ArtificialIntelligence

Ziniu Yu

Sep 14, 2021

CONTENTS:

Study notes of Artificail Intelligence A Modern Approach, 4th Edition by Stuart Russell and Peter Norvig

CHAPTER

ONE

PART 1 ARTIFICIAL INTELLIGENCE

1.1 Chapter 1 Introduction

1.1.1 1.1 What Is AI?

Some definitions of artificial intelligence:

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

1.1.1 Acting humanly: The Turing test approach

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

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

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

- **computer vision** and speech recognition to perceive the world

- **robotics** to manipulate objects and move about.

1.1.2 Thinking humanly: The cognitive modeling approach

We can learn about human thought in three ways:

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

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

1.1.3 Thinking rationally: The "laws of thought" approach

Several Greek schools developed various forms of **logic**, which provides a precise notation for statements bout objects in the world and the relations among them.

1.1.4 Acting rationally: The rational agent approach

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

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

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

1.1.5 Beneficial machines

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

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

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

1.1.2 1.2 The Foundations of Artificial Intelligence

1.2.1 Philosophy

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

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

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

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

The **logical positivism** holds that all knowledge can be characterized by logical theories connected, ultimately, to **observation sentences** that correspond to sensory inputs; thus logical positivism combines rationalism and empiricism.

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

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

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

1.2.2 Mathematics

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

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

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

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

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

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

1.2.3 Economics

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

Decision theory, which combines probability theory with utility theory, provides a formal and complete framework for individual decisions (economic or otherwise) made under uncertainty—that is, in cases where probabilistic descriptions appropriately capture the decision maker's environment.

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

1.2.4 Neuroscience

- How do brains process information?

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

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

1.2.5 Psychology

- How do humans and animals think and act?

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

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

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

1.2.6 Computer engineering

- How can we build an efficient computer?

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

1.2.7 Control theory and cybernetics

- How can artifacts operate under their own control?

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

1.2.8 Linguistics

- How does language relate to thought?

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

1.1.3 1.3 The History of Artificial Intelligence

1.3.1 The inception of artificial intelligence (1943-1956)

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

1.3.2 Early enthusiasm, great expectations (1952-1969)

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

1.3.3 A does of reality (1966-1973)

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

1.3.4 Expert systems (1969-1986)

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

1.3.5 The return of neural networks (1986-present)

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

1.3.6 Probabilistic reasoning and machine learning (1987-present)

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

1.3.7 Big data (2001-present)

Remarkable advances in computing power and the creation of the World Wide Web have facilitated the creation of very large data sets—a phenomenon sometimes known as **big data**.

1.3.8 Deep learning (2011-present)

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

1.1.4 1.4 The Sate of the Art

ROBOTIC VEHICLES

LEGGED LOCOMOTION

AUTONOMOUS PLANNING AND SCHEDULING

MACHINE TRANSLATION

SPEECH RECOGNITION

RECOMMENDATIONS

GAME PLAYING

IMAGE UNDERSTANDING

MEDICINE

CLIMATE SCIENCE

1.1.5 1.5 Risks and Benefits of AI

LETHAL AUTONOMOUS WEAPONS

SURVEILLANCE AND PERSUASION

BIASED DECISION MAKING

IMPACT ON EMPLOYMENT

SAFETY-CRITICAL APPLICATIONS

CYBERSECURITY

1.2 Chapter 2 Intelligent Agents

1.2.1 2.1 Agents and Environments

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

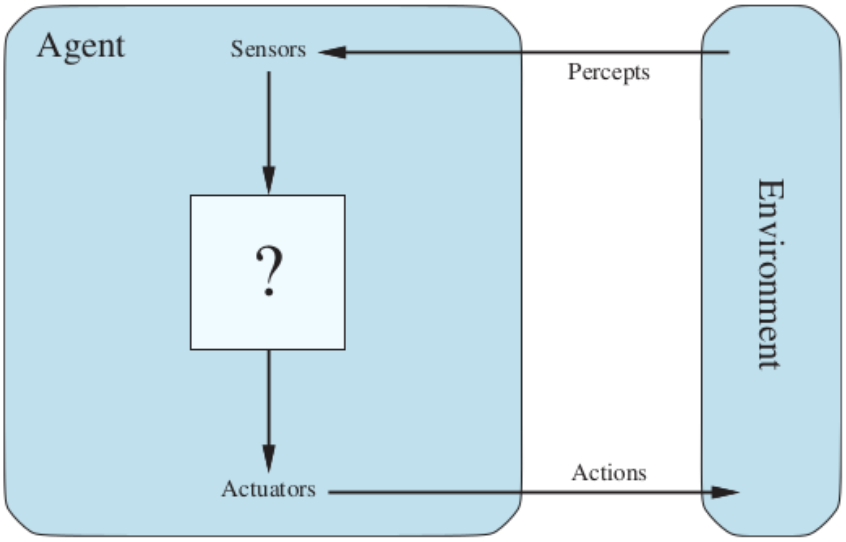

Figure 2.1 Agents interact with environments through sensors and actuators.

