UNIT I INTELLIGENT AGENTS

Introduction to AI –Agents and Environments –Concept of rationality –Nature of environments –Structure of agents -Problem solving agents –search algorithms –uninformed search strategies

Introduction

- **Artificial Intelligence** is the branch of computer science concerned with making computers behave like humans. (Or) Artificial Intelligence is the ability of a computer to act and think like human
- Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success.
- **John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."
- The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :
  - **Systems that think like human**
  - **Systems that act like human**
  - **Systems that think rationally**
  - **Systems that act rationally**

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"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	**Acting Rationally**
"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)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

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

Thinking humanly: The cognitive modeling approach

To say that a program thinks like a human, consider the human thinking which can be expressed in three ways:

introspection—trying to catch our own thoughts as they go by;

psychological experiments—observing a person in action;
brain imaging—observing the brain in action.
Once there is sufficient precise theory of the mind, it becomes possible to express the theory as a computer program

Cognitive Study of Human mind:

It is a highly interdisciplinary field which combines ideas and methods from psychology, computer science, philosophy, linguistics and neuroscience.
The goal of cognitive science is to characterize the nature of human knowledge and how that knowledge is used, processed and acquired

Acting humanly: The Turing Test approach

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 new conclusions;
• **machine learning** to adapt to new circumstances and to detect and to discover new patterns
• **computer vision** to perceive object and
• **robotics** to manipulate objects and to move about

Thinking rationally: The "laws of thought" approach

- Aristotle The concept of "right thinking" was proposed by Aristotle. His syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises.
- The canonical example starts with Socrates is a man and all men are mortal and concludes that Socrates is mortal.
- These laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic.
- Logicians developed a precise notation for statements about objects in the world and the relations among them .Logics are needed to create intelligent systems.

Drawbacks:

First, it is not easy to convert statements into logic 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. Even problems with just a few hundred facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first.

Acting rationally: The rational agent approach

Agent

An agent is just something that acts operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, create and pursue goals. A rational agent is one that acts so as to achieve the best outcome

One way to act rationally is to deduce that a given action is best and then to act on that conclusion. On the other hand, there are ways of acting rationally that cannot be said to involve inference.

The rational-agent approach to AI has **two 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 suitable for scientific development.

Beneficial machines

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

For an artificially defined task such as chess or shortest-path computation, the task comes with an objective built in—so the standard model is applicable. As with the real world, however, it becomes more and more difficult to specify the objective completely and correctly

The value alignment problem: the values or objectives put into the machine must be aligned with those of the human.

The Foundations of Artificial Intelligence

Philosophy:

Aristotle was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which in principle allowed one to generate conclusions mechanically, given initial premises. This constitutes rationalism

Dualism:There is a part of the human mind that is outside of nature, exempt from physical laws. An alternative to dualism is **materialism**, which holds that the brain's operation according to the laws of physics *constitutes* the mind.

Empiricism: Knowledge is based on experience and experimentation.

Principle of **induction**: General rules acquired by exposure to repeated associations between their elements.

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

The final element in the philosophical picture of the mind is the connection between knowledge and action. Jeremy Bentham and John Stuart Mill promoted the idea of **utilitarianism**: that rational decision making based on maximizing utility should apply to all spheres of human activity, including public policy decisions made on behalf of many individuals. Utilitarianism is a specific kind of **consequentialism**: the idea that what is right and wrong is determined by the expected outcomes of an action. In contrast, Immanuel Kant, in 1875 proposed a theory of rule-based or **deontological ethics**, in which "doing the right thing" is determined not by outcomes but by universal social laws that govern allowable actions, such as "don't lie" or "don't kill."

Mathematics

The fundamental ideas of AI required the mathematization of logic and probability and the introduction of a new branch of mathematics: computation.

The idea of **formal logic** can be traced back to the philosophers of ancient Greece, India, and China, but its mathematical development really began with the work of George Boole who worked out the details of propositional, or Boolean, logic.

The theory of **probability** can be seen as generalizing logic to situations with uncertain information—a consideration of great importance for AI. Gerolamo Cardano first framed the idea of probability, describing it in terms of the possible outcomes of gambling events.

The history of computation is as old as the history of numbers, but the first nontrivial **algorithm** is thought to be Euclid's algorithm for computing greatest common divisors. Alan Turing tried to characterize exactly which functions *are* **computable**—capable of being computed by an effective procedure. The Church–Turing thesis proposes to identify the general notion of computability with functions computed by a Turing machine (Turing, 1936). Turing also showed that there were some functions that no Turing machine can compute. The theory of **NP-completeness**, pioneered by Cook and Karp , provides a basis for analyzing the tractability of problems.

Economics

Most people think of economics as about money but economist will say that Economics is no longer the study of money; rather it is the study of desires and preferences.

Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for individual decisions (economic or otherwise) made under uncertainty.

A formalized class of sequential decision problems called **Markov decision processes** is used in decision theory

Work in economics and operations research has contributed much to the notion of rational agents.

Neuroscience

Neuroscience is the study of the nervous system, particularly the brain. There are localized areas of brain responsible for specific cognitive functions for example area in the left hemisphere is responsible for speech production. Brain consists of largely of nerve cells, or **neurons**, Camillo Golgi developed a staining technique allowing the observation of individual neurons. *a* Thus a collection of simple cells can lead to thought, action, and consciousness. There are mapping between areas of the brain and the parts of the body that they control or from which they receive sensory input.

The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG). The development of functional magnetic resonance imaging (fMRI) is giving neuroscientists unprecedentedly detailed images of brain activity.

Psychology

The origins of scientific psychology are usually traced to the work of the German physicist, the scientific method to the study of human vision. In 1879, Wundt opened the first laboratory of experimental psychology, at the University of Leipzig. Wundt insisted on carefully controlled

experiments in which his workers would perform a perceptual or associative task while introspecting on their thought processes.

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

Today IA(Intelligence Augumentation) and AI are the two sides of the same coin, with the former emphasizing human control and the latter emphasizing intelligent behavior on the part of the machine. Both are needed for machines to be useful to humans.

Computer engineering

Intelligence and atrifacts are needed for the success of AI. The first *operational* computer was used for deciphering German messages.The first operational *programmable* computer was the Z-3.The first *electronic* computer, the ABC, was assembled it was then ENIAC, developed as part of a secret military project that proved to be the most influential forerunner of modern computers.Each generation of computer brings in speed and capacity and decrease in price

Of course, there were calculating devices before the electronic computer.The first *programmable* machine was a loom, that used punched cards to store instructions for the pattern to be woven. In the mid-19th century, Charles Babbage designed two computing machines, neither of which he completed. The Difference Engine was intended to compute mathematical tables for engineering and scientific projects. Analytical Engine included addressable memory, stored programs based on Jacquard's punched cards, and conditional jumps. AI also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs

Control theory and cybernetics

The first self-controlling machine developed was a water clock with a regulator that maintained a constant flow rate. Other examples of self-regulating feedback control systems include the steam engine governor, and the thermostat.

James Clerk Maxwell initiated the mathematical theory of control systems.The **control theory** developed an interest in biological and mechanical control systems and their connection to cognition.They viewed purposive behavior as arising from a regulatory mechanism trying to minimize "error"—the difference between current state and goal state.

Cybernetics is the science of communications and automatic control systems in both machine and living things. Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize a **cost function** over time.

Linguistics

Verbal Behavior was a comprehensive, detailed account of the behaviorist approach to language learning. Chomsky pointed out that the behaviorist theory did not address the notion of creativity in language—it did not explain how children could understand and make up sentences that they had never heard before. syntactic models could handle this as it was was formal enough that it could in principle be programmed.

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

The History of Artificial Intelligence

One quick way to summarize the milestones in AI history is to list the Turing Award winners: Marvin Minsky (1969) and John McCarthy (1971) for defining the foundations of the field based on representation and reasoning; Ed Feigenbaum and Raj Reddy (1994) for developing expert systems that encode human knowledge to solve real-world problems; Judea Pearl (2011) for developing probabilistic reasoning techniques that deal with uncertainty in a principled manner; and finally Yoshua Bengio, Geoffrey Hinton, and Yann LeCun (2019) for making "deep learning" (multilayer neural networks) a critical part of modern computing. The rest of this section goes into more detail on each phase of AI history.

The inception of artificial intelligence (1943–1956)

- The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943).They proposed a model of artificial neurons in which each neuron is characterized as being "on" or "off," with a switch to "on" occurring in response to stimulation by a sufficient number of neighboring neurons. McCulloch and Pitts also suggested that suitably defined networks could learn.
- Edmonds, built the first neural network computer in 1950. The SNARC, as it was called, used 3000 vacuum tubes and a surplus automatic pilot mechanism that can simulate a network of 40 neurons. Later, at Princeton, Minsky studied universal computation in neural networks. Alan Turing introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning.
- In 1955, John McCarthy of Dartmouth College convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at Dartmouth in the summer of 1956. This was the first official usage of McCarthy's term *artificial intelligence*.
- Despite this optimistic prediction, the Dartmouth workshop did not lead to any breakthroughs. Newell and Simon presented perhaps the most mature work, a mathematical theorem-proving system called the Logic Theorist (LT). Simon claimed, "We have invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind–body problem."
- Newell and Simon also invented a list-processing language, IPL, to write LT. They had no compiler and translated it into machine code by hand. To avoid errors, they worked in parallel, calling out binary numbers to each other as they wrote each instruction to make sure they agreed.

