GRUNDZÜGER DER ARTIFICIAL INTELLIGENCE

1. Chapter 1 (Introduction)

1.1. What is AI?

The definition of Artificial Intelligence can be grouped in two dimensions: *Thinking* and *Acting.*

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 .. ." (Hellman, 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 at,* 1998) "Al ... is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

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

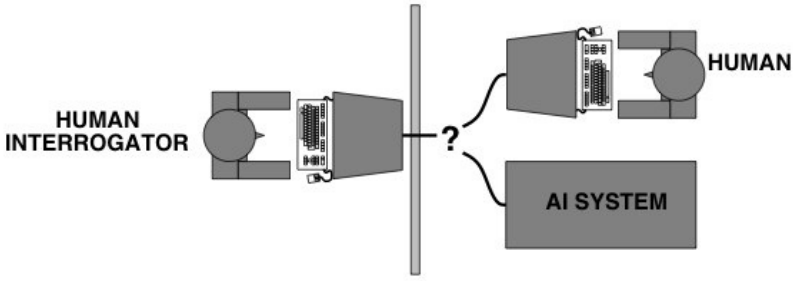
In Figure 1.1 we see eight definitions of AI, laid out along two dimensions. The definitions on top are concerned with thought processes and reasoning, whereas the ones on the bottom address behavior. The definitions on the left measure success in terms of fidelity to human performance, whereas the ones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

1.1. and 2.1. check how good the human performance can be imitated by the AI. 1.2 and 2.2 check if a system is rational (System is rational if it makes the right choices according to its knowledge).

1.2. Describe the four approaches to AI

Acting humanly: The Turing Test approach

The Turing Test, 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.



For now, we note that programming a computer to pass a rigorously applied test provides plenty to work on (natural language processing, knowledge representation, automated reasoning, machine learning)

By distinguishing between human and rational behavior, we are not suggesting that humans are necessarily "irrational - in the sense of - emotionally unstable" or "insane. One merely need note that we are not perfect: not all chess players are grandmasters; and, unfortunately, not everyone gets an A on the exam. Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because physical simulation of a person is unnecessary for intelligence. However, the so-called total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the total Turing Test, the computer will need computer vision to perceive objects, and robotics to manipulate objects and move about.

These six disciplines compose most of Al, and Turing deserves credit for designing a test that remains relevant 60 years later. Yet Al researchers have devoted little effort to passing the Turing Test, believing that it is more important to study the underlying principles of intelligence than to duplicate an exemplar. The quest for "artificial flight" succeeded when the Wright brothers and others stopped imitating birds and started using wind tunnels and learning about aerodynamics. Aeronautical engineering texts do not define the goal of their field as making "machines that fly so exactly like pigeons that they can fool even other pigeons."

Problem: Turing test is not reproducible, constructive, or amenable to mathematical analysis.

Thinking humanly: The cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get inside the actual workings of human minds.

There are three ways to do this:

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

2. through psychological experiments — observing a person in action;

3. and through brain imaging — observing the brain in action.

Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input—output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans. For example, Allen Newell and Herbert Simon were not concerned merely to have their program solve problems correctly but with comparing the trace of its reasoning steps to traces of human subjects solving the same problems.

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. Cognitive science is necessarily based on experimental investigation of actual humans or animals.

In the early days of AI there was often confusion between the approaches: an author would argue that an algorithm performs well on a task and that it is therefore a good model of human performance, or vice versa. Modem authors separate the two kinds of claims; this distinction has allowed both AI and cognitive science to develop more rapidly. The two fields continue to fertilize each other, must notably in computer vision, which incorporates neurophysiological evidence into computational models.

Both approaches (roughly, Cognitive Science and Cognitive Neuroscience) are now distinct from AI. Both share with AI the following characteristic: the available theories do not explain (or engender) anything resembling human-level general intelligence.

Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is, irrefutable reasoning processes. His syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example, "Socrates is a man; all men are mortal; therefore, Socrates is mortal." These laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic.

Logicians in the 19th century developed a precise notation fur statements about all kinds of objects in the world and the relations among them. By 1965, programs existed that could, in principle, solve any solvable problem described in logical notation. (Although if no solution exists, the program might loop forever.) The so-called logicist tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain (uncertainty). 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. Although both of these obstacles apply to any attempt to build computational reasoning systems, they appeared first in the logicist tradition.

Normative (or prescriptive) rather than descriptive.

Problems:

1. Not all intelligent behavior is mediated by logical deliberation

2. What is the purpose of thinking? What thoughts should I have out of all the thoughts (logical or otherwise) that I could have?

Acting rationally: The rational agent approach

An agent is just something that acts (agent comes from the Latin agere, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate

autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

In the "laws of thought" approach to AI, the emphasis was on correct inferences (Schlussfolgerungen, create new knowledge by combining known knowledge). Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals and then to act on that conclusion.

On the other hand, correct inference is not all of rationality; in some situations, there is no provably correct thing to do, but something must still be done. There are also ways of acting rationally that cannot be said to involve inference. For example, recoiling from a hot stove is a reflex action that is usually more successful than a slower action taken after careful deliberation.

All the skills needed for the Turing Test also allow an agent to act rationally, Knowledge representation and reasoning enable agents to reach good decisions. We need to be able to generate comprehensible sentences in natural language to get by in a complex society. We need learning not only for erudition, but also because it improves our ability to generate effective behavior.

The rational-agent approach 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 more amenable to scientific development than are approaches based on human behavior or human thought be- cause the standard of rationality is clearly defined and completely general. Human behavior, on the other hand, is well-adapted for one specific environment and is the product, in part, of a complicated and largely unknown evolutionary process that still is far from producing perfection

1.3. What can AI do today?

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

Robotic vehicles, Speech recognition, Games (IBM program beats chessmaster, 1997), Autonomous planning and scheduling, ...

State of the art

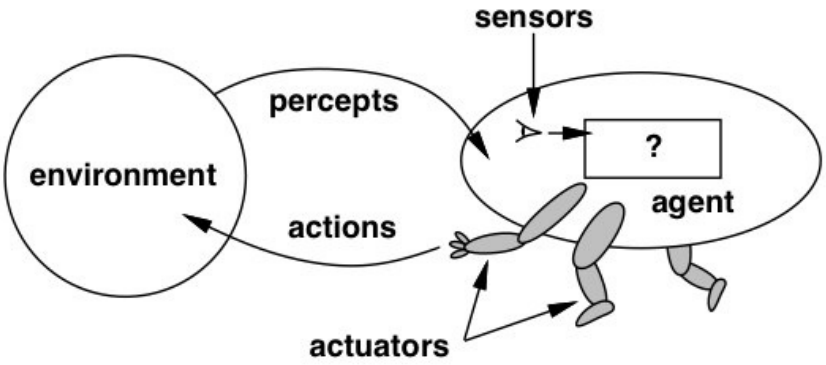
Which of the following can be done at present?

- ♢ Play a decent game of table tennis
- ♢ Drive safely along a curving mountain road
- ♢ Drive safely along Telegraph Avenue
- ♢ Buy a week's worth of groceries on the web
- ♢ Buy a week's worth of groceries at Berkeley Bowl
- ♢ Play a decent game of bridge
- ♢ Discover and prove a new mathematical theorem
- ♢ Design and execute a research program in molecular biology
- ♢ Write an intentionally funny story
- ♢ Give competent legal advice in a specialized area of law
- ♢ Translate spoken English into spoken Swedish in real time
- ♢ Converse successfully with another person for an hour
- ♢ Perform a complex surgical operation
- ♢ Unload any dishwasher and put everything away

2. Chapter 2 (Intelligent Agents)

2.1. What is an Agent

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



Agents include humans, robots, softbots, thermostats, etc. **Sensors** for perceiving the "world" (They produce perception sequeneces). **Actuators** for acting. The **agent function** maps from percept histories to actions: *f: P* → A* The **agent program** runs on the physical architecture to produce the agent function f.

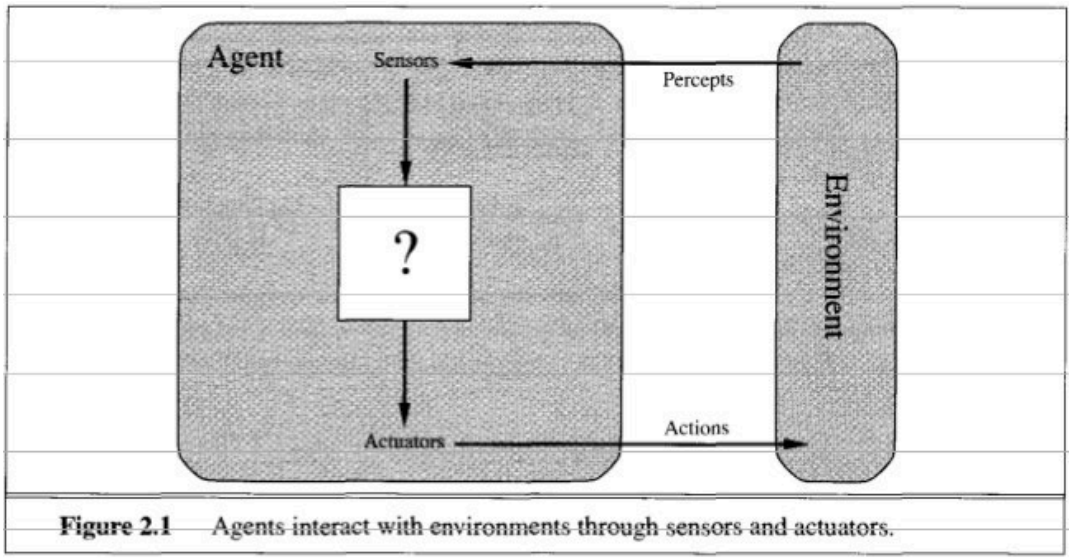


Figure 2.1 Agents interact with environments through sensors and actuators.

2.2. What is a percept sequence (Wahrnehmungsfolge)?

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 the entire percept sequence observed to date, but not on anything it hasn't perceived.

2.3. Agent function vs. agent program

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**.

2.4. What is a rational Agent

A **rational agent** is one that does the right thing—conceptually speaking, every entry in the table for the agent function is filled out correctly.

How to find out what is the right thing? Answere: by considering the consequences of the agent's behavior. 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.

Notice that we said environment states, not agent states. If we define success in terms of agent's opinion of its own performance, an agent could achieve perfect rationality simply by deluding itself that its performance was perfect. Human agents in particular are notorious for "sour grapes"—believing they did not really want something (e.g., a Nobel Prize) after not getting it.

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.

2.5. Performance measure

Obviously, there is not one fixed performance measure for all tasks and agents; typically, a designer will devise one appropriate to the circumstances. This is not as easy as it sounds. Consider, for example, the vacuum-cleaner agent. We might propose to measure performance by the amount of dirt cleaned up in a single eight-hour shift. With a rational agent, of course, what you ask for is what you get. A rational agent can maximize this performance measure by cleaning up the dirt, then dumping it all on the floor, then cleaning it up again, and so on. A more suitable performance measure would reward the agent for having a clean floor. For example, one point could be awarded for each clean square at each time step (perhaps with a penalty for electricity consumed and noise generated). As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Even when the obvious pitfalls are avoided, there remain some knotty issues to untangle. For example, the notion of "clean floor" in the preceding paragraph is based on average cleanliness over time. Yet the same average cleanliness can be achieved by two different agents, one of which does a mediocre job all the time while the other cleans energetically but takes long breaks. Which is preferable might seem to be a fine point of janitorial science, but in fact it is a deep philosophical question with far-reaching implications. Which is better— a reckless life of highs and lows, or a safe but humdrum existence? Which is better—an economy where everyone lives in moderate poverty, or one in which some live in plenty while others are very poor?

Eine Leistungsbewertung verkörpert das Kriterium für den Erfolg des Verhaltens eines Agenten. Als Faustregel kann gesagt werden, es sei besser eine Leistungsbewertung nicht danach zu entwickeln, was man in der Umgebung haben will, sondern danach, wie man glaubt, dass sich der Agent verhalten soll. die Auswahl einer geeigneten Leistungsbewertung ist oft schwierig.

2.6. On what does rationality depend?

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

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

This leads to a definition of a rational agent:

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

Omniscience (Allwissenheit)?

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. Consider the following example: I am walking along the Champs Elysees one day and I see an old friend across the street There is no traffic nearby and I'm not otherwise engaged, so, being rational, I start to cross the street Meanwhile, at 33,000 feet, a cargo door falls off a passing airliner, and before I make it to the other side of the street I am flattened. Was I irrational to cross the street? It is unlikely that my obituary would read "Idiot attempts to cross street."

This example shows that rationality is not the same as perfection. Rationality maximizes expected performance, while perfection maximizes actual performance. Retreating from a requirement of perfection is not just a question of being fair to agents. The point is that if we expect an agent to do what turns out to be the best action after the fact, it will be impossible to design an agent to fulfill this specification—unless we improve the performance of crystal balls or time machines.

Rational != omniscient: percepts may not supply all relevant information

Learning agents

Doing actions in order to **modify future percepts**—sometimes called information gathering—is an important part of rationality. An example of information gathering is provided by the exploration that must be undertaken by a vacuum-cleaning agent in an initially unknown environment.

Our definition requires a rational agent not only to gather information but also to learn as much as possible from what it perceives. The agent's initial configuration could reflect sonic prior knowledge of the environment, but as the agent gains experience this may be modified and augmented.

Autonomous agents

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, 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. For example, a vacuum-cleaning agent that learns to foresee where and when additional dirt will appear will do better than one that does not. As a practical matter, one seldom requires complete

autonomy from the start: when the agent has had little or no experience, it would have to act randomly unless the designer gave some assistance. So, just as evolution provides animals with enough built-in reflexes to survive long enough to learn for themselves, it would be reasonable to provide an artificial intelligent agent with some initial knowledge as well as an ability to learn. 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.

2.7. Specifying the task environment? PEAS!

Now that we have a definition of rationality, we are almost ready to think about building rational agents. First, however, we must think about task environments, which are essentially the "problems" to which rational agents are the "solutions."

In our discussion of the rationality of the simple vacuum-cleaner agent, we had to specify the **performance measure,** the **environment**, and the agent's **actuators** and **sensors**. We group all these under the heading of the task environment. For the acronymically minded, we call this the **PEAS (Performance, Environment, Actuators, Sensors)** description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

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

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

First, what is the performance measure to which we would like our automated driver to aspire? 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. Obviously, some of these goals conflict, so tradeoffs will be required.

Next, what is the driving environment that the taxi will face? 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 taxi must also interact with potential and actual passengers. There are also some optional choices. The taxi might need to operate in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not. It could always be driving on the right, or we might want it to be flexible enough to drive on the left when in Britain or Japan. Obviously, the more restricted the environment, the easier the design problem.

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 controllable video cameras so that it can see the road; it might augment these with infrared or sonar sensors to detect distances to other cars and obstacles. To avoid speeding tickets, the taxi should have a speedometer, and control the vehicle properly, especially on curves, it should have an accelerometer. To determine the mechanical state of the vehicle, it will need the usual array of engine, fuel, and electrical system sensors. Like many human drivers, it might want a global positioning system (GPS) so that it doesn't get lost. Finally, it will need a keyboard or microphone for the passenger to request a destination.

2.8. 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. A task environment is effectively fully observable if the sensors detect all aspects that are relevant to the choice of action. 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, and an automated taxi cannot see what other drivers are thinking. If the agent has no sensors at all then the environment is unobservable.

Single agent vs. multi-agent: The distinction between single-agent and multi-agent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment. There are, however, some subtle issues. First, we have described how an entity may be viewed as an agent, but we have not explained which entities must be viewed as agents. Does an agent A (the taxi driver for example) have to treat an object B (another vehicle) as an agent. or can it be treated merely as an object behaving according to the laws of physics, analogous to waves at the beach or leaves blowing in the wind? The key distinction is whether B's behavior is best described as maximizing a performance measure whose value depends on agent A's behavior. For example, in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent As performance measure. Thus, chess is a competitive multi-agent environment. In the taxi-driving environment, on the other hand, avoiding collisions maximizes the performance measure of all agents, so it is a partially cooperative multi-agent environment. It is also partially competitive because, for example, only one car can occupy a parking space. The agent-design problems in multi-agent environments are often quite different from those in single-agent environments; for example, communication often emerges as a rational behavior in multi-agent environments; in some competitive environments, randomized behavior is rational because it avoids the pitfalls of predictability.

Deterministic vs. stochastic: If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic. In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment. If the environment is partially observable, however, then it could appear to be stochastic. Most real situations are so complex that it is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as stochastic. Taxi driving is clearly stochastic in this sense, because one can never predict the behavior of traffic exactly; moreover, one's tires blow out and one's engine seizes up without warning. The vacuum world as we described it is deterministic, but variations can include stochastic elements such as randomly appearing dirt and an unreliable suction mechanism. We say an environment is uncertain if it is not fully observable or not deterministic. One final note: our use of the word "stochastic" generally implies that uncertainty about outcomes is quantified in terms of probabilities; a nondeterministic environment is one in which actions are characterized by their possible outcomes, but no probabilities are attached to them. Nondeterministic environment descriptions are usually

associated with performance measures that require the agent to succeed for all possible outcomes of its actions.

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. Crucially, 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 arc 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.

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. 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 we say the environment is semi-dynamic. 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 semi-dynamic. Crossword puzzles are static.

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. For example, the chess environment has a finite number of distinct states (excluding the clock), Chess also has a discrete set of percepts and actions. 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 and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.

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
Interactive. English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

2.9. 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 architecture:

agent = architecture + program .

Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like Walk, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several onboard computers, cameras, and other sensors. 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.

