



A T M E
College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Notes of Lesson

FACULTY NAME: DR. SHILPA B L

SUBJECT TITLE AND CODE: ARTIFICIAL INTELLIGENCE – BCS515B

SEMESTER AND SCHEME: V SEMESTER





DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Department of Computer Science And Engineering

Vision of the Department

To develop highly talented individuals in Computer Science and Engineering to deal with real world challenges in industry, education, research and society

Mission of the Department

- *To inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.*
- *Motivate students to put their thoughts and ideas adoptable by industry or to pursue higher studies leading to research.*

Program Educational Objectives (PEO'S):

1. Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.
2. Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.
3. Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.
4. Ability to function ethically and responsibly in a rapidly changing environment by applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.



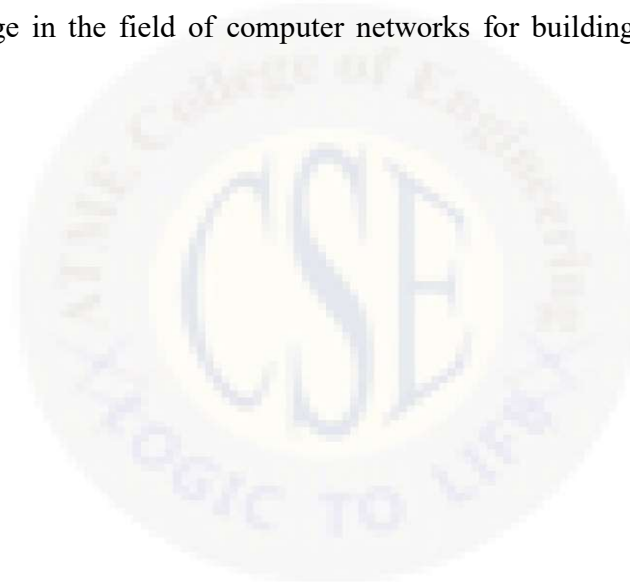
A T M E
College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Program Specific Outcomes (PSOs)

1. Ability to apply skills in the field of algorithms, database design, web design, cloud computing and data analytics.
2. Apply knowledge in the field of computer networks for building network and internet-based applications.



ARTIFICIAL INTELLIGENCE		Semester	V
Course Code	BCS515B	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	3:0:0:0	SEE Marks	50
Total Hours of Pedagogy	40	Total Marks	100
Credits	03	Exam Hours	3
Examination type (SEE)	Theory		
Course objectives: • Learn the basic principles and theories underlying artificial intelligence, including machine learning, neural networks, natural language processing, and robotics. • Apply AI techniques to solve real-world problems, including search algorithms, optimization, and decision-making processes. • Understand the ethical, legal, and societal implications of AI, including topics such as bias, fairness, accountability, and the impact of AI on the workforce and privacy.			
Teaching-Learning Process (General Instructions) These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes. 1. Use of Video/Animation to explain functioning of various concepts. 2. Encourage collaborative (Group Learning) Learning in the class. 3. Discuss application of every concept to solve the real-world problems. 4. Demonstrate ways to solve the same problem and encourage the students to come up with their own creative solutions.			
Module-1			
Introduction: What Is AI? , The State of The Art. Intelligent Agents: Agents and environment, Concept of Rationality, The nature of environment, The structure of agents. Chapter 1 - 1.1, 1.4 Chapter 2 - 2.1, 2.2, 2.3, 2.4			
Module-2			
Problem-solving: Problem-solving agents, Example problems, Searching for Solutions Uninformed Search Strategies Chapter 3 - 3.1, 3.2, 3.3, 3.4			
Module-3			
Problem-solving: Informed Search Strategies, Heuristic functions Logical Agents: Knowledge-based agents, The Wumpus world, Logic, Propositional logic, Reasoning patterns in Propositional Logic Chapter 3 - 3.5, 7.6 Chapter 7 - 7.1, 7.2, 7.3, 7.4			
Module-4			
First Order Logic: Representation Revisited, Syntax and Semantics of First Order logic, Using First Order logic, Knowledge Engineering In First-Order Logic Inference in First Order Logic: Propositional Versus First Order Inference, Unification, Forward Chaining Chapter 8- 8.1, 8.2, 8.3, 8.4 Chapter 9- 9.1, 9.2, 9.3			

Module-5
Inference in First Order Logic: Backward Chaining, Resolution Classical Planning: Definition of Classical Planning, Algorithms for Planning as State-Space Search, Planning Graphs Chapter 9-9.4, 9.5 Chapter 10- 10.1,10.2,10.3
Course outcomes (Course Skill Set) At the end of the course, the student will be able to: 1. Explain the architecture and components of intelligent agents, including their interaction with the AI environment. 2. Apply problem-solving agents and various search strategies to solve a given problem. 3. Illustrate logical reasoning and knowledge representation using propositional and first-order logic. 4. Demonstrate proficiency in representing knowledge and solving problems using first-order logic. 5. Describe classical planning in the context of artificial intelligence, including its goals, constraints, and applications in problem-solving.
Assessment Details (both CIE and SEE) The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together. Continuous Internal Evaluation: • For the Assignment component of the CIE, there are 25 marks and for the Internal Assessment Test component, there are 25 marks. • The first test will be administered after 40-50% of the syllabus has been covered, and the second test will be administered after 85-90% of the syllabus has been covered • Any two assignment methods mentioned in the 22OB2.4, if an assignment is project-based then only one assignment for the course shall be planned. The teacher should not conduct two assignments at the end of the semester if two assignments are planned. • For the course, CIE marks will be based on a scaled-down sum of two tests and other methods of assessment. Internal Assessment Test question paper is designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course. Semester-End Examination: Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (duration 03 hours). 1. The question paper will have ten questions. Each question is set for 20 marks. 2. There will be 2 questions from each module. Each of the two questions under a module (with

Suggested Learning Resources:**Text Book**

Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson, 2015

Reference Books

1. Elaine Rich, Kevin Knight, Artificial Intelligence, 3rd edition, Tata McGraw Hill, 2013
2. George F Luger, Artificial Intelligence Structure and strategies for complex, Pearson Education, 5th Edition, 2011
3. Nils J. Nilsson, Principles of Artificial Intelligence, Elsevier, 1980
4. Saroj Kaushik, Artificial Intelligence, Cengage learning, 2014

Web links and Video Lectures (e-Resources):

1. <https://www.kdnuggets.com/2019/11/10-free-must-read-books-ai.html>
2. <https://www.udacity.com/course/knowledge-based-ai-cognitive-systems--ud409>
3. <https://nptel.ac.in/courses/106/105/106105077/>

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

1. Using OpenAI tool, develop a chatbot (25 marks)

MODULE - 1

Introduction to Artificial Intelligence:

AI is one of the newest fields in science and engineering. Work started in earnest soon after World War II, and the name itself was coined in 1956. AI currently encompasses a huge variety of subfields, ranging from the general (learning and perception) to the specific, such as playing chess, proving mathematical theorems, writing poetry, driving a car on a crowded street, and diagnosing diseases. AI is relevant to any intellectual task; it is truly a universal field.

1.1 WHAT IS AI?

In Figure 1.1 we see eight definitions of AI, laid out along two dimensions. The definitions on top are concerned with thought processes & reasoning, whereas the ones on the bottom address behavior. The definitions on the left measure success in terms of fidelity to human performance, whereas the ones on the right measure against an ideal performance measure, called rationality. A system is rational if it does the “right thing,” given what it knows. Historically, all four

Thinking Humanly “The exciting new effort to make computers think . . . <i>machines with minds</i> , 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)	Thinking Rationally “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)
Acting Humanly “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, 1991)	Acting Rationally “Computational Intelligence is the study of the design of intelligent agents.” (Poole <i>et al.</i> , 1998) “AI . . . is concerned with intelligent behavior in artifacts.” (Nilsson, 1998)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

approaches to AI have been followed, each by different people with different methods. A human-centered approach must be in part an empirical science, involving observations and hypotheses about human behavior. A rationalist approach involves a combination of mathematics and engineering.

1.1.1 Acting humanly: The Turing Test approach TURING TEST

The Turing Test, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. The computer would need to possess the following capabilities:

- Natural language processing to enable it to communicate successfully in English;
- Knowledge representation to store what it knows or hears;
- Automated reasoning to use the stored information to answer questions and to draw conclusions;
- Machine learning to adapt one circumstances and to detect and extrapolate patterns.

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because physical simulation of a person is unnecessary for intelligence. However, the so-called total Turing Test includes a video signal so that interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the total Turing Test, the computer will need

- Computer vision to perceive objects, and
- Robotics to manipulate objects and move about.

1.1.2 Thinking humanly: The cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get inside the actual workings of human minds. There are three ways to do this: through introspection—trying to catch our own thoughts as they go by; through psychological experiments—observing a person in action; and through brain imaging—observing the brain in action . Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input-output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans.

1.1.3 Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is, certain reasoning processes. His syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises. Logicians in the 19th century developed a precise notation for statements about all kinds of objects in the world

and the relations among them. The so-called logic is tradition within artificial intelligence hopes to build on such programs to create intelligent systems. There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between solving a problem “in principle” and solving it in practice.

1.1.4 Acting rationally: The rational agent approach

An agent is just something that acts (agent comes from the Latin *agere*, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. In the “laws of thought” approach to AI, the emphasis was on correct inferences. Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one’s goals and then to act on that conclusion.

The rational agent approach has advantages over the other approaches. First, it is more general than the “laws of thought” approach because correct inference is just one of several possible mechanisms for achieving rationality. Second, it is more amenable to scientific development than are approaches based on human behavior or human thought. The standard of rationality is mathematically well defined and completely general and can be “unpacked” to generate agent designs that provably achieve it.

1.4 The State of The Art

What can AI do today? A concise answer is difficult because there are so many activities in so many subfields. Here we sample a few applications:

Robotic vehicles:

A driverless robotic car named STANLEY sped through the rough terrain of the Mojave desert at 22 mph, finishing the 132-mile course first to win the 2005 DARPA Grand Challenge. STANLEY is a Volkswagen Touareg outfitted with cameras, radar, and laser rangefinders to sense the environment and onboard software to command the steering, braking, and

acceleration (Thrun, 2006). The following year CMU's BOSS won the Urban Challenge, safely driving in traffic through the streets of a closed Air Force base, obeying traffic rules and avoiding pedestrians and other vehicles.

Speech recognition:

A traveler calling United Airlines to book a flight can have the entire conversation guided by an automated speech recognition and dialog management system.

Autonomous planning and scheduling:

A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson et al., 2000). REMOTE AGENT generated plans from high-level goals specified from the ground and monitored the execution of those plans—detecting, diagnosing, and recovering from problems as they occurred. Successor program MAPGEN (Al-Chang et al., 2004) plans the daily operations for NASA's Mars Exploration Rovers, and MEXAR2 (Cesta et al., 2007) did mission planning—both logistics and science planning—for the European Space Agency's Mars Express mission in 2008

Game playing:

IBM's DEEP BLUE became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997). Kasparov said that he felt a “new kind of intelligence” across the board from him. Newsweek magazine described the match as “The brain's last stand.” The value of IBM's stock increased by \$18 billion. Human champions studied Kasparov's loss and were able to draw a few matches in subsequent years, but the most recent human-computer matches have been won convincingly by the computer.

Spam fighting: Each day, learning algorithms classify over a billion messages as spam, saving the recipient from having to waste time deleting what, for many users, could comprise 80% or 90% of all messages, if not classified away by algorithms. Because the spammers are continually updating their tactics, it is difficult for a static programmed approach to keep up, and learning algorithms work best (Sahami et al., 1998; Goodman and Heckerman, 2004).

Logistics planning:

During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques generated in hours a plan that would have taken weeks

with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

Robotics: The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use. The company also deploys the more rugged PackBot to Iraq and Afghanistan, where it is used to handle hazardous materials, clear explosives, and identify the location of snipers.

Machine Translation: A computer program automatically translates from Arabic to English, allowing an English speaker to see the headline "Ardogan Confirms That Turkey Would Not Accept Any Pressure, Urging Them to Recognize Cyprus." The program uses a statistical model built from examples of Arabic-to-English translations and from examples of English text totaling two trillion words (Brants et al., 2007). None of the computer scientists on the team speak Arabic, but they do understand statistics and machine learning algorithms.

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—but rather science, engineering, and mathematics.

Chapter-02: - Problem-solving agents

Agents and Environment:

An agent is anything that can be viewed as perceiving its environment through sensors and SENSOR acting upon that environment through actuators. This simple idea is illustrated in Figure 2.1. ACTUATOR A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.

We use the term percept to refer to the agent's perceptual inputs at any given instant. An PERCEPT SEQUENCE agent's percept sequence is the complete history of everything the agent has ever perceived.

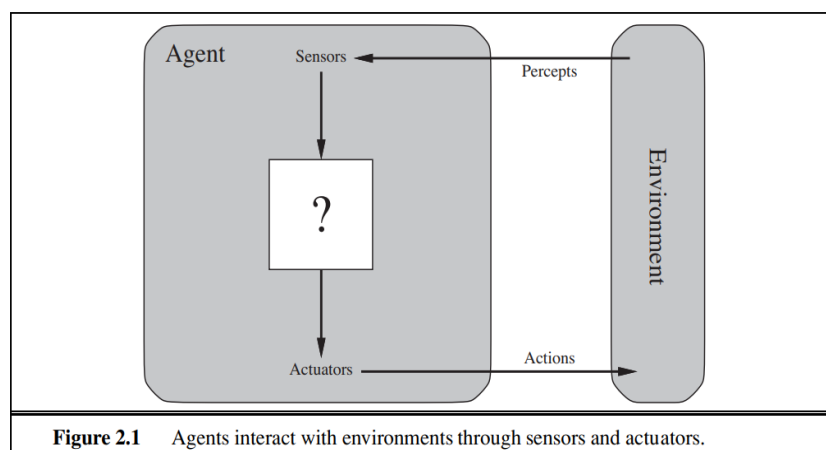


Figure 2.1 Agents interact with environments through sensors and actuators.

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

We can imagine tabulating the agent function that describes any given agent;

Given an agent to experiment with, we can, in principle, construct this table by trying out all possible percept sequences and recording which actions the agent does in response. The table is, of course, an external characterization of the agent. Internally, the agent function for an artificial agent will be implemented by an AGENT PROGRAM agent program.

It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.

The Vacuum-Cleaner World:

- This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. 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.

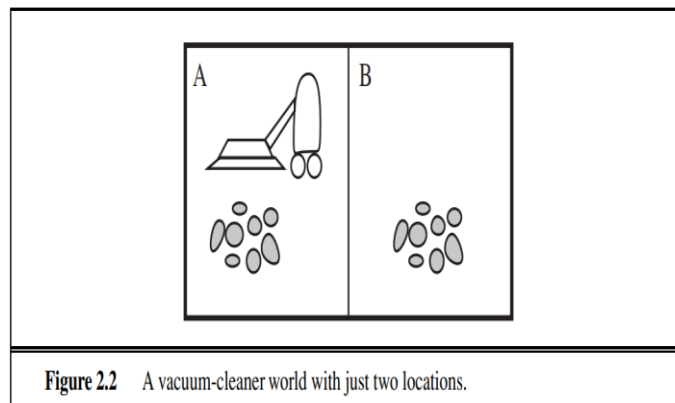


Figure 2.2 A vacuum-cleaner world with just two locations.

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
⋮	⋮
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

Program implements the agent function tabulated in Fig. 2.3

Function Reflex-Vacuum-Agent(*location, status*) return an action

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

2.2 Good Behavior: The Concept of Rationality

Rationality in AI agents involve performing the "right" actions to achieve desirable outcomes, which are evaluated through a performance measure. This measure assesses the desirability of the resulting states in the environment, rather than the agent's own perception of success. A key challenge is designing performance measures that truly reflect the goals of the environment, as poor design can lead to unintended behaviors. For example, a vacuum-cleaner agent might focus on cleaning up dirt repeatedly rather than maintaining a clean floor. Additionally, complex philosophical questions arise, such as whether consistent mediocre performance is preferable to alternating highs and lows. Ultimately, rationality in agents depends on aligning their actions with well-defined performance measures that ensure the desired outcomes are achieved in the environment.

2.2.1 Rationality

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

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

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

2.2.2 Omniscience, learning, and autonomy

It's important to distinguish between rationality and omniscience. An omniscient agent would know the exact outcomes of its actions and could act accordingly, but in reality, omniscience is impossible. For example, if I decide to cross a street after seeing a friend and, unexpectedly, a cargo door falls from an airplane and hits me, this does not mean my decision to cross the street was irrational. Rationality involves making decisions that maximize expected performance based on available information, not achieving perfect outcomes. Expecting an

agent to always make the best possible decision after the fact is unrealistic, as it would require knowledge of future events—something only a time machine or crystal ball could provide. Rationality is about doing the best one can with the information at hand, not achieving flawless outcomes.

Our definition of rationality does not require omniscience because rationality depends only on the percept sequence received by the agent so far, not on future knowledge. However, rationality also means avoiding actions that are underinformed or risky. For example, if an agent doesn't look both ways before crossing a busy road, it lacks crucial information, and crossing would be irrational. A rational agent should engage in information gathering, such as looking before crossing, to maximize expected performance. Information gathering and exploration are key aspects of rationality, as they help the agent improve decision-making by acquiring necessary data.

Additionally, rational agents must not only gather information but also learn from their experiences. While an agent may start with some prior knowledge, it should adapt and update its understanding as it interacts with the environment. This learning process helps the agent become more autonomous, reducing reliance on initial assumptions and making it more effective in diverse environments. For instance, a vacuum-cleaning agent that learns where dirt typically appears will perform better than one that does not.

Autonomy is essential for rationality. An agent that depends too much on its designer's prior knowledge without learning from its own percepts lacks autonomy. Over time, a rational agent should be able to act independently of its initial programming, adjusting its behavior based on experience. This combination of initial knowledge and continuous learning enables the design of a single rational agent capable of succeeding in a wide range of environments.

2.3 The nature of environment

Now that we understand what rationality means, the next step in creating rational agents is to think about **task environments**. A task environment is basically the "problem" that the agent is designed to solve. Before we can build a rational agent, we need to clearly define the task environment, including what the agent needs to achieve, what information it receives, and what actions it can take. Different environments have different needs, so the way we design the agent depends on the specific environment it will operate in. In the next sections, we'll look at examples of task environments and see how they influence the design of the agent.

2.3.1 Specifying the task environment:

When discussing the rationality of the simple vacuum-cleaner agent, we had to define four key elements: the **Performance measure** (how success is judged), the **Environment** (where the agent operates), the **Actuators** (the agent's actions), and the **Sensors** (how the agent perceives its surroundings). These elements are grouped under the heading of the **task environment**, and we use the acronym **PEAS** (Performance, Environment, Actuators, Sensors) to remember them. When designing any agent, the first step is to fully specify its task environment using this PEAS framework.

For example, consider the more complex problem of designing an automated taxi driver. Unlike the simple vacuum-cleaner scenario, the task of driving a taxi is extremely open-ended, with countless possible situations that the agent might encounter. While fully automated taxis are still beyond current technology, this example highlights the importance of carefully defining the task environment to ensure the agent is designed effectively. By specifying the performance measures, environment, actuators, and sensors, we can better understand the challenges the agent will face and how it should be designed to handle them.

Figure 2.4 summarizes the PEAS description for the taxi's task environment.

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
Figure 2.4 PEAS description of the task environment for an automated taxi.				

2.3.2 Properties of task environments

The range of task environments in AI is incredibly broad, but we can categorize them using a small set of key **dimensions**. These dimensions help determine the best way to design the agent and choose the appropriate techniques for implementation. By understanding these dimensions, we can better match the agent's design to the specific challenges it will face in different environments. First, we'll list these dimensions, and then we'll analyze several task environments to see how these ideas apply in practice. While the definitions provided here are informal, later chapters will offer more detailed explanations and examples of each type of

environment.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

2.4 The structure of agents:

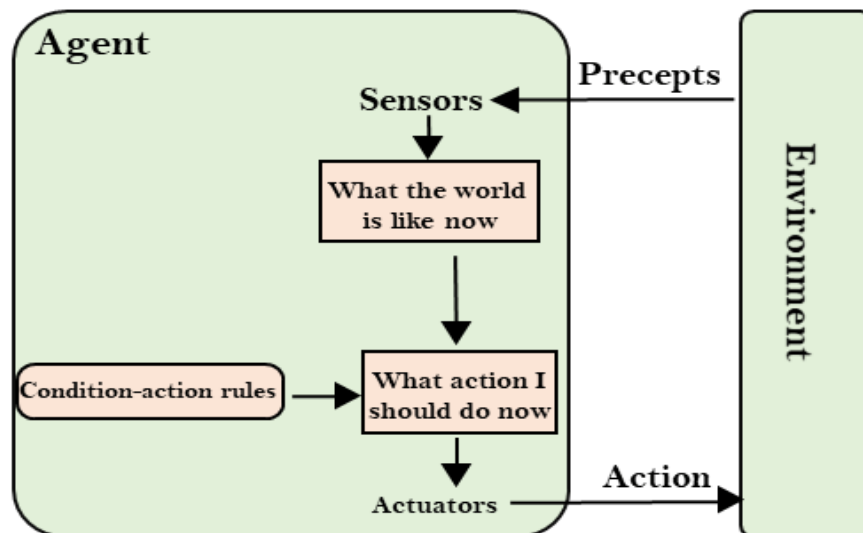
- Agents can be grouped into five classes based on their degree of perceived intelligence and capability.
- All these agents can improve their performance and generate better action over the time. These are given below:

1) Simple Reflex Agent

The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current precepts and ignore the rest of the percept history.

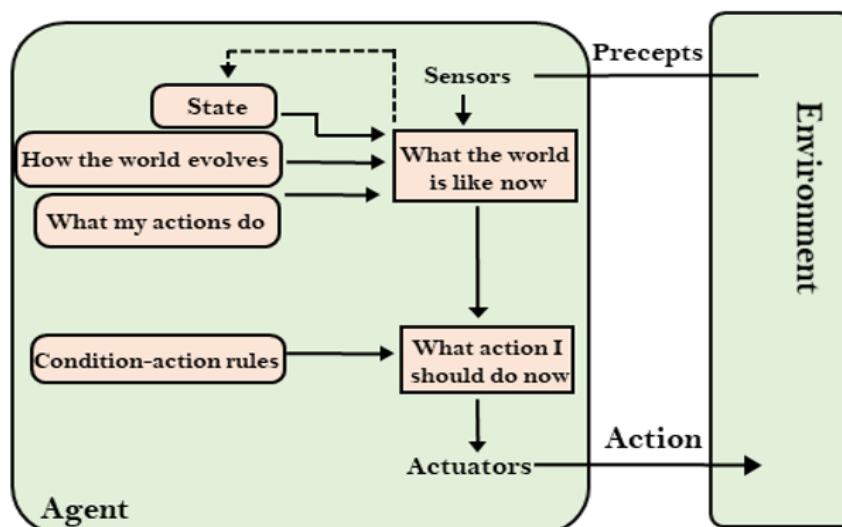
These agents only succeed in the fully observable environment.

The Simple reflex agent does not consider any part of precepts history during their decision and action process.



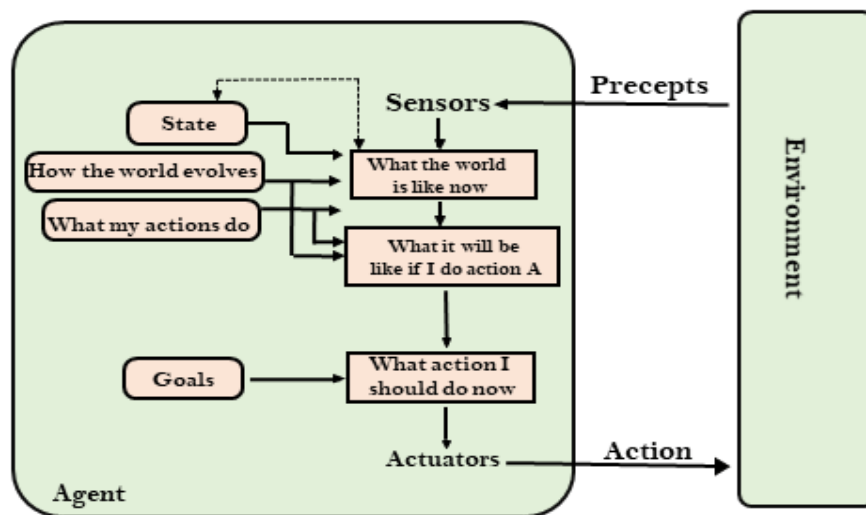
- **Model-based reflex agent**

- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
- 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.
- Updating the agent state requires information about:
 - How the world evolves
 - How the agent's action affects the world.



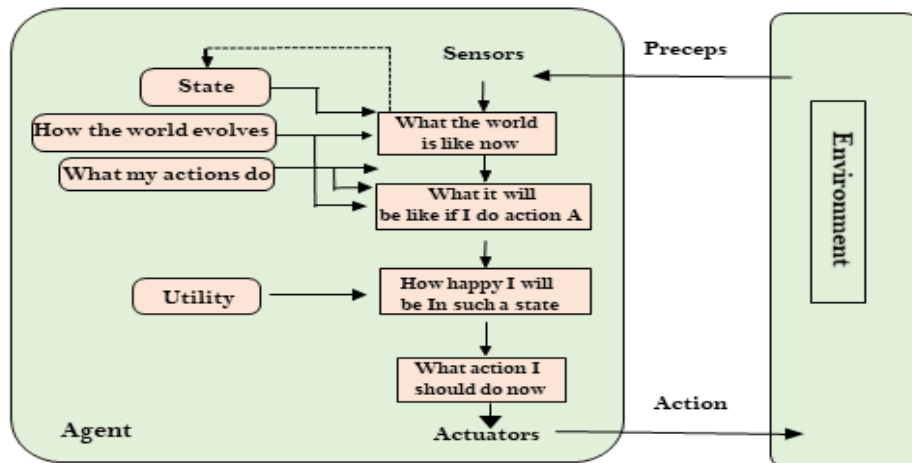
2. Goal-based agents

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



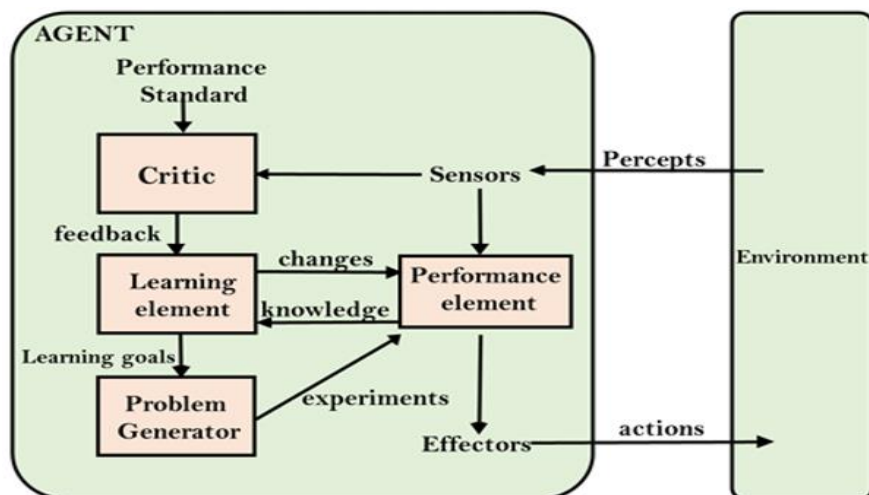
3. Utility-based agent

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



4. Learning agent

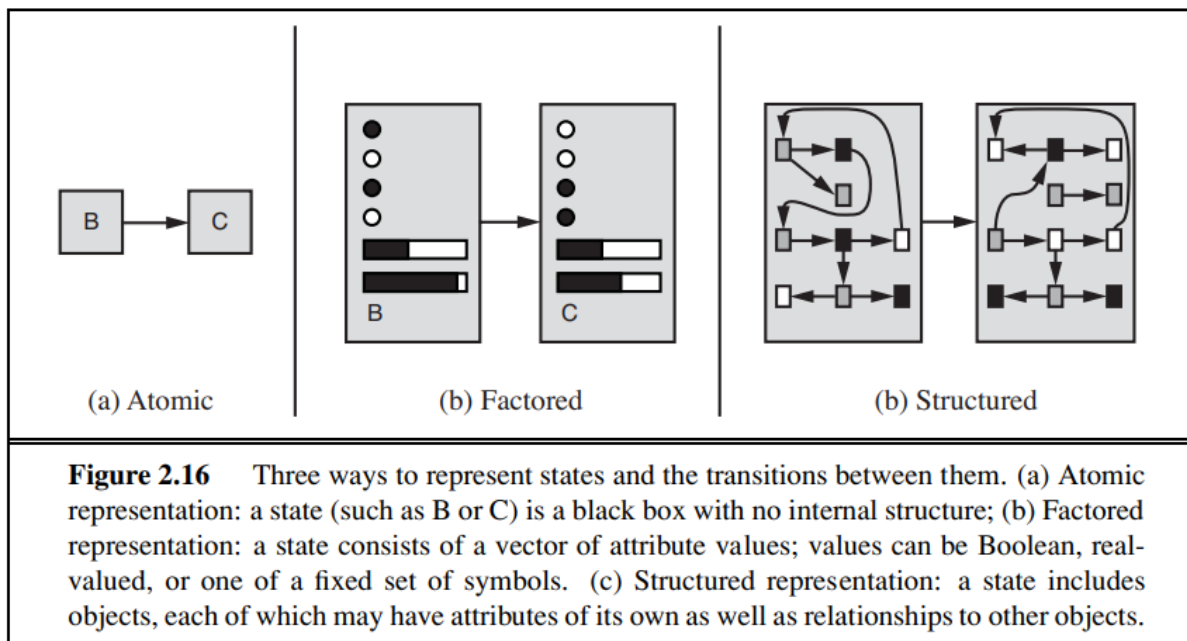
- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
 - **Learning 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.