Early enthusiasm, great expectations (1952–1969)

- Newell and Simon followed up their success with LT with the General Problem Solver, or GPS. Unlike LT, this program was designed from the start to imitate human problem-solving protocols. Within the limited class of puzzles it could handle, it turned out that the order in which the program considered subgoals and possible actions was similar to that in which humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach.
- The success of GPS and subsequent programs as models of cognition led Newell and Simon (1976) to formulate the famous **physical symbol system** hypothesis, which states that "a physical symbol system has the necessary and sufficient means for general intelligent action."

- At IBM, Nathaniel Rochester and his colleagues produced some of the first AI programs. Herbert Gelernter (1959) constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky.
- Samuel wrote program on checkers (draughts) that eventually learned to play . He thereby disproved the idea that computers can do only what they are told to: his program quickly learned to play a better game than its creator.
- In 1958, John McCarthy made two important contributions to AI. In MIT AI Lab .He defined the high-level language **Lisp**, which was to become the dominant AI programming language for the next 30 years. McCarthy also invented the Advice Taker, a hypothetical program that would embody general knowledge of the world and could use it to derive plans of action. 1958 also marked the year that Marvin Minsky moved to MIT.
- In 1963, McCarthy started the AI lab at Stanford. His plan to use logic to build the ultimate Advice Taker was advanced by J. A. Robinson's discovery in 1965 of the resolution method (a complete theorem-proving algorithm for first-order logic;).
- At MIT, Minsky supervised a series of students who chose limited problems that appeared to require intelligence to solve. These limited domains are known as **microworlds**.
- James Slagle's SAINT program (1963) was able to solve closed-form calculus integration problems typical of first-year college courses.
- Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests.
- Daniel Bobrow's STUDENT program (1967) solved algebra story problems.The most famous microworld is the **blocks world**, which consists of a set of solid blocks placed on a tabletop (or more often, a simulation of a tabletop). A typical task in this world is to rearrange the blocks in a certain way, using a robot hand that can pick up one block at a time.

A dose of reality (1966–1973)

- The early AI Systems tuned out to fail when tried on wider selection of problems and more difficult problems
- There were two main reasons for this failure. The first was that many early AI systems were based primarily on "informed introspection" as to how humans perform a task, rather than on a careful analysis of the task, what it means to be a solution, and what an algorithm would need to do to reliably produce such solutions.
- The second reason for failure was a lack of appreciation of the intractability of many of the problems that AI was attempting to solve. Most of the early problem-solving systems worked by trying out different combinations of steps until the solution was found. This strategy worked initially because microworlds contained very few objects and hence very few possible actions and very short solution sequences.
- A third difficulty arose because of some fundamental limitations on the basic structures being used to generate intelligent behavior. For example, Perceptrons (a simple form of neural network) could be shown to learn anything they were capable of representing, they could represent very little.

Expert systems (1969–1986)

The picture of problem solving that had arisen during the first decade of AI research was of a general-purpose search and now is to use more powerful, domain-specific knowledge that allows larger reasoning steps and can more easily handle typically occurring cases in narrow areas of expertise.

- The DENDRAL program was an early example of this approach. It helps to solve the problem of inferring molecular structure from the information provided by a mass spectrometer. The significance of DENDRAL is that it uses large number of special rules.
- The next major effort was the MYCIN system for diagnosing blood infections. With about 450 rules, MYCIN was able to perform as well as some experts, and considerably better than junior doctors.
- The importance of domain knowledge was also apparent in the area of natural language understanding. SHRDLU system was designed for this purpose.
- The first successful commercial expert system, R1, began operation at the Digital Equipment Corporation (McDermott, 1982). The program helped configure orders for new computer systems.
- Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988, including hundreds of companies building expert systems, vision systems, robots, and software and hardware specialized for these purposes.

The return of neural networks (1986–present)

In the mid-1980s at least four different groups reinvented the **back-propagation** learning algorithm first developed in the early 1960s. The algorithm was applied to many learning problems in computer science and psychology, and the widespread dissemination of the results in the collection *Parallel Distributed Processing* called "**Connectionist Model"**

Probabilistic reasoning and machine learning (1987– present)

- The brittleness of expert systems led to a new, more scientific approach incorporating probability rather than Boolean logic, machine learning rather than hand-coding, and experimental results rather than philosophical claims.
- In the 1980s, approaches using **hidden Markov models** (HMMs) came to dominate the area of speech recognition. Two aspects of HMMs are relevant. First, they are based on a rigorous mathematical theory. Second, they are generated by a process of training on a large corpus of real speech data.
- 1988 was an important year for the connection between AI and other fields, including statistics, operations research, decision theory, and control theory. Judea Pearl's *Probabilistic Reasoning in Intelligent Systems* led to a new acceptance of probability and decision theory in AI. Pearl's development of **Bayesian networks** yielded a rigorous and efficient formalism for representing uncertain knowledge as well as practical algorithms for probabilistic reasoning.

Big data (2001–present)

Remarkable advances in computing power and the creation of the World Wide Web have facilitated the creation of very large data sets—a phenomenon sometimes known as **big data**. These data sets include trillions of words of text, billions of images, and billions of hours of speech and video, as well as vast amounts of genomic data, vehicle tracking data, clickstream data, social network data, and so on. This has led to the development of learning algorithms specially designed to take advantage of very large data sets. Often, the vast majority of examples in such data sets are *unlabeled*; The availability of big data and the shift towards machine learning helped AI recover commercial attractiveness.

Deep learning (2011–present)

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

recognition in the 1990s . It was not until 2011, however, that deep learning methods really took off. This occurred first in speech recognition and then in visual object recognition. In the 2012 ImageNet competition, which required classifying images into one of a thousand categories (armadillo, bookshelf, corkscrew, etc.), a deep learning system created in Geoffrey Hinton's group at the University of Toronto demonstrated a dramatic improvement over previous systems, which were based largely on handcrafted features. Since then, deep learning systems have exceeded human performance on some vision tasks

Deep learning relies heavily on powerful hardware. Whereas a standard computer CPU can do 10^9 or 10^{10} operations per second. a deep learning algorithm running on specialized hardware (e.g., GPU, TPU, or FPGA) might consume between 10^{14} and 10^{17} operations per second, mostly in the form of highly parallelized matrix and vector operations. Of course, deep learning also depends on the availability of large amounts of training data, and on a few algorithmic tricks

The State of the Art or The Applications of AI

ROBOTIC VEHICLES:

The history of robotic vehicles stretches back to radio-controlled cars of the 1920s, but the first demonstrations of autonomous road driving without special guides occurred in the 1980s. A driverless robotic car named STANLEY outfitted with cameras, radar, and laser rangefinders to sense the environment and onboard software to command the steering, braking, and acceleration.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 . 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 plans the daily operations for NASA's Mars Exploration Rovers, and MEXAR2 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.

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

Logistics planning: During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART to do automated logistics planning and scheduling for transportation.

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 one language to another.

1.2Agents and Environments

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in **Figure**

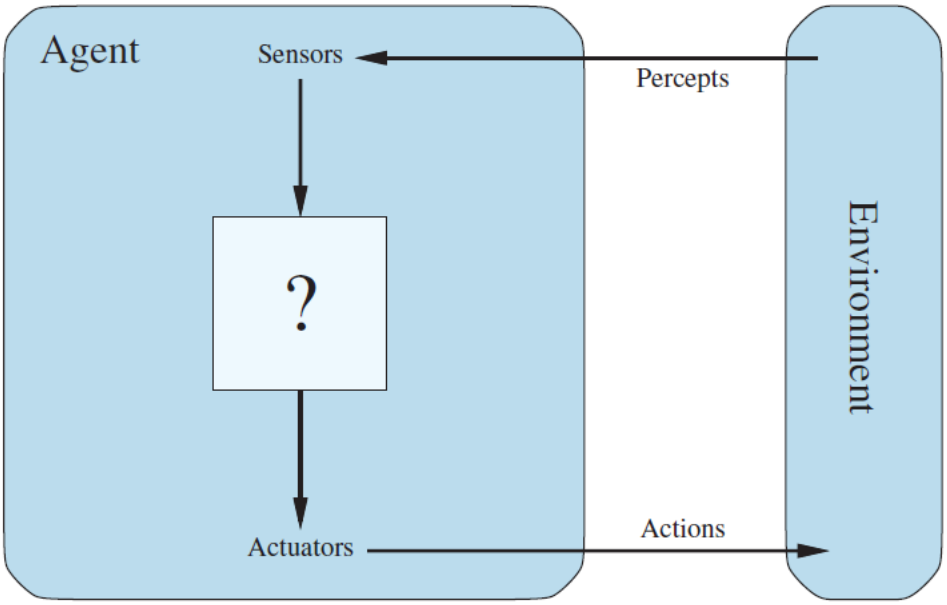


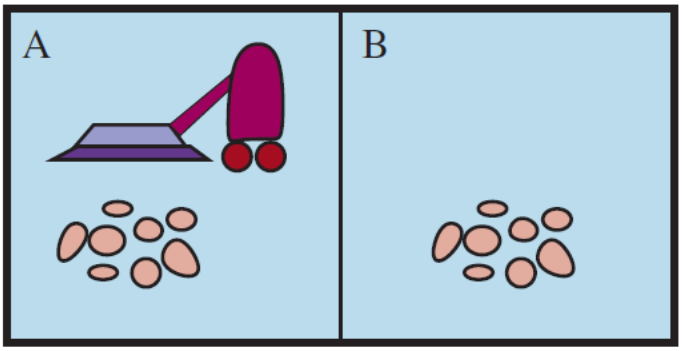
Figure :Agents interact with environments through sensors and actuators.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives file contents, network packets, and human input (keyboard/mouse/touchscreen/voice) as sensory inputs and acts on the environment by writing files, sending network packets, and displaying information or generating sounds.
- The environment could be everything—the entire universe!

