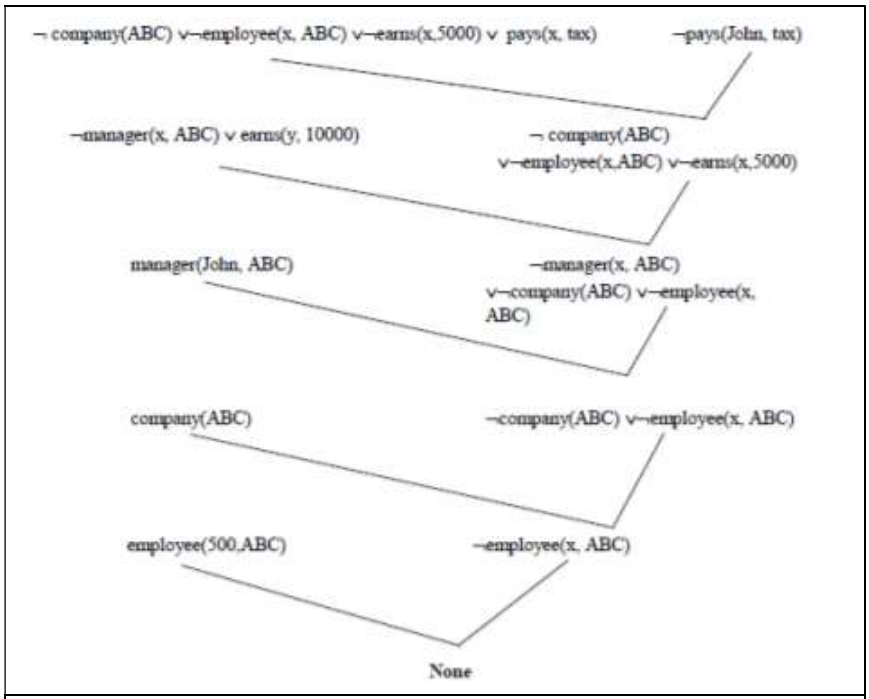


Figure 3.22 Standardize variables

Thus, proved john pays tax.

EXAMPLE 6 & EXAMPLE 7

Problem 4

If a perfect square is divisible by a prime p then it is also divisible by square of p.

Every perfect square is divisible by some prime.

36 is a perfect square.

Convert into predicate Logic

1. ∀xyperfect_sq(x) ^ prime(h) ^divideds(x,y) → divides(x,square(y))

2. ∀x∃y perfect)sq(x) ^prime(y) ^divides(x,y)
3. perfect_sq(36)

Problem 5

1. Marcus was a a man
   man(Marcus)
2. Marcus was a Pompeian
   Pompeian(Marcus)
3. All Pompeians were Romans
   ∀x (Pompeians(x) → Roman(x))
4. Caesar was ruler
   Ruler(Caesar)
5. All Romans were either loyal to Caesar or hated him.
   ∃x (Roman(x) → loyalto(x,Caesar) v hate(x,Caesar))
6. Everyone is loyal to someone
   ∀x ∃y (person(x) → person(y) ^ loyalto(x,y))
7. People only try to assassinate rulers they are not loyal to
   ∀x ∃y (person(x) ^ ruler(y)^ tryassassinate(x,y) → -loyalto(x,y))
8. Marcus tried to assassinate Caesar
   tryassassinate(Marcus, Caesar)
9. All men are persons
   ∀x (max(x) → person(x))

Example

Trace the operation of the unification algorithm on each of the following pairs of literals:

i) f(Marcus) and f(Caesar)

ii) f(x) and f(g(y))

iii) f(marcus,g(x,y)) and f(x,g(Caesar, Marcus))

In propositional logic it is easy to determine that two literals can not both be true at the same time. Simply look for L and ~L. In predicate logic, this matching process is more complicated, since bindings of variables must be considered.

For example man (john) and man(john) is a contradiction while man (john) and man(Himalayas) is not. Thus in order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical. There is a recursive procedure that does this matching. It is called Unification algorithm.