Architecture: programming device + sensors + actuators

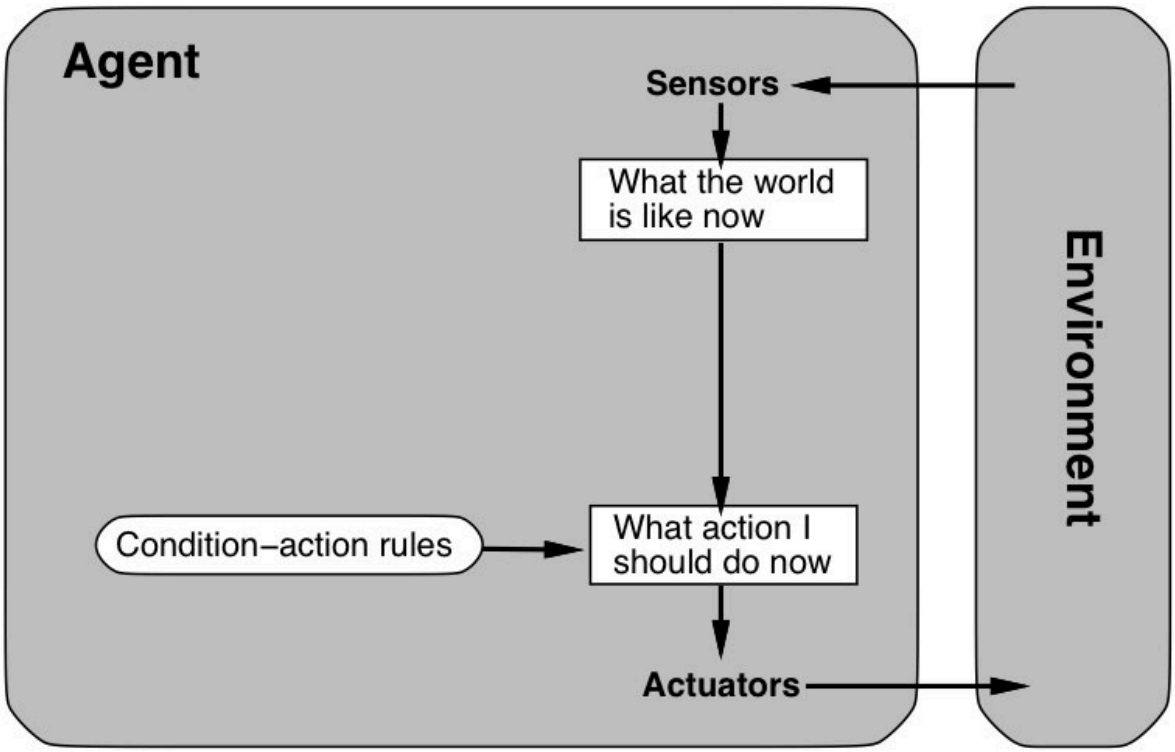
Program: gets sensor data, returns actions for the actuators

2.10. Agent programs

Four basic types of agent programs in order of increasing generality (All these can be turned into learning agents):

- simple reflex agents
- reflex agents with state
- goal-based agents
- utility-based agents

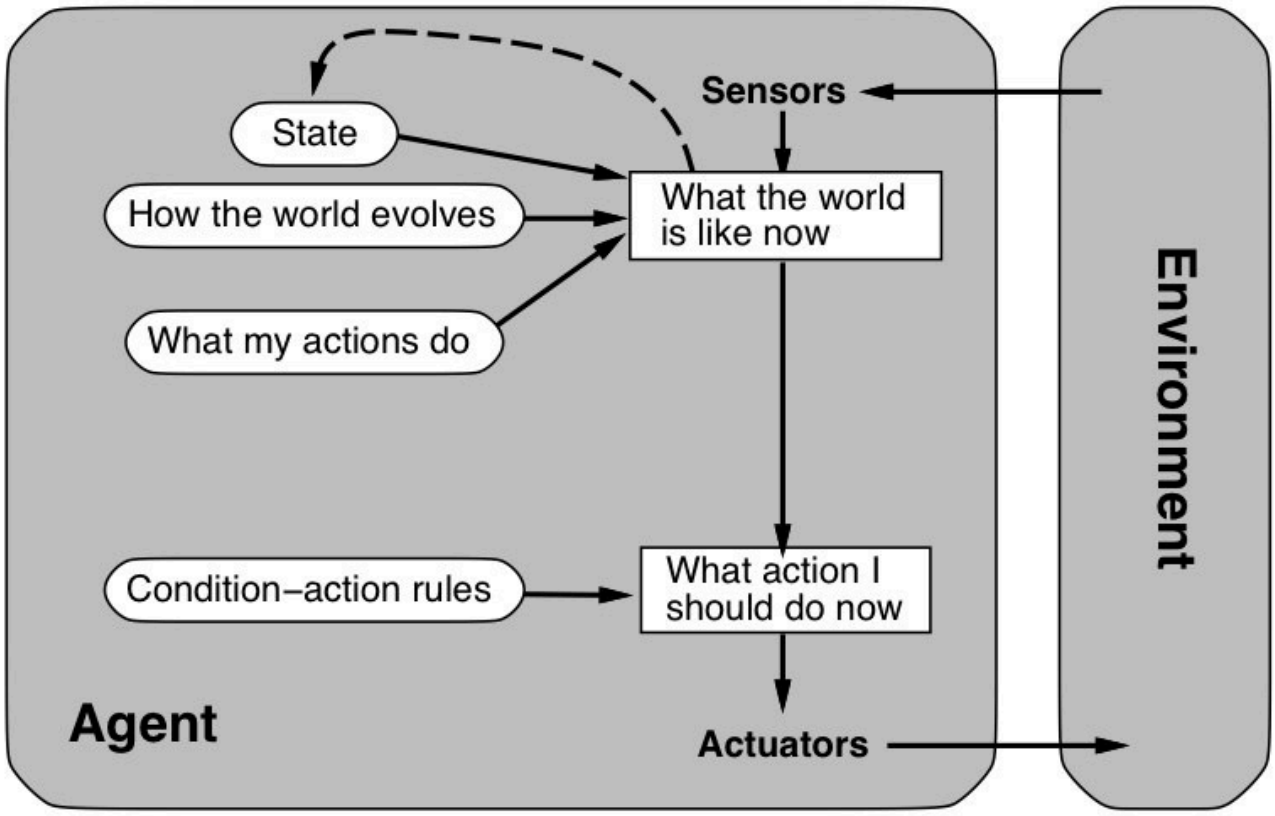
Simple reflex agents



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. For example, the vacuum agent is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. Art agent program for this agent is shown in Figure 2.8.

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. Even a little bit of unobservability can cause serious trouble.

Model-based reflex agents (Reflex agents with state)



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. For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously. For other driving tasks such as changing lanes, the agent needs to keep track of where the other cars are if it can't see them all at once.

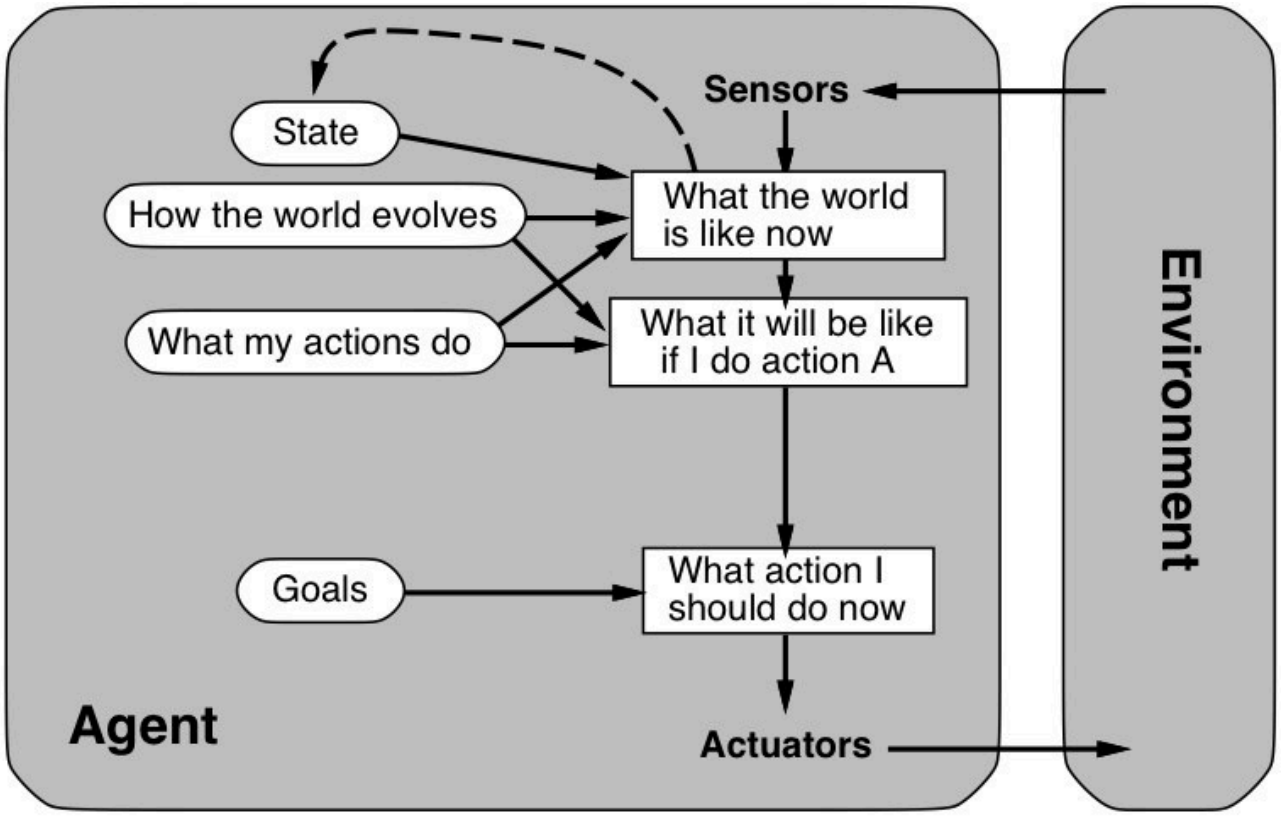
And for any driving to be possible at all, the agent needs to keep track of where its keys are Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program.

First, we need some information about how the world evolves independently of the agent—for example, that an overtaking car generally will be closer behind than it was a moment ago.

Second, we need some information about how the agent's own actions affect the world—for example, that when the agent turns the steering wheel clockwise, the car turns to the right, or that after driving for five minutes northbound on the freeway, one is usually about five miles

north of where one was five minutes ago. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a model of the world. An agent that uses such a model is called a model-based agent.

Goal-based agents



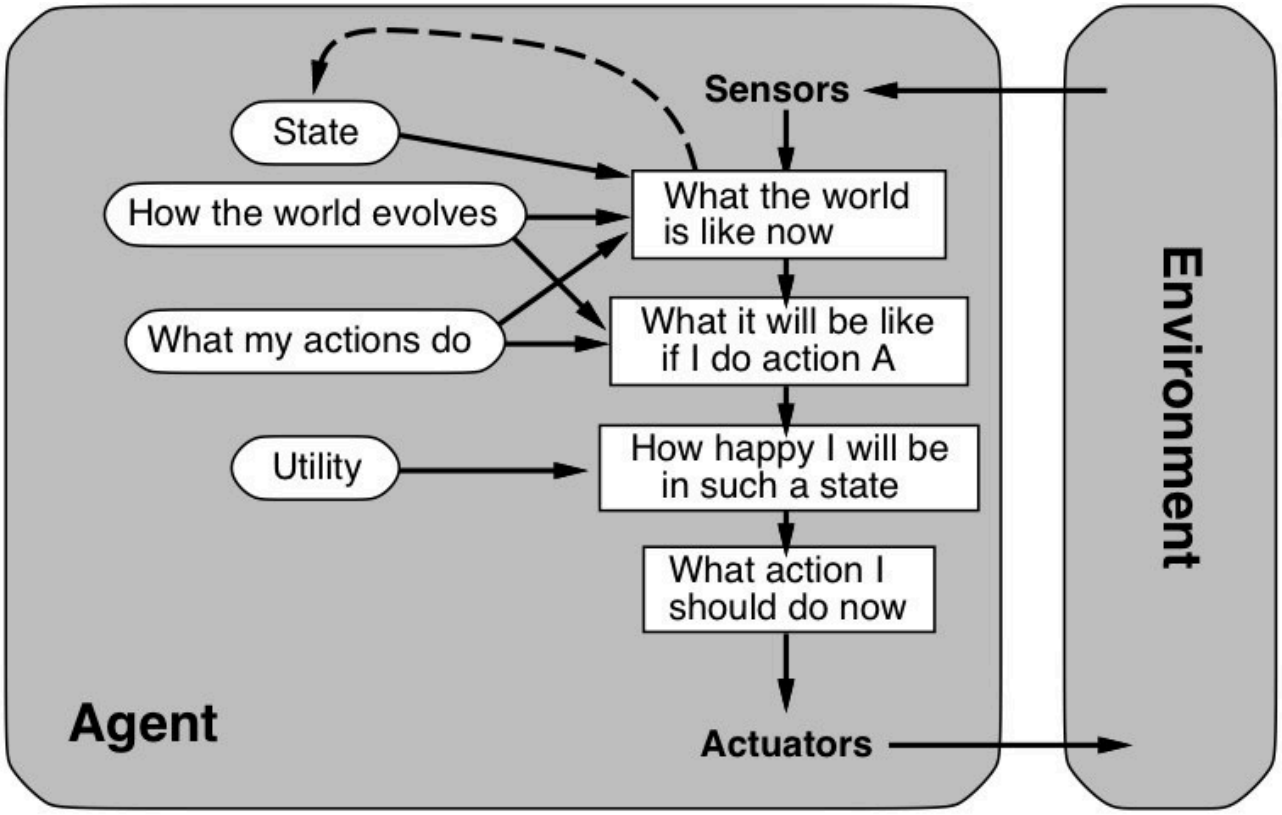
Knowing something about the current state of the environment is not always enough to decide what to do. For example. at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state 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.

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 Al devoted to finding action sequences that achieve the agent's goals.

Notice that decision making of this kind is fundamentally different from the condition-action rules described earlier, in that it involves consideration of the future—both "What will happen if I do such-and-such?"and "Will that make me happy?"

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.

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. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term utility instead.

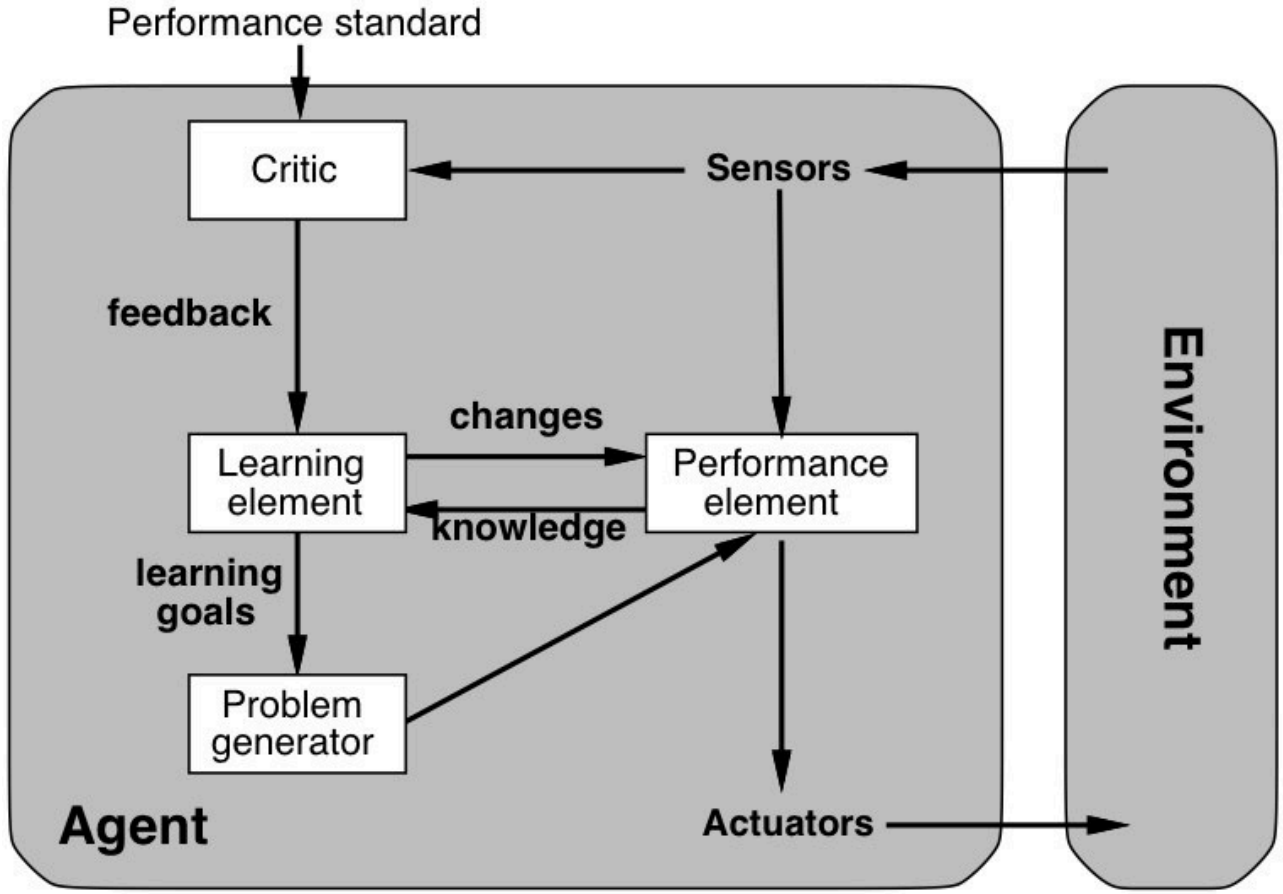
We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination. An agent's utility function is essentially an internalization of the performance measure. If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

Let us emphasize again that this is not the only way to be rational—we have already seen a rational agent program for the vacuum world that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions.

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.

Learning agents



We have described agent programs with various methods for selecting actions. We have not, so far, explained how the agent programs come into being. In his famous early paper, Turing considers the idea of actually programming his intelligent machines by hand. He estimates how much work this might take and concludes "Some more expeditious method seems desirable." The method he proposes is to build learning machines and then to teach them.

In many areas of AI, this is now the preferred method for creating state-of-the-art systems. Learning has another advantage, as we noted earlier it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow.

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. 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. The point is that if the performance element had its way, it would keep doing the actions that are best ; given what it knows. But if the agent is willing to explore a little and do some perhaps suboptimal actions in the short run, it might discover much better actions for the long run. The problem generator's job is to suggest these exploratory actions.

3. Chapter 3 (Solving problems by searching)

3.1. Problem solving agents

Goal formulation

Is based on the current situation and the agent's performance measure, is the first step in problem solving.

We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent's task is to find out how to act, now and in the future, so that it reaches a goal state. Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider. If it were to consider actions at the level of "move the left foot forward an inch" or "turn the steering wheel one degree left." the agent would probably never find its way out of the parking lot, let alone to Bucharest, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution.

Problem formulation

Is the process of deciding what actions and states to consider, given a goal. For now, let us assume that the agent will consider actions at the level of driving from one major town to another. Each state therefore corresponds to being in a particular town.

Find solution

Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.' In other words, the agent will not know witch of its possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no additional information then it is has no choice but to try one of the actions at random.