The term **percept** to refer to the content an agent's sensors are perceiving. An agent's **percept sequence** is the complete history of everything the agent has ever perceived. An agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

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

Consider a simple example—the vacuum-cleaner world, which consists of a robotic vacuum-cleaning agent in a world consisting of squares that can be either dirty or clean. **Figure** shows a configuration with just two squares A and B .



The vacuum agent perceives which square it is in and whether there is dirt in the square. The agent starts in square A . The available actions are to move to the right, move to the left, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. A partial tabulation of this agent function is shown in **Figure**

Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Dirty]$	$Suck$
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Clean], [A, Dirty]$	$Suck$
⋮	⋮

1.3 The Concept of Rationality(Good Behaviour)

- A **rational agent** is one that does the right thing.
- An agent's behavior is evaluated by its consequences.
- When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.
- This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

Rationality 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.

Omniscience, learning, and autonomy

- An omniscient agent knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality
- Rationality maximizes *expected* performance, while perfection maximizes *actual* performance.
- Doing actions *in order to modify future percepts*— sometimes called **information gathering**—is an important part of rationality
- A rational agent not only gathers information but also has to **learn** as much as possible from what it perceives.
- The agent's initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented.
- There are extreme cases in which the environment is completely known *a priori* and completely predictable. In such cases, the agent need not perceive or learn; it simply acts correctly.
- When an agent relies on the prior knowledge of its designer rather than on its own percepts and learning processes, the agent lacks **autonomy**.
- A rational agent should be autonomous—it should learn what it can compensate for partial or incorrect prior knowledge.
- After sufficient experience of its environment, the behavior of a rational agent can become effectively *independent* of its prior knowledge. Hence, the incorporation of learning allows one to design a single rational agent that will succeed in a vast variety of environments.

1.4 The Nature of Environments

T**ask environments** are essentially the "problems" to which rational agents are the "solutions."
The nature of the task environment directly affects the appropriate design for the agent program.

Specifying the task environment

When designing an agent, the first step is to specify the task environment as fully as possible using

***PEAS*(Performance Measure Environment Actuator Sensor)**

Example :PEAS description of the task environment for an automated taxi driver.

Performance Measure

Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits.

E**nvironment**

Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways. The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles, and potholes.

The **actuators** for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise.
The basic **sensors** for the taxi will include one or more video cameras and ultrasound sensors to detect distances to other cars and obstacles.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen

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

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, raw materials, operators	Valves, pumps, heaters, stirrers, displays	Temperature, pressure, flow, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, feedback, speech	Keyboard entry, voice

Figure: Examples of agent types and their PEAS descriptions

Properties of task environments(Types of environment)

(i)FULLY OBSERVABLE VS. PARTIALLY OBSERVABLE:

- If an agent's sensors gives access to the complete state of the environment at each point in time, then the task environment is fully observable.
- Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world.
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data
- For example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares.
- If the agent has no sensors at all then the environment is **unobservable**.

(ii)SINGLE-AGENT VS. MULTIAGENT:

In a single agent environment there is well defined agent who takes the decision
For example an agent solving a crossword puzzle by itself is clearly in a single-agent environment,
In a multiagent agent environment a group of agents are involved in taking the decision
Example an agent playing chess is in a two agent environment.
Multiagents are of two types

- **Competitive and**
- **Cooperative**

(iii)DETERMINISTIC VS. NONDETERMINISTIC.

- If the next state of the environment is completely determined by the current state and the action executed by the agent(s), then the environment is deterministic; otherwise, it is nondeterministic. Taxi driving is clearly nondeterministic in this sense, because one can never predict the behavior of traffic exactly;
- The vacuum world is deterministic, but variations can include nondeterministic elements such as randomly appearing dirt and an unreliable suction mechanism.
- The word **stochastic** is used as a synonym for "nondeterministic," a distinction between the two terms; A model of the environment is stochastic if it explicitly deals with probabilities (e.g., "there's a 25% chance of rain tomorrow") and "nondeterministic" if the possibilities are listed without being quantified (e.g., "there's a chance of rain tomorrow").

(iv)EPISODIC VS. SEQUENTIAL:

- In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action.
- The next episode does not depend on the actions taken in previous episodes.
- Many classification tasks are episodic. For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective.
- In sequential environments, on the other hand, the current decision could affect all future decisions.
- Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences.

- Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

(v)STATIC VS. DYNAMIC:

- If the environment can change while an agent is deliberating, then the environment is dynamic for that agent; otherwise, it is static.
- Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time.
- Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing. If the environment itself does not change with the passage of time but the agent's performance score does, then the environment is **semidynamic**.
- Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next. Chess, when played with a clock, is semidynamic. Crossword puzzles are static.

(vi)DISCRETE VS. CONTINUOUS:

- A discrete environment has a finite number of distinct states over time. Each state has associated percepts and actions on the agent. Eg: Chess has a discrete set of percepts and actions.
- In a continuous environment , the environment is not stable at any given point of time and also it changes continuously. Eg: Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values

(vii)KNOWN VS. UNKNOWN:

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

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

Figure: Examples of task environments and their characteristics.

1.5 The Structure of Agents

The job of AI is to design an **agent program** that implements the agent function—the mapping from percepts to actions.

Agent architecture is a program that runs on some sort of computing device with physical sensors and actuators.
In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated.

agent = architecture + program

Agent programs

- The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators.

- The difference between the agent program and the agent function is, the agent program takes the current percept as input, and the agent function, which may depend on the entire percept history.

- The agent program has no choice but has to take just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

- The agent programs are described using a simple pseudocode language
- For example, Figure shows a trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do.

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

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

The four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents.

Simple reflex agents

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

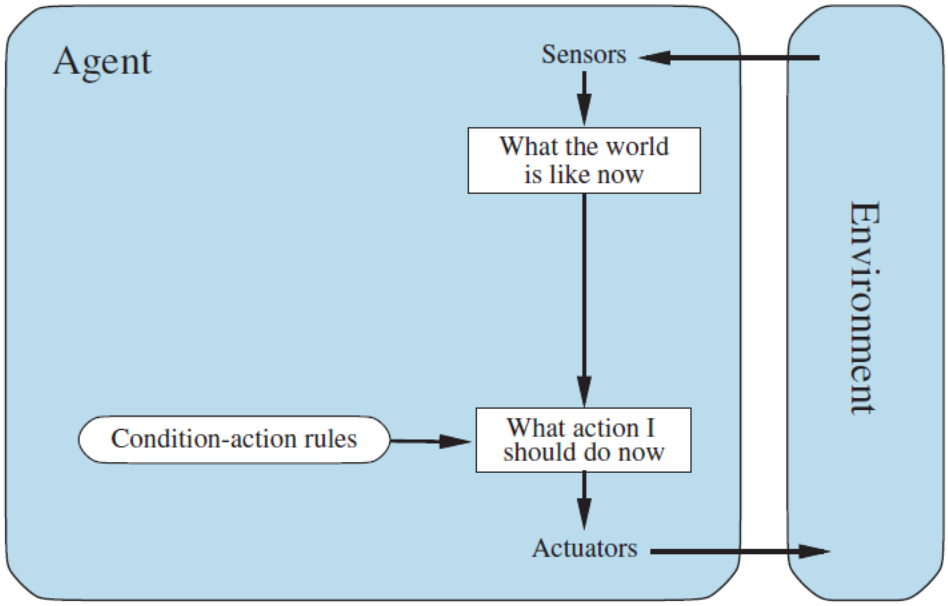


Figure: Schematic diagram of a simple reflex agent. Rectangles denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

For example, the vacuum agent whose agent function is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt.

An agent program for this agent is shown in Figure .

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

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

Figure gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.

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

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

The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULEMATCH function returns the first rule from the set of rules that matches the given state description.

Simple reflex agents have the admirable property of being simple, but they turn out to be of limited intelligence. The agent will work *only if the correct decision can be made on the basis of only the current percept—that is, only if the environment is fully observable.*
Suppose that a simple reflex ,vacuum agent in which its location sensor got damaged and has only a dirt sensor. Such an agent has just two possible percepts: [Dirty] and [Clean]. It can Suck in response to [Dirty]; and there will not be any response to [Clean]. Moving Left fails (forever) if it happens to start in square A, and moving Right fails (forever) if it happens to start in square B.