How the components of agent programs work:

We've discussed agent programs at a high level, focusing on their components that answer questions like "What is the world like now?" and "What action should I take?" Now, the next step for AI students is to understand how these components actually work. This is a complex topic that requires much deeper exploration, but we can start by highlighting some basic distinctions in how these components represent the environment.

These representations can be categorized along a spectrum of complexity and expressive power: **atomic**, **factored**, and **structured**. This classification helps us understand how an agent models its environment and decides on actions. For example, consider the component that handles "What my actions do"—this part of the agent program represents the changes in the environment that result from the agent's actions. By understanding these different levels of representation, we can begin to grasp the various ways in which an agent can interpret and interact with its environment. Figure 2.16 provides schematic depictions of how those transitions might be represented.



MODULE - 2

3.1 Problem solving agents

Intelligent agents are supposed to maximize their performance measure. Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. The agent's task is to find out how to act, now and in the future, so that it reaches a goal state. Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider.

Problem formulation is the process of deciding what actions and states to consider, given a goal. The agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no additional information—i.e., if the environment is unknown in the sense defined then it has no choice but to try one of the actions at random.

The process of looking for a sequence of actions that reaches the goal is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. Thus, we have a simple “formulate, search, execute” design for the agent, as shown in Figure 3.1. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal. Notice that while the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance what they will be. An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on. Control theorists call this an open-loop system, because ignoring the percepts breaks the loop between agent and environment. We first describe the process of problem formulation, and then devote the bulk of the chapter to various algorithms for the SEARCH function. We do not discuss the workings of the UPDATE-STATE and FORMULATE-GOAL functions further in this chapter.

3.1.1 Well-defined problems and solutions

A problem can be defined formally by five components:

- The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as In (Arad)

```

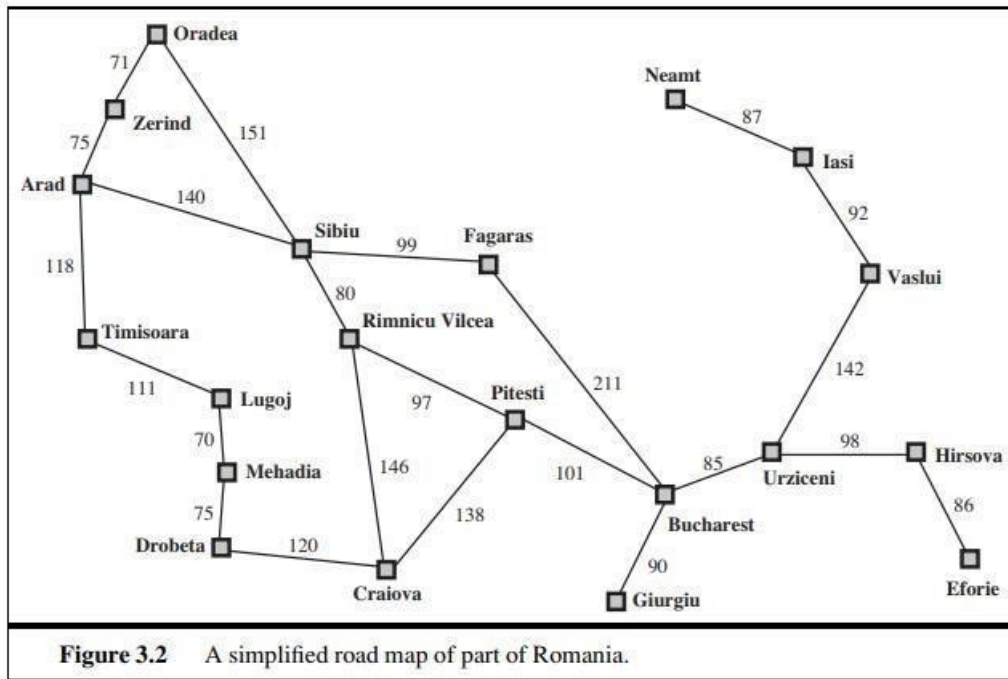
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
               state, some description of the current world state
               goal, a goal, initially null
               problem, a problem formulation

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    if seq = failure then return a null action
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action

```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

- A description of the possible actions available to the agent. Given a particular states, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s. For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.
- A description of what each action does; the formal name for this is the transition model, specified by a function RESULT(s, a) that returns the state that results from doing action a in states. We also use the term successor to refer to any state reachable from a given state by a single action.² For example, we have RESULT(In(Arad), Go(Zerind)) = In(Zerind). Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions. (The map of Romania shown in Figure 3.2 can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction.) A path in the state space is a sequence of states connected by a sequence of actions.
- The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the single tonset {In(Bucharest)}.



Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called “check mate,” where the opponent’s king is under attack and can’t escape.

- A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. In this chapter, we assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The step cost of taking action a in states to reach states' is denoted by $c(s, a, s')$. The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are nonnegative. The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm. A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

3.1.2 Formulating problems

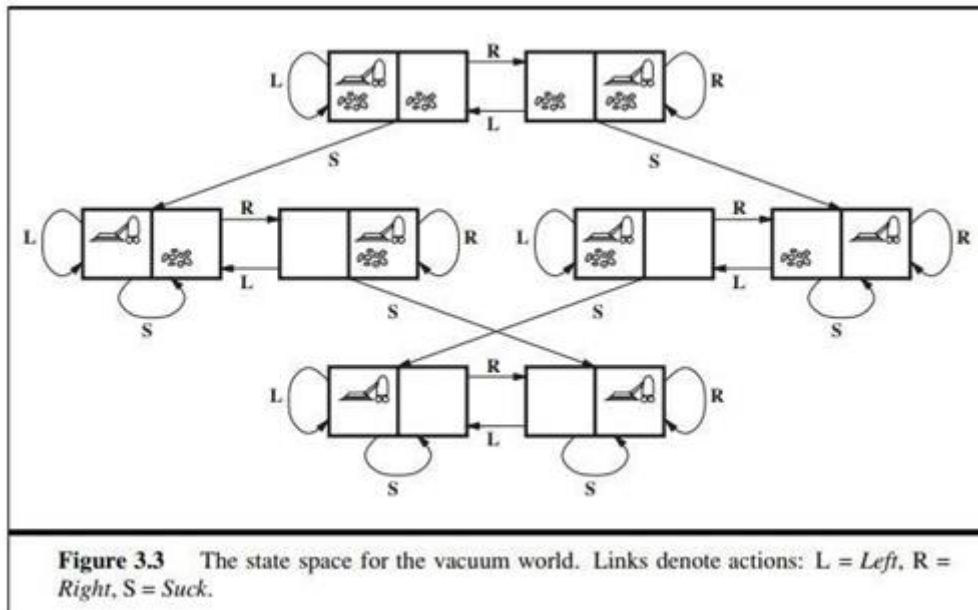
In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a model—an abstract mathematical description—and not there al l thing. Compare the simple state description we have chosen, $In(Arad)$, to an actual cross

country trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a presentation is called abstraction. In addition to abstracting the state description, we must abstract the actions themselves. A driving action has many effects. Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent (as they say, travel is broadening). Our formulation takes into account only the change in location. Also, there are many actions that we omit altogether: turning on the radio, looking out of the window, slowing down for law enforcement officers, and soon. And of course, we don't specify actions at the level of "turn steering wheel to the left by one degree." Can we be more precise about defining the appropriate level of abstraction? Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences. Now consider a solution to the abstract problem: for example, the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths. For example, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip. The abstraction is valid if we can expand any abstract solution into a solution in the more detailed world; a sufficient condition is that for every detailed state that is "in Arad," there is a detailed path to some state that is "in Sibiu," and so on. The abstraction is useful if carrying out each of the actions in the solution is easier than the original problem; in this case they are easy enough that they can be carried out without further search or planning by an average driving agent. The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

3.2 EXAMPLE PROBLEMS

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between toy and real-world problems. A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms. A real-world problem is one whose solutions people actually care

about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.



3.2.1 Toy problems

The first example we examine is the vacuum world first introduced in Chapter 2. (See Figure 2.2.) This can be formulated as a problem as follows:

- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment within locations has $n \cdot 2^n$ 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 left most square, moving Right in the right most square, and Sucking in a clean square have no effect. The complete state space is shown in Figure 3.3.
- **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. Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier. Chapter 4 relaxes some of these assumptions. The 8-puzzle, an instance of which is shown in

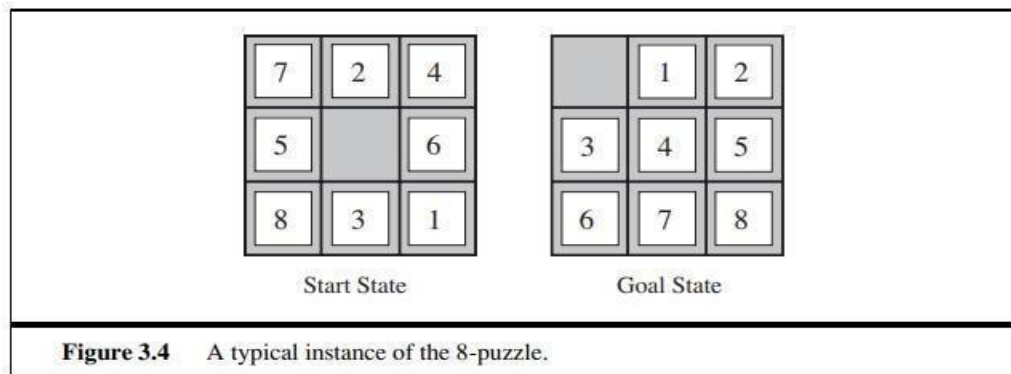


Figure 3.4, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of then in 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 (Exercise3.4).
- **Actions:** 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 star state in Figure3.4, the resulting state has the 5 and the blank switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost:** Each step costs 1,so the path cost is the number of steps in the path.

What abstractions have we included here? The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding. We have abstracted away actions such as shaking the board when pieces get stuck and ruled out extracting the pieces with a knife and putting them back again. We are left with a description of the rules of the puzzle, avoiding all the details of physical manipulations. The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has $9!/2=181,440$ reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5 × 5 board) has around 1025 states, and random instances take several hours to solve optimally.

The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) Figure 3.5 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left. Although efficient special-purpose algorithms exist for this problem and for the whole n-queens family, it remains a useful test problem for search algorithms. There are two main kinds of formulation. An incremental formulation involves operators that augment the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state. A complete state formulation starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:

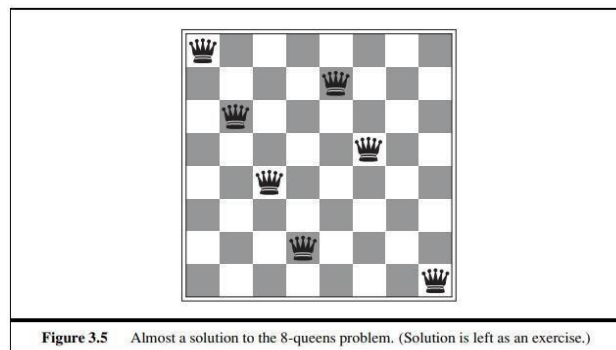


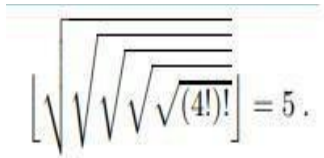
Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

- States: Any arrangement of 0 to 8 queens on the board is a state.
- Initial state: No queens on the board.
- Actions: Add a queen to any empty square.
- Transition model: Returns the board with a queen added to the specified square.
- Goal test: 8 queens are on the board, none attacked. In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:
- States: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the reduction is from roughly 10^{400} states to about 10^{52} states (Exercise 3.5)—a big improvement, but not enough to make the

problem tractable. Section

4.1 describes the complete-state formulation, and Chapter 6 gives a simple algorithm that solves even the million-queens problem with ease. Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer. For example, we can reach from 4 as follows



$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{4!}}}}} \rfloor = 5.$$

The problem definition is very simple:

- States: Positive numbers.
- Initial state: 4.
- Actions: Apply factorial, square root, or floor operation (factorial for integers only).
- Transition model: As given by the mathematical definitions of the operations.
- Goal test: State is the desired positive integer. To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target—for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5—so the state space for this problem is infinite. Such state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

3.2.2 Real-world problems

We have already seen how the route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications. Some, such as Websites and in-car systems that provide driving directions, are relatively straight forward extensions of the Romania example. Others, such as routing video streams in computer networks, military operations planning, and air line travel-planning systems, involve much more complex specifications. Consider the air line travel problems that must be solved by a travel-planning Website:

- States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record information about these “historical” aspects.

- Initial state: This is specified by the user's query.
- Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-air port transfer if needed.
- Transition model: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- Goal test: Are we at the final destination specified by the user?
- Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and soon.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the byzantine fare structures that airlines impose. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—such as back up reservations on alternate flights—to the extent that these are justified by the cost and likelihood of failure of the original plan.

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem “Visit every city in Figure 3.2 at least once, starting and ending in Bucharest.” As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also these to cities the agent has visited. So the initial state would be $In(Bucharest)$, $Visited(\{Bucharest\})$, a typical intermediate state would be $In(Vaslui)$, $Visited(\{Bucharest, Urziceni, Vaslui\})$, and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

The traveling sales person problem (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machine son shopfloors.

A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually

split into two parts: cell layout and channel routing. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed foot print(size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving. Later in this chapter, we present some algorithms capable of solving them. Robot navigation is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. In addition to the complexity of the problem, real robot must also deal with error in their sensor readings and motor controls.

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY(Michie, 1972). Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find in ordering which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. Another important assembly problem is protein design, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

3.3 SEARCHING FOR SOLUTIONS

Having formulated some problems, we now need to solve them. A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state NODE at the root; the branches are actions and the nodes correspond to states in the state space of the problem. Figure 3.6 shows the first few steps in growing the search tree for

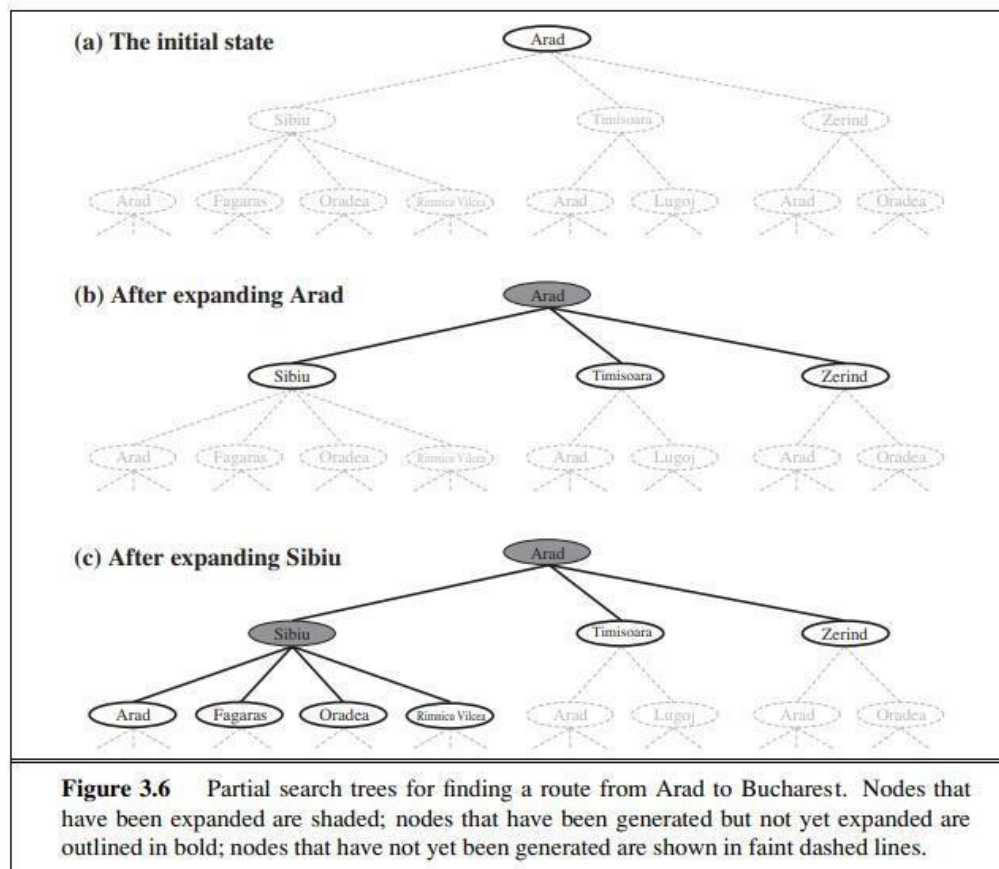
finding a route from Arad to Bucharest. The root node of the tree corresponds to the initial state, $In(Arad)$. The first step is to test whether this is a goal state. (Clearly it is not, but it is important to check so that we can solve trick problems like “starting in Arad, get to Arad.”) Then we need to consider taking various actions. We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating an ewset of states. In this case, we add three branches from the parent node $In(Arad)$ leading to three new child nodes: $In(Sibiu)$, $In(Timisoara)$, and $In(Zerind)$. Now we must choose which of these three possibilities to consider further.

This is the essence of search—following upon option now and putting the others a side for later, in case the first choice does not lead to a solution. Suppose we choose S first. We check to see whether it is a goal state (it is not) and then expand it to get $In(Arad)$, $In(Fagaras)$, $In(Oradea)$, and $In(RimnicuVilcea)$. We can then choose any of these four or go back and choose Timisoara or Zerind. Each of these six nodes is a leaf node, that is, a node with no children in the tree. The set of all leaf nodes available for expansion at any given point is called the frontier. (Many authors call it the open list, which is both geographically less evocative and less accurate, because other data structures are better suited than a list.). In Figure 3.6, the frontier of each tree consists of those nodes with bold outlines.

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. The general TREE-SEARCH algorithm is shown informally in Figure 3.7. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called search strategy.

The eagle eyed reader will notice one peculiar thing about the search tree shown in Figure 3.6: it includes the path from Arad to Sibiu and back to Arad again! We say that $In(Arad)$ is a repeated state in the search tree, generated in this case by a loop y path. Considering such loop y paths means that the complete search tree for Romania is infinite because there is no limit to how often one can traverse a loop. On the other hand, the state space—the map shown in Figure 3.2—has only 20 states. As we discuss in Section 3.4, loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable. Fortunately, there is no need to consider loopy paths. We can rely on more than intuition for this: because path cost are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed. Loopy paths are a special case of the more general concept of redundant paths, which exist whenever there is more than one way to get from one state to another. Consider the paths

Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant it's just a worse way to get to the same state. If you are concerned about reaching the goal, there's never any reason to keep more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other. In some cases, it is possible to define the problem itself so as to eliminate redundant paths. For example, if we formulate the 8-queens problem (page 71) so that a queen can be placed in any column, then each state with n queens can be reached by $n!$ different paths; but if we reformulate the problem so that each new queen is placed in the left most empty column, then each state can be reached only through one path.



```

function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

```

```

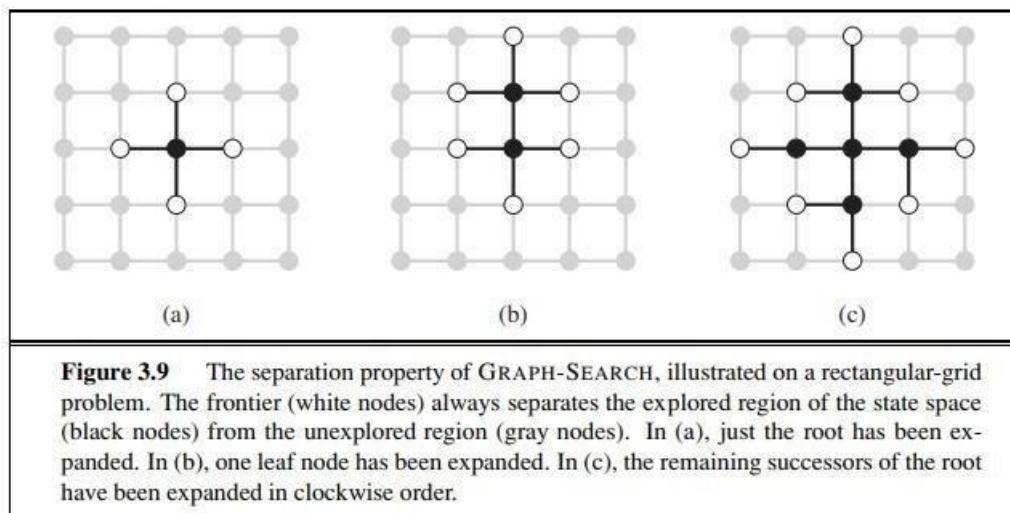
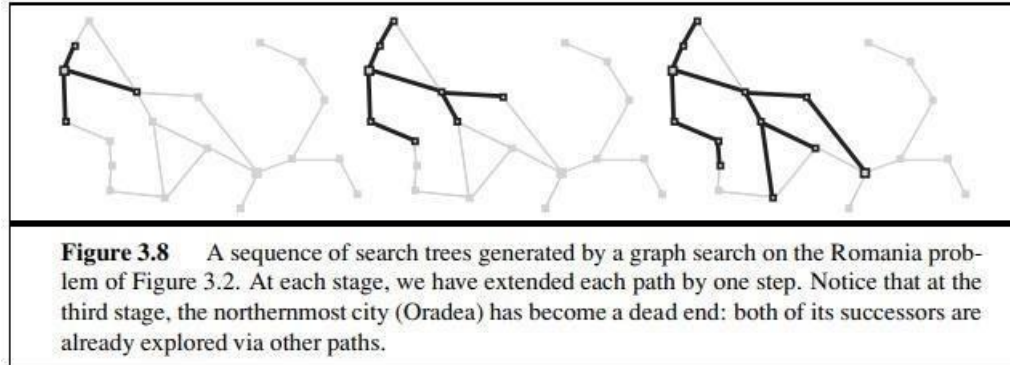
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set

```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

In other cases, redundant paths are unavoidable. This includes all problems where the actions are reversible, such as route-finding problems and sliding-block puzzles. Route finding on a rectangular grid (like the one used later for Figure 3.9) is a particularly important example in computer games. In such a grid, each state has four successors, so a search tree of depth that includes repeated states has 4^d leaves; but there are only about 800 distinct states within steps of any given state. For $d=20$, this means about a trillion nodes but only about 800 distinct states. Thus, following redundant paths can cause a tractable problem to become intractable. This is true even for algorithms that know how to avoid infinite loops. As the saying goes, algorithms that forget their history are doomed to repeat it. The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the explored set (also known as the closed list), which remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier. The new algorithm, called GRAPH-SEARCH, is shown informally in Figure 3.7. The specific algorithms in this chapter draw on this general design. Clearly, the search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph, as shown in Figure 3.8. The algorithm has another nice property: the frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier. (If this seems completely obvious, try Exercise 3.13 now.). This

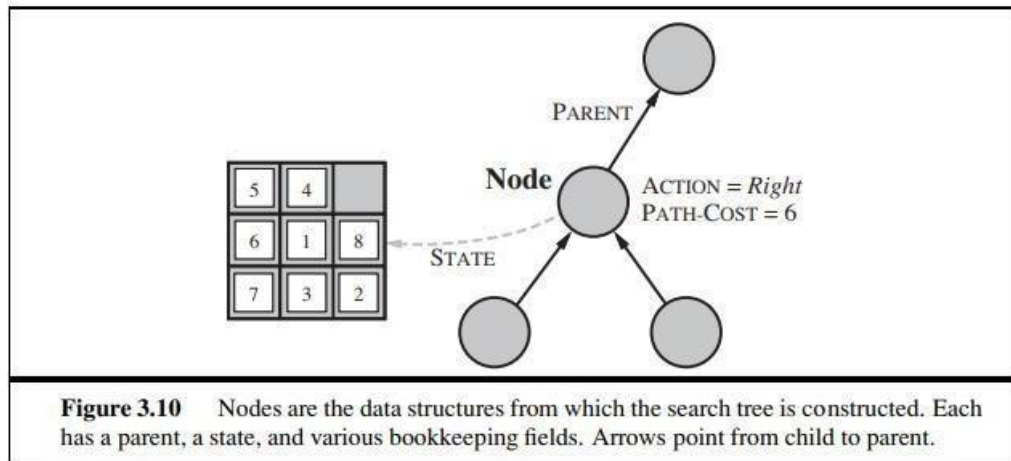
property is illustrated in Figure 3.9. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.



3.3.1 Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node of the tree, we have a structure that contains four components:

- n. STATE: the state in the state space to which the node corresponds;
- n. PARENT: the node in the search tree that generated this node;
- n. ACTION: the action that was applied to the parent to generate the node;
- n. PATH COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.



Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

The node data structure is depicted in Figure 3.10. Notice how the PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found; we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root. Up to now, we have not been very careful to distinguish between nodes and states, but in writing detailed algorithms it's important to make that distinction. A node is a book keeping data structure used to represent the search tree. A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not. Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths. Now that we have nodes, we need some where to put them. The frontier needs to be ordered in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a queue. The operations on a queue are as follows:

- EMPTY?(queue) returns true only if there are no more elements in the queue.
- POP(queue) removes the first element of the queue and returns it.
- INSERT(element, queue) inserts an element and returns the resulting queue.