But suppose the agent has a map of Romania. The point of a map is to provide the agent with information about the states it might get itself into and the actions it can take. The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest. Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying cut the driving actions that correspond to the legs of the journey. In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

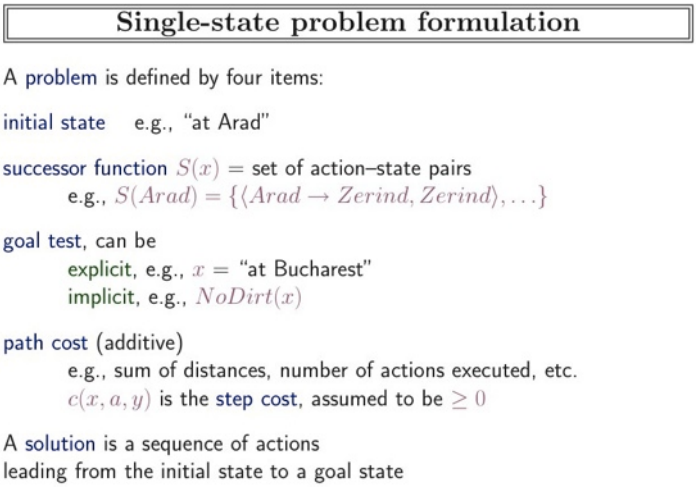
The process of looking for a sequence of actions that reaches the goal is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. After formulating a goal and a problem to solve. the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal

Assumptions for the environment: static, observable, discrete and deterministic.

3.2. Well-defined problems and solutions

A problem can be defined formally by four components:

- The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as *In(Arad)*.
- A description of the possible actions available to the agent Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is applicable in s. For example, from the state *In(Arad)*, the applicable actions are *{Go(Sibiu), Go(Timisoara), Go(Zerind)}*.



- The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set *{In(Bucharest)}*.
- A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

3.3. Formulating problems / Selecting a state space

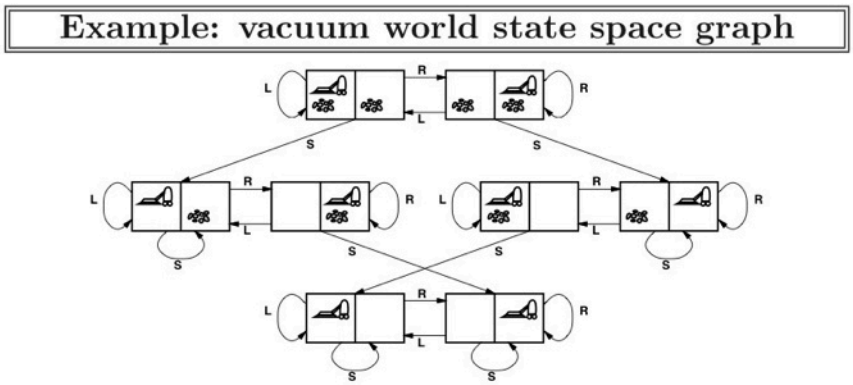
In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a model—an abstract mathematical description—and not the real thing. Compare the simple state description we have chosen, In(Arad), to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called **abstraction**.

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.

For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind" (Abstract) solution = set of real paths that are solutions in the real world.

Each abstract action should be "easier" than the original problem!



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??: 1 per action (0 for *NoOp*)

3.4. Searching for solutions

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

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Infrastructure for search algorithms

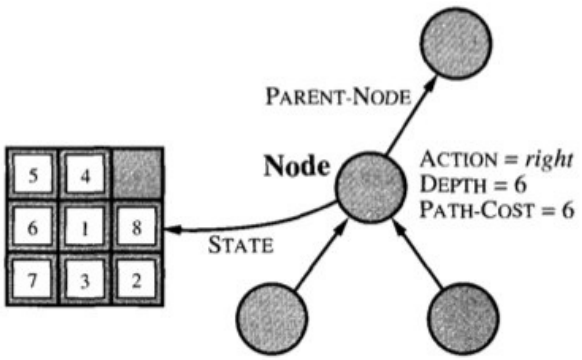
There are many ways to represent nodes, but we will assume that a node is a data structure with five components:

STATE: the state in the state space to which the node corresponds;

PARENT-NODE: the node in the search tree that generated this node;

ACTION: the action that was applied to the parent to generate the node;

PATH-COST: the cost, traditionally denoted by *g(n)*, of the path from the initial state to the node, as indicated by the parent pointers; and



DEPTH: the number of steps along the path from the initial state.

It is important to remember the distinction between nodes and states. A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world.

The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.

We also need to represent the collection of nodes that have been generated but not yet expanded - this collection is called the **fringe**. Each element of the fringe is a leaf node, that is, a node with no successors in the tree. Therefore, we will assume that the collection of nodes is implemented as a queue. The operations on a queue are as follows:

- MAKE-QUEUE(element, ...) creates a queue with the given element(s).
- EMPTY?(queue) returns true only if there are no more elements in the queue.
- FIRST(queue) returns the first element of the queue.
- REMOVE-FRONT(queue) returns FIRST(queue) and removes it from the queue.
- INSERT(element, queue) inserts an element into the queue and returns the resulting queue. (We will see that different types of queues insert elements in different orders.)
- INSERT-ALL(elements, queue) inserts a set of elements into the queue and returns the resulting queue.

With these definitions, we can write the more formal version of the general tree-search algorithm shown below.

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action,
result)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

3.5. Measuring problem-solving performance

The output of a problem-solving algorithm is either failure or a solution. (Some algorithms might get stuck in an infinite loop and never return an output.) We will evaluate an algorithm's performance in four ways:

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

Time and space complexity are measured in terms of

b—maximum branching factor of the search tree

d—depth of the least-cost solution

m—maximum depth of the state space (may be ∞)

3.6. Uniformed search strategies

Breadth first search (Breitensuche)

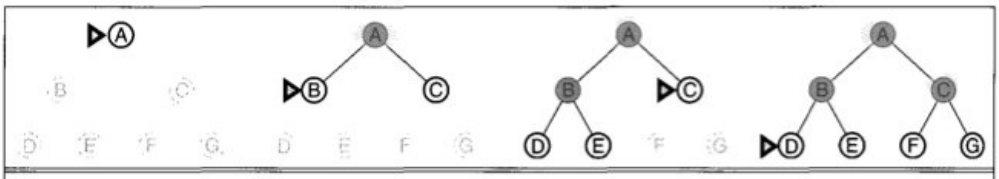


Figure 3.10 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

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

Breadth-first search can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling TREE- SEARCH(problem,FIFO-QUEUE()) results in a breadth-first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes.

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + \ldots + b^d = O(b^d)$, i.e., exp. in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Uniform-cost search

Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost. Note that if all step costs are equal, this is identical to breadth-first search.

Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state (for example, a *NoOp* action). We can guarantee completeness provided the cost of every step is greater than or equal to some small positive constant *e*. This condition is also sufficient to ensure optimality. It means that the cost of a path always increases as we go along the path.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:
$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search (Tiefensuche)

Depth-first search always expands the deepest node *n* of the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack. As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

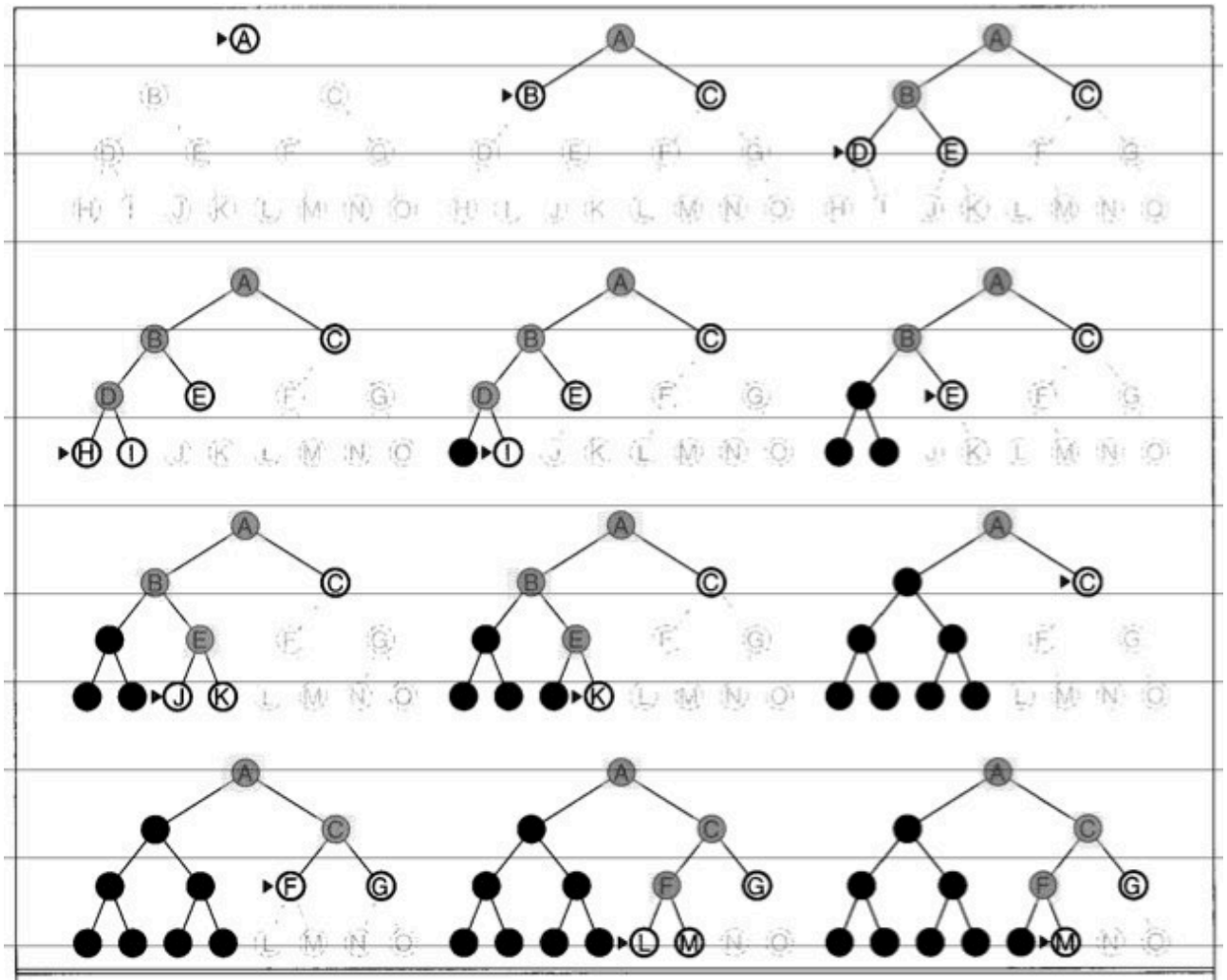


Figure 3.12 Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Depth-limited search

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit *p*. That is, nodes at depth *p* are treated as if they have no successors. This approach is called depth-limited search. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $p<d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.)

Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first search, that finds the best depth limit. It does this by gradually increasing the limit-first 0, then 1, then 2, and so on - until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are very modest

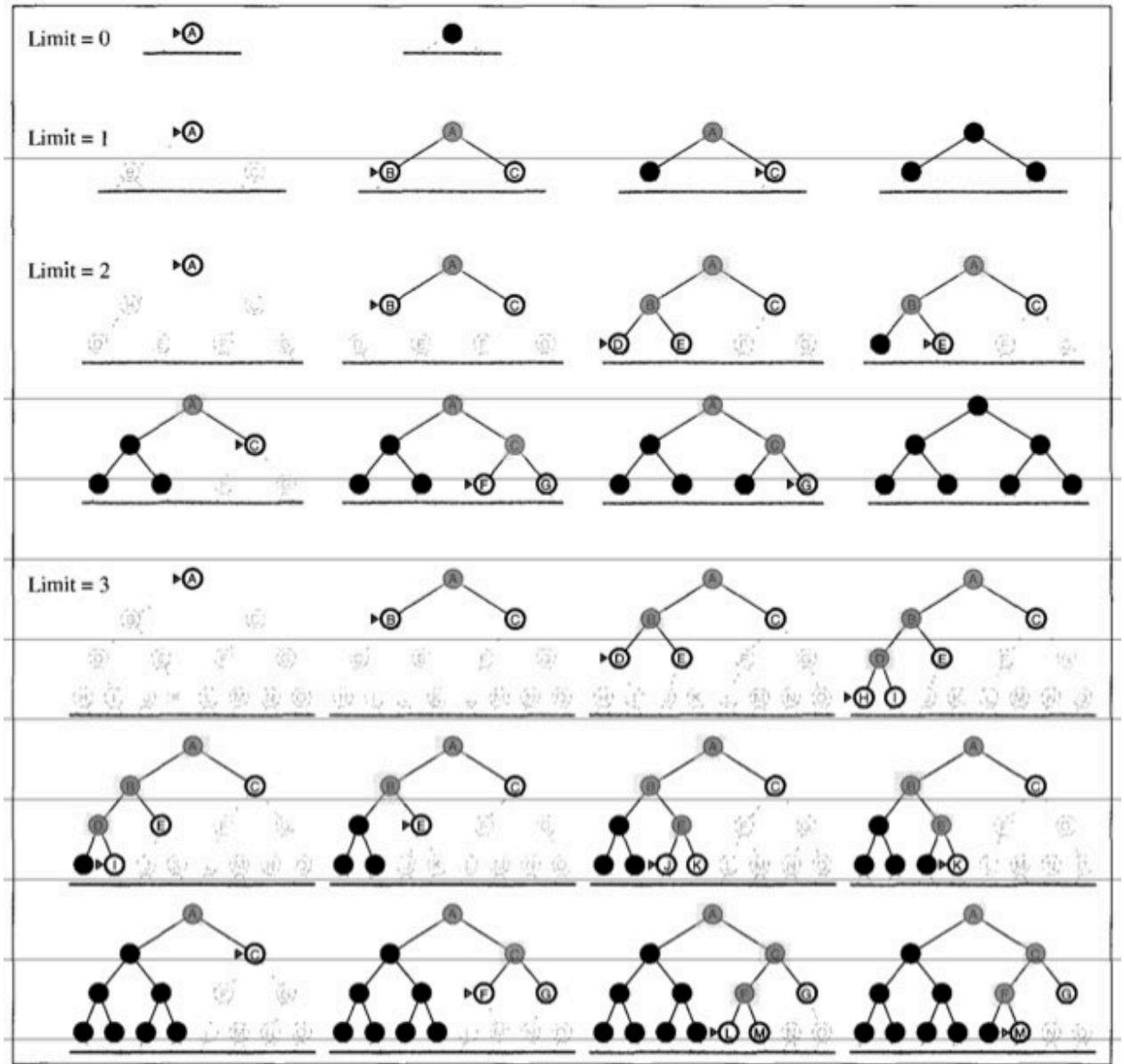


Figure 3.15 Four iterations of iterative deepening search on a binary tree.

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$ (We will prove this)

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is **generated**

3.7. Summary of algorithms

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

Figure 3.17 Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

3.8. Repeated states

Up to this point, we have all but ignored one of the most important complications to the search process: the possibility of wasting time by expanding states that have already been encountered and expanded before. For some problems, this possibility never comes up; the state space is a tree and there is only one path to each state.

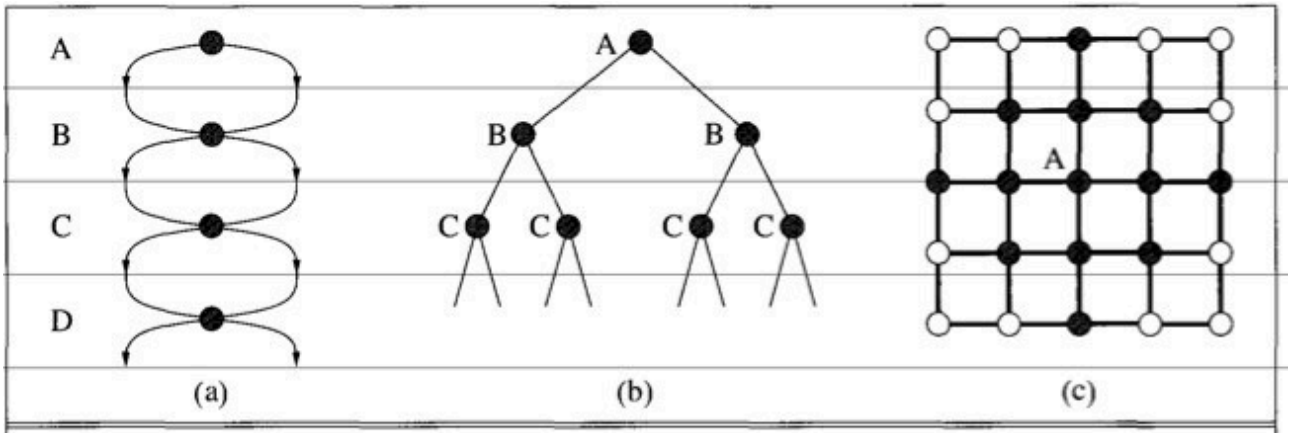


Figure 3.18 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

Solving adding a list of closed states:

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

4. Chapter 4 (Informed search exploration)

4.1. best-first search

The general approach we consider is called best-first search. Best-first search is an instance of the general *TREE-SEARCH* or *GRAPH-SEARCH* algorithm in which a node is selected for expansion based on an evaluation function *f(n).* The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of *f* instead of *g* to order the priority queue.

The choice of *f* determines the search strategy. Most best-first algorithms include as a component of *f* a heuristic function, denoted *h(n)*:

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

(Notice that *h(n)* takes a node as input, but, unlike *g(n)*, it depends only on the state at that node.) For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. For now, we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint! If *n* is a goal node, then *h(n)=0*. The remainder of this section covers two ways to use heuristic information to guide search.