Model-based reflex agents

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now*. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.
Updating this internal state information requires two kinds of knowledge.
First, some information about how the world changes over time is needed, which can be divided roughly into two parts: the effects of the agent's actions and how the world evolves independently of the agent. A knowledge about "how the world works"—is called a **transition model** of the world.Second, some information about how the state of the world is reflected in the agent's percepts is also needed. This kind of knowledge is called a **sensor model**.Together, the transition model and sensor model allow an agent to keep track of the state of the world—to the extent possible given the limitations of the agent's sensors. An agent that uses such models is called a **model-based agent**.
Figure gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works.

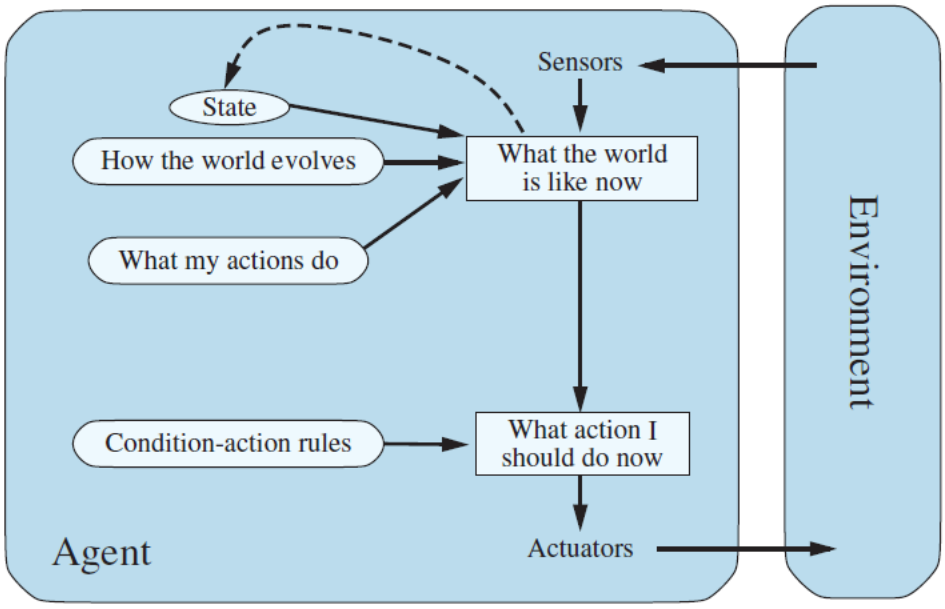


Figure A model-based reflex agent

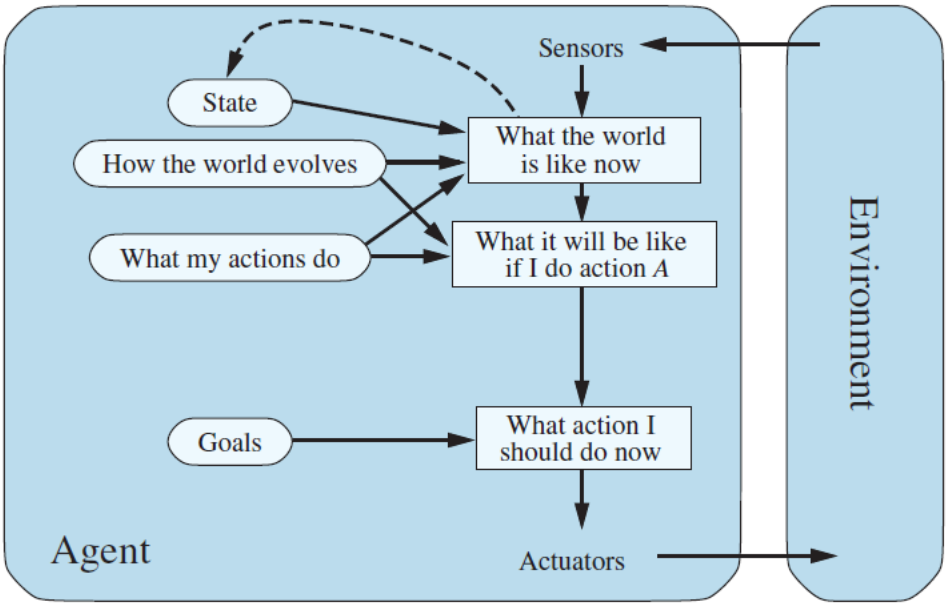
```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              transition_model, a description of how the next state depends on
                    the current state and action
              sensor_model, a description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

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

A model-based reflex agent. keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.
The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description

Goal-based agents
The current state of the environment is not always enough to decide what to do next. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, along with current state description, the agent needs some sort of **goal** information that describes situations that are desirable—for example, being at a particular destination.
The agent program can combine with the model-based reflex agent to choose actions that achieve the goal.

Figure shows the goal-based agent's structure.



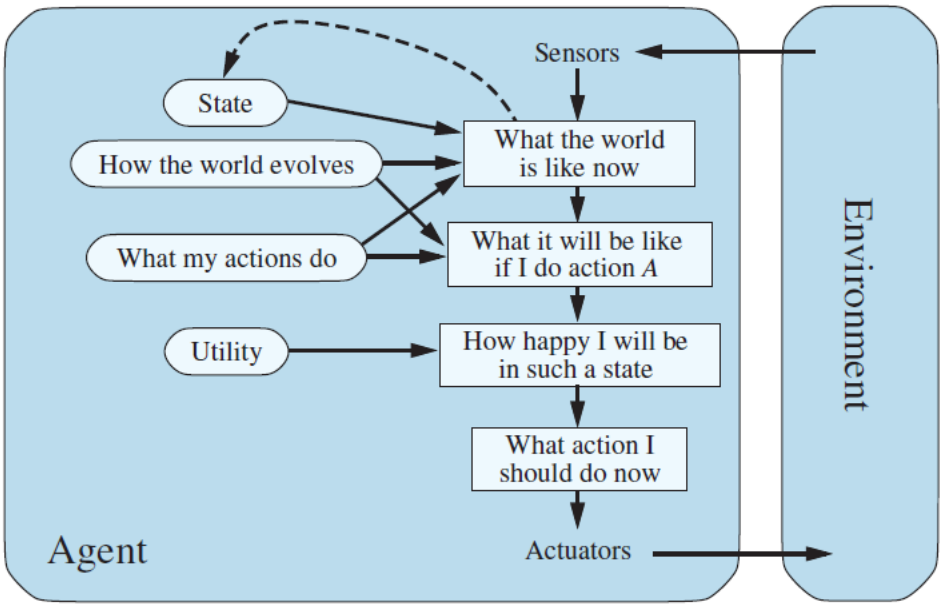
A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals. Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal. **Search** and **planning** are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. For example, a goal-based agent's behavior can easily be changed to go to a different destination, simply by specifying that destination as the goal. The reflex agent's rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal), but some are quicker, safer, more reliable, or cheaper than others.

Figure A model-based, utility-based agent.



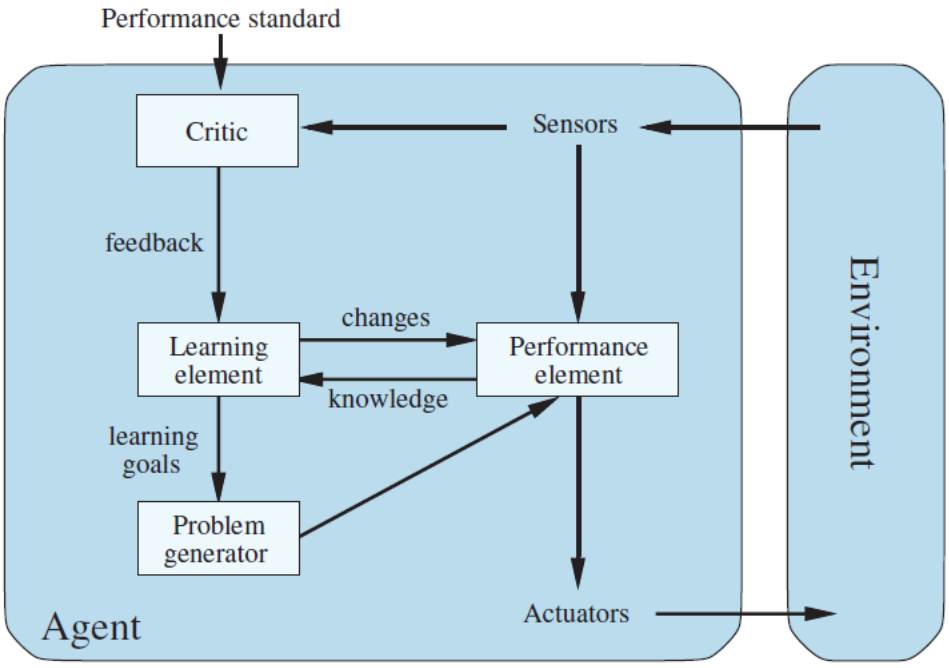
A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states,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

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

In two cases utility-based agents are more better than goal based agents
First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff.
Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning. A **model-free agent** can learn what action is best in a particular situation without ever learning exactly how that action changes the environment.

Learning agents
Any type of agent (model-based, goal-based, utility-based, etc.) can be built as a learning agent (or not).
Learning has another advantage it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge
A learning agent can be divided into four conceptual components, as shown in **Figure**



The **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is considered to be the entire agent: it takes in percepts and decides on actions. The learning element

uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

The design of the learning element depends very much on the design of the performance element. The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent's success. The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and informative experiences.

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

Working of the components of agent programs

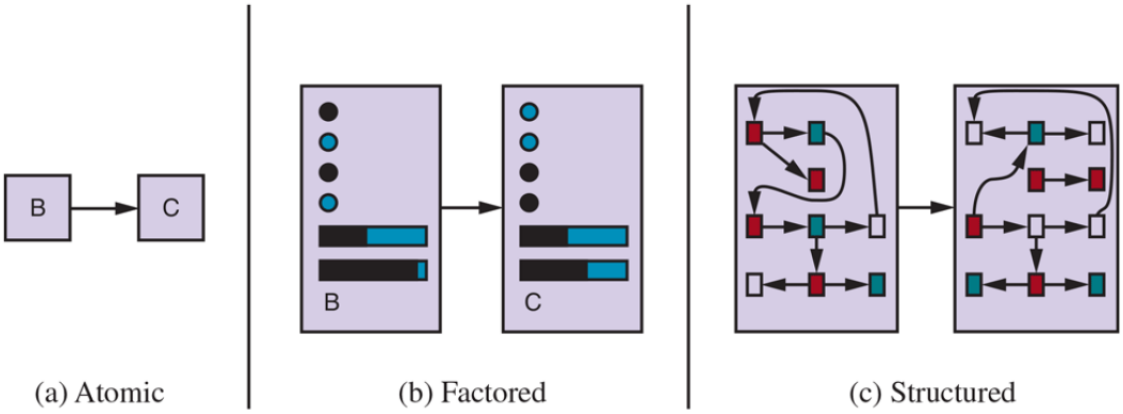
The three representations—atomic, factored, and structured.

In an **atomic representation** each state of the world is indivisible—it has no internal structure.

A **factored representation** splits up each state into a fixed set of **variables** or **attributes**, each of which can have a **value**.

Many important areas of AI are based on factored representations, including constraint satisfaction algorithms, propositional logic ,planning , Bayesian networks , and various machine learning algorithms.

Figure provides schematic depictions of how those transitions might be represented.



Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects. Structured representations underlie relational databases and first-order logic, first-order probability models , and much of natural language understanding .

Another axis for representation involves the mapping of concepts to locations in physical memory, whether in a computer or in a brain. If there is a one-to-one mapping between concepts and memory it is called **localist representation**. On the other hand, if the representation of a concept is spread over many memory locations, and each memory location is employed as part of the representation

of multiple different concepts ,that is called a **distributed representation**. Distributed representations are more robust against noise and information loss.

Problem Solving Agents

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

The agent can follow this four-phase problem-solving process:
GOAL FORMULATION: Goals organize behavior by limiting the objectives and hence the actions to be considered.

PROBLEM FORMULATION: The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world.