Queues are characterized by the order in which they store the inserted nodes. Three common

variants are the first-in, first-out or FIFO queue, which pops the oldest element of the queue; the last-in, first-out or LIFO queue (also known as a stack), which pops the newest element of the queue; and the priority queue, which pops the element of the queue with the highest priority according to some ordering function. The explored set can be implemented with a hash table to allow efficient checking for repeated states. With a good implementation, insertion and look up can be done in roughly constant time no matter how many states are stored. One must take care to implement the hash table with the right notion of equality between states. For example, in the traveling sales person problem the hash table needs to know that the set of visited cities {Bucharest, Urziceni, Vaslui} is the same as {Urziceni, Vaslui, Bucharest}. Sometimes this can be achieved most easily by insisting that the data structures for states be in some canonical form; that is, logically equivalent states should map to the same data structure. In the case of states described by sets, for example, a bit-vector representation or a sorted list without repetition would be canonical, whereas an unsorted list would not.

3.3.2 Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution?
- Time complexity: How long does it take to find a solution?
- Space complexity: How much memory is needed to perform the search?

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space graph, $|V| |E|$, where V is the set of vertices (nodes) of the graph and E is the set of edges (links). This is appropriate when the graph is an explicit data structure that is input to the search program. (The map of Romania is an example of this.) In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities: b , the branching factor or maximum number of successors of any node; d , the depth of the shallowest goal node (i.e., the number of steps along the path from the root); and m , the maximum length of any path in the state space. Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory. For the most part, we

describe time and space complexity for search on a tree; for a graph, the answer depends on how “redundant” the paths in the state space are. To assess the effectiveness of a search algorithm, we can consider just the search cost—which typically depends on the time complexity but can also include a term for memory usage—or we can use the total cost, which combines the search cost and the path cost of the solution found. For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the solution cost is the total length of the path in kilometers. Thus, to compute the total cost, we have to add milliseconds and kilometers. There is no “official exchange rate” between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car’s average speed (because time is what the agent cares about). This enables the agent to find an optimal tradeoff point at which further computation to find a shorter path becomes counterproductive. The more general problem of tradeoffs between different goods.

3.4 UNINFORMED SEARCH STRATEGIES

This section covers several search strategies that come under the heading of uninformed search (also called blind search). The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded. Strategies that know whether one non-goal state is “more promising” than another are called informed search or heuristic search strategies.

3.4.1 Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)

```

Figure 3.11 Breadth-first search on a graph.

expanded. Breadth-first search is an instance of the general graph-search algorithm (Figure 3.11) in which the shallowest unexpanded node is chosen for expansion. This is achieved very

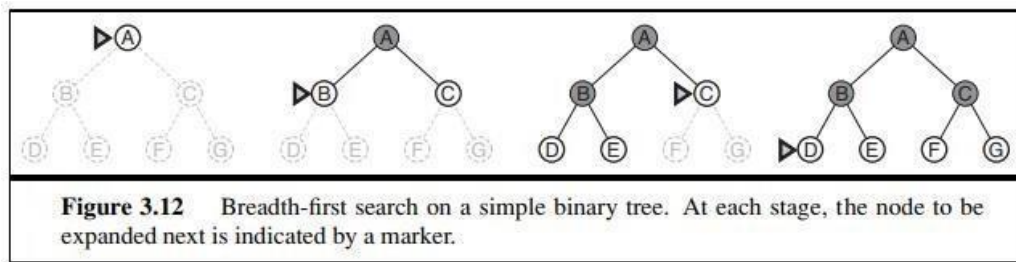
simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity.

Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth first search always has the shallowest path to every node on the frontier. Pseudo code is given in Figure 3.11. Figure 3.12 shows the progress of the search on a simple binary tree. How does breadth-first search rate according to the four criteria from the previous section? We can easily see that it is complete—if the shallowest goal node is at some finite depth d , breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallow nodes must have been generated already and failed the goal test. Now, the shallowest goal node is not necessarily the optimal one. technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node.

The most common such scenario is that all actions have the same cost. So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at these second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and soon. Now suppose that the solution is at depth d . In the worst case, it is the last node generated at that level. Then the total number of nodes generated is $b + b^2 + b^3 + \dots + b^d = O(b^d)$. (If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.)

As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graphsearch in particular, every node generated remains in memory. There will be

nodes in the explored set and $O(b^d)$ nodes in the frontier,



Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

$O(b^{d-1})$

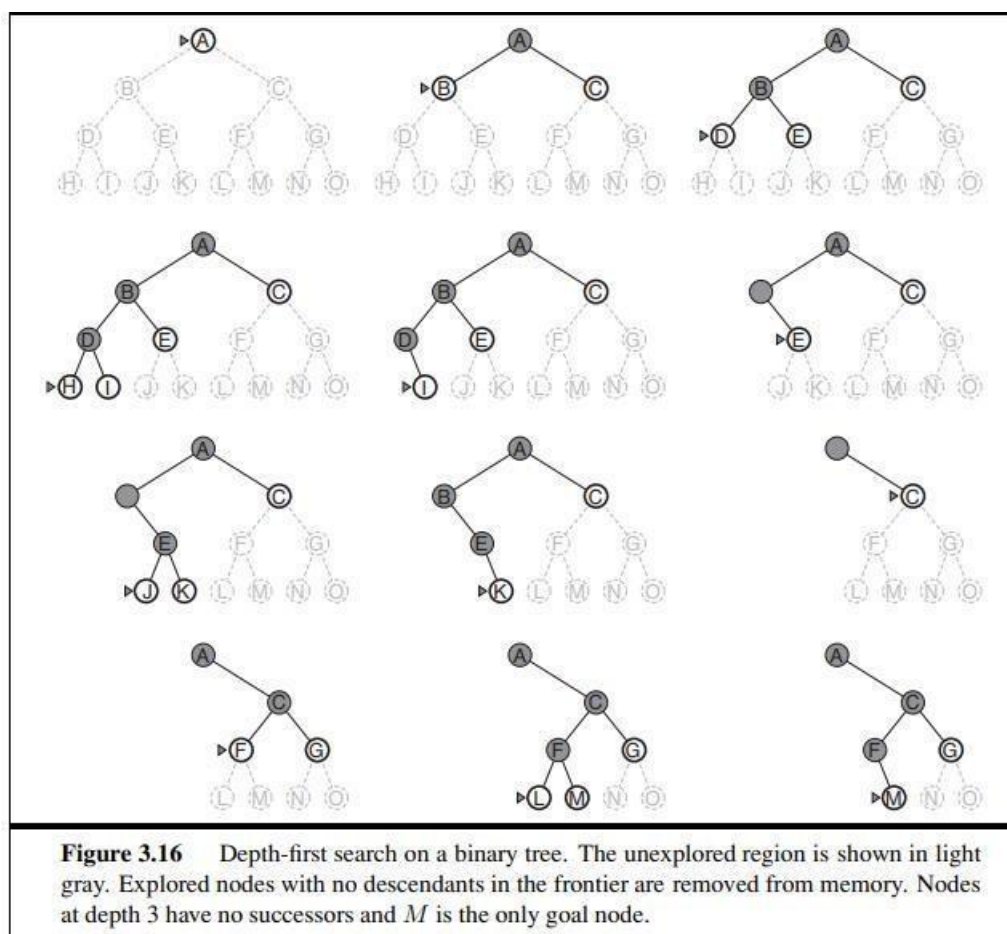
Figure 3.13 Time and memory requirements for breadth-first search. The numbers assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

so the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier. Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time. An exponential complexity bound such as $O(b^d)$ is scary. Figure 3.13 shows why. It lists, for various values of the solution depth d , the time and memory required for a breadth first search with branching factor $b = 10$. The table assumes that 1 million nodes can be generated per second and that a node requires 1000 bytes of storage. Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

Two lessons can be learned from Figure 3.13. First, the memory requirements are a bigger problem for breadth-first search than is the execution time. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Fortunately, other strategies require less memory. These strategies have time as a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it. In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

3.4.3 Depth-first search

Depth-first search always expands the deepest node in the current frontier of the search tree. The progress of the search is illustrated in Figure 3.16. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors. The depth-first search algorithm is an instance of the graph-search algorithm in Figure 3.7; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected. As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn. (A recursive depth-first algorithm incorporating a depth limit is shown in Figure 3.17.)



The properties of depth-first search depend strongly on whether the graph-search or tree-search

version is used. The graph-search version, which avoids repeated states and redundant paths, is complete infinite state spaces because it will eventually expand every node. The tree-search version, on the other hand.

MODULE - 3

3.5 INFORMED (HEURISTIC) SEARCH STRATEGIES

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 coupled 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.

3.5.1 Greedy best-first search

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

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

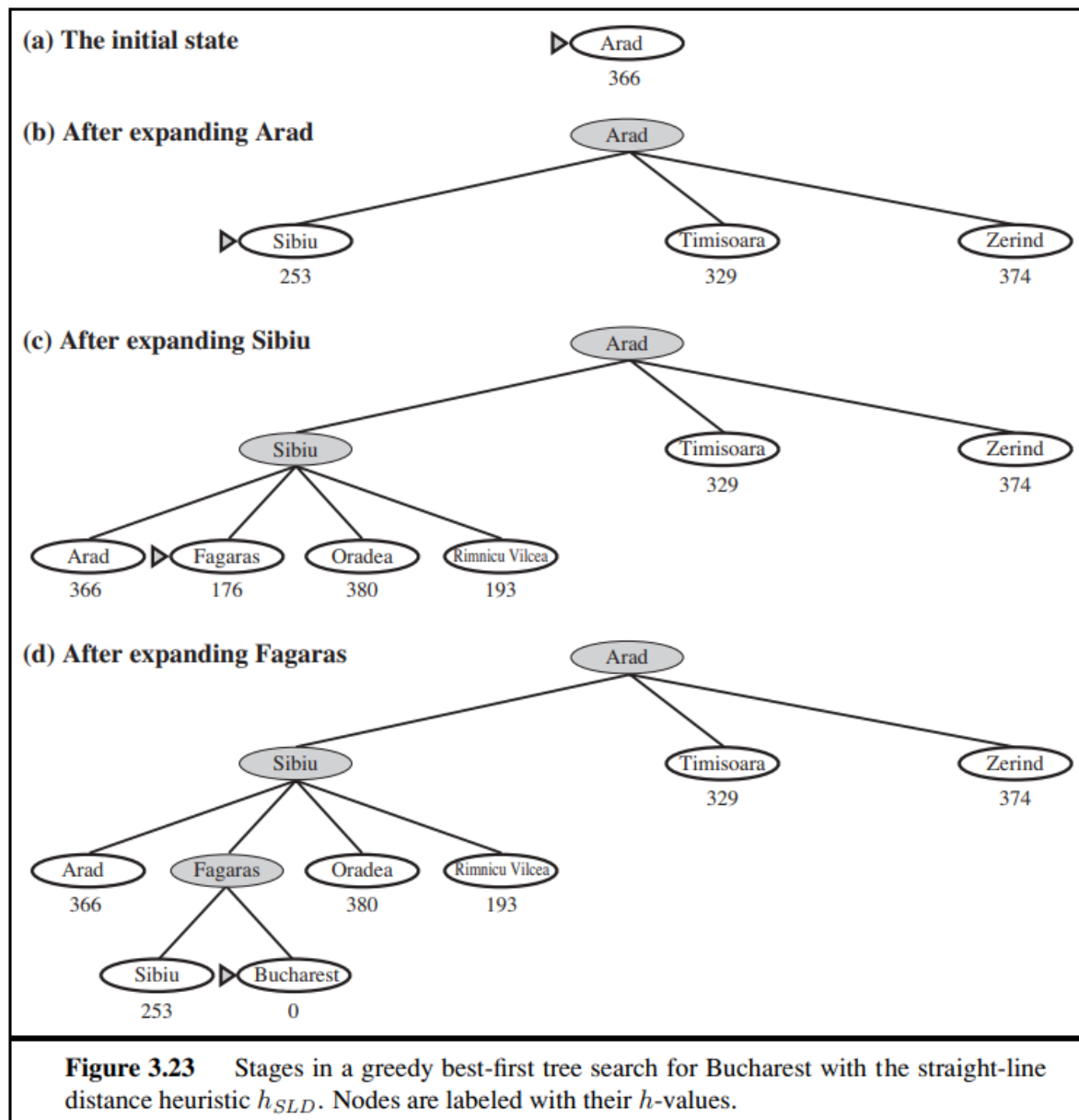
Taking the example of Route-finding problems in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in Figure 3.22. For example, the initial state is In (Arad), and the straight line distance heuristic 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
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

Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.

Figure 3.23 shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras,

because it is closest. Fagaras in turn generates Bucharest, which is the goal.



Evaluation Criterion of Greedy Search

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

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

3.5.2 A* search: Minimizing the total estimated solution cost:

The most widely known form of best-first search is called A* search (pronounced “A-star * search”). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal: $f(n) = g(n) + h(n)$. Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n

Conditions for optimality: Admissibility and consistency:

The first condition we require for optimality is that $h(n)$ be an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal. Because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, we have as an immediate consequence that $f(n)$ never overestimates the true cost of a solution along the current path through n .

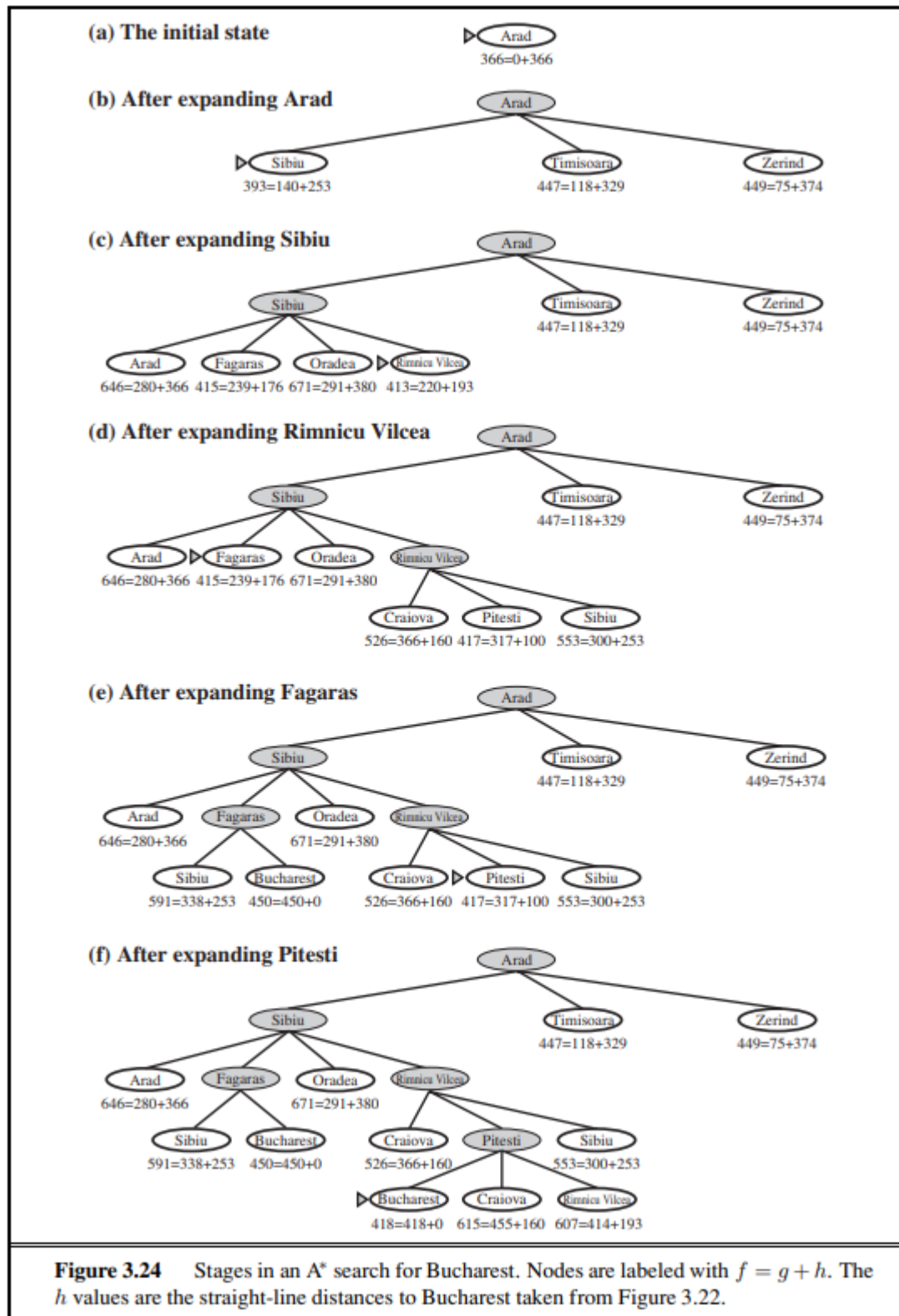
In Figure 3.24, we show the progress of an A* tree search for Bucharest. Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f -cost (450) is higher than that of Pitesti (417). Another way to say this is that there might be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A* to graph search.⁹ A heuristic $h(n)$ is consistent if, for every node n and every successor n^1 of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n^1 plus the estimated cost of reaching the goal from n^1 : $h(n) \leq c(n, a, n^1) + h(n^1)$. This is a form of the general triangle inequality, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n , n^1 , and the goal G_n closest to n .

3.5.3 Memory-bounded heuristic search

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A* (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f -cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest f -cost of any node that exceeded the cutoff on the previous iteration.

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 3.26.



Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f limit variable to keep track of the f -value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path.

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE),  $\infty$ )

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors  $\leftarrow$  []
  for each action in problem.ACTIONS(node.STATE) do
    add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure,  $\infty$ 
  for each s in successors do /* update f with value from previous search, if any */
    s.f  $\leftarrow$  max(s.g + s.h, node.f)
  loop do
    best  $\leftarrow$  the lowest f-value node in successors
    if best.f > f_limit then return failure, best.f
    alternative  $\leftarrow$  the second-lowest f-value among successors
    result, best.f  $\leftarrow$  RBFS(problem, best, min(f_limit, alternative))
    if result  $\neq$  failure then return result

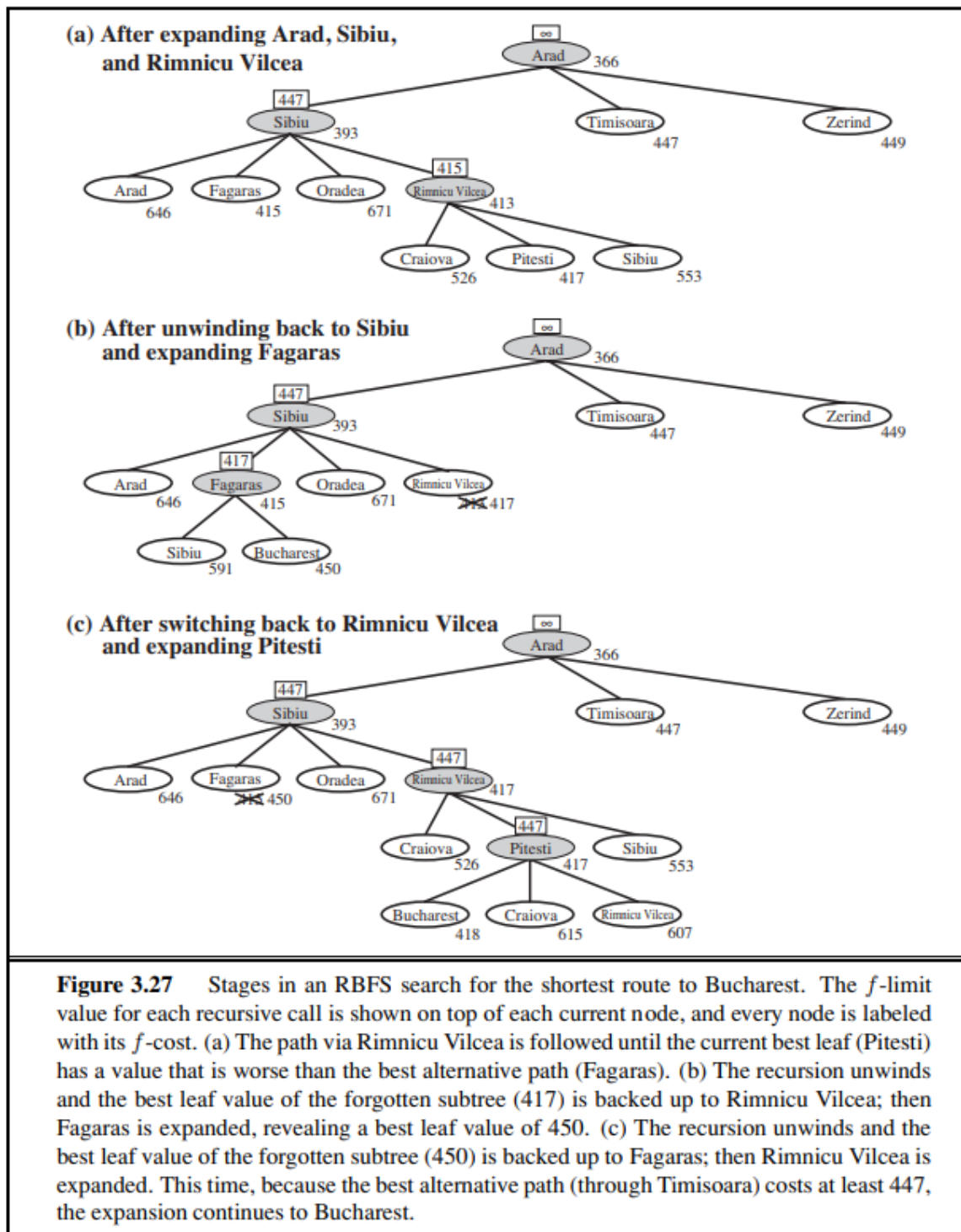
```

Figure 3.26 The algorithm for recursive best-first search.

As the recursion unwinds, RBFS replaces the *f*-value of each node along the path with a backed-up value—the best *f*-value of its children. In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 3.27 shows how RBFS reaches Bucharest. RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. In the example in Figure 3.27, RBFS follows the path via Rimnicu Vilcea, the “changes its mind” and tries Fagaras, and then changes its mind back again.

Like A* tree search, RBFS is an optimal algorithm if the heuristic function $h(n)$ is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. IDA* and RBFS suffer from using too little memory. Between iterations, IDA* retains only a single number: the current *f*-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it.

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. That is to say, memory limitations can make a problem intractable from the point of view of computation time. Although no current theory explains the tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.



3.5.4 Learning to search better

Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an object-level state space such as Romania. For example, the internal state of the A* algorithm consists of the current search tree. Each action in the metalevel state space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree.

3.6 Heuristic functions

- Heuristic functions in order to reach the goal node in a more prominent way.
- Therefore, there are several pathways in a search tree to reach the goal node from the current node.
- The selection of a good heuristic function matters certainly.
- A good heuristic function is determined by its efficiency.
- More is the information about the problem, more is the processing time.

Example:

- Consider the following 8-puzzle problem where we have a start state and a goal state.
- Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state. There can be four moves either left, right, up, or down.
- There can be several ways to convert the current/start state to the goal state, but we can use a heuristic function $h(n)$ to solve the problem more efficiently.

1	2	3
8	6	
7	5	4

Start State

1	2	3
8		4
7	6	5

Goal State

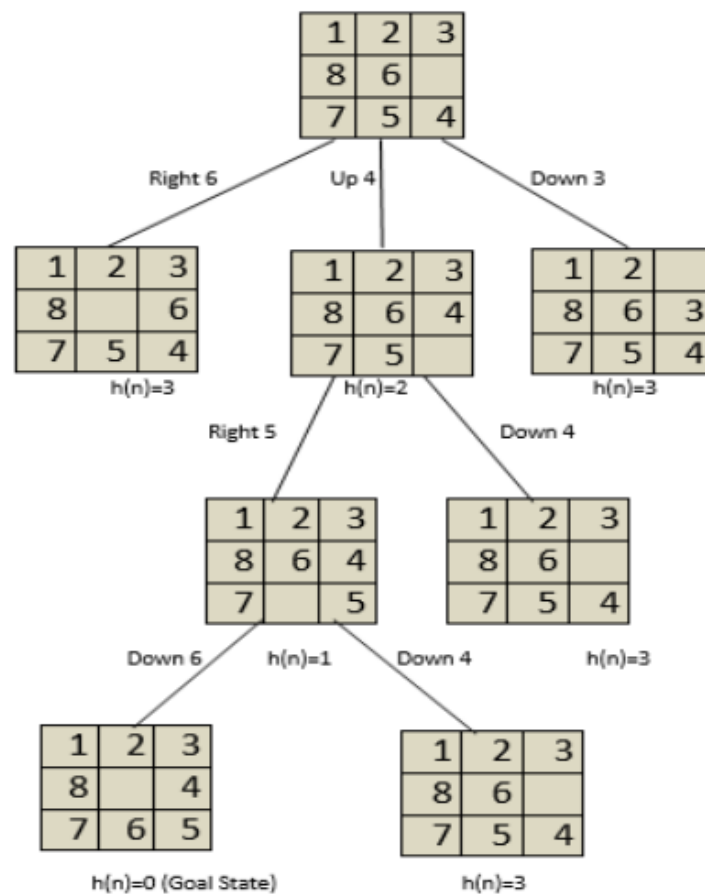
A heuristic function for the 8-puzzle problem is defined below:

$h(n)$ = Number of tiles out of position.

So, there is total of three tiles out of position i.e., 6, 5 and 4. Do not count the empty tile present in the goal state). i.e. $h(n)=3$. Now, we require to minimize the value of $h(n) = 0$.

We can construct a state-space tree to minimize the $h(n)$ value to 0, as shown below:

- It is seen from the above state space tree that the goal state is minimized from $h(n)=3$ to $h(n)=0$. However, we can create and use several heuristic functions as per the requirement.
- It is also clear from the above example that a heuristic function $h(n)$ can be defined as the information required to solve a given problem more efficiently.
- The information can be related to the nature of the state, cost of transforming from one state to another, goal node characteristics, etc., which is expressed as a heuristic function.



Properties of a Heuristic search Algorithm

Use of heuristic function in a heuristic search algorithm leads to following properties of a heuristic search algorithm:

- **Admissible Condition:** An algorithm is said to be admissible, if it returns an optimal solution.
- **Completeness:** An algorithm is said to be complete, if it terminates with a solution (if the solution exists).
- **Dominance Property:** If there are two admissible heuristic algorithms A1 and A2 having h_1 and h_2 heuristic functions, then A1 is said to dominate A2 if h_1 is better than h_2 for all the values of node n .
- **Optimality Property:** If an algorithm is complete, admissible, and dominating other algorithms, it will be the best one and will definitely give an optimal solution

Chapter 7:

Logical Agents

7.1 Knowledge-Based Agent in Artificial intelligence

An intelligent agent needs **knowledge** about the real world for taking decisions and **reasoning** to act efficiently.

Knowledge-based agents are those agents who have the capability of **maintaining an internal state of knowledge, reason over that** knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently.

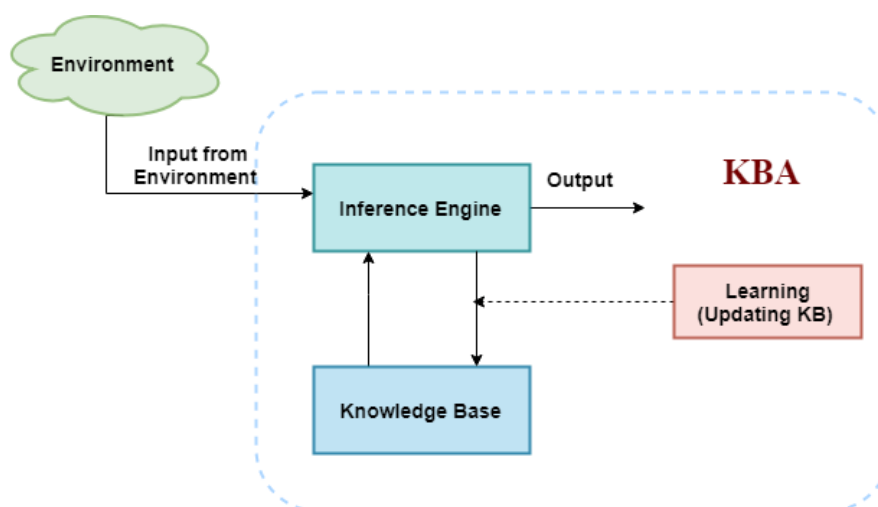
Knowledge-based agents are composed of two main parts:

- **Knowledge-base and**
- **Inference system.**

A knowledge-based agent must able to do the following:

- An agent should be able to represent states, actions, etc.
- An agent should be able to incorporate new percepts.
- An agent can update the internal representation of the world
- An agent can deduce the internal representation of the world
- An agent can deduce appropriate actions.

