14. Artificial Intelligence and Automation

Dana S. Nau

Artificial intelligence (AI) focuses on getting machines to do things that we would call intelligent behavior. Intelligence – whether artificial or otherwise – does not have a precise definition, but there are many activities and behaviors that are considered intelligent when exhibited by humans and animals. Examples include seeing, learning, using tools, understanding human speech, reasoning, making good guesses, playing games, and formulating plans and objectives. AI focuses on how to get machines or computers to perform these same kinds of activities, though not necessarily in the same way that humans or animals might do them.

To most readers, *artificial intelligence* probably brings to mind science-fiction images of robots or computers that can perform a large number of human-like activities: seeing, learning, using tools, understanding human speech, reasoning, making good guesses, playing games, and formulating plans and objectives. And indeed, AI research focuses on how to get machines or computers to carry out activities such as these. On the other hand, it is important to note that the goal of AI is not to simulate biological intelligence. Instead, the objective is to get machines to behave or think intelligently, regardless of whether or not the internal computational processes are the same as in people or animals.

Most AI research has focused on ways to achieve intelligence by manipulating symbolic representations of problems. The notion that symbol manipulation is sufficient for artificial intelligence was summarized by *Newell* and *Simon* in their famous physical-symbol system hypothesis: *A physical-symbol system has the necessary and sufficient means for general intelligent action* and their heuristic search hypothesis [14.1]:

> *The solutions to problems are presented as symbol structures. A physical-symbol system exercises its intelligence in problem solving by search – that is – by generating and progressively modifying symbol structures until it produces a solution structure.*

On the other hand, there are several important topics of AI research – particularly machine-learning techniques such as neural networks and swarm intelligence – that are *subsymbolic* in nature, in the sense that they deal with vectors of real-valued numbers without attaching any explicit meaning to those numbers.

AI has achieved many notable successes [14.2]. Here are a few examples:

- Telephone-answering systems that understand human speech are now in routine use in many companies.

- Simple room-cleaning robots are now sold as consumer products.
- Automated vision systems that read handwritten zip codes are used by the US Postal Service to route mail.
- Machine-learning techniques are used by banks and stock markets to look for fraudulent transactions and alert staff to suspicious activity.
- Several web-search engines use machine-learning techniques to extract information and classify data scoured from the web.
- Automated planning and control systems are used in unmanned aerial vehicles, for missions that are too *dull, dirty or dangerous* for manned aircraft.
- Automated planning and scheduling techniques were used by the National Aeronautics and Space Administration (NASA) in their famous Mars rovers.

AI is divided into a number of subfields that correspond roughly to the various kinds of activities mentioned in the first paragraph. Three of the most important subfields are discussed in other chapters: machine learning in Chaps. 12 and 29, computer vision in Chap. 20, and robotics in Chaps. 1, 78, 82, and 84. This chapter discusses other topics in AI, including search procedures (Sect. 14.1.1), logical reasoning (Sect. 14.1.2), reasoning about uncertain information (Sect. 14.1.3), planning (Sect. 14.1.4), games (Sect. 14.1.5), natural-language processing (Sect. 14.1.6), expert systems (Sect. 14.1.7), and AI programming (Sect. 14.1.8).

14.1 Methods and Application Examples

14.1.1 Search Procedures

Many AI problems require a trial-and-error search through a search space that consists of *states of the world* (or *states*, for short), to find a path to a state s that satisfies some *goal condition* g. Usually the set of states is finite but very large: far too large to give a list of all the states (as a control theorist might do, for example, when writing a state-transition matrix). Instead, an initial state s_0 is given, along with a set O of *operators* for producing new states from existing ones.