SEARCH: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution**. The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution (such as going from Arad to Sibiu to Fagaras to Bucharest), or it will find that no solution is possible.

EXECUTION: The agent can now execute the actions in the solution, one at a time.

It is an important property that in a fully observable, deterministic, known environment, *the solution to any problem is a fixed sequence of actions*. If the model is correct, then once the agent has found a solution, it can ignore its percepts while it is executing the actions—because the solution is guaranteed to lead to the goal. Control theorists call this an **open-loop** system: ignoring the percepts breaks the loop between agent and environment. If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent would be safer
using a **closed-loop** approach that monitors the percepts

Search problems and solutions

A search problem can be defined formally as follows:
A set of possible **states** that the environment can be in. This is the **state space**. The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions. The **initial state** that the agent starts in. A set of one or more **goal states**. Sometimes there is one goal state, sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states.The **actions** available to the agent. Given a state s ACTIONS(s) returns a finite set of actions that can be executed in s .
Example: ACTIONS(Arad) = {ToSibiu, ToTimisoara, ToZerind}.

A **transition model**, which describes what each action does. RESULT(s,a) returns the state that results from doing action in state

Example: RESULT(Arad, ToZerind) = Zerind.

An **action cost function**, denoted by ACTION-COST(s,a,s') gives the numeric cost of applying action a in state s to reach state s'. A problem-solving agent should use a cost function that reflects its own performance measure;
for example, for route-finding agents, the cost of an action might be the length in miles , or it might be the time it takes to complete the action.

A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state. An **optimal solution** has the lowest path cost among all solutions.

Formulating problems

A **model**—an abstract mathematical description—and not the real thing. The process of removing detail from a representation is called **abstraction**. A good problem formulation has the right level of detail. The abstraction is *valid* if it can elaborate any abstract solution into a solution in the more detailed world; The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem. 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.

Example Problems

The problem-solving approach has been applied to a vast array of task environments. A **standardized problem(toy problem)** is intended to illustrate or exercise various problem solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.
Examples: Grid world problem ,8-Queens,8-puzzle,Water Jug Problem, Crypt arithmetic
A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is unique not standardized, for example, each robot has different sensors that produce different data.
Examples: Route finding problem,Robot navigation, Touring problem, Travelling salesman

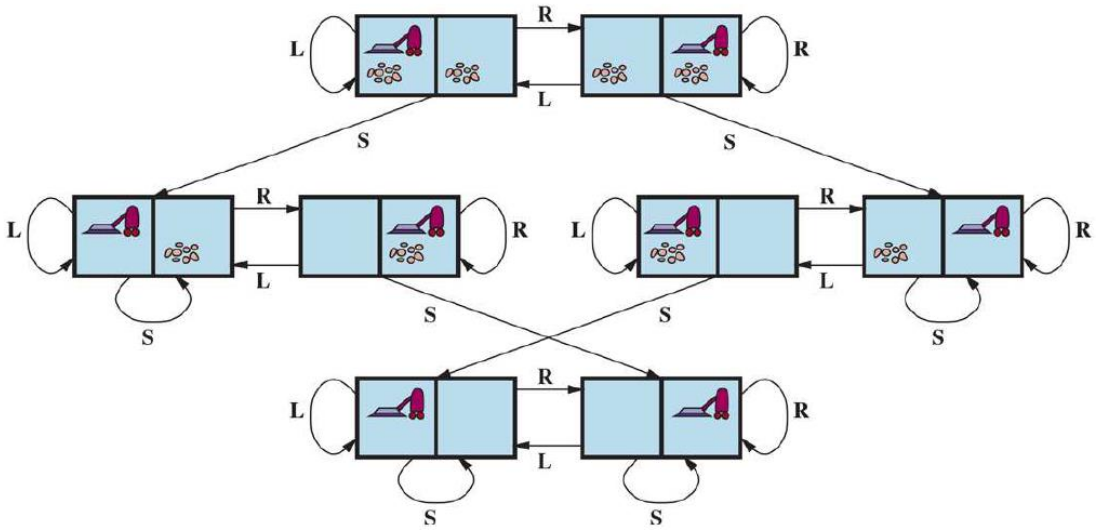
Standardized problem(Toy problem)
A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell—horizontally or vertically and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other strong obstacle in a cell prevents an agent from moving into that cell.

The **vacuum world** can be formulated as a grid world problem as follows:
STATES: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are **2.2.2=8states** . In general, a vacuum environment with n cells has **n. 2^n states.**
INITIAL STATE: Any state can be designated as the initial state.
ACTIONS: In the two-cell world three actions defined are: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world more movement actions are needed such as *Upward* and

Downward, giving the four **absolute** movement actions defined relative to the viewpoint of the agent

For example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.

Figure :The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: Left ,Right,Suc



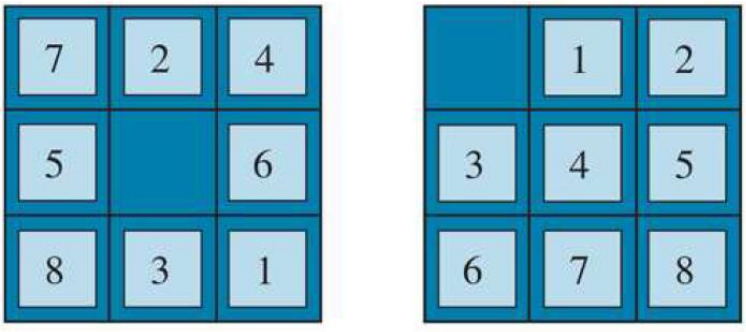
TRANSITION MODEL: *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90^o

GOAL STATES: The states in which every cell is clean.

ACTION COST: Each action costs 1.

Another type of grid world is the **sokoban puzzle**, in which the agent's goal is to push a number of boxes, scattered about the grid, to designated storage locations.

In a **sliding-tile puzzle**, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. the best-known variant is the **8- puzzle** , which consists of a grid with eight numbered tiles and one blank space, and the **15-puzzle** on a grid. The object is to reach a specified goal state, such as the one shown on the right of the figure.



A typical instance of the 8-puzzle.

The standard formulation of the 8 puzzle is as follows:
STATES: A state description specifies the location of each of the tiles.
INITIAL STATE: Any state can be designated as the initial state
ACTIONS: While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving *Left*, *Right*, *Up*, or *Down*. If the blank is at an edge or corner then not all actions will be applicable.
TRANSITION MODEL: Maps a state and action to a resulting state; for example, if the action is apply *Left* to the start state in Figure, the resulting state has 5 and the blank switched.
GOAL STATE: Although any state could be the goal, typically specify a state with the numbers in order.
ACTION COST: Each action costs 1.

One standardized problem devised by Donald Knuth illustrates how infinite state spaces can arise. Knuth conjectured that starting with the number 4, a sequence of square root, floor, and factorial operations can reach any desired positive integer