The architecture of knowledge-based agent:



The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.

Knowledge base: Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

Why use a knowledge base?

Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Inference system:

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.

Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:

- o Forward chaining
- o Backward chaining

Operations Performed by KBA

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

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

A generic knowledge-based agent:

Following is the structure outline of a generic knowledge-based agents program:

```
function KB-AGENT (percept) :  
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
```

The knowledge-based agent takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.

Each time when the function is called, it performs its three operations:

- Firstly it TELLS the KB what it perceives.
- Secondly, it asks KB what action it should take
- Third agent program TELLS the KB that which action was chosen.

The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.

The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.

MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

Various levels of knowledge-based agent:

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

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to

specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Approaches to designing a knowledge-based agent:

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

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

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

What is knowledge representation?

Humans are best at understanding, reasoning, and interpreting knowledge. Human knows things, which is knowledge and as per their knowledge they perform various actions in the real world. **But how machines do all these things comes under knowledge representation and reasoning.** Hence we can describe Knowledge representation as following:

- Knowledge representation and reasoning (KR, KRR) is the part of Artificial intelligence which concerned with AI agents thinking and how thinking contributes to

intelligent behavior of agents.

- It is responsible for representing information about the real world so that a computer can understand and can utilize this knowledge to solve the complex real world problems such as diagnosis a medical condition or communicating with humans in natural language.
- It is also a way which describes how we can represent knowledge in artificial intelligence. Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

What to Represent:

Following are the kind of knowledge which needs to be represented in AI systems:

- **Object:** All the facts about objects in our world domain. E.g., Guitars contains strings, trumpets are brass instruments.
- **Events:** Events are the actions which occur in our world.
- **Performance:** It describe behavior which involves knowledge
- **Meta-knowledge:** It is knowledge about what we know.
- **Facts:** Facts are the truths about the real world and what we represent.
- **Knowledge-Base:** The central component of the knowledge- based agents is the knowledge base. It is represented as KB. The Knowledgebase is a group of the Sentences (Here, sentences are used as a technical term and not identical with the English language).

Knowledge: Knowledge is awareness or familiarity gained by experiences of facts, data, and situations. Following are the types of knowledge in artificial intelligence:

Types of knowledge

- Following are the various types of knowledge:



1. Declarative Knowledge:

- Declarative knowledge is to know about something.
- It includes concepts, facts, and objects.
- It is also called descriptive knowledge and expressed in declarative sentences.
- It is simpler than procedural language.

2. Procedural Knowledge

- It is also known as imperative knowledge.
- Procedural knowledge is a type of knowledge which is responsible for knowing how to do something.
- It can be directly applied to any task.
- It includes rules, strategies, procedures, agendas, etc.
- Procedural knowledge depends on the task on which it can be applied.

3. Meta-knowledge:

- Knowledge about the other types of knowledge is called Meta- knowledge.

4. Heuristic knowledge:

- Heuristic knowledge is representing knowledge of some experts in a filed or subject.
- Heuristic knowledge is rules of thumb based on previous experiences, awareness

of approaches, and which are good to work but not guaranteed.

5. Structural knowledge:

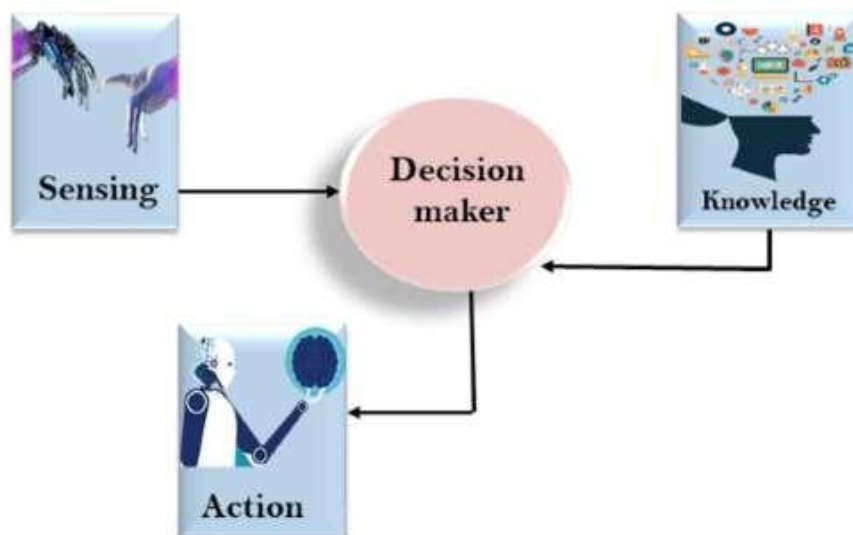
- Structural knowledge is basic knowledge to problem-solving.
- It describes relationships between various concepts such as kind of, part of, and grouping of something.
- It describes the relationship that exists between concepts or objects.

The relation between knowledge and intelligence:

Knowledge of real-worlds plays a vital role in intelligence and same for creating artificial intelligence. Knowledge plays an important role in demonstrating intelligent behavior in AI agents. An agent is only able to accurately act on some input when he has some knowledge or experience about that input.

Let's suppose if you met some person who is speaking in a language which you don't know, then how you will be able to act on that. The same thing applies to the intelligent behavior of the agents.

As we can see in below diagram, there is one decision maker which acts by sensing the environment and using knowledge. But if the knowledge part will not be present then, it cannot display intelligent behavior.



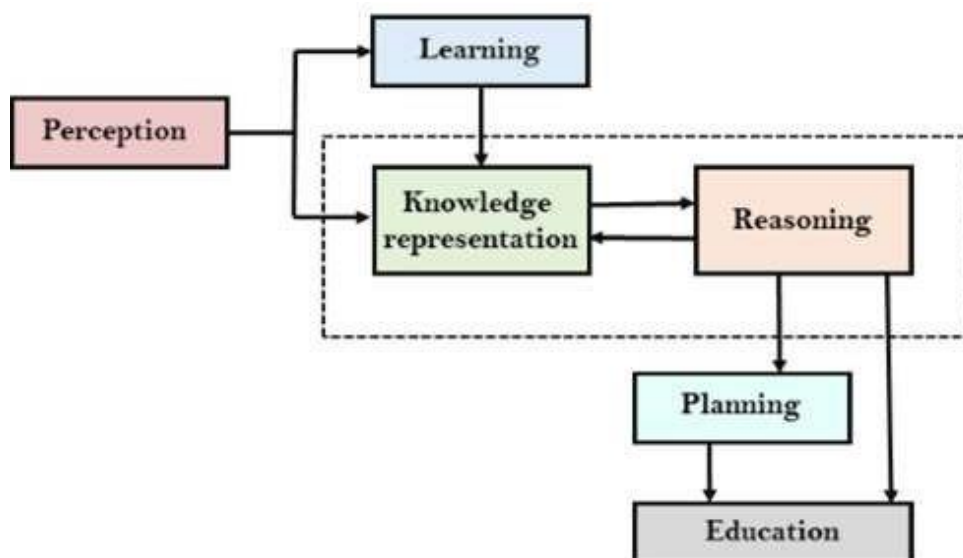
AI knowledge cycle:

An Artificial intelligence system has the following components for displaying

intelligent behavior:

- Perception
- Learning
- Knowledge Representation and Reasoning
- Planning
- Execution

The below diagram is showing how an AI system can interact with the real world and what components help it to show intelligence. AI system has Perception component by which it retrieves information from its environment. It can be visual, audio or another form of sensory input. The learning component is responsible for learning from data captured by Perception component. In the complete cycle, the main components are knowledge representation and Reasoning. These two components are involved in showing the intelligence in machine-like humans. These two components are independent with each other but also coupled together. The planning and execution depend on analysis of Knowledge representation and reasoning.



Approaches to knowledge representation:

There are mainly four approaches to knowledge representation, which are given below:

1. Simple relational knowledge:

- It is the simplest way of storing facts which uses the relational method, and each fact about a set of the object is set out systematically in columns.
- This approach of knowledge representation is famous in database systems where the relationship between different entities is represented.
- This approach has little opportunity for inference.

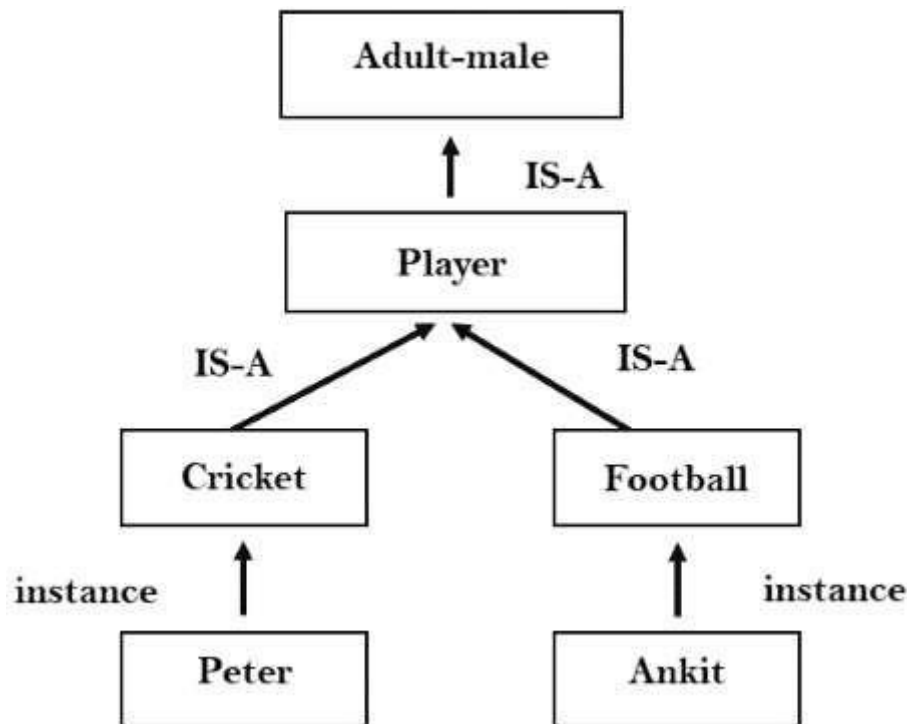
Example: The following is the simple relational knowledge representation.

Player	Weight	Age
Player1	65	23
Player2	58	18
Player3	75	24

2. Inheritable knowledge:

- In the inheritable knowledge approach, all data must be stored into a hierarchy of classes.
- All classes should be arranged in a generalized form or a hierarchal manner.
- In this approach, we apply inheritance property.
- Elements inherit values from other members of a class.
- This approach contains inheritable knowledge which shows a relation between instance and class, and it is called instance relation.
- Every individual frame can represent the collection of attributes and its value.
- In this approach, objects and values are represented in Boxed nodes.
- We use Arrows which point from objects to their values.

Example:



3. Inferential knowledge:

- Inferential knowledge approach represents knowledge in the form of formal logics.
- This approach can be used to derive more facts.
- It guaranteed correctness.
- **Example:** Let's suppose there are two statements:
 - a. Marcus is a man
 - b. All men are mortal
 Then it can represent as;

man(Marcus)

$\forall x = \text{man}(x) \text{-----} > \text{mortal}(x)s$

4. Procedural knowledge:

- Procedural knowledge approach uses small programs and codes which describes how to do specific things, and how to proceed.
- In this approach, one important rule is used which is **If-Then rule**.
- In this knowledge, we can use various coding languages such as **LISP language** and **Prolog language**.
- We can easily represent heuristic or domain-specific knowledge using this approach.
- But it is not necessary that we can represent all cases in this approach.

Requirements for knowledge Representation system:

A good knowledge representation system must possess the following properties.

1. Representational Accuracy:

KR system should have the ability to represent all kind of required knowledge.

2. Inferential Adequacy:

KR system should have ability to manipulate the representational structures to produce new knowledge corresponding to existing structure.

3. Inferential Efficiency:

The ability to direct the inferential knowledge mechanism into the most productive directions by storing appropriate guides.

4. Acquisitional efficiency- The ability to acquire the new knowledge easily using automatic methods.

The Wumpus World in Artificial intelligence

Wumpus world:

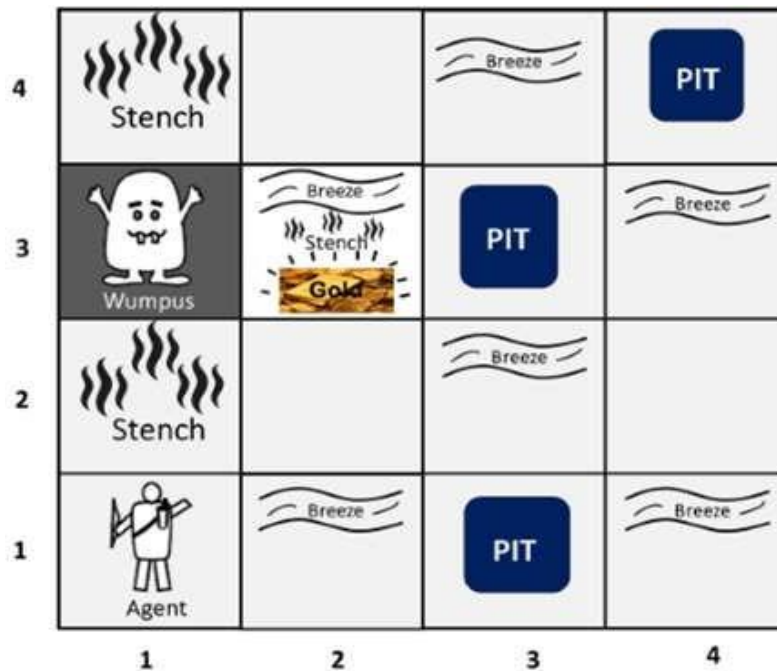
The Wumpus world is a simple world example to illustrate the worth of a knowledge-based agent and to represent knowledge representation. It was inspired by a video game **Hunt the Wumpus** by Gregory Yob in 1973.

The Wumpus world is a cave which has 4/4 rooms connected with passageways. So there are total 16 rooms which are connected with each other. We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room. The Wumpus can be shot by the agent, but the agent has a single arrow. In the Wumpus world, there are some Pits rooms which are bottomless, and if agent falls in Pits, then he will be stuck there forever. The exciting thing with this cave is that in one room there is a possibility of finding a heap of gold. So the agent goal is to find the gold and climb out the cave without fallen into Pits or eaten by Wumpus. The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit.

Note: *Here Wumpus is static and cannot move.*

Following is a sample diagram for representing the Wumpus world. It is showing some

rooms with Pits, one room with Wumpus and one agent at (1, 1) square location of the world.



There are also some components which can help the agent to navigate the cave. These components are given as follows:

- The rooms adjacent to the Wumpus room are smelly, so that it would have some stench.
- The room adjacent to PITs has a breeze, so if the agent reaches near to PIT, then he will perceive the breeze.
- There will be glitter in the room if and only if the room has gold.
- The Wumpus can be killed by the agent if the agent is facing to it, and Wumpus will emit a horrible scream which can be heard anywhere in the cave.

PEAS description of Wumpus world:

To explain the Wumpus world we have given PEAS description as below

Performance measure:

- +1000 reward points if the agent comes out of the cave with the gold.
- -1000 points penalty for being eaten by the Wumpus or falling into the pit.
- -1 for each action, and -10 for using an arrow.
- The game ends if either agent dies or came out of the cave.

Environment:

- A 4*4 grid of rooms.
- The agent initially in room square [1, 1], facing toward the right.
- Location of Wumpus and gold are chosen randomly except the first square

[1,1].

- Each square of the cave can be a pit with probability 0.2 except the first square.

Actuators:

- Left turn,
- Right turn
- Move forward
- Grab
- Release
- Shoot.

Sensors:

- The agent will perceive the **stench** if he is in the room adjacent to the Wumpus. (Not diagonally).
- The agent will perceive **breeze** if he is in the room directly adjacent to the Pit.
- The agent will perceive the **glitter** in the room where the gold is present.
- The agent will perceive the **bump** if he walks into a wall.
- When the Wumpus is shot, it emits a horrible **scream** which can be perceived anywhere in the cave
- These percepts can be represented as five element list, in which we will have different indicators for each sensor.
- Example if agent perceives stench, breeze, but no glitter, no bump, and no scream then it can be represented as:
[Stench, Breeze, None, None, None].

The Wumpus world Properties:

- **Partially observable:** The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
- **Deterministic:** It is deterministic, as the result and outcome of the world are already known.
- **Sequential:** The order is important, so it is sequential.
- **Static:** It is static as Wumpus and Pits are not moving.
- **Discrete:** The environment is discrete.
- **One agent:** The environment is a single agent as we have one agent only and Wumpus is not considered as an agent.

Exploring the Wumpus world:

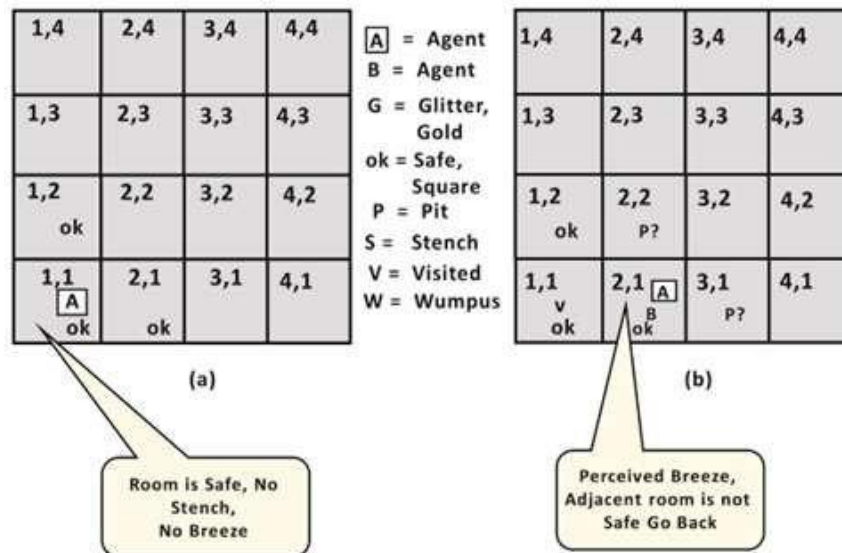
Now we will explore the Wumpus world and will determine how the agent will find its goal by applying logical reasoning.

Agent's First step:

Initially, the agent is in the first room or on the square [1,1], and we already know that this

room is safe for the agent, so to represent on the below diagram (a) that room is safe we will add symbol OK. Symbol A is used to represent agent, symbol B for the breeze, G for Glitter or gold, V for the visited room, P for pits, W for Wumpus.

At Room [1,1] agent does not feel any breeze or any Stench which means the adjacent squares are also OK.



Agent's second Step:

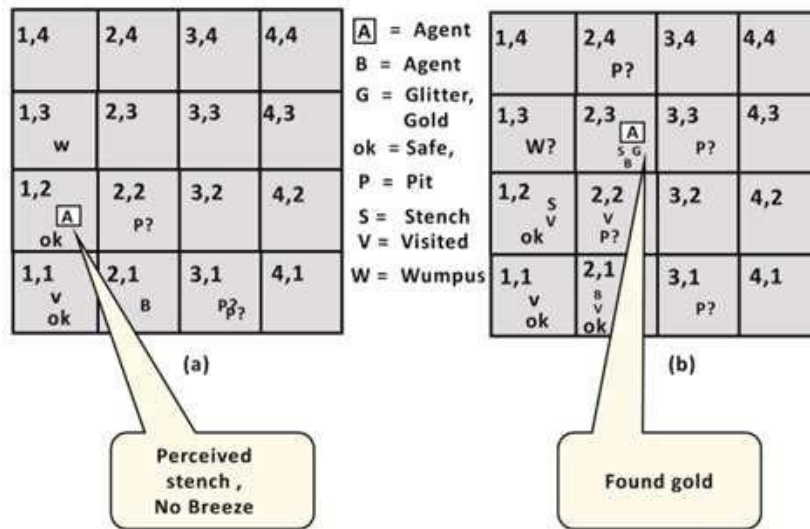
Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives some breeze which means Pit is around this room. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?

Now agent will stop and think and will not make any harmful move. The agent will go back to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

Agent's third step:

At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby. But Wumpus cannot be in the room [1,1] as by rules of the game, and also not in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore agent infers that Wumpus is in the room [1,3], and

in current state, there is no breeze which means in [2,2] there is no Pit and no Wumpus. So it is safe, and we will mark it OK, and the agent moves further in [2,2].



Agent's fourth step:

At room [2,2], here no stench and no breezes present so let's suppose agent decides to move to [2,3]. At room [2,3] agent perceives glitter, so it should grab the gold and climb out of the cave.

Propositional logic in Artificial intelligence

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

- It is Sunday.
- The Sun rises from West (False proposition)
- $3+3=7$ (False proposition)
- 5 is a prime number.

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can

use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.

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

Syntax of propositional logic:

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

a. **Atomic Propositions**

b. **Compound propositions**

. **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

a) $2+2$ is 4, it is an atomic proposition as it is a **true** fact.

b) "The Sun is cold" is also a proposition as it is a **false** fact.

a. **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

a) "It is raining today, and street is wet."

b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a

sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. **Negation:** A sentence such as $\neg P$ is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has A connective such as, $P \wedge Q$ is called a conjunction.

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

P= Rohan is intelligent,

Q= Rohan is hardworking. $\rightarrow PA Q$.

3. **Disjunction:** A sentence which has $\vee$ connective, such as $P \vee Q$. is called disjunction, where P and Q are the propositions.

Example: "Ritika is a doctor or Engineer",

Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as $P \vee Q$.

4. **Implication:** A sentence such as $P \rightarrow Q$, is called an implication.

Implications are also known as if–then rules. It can be represented as

If it is raining, then the street is wet.

Let P= It is raining, and Q= Street is wet, so it is represented as $P \rightarrow Q$

5. **Biconditional:** A sentence such as $P \Leftrightarrow Q$ is a Biconditional sentence, example If I am breathing, then I am alive

P= I am breathing, Q= I am alive, it can be represented as $P \Leftrightarrow Q$.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
$\wedge$	AND	Conjunction	$A \wedge B$
$\vee$	OR	Disjunction	$A \vee B$
$\rightarrow$	Implies	Implication	$A \rightarrow B$
$\Leftrightarrow$	If and only if	Biconditional	$A \Leftrightarrow B$
$\neg$ or $\sim$	Not	Negation	$\neg A$ or $\neg B$

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called Truth table. Following are the truth table for all logical connectives:

For Biconditional:

P	Q	$P \leftrightarrow Q$
True	True	True
True	False	False
False	True	False
False	False	True

For Negation:

P	$\neg P$
True	False
False	True

For Conjunction:

P	Q	$P \wedge Q$
True	True	True
True	False	False
False	True	False
False	False	False

For disjunction:

P	Q	$P \vee Q$
True	True	True
False	True	True
True	False	True
False	False	False

For Implication:

P	Q	$P \rightarrow Q$
True	True	True
True	False	False
False	True	True
False	False	True

Truth table with three propositions:

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8n Tuples as we have taken three proposition symbols.

P	Q	R	$\neg R$	$P \vee Q$	$P \vee Q \rightarrow \neg R$
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

Precedence of connectives:

Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem.

Following is the list of the precedence order for operators:

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

Note: For better understanding use parenthesis to make sure of the correct interpretations. Such as $\neg R \vee Q$, It can be interpreted as $(\neg R) \vee Q$.

Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as $A \Leftrightarrow B$. In

below truth table we can see that column for $\neg A \vee B$ and $A \rightarrow B$, are identical hence A is Equivalent to B

A	B	$\neg A$	$\neg A \vee B$	$A \rightarrow B$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Properties of Operators:

Commutativity:

$P \wedge Q = Q \wedge P$, or $P \vee Q = Q \vee P$.

Associativity:

$(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$, $(P \vee Q) \vee R = P \vee (Q \vee R)$

Identity element:

$P \wedge \text{True} = P$,

$P \vee \text{True} = \text{True}$.

Distributive:

$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$. $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$. DE Morgan's Law:

$\neg (P \wedge Q) = (\neg P) \vee (\neg Q)$

$\neg (P \vee Q) = (\neg P) \wedge (\neg Q)$.

Double-negation elimination:

$\neg (\neg P) = P$.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic.

Example:

- All the girls are intelligent.
 - Some apples are sweet.
- b. Propositional logic has limited expressive power.
- c. In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

MODULE - 4

4.1 First Order Logic: Representation Revisited

Propositional logic is too puny a language to represent knowledge of complex environments in a concise way. In this chapter, we examine first-order logic.

Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent, in a direct sense, only computational processes. Data structures within programs can represent facts; for example, a program could use a 4×4 array to represent the contents of the wumpus world. Thus, the Programming language statement `World [2,2]←Pit` is a fairly natural way to assert that there is a pit in square [2,2]. (Such representations might be considered ad hoc; database systems were developed precisely to provide a more general, domain-independent way to store and retrieve facts.) What programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain.

This procedural approach can be contrasted with the declarative nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain independent. A second drawback of data structures in programs (and of databases, for that matter) is the lack of any easy way to say, for example, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].” Programs can store a single value for each variable, and some systems allow the value to be “unknown,” but they lack the expressiveness required to handle partial information.

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. Propositional logic has a third property that is desirable in representation languages, namely, compositionality. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, the meaning of “ $S_{1,4} \wedge S_{1,2}$ ” is related to the meanings “ $S_{1,4}$ ” and “ $S_{1,2}$.” It would be very strange if “ $S_{1,4}$ ” meant that there is a stench in square [1,4] and “ $S_{1,2}$ ” meant that there is a stench in square [1,2], but “ $S_{1,4} \wedge S_{1,2}$ ” meant that France

and Poland drew 1-1 in last week's ice hockey qualifying match. Clearly, non compositionality makes life much more difficult for the reasoning system.

However, propositional logic lacks the expressive power to concisely describe an environment with many objects. For example, we were forced to write a separate rule about breezes and pits for each square, such as $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.

In English, on the other hand, it seems easy enough to say, once and for all, "Squares adjacent to pits are breezy." The syntax and semantics of English somehow make it possible to describe the environment concisely.

4.1.1 The language of thought

The modern view of natural language is that it serves as a medium for communication rather than pure representation. When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the context in which the sentence was spoken. Clearly, one could not store a sentence such as "Look!" in a knowledge base and expect to recover its meaning without also storing a representation of the context—which raises the question of how the context itself can be represented. Natural languages also suffer from ambiguity, a problem for a representation language.

As Pinker (1995) puts it as "When people think about spring, surely they are not confused as to whether they are thinking about a season or something that goes boing—and if one word can correspond to two thoughts, thoughts can't be words." The famous Sapir-Whorf hypothesis claims that our understanding of the world is strongly influenced by the language we speak. Whorf (1956) wrote "We cut nature up, organize it into concepts, and ascribe significances as we do, largely because we are parties to an agreement to organize it this way—an agreement that holds throughout our speech community and is codified in the patterns of our language." It is certainly true that different speech communities divide up the world differently.