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list. For example we can have literals as

(tryassassinate Marcus Caesar)

(tryassassinate Marcus (ruler of Rome))

To unify two literals, first check if their first elements re same. If so proceed. Otherwise they can not be unified. For example the literals

(try assassinate Marcus Caesar)

(hate Marcus Caesar)

Can not be Unfied. The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are :

i) Different constants, functions or predicates can not match, whereas identical ones can.

ii) A variable can match another variable, any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).

iii) The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent (a substitution y for x written as y/x).

The Unification algorithm is listed below as a procedure UNIFY (L1, L2). It returns a list representing the composition of the substitutions that were performed during the match. An empty list NIL indicates that a match was found without any substitutions. If the list contains a single value F, it indicates that the unification procedure failed.

UNIFY (L1, L2)

1. if L1 or L2 is an atom part of same thing do

(a) if L1 or L2 are identical then return NIL

(b) else if L1 is a variable then do

(i) if L1 occurs in L2 then return F else return (L2/L1)

© else if L2 is a variable then do

(i) if L2 occurs in L1 then return F else return (L1/L2)

else return F.

2. If length (L!) is not equal to length (L2) then return F.

3. Set SUBST to NIL

(at the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2).

4. For I = 1 to number of elements in L1 do

i) call UNIFY with the ith element of L1 and I'th element of L2, putting the result in S

ii) if S = F then return F

iii) if S is not equal to NIL then do

(A) apply S to the remainder of both L1 and L2

(B) SUBST := APPEND (S, SUBST) return SUBST.

Consider a knowledge base containing just two sentences: P(a) and P(b). Does this knowledge base entail ∀ x P(x)? Explain your answer in terms of models.

The knowledge base does not entail $\forall x\ P(x)$. To show this, we must give a model where $P(a)$ and $P(b)$ but $\forall x\ P(x)$ is false. Consider any model with three domain elements, where a and b refer to the first two elements and the relation referred to by P holds only for those two elements.

Is the sentence ∃ x, y x = y valid? Explain

The sentence $\exists x, y\ x=y$ is valid. A sentence is valid if it is true in every model. An existentially quantified sentence is true in a model if it holds under any extended interpretation in which its variables are assigned to domain elements. According to the standard semantics of FOL as given in the chapter, every model contains at least one domain element, hence, for any model, there is an extended interpretation in which x and y are assigned to the first domain element. In such an interpretation, $x=y$ is true.

What is ontological commitment (what exists in the world) of first order logic? Represent the sentence "Brothers are siblings" in first order logic?

Ontological commitment means what assumptions language makes about the nature if reality. Representation of "Brothers are siblings" in first order logic is ∀ x, y [Brother (x, y) ⇔ Siblings (x, y)]

Differentiate between propositional and first order predicate logic?

Following are the comparative differences versus first order logic and propositional logic.

1) Propositional logic is less expressive and do not reflect individual object`s properties explicitly. First order logic is more expressive and can represent individual object along with all its properties.

2) Propositional logic cannot represent relationship among objects whereas first order logic can represent relationship.

3) Propositional logic does not consider generalization of objects where as first order logic handles generalization.

4) Propositional logic includes sentence letters (A, B, and C) and logical connectives, but not quantifier. First order logic has the same connectives as

propositional logic, but it also has variables for individual objects, quantifier, symbols for functions and symbols for relations.

Represent the following sentence in predicate form:

"All the children like sweets"

$\forall x$ child(x) $\cap$ sweet(y) $\cap$ likes (x,y).