For example, 5 can be reached from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5.$$

The problem definition is simple:
STATES: Positive real numbers.
INITIAL STATE: 4.
ACTIONS: Apply square root, floor, or factorial operation (factorial for integers only).
TRANSITION MODEL: As given by the mathematical definitions of the operations.
GOAL STATE: The desired positive integer.
ACTION COST: Each action costs 1.
The state space for this problem is infinite: for any integer greater than 2 the factorial operator will always yield a larger integer. The problem is interesting because it explores very large numbers. Infinite state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

Real-world problems
Consider the **route-finding problem** defined in terms of specified locations and transitions along edges between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example. Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.

Consider the airline travel problems that must be solved by a travel-planning Web site:
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 extra information about these "historical" aspects.

INITIAL STATE: The user's home airport.
ACTIONS: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
TRANSITION MODEL: The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
GOAL STATE: A destination city. Sometimes the goal can be more complex, such as "arrive at the destination on a nonstop flight."
ACTION COST: A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Touring problems describe a set of locations that must be visited, rather than a single goal destination. The **traveling salesperson problem (TSP)** is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost < C (or in the optimization version, to find a tour with the lowest cost possible). An enormous amount of effort has been expended to improve the capabilities of TSP algorithms. The algorithms can also be extended to handle fleets of vehicles. In addition to planning trips, search algorithms have been used for tasks such as planning the movements of automatic circuit board drills and of stocking machines on shop floors.

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 footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

Robot navigation is a generalization of the route-finding problem described earlier. Rather than following distinct paths (such as the roads in Romania), a robot can roam around, in effect making its own paths. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs that must also be controlled, the search space becomes many-dimensional—one dimension for each joint angle. Advanced techniques are required just to make the essentially continuous search space finite . In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls, with partial observability, and with other agents that might alter the environment.

Automatic assembly sequencing of complex objects (such as electric motors) by a robot has been standard industry practice since the 1970s. Algorithms first find a feasible assembly sequence and then work to optimize the process. Minimizing the amount of manual human labor on the assembly line can produce significant savings in time and cost. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking an action 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. One 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.

Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure. Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem

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

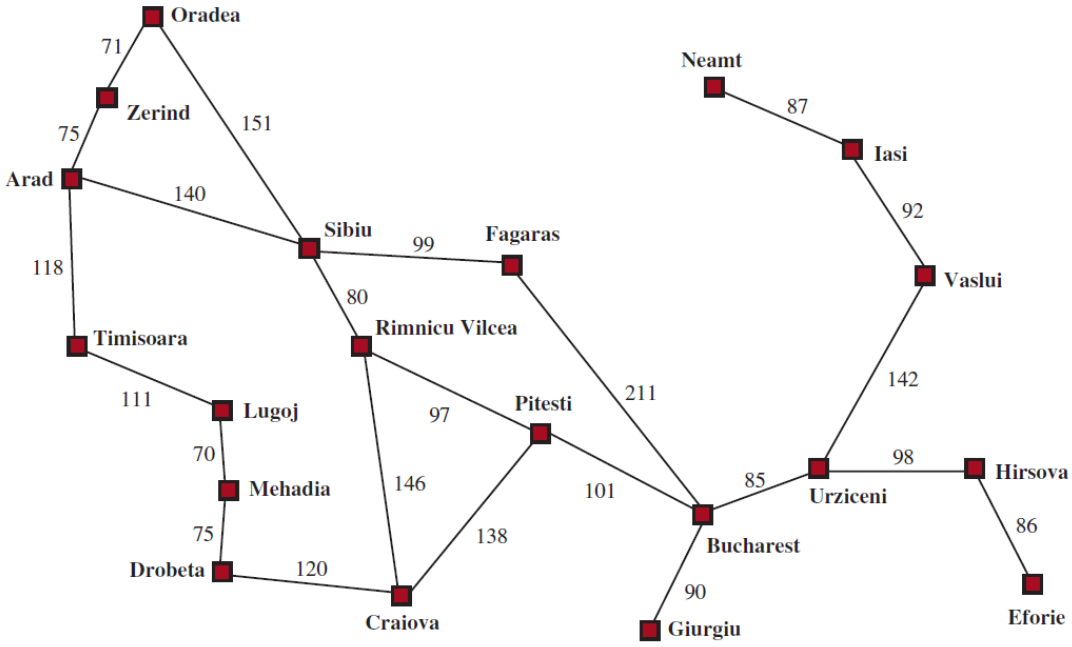


Figure :A simplified road map of a part of Romania with road distance in miles

Figure shows the first few steps in finding a path from Arad to Bucharest. The root node of the search tree is at the initial state, *Arad*. This node can be **expanded**, by considering the available ACTIONS for that state, using the RESULT function to see where those actions lead to, and **generating** a new node (called a **child node** or **successor node**) for each of the resulting states. Each child node has *Arad* as its **parent node**.

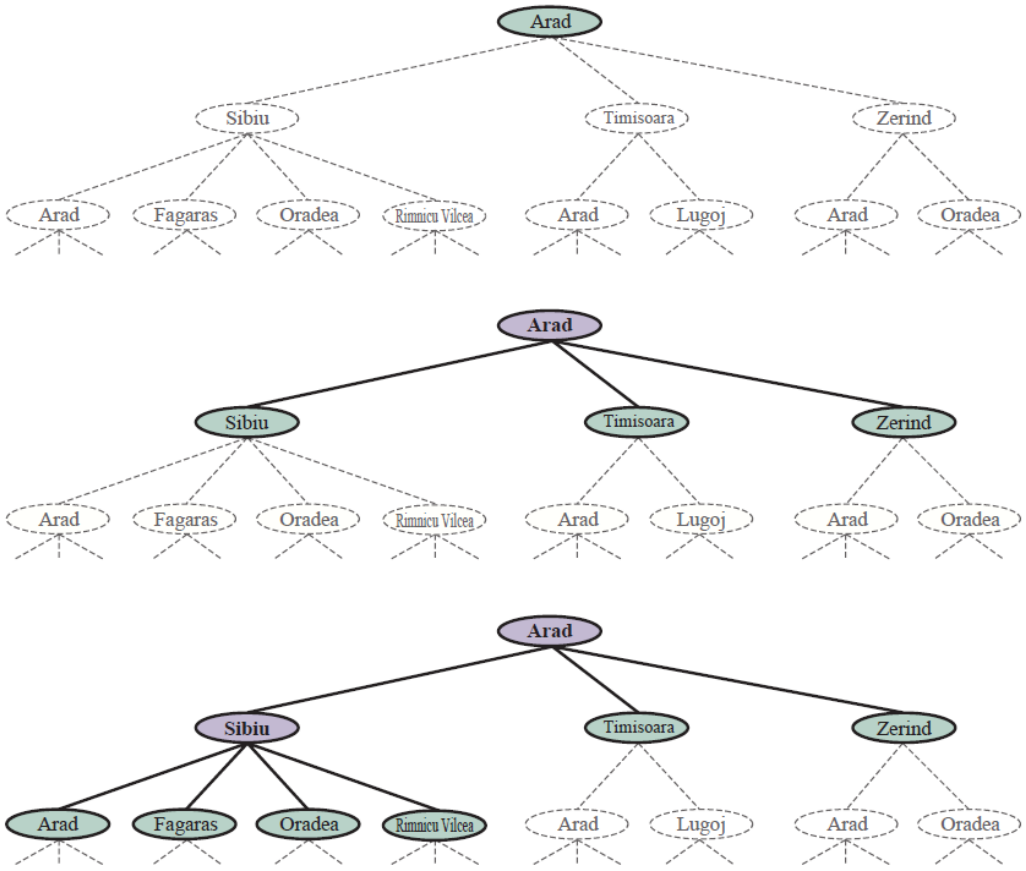
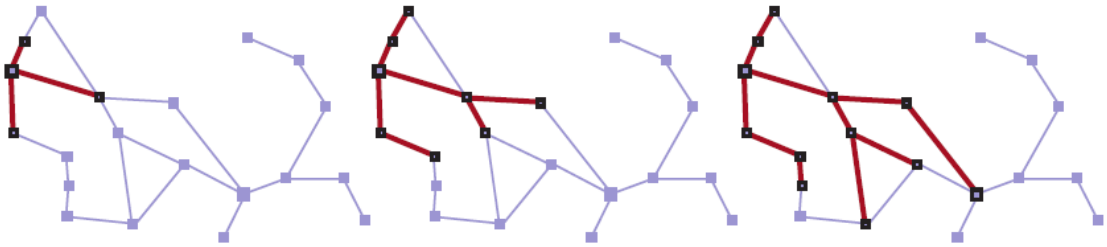


Figure shows the three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* ; nodes on the **frontier**(the set of all leaf nodes available for expansion)have been *generated*

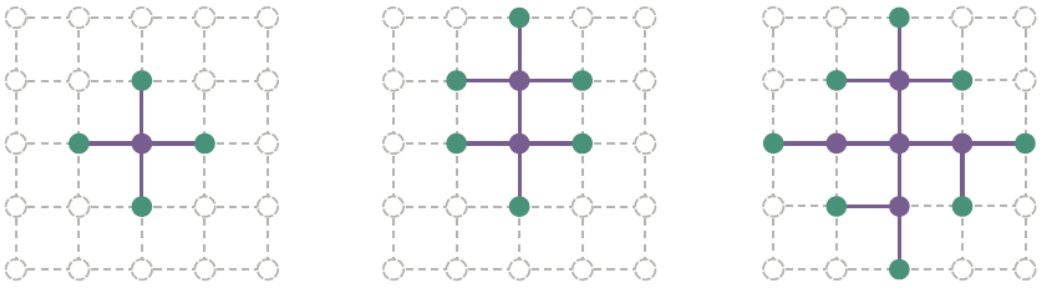
The set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. In the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

Figure below shows the sequence of search trees generated by a graph search on the Romania problem of Figure.



At each stage, every node on the frontier is expanded resulting in a state that has not been already reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

Note that the frontier separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached. This property is illustrated in Figure .



The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

Best-first search

A very general approach is called best-first search, in which a node n is chosen, with minimum value of some evaluation function f(n), Figure shows the algorithm.

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

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

On each iteration a node on the frontier with minimum value is chosen , return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path. The algorithm returns either an indication of failure, or a node that represents a path to a goal.

Search data structures

Search algorithms require a data structure to keep track of the search tree. A **node** in the tree is represented by a data structure with four components:
***node*.STATE:** the state to which the node corresponds;
***node*.PARENT:** the node in the tree that generated this node;
***node*.ACTION:** the action that was applied to the parent's state to generate this node;
***node*.PATH-COST:** the total cost of the path from the initial state to this node.

The operations on a frontier are:
IS-EMPTY(*frontier*) returns true only if there are no nodes in the frontier.
POP(*frontier*) removes the top node from the frontier and returns it.
TOP(*frontier*) returns (but does not remove) the top node of the frontier.
ADD(*node, frontier*) inserts node into its proper place in the queue.

Three kinds of queues are used in search algorithms:
A **priority queue** first pops the node with the minimum cost according to some evaluation function, f. It is used in best-first search.

A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; it is used in breadth-first search.

A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node,it is used in depth-first search.

The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

Redundant paths

A **repeated state** in the search tree, generated a **cycle** (also known as a **loopy path**). A cycle is a special case of a **redundant path**.

Measuring problem-solving performance

An algorithm's performance can be evaluated in four ways:

COMPLETENESS: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
COST OPTIMALITY: Does it find a solution with the lowest path cost of all solutions?

TIME COMPLEXITY: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
SPACE COMPLEXITY: How much memory is needed to perform the search?
To be complete, a search algorithm must be **systematic** in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state.

Time and space complexity are 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 number of vertices (state nodes) of the graph and |E| is the number of edges (distinct state/action pairs).

But in many AI problems, the graph is represented only *implicitly* by the initial state, actions, and transition model. For an implicit state space, complexity can be measured in terms of the d,**depth** or number of actions in an optimal solution; m,the maximum number of actions in any path; and b, the **branching factor** or number of successors of a node that need to be considered.

Uninformed Search Strategies(Blind Search)

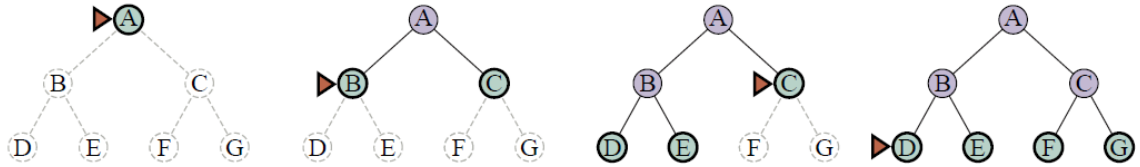
The term blind means that the strategies have no additional information about states beyond that is provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded. Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic search strategies

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

BREADTH FIRST SEARCH

- When all actions have the same cost, an appropriate strategy is breadth-first search, in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- This can be implemented as a call to BEST-FIRST-SEARCH where the evaluation function f(n) is the depth of the node—that is, the number of actions it takes to reach the node.
- A first-in-first-out queue is used, this gives the correct order of nodes: 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.
- A reached state can be a set of states rather than a mapping from states to nodes, because once a state is reached, a better path to the state can never be found.
- In addition an **early goal test** can be done to check whether a node is a solution as soon as it is *generated*, rather than the **late goal test** that best-first search uses, waiting until a node is popped off the queue.

Figure shows the progress of a breadth-first search on a binary tree,



Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Figure : Algorithm

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

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

Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth d it has already generated all the nodes at depth d-1. That means it is cost-optimal for problems where all actions have the same cost, but not for problems that don't have that property.

In terms of time and space, consider searching a uniform tree where every state has b successors. The root of the search tree generates b nodes, each of which generates b more nodes, for a total b^2 of at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d Then the total number of nodes generated is

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

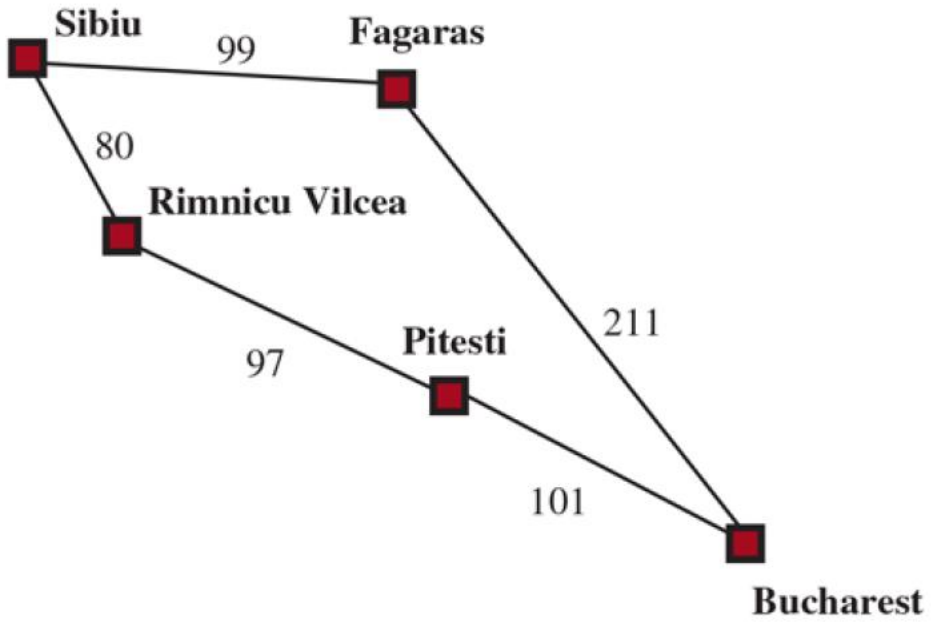
All the nodes remain in memory, so both time and space complexity are **$O(b^d)$.**

Dijkstra's algorithm or uniform-cost search

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's

algorithm by the theoretical computer science community, and **uniform-cost search** by the AI community.
The idea is that while breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to BEST FIRSTSEARCH with PATH-COST as the evaluation function.

Consider the example , where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80+97=177. The least cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310.Bucharest is the goal, but the algorithm tests for goals only when it expands a node, not when it generates a node, so it has not yet detected that this is a path to the goal.



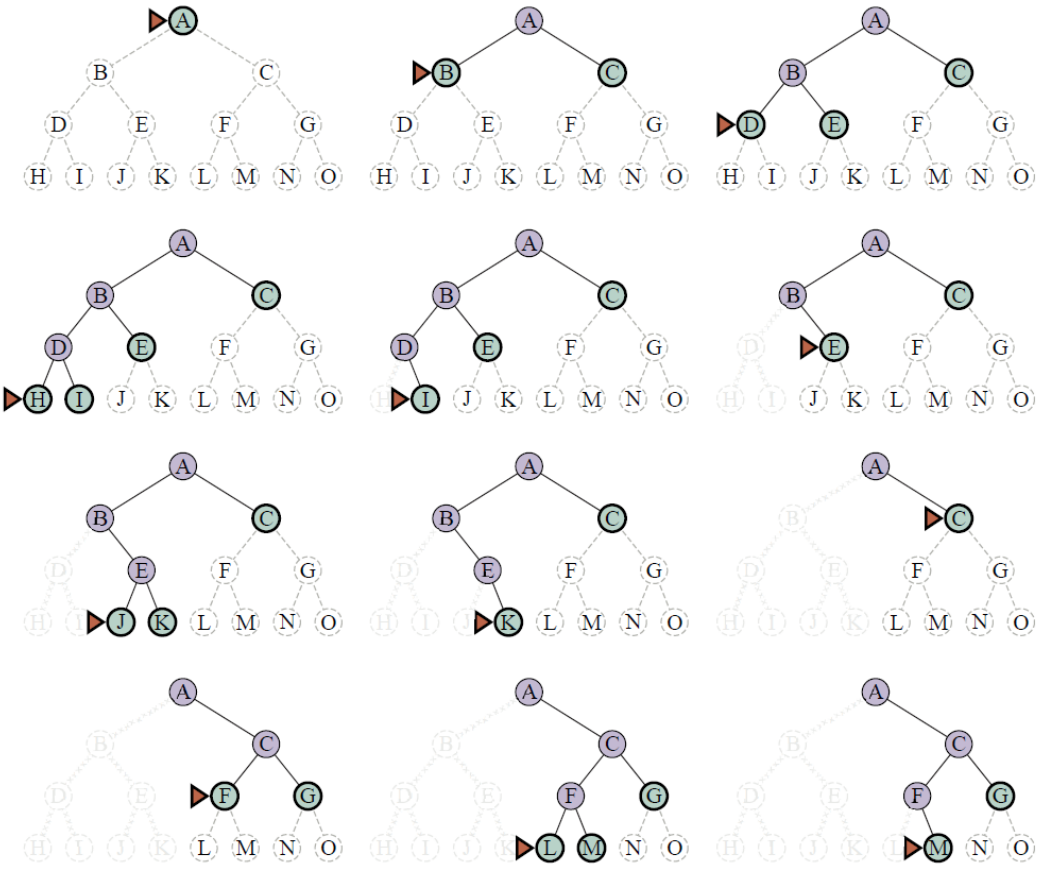
The algorithm continues on, choosing Pitesti for expansion next and adding a second path to Bucharest with cost 80 + 97 + 101 = 278.It has a lower cost, so it replaces the previous path in reached and is added to the frontier. It turns out this node now has the lowest cost, so it is considered next, found to be a goal, and returned. Note that if we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, then we would have returned a higher-cost path (the one through Fagaras).