In a first-order logic reasoning system that uses CNF, we can see that the linguistic form

$\neg(A \vee B)$ and $\neg A \wedge \neg B$ are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form. Can we do that with the human brain? Until recently the answer was “no,” but now it is “maybe.” Mitchell et al. (2008) put subjects in an fMRI (functional magnetic resonance imaging) machine, showed them words such as “celery,” and imaged their brains. The researchers were then able to train a computer program to predict, from a brain image, what word the subject had been presented with. Given two choices (e.g., “celery” or “airplane”), the system predicts correctly 77% of the time. The system can even predict at above-chance levels for words it has never seen an fMRI image of before (by considering the images of related words) and for people it has never seen before (proving that fMRI reveals some level of common representation across people). This type of work is still in its infancy, but fMRI (and other imaging technology such as intracranial electrophysiology (Sahin et al., 2009)) promises to give us much more concrete ideas of what human knowledge representations are like.

4.1.2 Combining the best of formal and natural languages

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks. When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to objects (squares, pits, wumpuses) and verbs and verb phrases that refer to relations among objects (is breezy, is adjacent to, shoots). Some of these relations are functions—relations in which there is only one “value” for a given “input.” It is easy to start listing examples of objects, relations, and functions:

- **Objects:** people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries.
- **Relations:** these can be unary relations or properties such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between.
- **Functions:** father of, best friend, third inning of, one more than, beginning of. Indeed, almost any assertion can be thought of as referring to objects and properties or relations. Some examples follow:
 - **“One plus two equals three”**

Objects: one, two, three, one plus two; **Relation:** equals; **Function:** plus. (“One plus two” is a name for the object that is obtained by applying the function “plus” to the objects “one” and “two.” “Three” is another name for this object.)

- “Squares neighboring the wumpus are smelly.”

Objects: wumpus, squares; **Property:** smelly; **Relation:** neighboring.

“Evil King John ruled England in 1200.”

Objects: John, England, 1200; **Relation:** ruled; **Properties:** evil, king.

The **primary difference between propositional and first-order logic** lies in the **ontological** commitment made by each language—that is, what it assumes about the nature of reality. Mathematically, this commitment is expressed through the nature of the formal models with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false, and each model assigns true or false to each proposition symbol.

Higher-order logic views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about all relations—for example, one could wish to define what it means for a relation to be transitive. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

A logic can also be characterized by its **epistemological** commitments—the possible states of knowledge that it allows with respect to each fact. In both propositional and first order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using probability theory, on the other hand, can have any degree of belief, ranging from 0 (total disbelief) to 1 (total belief). 3 **For example**, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75. The ontological and epistemological commitments of five different logics are summarized in Figure 4.1.

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 $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure: 4.1 Formal languages and their ontological and epistemological commitments.

4.2 Syntax and Semantics of First Order logic

We begin this section by specifying more precisely the way in which the possible worlds of first-order logic reflect the ontological commitment to objects and relations.

4.2.1 Models for first-order logic

Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic are much more interesting. First, they have objects in them! The domain of a model is the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object. Mathematically speaking, it doesn't matter what these objects are—all that matters is how many there are in each particular model—but for pedagogical purposes we'll use a concrete example. Figure 4.2 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 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 }.

(Here we have named the objects in English, but you may, if you wish, mentally substitute the pictures for the names.) 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. The model also contains unary relations, or properties: the “person” property is true of both Richard and John; the “king” property is true only of John (presumably because Richard is dead at this point); and the “crown” property is true only of the crown. 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:

$\{\text{Richard the Lionheart}\} \rightarrow \text{Richard's left leg}(A)$

$\{\text{King John}\} \rightarrow \text{John's left leg}(B)$

Strictly speaking, models in first-order logic require total functions, that is, there must be a value for every input tuple. Thus, the crown must have a left leg and so must each of the left legs. There is a technical solution to this awkward problem involving an additional “invisible”

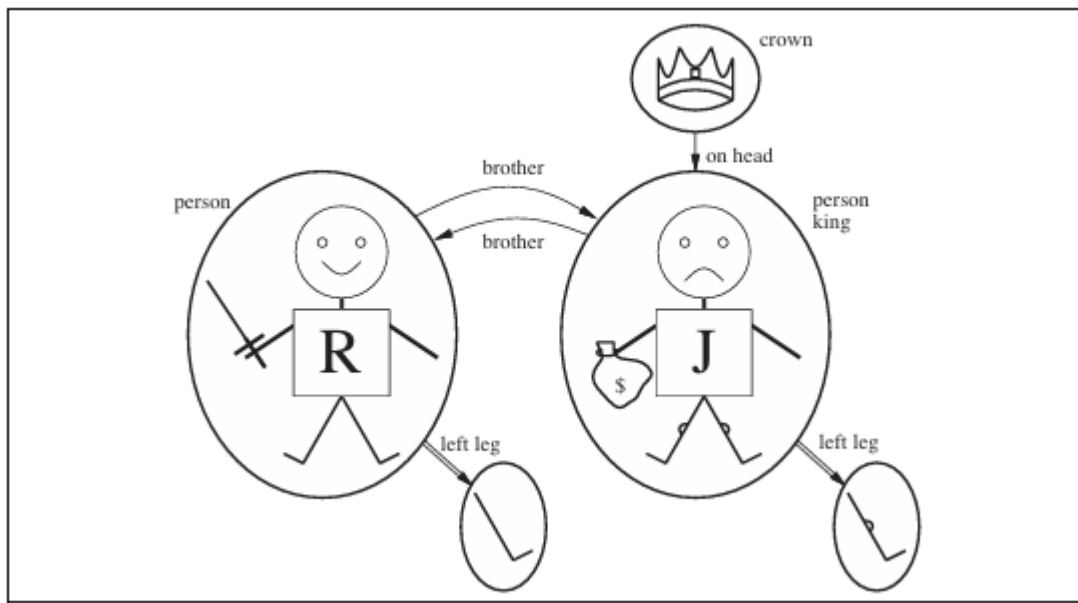


Figure 4.2: A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

4.2.2 Symbols and interpretations

We turn now to the syntax of first-order logic. The impatient reader can obtain a complete description from the formal grammar in Figure 4.3. The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: constant symbols, which stand for objects; predicate

symbols, which stand for relations; and function symbols, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols Richard and John; the predicate symbols Brother, On Head, Person, King, and Crown; and the function symbol LeftLeg. As with proposition symbols, the choice of names is entirely up to the user. Each predicate and function symbol comes with an arity that fixes the number of arguments.

As in propositional logic, every model must provide the information required to determine if any given sentence is true or false. Thus, in addition to its objects, relations, and functions, each model includes an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols. One possible interpretation for our example—which a logician would call 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 as mentioned in Equation (A); On Head refers to the “on head” relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.
- LeftLeg refers to the “left leg” function, that is, the mapping given in Equation (B).

There are many other possible interpretations, of course. For example, one interpretation maps Richard to the crown and John to King John’s left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols Richard and John.

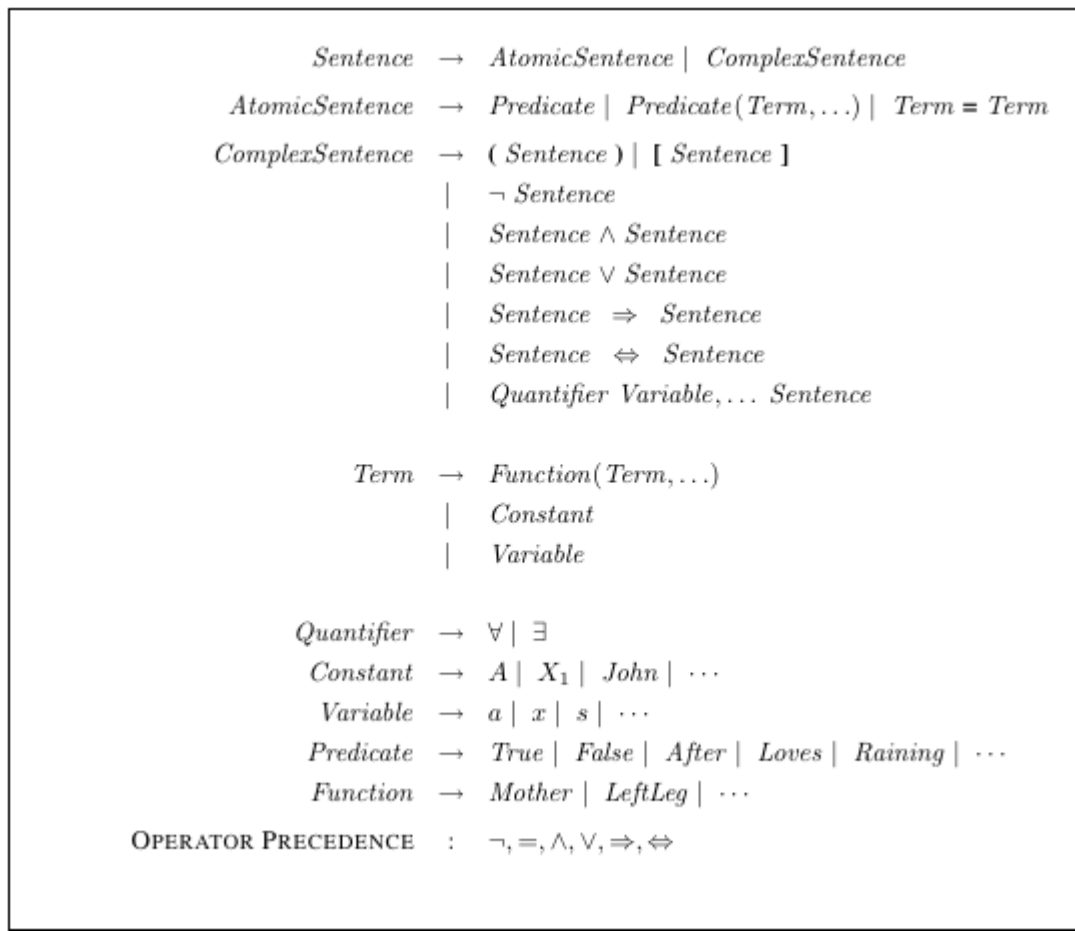


Figure 4.3: The syntax of first-order logic with equality, specified in Backus-Naur form. Operator precedence's are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

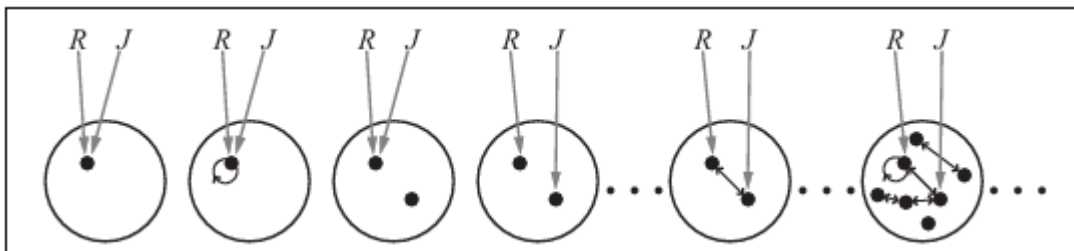


Figure 4.4: Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects. Just as with propositional logic, entailment, validity, and so on are defined in terms of all possible models. To get an idea of what the

set of all possible models looks like, see Figure 4.4. It shows that models vary in how many objects they contain—from one up to infinity—and in the way the constant symbols map to objects. If there are two constant symbols and one object, then both symbols must refer to the same object; but this can still happen even with more objects. When there are more objects than constant symbols, some of the objects will have no names. Because the number of possible models is unbounded, checking entailment by the enumeration of all possible models is not feasible for first-order logic (unlike propositional logic). Even if the number of objects is restricted, the number of combinations can be very large. For example in Figure 4.4, there are 137,506,194,466 models with six or fewer objects.

4.2.3 Terms

A term is a logical expression that refers 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 `LeftLeg(John)`. In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a “subroutine call” that “returns a value.” There is no `LeftLeg` subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of `Left Leg`.

This is something that cannot be done with subroutines in programming languages. The formal semantics of terms is straightforward. Consider a term $f(t_1, \dots, t_n)$. The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them $d_1, \dots, d_n$); and the term as a whole refers to the object that is the value of the function F applied to $d_1, \dots, d_n$. For example, suppose the `Left Leg` function symbol refers to the function shown in Equation (B) and `John` refers to King John, then `LeftLeg(John)` refers to King John’s left leg. In this way, the interpretation fixes the referent of every term.

4.2.4 Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to

relations, we can put them together to make atomic sentences that state facts. An atomic sentence (or atom for short) is formed from a predicate symbol optionally followed by parenthesized list of terms, such as *Brother(Richard,John)*.

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John.⁶ 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 (again, under a suitable interpretation).

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

4.2.5 Complex sentences

We can use logical connectives to construct more complex sentences, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of Figure 4.2 under our intended interpretation:

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

4.2.6 Quantifiers

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. Quantifiers let us do this. First-order logic contains two standard quantifiers, called universal and existential.

1) Universal quantification ($\forall$)

The expression of general rules in propositional logic. Rules such as “Squares neighboring the wumpus are smelly” and “All kings are persons” are the bread and butter of first-order logic. The second rule, “All kings are persons,” is written in first-order logic as $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$.

$\forall$ is usually pronounced “For all ...”. (Remember that the upside-down A stands 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. By convention, variables are lowercase letters. 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.

Intuitively, the sentence $\forall xP$, where P is any logical expression, says that P is true for every object x . More precisely, $\forall xP$ is true in a given model if P is true in all possible extended interpretations constructed from the interpretation given in the model, where each extended interpretation specifies a domain element to which x refers. This sounds complicated, but it is really just a careful way of stating the intuitive meaning of universal quantification. Consider the model shown in Figure 4.2 and the intended interpretation that goes with it. We can extend the interpretation in five ways:

$x \rightarrow \text{Richard the Lionheart},$

$x \rightarrow \text{King John},$

$x \rightarrow \text{Richard's left leg},$

$x \rightarrow \text{John's left leg},$

$x \rightarrow \text{the crown}.$

The universally quantified sentence $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$ is true in the original model if the sentence $\text{King}(x) \Rightarrow \text{Person}(x)$ 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 $\Rightarrow$ Richard the Lionheart is a person.

King John is a king $\Rightarrow$ King John is a person.

Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person.

John's left leg is a king $\Rightarrow$ John's left leg is a person.

The crown is a king $\Rightarrow$ the crown is a person.

Let us look carefully at this set of assertions. Since, in our model, King John is the only king, the second sentence asserts that he is a person, as we would hope. But what about the other four sentences, which appear to make claims about legs and crowns? Is that part of the meaning of “All kings are persons”? In fact, the other four assertions are true in the model, but make no claim whatsoever about the personhood qualifications of legs, crowns, or indeed Richard. This is because none of these objects is a king. Looking at the truth table for $\Rightarrow$ we see that the implication is true whenever its premise is false – regardless of the truth of the conclusion. Thus, by asserting the universally quantified sentence, which is equivalent to asserting a whole list of individual implications, we end up asserting the conclusion of the rule just for those objects for whom the premise is true and saying nothing at all about those individuals for whom the premise is false. Thus, the truth-table definition of $\Rightarrow$ turns out to be perfect for writing general rules with universal quantifiers.

A common mistake, made frequently even by diligent readers who have read this paragraph several times, is to use conjunction instead of implication. The sentence

$$\forall x \text{ King}(x) \wedge \text{Person}(x)$$

would be equivalent to asserting

Richard the Lionheart is a king $\wedge$ Richard the Lionheart is a person,

King John is a king $\wedge$ King John is a person,

Richard's left leg is a king $\wedge$ Richard's left leg is a person,

and so on. Obviously, this does not capture what we want.

2) Existential quantification ($\exists$)

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

$$\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John}).$$

$\exists x$ is pronounced “There exists an x such that ...” or “For some x ...”. Intuitively, the sentence $\exists x P$ says that P is true for at least one object x . More precisely, $\exists x P$ 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 $\wedge$ Richard the Lionheart is on John's head;

King John is a crown $\wedge$ King John is on John's head;

Richard's left leg is a crown $\wedge$ Richard's left leg is on John's head;

John's left leg is a crown $\wedge$ John's left leg is on John's head;

The crown is a crown $\wedge$ 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. Notice that, by our definition, the sentence would also be true in a model in which King John was wearing two crowns. This is entirely consistent with the original sentence “King John has a crown on his head.”

Just as $\Rightarrow$ appears to be the natural connective to use with $\forall$, $\wedge$ is the natural connective to use with $\exists$. Using $\wedge$ as the main connective with $\forall$ led to an overly strong statement in the example in the previous section; using $\Rightarrow$ with $\exists$ usually leads to a very weak statement, indeed. Consider the following sentence:

$\exists x \text{Crown}(x) \Rightarrow \text{OnHead}(x, \text{John})$.

On the surface, this might look like a reasonable rendition of our sentence. 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 $\Rightarrow$ Richard the Lionheart is on John ' s head;

King John is a crown $\Rightarrow$ King John is on John ' s head;

Richard ' s left leg is a crown $\Rightarrow$ 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; hence such sentences really do not say much at all.

3) Nested quantifiers

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

$\forall x \forall y \text{Brother}(x,y) \Rightarrow \text{Sibling}(x,y)$.

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, we can write

$\forall x,y \text{Sibling}(x,y) \Leftrightarrow \text{Sibling}(y,x)$.

In other cases we will have mixtures. “Everybody loves somebody” means that for every person, there is someone that person loves:

$\forall x \exists y \text{Loves}(x,y)$.

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

$\exists y \forall x \text{Loves}(x,y)$

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. $\forall x (\exists y \text{Loves}(x,y))$ says that everyone has a particular property, namely, the property that they love someone. On the other hand, $\exists y (\forall x \text{Loves}(x,y))$ says that someone in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name.

Consider the sentence

$\forall x (\text{Crown}(x) \vee (\exists x \text{Brother}(\text{Richard}, x)))$.

Here the x in $Brother(Richard, x)$ is existentially quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification. Another way to think of it is this: $\exists x Brother(Richard, x)$ is a sentence about Richard (that he has a brother), not about x ; so putting a $\forall x$ outside it has no effect. It could equally well have been written $\exists z Brother(Richard, z)$. Because this can be a source of confusion, we will always use different variable names with nested quantifiers.

4) Connections between $\forall$ and $\exists$

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

$\forall x \neg Likes(x, Parsnips)$ is equivalent to $\neg \exists x Likes(x, Parsnips)$

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

$\forall x Likes(x, IceCream)$ is equivalent to $\neg \exists x \neg Likes(x, IceCream)$.

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

$$\begin{array}{ll} \forall x \neg P & \equiv \neg \exists x P & \neg(P \vee Q) & \equiv \neg P \wedge \neg Q \\ \neg \forall x P & \equiv \exists x \neg P & \neg(P \wedge Q) & \equiv \neg P \vee \neg Q \\ \forall x P & \equiv \neg \exists x \neg P & P \wedge Q & \equiv \neg(\neg P \vee \neg Q) \\ \exists x P & \equiv \neg \forall x \neg P & P \vee Q & \equiv \neg(\neg P \wedge \neg Q) \end{array}$$

Thus, we do not really need both $\forall$ and $\exists$, just as we do not really need both $\wedge$ and $\vee$. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

4.2.7 Equality

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

$Father(John) = Henry$

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

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

$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard}) \wedge \neg(x=y)$.

The sentence

$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard})$

does not have the intended meaning. In particular, it is true in the model of Figure 4.2, where Richard has only one brother. To see this, consider the extended interpretation in which both x and y are assigned to King John. The addition of $\neg(x=y)$ rules out such models. The notation $x=y$ is sometimes used as an abbreviation for $\neg(x \neq y)$

4.2.8 An alternative semantics?

Suppose that we believe that Richard has two brothers, John and Geoffrey.⁸ Can we capture this state of affairs by asserting

$\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard})$?

Not quite. First, this assertion is true in a model where Richard has only one brother— we need to add $\text{John} = \text{Geoffrey}$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

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

For many purposes, this seems much more cumbersome than the corresponding natural language expression. As a consequence, humans may make mistakes in translating their knowledge into first-order logic, resulting in unintuitive behaviors from logical reasoning systems that use the knowledge. Can we devise a semantics that allows a more straightforward logical expression?

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object—the so-called unique-names assumption. Second, we assume that atomic sentences not known to be true are in fact false—the closed-world assumption. Finally, we invoke domain closure, meaning that each model contains no more domain elements than those named by the constant symbols. Under the resulting semantics, which we call database semantics to distinguish it from the standard semantics of

first-order logic, the sentence Equation (B) does indeed state that Richard's two brothers are John and Geoffrey.

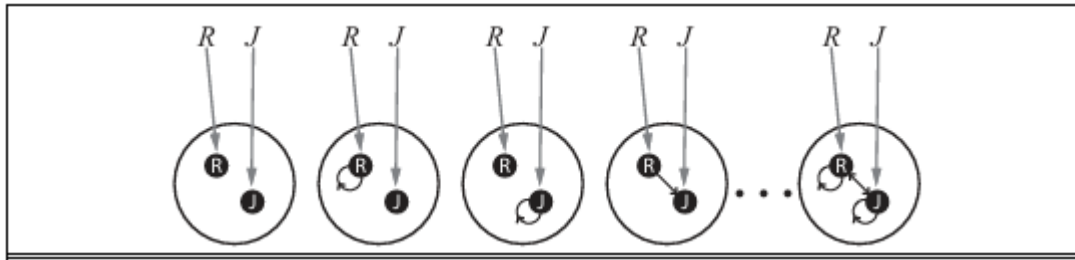


Figure 4.5: Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

It is instructive to consider the set of all possible models under database semantics for the same case as shown in Figure 4.4. Figure 4.5 shows some of the models, ranging from the model with no tuples satisfying the relation to the model with all tuples satisfying the relation. With two objects, there are four possible two-element tuples, so there are $2^4 = 16$ different subsets of tuples that can satisfy the relation. Thus, there are 16 possible models in all—a lot fewer than the infinitely many models for the standard first-order semantics.

On the other hand, the database semantics requires definite knowledge of what the world contains. This example brings up an important point: there is no one “correct” semantics for logic. The usefulness of any proposed semantics depends on how concise and intuitive it makes the expression of the kinds of knowledge we want to write down, and on how easy and natural it is to develop the corresponding rules of inference. Database semantics is most useful when we are certain about the identity of all the objects described in the knowledge base and when we have all the facts at hand; in other cases, it is quite awkward. We assume the standard semantics while noting instances in which this choice leads to cumbersome expressions.

4.3 Using First Order Logic

We begin with a brief description of the TELL/ASK interface for first-order knowledge bases. Then we look at the domains of family relationships, numbers, sets, and lists, and at the wumpus world.

4.3.1 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, Richard is a person, and all kings are persons:

TELL(KB, King(John)) .

TELL(KB, Person(Richard)) .

TELL(KB, $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$) .

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

ASK(KB, King(John))

returns true. Questions asked with ASK are called queries or goals. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively.

For example, given the two preceding assertions, the query

ASK(KB, Person(John))

should also return true. We can ask quantified queries, such as

ASK(KB, $\exists x \text{ Person}(x)$)

The answer is true, but this is perhaps not as helpful as we would like. It is rather like answering “Can you tell me the time?” with “Yes.” If we want to know what value of x makes the sentence true, we will need a different function, ASKVARs, which we call with

ASKVARs(KB, Person(x))

and which yields a stream of answers. In this case there will be two answers: $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. Such an answer is called a substitution or binding list. ASKVARs is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values. That is not the case with first-order logic; if KB has been told $\text{King}(\text{John}) \vee \text{King}(\text{Richard})$, then there is no binding to x for the query $\exists x \text{ King}(x)$, even though the query is true.

4.3.2 The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as “Elizabeth is the mother of Charles” and “Charles is the father of William” and rules such as “One’s grandmother is the mother of one’s parent.” Clearly, the objects in our domain are people. We have two unary predicates, Male and Female. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son,

Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle. We use functions for Mother and Father, because every person has exactly one of each of these (at least according to nature's design).

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

$$\forall m, c \text{ Mother}(c)=m \Leftrightarrow \text{Female}(m) \wedge \text{Parent}(m, c).$$

One's husband is one's male spouse:

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

Male and female are disjoint categories:

$$\forall x \text{ Male}(x) \Leftrightarrow \neg \text{Female}(x).$$

Parent and child are inverse relations:

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

A grandparent is a parent of one's parent:

$$\forall g, c \text{ Grandparent}(g, c) \Leftrightarrow \exists p \text{ Parent}(g, p) \wedge \text{Parent}(p, c).$$

A sibling is another child of one's parents:

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

Not all logical sentences about a domain are axioms. Some are theorems—that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

$$\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x).$$

Is this an axiom or a theorem? In fact, it is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return true.

From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time, rather like a physicist having to re-derive the rules of calculus for every new problem.

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully. For example, there is no obvious definitive way to complete the sentence

$\forall x \text{ Person}(x) \Leftrightarrow \dots$

Fortunately, first-order logic allows us to make use of the Person predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$\forall x \text{ Person}(x) \Rightarrow \dots$

$\forall x \dots \Rightarrow \text{Person}(x).$

Axioms can also be “*just plain facts*,” such as *Male(Jim)* and *Spouse(Jim,Laura)*. Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms. Often, one finds that the expected answers are not forthcoming—for example, from *Spouse(Jim,Laura)* one expects (under the laws of many countries) to be able to infer $\neg \text{Spouse}(\text{George}, \text{Laura})$; but this does not follow from the axioms given earlier—even after we add $\text{Jim} = \text{George}$ as suggested.

4.3.3 Numbers, sets, and lists

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or non-negative integers. We need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, S (successor). The Peano axioms define natural numbers and addition.⁹ Natural numbers are defined recursively:

$\text{NatNum}(0).$

$\forall n \text{ NatNum}(n) \Rightarrow \text{NatNum}(S(n))$

That is, 0 is a natural number, and for every object n, if n is a natural number, then S(n) is a natural number. So the natural numbers are 0, S(0), S(S(0)), and so on. We also need axioms to constrain the successor function:

$\forall n 0 \neq S(n).$

$\forall m, n \ m = n \Rightarrow S(m) = S(n).$

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

$\forall m \text{ NatNum}(m) \Rightarrow +(0, m) = m.$

$\forall m, n \text{ NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow +(S(m), n) = S(+(m, n)).$

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol “+” in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using infix notation. (The notation we have used for first-order

logic is called prefix.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n+1$, so the second axiom becomes

$$\forall m, n \text{ NatNum}(m) \wedge \text{NatNum}(n) \Rightarrow (m+1)+n = (m+n)+1.$$

This axiom reduces addition to repeated application of the successor function. The use of infix notation is an example of syntactic sugar, that is, an extension to or abbreviation of the standard syntax that does not change the semantics. Any sentence that uses sugar can be “desugared” to produce an equivalent sentence in ordinary first-order logic.

Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms. The domain of sets is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to define number theory in terms of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets by adding an element to a set or taking the union or intersection of two sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets.

We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, Set , which is true of sets. The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2). The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining element x to set s). One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:

$$\forall s \text{ Set}(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \text{ Set}(s_2) \wedge s = \{x|s_2\}).$$

2. The empty set has no elements adjoined into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:

$$\neg \exists x, s \{x|s\} = \{\}.$$

3. Adjoining an element already in the set has no effect:

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

4. The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined

with some element y , where either y is the same as x or x is a member of s_2 :

$$\forall x, s \ x \in s \Leftrightarrow \exists y, s_2 \ (s = \{y | s_2\} \wedge (x = y \vee x \in s_2)).$$

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

$$\forall s_1, s_2 \ s_1 \subseteq s_2 \Leftrightarrow (\forall x x \in s_1 \Rightarrow x \in s_2).$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s_1, s_2 \ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$$

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

$$\forall x, s_1, s_2 \ x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2).$$

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

$$\forall x, s_1, s_2 \ x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2).$$

LIST

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list. We can use the vocabulary of Lisp for lists: Nil is the constant list with no elements; Cons, Append, First, and Rest are functions; and Find is the predicate that does for lists what Member does for sets. List? is a predicate that is true only of lists. As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is []. The term $Cons(x, y)$, where y is a nonempty list, is written $[x|y]$. The term $Cons(x, Nil)$ (i.e., the list containing the element x) is written as $[x]$. A list of several elements, such as $[A, B, C]$, corresponds to the nested term $Cons(A, Cons(B, Cons(C, Nil)))$.