Illustrate the use of first order logic to represent knowledge. The best way to find usage of First order logic is through examples. The examples can be taken from some simple domains. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge. Assertions and queries in first-order logic Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions. For example, we can assert that John is a king and that kings are persons: Where KB is knowledge base. TELL(KB, $\forall x$ King(x) => Person(x)). We can ask questions of the knowledge base using AS K. For example, returns true. Questions asked using ASK are called queries or goals ASK(KB, Person(John)) Will return true. (ASK KBto find whether Jon is a king) ASK (KB, $\exists x$ person(x)) The kinship domain The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William7' and rules such as "One's grandmother is the mother of one's parent." Clearly, the objects in our domain are people. We will have two unary predicates, Male and Female. Kinship relations-parenthood, brotherhood, marriage, and so on-will be represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle. We will use functions for Mother and Father.

What are the characteristics of multi agent systems?

Each agent has just incomplete information and is restricted in is capabilities. The system control is distributed. Data is decentralized. Computation is asynchronous Multi agent environments are typically open ad have no centralized designer. Multi agent environments provide an infrastructure specifying communication and interaction protocols. Multi agent environments have agents that are autonomous and distributed, and may be self interested or cooperative.

Acting under Uncertainty

- Cause of uncertainty – Partial observability and/or nondeterminism
- Example of uncertain reasoning – Patient with toothache
  - St1- $Toothache \Rightarrow Cavity$
  - St2- $Toothache \Rightarrow Cavity \vee GumProblem \vee Abscess \ldots$
  - St3- $Cavity \Rightarrow Toothache$
- Three main reasons for failures
  - Laziness – too many antecedents/consequents
  - Theoretical ignorance – no complete knowledge
  - Practical ignorance – not all tests can be run
- Solution – *degree of belief*
- Tool: Probability theory – 80% of the toothache patients seen so far have had cavities (statistical data)
- Probability theory – provides a way of summarizing the uncertainty that comes from the laziness and ignorance

Acting under Uncertainty

- Uncertainty and rational decisions
  - Example- Automated taxi
    - Plan 1 (A90) – Leave 90 mins early
    - Plan 2 (A180) – Leave 180 mins early
    - Plan 3 (A1440) – Leave 24 hours early
  - Which is the rational choice?
- Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called decision theory:

$$\textit{Decision theory} = \textit{probability theory} + \textit{utility theory}$$

- Decision theory combines the agent's beliefs and desires, defining the best action as the one that maximizes expected utility.

Basic Probability Notation

- Sample space – the set of all possible worlds
  - For example, two dice are rolled – 36 possible worlds
- Probability model – associates a numerical probability with each possible world

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1$$

- Unconditional Probability / Prior Probability
  - For example, rolling the 2 dices and they add up to 11, P(Total=11) = P(<5,6>)+P(<6,5>)
- Conditional Probability / Posterior Probability
  - For example, rolling the 2 dices given that the first die is a 5, P(doubles|Die_1=5)

Basic Probability Notation

- Mathematically, conditional probability

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

which holds whenever $P(b) > 0$. For example,

$$P(doubles \mid Die_1 = 5) = \frac{P(doubles \wedge Die_1 = 5)}{P(Die_1 = 5)}$$

- Product rule

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

- Random variables – variables in probability theory
- Domain
- Example - "The probability that the patient has a cavity, given that she is a teenager with no toothache, is 0.1"

$$P(cavity \mid \neg toothache \wedge teen) = 0.1$$

Basic Probability Notation

- Probability distribution
  - For example, Weather = {sunny, rain, cloudy, snow}

Sometimes we will want to talk about the probabilities of *all* the possible values of a random variable. We could write:

$$P(Weather = sunny) = 0.6$$
$$P(Weather = rain) = 0.1$$
$$P(Weather = cloudy) = 0.29$$
$$P(Weather = snow) = 0.01 ,$$

but as an abbreviation we will allow

$$\mathbf{P}(Weather) = \langle 0.6, 0.1, 0.29, 0.01 \rangle ,$$

  - Statement P defines a probability distributions for the random variable Weather
  - For a continuous variables, P defines the probability density function (pdf)