Implementation: fringe is a queue sorted in decreasing order of desirability

4.2. Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, *f(n) = h(n)*. *h(n)* = estimate of cost from n to the closest goal

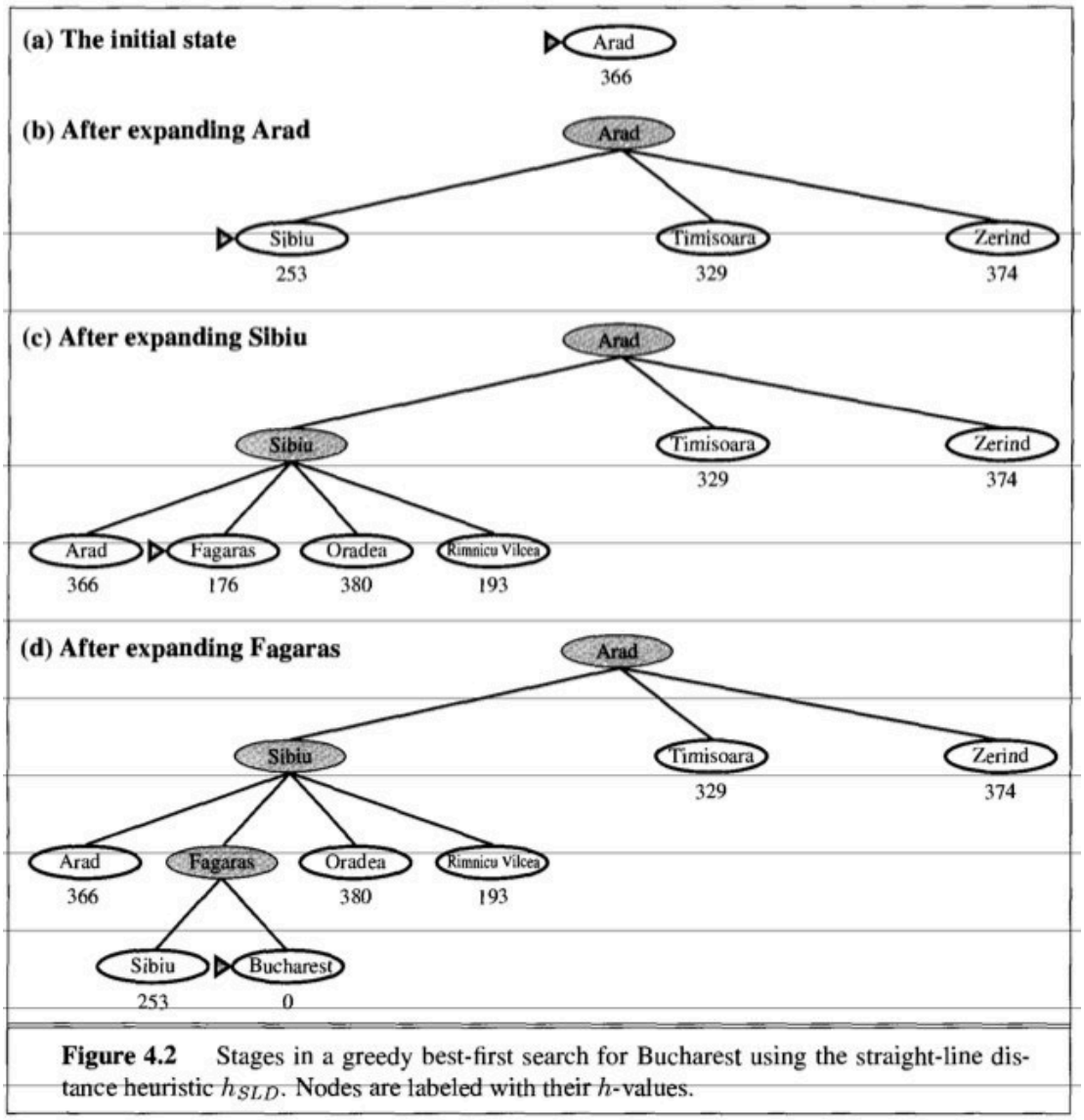


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

E.g., hSLD(n) = straight-line distance from n to Bucharest Greedy search expands the node that appears to be closest to goal

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,
Iasi → Neamt → Iasi → Neamt →
Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

4.3. A search: Minimizing the total estimated solution cost*

The most widely-known form of best-first search is called A* search (pronounced "A-star search"). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

Since *g(n)* gives the path cost from the start node to node *n*, and *h(n)* is the estimated cost of the cheapest path from *n* to the goal, we have

f(n) = estimated cost of the cheapest solution through *n*.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of *g(n)+h(n)*. It turns out that this strategy is more than just reasonable: provided that the heuristic function *h(n)* satisfies certain conditions, A* search is both complete and optimal.

The optimality of A* is straightforward to analyze if it is used with TREE-SEARCH. In this case, A* is optimal if *h(n)* is an admissible heuristic - that is, provided that *h(n)* never overestimates the cost to reach the goal. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since *g(n)* is the exact cost to reach n, we have as immediate consequence that *f(n)* never overestimates the true cost of a solution through n.

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in h × length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$

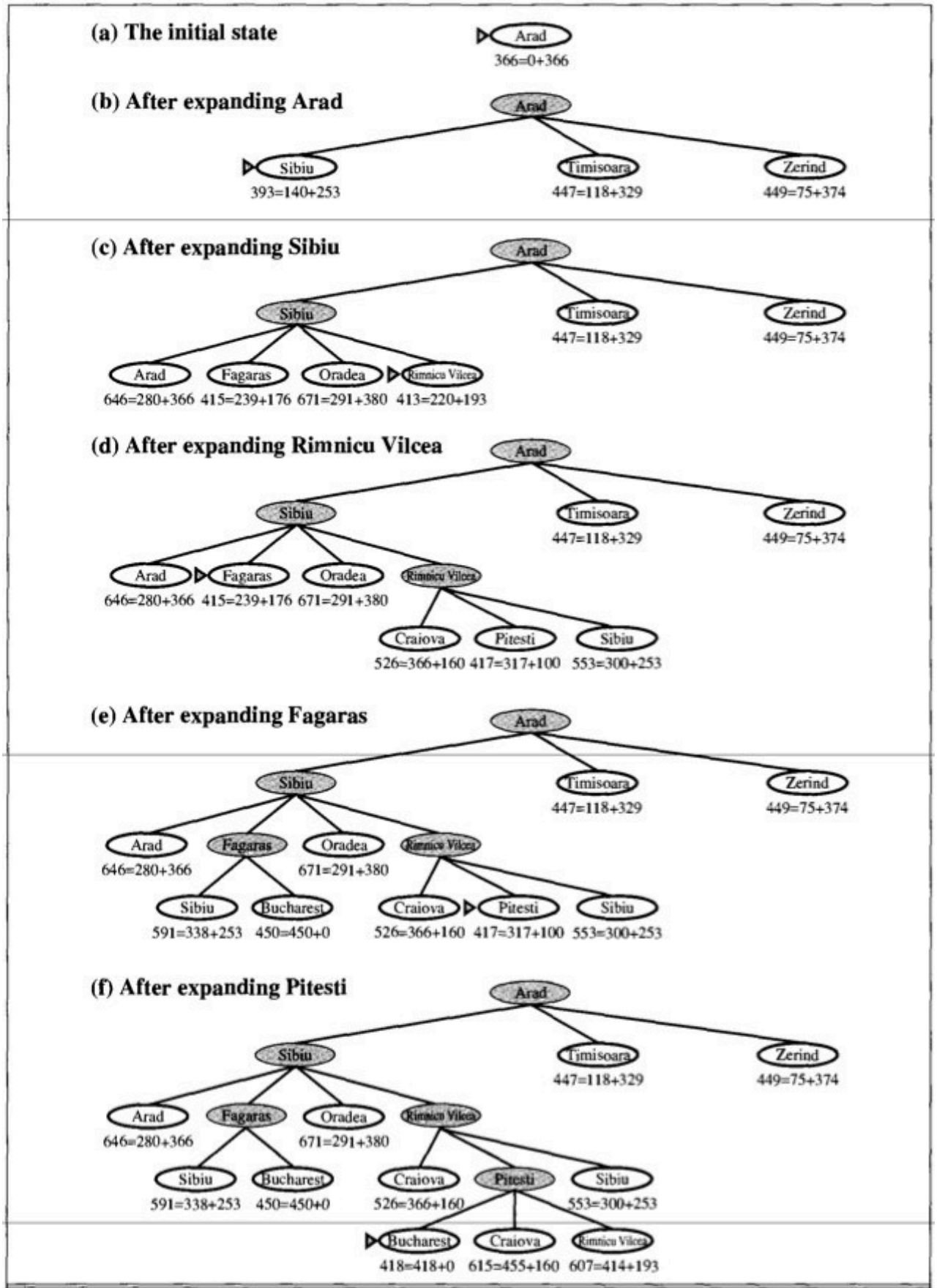
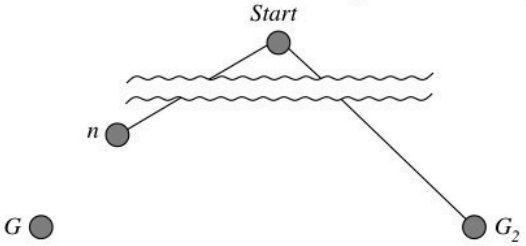


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

Optimality of A* (standard proof)

Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



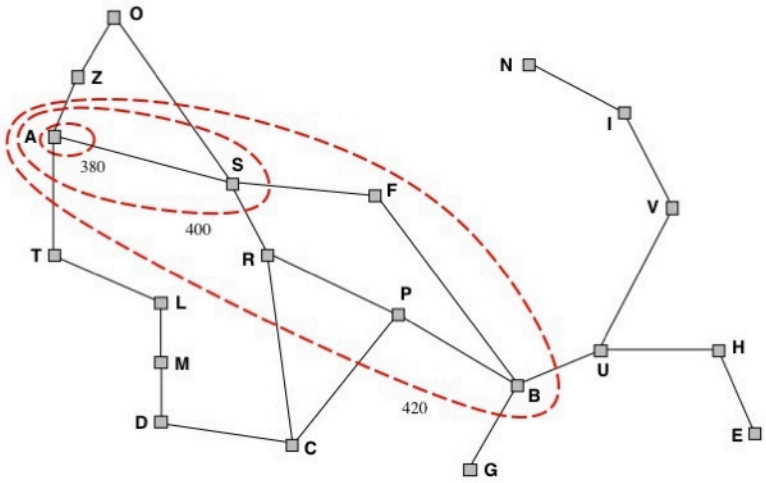
$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



4.4. Proof of lemma (consistent heuristics)

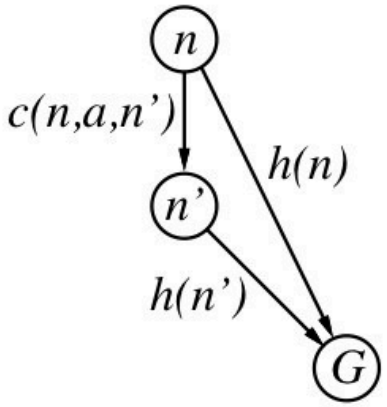
A heuristic is consistent if

$$h(n) \leq c(n, a, n') + h(n')$$

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path.



n' is successor of *n*. *c(n,a,n')* = costs from *n* to *n'* by using action *a*

4.5. Admissible heuristics

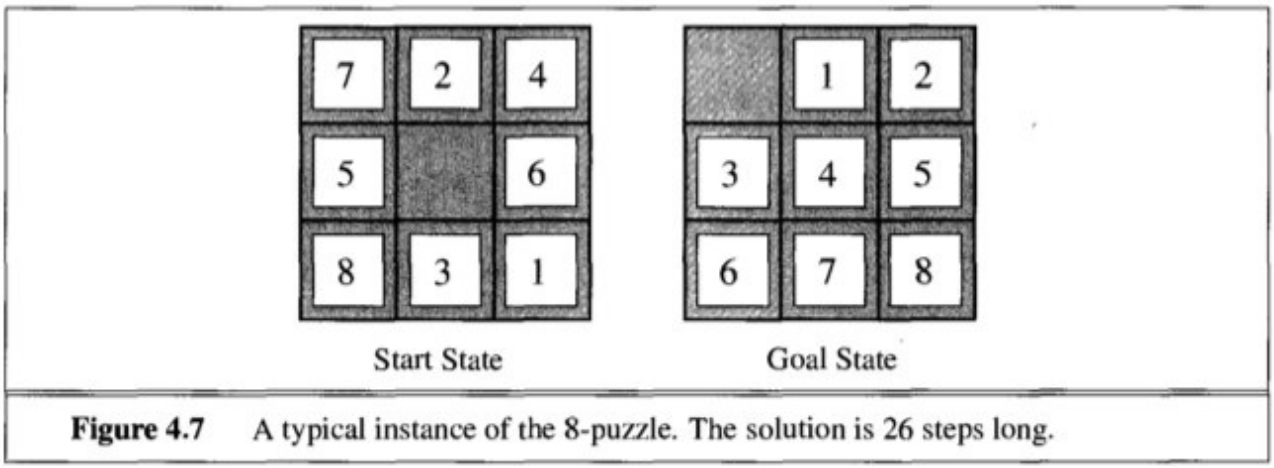


Figure 4.7 A typical instance of the 8-puzzle. The solution is 26 steps long.

If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. Here are two commonly-used candidates:

1. h_1 = the number of misplaced tiles. For Figure 4.7, all of the eight tiles are out of position, so the start state would have h=8. h_1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

2. h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. h is also admissible, because all any move can do is move one tile one step 2 closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of h_2= 3+1+2+2+2+3+3+2= 18.

As we would hope, neither of these overestimates the true solution cost, which is 26.

4.6. Dominance

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 4.8 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

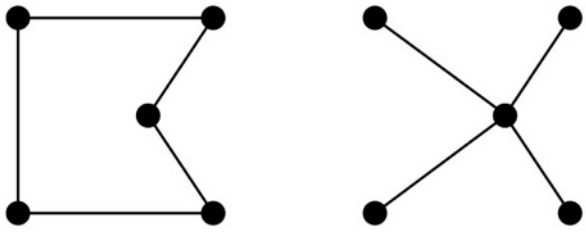
One might ask whether h_2 is always better than h_1. The answer is "Essentially, yes." 1 is easy to see from the definitions of the two heuristics that, for any node n, $h_2(n) > h_1(n)$ We thus say that h_2 dominates h_1. Domination translates directly into efficiency: A* using h_2 will never expand more nodes than A* using h_1 (except possibly for some nodes with $f(n) = C^*$).

4.7. Relaxed problems

We have seen that both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h_2 is better. How might one have come up with h_2? Is it possible for a computer to invent such a heuristic mechanically?

h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere, instead of just to the adjacent empty square, then h_1 would give the exact number of steps in the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a **relaxed problem**. The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem. The heuristic is admissible because the optimal solution in the original problem is, by definition, also a solution in the relaxed problem and therefore must be at least as expensive as the optimal solution in the relaxed problem. Because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent.

Well-known example: travelling salesperson problem (TSP)
Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour

4.8. Summary

Heuristic functions estimate costs of shortest paths. Good heuristics can dramatically reduce search cost.

Greedy best-first search expands lowest *h(n)*

– incomplete and not always optimal

A$*$ search expands lowest *g(n) + h(n)*

– complete and optimal

– also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

5. Chapter 6 (Adversarial Search, Games)

Chapter 2 introduced multi-agent environments, in which each agent needs to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce contingencies into the agent's problem-solving process. In this chapter we cover competitive environments, in which the agents' goals are in conflict, giving rise to adversarial search problems — often known as games.

"Unpredictable" opponent ⇒ solution is a strategy specifying a move for every possible opponent reply

Time limits ⇒ unlikely to find goal, must approximate Plan of attack:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)

5.1. Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

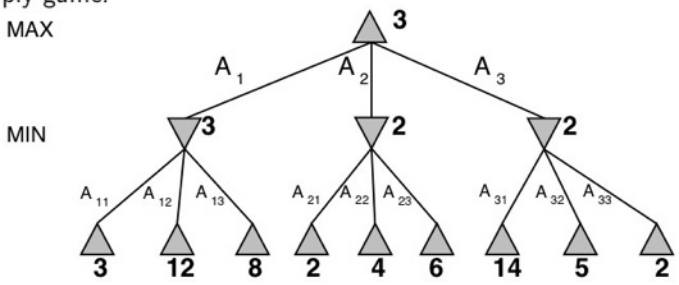
5.2. Minimax strategy

Given a game tree, the optimal strategy can be determined from the *minimax* value of each node, which we write as *Minimax(n)*. The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:




```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
  return v
```

Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exact solution completely infeasible

But do we need to explore every path?

5.3. α–β pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree b^m. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of pruning from Chapter 3 to eliminate large parts of the tree from consideration. The particular technique we examine is called alpha—beta pruning. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

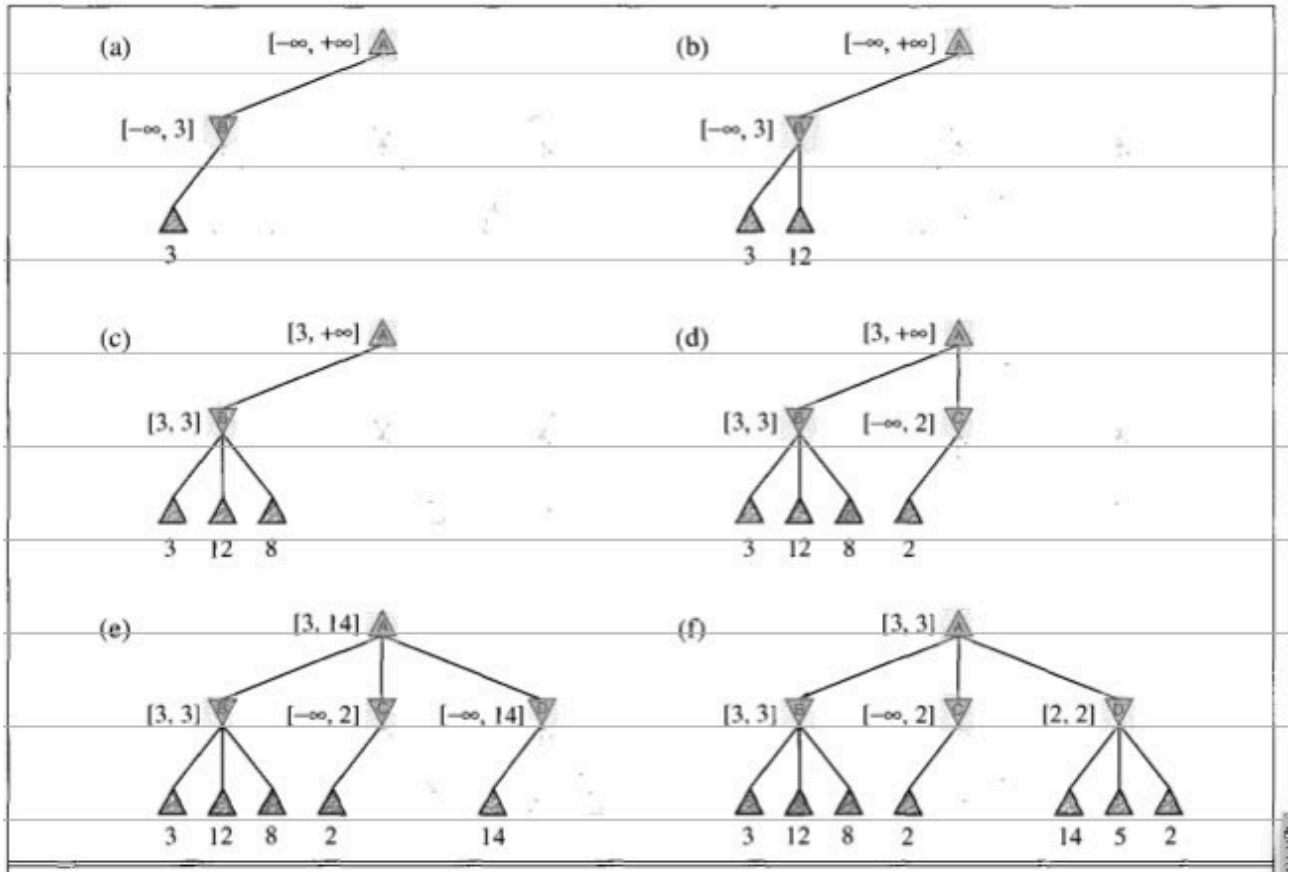


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha—beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n

```
function ALPHA-BETA-DECISION(state) returns an action
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  same as MAX-VALUE but with roles of α, β reversed
```

Properties of α–β

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

Unfortunately, 35^{50} is still impossible!