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

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

1.2.2 2.2 Good Behavior: The Concept of Rationality

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

2.2.1 Performance measures

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

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

2.2.2 Rationality

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

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

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

2.2.3 Omniscience, learning, and autonomy

We need to be careful to distinguish between rationality and **omniscience**. An omniscient agent knows the *actual* 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**. Our definition requires a rational agent not only to gather information but also to **learn** as much as possible from what it perceives.

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

1.2.3 2.3 The Nature of Environments

2.3.1 Specifying the task environment

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

2.3.2 Properties of task environments

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

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

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

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

STATIC VS. DYNAMIC: If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**.

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

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

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

1.2.4 2.4 The Structure of Agents

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

agent = *architecture* + *program*.

2.4.1 Agent programs

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

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

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

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

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

2.4.2 Simple reflex Agents

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

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

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

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

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

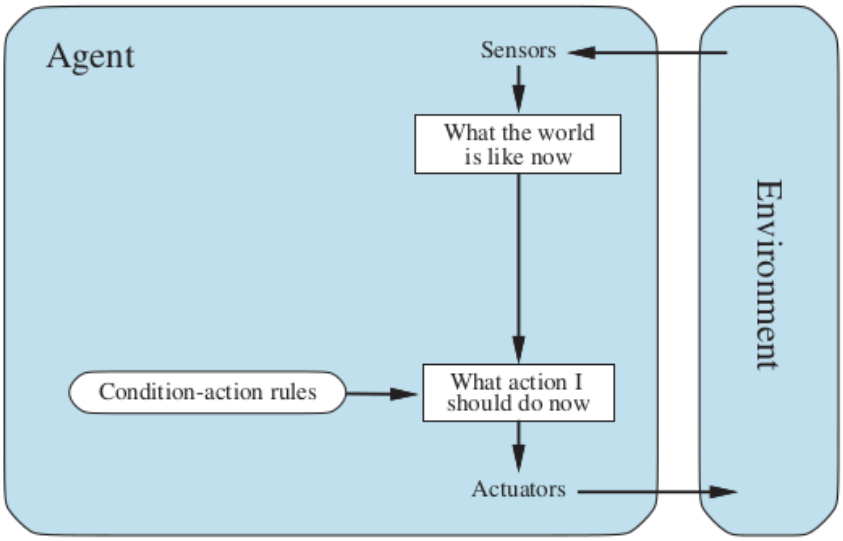

Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

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

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

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

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

2.4.3 Model-based reflex agents

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

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

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

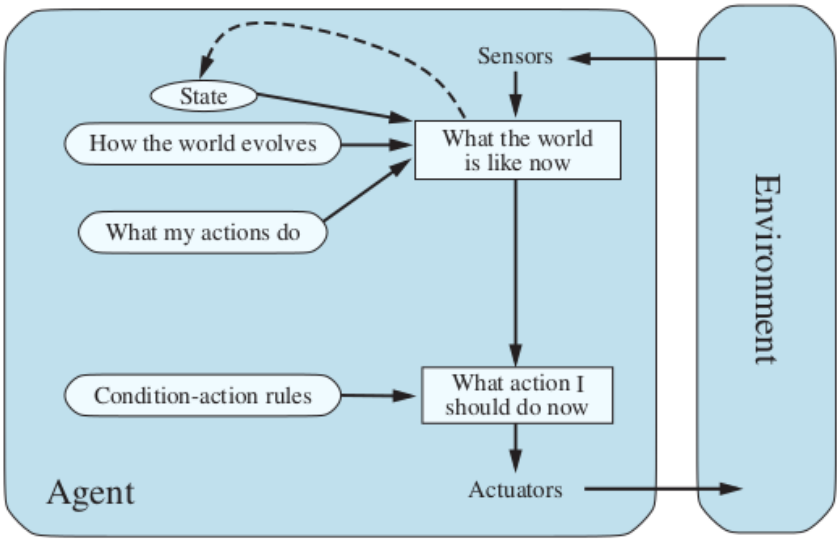

Figure 2.11 A model-based reflex agent.