Basic Probability Notation

- Probability axioms
  - Relationship between a proposition and its negation

$$\begin{aligned} P(\neg a) &= \sum_{\omega \in \neg a} P(\omega) \\ &= \sum_{\omega \in \neg a} P(\omega) + \sum_{\omega \in a} P(\omega) - \sum_{\omega \in a} P(\omega) \\ &= \sum_{\omega \in \Omega} P(\omega) - \sum_{\omega \in a} P(\omega) \\ &= 1 - P(a) \end{aligned}$$

  - Inclusion-exclusion principle

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

Full Joint Probability Distribution

- Distributions on multiple variables
- For example,
  - Weather = {sunny, rain, cloudy, snow}
  - Cavity = {cavity, ¬ cavity}
  - Joint Probability Distribution of Weather & Cavity

$$
\begin{aligned}
&P(W = sunny \wedge C = true) = P(W = sunny|C = true)\, P(C = true)\\
&P(W = rain \wedge C = true) = P(W = rain|C = true)\, P(C = true)\\
&P(W = cloudy \wedge C = true) = P(W = cloudy|C = true)\, P(C = true)\\
&P(W = snow \wedge C = true) = P(W = snow|C = true)\, P(C = true)\\
&P(W = sunny \wedge C = false) = P(W = sunny|C = false)\, P(C = false)\\
&P(W = rain \wedge C = false) = P(W = rain|C = false)\, P(C = false)\\
&P(W = cloudy \wedge C = false) = P(W = cloudy|C = false)\, P(C = false)\\
&P(W = snow \wedge C = false) = P(W = snow|C = false)\, P(C = false)\,.
\end{aligned}
$$

  - can be written as a single equation:

$$\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather \mid Cavity)\mathbf{P}(Cavity)$$

Basic Probability Notation

- Sample space – the set of all possible worlds
  - For example, two dice are rolled – 36 possible worlds
- Probability model – associates a numerical probability with each possible world

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1$$

- Unconditional Probability / Prior Probability
  - For example, rolling the 2 dices and they add up to 11, P(Total=11) = P(<5,6>)+P(<6,5>)
- Conditional Probability / Posterior Probability
  - For example, rolling the 2 dices given that the first die is a 5, P(doubles|Die_1=5)

Basic Probability Notation

- Mathematically, conditional probability

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

which holds whenever $P(b) > 0$. For example,

$$P(doubles \mid Die_1 = 5) = \frac{P(doubles \wedge Die_1 = 5)}{P(Die_1 = 5)}$$

- Product rule

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

- Random variables – variables in probability theory
- Domain
- Example - "The probability that the patient has a cavity, given that she is a teenager with no toothache, is 0.1"

$$P(cavity \mid \neg toothache \wedge teen) = 0.1$$

Basic Probability Notation

- Probability distribution
  - For example, Weather = {sunny, rain, cloudy, snow}

Sometimes we will want to talk about the probabilities of *all* the possible values of a random variable. We could write:

$$P(Weather = sunny) = 0.6$$
$$P(Weather = rain) = 0.1$$
$$P(Weather = cloudy) = 0.29$$
$$P(Weather = snow) = 0.01 ,$$

but as an abbreviation we will allow

$$\mathbf{P}(Weather) = \langle 0.6, 0.1, 0.29, 0.01 \rangle ,$$

  - Statement P defines a probability distributions for the random variable Weather
  - For a continuous variables, P defines the probability density function (pdf)

Basic Probability Notation

- Probability axioms
  - Relationship between a proposition and its negation

$$\begin{aligned} P(\neg a) &= \sum_{\omega \in \neg a} P(\omega) \\ &= \sum_{\omega \in \neg a} P(\omega) + \sum_{\omega \in a} P(\omega) - \sum_{\omega \in a} P(\omega) \\ &= \sum_{\omega \in \Omega} P(\omega) - \sum_{\omega \in a} P(\omega) \\ &= 1 - P(a) \end{aligned}$$