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

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

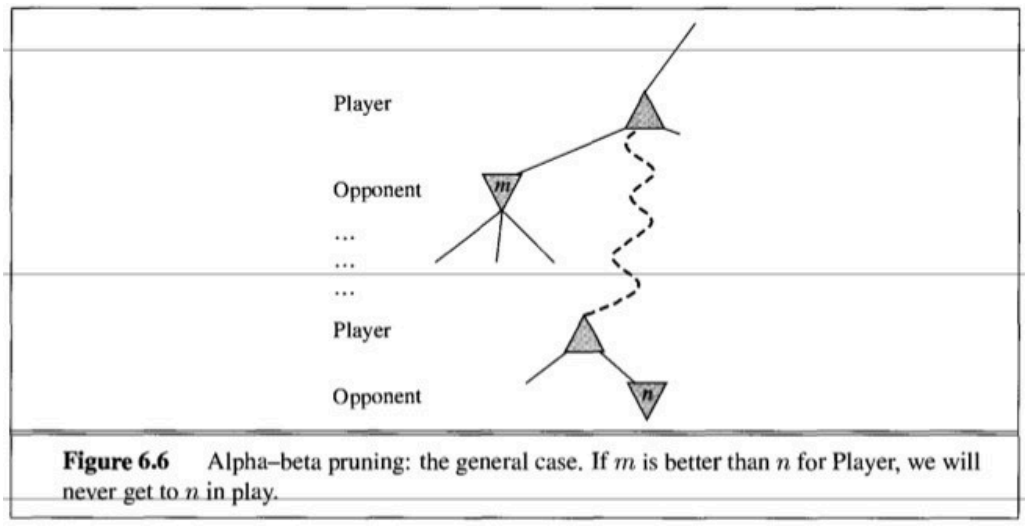


Figure 6.6 Alpha–beta pruning: the general case. If m is better than n for Player, we will never get to n in play.

5.4. Resource limits

The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it. However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. Claude Shannon's paper Programming a Computer for Playing Chess (1950) proposed instead that programs should cut off the search earlier and apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves. In other words, the suggestion is to alter minimax or alpha-beta in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and replace the terminal test by a cutoff test that decides when to apply EVAL.

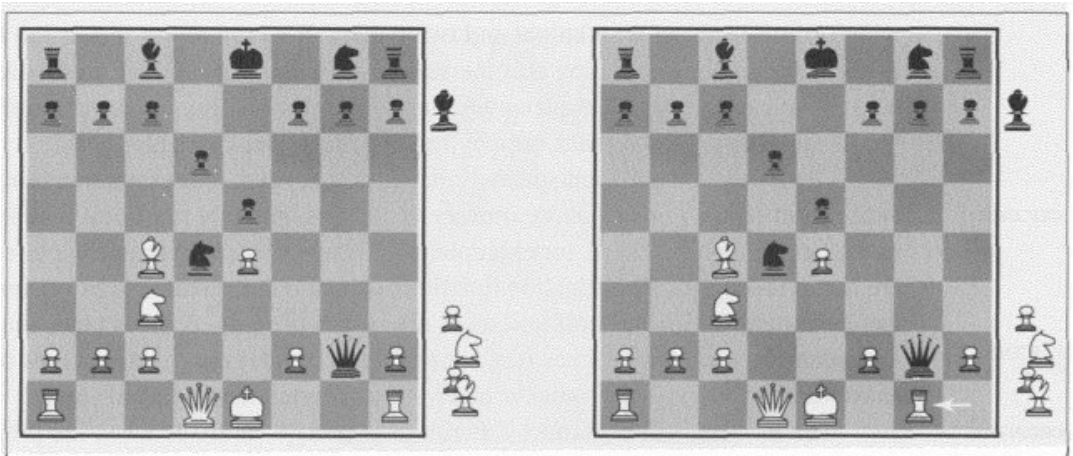
Standard approach:

- Use Cutoff-Test instead of Terminal-Test (e.g., depth limit (perhaps add quiescence search))
- Use Eval instead of Utility (i.e., evaluation function that estimates desirability of position)

Suppose we have 100 seconds, explore 104 nodes/second ⇒ 106 nodes per move ≈ 358/2 ⇒ α–β reaches depth 8 ⇒ pretty good chess program

Evaluation function

An evaluation function returns an estimate of the expected utility of the game from a given position, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal. The idea of an estimator was not new when Shannon proposed it. For centuries, chess players (and aficionados of other games) have developed ways of judging the value of a position because humans are even more limited in the amount of search they can do than are computer programs. It should be clear that the performance of a game-playing program depends strongly on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.



(a) White to move (b) White to move

Figure 6.8 Two slightly different chess positions. In (a), black has an advantage of a knight and two pawns and will win the game. In (b), black will lose after white captures the queen.

For chess, typically linear weighted sum of features $Eval(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)$

e.g., $w_1 = 9$ with $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Cutting off search

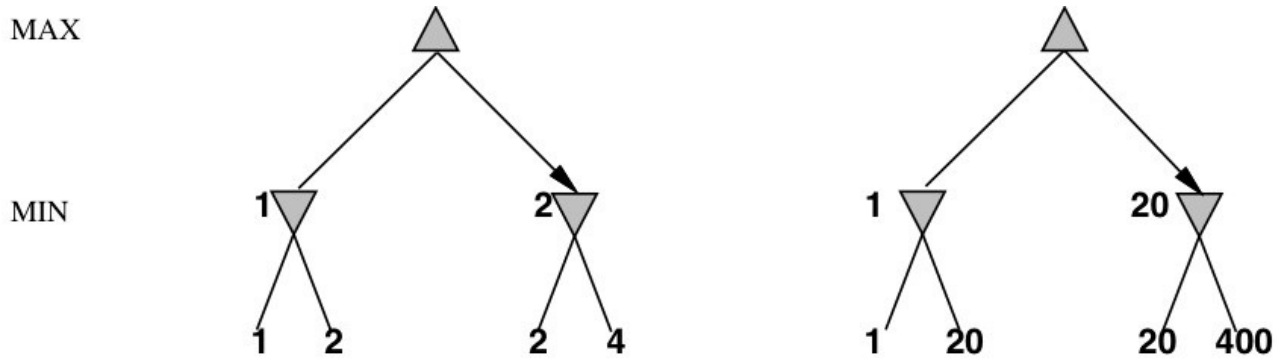
The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. We replace the two lines in Figure 5.7 that mention TERMINAL-TEST with the following line:

if CUTOFF-TEST(state, depth) then return EVAL(state)

We also must arrange for some bookkeeping so that the current depth is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that CUTOFF-TEST(state, depth) returns true for all depth greater than some fixed depth d. (It must also return true for all terminal states, just as TERMINAL-TEST did.) The depth d is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening. When time runs out, the program returns the move selected by the deepest completed search. As a bonus, iterative deepening also helps with move ordering.

5.5. Digression: Exact values don't matter

Behavior is preserved under any monotonic transformation of Eval. Only the order matters: payoff in deterministic games acts as an ordinal utility function



5.6. Deterministic games in practice

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Chess: Deep Blue defeated human world champion Gary Kasparov in a six- game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, b > 300, so most programs use pattern knowledge bases to suggest plausible moves.

5.7. Nondeterministic (stochastic) games

In real life, many unpredictable external events can put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as the throwing of dice. We call these stochastic games. Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn determine the legal moves. In the backgammon position of Figure 5.10, for example, White has rolled a 6-5 and has four possible moves.

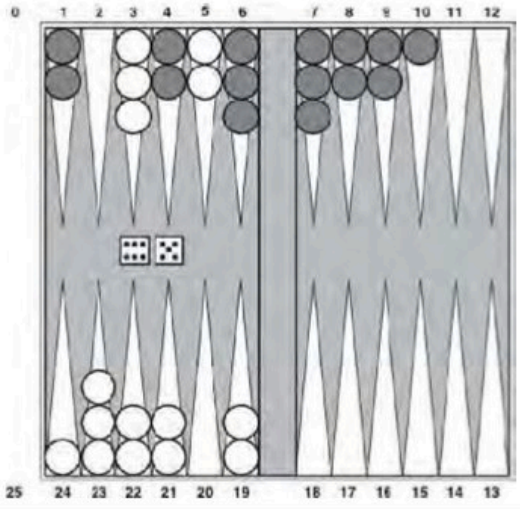
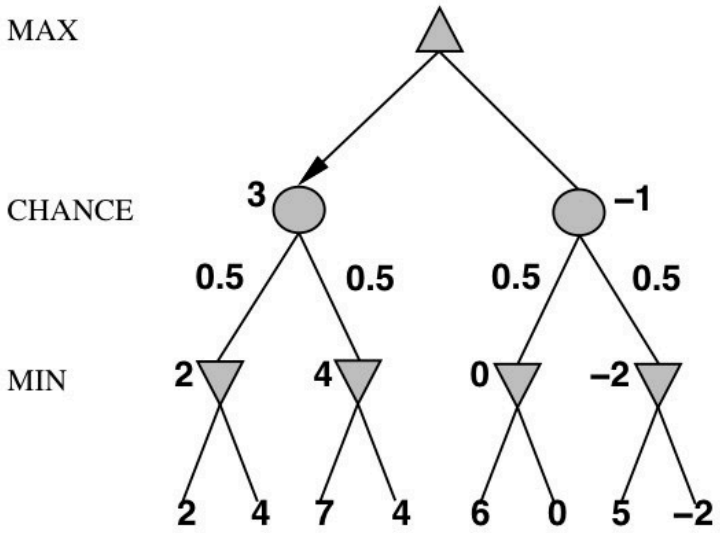


Figure 5.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6-5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5-11,11-16), where the notation (5-11,11-16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be. That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes.

Simplified example with coin-flipping:



Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

```
...
if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
...
```

5.8. Nondeterministic games in practice

Dice rolls increase b:21 possible rolls with 2 dice Backgammon ≈ 20 legal moves (can be 6,000 with 1-1 roll)

depth 4=20×(21×20)³ ≈1.2×109

As depth increases, probability of reaching a given node shrinks ⇒ value of lookahead is diminished

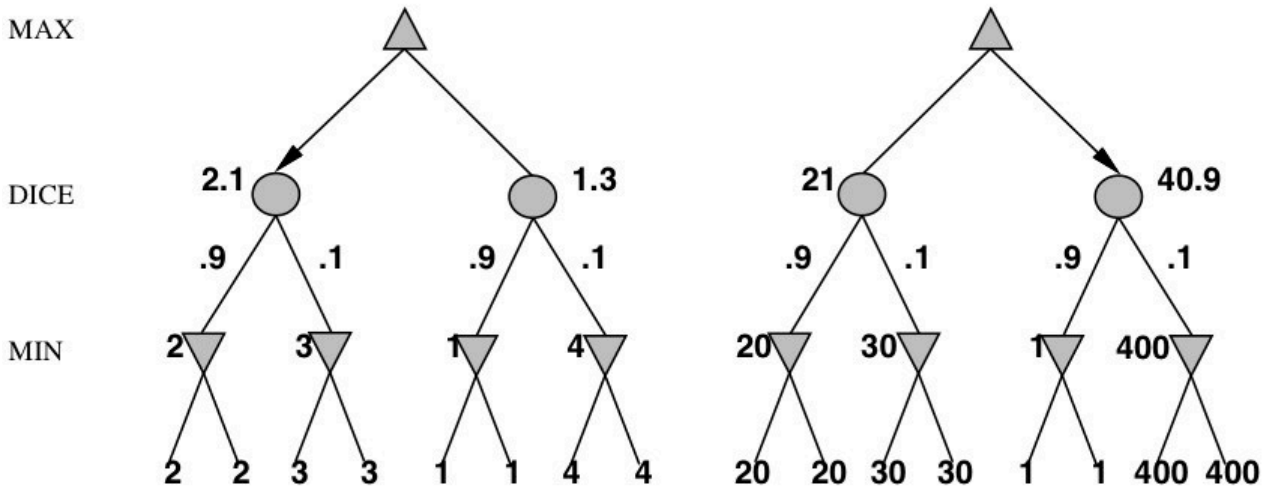
α–β pruning is much less effective

TDG_{ammon} uses depth-2 search + very good Eval ≈ world-champion level

5.9. Digression: Exact values DO matter

Behavior is preserved only by positive linear transformation of Eval.

Hence: Eval should be proportional to the expected payof.

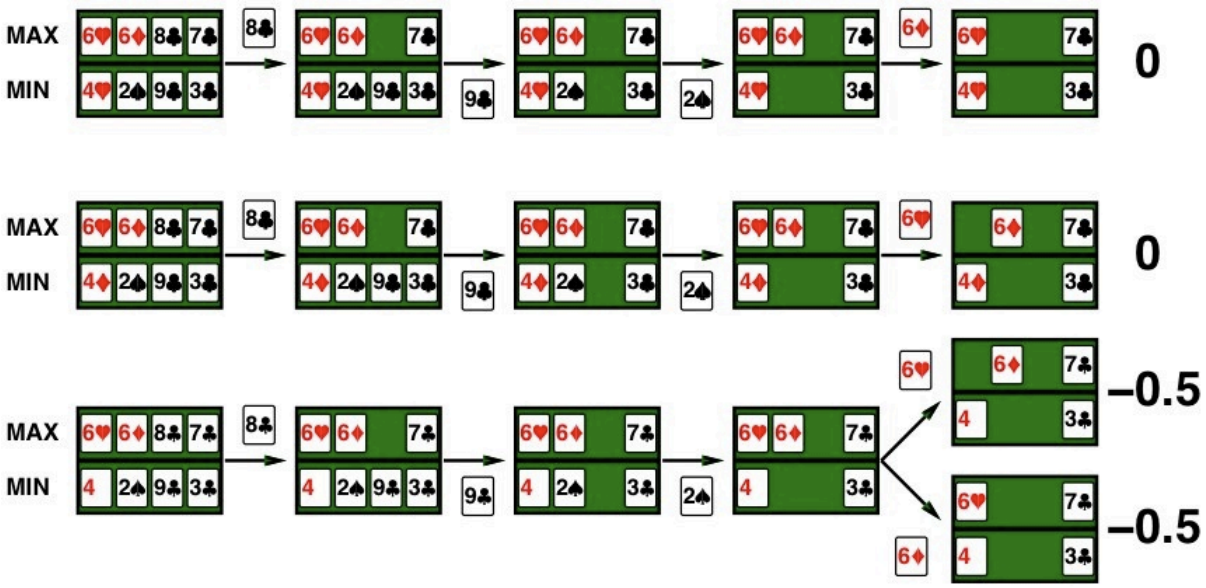


5.10. Games of imperfect information

E.g., card games, where opponent's initial cards are unknown. Typically we can calculate a probability for each possible deal. Seems just like having one big dice roll at the beginning of the game. Idea: compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals*

Special case: if an action is optimal for all deals, it's optimal.

GIB, current best bridge program, approximates this idea by 1) generating 100 deals consistent with bidding information 2) picking the action that wins most tricks on average.



5.11. Proper analysis

Intuition that the value of an action is the average of its values in all actual states is WRONG

With partial observability, value of an action depends on the information state or belief state the agent is in. Can generate and search a tree of information states.

Leads to rational behaviors such as ♦ Acting to obtain information ♦ Signalling to one's partner ♦ Acting randomly to minimize information disclosure

5.12. Summary

Games are fun to work on! (and dangerous). They illustrate several important points about AI

♦ perfection is unattainable ⇒ must approximate

♦ good idea to think about what to think about

♦ uncertainty constrains the assignment of values to states

♦ optimal decisions depend on information state, not real state

Games are to AI as grand prix racing is to automobile design

6. Chapter 10 (Knowledge Representation)

6.1. What is knowledge representation?

The representation of knowledge and reasoning from knowledge are central for AI, after all, humans know things and do reasoning. Knowledge and reasoning play a crucial role in dealing with partially observable environments.

- A knowledge-based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions.
- E.g., a physician diagnoses a patient prior to choosing a treatment ⇒ For diagnosing, the physician uses knowledge from textbooks and teachers as well as association patterns the physician cannot consciously describe.

Understanding natural language also involves inferring hidden states — namely the intention of the speaker. E.g., when we hear "John threw the stone against the mirror and broke it", we know that "it" refers to "mirror" and not to "stone".

In general, the goal of knowledge representation is the following:

- representing implicit knowledge about a certain area in such a way that it can be processed by computers
- original knowledge is encoded in suitable data structures and algorithms.

Knowledge representation is a multidisciplinary field involving methods and techniques from:

- logic: provides the formal structures and rules for performing deductions
- ontology: defines the kinds of objects in the considered application area
- computer science: supports the applications which distinguishes knowledge representation from pure philosophy.

In short: knowledge representation = application of logic and ontology for providing computational models.

6.2. Declarative vs. procedural approaches

Declarative knowledge representation **techniques**: knowledge is expressed as sentences in some suitable formal language which are accessed by the procedures using this knowledge ⇒ separation between the explicit representation of knowledge and the processing for answering queries.

Advantages: increased versatility for performing complex tasks, changes can be easily incorporated (modularity)

Procedural techniques: knowledge is implicitly stored in a sequence of operations, manifested in the actual execution of the operations (i.e., directly as program code).

Advantages: minimizing the role of explicit representation and reasoning can yield more efficient systems

```
printColor(snow) :- write(''It's white.'').
printColor(grass) :- write(''It's green.'').
printColor(sky) :- write(''It's yellow.'').
```

vs.

```
printColor(X) :- color(X,Y),
     write(''It's '''), write(Y), write(''.'').
color(snow,white).
color(sky,yellow).
color(X,Y) :- madeof(X,Z), color(Z,Y).
madeof(grass,vegetation).
color(vegetation,green).
```

Successful agents should combine both declarative and procedural elements in their designs.