4.3.4 The wumpus world

The first order axioms in this section are much more concise, capturing in a natural way exactly what we want to say. Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be $Percept([Stench, Breeze, Glitter, None, None], 5)$.

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

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

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

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

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:

$\forall s,g,m,c \text{ Percept}([s,Breeze,g,m,c],t) \Rightarrow Breeze(t)$,

$\forall s,b,m,c \text{ Percept}([s,b,Glitter,m,c],t) \Rightarrow Glitter(t)$,

and so on. These rules exhibit a trivial form of the reasoning process called perception. Notice the quantification over time t . In propositional logic, we would need copies of each sentence for each time step. Simple “reflex” behavior can also be implemented by quantified implication sentences. For example, we have

$\forall t \text{ Glitter}(t) \Rightarrow \text{BestAction}(Grab,t)$.

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion $\text{BestAction}(Grab,5)$ —that is, Grab is the right thing to do. We have represented the agent’s inputs and outputs; now it is time to represent the environment itself. Let us begin with objects. Obvious candidates are squares, pits, and the wumpus. We could name each square—Square1,2 and soon—but then the fact that Square1,2 and Square1,3 are adjacent would have to be an “extra” fact, and we would need one such fact for each pair of squares. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term $[1,2]$. Adjacency of any two squares can be defined as

$\forall x,y,a,b \text{ Adjacent}([x,y],[a,b]) \Leftrightarrow$

$(x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1))$.

We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits.¹⁰ It is simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant Wumpus is just as good as a unary predicate (and perhaps more dignified from the wumpus’s viewpoint). The agent’s location changes over time, so we write $At(\text{Agent},s,t)$ to mean that the agent is at square s at time t . We can fix the wumpus’s location with $\forall t At(\text{Wumpus},[2,2],t)$. We can then say that objects can only be at one location at a time:

$\forall x,s1,s2,t \text{ At}(x,s1,t) \wedge \text{At}(x,s2,t) \Rightarrow s1 = s2$.

Given its current location, the agent can infer properties of the square from properties of its current percept. For example, if the agent is at a square and perceives a breeze, then that

square is breezy:

$$\forall s, t \text{ At}(\text{Agent}, s, t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(s) .$$

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). Whereas propositional logic necessitates a separate axiom for each square and would need a different set of axioms for each geographical layout of the world, first-order logic just needs one axiom:

$$\forall s \text{ Breezy}(s) \Leftrightarrow \exists r \text{ Adjacent}(r, s) \wedge \text{Pit}(r) .$$

Similarly, in first-order logic we can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step. For example, the axiom for the arrow becomes

$$\forall t \text{ HaveArrow}(t+1) \Leftrightarrow (\text{HaveArrow}(t) \wedge \neg \text{Action}(\text{Shoot}, t)) .$$

From these two example sentences, we can see that the first-order logic formulation is no less concise than the original English-language.

4.4 Knowledge Engineering In First-Order Logic

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge-base construction—a process called knowledge engineering. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar, so that we can concentrate on the representational issues involved. The approach we take is suitable for developing special-purpose knowledge bases whose domain is carefully circumscribed and whose range of queries is known in advance. General-purpose knowledge bases, which cover a broad range of human knowledge and are intended to support tasks such as natural language understanding.

4.4.1 The knowledge-engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such

projects include the following steps:

1. Identify the task. The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers.

2. Assemble the relevant knowledge. The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called knowledge acquisition. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works. For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.

3. Decide on a vocabulary of predicates, functions, and constants. That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering style. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the ontology of the domain. The word ontology means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

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

5. Encode a description of the specific problem instance. If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a “disembodied” knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. Pose queries to the inference procedure and get answers. This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.

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

$$\forall s \text{ Smelly}(s) \Rightarrow \text{Adjacent}(\text{Home}(\text{Wumpus}),s) ,$$

instead of the biconditional, then the agent will never be able to prove the absence of wumpuses. Incorrect axioms can be identified because they are false statements about the world. For example, the sentence

$$\forall x \text{ NumOfLegs}(x,4) \Rightarrow \text{Mammal}(x)$$

is false for reptiles, amphibians, and, more importantly, tables. The falsehood of this sentence can be determined independently of the rest of the knowledge base. In contrast, a typical error in a program looks like this:

$$\text{offset} = \text{position} + 1.$$

It is impossible to tell whether this statement is correct without looking at the rest of the program to see whether, for example, offset is used to refer to the current position, or to one beyond the current position, or whether the value of position is change by another statement

and so offset should also be changed again.

4.4.2 The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 4.6. We follow the seven-step process for knowledge engineering.

1) Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 4.6 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

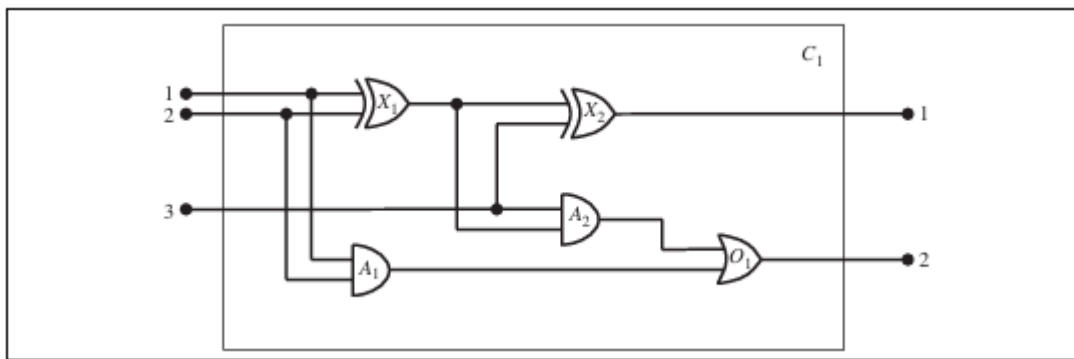


Figure 4.6: A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

2) Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All

gates have one output terminal. Circuits, like gates, have input and output terminals.

3) Decide on a vocabulary

We now know that, we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented as an object named by a constant, about which we assert that it is a gate with, say, $Gate(X1)$. The behavior of each gate is determined by its type: one of the constants $AND, OR, XOR, or NOT$. Because a gate has exactly one type, a function is appropriate: $Type(X1)=XOR$. Circuits, like gates, are identified by a predicate: $Circuit(C1)$. Next we consider terminals, which are identified by the predicate $Terminal(x)$. A gate or circuit can have one or more input terminals and one or more output terminals. We use the function $In(1,X1)$ to denote the first input terminal for gate $X1$. A similar function Out is used for output terminals. The function $Arity(c,i,j)$ says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, $Connected$, which takes two terminals as arguments, as in $Connected(Out(1,X1),In(1,X2))$. Finally, we need to know whether a signal is on or off.

4) Encode general knowledge of the domain

One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

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

$$\forall t1,t2 \text{ Terminal}(t1) \wedge \text{Terminal}(t2) \wedge \text{Connected}(t1,t2) \Rightarrow \text{Signal}(t1)=\text{Signal}(t2)$$

2. The signal at every terminal is either 1 or 0:

$$\forall t \text{ Terminal}(t) \Rightarrow \text{Signal}(t)=1 \vee \text{Signal}(t)=0 .$$

3. $Connected$ is commutative:

$$\forall t1,t2 \text{ Connected}(t1,t2) \Leftrightarrow \text{Connected}(t2,t1) .$$

4. There are four types of gates:

$$\forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \Rightarrow k = AND \vee k = OR \vee k = XOR \vee k = NOT .$$

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

$$\begin{aligned} \forall g \text{ Gate}(g) \wedge \text{Type}(g)=AND \Rightarrow \\ \text{Signal}(\text{Out}(1,g))=0 \Leftrightarrow \exists n \text{ Signal}(\text{In}(n,g))=0 . \end{aligned}$$

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

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

7. AnXORgate' s output is 1 if and only if its inputs are different:

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

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

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

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

$$\forall g \text{ Gate}(g) \wedge \text{Type}(g) \neq \text{NOT} \Rightarrow \text{Arity}(g, 1, 1) . \\ \forall g \text{ Gate}(g) \wedge k = \text{Type}(g) \wedge (k = \text{AND} \vee k = \text{OR} \vee k = \text{XOR}) \Rightarrow \\ \text{Arity}(g, 2, 1)$$

10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:

$$\forall c, i, j \text{ Circuit}(c) \wedge \text{Arity}(c, i, j) \Rightarrow \\ \forall n (n \leq i \Rightarrow \text{Terminal}(\text{In}(c, n))) \wedge (n > i \Rightarrow \text{In}(c, n) = \text{Nothing}) \wedge \\ \forall n (n \leq j \Rightarrow \text{Terminal}(\text{Out}(c, n))) \wedge (n > j \Rightarrow \text{Out}(c, n) = \text{Nothing})$$

11. Gates, terminals, signals, gate types, and Nothing are all distinct.

$$\forall g, t \text{ Gate}(g) \wedge \text{Terminal}(t) \Rightarrow \\ g \neq t \neq 1 \neq 0 \neq \text{OR} \neq \text{AND} \neq \text{XOR} \neq \text{NOT} \neq \text{Nothing} .$$

12. Gates are circuits.

$$\forall g \text{ Gate}(g) \Rightarrow \text{Circuit}(g)$$

5) Encode the specific problem instance

The circuit shown in Figure 8.6 is encoded as circuit C1 with the following description. First, we categorize the circuit and its component gates:

$$\text{Circuit}(C1) \wedge \text{Arity}(C1, 3, 2) \\ \text{Gate}(X1) \wedge \text{Type}(X1) = \text{XOR} \\ \text{Gate}(X2) \wedge \text{Type}(X2) = \text{XOR} \\ \text{Gate}(A1) \wedge \text{Type}(A1) = \text{AND} \\ \text{Gate}(A2) \wedge \text{Type}(A2) = \text{AND} \\ \text{Gate}(O1) \wedge \text{Type}(O1) = \text{OR} .$$

Then, we show the connections between them:

$Connected(Out(1, X_1), In(1, X_2))$ $Connected(In(1, C_1), In(1, X_1))$
 $Connected(Out(1, X_1), In(2, A_2))$ $Connected(In(1, C_1), In(1, A_1))$
 $Connected(Out(1, A_2), In(1, O_1))$ $Connected(In(2, C_1), In(2, X_1))$
 $Connected(Out(1, A_1), In(2, O_1))$ $Connected(In(2, C_1), In(2, A_1))$
 $Connected(Out(1, X_2), Out(1, C_1))$ $Connected(In(3, C_1), In(2, X_2))$
 $Connected(Out(1, O_1), Out(2, C_1))$ $Connected(In(3, C_1), In(1, A_2))$.

6) Pose queries to the inference procedure

What combinations of inputs would cause the first output of C1 (the sum bit) to be 0 and the second output of C1 (the carry bit) to be 1?

$\exists i1, i2, i3 \text{ Signal}(In(1, C1))=i1 \wedge \text{Signal}(In(2, C1))=i2 \wedge \text{Signal}(In(3, C1))=i3$
 $\wedge \text{Signal}(Out(1, C1))=0 \wedge \text{Signal}(Out(2, C1))=1$.

The answers are substitutions for the variables i1, i2, and i3 such that the resulting sentence is entailed by the knowledge base. ASKVARs will give us three such substitutions:

$\{i1/1, i2/1, i3/0\}$ {
 $i1/1, i2/0, i3/1\}$ {
 $i1/0, i2/1, i3/1\}$.

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

$\exists i1, i2, i3, o1, o2 \text{ Signal}(In(1, C1))=i1 \wedge \text{Signal}(In(2, C1))=i2$
 $\wedge \text{Signal}(In(3, C1))=i3 \wedge \text{Signal}(Out(1, C1))=o1 \wedge \text{Signal}(Out(2, C1))=o2$.

This final query will return a complete input-output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of circuit verification. We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out.

7) Debug the knowledge base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, hence forget to assert that 1=0. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

$\exists i1, i2, o \text{ Signal}(In(1, C1))=i1 \wedge \text{Signal}(In(2, C1))=i2 \wedge \text{Signal}(Out(1, X1))$,

which reveals that no outputs are known at X1 for the input cases 10 and 01. Then, we look at the axiom for XOR gates, as applied to X1:

$\text{Signal}(Out(1, X1))=1 \Leftrightarrow \text{Signal}(In(1, X1)) \neq \text{Signal}(In(2, X1))$.

If the inputs are known to be, say, 1 and 0, then this reduces to

$\text{Signal}(Out(1, X1))=1 \Leftrightarrow 1=0$.

Now the problem is apparent: the system is unable to infer that $Signal(Out(1,X1))=1$, so we need to tell it that $1=0$.

Inference in First Order Logic

4.5 Propositional Versus First Order Inference

The idea of unification, showing how it can be used to construct inference rules that work directly with first-order sentences. We then discuss three major families of first-order inference algorithms. We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that first-order inference can be done by converting the knowledge base to propositional logic and using propositional inference, which we already know how to do. The next section points out an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

4.5.1 Inference rules for quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x).$$

Then it seems quite permissible to infer any of the following sentences:

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$$

$$\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \Rightarrow \text{Evil}(\text{Father}(\text{John})).$$

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.¹ To write out the inference rule formally. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α . Then the rule is written

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

for any variable v and ground term g . For example, the three sentences given earlier are obtained with the substitutions $\{x/\text{John}\}$, $\{x/\text{Richard}\}$, and $\{x/\text{Father}(\text{John})\}$.

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,

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

For example, from the sentence

$$\exists x \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$$

we can infer the sentence

$$\text{Crown}(C1) \wedge \text{OnHead}(C1, \text{John})$$

As long as $C1$ does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. Of course, that name must not already belong to another object. Mathematics provides a nice example: suppose we discover that there is a number that is a little bigger than 2.71828 and that satisfies the equation $d(xy)/dy = xy$ for x . We can give this number a name, such as e , but it would be a mistake to give it the name of an existing object, such as π . In logic, the new name is called a **Skolem constant**. Existential Instantiation is a special case of a more general process called **skolemization**.

4.5.2 Reduction to propositional inference

Once we have rules for inferring nonquantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of all possible instantiations. For example, suppose our knowledge base contains just the sentences

$$\forall x \text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$$

$$\text{King}(\text{John})$$

$$\text{Greedy}(\text{John})$$

$$\text{Brother}(\text{Richard}, \text{John}) .$$

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. We obtain

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard}) ,$$

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.

4.6 Unification

The preceding section described the understanding of first-order inference that existed up to the early 1960s. The sharp-eyed reader (and certainly the computational logicians of the early 1960s) will have noticed that the **propositionalization** approach is rather inefficient. For example, given the query *Evil(x)* and the knowledge base, it seems per verse to generate sentences such as *King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)*.

Indeed, the inference of *Evil(John)* from the sentences

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$$

King(John)

Greedy(John)

Seems completely obvious to a human being. We now show how to make it completely obvious to a computer.

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:

$$\text{UNIFY}(p,q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query *AskVars(Knows(John,x))*: whom does John know? Answers to this query 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:

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

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 $\text{Knows}(x, \text{Elizabeth})$ to $x17$ (a new variable name) without changing its meaning. Now the unification will work:

$$\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x17, \text{Elizabeth})) = \{x/\text{Elizabeth}, x17/\text{John}\}.$$

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z))$ could return $\{y/\text{John}, x/z\}$ or $\{y/\text{John}, x/\text{John}, z/\text{John}\}$. The first unifier gives $\text{Knows}(\text{John}, z)$ as the result of unification, whereas the second gives $\text{Knows}(\text{John}, \text{John})$. The second result could be obtained from the first by an additional substitution $\{z/\text{John}\}$; we say that the first unifier is more general than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single most general *unifier* (or *MGU*) that is unique up to renaming and substitution of variables. (For example, $\{x/\text{John}\}$ and $\{y/\text{John}\}$ are considered equivalent, as are $\{x/\text{John}, y/\text{John}\}$ and $\{x/\text{John}, y/x\}$.) In this case it is $\{y/\text{John}, x/z\}$.

An algorithm for computing most general unifiers is shown in Figure 4.7. 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. For example, $S(x)$ can't unify with $S(S(x))$. This so called occur check makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

```

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

if  $\theta = \text{failure}$  then return failure
else if  $x = y$  then return  $\theta$ 
else if VARIABLE?( $x$ ) then return UNIFY-VAR( $x, y, \theta$ )
else if VARIABLE?( $y$ ) then return UNIFY-VAR( $y, x, \theta$ )
else if COMPOUND?( $x$ ) and COMPOUND?( $y$ ) then
    return UNIFY( $x$ .ARGS,  $y$ .ARGS, UNIFY( $x$ .OP,  $y$ .OP,  $\theta$ ))
else if LIST?( $x$ ) and LIST?( $y$ ) then
    return UNIFY( $x$ .REST,  $y$ .REST, UNIFY( $x$ .FIRST,  $y$ .FIRST,  $\theta$ ))
else return failure



---


function UNIFY-VAR( $var, x, \theta$ ) returns a substitution

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

```

Figure 4.7: The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A,B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A,B) .

4.7 Forward Chaining

The idea is simple: start with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made. Here, we explain how the algorithm is applied to first-order definite clauses. Definite clauses such as $\text{Situation} \Rightarrow \text{Response}$ are especially useful for systems that make inferences in response to newly arrived information. Many systems can be defined this way, and forward chaining can be implemented very efficiently.

4.7.1 First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses (page 256): they are disjunctions of literals of which exactly one is positive. A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$\text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$.

King(John) .

Greedy(y)

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified. (Typically, we omit universal quantifiers when writing definite clauses.) Not every knowledge base can be converted into a set of definite clauses because of the single-positive-literal restriction, but many can. Consider the following problem: The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

We will prove that west is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

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

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

“Nono ...has some missiles.” The sentence $\exists x Owns(Nono,x) \wedge Missile(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M1:

$Owns(Nono,M1)$

$Missile(M1)$

“All of its missiles were sold to it by Colonel West” :

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

We will also need to know that missiles are weapons:

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

and we must know that an enemy of America counts as “hostile” :

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

“West, who is American ...” :

$American(West) .$

“The country Nono, an enemy of America ...” :

$Enemy(Nono,America) .$

This knowledge base contains no function symbols and is therefore an instance of the class of Datalog knowledge bases. Datalog is a language that is restricted to first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases.

4.7.2 A simple forward-chaining algorithm

The first forward-chaining algorithm we consider is a simple one, shown in Figure 4.8. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. Notice that a fact is not “new” if it is just a renaming of a known fact. One sentence is a renaming of another if they are identical except for the names of the variables. For example, *Likes(x,IceCream)* and *Likes(y,IceCream)* are renamings of each other because they differ only in the choice of *x* or *y*; their meanings are identical: everyone likes ice cream. We use our crime problem to illustrate how FOL-FC-ASK works. Two iterations are required:

- On the first iteration, rule (9.3) has unsatisfied premises.
Rule (9.6) is satisfied with $\{x/M_1\}$, and *Sells(West, M₁, Nono)* is added.
Rule (9.7) is satisfied with $\{x/M_1\}$, and *Weapon(M₁)* is added.
Rule (9.8) is satisfied with $\{x/Nono\}$, and *Hostile(Nono)* is added.
- On the second iteration, rule (9.3) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and *Criminal(West)* is added.

Figure 4.9 shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a fixed point of the inference process. Fixed points reached by forward chaining with first-order definite clauses are similar to those for propositional forward chaining, the principal difference is that a first order fixed point can include universally quantified atomic sentences. FOL-FC-ASK is easy to analyze.

First, it is sound, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is complete for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses. For Datalog knowledge bases, which contain no function symbols, the proof of completeness is fairly easy. We begin by counting the number of possible facts that can be added, which determines the maximum number of iterations. Let *k* be the maximum arity (number of arguments) of any predicate, *p* be the number of predicates, and *n* be the number of constant symbols. Clearly, there can be no more than $p \cdot n^k$ distinct ground facts, so after this many iterations the algorithm must have reached a fixed point.

Then we can make an argument very similar to the proof of completeness for propositional forward chaining. The details of how to make the transition from propositional to first-order

completeness, For general definite clauses with function symbols, FOL-FC-ASK can generate infinitely many new facts, so we need to be more careful. For the case in which an answer to the query sentence q is entailed, we must appeal to Herbrand's theorem to establish that the algorithm will find a proof. If the query has no answer, the algorithm could fail to terminate in some cases. For example, if the knowledge base includes the Peano axioms

$NatNum(0)$

$\forall n \text{ } NatNum(n) \Rightarrow NatNum(S(n))$,

then forward chaining adds $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$, and so on. This problem is unavoidable in general. As with general first-order logic, entailment with definite clauses is semi decidable.

```

function FOL-FC-ASK( $KB, \alpha$ ) returns a substitution or false
  inputs:  $KB$ , the knowledge base, a set of first-order definite clauses
            $\alpha$ , the query, an atomic sentence
  local variables:  $new$ , the new sentences inferred on each iteration

  repeat until  $new$  is empty
     $new \leftarrow \{ \}$ 
    for each  $rule$  in  $KB$  do
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(rule)$ 
      for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$ 
        for some  $p'_1, \dots, p'_n$  in  $KB$ 
           $q' \leftarrow \text{SUBST}(\theta, q)$ 
          if  $q'$  does not unify with some sentence already in  $KB$  or  $new$  then
            add  $q'$  to  $new$ 
             $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
            if  $\phi$  is not fail then return  $\phi$ 
    add  $new$  to  $KB$ 
  return false

```

Figure 4.8: A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

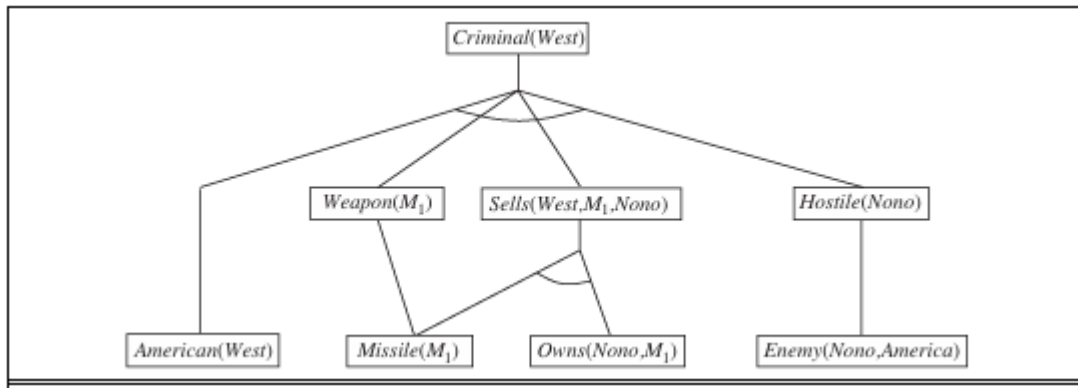


Figure 4.9: The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

MODULE - 5

Inference in First Order Logic:

5.1 Backward Chaining

Logical inference algorithms use the backward chaining approach. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. We describe the basic algorithm, and then we describe how it is used in logic programming, which is the most widely used form of automated reasoning.

5.1.1 A backward-chaining algorithm

Figure 5.1 shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(KB, goal) will be proved if the knowledge base contains a clause of the form $\text{lhs} \Rightarrow \text{goal}$, where lhs (left-hand side) is a list of conjuncts. An atomic fact like `American(West)` is considered as a clause whose lhs is the empty list. Now a query that contains variables might be proved in multiple ways. For example, the query `Person(x)` could be proved with the substitution $\{x/\text{John}\}$ as well as with $\{x/\text{Richard}\}$. So we implement FOL-BC-ASK as a generator—a function that returns multiple times, each time giving one possible result.

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as we go. Figure 5.2 is the proof tree for deriving `Criminal(West)`. Backward chaining, as we have written it, is clearly a depth-first search algorithm.

This means that its space requirements are linear in the size of the proof (neglecting, for now, the space required to accumulate the solutions). It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. We will discuss these problems and some potential solutions, but first we show how backward chaining is used in logic programming systems.

```

function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })



---


generator FOL-BC-OR(KB, goal,  $\theta$ ) yields a substitution
  for each rule (lhs  $\Rightarrow$  rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs)  $\leftarrow$  STANDARDIZE-VARIABLES((lhs, rhs))
    for each  $\theta'$  in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal,  $\theta$ )) do
      yield  $\theta'$ 



---


generator FOL-BC-AND(KB, goals,  $\theta$ ) yields a substitution
  if  $\theta$  = failure then return
  else if LENGTH(goals) = 0 then yield  $\theta$ 
  else do
    first, rest  $\leftarrow$  FIRST(goals), REST(goals)
    for each  $\theta'$  in FOL-BC-OR(KB, SUBST( $\theta$ , first),  $\theta$ ) do
      for each  $\theta''$  in FOL-BC-AND(KB, rest,  $\theta'$ ) do
        yield  $\theta''$ 

```

Figure 5.1: A simple backward-chaining algorithm for first-order knowledge bases.

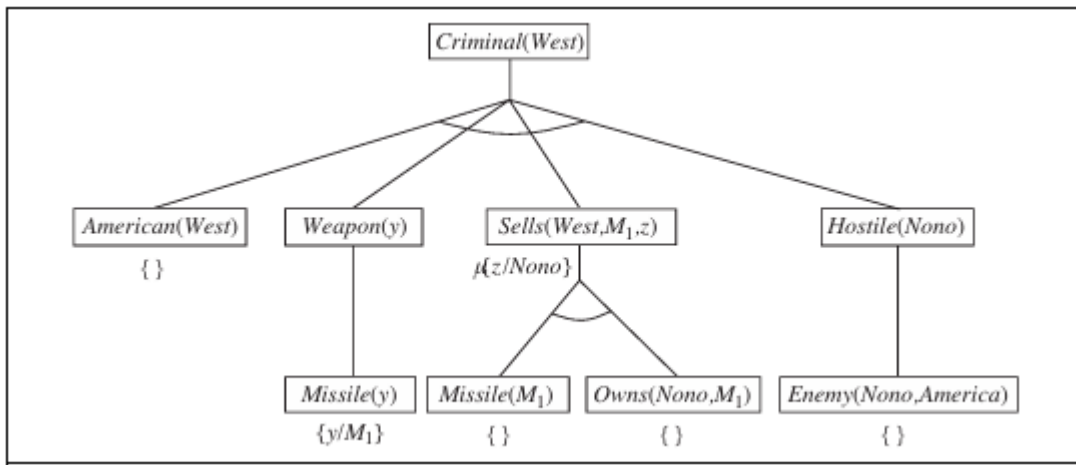


Figure 5.2: Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally Hostile(*z*), *z* is already bound to Nonono.

5.1.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative, that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is

summed up in Robert Kowalski's equation,

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

Prolog is the most widely used logic programming language. It is used primarily as a rapid prototyping language and for symbol-manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains. Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants—the opposite of our convention for logic. Commas separate conjuncts in a clause and the clause is written “backwards” from what we are used to; instead of $A \wedge B \Rightarrow C$ in Prolog we have $C:-A,B$. Here is a typical example:

criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).