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

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

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

2.4.4 Goal-based agents

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

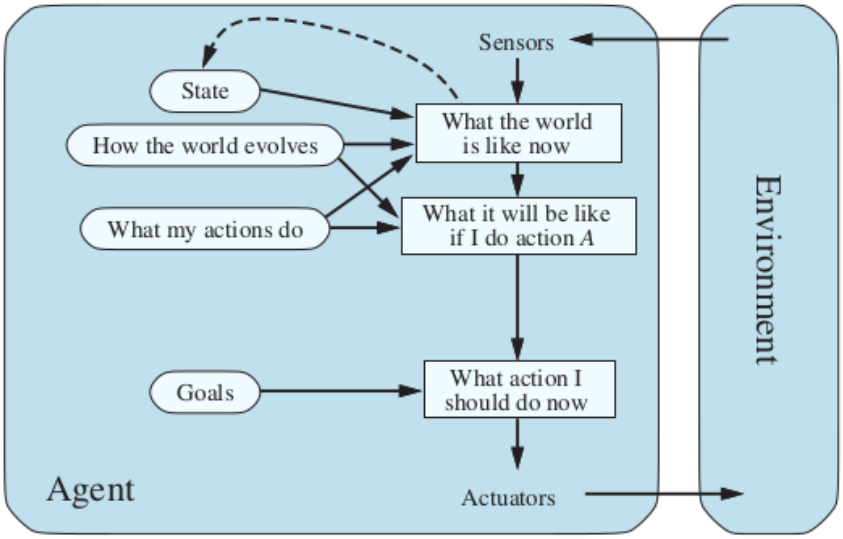

Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

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

2.4.5 Utility-based agents

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

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

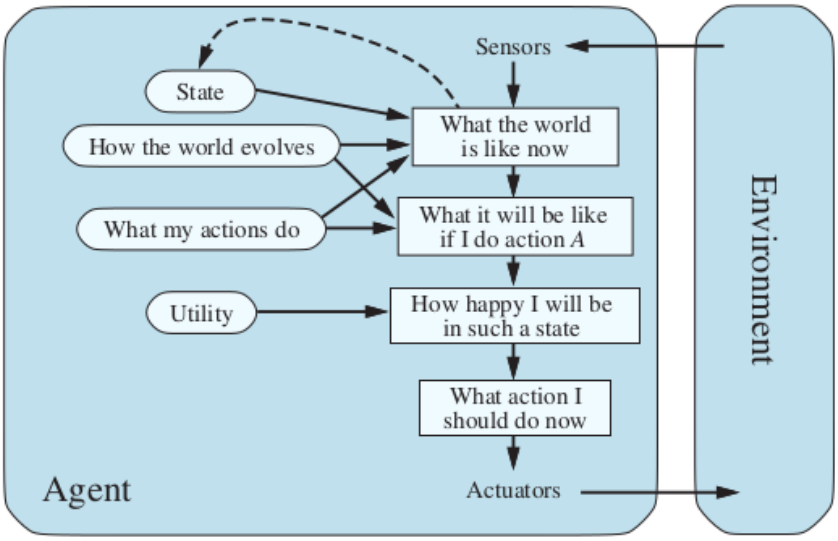

Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome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