  - Inclusion-exclusion principle

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

Full Joint Probability Distribution

- Distributions on multiple variables
- For example,
  - Weather = {sunny, rain, cloudy, snow}
  - Cavity = {cavity, ¬ cavity}
  - Joint Probability Distribution of Weather & Cavity

$$
\begin{aligned}
&P(W = sunny \wedge C = true) = P(W = sunny|C = true)\, P(C = true)\\
&P(W = rain \wedge C = true) = P(W = rain|C = true)\, P(C = true)\\
&P(W = cloudy \wedge C = true) = P(W = cloudy|C = true)\, P(C = true)\\
&P(W = snow \wedge C = true) = P(W = snow|C = true)\, P(C = true)\\
&P(W = sunny \wedge C = false) = P(W = sunny|C = false)\, P(C = false)\\
&P(W = rain \wedge C = false) = P(W = rain|C = false)\, P(C = false)\\
&P(W = cloudy \wedge C = false) = P(W = cloudy|C = false)\, P(C = false)\\
&P(W = snow \wedge C = false) = P(W = snow|C = false)\, P(C = false)\,.
\end{aligned}
$$

  - can be written as a single equation:

$$\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather \mid Cavity)\mathbf{P}(Cavity)$$

Full Joint Probability Distribution

- Distributions on multiple variables
- For example,
  - Weather = {sunny, rain, cloudy, snow}
  - Cavity = {cavity, ¬ cavity}
  - Joint Probability Distribution of Weather & Cavity

$$
\begin{aligned}
P(W = sunny \wedge C = true) &= P(W = sunny | C = true)\, P(C = true) \\
P(W = rain \wedge C = true) &= P(W = rain | C = true)\, P(C = true) \\
P(W = cloudy \wedge C = true) &= P(W = cloudy | C = true)\, P(C = true) \\
P(W = snow \wedge C = true) &= P(W = snow | C = true)\, P(C = true) \\
P(W = sunny \wedge C = false) &= P(W = sunny | C = false)\, P(C = false) \\
P(W = rain \wedge C = false) &= P(W = rain | C = false)\, P(C = false) \\
P(W = cloudy \wedge C = false) &= P(W = cloudy | C = false)\, P(C = false) \\
P(W = snow \wedge C = false) &= P(W = snow | C = false)\, P(C = false)\ .
\end{aligned}
$$

  - can be written as a single equation:

$$\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather \mid Cavity)\mathbf{P}(Cavity)$$

Inference Using Full Joint Distributions

- To study method for probabilistic inference
- For example,

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

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

- How to determine the probability of any proposition?

ity of any proposition, simple or complex: simply identify those possible worlds in which the proposition is true and add up their probabilities. For example, there are six possible worlds in which $cavity \vee toothache$ holds:

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

Inference Using Full Joint Distributions

- To compute conditional probabilities

Equation (13.3) to obtain an expression in terms of unconditional probabilities and then evaluating the expression from the full joint distribution. For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

$$P(cavity \mid toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6 .$$

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

$$P(\neg cavity \mid toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 .$$

Independence

- Independence between propositions a and b can be written as

$$P(a \mid b) = P(a) \quad \text{or} \quad P(b \mid a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b)$$

Bayes' Rule and Its Use

- Two forms of the product rule

$$P(a \wedge b) = P(a \mid b)P(b) \qquad \text{and} \qquad P(a \wedge b) = P(b \mid a)P(a) \ .$$

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

$$P(b \mid a) = \frac{P(a \mid b)P(b)}{P(a)} \ . \tag{13.12}$$

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.

Applying Bayes' rule

- Example,

diagnosis, we often have conditional probabilities on causal relationships (that is, the doctor knows $P(symptoms \mid disease)$) and want to derive a diagnosis, $P(disease \mid symptoms)$. For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

$$\begin{aligned} P(s \mid m) &= 0.7 \\ P(m) &= 1/50000 \\ P(s) &= 0.01 \\ P(m \mid s) &= \frac{P(s \mid m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014 . \end{aligned} \tag{13.14}$$

That is, we expect less than 1 in 700 patients with a stiff neck to have meningitis. Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in the patient remains small. This is because the prior probability of stiff necks is much higher than that of meningitis.