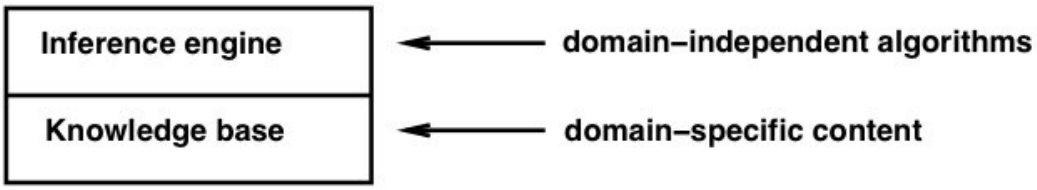
6.3. Knowledge-based agents

Central components of a knowledge-based agent:

- a **knowledge base**: set of sentences in a formal language
- **methods** to add new sentences and methods to query what is known
  - we use *Tell* and *Ask* as generic names for these tasks
  - both tasks may involve inference — i.e., deriving new sentences from old.

☞ In logical agents, answers to the Ask procedure is by means of logic!

Schematic architecture:



6.4. A simple knowledge-based agent

The agent must be able to:

- represent states, actions, etc.;
- incorporate new percepts;
- update internal representations of the world;
- deduce hidden properties of the world; – deduce appropriate actions.

Each time the agent program is called, it does three things:

1. It Tells the knowledge base what it perceives;

2. it Asks the knowledge base what action it should perform;

3. it records its choice with Tell and executes the action.

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time
  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

- MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time.
- MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time.
- MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed.

⟹ Details of the inference mechanisms are hidden inside TELL and ASK!

6.5. Entailment

Entailment means that one thing follows from another: knowledge base KB entails sentence α, symbolically $KB \models \alpha$, iff α is true in all worlds where KB is true.

E.g., the KB containing "Pooh laughs" and "Tigger laughs" entails "Either Pooh laughs or Tigger laughs".

E.g., x+y=4 entails 4=x+y.

☞ Entailment is a relationship between sentences (i.e., syntax) that is based on semantics.

6.1. Models

Semantics is defined in terms of interpretations, which are formally structured worlds with respect to which truth can be evaluated. We say that interpretation m is a model of a sentence α if α is true in m. M(α) is the set of all models of α.

Then $KB \models \alpha$ if and only if $M(KB) \subseteq M(\alpha)$.

E.g. KB = Pooh laughs and Tigger laughs α=Tiggerlaughs

6.2. Important semantical notions

➤ Two sentences are logically equivalent iff true in the same models:

α ≡ β if and only if α |= β and β |= α

➤ A sentence is valid if all interpretations are models of it.

➤ A sentence is satisfiable if it has some model.

➤ A sentence is unsatisfiable if it has no model.

➤ Writing ¬α for the negation of α (with the meaning that ¬α is true precisely when α is not true), we can state:

– α is valid if and only if ¬α is unsatisfiable;

– KB |= α if and only if KB ∪ {¬α} is unsatisfiable i.e., prove α by reductio ad absurdum.

6.3. Inference

KB ⊢*i α* ⇒ sentence α can be derived from KB by procedure i

Derivation from KB = a sequence of inference rule applications using axioms and elements of KB

Intuitively: Consequences of KB are a haystack; α is a needle. Entailment = needle in haystack; inference = finding it

Soundness: i is sound if whenever KB ⊢i α, it is also true that KB |= α

Completeness: i is complete if whenever KB |= α, it is also true that KB ⊢i α

6.4. Two fundamental logics

There are many different logics, designed for different purposes. Two logics are pre-eminent:

- propositional logic
- first-order logic (FOL), a.k.a. predicate logic

Propositional logic is simple, assuming that the world consists of facts which can be composed from atomic formulas using connectives

¬S (negation), S1 ∧ S2 (conjunction), S1 ∨ S2 (disjunction), S1 ⇒ S2 (implication), S1 ⇔ S2 (biconditional).

E.g. ¬A ⇒ (B∨C) states that if A is not the case, then one of B or C holds.

This formula may represent, e.g., the following sentence: If the car is not proceeding, then it is broken or out of gas.

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

Unlike natural language, propositional logic has, however, only very limited expressive power. E.g., the following argument (valid in natural language) cannot be adequately dealt with in propositional logic:

All superheroes are brave. Superman is a superhero. Therefore: Superman is brave.

In propositional logic, the three sentences would be formalized using atomic sentences A, B, C but A, B |= C does not hold ⇒ This is where FOL comes in!

FOL assumes that the world contains

- **Objects**: people, houses, numbers, theories, Superman, Tigger, colors, centuries, . . .
- **Relations**: red, round, bogus, prime, multistoried . . ., brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, . . .
- **Functions**: father of, best friend, addition, one more than, end of . . .

Syntax of FOL: Basic elements

Constants	$Superman, KingJohn, 2, \ldots$
Predicates	$Friend, >, \ldots$
Functions	$Sqrt, LeftLegOf, \ldots$
Variables	$x, y, a, b, \ldots$
Connectives	$\wedge \vee \neg \Rightarrow \Leftrightarrow$
Equality	$=$
Quantifiers	$\forall$ (universal quantifier) $\exists$ (existential quantifier)

Atomic sentences

Atomics sentence = *predicate(term₁,...,termₙ)* or *term₁ = term₂*

Term = function($term_1$,...,$term_n$) or constant or variable

E.g., *Friend(Pooh,Tigger) > (Length(LeftLegOf (Hulk)),Length(LeftLegOf (Spider-Man)))*

Complex sentences

Complex sentences are made from atomic sentences using connectives and quantifiers

$\forall xS$ ("for all x, S"), $\exists xS$ ("for some x, S")

E.g. *Archfiend(LexLuthor,Superman) ⇒ Fights(LexLuthor,Superman)*

>(1, 2) ∨ ≤(1, 2)
>(1, 2) ∧ ¬>(1, 2)
∀x∃y(Country(x) ⇒ Capitol(y,x))

6.5. Truth in first-order logic

Sentences are true with respect to a domain and an interpretation. The domain contains ≥ 1 objects (domain elements) for specifying relations among them. Interpretation specifies referents over the domain for

- constant symbols → objects
- predicate symbols → relations
- function symbols → functional relations

An atomic sentence *predicate(term$_1$, . . . , term$_n$)* is true iff the objects referred to by term$_1$, . . . , term$_n$ are in the relation referred to by predicate.

Consider the formula *Brother(Richard,John)* and the following interpretation:

Richard → Richard the Lionheart

John → the evil King John

Brother → the brotherhood relation

Under this interpretation, Brother(Richard, John) is true just in case Richard the Lionheart and the evil King John are in the brotherhood relation

Common mistakes to avoid

- Typically, $\Rightarrow$ is the main connective with $\forall$ as in:
  all S are P: $\forall x \ (S(x) \Rightarrow P(x))$
  - Common mistake: using $\wedge$ as the main connective with $\forall$:
    $\forall x \ At(x, Berkeley) \wedge Smart(x)$
    means "Everyone is at Berkeley and everyone is smart"
- Typically, $\wedge$ is the main connective with $\exists$ as in:
  some S are P: $\exists x \ (S(x) \wedge P(x))$
  - Common mistake: using $\Rightarrow$ as the main connective with $\exists$:
    $\exists x \ (At(x, Stanford) \Rightarrow Smart(x))$
    is true if there is anyone who is not at Stanford!

6.6. Some ambiguities

In natural language, "all S are P " would normally not be asserted if it is already known that S does not hold. Indeed, people would not consider "all S are P " true if S is false. =⇒ "all S are P" would in this sense be translated as

$\exists x S(x) \wedge \forall x(S(x) \Rightarrow P(x))$ rather than as $\forall x(S(x) \Rightarrow P(x))$.

Sometimes "all S are not-P " is understood as "not all S are P ". Examples:

- "All that glisters is not gold" (Shakespeare, Merchant of Venice).

- "All women are not gold diggers".

  ⇒ Translations would be of form $\neg\forall x(S(x) \Rightarrow P(x))$ but not of form $\forall x(A(x) \Rightarrow \neg P(x))$.

The indefinite article "a" or "an" has sometimes different meaning:

- "A child needs affection." ⇒ $\forall x(C(x) \Rightarrow A(x))$
- "A man climbed the Mount Everest." =⇒ $\exists x(M(x) \wedge E(x))$

Also, the meaning of the expression "any" depends on the context:

- When an any-expression stands by itself, it has the same force as "all".
- But when an any-expression D is put into contexts ¬D or D ⇒ E, the meaning of "any" normally changes from "all" to "some".

Examples:

- "I would do that for anyone." =⇒ $\forall xA(x)$
- "I wouldn't do that for anyone." ⇒ $\neg\exists xA(x)$
- "If any man is godfearing, he is just." ⇒ $\forall x(G(x) \Rightarrow J(x))$
- "If any man is just, Aristides is just." ⇒ $(\exists xJ(x)) \Rightarrow J(a)$
- "If Superman is a villain, then any man is a villain." ⇒ $V(s) \Rightarrow \forall xV(x)$,

6.7. The story so far

Logical agents apply inference to a knowledge base to derive new information and make decisions. Basic concepts of logic:

- **syntax**: formal structure of sentences
- **semantics**: truth of sentences with respect to models
- **entailment**: necessary truth of one sentence given another
- **inference**: deriving sentences from other sentences
- **soundness**: derivations produce only entailed sentences
- **completeness**: derivations can produce all entailed sentences

First-order logic:

- objects and relations are semantic primitives
- syntax: constants, functions, predicates, equality, quantifiers

⇒ Increased expressive power: sufficient to capture many aspects of natural language

6.8. Ontological engineering

Ontological engineering ⇒ create representations of general concepts like actions, time, physical objects, and beliefs.

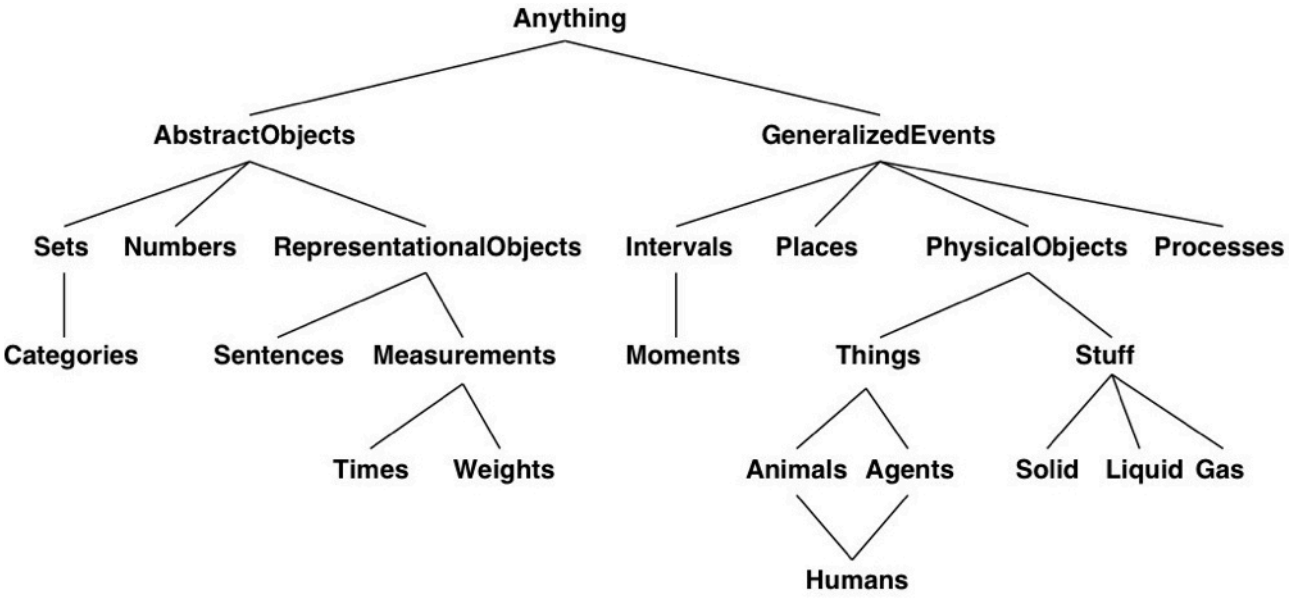
Initial disclaimer: we cannot represent everything in the world! But we will leave placeholders where new knowledge for any domain can fit in. E.g., we can define the concept of an physical object, but the details of different types of objects—like robots, televisions, books, DVD-players, etc.—can be filled in later. In particular, the principal difficulty is that almost all generalizations have exceptions, or hold only to a degree. E.g.: "tomatoes are red" is a useful rule, but some tomatoes are green, yellow, or orange.

Other formalisms than FOL have been designed to adequately handle such patterns:

- nonmonotonic logics (like default logic or circumscription)
- probabilistic reasoning

6.9. Upper ontology

The general framework of concepts is called an upper ontology, because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them



6.10. Categories and objects

The organization of objects into categories is a vital part of knowledge representation. Categories serve to make predictions about objects once they are classified.

- One infers the presence of certain objects from perceptual input,
- infers category membership from the perceived properties of the objects,
- and then uses category information to make predictions about such objects.

E.g., given an object with green, mottled skin, large size and ovoid shape, one can infer that it is an watermelon. From this, one can further infer that it is useful for fruit salad.

There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate *Basketball(b)*, or we can reify the category as an object, *Basketballs*.

> **reification**: from the Latin res, meaning thing, object ⇒ reification = "thing-ification"

We could then say *Member(b, Basketballs)*, which we will abbreviate as $b \in Basketballs$, to say that *b* is a member of the category of *basketballs*. We say *Subset(Basketballs, Balls)*, abbreviated as $Basketballs \subset Balls$, to say that *Basketballs* is a subcategory of *Balls*. We will use subcategory, subclass, and subset interchangeably.

6.11. Inheritance and Taxonomy

Categories serve to organize and simplify the knowledge base through inheritance. E.g., if we say that all instances of the category Man are mortal, and if we assert that Greeks is a subclass of Man, then we know that all Greeks are mortal. ⇒ Individual Greeks (like Aristotle) inherit the property of mortality. Subclass relations organize categories into a taxonomy, or taxonomic hierarchy.

- Taxonomies have a centuries-long tradition in technical fields.
- E.g., systematic biology aims to provide a taxonomy of all living and extinct species; library science has developed a taxonomy of all fields of knowledge; etc.

6.12. FOL and categories

FOL makes it easy to state facts about categories:

- An object is a member of a category:

  $BBg \in Basketballs$

- A category is a subclass of another category:

  $Basketballs \subset Balls$

- All members of a category have some property:

  $x \in Basketballs \Rightarrow Round(x)$

- Members of a category can be recognized by certain properties:

  $Orange(x) \wedge Round(x) \wedge Diameter(x)=9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$

- A category as a whole has some properties:

  $Dogs \in DomesticatedSpecies$

Two or more categories are disjoint iff they have no members in common:

$Disjoint(s) \Leftrightarrow \forall c_1,c_2 \; ((c_1 \in s \wedge c_2 \in s \wedge c_1 != c_2) \Rightarrow \text{Intersection}(c1, c2) = \{\})$

If members of a given category s constitute all elements of another category c, we have an exhaustive decomposition:

$ExhaustiveDecomposition(s,c) \Leftrightarrow \forall i\ (i \in c \Leftrightarrow \exists c_2\ c_2 \in s \wedge i \in c_2)$

A disjoint exhaustive decomposition is a partition:

$Partition(s,c) \Leftrightarrow (Disjoint(s) \wedge ExhaustiveDecomposition(s,c))$

6.13. Physical composition

Objects can be part of other objects ⇒ This can be expressed using the *PartOf* predicate.

For instance: *PartOf(Gramatneusiedl,Austria); PartOf(Austria,CentralEurope); PartOf (CentralEurope,Europe); PartOf (Europe,Earth)*.

The PartOf predicate is transitive and reflexive: $PartOf(x,y) \wedge PartOf(y,z) \Rightarrow PartOf(x,z)$ ⇒ We can conclude *PartOf (Gramatneusiedl, Earth)*.

It is also useful to define composite objects with definite parts but no particular structure.

Example: we might want to say "The apples in this bag weigh one kilogram" - might be tempted to ascribe the weight to the set of apples in this bag, but this would be a mistake because sets have no weight. ⇒ Introduce new concept: a bunch. E.g., if the apples are Apple1, Apple2, Apple3, then BunchOf ({Apple1, Apple2, Apple3}) denotes the composite object with the three apples as parts (not elements) can use the bunch as a normal, unstructured object.

We can define BunchOf by logical minimization in terms of PartOf :

1. each element of s is part of BunchOf (s):

$\forall x\ x \in s \Rightarrow PartOf(x,BunchOf(s))$.

2. BunchOf (s) is the smallest object satisfying Condition 1:

$\forall y\ [\forall x\ x \in s \Rightarrow PartOf(x,y)] \Rightarrow PartOf(BunchOf(s),y)$

(i.e., BunchOf (s) must be part of any object that has all the elements of s as parts).

6.14. Substances and objects

A significant portion of reality seems to defy individuation—the division into distinct objects.

Example: Suppose I have some butter and some aardvark in front of me. We can say there is one aardvark but there is no obvious number of "butter-objects", as any part of a butter-object is also a butter object.

We call elements defying individuation stuff. ☞ The distinction things vs. stuff corresponds to the following distinction from linguistics:

- count nouns: aardvarks, cars, rockets, theorems, . . .
- mass nouns: butter, water, energy, . . .

To represent stuff in our ontology, we need to have as objects at least the gross "lumps" of stuff we interact with. E.g., we may recognize a lump of butter as the same butter that was left on the table yesterday. ⇒ In this sense, it is an object like the aardvark, say we call it Butter3.

We also define the category Butter —its elements are all those things of which we might say "It's butter", including Butter3.

- –Any part of a butter-object is also a butter-object:

  $x \in Butter \wedge PartOf(y,x) \Rightarrow y \in Butter$

We can say that butter is yellow, melts at around 30 degrees, is less dense than water, has high fat content. But butter has no particular shape, size, or weight.

Important distinction:

- **intrinsic properties**: belong to the very substance of the object, rather than to the object as a whole.
  - When you cut a substance in half, the pieces retain the same set of intrinsic properties—things like density, boiling point, flavor, color, ownership, etc.
- **extrinsic properties**: those which are not retained under subdivision. – Examples: weight, length, shape, function, etc.

We can therefore say: a substance, or a mass noun, is a class of objects that includes in its definition only intrinsic properties and a count noun is a class that includes some extrinsic property in its definition.

Consequently:

- The category Stuff is the most general substance category, specifying no intrinsic properties.
- The category Thing is the most general discrete object category, specifying no extrinsic properties.

☞ All physical objects belong to both categories, so the categories are co- extensive—they refer to the same entities.

6.1. Actions and situations

Reasoning about the results of actions is central to the operation of a knowledge-based agent. One way represent actions with FOL is to use axioms expressing things like

$\forall$ t, such-and-such is the result at t + 1 of doing action at t.

Situation calculus

Involves the following ontology:

- **Actions**: logical terms like Watch(Movie) or Turn(Right).
- **Situations**: logical terms consisting of the initial situation (usually called S0) and all situations that are generated by applying an action to a situation.
  - The function Result(a,s) (sometimes also called Do) names the situation that results when action a is executed in situation s.
- **Fluents**: functions and predicates that vary from one situation to the next.
  - By convention, the situation is always the last argument of a fluent.