The notation $[E|L]$ denotes a list whose first element is E and whose rest is L. Here is a Prolog program for $\text{append}(X,Y,Z)$, which succeeds if list Z is the result of appending

lists X and Y:

append([],Y,Y).

append([A|X],Y,[A|Z]) :- append(X,Y,Z).

In English, we can read these clauses as (1) appending an empty list with a list Y produces the same list Y and (2) $[A|Z]$ is the result of appending $[A|X]$ onto Y, provided that Z is the result of appending X onto Y. In most high-level languages we can write a similar recursive function that describes how to append two lists. The Prolog definition is actually much more powerful, however, because it describes a relation that holds among three arguments, rather than a function computed from two arguments. For example, we can ask the query $\text{append}(X,Y,[1,2])$: what two lists can be appended to give $[1,2]$? We get back the solutions

$X=[]$

$Y=[1,2];$

$X=[1] \ Y=[2];$

$X=[1,2] \ Y=[]$

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. Some aspects

of Prolog fall outside standard logical inference:

- Prolog uses the database semantics of rather than first-order semantics, and this is apparent in its treatment of equality and negation.
- There is a set of built-in functions for arithmetic. Literals using these function symbols are “proved” by executing code rather than doing further inference. For example, the goal “X is 4+3” succeeds with X bound to 7. On the other hand, the goal “is X+Y” fails, because the built-in functions do not do arbitrary equation solving.
- There are built-in predicates that have side effects when executed. These include input output predicates and the assert/retract predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce confusing results for example, if facts are asserted in a branch of the proof tree that eventually fails.
- The occur check is omitted from Prolog’s unification algorithm. This means that some unsound inferences can be made; these are almost never a problem in practice.
- Prolog uses depth-first backward-chaining search with no checks for infinite recursion. This makes it very fast when given the right set of axioms, but incomplete when given the wrong ones.

5.1.3 Efficient implementation of logic programs

The execution of a Prolog program can happen in two modes: interpreted and compiled. Interpretation essentially amounts to running the FOL-BC-ASK algorithm from Figure 5.3, with the program as the knowledge base. We say “essentially” because Prolog interpreters contain a variety of improvements designed to maximize speed. Here we consider only two **First, our implementation** had to explicitly manage the iteration over possible results generated by each of the subfunctions. Prolog interpreters have a global data structure a stack of choice points, to keep track of the multiple possibilities that we considered in FOL-BC-OR. This global stack is more efficient, and it makes debugging easier, because the debugger can move up and down the stack.

Second, our simple implementation of FOL-BC-ASK spends a good deal of time generating substitutions. Instead of explicitly constructing substitutions, Prolog has logic variables that remember their current binding. At any point in time, every variable in the

program either is unbound or is bound to some value. Together, these variables and values implicitly define the substitution for the current branch of the proof. Extending the path can only add new variable bindings, because an attempt to add a different binding for an already bound variable results in a failure of unification. When a path in the search fails, Prolog will back up to a previous choice point, and then it might have to unbind some variables. This is done by keeping track of all the variables that have been bound in a stack called the trail. As each new variable is bound by UNIFY-VAR, the variable is pushed onto the trail. When a goal fails and it is time to back up to a previous choice point, each of the variables is unbound as it is removed from the trail.

Even the most efficient Prolog interpreters require several thousand machine instructions per inference step because of the cost of index lookup, unification, and building the recursive call stack. In effect, the interpreter always behaves as if it has never seen the program before; for example, it has to find clauses that match the goal. A compiled Prolog program, on the other hand, is an inference procedure for a specific set of clauses, so it knows what clauses match the goal. Prolog basically generates a miniature theorem prove for each different predicate, thereby eliminating much of the overhead of interpretation. It is also possible to open-code the unification routine for each different call, thereby avoiding explicit analysis of term structure. (For details of open-coded unification, see Warren et al. (1977).)

For example, the definition of the Append predicate can be compiled into the code shown in Figure 5.3. Several points are worth mentioning:

- Rather than having to search the knowledge base for Append clauses, the clauses become a procedure and the inferences are carried out simply by calling the procedure.
- As described earlier, the current variable bindings are kept on a trail. The first step of the procedure saves the current state of the trail, so that it can be restored by RESET-TRAIL if the first clause fails. This will undo any bindings generated by the first call to UNIFY.
- The trickiest part is the use of continuations to implement choice points. You can think of a continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds. It would not do just to return from a procedure like APPEND when the goal succeeds,

because it could succeed in several ways, and each of them has to be explored. The continuation argument solves this problem because it can be called each time the goal succeeds. In the APPEND code, if the first argument is empty and the second argument unifies with the third, then the APPEND predicate has succeeded. We then CALL the continuation, with the appropriate bindings on the trail, to do whatever should be done next. For example, if the call to APPEND were at the top level, the continuation would print the bindings of the variables.

Parallelization can also provide substantial speedup. There are two principal sources of parallelism. The first, called OR-parallelism, comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel. The second, called AND-parallelism, comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables. Each conjunctive branch must communicate with the other branches to ensure a global solution.

```
procedure APPEND(ax, y, az, continuation)
    trail ← GLOBAL-TRAIL-POINTER()
    if ax = [] and UNIFY(y, az) then CALL(continuation)
    RESET-TRAIL(trail)
    a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
    if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)
```

Figure 5.3: Pseudo code representing the result of compiling the Append predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables used so far. The procedure CALL(*continuation*) continues execution with the specified continuation.

5.1.4 Redundant inference and infinite loops

We now turn to the Achilles heel of Prolog: the mismatch between depth-first search and search trees that include repeated states and infinite paths. Consider the following logic program that decides if a path exists between two points on a directed graph:

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

A simple three-node graph, described by the facts `link(a,b)` and `link(b,c)`, is shown in Figure 5.4(a). With this program, the query `path(a,c)` generates the proof tree shown in Figure 5.5(a). On the other hand, if we put the two clauses in the order

```
path(X,Z) :- path(X,Y), link(Y,Z).
```

```
path(X,Z) :- link(X,Z).
```

Then Prolog follows the infinite path shown in Figure 5.4(b). Prolog is therefore incomplete as a theorem prover for definite clauses—even for Datalog programs, as this example shows—because, for some knowledge bases, it fails to prove sentences that are entailed.

Notice that forward chaining does not suffer from this problem: once `path(a,b)`, `path(b,c)`, and `path(a,c)` are inferred, forward chaining halts. Depth-first backward chaining also has problems with redundant computations. For example, when finding a path from A1 to J4 in Figure 5.5(b), Prolog performs 877 inferences, most of which involve finding all possible paths to nodes from which the goal is unreachable. The total amount of inference can be exponential in the number of ground facts that are generated. If we apply forward chaining instead, at most n^2 `path(X,Y)` facts can be generated linking n nodes. For the problem in Figure 5.5(b), only 62 inferences are needed.

Forward chaining on graph search problems is an example of dynamic programming, in which the solutions to subproblems are constructed incrementally from those of smaller subproblems and are cached to avoid recomputation. We can obtain a similar effect in a backward chaining system using memoization—that is, caching solutions to subgoals as they are found and then reusing those solutions when the subgoal recurs, rather than repeating the previous computation. This is the approach taken by tabled logic programming systems, which use efficient storage and retrieval mechanisms to perform memoization. Tabled logic programming combines the goal-directedness of backward chaining with the dynamic programming efficiency of forward chaining. It is also complete for Datalog knowledge bases, which means that the programmer need worry less about infinite loops.

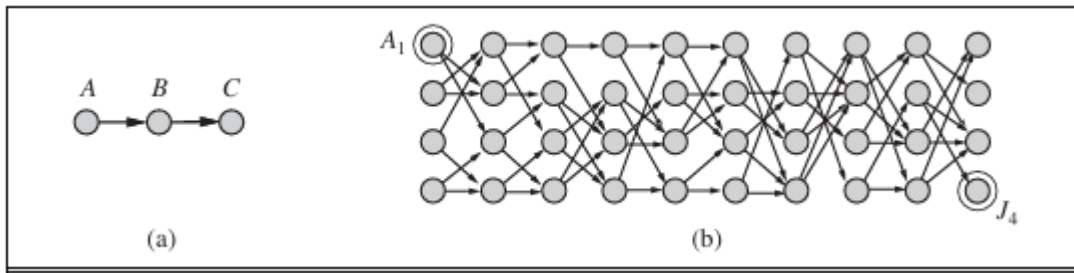


Figure 5.4: (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

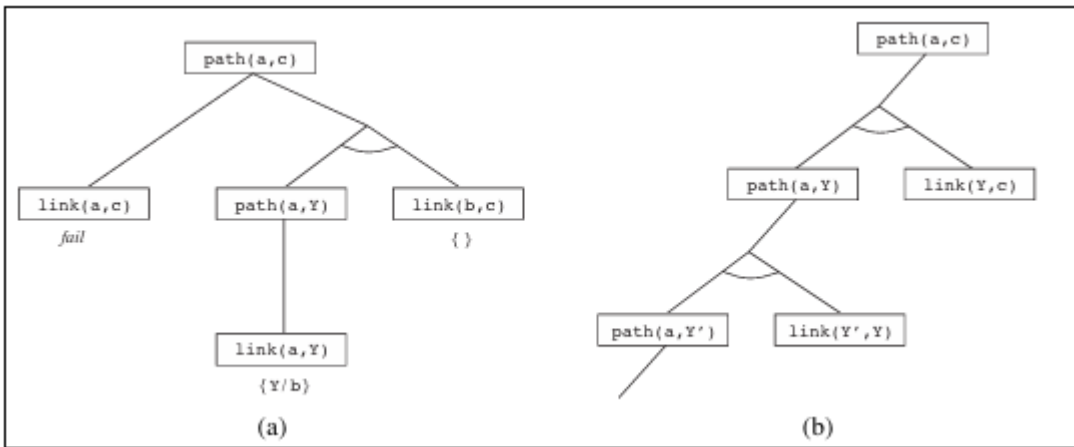


Figure 5.5: (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the “wrong” order.

5.1.5 Database semantics of Prolog

Prolog uses database semantics. The unique names assumption says that every Prolog constant and every ground term refers to a distinct object, and the closed world assumption says that the only sentences that are true are those that are entailed by the knowledge base. There is no way to assert that a sentence is false in Prolog. This makes Prolog less expressive than first-order logic, but it is part of what makes Prolog more efficient and more concise. Consider the following Prolog assertions about some course offerings:

Course(CS,101), Course(CS,102), Course(CS,106), Course(EE,101)

Under the unique names assumption, CS and EE are different (as are 101, 102, and 106), so this means that there are four distinct courses. Under the closed-world assumption there are no other courses, so there are exactly four courses. But if these were assertions in FOL rather than in Prolog, then all we could say is that there are somewhere between one and infinity courses. That's because the assertions (in FOL) do not deny the possibility that other

unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other. If we wanted to translate Equation (9.11) into FOL, we would get this:

$$\begin{aligned} \text{Course}(d,n) \Leftrightarrow & (d=CS \wedge n=101) \vee (d=CS \wedge n=102) \\ & \vee (d=CS \wedge n=106) \vee (d=EE \wedge n=101). \end{aligned}$$

This is called the completion of Equation. It expresses in FOL the idea that there are at most four courses. To express in FOL the idea that there are at least four courses, we need to write the completion of the equality predicate:

$$\begin{aligned} x=y \Leftrightarrow & (x=CS \wedge y=CS) \vee (x=EE \wedge y=EE) \vee (x=101 \wedge y=101) \\ & \vee (x=102 \wedge y=102) \vee (x=106 \wedge y=106). \end{aligned}$$

The completion is useful for understanding database semantics, but for practical purposes, if your problem can be described with database semantics, it is more efficient to reason with Prolog or some other database semantics system, rather than translating into FOL and reasoning with a full FOL theorem prover.

5.1.6 Constraint logic programming

Backtracking enumerates the domains of the variables, it works only for finite domain CSPs. In Prolog terms, there must be a finite number of solutions for any goal with unbound variables. Infinite-domain CSPs—for example, with integer or real-valued variables—require quite different algorithms, such as bounds propagation or linear programming.

Consider the following example. We define $\text{triangle}(X,Y,Z)$ as a predicate that holds if the three arguments are numbers that satisfy the triangle inequality:

$$\text{triangle}(X,Y,Z) : X>0, Y>0, Z>0, X+Y \geq Z, Y+Z \geq X, X+Z \geq Y.$$

If we ask Prolog the query $\text{triangle}(3,4,5)$, it succeeds. On the other hand, if we ask $\text{triangle}(3,4,Z)$, no solution will be found, because the subgoal $Z \geq 0$ cannot be handled by Prolog; we can't compare an unbound value to 0. Constraint logic programming (CLP) allows variables to be constrained rather than bound. A CLP solution is the most specific set of constraints on the query variables that can be derived from the knowledge base. For example, the solution to the $\text{triangle}(3,4,Z)$ query is the constraint $7 \geq Z \geq 1$. Standard logic programs are just a special case of CLP in which the solution constraints must be equality constraints—that is, bindings.

5.2 RESOLUTION

The last of our three families of logical systems is based on resolution. We saw on page 250 that propositional resolution using refutation is a complete inference procedure for propositional logic.

5.2.1 Conjunctive normal form for first-order logic

As in the propositional case, first-order resolution requires that sentences be in conjunctive normal form (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals.⁶ Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x \text{ American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$$

becomes, in CNF,

$$\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x) .$$

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.

The procedure for conversion to CNF is similar to the propositional case, which we saw on page 253. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence “Everyone who loves all animals is loved by someone,” or

$$\forall x [\forall y \text{ Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow [\exists y \text{ Loves}(y, x)] .$$

The steps are as follows:

- Eliminate implications:

$$\forall x [\neg \forall y \neg \text{Animal}(y) \vee \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] .$$

- Move $\neg$ -inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\neg \forall x p \text{ becomes } \exists x \neg p$$

$$\neg \exists x p \text{ becomes } \forall x \neg p .$$

Our sentence goes through the following transformations:

$$\forall x [\exists y \neg (\neg \text{Animal}(y) \vee \text{Loves}(x, y))] \vee [\exists y \text{ Loves}(y, x)] .$$

$$\forall x [\exists y \neg \neg \text{Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] .$$

$$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)] .$$

- Standardize variables: For sentences like $(\exists x P(x)) \vee (\exists x Q(x))$ which use the same

variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x [\exists y \text{ Animal}(y) \wedge \neg \text{Loves}(x,y)] \vee [\exists z \text{ Loves}(z,x)] .$$

- **Skolemize:** Skolemization is the process of removing existential quantifiers by elimination. translate $\exists x P(x)$ into $P(A)$, where A is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern $\exists v \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get

$$\forall x [\text{Animal}(A) \wedge \neg \text{Loves}(x,A)] \vee \text{Loves}(B,x) ,$$

which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B . In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x and z :

$$\forall x [\text{Animal}(F(x)) \wedge \neg \text{Loves}(x,F(x))] \vee \text{Loves}(G(z),x) .$$

Here F and G are Skolem functions. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

- **Drop universal quantifiers:** At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

$$[\text{Animal}(F(x)) \wedge \neg \text{Loves}(x,F(x))] \vee \text{Loves}(G(z),x) .$$

- **Distribute $\vee$ over $\wedge$:**

$$[\text{Animal}(F(x)) \vee \text{Loves}(G(z),x)] \wedge [\neg \text{Loves}(x,F(x)) \vee \text{Loves}(G(z),x)] .$$

This step may also require flattening out nested conjunctions and disjunctions.

5.2.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule given on page 253. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one unifies with the negation of the other. Thus, we have

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$[\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)] \quad \text{and} \quad [\neg \text{Loves}(u, v) \vee \neg \text{Kills}(u, v)]$$

by eliminating the complementary literals $\text{Loves}(G(x), x)$ and $\neg \text{Loves}(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

$$[\text{Animal}(F(x)) \vee \neg \text{Kills}(G(x), x)] .$$

This rule is called the binary resolution rule because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend factoring—the removal of redundant literals—to the first-order case. Propositional factoring reduces two literals to one if they are identical; first-order factoring reduces two literals to one if they are unifiable. The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

5.2.3 Example proofs

Resolution proves that $\text{KB} \models \alpha$ by proving $\text{KB} \wedge \neg \alpha$ unsatisfiable, that is, by deriving the empty clause. The algorithmic approach is identical to the propositional case. Instead, we give two example proofs. The first is the crime example, The sentences in CNF are

$$\begin{array}{ll} \neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x) & \\ \neg \text{Missile}(x) \vee \neg \text{Owns}(\text{Nono}, x) \vee \text{Sells}(\text{West}, x, \text{Nono}) & \\ \neg \text{Enemy}(x, \text{America}) \vee \text{Hostile}(x) & \\ \neg \text{Missile}(x) \vee \text{Weapon}(x) & \\ \text{Owns}(\text{Nono}, M_1) & \text{Missile}(M_1) \\ \text{American}(\text{West}) & \text{Enemy}(\text{Nono}, \text{America}) . \end{array}$$

We also include the negated goal $\neg \text{Criminal}(\text{West})$. The resolution proof is shown in Figure 5.6. 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.

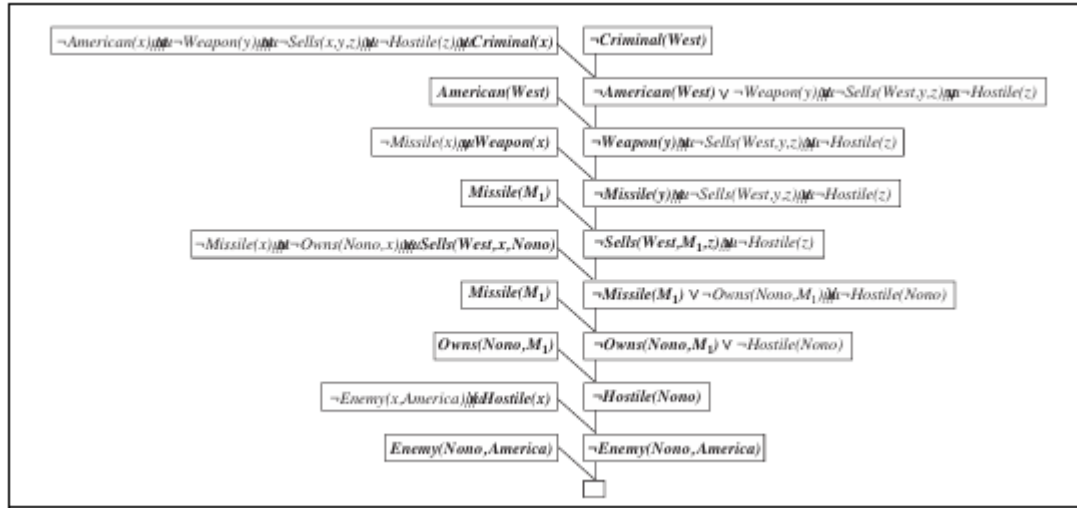


Figure 5.6: Are solution proof that West is a criminal. At each step, the literals that unify are in bold.

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 as follows:

Everyone who loves all animals is loved by someone.

Anyone who kills an animal 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. $\forall x [\exists z \text{ Animal}(z) \wedge \text{Kills}(x, z)] \Rightarrow [\forall y \neg \text{Loves}(y, x)]$
- C. $\forall x \text{ Animal}(x) \Rightarrow \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\forall x \text{ Cat}(x) \Rightarrow \text{Animal}(x)$
- ¬G. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

Now we apply the conversion procedure to convert each sentence to CNF:

- A1. $\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)$
- A2. $\neg \text{Loves}(x, F(x)) \vee \text{Loves}(G(x), x)$
- B. $\neg \text{Loves}(y, x) \vee \neg \text{Animal}(z) \vee \neg \text{Kills}(x, z)$
- C. $\neg \text{Animal}(x) \vee \text{Loves}(\text{Jack}, x)$
- D. $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$
- E. $\text{Cat}(\text{Tuna})$
- F. $\neg \text{Cat}(x) \vee \text{Animal}(x)$
- ¬G. $\neg \text{Kills}(\text{Curiosity}, \text{Tuna})$

The resolution proof that Curiosity killed the cat is given in Figure 5.7. 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.

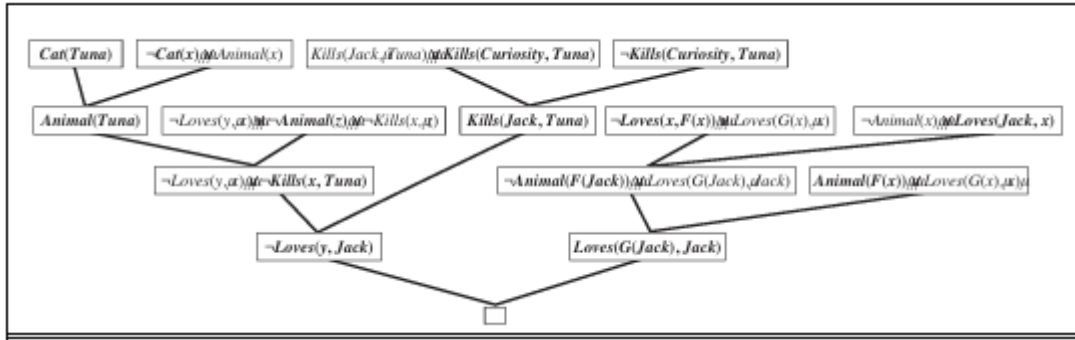


Figure 5.7: A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$. Notice also in the upper right, the unification of $Loves(x, F(x))$ and $Loves(Jack, x)$ can only succeed after the variables have been standardized apart.

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 $\exists w \text{ Kills}(w, Tuna)$, which, when negated, becomes $\neg \text{Kills}(w, Tuna)$ in CNF. Repeating the proof in Figure 5.7, 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.

Unfortunately, resolution can produce nonconstructive proofs for existential goals. For example, $\neg \text{Kills}(w, Tuna)$ resolves with $\text{Kills}(Jack, Tuna) \vee \text{Kills}(Curiosity, Tuna)$ to give $\text{Kills}(Jack, Tuna)$, which resolves again with $\neg \text{Kills}(w, Tuna)$ to yield the empty clause. Notice that w has two different bindings in this proof; resolution is telling us that, yes, someone killed Tuna—either Jack or Curiosity. This is no great surprise! One solution is to restrict the allowed resolution steps so that the query variables can be bound only once in a given proof; then we need to be able to backtrack over the possible bindings. Another solution is to add a special answer literal to the negated goal, which becomes $\neg \text{Kills}(w, Tuna) \vee \text{Answer}(w)$.

Now, the resolution process generates an answer whenever a clause is generated containing just a single answer literal. For the proof in Figure 5.7, this is $\text{Answer}(\text{Curiosity})$. The nonconstructive proof would generate the clause $\text{Answer}(\text{Curiosity}) \vee \text{Answer}(\text{Jack})$, which does not constitute an answer.

5.2.4 Completeness of resolution

The resolution is refutation-complete, which means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, $Q(x)$, by proving that $KB \wedge \neg Q(x)$ is unsatisfiable. We take it as given that any sentence in first-order logic (without equality) can be rewritten as a set of clauses in CNF. This can be proved by induction on the form of the sentence, using atomic sentences as the base case (Davis and Putnam, 1960). Our goal therefore is to prove the following: if S is an unsatisfiable set of clauses, then the application of a finite number of resolution steps to S will yield a contradiction.

Our proof sketch follows Robinson's original proof with some simplifications from Genesereth and Nilsson (1987). The basic structure of the proof (Figure 5.8) is as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of ground instances of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem).
2. We then appeal to the ground resolution theorem, which states that propositional resolution is complete for ground sentences.
3. We then use a lifting lemma to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

To carry out the first step, we need three new concepts:

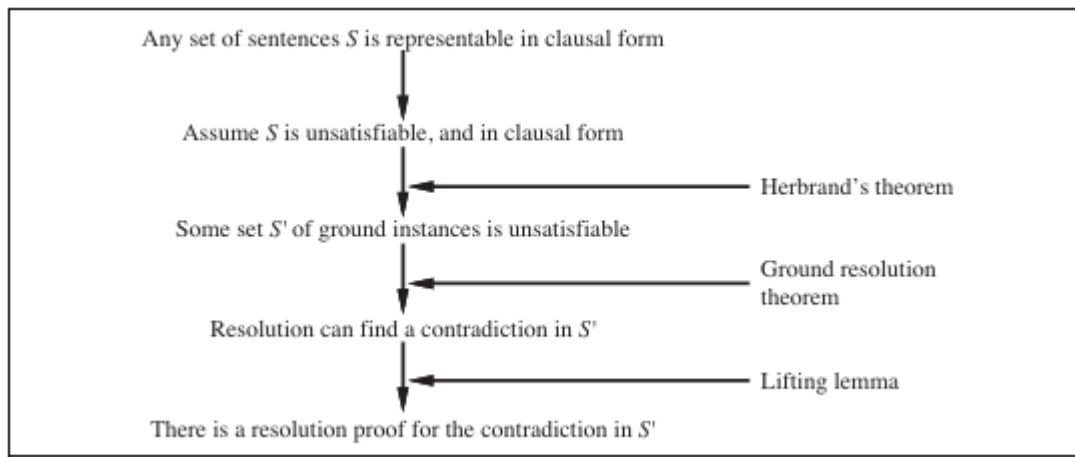


Figure 5.8: Structure of a completeness proof for resolution.

- **Herbrand universe:** If S is a set of clauses, then H_S , the Herbrand universe of S , is the set of all ground terms constructable from the following:

- a. The function symbols in S , if any.
- b. The constant symbols in S , if any; if none, then the constant symbol A .

For example, if S contains just the clause $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, then H_S is the following infinite set of ground terms:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \dots\}.$$

- **Saturation:** If S is a set of clauses and P is a set of ground terms, then $P(S)$, the saturation of S with respect to P , is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S .
- **Herbrand base:** The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S , written as $H_S(S)$. For example, if S contains solely the clause just given, then $H_S(S)$ is the infinite set of clauses

$$\begin{aligned} &\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ &\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ &\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ &\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots\} \end{aligned}$$

These definitions allow us to state a form of Herbrand's theorem (Herbrand, 1930): If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem (page 255) to show that the resolution closure $RC(S)$ contains the empty clause. That is, running propositional resolution to completion on S will derive a contradiction. Now that we have established that there is always a resolution proof involving some finite subset of the Herbrand base of S , the next step is to show that there is a resolution proof using the clauses of S itself, which are not necessarily ground clauses. We start by considering a single application of the resolution rule. Robinson stated this lemma:

Let C_1 and C_2 be two clauses with no shared variables, and let C'_1 and C'_2 be ground instances of C_1 and C_2 . If C' is a resolvent of C'_1 and C'_2 , then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C .

This is called a lifting lemma, because it lifts a proof step from ground clauses up to general first-order clauses. In order to prove his basic lifting lemma, Robinson had to invent unification and derive all of the properties of most general unifiers. Rather than repeat the proof here, we simply illustrate the lemma:

$$\begin{aligned} C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\ C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\ C'_1 &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C'_2 &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\ C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) . \end{aligned}$$

We see that indeed C is a ground instance of C . In general, for C_1 and C_2 to have any resolvents, they must be constructed by first applying to C_1 and C_2 the most general unifier of a pair of complementary literals in C_1 and C_2 . From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

For any clause C' in the resolution closure of S' there is a clause C in the resolution closure of S such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C' .

From this fact, it follows that if the empty clause appears in the resolution closure of S , it must also appear in the resolution closure of S . This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule. The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods were required to examine a huge number of arbitrary instantiations.