The complexity of uniform-cost search is characterized in terms of C*, the cost of the optimal solution, and a lower bound ϵ, on the cost of each action, with $\epsilon>0$ Then the algorithm's worst-case time and space complexity $\mathbf{O(b^{1+\lfloor C*/\epsilon \rfloor})}$ is which can be much greater than b^d This is because uniform-cost search can explore large trees of actions with low costs before exploring paths

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

Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are $> \epsilon > 0$).

Depth-first search and the problem of memory

Depth-first search always expands the *deepest* node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. The progress of the search is illustrated in **Figure ;**



Search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then "backs up" to the next deepest node that still has unexpanded successors. Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.

For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space.

In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles. Finally, in infinite state spaces, depth-first search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, depth-first search is incomplete.

For a finite tree-shaped state-space , a depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only O(bm), where b is the branching factor and m is the maximum depth of the tree.
A variant of depth-first search called backtracking search uses even less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In addition, successors are generated by modifying the current state description directly rather than allocating memory for a brand-new state. This reduces the memory requirements to just one state description and a path of O(m)actions; a significant savings over O(bm) states for depth-first search. With backtracking there is an option of maintaining an efficient set data structure for the states on the current path, allowing to check for a cyclic path in O(1) time rather than O(m). Backtracking is critical to the success of many problems with large state descriptions, such as robotic assembly.

Depth-limited and iterative deepening search

To keep depth-first search from wandering down an infinite path, **depth-limited search can be used**, a version of depth-first search in which a depth limit l is provided , and treat all nodes at depth l as if they had no successors . The time complexity is $O(b^l)$ and the space complexity is O(bl) .Unfortunately, if a poor choice is made for l, the algorithm will fail to reach the solution, making it incomplete again.

Algorithm:Iterative deepening and depth-limited tree-like search.

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

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

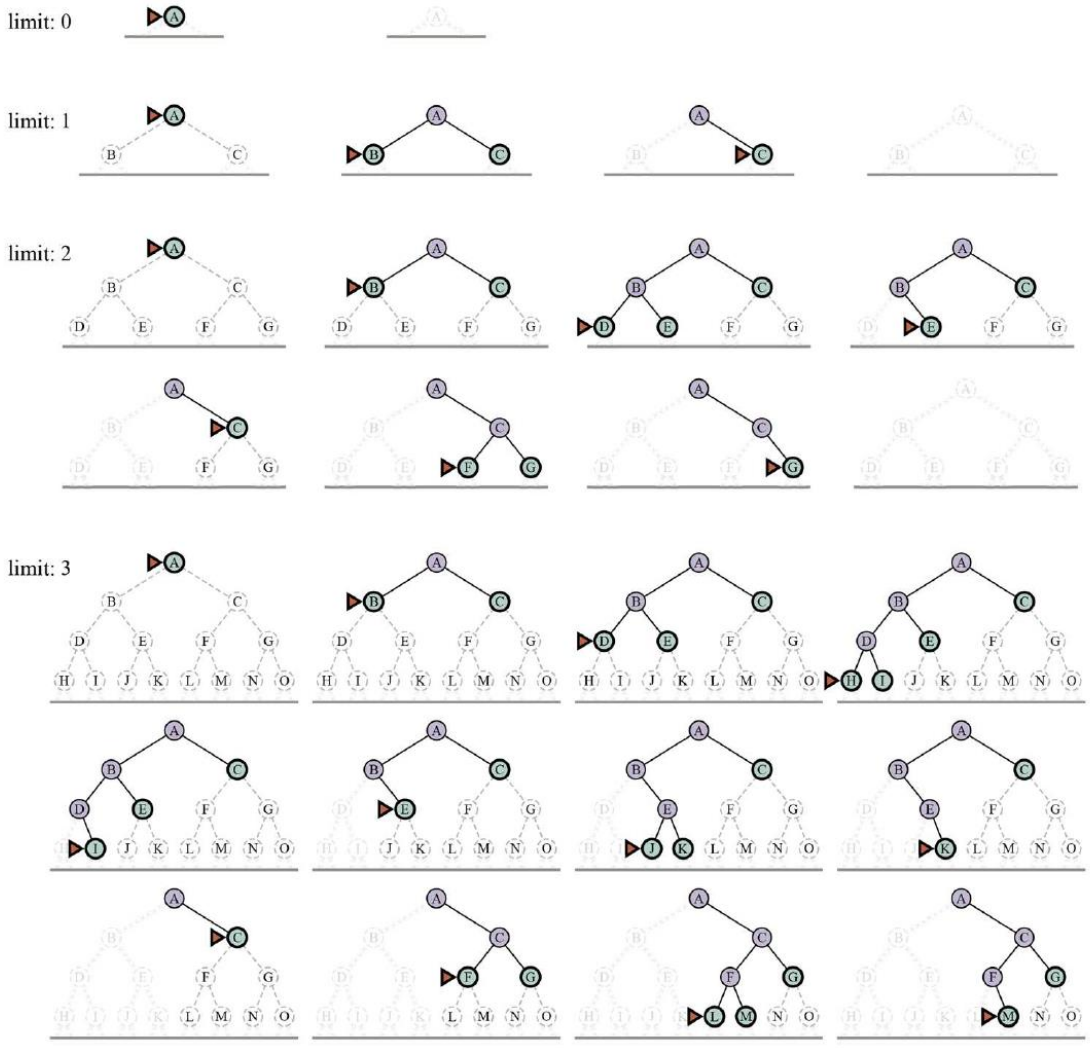
Iterative deepening repeatedly applies depth limited search with increasing limits. It returns one of three different types of values: either a solution node; or failure, when it has exhausted all nodes and proved there is no solution at any depth; or cutoff, to mean there might be a solution at a deeper depth than l. This is a tree-like search algorithm that does not keep track of reached states, and thus

uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check all cycles, then the algorithm may get caught in a loop.

Iterative deepening search solves the problem of picking a good value for l by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth limited search returns the failure value rather than the cutoff value . Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest:O(bd) when there is a solution, or O(bm) on finite state spaces with no solution. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.

The time complexity is $O(b^d)$when there is a solution, or $O(b^m)$when there is none. Each iteration of iterative deepening search generates a new level, in the same way that breadthfirst search does, but breadth-first does this by storing all nodes in memory, while iterativedeepening does it by repeating the previous levels, thereby saving memory at the cost of more time.

Figure shows four iterations of iterative-deepening search on a binary search tree, where the solution is found on the fourth iteration.



Iterative deepening search may seem wasteful because states near the top of the search tree are re-generated multiple times. But for many state spaces, most of the nodes are in the bottom level, so it does not matter much that the upper levels are repeated. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is $N(IDS) = (d)b^1 + (d - 1)b^2 + (d - 2)b^3 \ldots + b^d$, which gives a time complexity of $O(b^d)$—asymptotically the same as breadth-first search.

For example, if b=10 and d=5 the numbers are

N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110.

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

Bidirectional search

An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. For this to work, two frontiers and two tables of reached states, and reason backwards is needed. If state s' is a successor of s in the forward direction, then s is a successor of s' in the backward direction. Thus when the two frontiers collide there is a solution.

The general best-first bidirectional search algorithm is shown in Figure

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms. Here bidirectional best-first search is considered. Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier. When the evaluation function is the path cost, bidirectional uniform-cost search is got, and if the cost of the optimal path is C* then no node with cost >C*/2 will be expanded. This can result in a considerable speedup.

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

Comparing uninformed search algorithms

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

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