- E.g., ¬Holding (B1, S0) expresses that an agent does not hold object B1 in the initial situation S0; Age(Superman,S0) refers to the age of Superman in S0.

- **Atemporal** or eternal predicates and functions: e.g., the predicate Bananas(B1) or the function LeftLegOf (Superman).

In addition to single actions, it is also helpful to reason about action sequences. We define the results of sequences in terms of the results of individual actions:

1. Executing an empty sequence leaves the situation unchanged: Result([],s) = s.

2. Executing a nonempty sequence is the same as executing the first action and then executing the rest in the resulting situation: Result([a|seq],s) = Result(seq,Result (a,s)).

Basic tasks agents should be able to perform:

- projection task: the agent should be able to deduce the outcome of a given sequence of actions;

- planning task: the agent should be able to find a sequence of actions that achieves a desired effect.

Example — Monkey and Bananas

Consider a monkey who wants to grab some bananas.

We assume the monkey lives in a two-dimensional world. ⟨x, y⟩ denotes the position of the monkey (x, y are natural numbers). Suppose the monkey is at position ⟨1, 1⟩ and the bananas are at position ⟨1, 2⟩. ⇒ The aim is to have the bananas in ⟨1, 1⟩.

The fluent predicates are At (o, x, s) and Holding (o, s). ⇒ The initial knowledge base thus includes: At(Monkey, ⟨1, 1⟩, S0) ∧ At(B1, ⟨1, 2⟩, S0).

We also need to specify what is not true in S0: At(o,x,S0) ⇔ [(o = Monkey∧x = ⟨1,1⟩)∨(o = B1∧x = ⟨1,2⟩]. ¬Holding(o, S0).

Furthermore, we state that B1 are bananas and that ⟨1, 1⟩ and ⟨1, 2⟩ are adjacent: Bananas (B1)∧Adjacent(⟨1,1⟩,⟨1,2⟩)∧Adjacent(⟨1,2⟩,⟨1,1⟩).

The action predicates are Go(x,y), Grab(b), and Release(b). We would like to be able to prove that the monkey achieves its aim by going to ⟨1,2⟩, grabbing the bananas, and returning to ⟨1,1⟩, i.e., to prove:

At(B1, ⟨1, 1⟩, Result([Go(⟨1, 1⟩, ⟨1, 2⟩), Grab(B1), Go(⟨1, 2⟩, ⟨1, 1⟩)], S0)).

More interesting is the possibility of constructing a plan to get the bananas by answering the query "what sequence of actions results in the bananas being at ⟨1, 1⟩?":

∃seq At(B1,⟨1,1⟩,Result(seq,S0)). ⇒ More information has to be in the knowledge base to achieve this!

Describing actions

In the simplest version of the situation calculus, each action is described by two axioms:

- **possibility axiom**: describes when it is possible to execute the action Preconditions ⇒ Poss (a,s). (Poss(a,s): it is possible to execute action a in situation s.)
- **effect axiom**: describes what happens when a possible action is executed Poss(a,s) ⇒ Changes that result from taking action.

Continuing the monkey-and-bananas scenario, we assume that

- s ranges over situations,
- a over actions,
- o over objects,
- b over bananas, and
- x, y over locations.

The following possibility axioms state that the monkey – can go between adjacent locations, – grab bananas in the current location, and – release those it is holding:

$$At(Monkey,x,s) \wedge Adjacent(x,y) \Rightarrow Poss(Go(x,y),s).$$
$$Bananas(b) \wedge At(Monkey,x,s) \wedge At(b,x,s) \Rightarrow Poss(Grab(b),s).$$
$$Holding(b,s) \Rightarrow Poss(Release(b),s).$$

The following effect axioms state that, if an action is possible, then certain properties (fluents) will hold in the situation that results from executing the action:

- going from x to y results in being at y,
- grabbing the bananas results in holding them; and
- releasing the bananas results in not holding them:

$$Poss(Go(x,y),s) \Rightarrow At(Monkey,y,Result(Go(x,y),s)).$$
$$Poss(Grab(b),s) \Rightarrow Holding(b,Result(Grab(b),s)).$$
$$Poss(Release(b),s) \Rightarrow \neg Holding(b,Result(Release(b),s)).$$

Frame problem

The axioms given so far do not suffice for being able to prove that our plan achieves the goal. The problem is that the effect axioms say what changes but do not say what stays the same!

Representing all the things that stay the same is called the frame problem (first recognized by McCarthy and Hayes in 1969).

One approach to deal with the frame problem is to write explicit frame axioms that do say what stays the same.

– E.g., the monkey's movements leave other objects stationary unless they are held:

$$At(o,x,s) \wedge (o = Monkey) \wedge \neg Holding(o,s) \Rightarrow At(o, x, Result(Go(y, z), s)).$$

Observe: If there are F fluent predicates and A actions, then we will need O(AF) frame axioms. On the other hand, if each action has at most E effects (with E typically being much less than F), then we should be able to represent things with a (much smaller) knowledge base of size O(AE). ⇒ This is the representational frame problem.

We can solve the representational frame problem by means of the method of successor-state axioms (Reiter, 1991).

General form:

> Action is possible ⇒ (Fluent is true in result state ⇔ Effect of action made it true ∨ It as true before and action left it unaltered).

Here, we are specifying the truth value of each fluent in the next state as a function of the action and the truth value in the current state. Next state is completely specified from the current state and hence no additional frame axioms are needed! The total size of the axioms is O(AE) literals (thus, solving the representational frame problem): each of the E effects of each of the A actions is mentioned exactly once.

Ramification problem

We need to say that an implicit effect of the monkey moving from x to y is that any bananas it is carrying will move too - as will any spiders in the bananas, any bacteria in the spiders, any molecule in the bacteria, etc.⇒ Dealing with such implicit effects is called the ramification problem.

In our example, we can deal with the ramification problem by writing a more general successor-state axiom for At (subsuming the earlier one).

The new axiom states that – an object o is at position y if o went to y and o is the monkey or something the monkey was holding – or if o was already at y and o did not go elsewhere, with o being the monkey or something the monkey was holding:

> $Poss(a,s) \Rightarrow [At(o,y,Result(a,s)) \Leftrightarrow (a = Go(x, y) \wedge (o = Monkey \vee Holding(o, s))) \vee (At(o,y,s) \wedge \neg \exists z\ (y = z \wedge a = Go(y,z) \wedge (o = Monkey \vee Holding(o, s))))]$.

Unique names

One further technicality is required: we need to be able to prove nonidentities. The simplest kind of nonidentity is between constants—e.g., like stating Monkey= B1 ⇒ In general FOL, it is not assumed that different constants mean different objects!

Two possibilities to achieve nonidentity between constants:

1. We add an axiom into the knowledge base stating a disequality for every pair of constants (this is called the unique-names axiom); or

2. when nonidentity is assumed by the inference mechanism directly (rather than putting it into the knowledge base), it is called the unique-names assumption.

We also need to state disequalities between action terms—e.g., Go(⟨1, 1⟩, ⟨1, 2⟩) is a different action than Go(⟨1, 2⟩, ⟨1, 1⟩) or Grab(B1).

This is achieved by adding the following formulas:

1. For each pair of action names A and B, we assume $A(x1,...,xm) \neq B(y1,...,yn)$.

2. Two action terms with the same action name refer to the same action only if they involve all the same objects: $A(x1,...,xm)=A(y1,...,ym) \Leftrightarrow x1 = y1 \wedge ... \wedge xm = ym$.

⇒ These two kinds of formulas are collectively referred to as the unique-action axioms.

6.2. Discussion

A problem related to the frame problem and the ramification problem is the qualification problem—what are all the necessary conditions such that an action succeeds.

☞ There is no complete solution for this problem!

The situation calculus works well for single agents performing instantaneous, discrete actions, but for actions having duration with a possible overlap, the event calculus (Kowalski and Sergot, 1986) is better suited (which is based on time points rather than on situations). Also, Allen (1983) introduced the interval algebra for that purpose.

Systems for organizing and reasoning with categories include:

- semantic networks (Minsky, 1975; and tracing back to Peirce, 1909);

- description logics (going back to the KL-One system (Schmolze and Lipkis, 1983)).

7. Chapter 11 (Planning)

The task of coming up with a sequence of actions that will achieve a goal is called planning. This chapter is concerned primarily with scaling up to complex planning problems that defeat the approaches we have seen so far.

Situation calculus — FOL based approach to planning. Now, we are primarily concerned with approaches that scale up to more complex planning problems.

For this chapter, we consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called classical planning environments.

7.1. The language of planning problems

Key issues of a good planning language: expressive enough to describe a wide variety of problems but restrictive enough to allow efficient algorithms.

We start with STRIPS (Fikes and Nilsson, 1971), a basic representation language of classical planners. "STRIPS" stands for "Stanford Research Institute Problem Solver" and was designed as the planning component of the software for the Shakey robot project at SRI.

7.2. The syntax of STRIPS

consists of the following items:

Representation of states: Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals.

Literals can be propositional or first-order, but first-order literals must be ground and function-free. For instance, $*$ Rich $\wedge$ InJail may represent the state of some person, while At(x, y) or At (president(Russia), Kremlin) are not allowed. Furthermore, the closed-world assumption is used, meaning that any condition not mentioned in a state is assumed false.

Representation of goals: A goal is a partially specified state, represented as a conjunction of positive ground literals, such as Rich $\wedge$ Famous or At(P2,LakeTahoe). A propositional state s satisfies a goal g if s contains all the atoms in g (and possibly others). E.g., the state Rich $\wedge$ Famous $\wedge$ InJail satisfies the goal Rich $\wedge$ Famous.

Representation of actions: An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. E.g., an action for flying a plane from one location to another is:

$$\begin{aligned} &Action(Fly(p, from, to),\\ &\quad \text{PRECOND: } At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)\\ &\quad \text{EFFECT: } \neg At(p, from) \wedge At(p, to)) \end{aligned}$$

More precisely: this is actually an example of an action schema, $*$ representing a number of different actions that can be derived by instantiating the variables p, from, and to to different constants.

In general, an action schema consists of three parts:

1. The **action name** and **parameter list**—e.g., Fly(p, from, to).
2. The **precondition**: a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. ☞ Any variables in the precondition must also appear in the action's parameter list.
3. The **effect**: a conjunction of function-free literals describing how the state changes when the action is executed.
   - A positive literal P in the effect is true in the state resulting from the action, while a negative literal ¬P results in P being false.
   - Variables in the effect must also appear in the action's parameter list.

☞ Some planning systems divide the effect into the add list for positive literals and the delete list for negative literals.

7.1. STRIPS—semantics

An action is applicable in any state that satisfies the preconditions; otherwise, the action has no effect. For a first-order action schema, establishing applicability involves a sub- stitution for the variables in the precondition.E .g., suppose the current state is described by

$$At(P1,JFK)\wedge At(P2,SFO)\wedge Plane(P1)\wedge Plane(P2)\wedge Airport(JFK)\wedge Airport(SFO).$$

This state satisfies the precondition

$$At(p,from)\wedge Plane(p)\wedge Airport(from)\wedge Airport(to)$$

of action schema Fly(p,from,to) with substitution {p/P1,from/JFK,to/SFO}. ⇒ The concrete action Fly(P1,JFK,SFO) is applicable.

Starting in a state s, the result of executing an applicable action a is a state s' that is the same as s except that

- any positive literal P in the effect of a is added to s' and
- any negative literal ¬P in the effect results in removing P from s'.

Thus, for our flight example, after executing Fly(P1,JFK,SFO), the current state

$$At(P1,JFK)\wedge At(P2,SFO)\wedge Plane(P1)\wedge Plane(P2)\wedge Airport(JFK)\wedge Airport(SFO).$$

becomes

$$At(P1, SFO)\wedge At(P2, SFO)\wedge Plane(P1)\wedge Plane(P2)\wedge Airport(JFK)\wedge Airport(SFO).$$

Remarks:

If a positive effect is already in s it is not added twice, and if a negative effect is not in s, then that part of the effect is ignored.

The definition of the semantics of STRIPS embodies the so-called STRIPS assumption: every literal not mentioned in the effect remains unchanged. ⇒ In this way, STRIPS avoids the representational frame problem.

Finally, a solution for a planning problem is an action sequence that, when executed in the initial state, results in a state that satisfies the goal.

7.2. Language variants

For some real-world domains, STRIPS is not sufficient ⇒ many language variants have been developed as a consequence. An important such variant is ADL (Pednault, 1986), the Action Descrip- tion Language. In ADL, the Fly action can be written as follows:

$$\begin{aligned} &Action(Fly(p : Plane, from : Airport, to : Airport),\\ &\quad \textsc{Precond}: At(p, from) \wedge from \neq to\\ &\quad \textsc{Effect}: \neg At(p, from) \wedge At(p, to)) \end{aligned}$$

Note:

- ADL allows typing—e.g., the notation p : Plane is an abbreviation for Plane (p).
- The precondition from != to expresses that a flight cannot be made from an airport to itself. ⇒ This could not be expressed succinctly in STRIPS!

7.3. STRIPS vs. ADL

STRIPS	ADL
Only positive literals in states: $Rich \wedge InJail$	Positive and negative literals in states: $\neg Poor \wedge \neg Free$
Closed-World Assumption: Unmentioned literals are false	Open-World Assumption Unmentioned literals are unknown
Effect $P \wedge \neg Q$ means add P and delete Q	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q
Only ground literals in goals: $Rich \wedge InJail$	Quantified variables in goals: $\exists x \;\; At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions:	Conditional effects are allowed: $when P : E$ means E is an effect only if P is satisfied
No support for equality	Equality is built in
No support for types	Variables can have types, as in $(p : Plane)$

Remarks:

The various planning formalisms used in AI have been systematized within a standard syntax called the Planning Domain Definition Language, or PDDL ☞ PDDL includes sublanguages for STRIPS and ADL. STRIPS and ADL are adequate for many real-world domains, but they have some significant restrictions. An important one is that ramifications of actions cannot be represented in a natural way. Indirect actions, like dust particles moving with airplanes need to be represented as direct effects ⇒ it would be more natural if these changes could be derived from the location of the plane. Also, classical planning systems do not attempt to address the qualification problem.

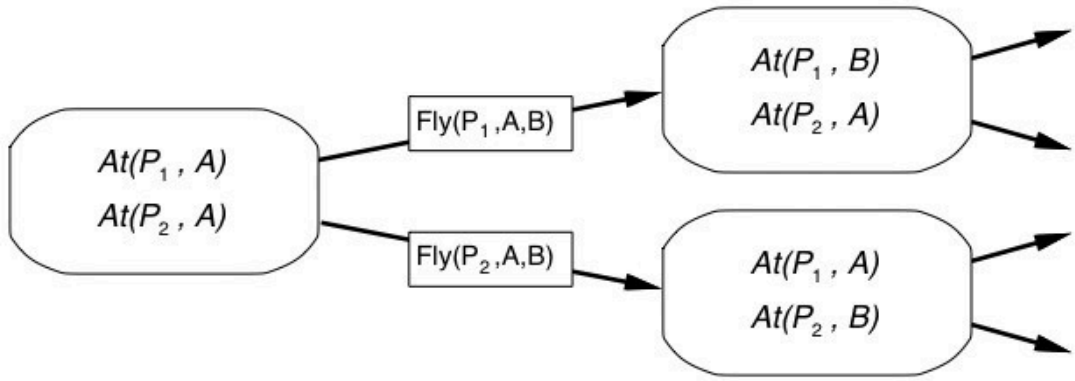
7.4. Planning with state-search

Now we turn our attention to planning algorithms. The most straightforward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either (a) forward from the initial state or (b) backward from the goal. We can also use the explicit action and goal representations to derive effective heuristics automatically

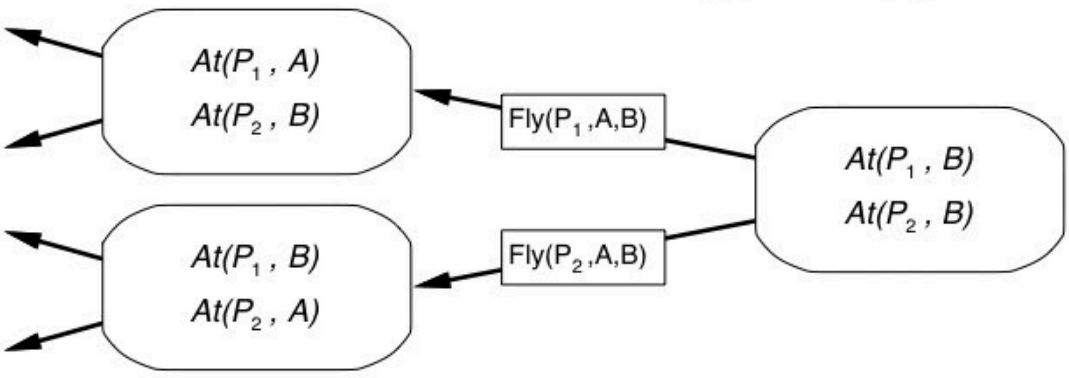
Two possibilities:

- forward state-space search (or progression planning): from initial state to goal;
- backward state-space search (or regression planning): from goal to initial state.

(a)



(b)



(a) Progression planning (forward state-space search)

We start in the problem's initial state, considering sequences of actions until we find a sequence that reaches a goal state. The formulation of planning problems as state-space search problems is as follows:

- The initial state of the search is the initial state of the planning problem.
  - Each state will be a set of positive ground literals;
  - literals not appearing are false.
- The actions that are applicable to a state are all those whose preconditions are satisfied.
  - The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals.
  - In case of FOL, we must apply the unifier from the preconditions to the effect literals.
- The goal test checks whether the state satisfies the goal of the planning problem.
- The step cost of each action is typically 1. ☞ Allowing different costs for different actions could be easily realised, but this is seldom done for STRIPS planners.

Note: in the absence of function symbols, the state space of a planning problem is finite ⇒ any complete graph search algorithm (like $A*$) yields a complete planning algorithm!

(b) Regression planning (backward state-space search)

The main advantage of backward search is that it allows to consider only relevant actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For instance:

Consider the cargo problem with 20 pieces of cargo, having the goal

$$At(C1,B) \wedge At(C2,B) \wedge ... \wedge At(C20,B)$$

Seeking actions having, e.g., the first conjunct as effect, we find only Unload(C1,p,B) as relevant. This action will work only if its preconditions are satisfied. ⇒ any predecessor state must include the preconditions

$$In(C1,p) \wedge At(p,B)$$

Moreover, the subgoal At(C1, B) should not be true in the predecessor state ⇒ The predecessor state description is $In(C1,p) \wedge At(p,B) \wedge At(C2,B) \wedge ... \wedge At(C20,B).$

Besides insisting that actions achieve some desired goal, they should not undo any desired literals -> Actions satisfying this restriction are called consistent. E.g., Load(C2,p,B) would not be consistent with the current goal as it would negate the literal At(C2, B).