5.2.5 Equality

None of the inference methods described so far in this chapter handle an assertion of the form $x=y$. Three distinct approaches can be taken. The first approach is to axiomatize equality—to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one

for each predicate and function:

$$\begin{aligned}
 &\forall x \ x=x \\
 &\forall x, y \ x=y \Rightarrow y=x \\
 &\forall x, y, z \ x=y \wedge y=z \Rightarrow x=z \\
 \\
 &\forall x, y \ x=y \Rightarrow (P_1(x) \Leftrightarrow P_1(y)) \\
 &\forall x, y \ x=y \Rightarrow (P_2(x) \Leftrightarrow P_2(y)) \\
 &\vdots \\
 &\forall w, x, y, z \ w=y \wedge x=z \Rightarrow (F_1(w, x) = F_1(y, z)) \\
 &\forall w, x, y, z \ w=y \wedge x=z \Rightarrow (F_2(w, x) = F_2(y, z)) \\
 &\vdots
 \end{aligned}$$

Given these sentences, a standard inference procedure such as resolution can perform tasks requiring equality reasoning, such as solving mathematical equations. However, these axioms will generate a lot of conclusions, most of them not helpful to a proof. So there has been a search for more efficient ways of handling equality. One alternative is to add inference rules rather than axioms. The simplest rule, demodulation, takes a unit clause $x=y$ and some clause α that contains the term x , and yields a new clause formed by substituting y for x within α . It works if the term within α unifies with x ; it need not be exactly equal to x .

$Father(Father(x)) = PaternalGrandfather(x)$
 $Birthdate(Father(Father(Bella)), 1926)$

we can conclude by demodulation

$Birthdate(PaternalGrandfather(Bella), 1926) .$

More formally, we have

- **Demodulation:** For any terms x , y , and z , where z appears somewhere in literal m_i and where $UNIFY(x, z) = \theta$,

$$\frac{x = y, \quad m_1 \vee \dots \vee m_n}{SUB(SUBST(\theta, x), SUBST(\theta, y), m_1 \vee \dots \vee m_n)} .$$

where SUBST is the usual substitution of a binding list, and $SUB(x, y, m)$ means to replace x with y everywhere that x occurs within m .

The rule can also be extended to handle non-unit clauses in which an equality literal appears:

- **Paramodulation:** For any terms x , y , and z , where z appears somewhere in literal m_i , and where $UNIFY(x, z) = \theta$,

$$\frac{\ell_1 \vee \dots \vee \ell_k \vee x = y, \quad m_1 \vee \dots \vee m_n}{SUB(SUBST(\theta, x), SUBST(\theta, y), SUBST(\theta, \ell_1 \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_n))} .$$

For example, from

$P(F(x, B), x) \vee Q(x)$ and $F(A, y) = y \vee R(y)$

we have $\theta = UNIFY(F(A, y), F(x, B)) = \{x/A, y/B\}$, and we can conclude by paramodulation the sentence

$P(B, A) \vee Q(A) \vee R(B) .$

A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are provably equal under some substitution, where “provably” allows for equality reasoning. For example, the terms $1+2$ and $2+1$ normally are not unifiable, but a unification algorithm that knows that $x + y = y + x$ could unify them with the empty substitution.

5.2.6 Resolution strategies

we examine strategies that help find proofs efficiently.

- A) **Unit preference:** This strategy prefers to do resolutions where one of the sentences is a single literal (also known as a unit clause). The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses. Resolving a unit sentence (such as P) with any other sentence (such as $\neg P \vee \neg Q \vee R$) always yields a clause (in this case, $\neg Q \vee R$) that is shorter than the other clause. When the unit preference strategy was first tried for propositional inference in 1964, it led to a dramatic speedup, making it feasible to prove theorems that could not be handled without the preference. **Unit resolution** is a restricted form of resolution in which every resolution step must involve a unit clause. Unit resolution is incomplete in

general, but complete for Horn clauses. Unit resolution proofs on Horn clauses resemble forward chaining.

- B) **Set of support:** Preferences that try certain resolutions first are helpful, but in general it is more effective to try to eliminate some potential resolutions altogether. For example, we can insist that every resolution step involve at least one element of a special set of clauses—the set of support. The resolvent is then added into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically. We have to be careful with this approach because a bad choice for the set of support will make the algorithm incomplete. However, if we choose the set of support S so that the remainder of the sentences are jointly satisfiable, then set-of-support resolution is complete. For example, one can use the negated query as the set of support, on the assumption that the original knowledge base is consistent.
- C) **Input resolution:** In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof in Figure 9.11 on page 348 uses only input resolutions and has the characteristic shape of a single “spine” with single sentences combining onto the spine. Clearly, the space of proof trees of this shape is smaller than the space of all proof graphs. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences. Thus, it is no surprise that input resolution is complete for knowledge bases that are in Horn form, but incomplete in the general case. The linear resolution strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.
- D) **Subsumption:** The subsumption method eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

5.3 Classical Planning

The hybrid propositional logical agent can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the

simple action of moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations. In response to this, planning researchers have settled on a factored representation— one in which a state of the world is represented by a collection of variables. We use a language called PDDL, the Planning Domain Definition Language, that allows us to express all $4Tn^2$ actions with one action schema. There have been several versions of PDDL; we select simple version and alter its syntax to be consistent with the rest of the book. We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

Each state is represented as a conjunction of fluent that are ground, functionless atoms. For example, $Poor \wedge Unknown$ might represent the state of a hapless agent, and a state in a package delivery problem might be $At(Truck1,Melbourne) \wedge At(Truck2,Sydney)$. Database semantics is used: the closed-world assumption means that any fluent that are not mentioned are false, and the unique names assumption means that $Truck1$ and $Truck2$ are distinct. The following fluents are not allowed in a state: $At(x,y)$ (because it is non-ground), $\neg Poor$ (because it is a negation), and $At(Father(Fred),Sydney)$ (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a set of fluents, which can be manipulated with set operations. The set semantics is sometimes easier to deal with.

Actions are described by a set of action schemas that implicitly define the $ACTIONS(s)$ and $RESULT(s,a)$ functions needed to do a problem-solving search, to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its location by a vector Δ . A concise description of the action should mention only Δ ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

A set of ground (variable-free) actions can be represented by a single action schema. The

schema is a lifted representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

Action(*Fly*(*p*, *from*, *to*),
 PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
 EFFECT: $\neg At(p, from) \wedge At(p, to)$)

The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect. Although we haven't said yet how the action schema converts into logical sentences, think of the variables as being universally quantified. We are free to choose whatever values we want to instantiate the variables. For example, here is one ground action that results from substituting values for all the variables:

Action(*Fly*(*P₁*, *SFO*, *JFK*),
 PRECOND: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$
 EFFECT: $\neg At(P_1, SFO) \wedge At(P_1, JFK)$)

The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action. An action *a* can be executed in state *s* if *s* entails the precondition of *a*. Entailment can also be expressed with the set semantics: $s \models q$ iff every positive literal in *q* is in *s* and every negated literal in *q* is not. In formal notation we say

$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a),$$

where any variables in *a* are universally quantified. For example,

$$\forall p, from, to \ (Fly(p, from, to) \in \text{ACTIONS}(s)) \Leftrightarrow s \models (At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to))$$

We say that action *a* is applicable in state *s* if the preconditions are satisfied by *s*. When an action schema *a* contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the *Fly* action can be instantiated as *Fly*(*P₁*, *SFO*, *JFK*) or as *Fly*(*P₂*, *JFK*, *SFO*), both of which are applicable in the initial state. If an action *a* has *v* variables, then, in a domain with *k* unique names of objects, it takes $O(vk)$ time in the worst case to find the applicable ground actions.

```

Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
    ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
    ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
    PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
    PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: At(c, a) ∧ ¬ In(c, p))
Action(Fly(p, from, to),
    PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    EFFECT: ¬ At(p, from) ∧ At(p, to))

```

Figure 5.9: APDDL description of an air cargo transportation planning problem.

5.3.1 Example: Air cargo transport

Figure 5.9 shows an air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load, Unload, and Fly. The actions affect two predicates: *In*(*c*,*p*) means that cargo *c* is inside plane *p*, and *At*(*x*,*a*) means that object *x* (either plane or cargo) is at airport *a*. Note that some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *at* really means “available for use at a given location.” The following plan is a solution to the problem:

```

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),
  Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)] .

```

Finally, there is the problem of spurious actions such as *Fly*(*P*₁,*JFK*,*JFK*), which should be a no-op, but which has contradictory effects (according to the definition, the effect would include *At*(*P*₁,*JFK*) ∧ ¬*At*(*P*₁,*JFK*)). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the from and to airports must be different; see another example of this in Figure 5.10.

```

Init(On(A, Table) ∧ On(B, Table) ∧ On(C, A)
    ∧ Block(A) ∧ Block(B) ∧ Block(C) ∧ Clear(B) ∧ Clear(C))
Goal(On(A, B) ∧ On(B, C))
Action(Move(b, x, y),
    PRECOND: On(b, x) ∧ Clear(b) ∧ Clear(y) ∧ Block(b) ∧ Block(y) ∧
        (b ≠ x) ∧ (b ≠ y) ∧ (x ≠ y),
    EFFECT: On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y))
Action(MoveToTable(b, x),
    PRECOND: On(b, x) ∧ Clear(b) ∧ Block(b) ∧ (b ≠ x),
    EFFECT: On(b, Table) ∧ Clear(x) ∧ ¬On(b, x))

```

Figure 5.10: A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)]$.

5.3.2 Example: The spare tire problem

Consider the problem of changing a flat tire (Figure 10.2). The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is an abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is $[Remove(Flat,Axle), Remove(Spare,Trunk), PutOn(Spare,Axle)]$.

```

Init(Tire(Flat) ∧ Tire(Spare) ∧ At(Flat, Axle) ∧ At(Spare, Trunk))
Goal(At(Spare, Axle))
Action(Remove(obj, loc),
    PRECOND: At(obj, loc)
    EFFECT: ¬At(obj, loc) ∧ At(obj, Ground))
Action(PutOn(t, Axle),
    PRECOND: Tire(t) ∧ At(t, Ground) ∧ ¬At(Flat, Axle)
    EFFECT: ¬At(t, Ground) ∧ At(t, Axle))
Action(LeaveOvernight,
    PRECOND:
    EFFECT: ¬At(Spare, Ground) ∧ ¬At(Spare, Axle) ∧ ¬At(Spare, Trunk)
        ∧ ¬At(Flat, Ground) ∧ ¬At(Flat, Axle) ∧ ¬At(Flat, Trunk))

```

Figure 5.11: A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)]$.

5.3.3 Example: The blocks world

One of the most famous planning domains is known as the blocks world. This domain consists of a set of cube-shaped blocks sitting on a table.² The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it

to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block B on C. From Figure 10.4. We use $On(b,x)$ to indicate that block b is on x, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b,x,y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg \exists x On(x,b)$ or, alternatively, $\forall x \neg On(x,b)$. Basic PDDL does not allow quantifiers, so instead we introduce a predicate $Clear(x)$ that is true when nothing is on x. The action Move moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first attempt at the Move schema is

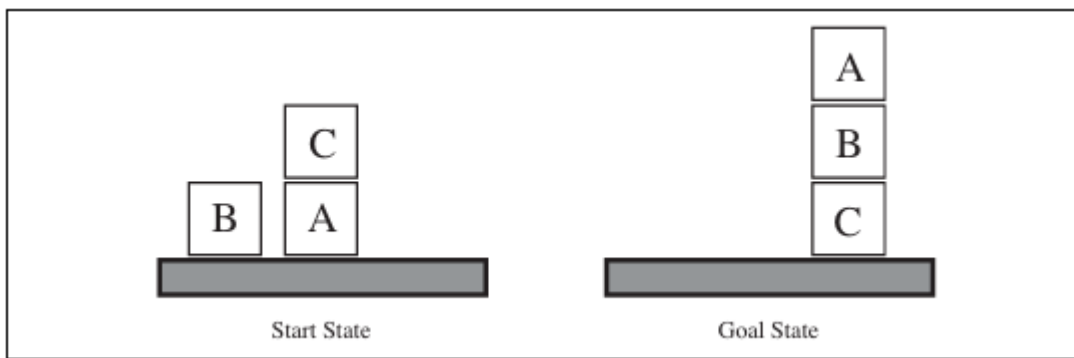


Figure 5.12: Diagram of the blocks-world problem.

Action($Move(b, x, y)$,
 PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y)$,
 EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$).

Unfortunately, this does not maintain $Clear$ properly when x or y is the table. When x is the Table, this action has the effect $Clear(Table)$, but the table should not become clear; and when $y=Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear for us to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

Action($MoveToTable(b, x)$,
 PRECOND: $On(b, x) \wedge Clear(b)$,
 EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$).

Second, we take the interpretation of $Clear(x)$ to be “there is a clear space on x to hold a block.” Under this interpretation, $Clear(Table)$ will always be true. The only problem is that nothing prevents the planner from using $Move(b,x,Table)$ instead of $MoveToTable(b,x)$. We

could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate Block and add Block (b) $\wedge$ Block (y) to the precondition of Move.

10.4 Algorithms for Planning as State-Space Search

The description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.4 compares forward and backward searches.

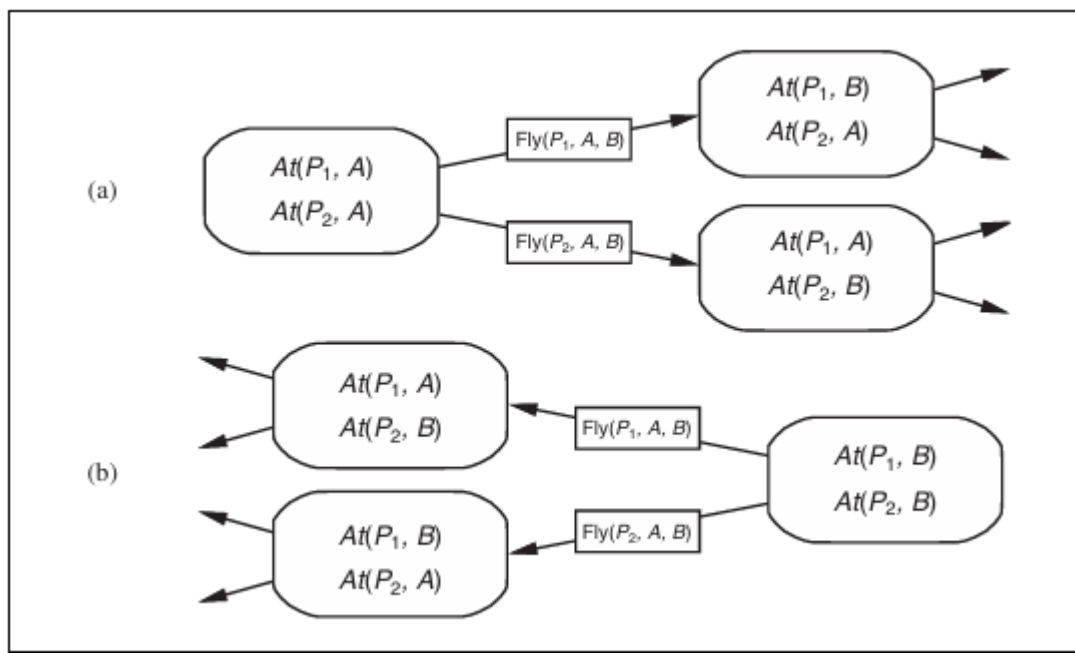


Figure 10.4: Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

10.4.1 Forward (progression) state-space search

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of AI: A Modern Approach from an online bookseller. Suppose there is an action schema $Buy(isbn)$ with effect $Own(isbn)$. ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal. **Second**, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where

each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 200041 nodes.

10.4.2 Backward (regression) relevant-states search

backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description g and a ground action a , the regression from g over a gives us a state description g' defined by

$$g' = (g - \text{ADD}(a)) \cup \text{Precond}(a).$$

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. For example, suppose the goal is to deliver a specific piece of cargo to SFO: $\text{At}(C_2, \text{SFO})$. That suggests the action $\text{Unload}(C_2, p, \text{SFO})$:

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

10.4.3 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic function. We look first at heuristics that add edges to the graph. For example, the ignore preconditions heuristic drops all preconditions from actions. Every action becomes applicable in every state, and any

single goal fluent can be achieved in one step. This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others. For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the set-cover problem. There is one minor irritation: the set-cover problem is NP-hard. Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of $\log n$ of the true minimum covering, where n is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only selected preconditions of actions. Consider the sliding block puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning problem involving tiles with a single schema Slide:

Action(Slide(t, s_1, s_2),
PRECOND: On(t, s_1) $\wedge$ Tile(t) $\wedge$ Blank(s_2) $\wedge$ Adjacent(s_1, s_2)
EFFECT: On(t, s_2) $\wedge$ Blank(s_1) $\wedge$ $\neg$ On(t, s_1) $\wedge$ $\neg$ Blank(s_2))

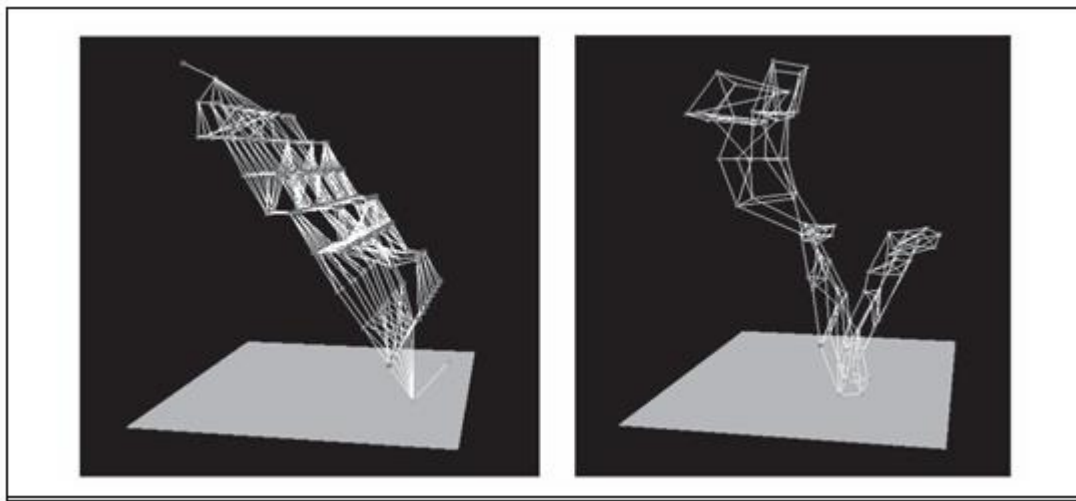


Figure 10.4.3.1: Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

Figure 10.4.3.1, diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the heuristic value. States on the bottom plane are solutions. In both these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hillclimbing search will easily find a solution to these problems (although it may not be an optimal solution).

10.5 Planning Graphs

All of the heuristics we have suggested can suffer from inaccuracies. This section shows how a special data structure called a planning graph can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN. A planning graph is a directed graph organized into levels: first a level S_0 for the initial state, consisting of nodes representing each fluent that holds in S_0 ; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0 ; then alternating levels S_i followed by A_i ; until we reach a termination condition.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned on page 368, it is straightforward to propositionalize a set of action schemas. Despite the resulting increase in the size of the problem description, planning graphs have proved to be effective tools for solving hard planning problems. Figure 10.7 shows a simple planning problem, and Figure 10.4.3.1 shows its planning graph. Each action at level A_i is connected to its preconditions at S_i and its effects at S_{i+1} .

So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it. This is represented by a persistence action (sometimes called a no-op). For every literal C , we add to the problem a persistence action with precondition C and effect C . Level A_0 in Figure 10.8 shows one “real” action, $\text{Eat}(\text{Cake})$, along with two persistence actions drawn as small square boxes. Level A_0 contains all the actions that could occur in state S_0 , but just as important it records conflicts between actions that would prevent them from occurring together. The gray lines in Figure 10.8 indicate mutual exclusion (or

mutex) links. For example, $\text{Eat}(\text{Cake})$ is mutually exclusive with the persistence of either $\text{Have}(\text{Cake})$ or $\neg\text{Eaten}(\text{Cake})$. We shall see shortly how mutex links are computed.

Level S_1 contains all the literals that could result from picking any subset of the actions in A_0 , as well as mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, $\text{Have}(\text{Cake})$ and $\text{Eaten}(\text{Cake})$ are mutex: depending on the choice of actions in A_0 , either, but not both, could be the result. In other words, S_1 represents a belief state: a set of possible states. The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.

We continue in this way, alternating between state level S_i and action level A_i until we reach a point where two consecutive levels are identical. At this point, we say that the graph has leveled off. The graph in Figure 10.4.3.1 levels off at S_2 . What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i , along with constraints saying that two actions cannot both be executed at the same level. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1} , along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does not require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links.

We now define mutex links for both actions and literals. A mutex relation holds between two actions at a given level if any of the following three conditions holds:

- Inconsistent effects: one action negates an effect of the other. For example, $\text{Eat}(\text{Cake})$ and the persistence of $\text{Have}(\text{Cake})$ have inconsistent effects because they disagree on the effect $\text{Have}(\text{Cake})$.
- Interference: one of the effects of one action is the negation of a precondition of the other. For example $\text{Eat}(\text{Cake})$ interferes with the persistence of $\text{Have}(\text{Cake})$ by negating its precondition.
- Competing needs: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $\text{Bake}(\text{Cake})$ and $\text{Eat}(\text{Cake})$ are mutex because they compete on the value of the $\text{Have}(\text{Cake})$ precondition.

A mutex relation holds between two literals at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive.

This condition is called inconsistent support. For example, Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat(Cake). In S2 the two literals are not mutex, because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex. A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each S_i has no more than l nodes and l^2 mutex links, and each A_i has no more than $a + l$ nodes (including the no-ops), $(a + l)^2$ mutex links, and $2(a + l)$ precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a + l)^2)$. The time to build the graph has the same complexity.

10.5.1 Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal g_i from state s as the level at which g_i first appears in the planning graph constructed from initial state s . We call this the level cost of g_i . In Figure 10.4.3.1, Have(Cake) has level cost 0 and Eaten(Cake) has level cost 1.

The estimate might not always be accurate, however, because planning graphs allow several actions at each level, whereas the heuristic counts just the level and not the number of actions. For this reason, it is common to use a serial planning graph for computing heuristics. A serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of non persistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

To estimate the cost of a conjunction of goals, there are three simple approaches. The max-level heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily accurate. The level sum heuristic, following the subgoal independence

assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable.

For our problem, the level-sum heuristic estimate for the conjunctive goal $Have(Cake) \wedge Eaten(Cake)$ will be $0+1=1$, where as the correct answer is 2, achieved by the plan $[Eat(Cake), Bake(Cake)]$. That doesn't seem so bad. A more serious error is that if $Bake(Cake)$ were not in the set of actions, then the estimate would still be 1, when in fact the conjunctive goal would be impossible.

Finally, the set-level heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without $Bake(Cake)$. It is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.

10.5.2 The GRAPHPLAN algorithm

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

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

```

Figure 10.5.2.1 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is

possible.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from page 370. The graph is shown in Figure 10.10. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph representing the initial state. The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents. Not shown are the unchanging positive literals (such as $\text{Tire}(\text{Spare})$) and the irrelevant negative literals.

The goal $\text{At}(\text{Spare}, \text{Axle})$ is not present in S_0 , so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into A_0 the three actions whose preconditions exist at level S_0 (i.e., all the actions except $\text{PutOn}(\text{Spare}, \text{Axle})$), along with persistence actions for all the literals in S_0 . The effects of the actions are added at level S_1 . EXPAND-GRAPH then looks for mutex relations and adds them to the graph. $\text{At}(\text{Spare}, \text{Axle})$ is still not present in S_1 , so again we do not call EXTRACT-SOLUTION.

We call EXPAND-GRAPH again, adding A_1 and S_1 and giving us the planning graph shown in Figure 10.5.2.2, Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:

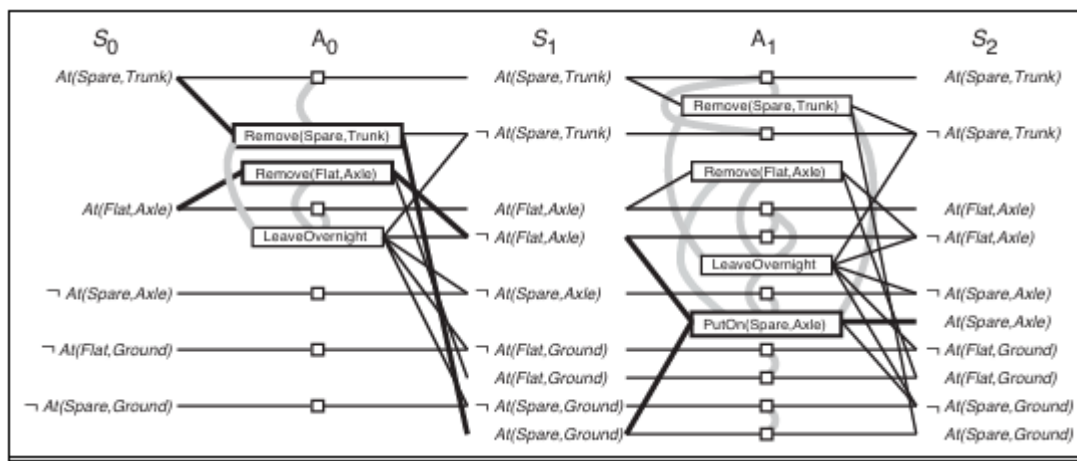


Figure 10.5.2.2 The planning graph for the spare tire problem after expansion to level S_2 .

Mutexlinks are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

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

This time, when we go back to the start of the loop, all the literals from the goal are present in S2, and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are in or out of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- The initial state is the last level of the planning graph, S_n , along with the set of goals from the planning problem.
- The actions available in a state at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. By “conflict free,” we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.

The goal is to reach a state at level S_0 such that all the goals are satisfied.

- The cost of each action is 1. For this particular problem, we start at S_2 with the goal

At(Spare,Axle). The only choice we have for achieving the goal set is PutOn(Spare,Axle). That brings us to a search state at S1 with goals At(Spare,Ground) and $\neg$ At(Flat,Axle). The former can be achieved only by Remove(Spare,Trunk), and the latter by either Remove(Flat,Axle) or LeaveOvernight.

But LeaveOvernight is mutex with Remove(Spare,Trunk), so the only solution is to choose Remove(Spare,Trunk) and Remove(Flat,Axle). That brings us to a search state at S0 with the goals At(Spare,Trunk) and At(Flat,Axle). Both of these are present in the state, so we have a solution: the actions Remove(Spare, Trunk) and Remove(Flat, Axle) in level A0, followed by PutOn(Spare, Axle) in A1. In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the (level,goals) pair as a no-good, just as we did in constraint learning for CSPs (page 220). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We see shortly that no-goods are also used in the termination test.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. For any set of goals, we proceed in the following order:

1. Pick first the literal with the highest level cost.
2. To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

10.5.3 Termination of GRAPHPLAN

How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So if it is possible that there might be fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution. Now

all we have to do is prove that the graph and the no-goods will always level off. The key to this proof is that certain properties of planning graphs are monotonically increasing or decreasing. “X increases monotonically” means that the set of Xs at level $i + 1$ is a superset (not necessarily proper) of the set at level i . The properties are as follows:

- **Literals increase monotonically:** Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- **Actions increase monotonically:** Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of the monotonic increase of literals; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
- **Mutexes decrease monotonically:** If two actions are mutex at a given level A_i , then they will also be mutex for all previous levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level S_i nor actions that cannot be executed at level A_i . We can see that “mutexes decrease monotonically” is true if you consider that these invisible literals and actions are mutex with everything. The proof can be handled by cases: if actions A and B are mutex at level A_i , it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at A_i , they will be mutex at every level. The third case, competing needs, depends on conditions at level S_i : that level must contain a precondition of A that is mutex with a precondition of B. Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so, by induction, the mutexes must be decreasing.
- **No-goods decrease monotonically:** If a set of goals is not achievable at a given level, then they are not achievable in any previous level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.