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

2.4.6 Learning agents

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

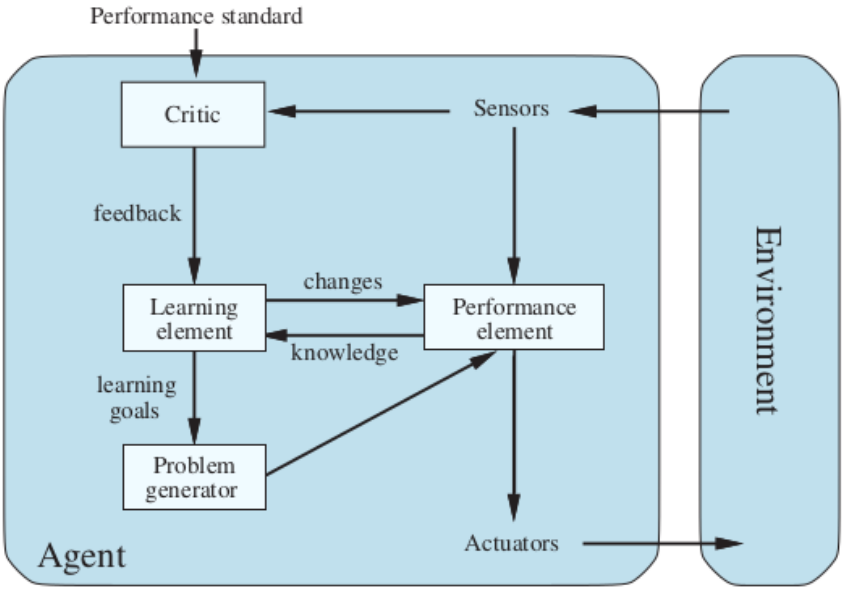

Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

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

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

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

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

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

2.4.7 How the components of agent programs work

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

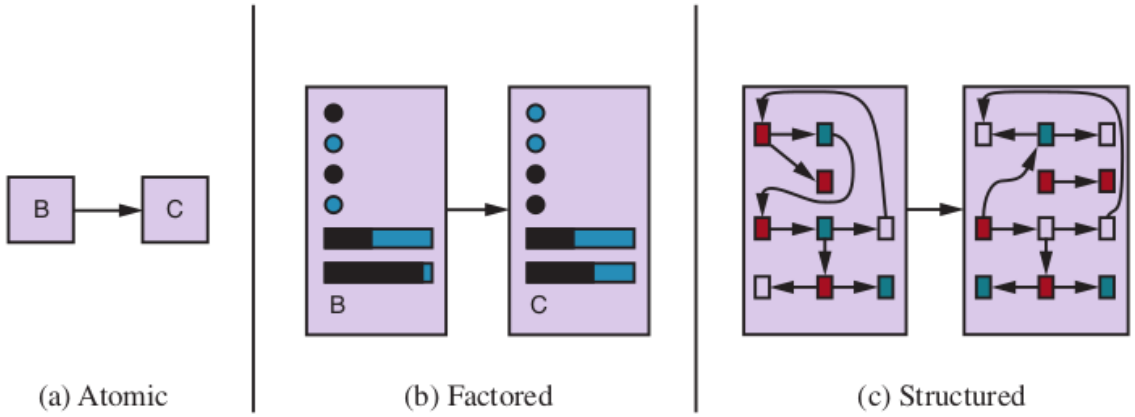

Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed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

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

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

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

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

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

representation. Distributed representations are more robust against noise and information loss.

CHAPTER

TWO

PART 2 PROBLEM-SOLVING

2.1 Chapter 3 Solving Problems by Searching

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

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

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

2.1.1 3.1 Problem-Solving Agents

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

- **GOAL FORMULATION**: Goals organize behavior by limiting the objectives and hence the actions to be 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**.
- **EXECUTION**: The agent can now execute the actions in the solution, one at a time.

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

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

3.1.1 Search problems and solutions

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

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

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

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

3.1.2 Formulating problems

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

2.1.2 3.2 Example Problems

A **standardized 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. A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

3.2.1 Standardized problems

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

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

3.2.2 Real-world problems

- **Route-finding problem**
- **Touring problems**
- **Trveling salesperson problem (TSP)**
- **VLSI layout problem**
- **Robot navigation**
- **Automatic assembly sequencing**

2.1.3 3.3 Search Algorithms

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

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

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

3.3.1 Best-first search

In **best-first search** we choose a node, n, with minimum value of some **evaluation function**, $f(n)$.

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

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

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

3.3.2 Search data structures

A **node** in the tree is represented by a data structure with four components

- $node.State$: the state to which the node corresponds;
- $node.Parent$: 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;
- $node.PathCost$: the total cost of the path from the initial state to this node. In mathematical formulas, we use $g(node)$ as a synonym for $PathCost$.

Following the $PARENT$ pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution.

We need a data structure to store the **frontier**. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:

- $IsEmpty(frontier)$ returns true only if there are no nodes in the frontier.
- $Pop(frontier)$ 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; we shall see it is used in breadth-first search.
- A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node; we shall see it is used in depth-first search.

3.3.3 Redundant paths

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

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

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

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

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

3.3.4 Measuring problem-solving performance

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

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

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

2.1.4 3.4 Uninformed Search Strategies

3.4.1 Breadth-first search

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

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

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

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

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

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

3.4.2 Dijkstra's algorithm or uniform-cost search

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

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

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

3.4.3 Depth-first search and the problem of memory

Depth-first search always expands the *deepest* node in the frontier first. It could be implemented as a call to $Best\text{-}FirstSearch$ where the evaluation function f is the negative of the depth.

For problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. 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.

3.4.4 Depth-limited and iterative deepening search

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

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

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

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

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

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

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

3.4.5 Bidirectional search

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

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

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

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

3.4.6 Comparing uninformed search algorithms

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

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

2.1.5 3.5 Informed (Heuristic) Search Strategies

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

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

3.5.1 Greedy best-first search

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

2.2 Chapter 4 Search in Complex Environments

2.3 Chapter 5

2.4 Chapter 6