We can now describe the general process of constructing predecessors for backward search.

- Given a goal description G, let A be an action that is relevant and consistent.
- The corresponding predecessor is as follows:
  - Any positive effects of A that appear in G are deleted.
  - Each precondition literal of A is added, unless it already appears.

Any standard search algorithm can be used to carry out the search. ☞ In the first-order case, satisfaction might require a substitution for variables in the predecessor description.

7.5. Partial-order planning

Forward and backward state-space search are particular forms of totally ordered plan search. They explore only strictly linear sequences of actions and do not take advantage of problem decomposition. Any planning algorithm that can place two actions into a plan without specifying which comes first is called a partial-order planner.

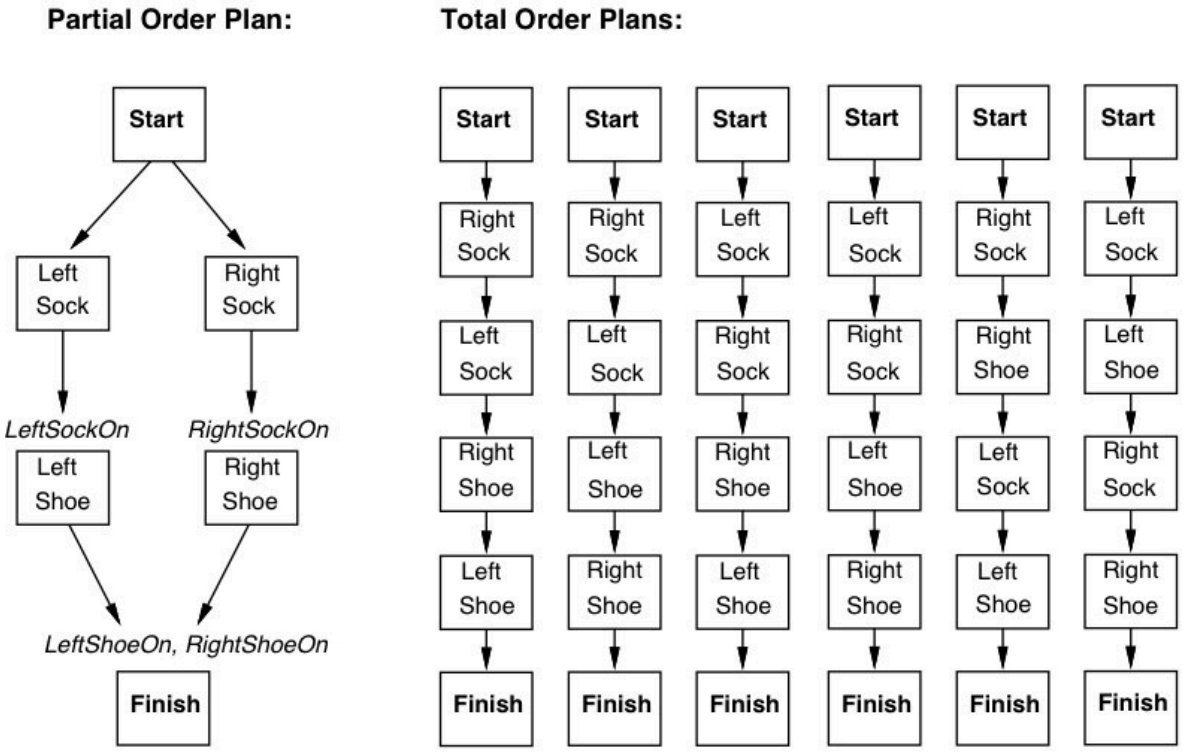
Consider a simple example of putting on a pair of shoes:

$$
\begin{array}{l}
Init() \\
Goal(RightShoeOn \wedge LeftShoeOn) \\
Action(RightShoe, \textsc{Precond} : RightSockOn, \textsc{Effect} : RightShoeOn) \\
Action(RightSock, \textsc{Effect} : RightSockOn) \\
Action(LeftShoe, \textsc{Precond} : LeftSockOn, \textsc{Effect} : LeftShoeOn) \\
Action(LeftSock, \textsc{Effect} : LeftSockOn)
\end{array}
$$

A partial-order planner should come up with the two-action sequences

1. [RightSock,RightShoe] to achieve the first conjunct of the goal and
2. [LeftSock,LeftShoe] for the second conjunct.

Then, the two sequences can be combined to yield the final plan. In doing so, the planner manipulates the two subsequences independently.



Partial-order planning can be implemented as a search in the space of partialorder plans:

1. We start with an empty plan.
2. Then we consider ways of refining the plan until we come up with a complete plan that solves the problem.
3. The actions in this search are not actions in the world but actions on plans:
   - adding a step to the plan;
   - imposing an ordering that puts one action before another; and so on.

⇒ We will define the POP algorithm for partial-order planning (as an instance of a search problem).

Components

Each plan has the following four components:

1. **a set of actions**; The set of actions constitutes the elements for making up the steps of the plan. The actions are taken from the set of actions in the planning problem. The empty plan contains just the Start and Finish actions.
   - Start has no preconditions and has as its effect all the literals in the initial state of the planning problem.
   - Finish has no effects and has as its preconditions the goal literals of the planning problem.
2. **a set of ordering constraints**; An ordering constraint is a pair of actions of the form A ≺ B, read as "A before B". A ≺ B means that action A must be executed sometime before

action B, but not necessarily immediately before. The ordering constraints must describe a proper partial order. Any cycle, like A ≺ B and B ≺ A, represents a contradiction ⇒ a ordering constraint cannot be added to the plan if it creates a cycle.

3. **a set of causal links**; A causal link between two actions A and B in the plan is an expression p of form *A −(p)→ B*, read as "A achieves p for B". E.g., the causal link - *RightSock -(RightSockOn)→ RightShoe* asserts that RightSockOn is an effect of the RightSock action and a precondition of RightShoe. It also asserts that RightSockOn must remain true from the time of action RightSock to the time of action RightShoe. In other words, the plan may not be extended by adding a new action C that conflicts with the causal link.

4. **a set of open preconditions**. An action C conflicts with A −(p)→ B if (1) C has the effect ¬p and (2) C could (according to the ordering constraints) come after A and before B. A precondition is open if it is not achieved by some action in the plan. Planners will work to reduce the set of open preconditions to the empty set, without introducing a contradiction.

For instance, the final plan in the shoe-and-sock example has the following components (omitting the ordering constraints that put every other action after Start and before Finish):

Actions: $\{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}$

Orderings: $\{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}$

Links: $\{RightSock \xrightarrow{RightSockOn} RightShoe, LeftSock \xrightarrow{LeftSockOn} LeftShoe,$
$RightShoe \xrightarrow{RightShoeOn} Finish, LeftShoe \xrightarrow{LeftShoeOn} Finish\}$

Open preconditions: $\{\}$

Solutions

We define a consistent plan as a plan in which has (1) no cycles in the ordering constraints and (2) no conflicts with the causal links. A solution is a consistent plan with no open preconditions. Every linearisation of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state. We can extend the notion of "executing a plan" from total-order plans to partial-order plans: A partial-order plan is executed by repeatedly choosing any of the possible next actions.

7.6. The POP algorithm

The initial plan contains

- Start and Finish,
- the ordering constraint Start ≺ Finish,
- no causal links, and
- all the preconditions in Finish as open preconditions.

The successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p.

Consistency is enforced as follows:

1. The causal link A $-(p)\rightarrow$ B and the ordering constraint A $\prec$ B are added to the plan. Action A may be an existing action in the plan or a new one. If it is new, add it to the plan and also add Start $\prec$ A and A$\prec$Finish.

2. We resolve conflicts between the new causal link and all existing actions and between action A (if it is new) and all existing causal links. A conflict between A $-(p)\rightarrow$ B and C is resolved by adding B $\prec$ C or C $\prec$ A. We add successor states for either or both if they result in consistent plans.

The goal test checks whether a plan is a solution to the original planning problem. Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

7.7. Summary

Planning systems are problem-solving algorithms that operate on explicit propositional or first-order representations of states and actions.

STRIPS language describes actions in terms of their preconditions and effects and describes the initial and goal states as conjunctions of positive literals. The ADL language relaxes some of these constraints, allowing disjunction, negation, and quantifiers.

State-space search can operate in the forward direction (progression) or the backward direction (regression). Partial-order planning algorithms explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to achieve each subgoal. They are particularly effective on problems amenable to a divide-and-conquer approach.

8. Chapter 13+14 (Uncertainty)

8.1. Acting under uncertainty

The general problem is that representing knowledge in classical logic makes the assumption to have complete knowledge about the considered domain. When a logical agent knows enough facts about its environment, it can derive plans which are guaranteed to work. But agents almost never have access to the whole truth about their environment.

⇒ Agents must act under uncertainty. Probabilistic reasoning is one of several approaches in AI to formalize reasoning under uncertainty.

8.2. Basic probability notation

The basic element of the language of probability theory is the random variable. Random variables are denoted by capitalized names. Each random variable has a domain of values ⇒ We will use lowercase letters for the names of values.

Atomic events

Atomic event = an assignment of values to all the random variables the world is composed of—i.e., a complete specification of the state of the world. E.g., if the world consists of only the Boolean variables Cavity and Toothache, then there are just four distinct atomic events—Cavity = true ∧ Toothache = false is one of these. Atomic events have the following important properties:

- They are mutually exclusive—at most one can actually be the case. E.g., cavity ∧ toothache and cavity ∧ ¬toothache cannot both be the case.
- The set of all possible atomic events is exhaustive—at least one must be the case. That is, the disjunction of all atomic events is logically equivalent to true .

Joint probability distribution

Let V1,...,Vn be some random variables and Ω the set of all atomic events over V1,...,Vn. A joint probability distribution (JPD) of V1, . . . , Vn is a mapping P assigning each atomic event ω = (V1 = v1,...,Vn = vn) a number P(ω) such that

1. $0 \leq P(\omega) \leq 1$;
2. SUMME[ω∈Ω] (P(ω)) = 1.

We also write P(V1,...,Vn) to refer to the JPD of V1,...,Vn. Accordingly, P(v1,...,vn) stands for P (V1 = v1,...,Vn = vn).

For a set V = {V1,...,Vn} of variables, P(V) stands for P(V1,...,Vn). More generally, for V = {V1, . . . , Vn} and W = {W1, . . . , Wm}, P(V, W)

denotes P(V1,...,Vn,W1,...,Wm). Vectors V = v refer to atomic events V1 = v1,...,Vn = vn.

8.3. Reduced Probabilities—Marginalisation

Extracting the distribution over some subset of a given set of variables is called marginalisation. That is, we are given P (V_1, ..., V_n) and we want to compute , say, P (V_{i1}, ..., V_{ik}) , where $1 \leq i_j \leq n$.

Special case k = 1: prior probability $P(V_i = v_i)$. Computation:

$$\mathrm{P}(V_i = v_i) = \sum_{V_i = v_i} \mathrm{P}(V_1, \ldots, V_n)$$

– e.g., P(cavity) = P(cavity, toothache) + P(cavity, ¬toothache).

In general:

$$\mathrm{P}(V_{i_1} = v_{i_1}, \ldots, V_{i_k} = v_{i_k}) = \sum_{V_{i_1} = v_{i_1}, \ldots, V_{i_k} = v_{i_k}} \mathrm{P}(V_1, \ldots, V_n)$$

8.4. Conditional probability

Conditional probability $P(V_i = v_i | V_j = v_j)$: gives probability that $V_i = v_i$ by given evidence $V_j = v_j$ —i.e., gives probability that $V_i = v_i$ if all the agent knows is $V_j = v_j$.

Conditional probability can be computed using the equation

$$\mathrm{P}(V_i | V_j) = \frac{\mathrm{P}(V_i, V_j)}{\mathrm{P}(V_j)}. \quad \boxed{P(B|A) = \frac{P(A \wedge B)}{P(A)}}$$

This holds also for several variables, e.g., for Boolean variables B, T, A, W

$$\mathrm{P}(b, \neg t | \neg a, w) = \frac{\mathrm{P}(b, \neg t, \neg a, w)}{\mathrm{P}(\neg a, w)}$$

8.5. Product rule

From the basic relation between prior and conditional probability, we get the following product rule:

$$\mathrm{P}(V_1, \ldots, V_n) = \prod_{i=1}^{n} \mathrm{P}(V_i | V_{i+1}, \ldots, V_n).$$

Factorization is independent of order! Important for applications since conditional probabilities can be obtained easier in general than entries for JPD for atomic events. The general, conditionalised, form of the product rule is:

$$\mathrm{P}(V_1, \ldots, V_n | \mathbf{V}) = \prod_{i=1}^{n} \mathrm{P}(V_i | V_{i+1}, \ldots, V_n, \mathbf{V})$$

8.6. Bayes rule

Reconsider two instances of the product rule:

P(A, B) = P(A|B)P(B)

P(A, B) = P(B|A)P(A)

Equating the two right-hand sides and dividing by P(Vj), we get

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

Proof:

$$\frac{P(A \mid B)P(B)}{P(A)} = \frac{\frac{P(A \wedge B)}{P(B)}P(B)}{P(A)} = \frac{P(A \wedge B)}{P(A)} = P(B \mid A)$$

8.7. Conditional Independence

Let V be a variable, and V_i, E sets of variables. Then, V is conditionally independent of V_i by given evidence E iff P(V |V_i, E) = P(V |E)

Intuitively: Additional information of Vi by given evidence E yields no additional knowledge about V .

Note: P(V |Vi, E) = P(V |E) is a system of equations, one for each value of the variables.

8.8. Bayesian networks

A Bayesian network is a concise graphical data structure for representing any full JPD taking independence between variables into account. It is a directed graph such that each node is annotated with probability information. Formally, a Bayesian network consists of the following items:

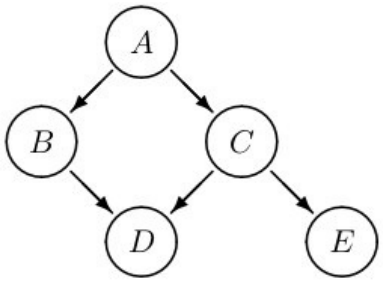
- A set of random variables making up the nodes of the network.
- A set of directed links between nodes. If there is a link V1 →V2, then V1 is a parent of V2.
- Each node V has a conditional probability distribution P(V |Parents(V)), where Parents(V) is the set of parent nodes of V .
- The graph has no directed cycles, i.e., it is a directed acyclic graph (DAG).

The topology of the network specifies the conditional independence relationships that hold in the domain, namely:

- Each node is conditionally independent of its non-descendants, given its parents.
- Hence, P(V|W,Parents(V)) = P(V|Parents(V)), for any set W of nodes which are neither parents nor descendants of V.

Furthermore, the network provides full information about the JPD P(V1, . . . , Vn) in the following way:

$$\mathrm{P}(V_1, \ldots, V_n) = \prod_{i=1}^{n} \mathrm{P}(V_i | Parents(V_i))$$

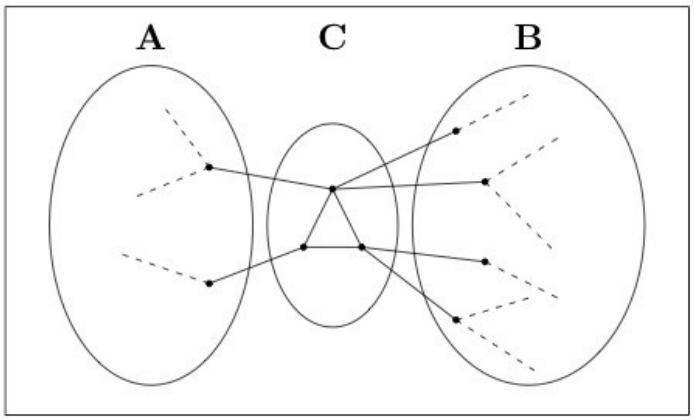


$$P(A, B, C, D, E) = P(A)P(B|A)P(C|A)P(D|BC)P(E|C)$$

D-Separation

The construction of a Bayesian network assumes certain conditional independence relations.

But: There can be further such relations! Question: How can we determine them? Answer: Notion of d-separation.



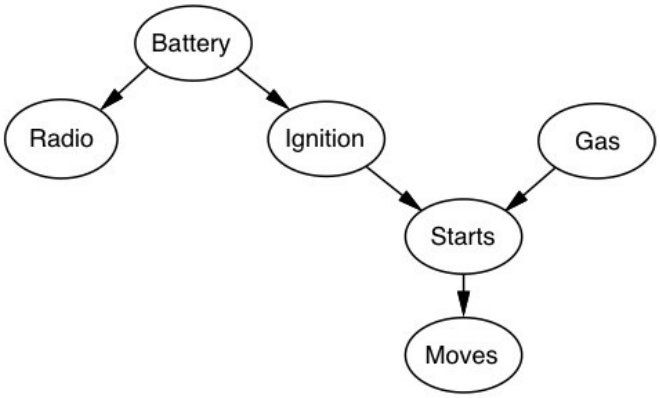
$$\mathbf{A} \perp\!\!\!\perp_{\mathcal{G}} \mathbf{B} \mid \mathbf{C}$$

All paths from A to B lead to C, so A and B depends on C. given C -> A and B are independent.

Path P is blocked relative to C if there is some node Z ∈ P such that one of the following conditions hold:

- Z ∈ C and one of the two arcs on P is incoming and the other is outgoing.
- Z ∈ C and both arcs on P are outgoing

Example:



Gas and Radio are independent given Ignition since the path P = (Gas, Starts, Ignition, Battery, Radio) is blocked by Ignition in view of the first condition of d-separation (due to the direction of the graph it is not possible to reach Radio from Gas or vice versa).

Gas and Radio are independent given Battery since path P is blocked by Battery in view of the second condition of d-separation.

8.9. Discussion

General evaluation of the probabilistic method: Large datasets are generally needed to determine the required conditional probabilities. Also, it is difficult to verify whether independence between variables is actually the case. In contrast to nonmonotonic logics, probabilistic reasoning does not allow to represent non-knowledge ⇒ each situation must be assigned with a probability. Contradictory information is not detected, rather it is propagated further on.

The use of networks goes back to the work of Sewall Wright on the analysis of genetic inheritance and animal growth factors (Wright, 1921, 1934).

Alternative probabilistic approaches:

Certainty factors: used in rule-based systems like the Mycin medical expert system.

Dempster-Shafer theory: based on probability intervals =⇒ distinguishes between uncertainty and ignorance.

Fuzzy logic: models vague concepts.