As a simple example, consider Klondike, the most popular version of solitaire [14.3]. As illustrated in Fig. 14.1a, the initial state of the game is determined by dealing 28 cards from a 52-card deck into an arrangement called the tableau; the other 28 cards then go into a pile called the stock. New states are formed from old ones by moving cards around according to the rules of the game; for example, in Fig. 14.1a there are two possible moves: either move the ace of hearts to one of the foundations and turn up the card beneath the ace as shown in Fig. 14.1b, or move three cards from the stock to the waste. The goal is to produce a state in which all of the cards are in the foundation piles, with each suit in a different pile, in numerical order from the ace at the bottom to the king at the top. A *solution* is any path (a sequence of moves, or equivalently, the sequence of states that these moves take us to) from the initial state to a goal state.

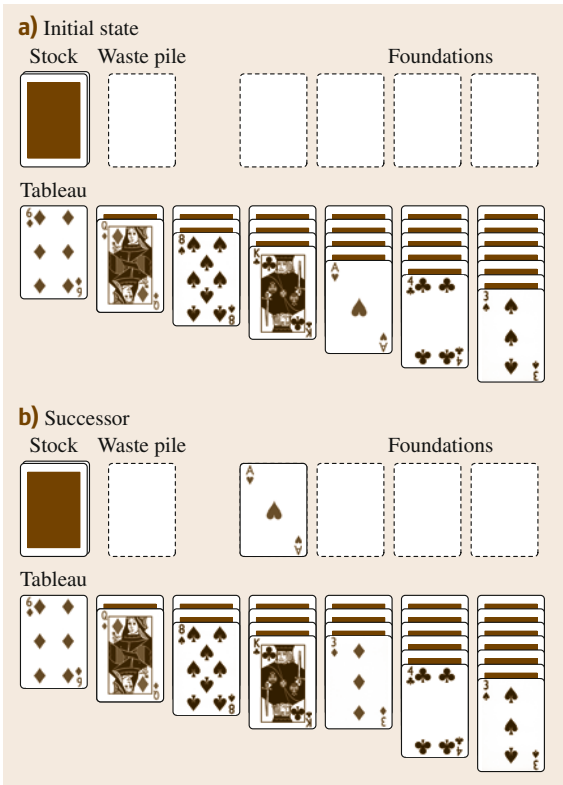

Fig. 14.1 (a) An initial state and (b) one of its two possible successors

Klondike has several characteristics that are typical of AI search problems:

- Each state is a combination of a finite set of features (in this case the cards and their locations), and the task is to find a path that leads from the initial state to a goal state.
- The rules for getting from one state to another can be represented using symbolic logic and discrete mathematics, but continuous mathematics is not as useful here, since there is no reasonable way to model the state space with continuous numeric functions.
- It is not clear a priori which paths, if any, will lead from the initial state to the goal states. The only obvious way to solve the problem is to do a trial-and-error search, trying various sequences of moves to see which ones might work.
- Combinatorial explosion is a big problem. The number of possible states in Klondike is well over 52!, which is many orders of magnitude larger than the number of atoms in the Earth. Hence a trial-and-error search will not terminate in a reasonable amount of time unless we can somehow restrict the search to a very small part of the search space – hopefully a part of the search space that actually contains a solution.
- In setting up the state space, we took for granted that the problem representation should correspond directly to the states of the physical system, but sometimes it is possible to make a problem much easier to solve by adapting a different representation; for example, [14.4] shows how to make Klondike much easier to solve by searching a different state space.

In many trial-and-error search problems, each solution path π will have a numeric measure $F(\pi)$ telling how desirable π is; for example, in Klondike, if we consider shorter solution paths to be more desirable than long ones, we can define $F(\pi)$ to be π's length. In such cases, we may be interested in finding either an *optimal* solution, i. e., a solution π such that $F(\pi)$ is as small as possible, or a *near-optimal* solution in which $F(\pi)$ is close to the optimal value.

Heuristic Search

The pseudocode in Fig. 14.2 provides an abstract model of state-space search. The input parameters include an initial state s_0 and a set of operators O. The procedure either fails, or returns a *solution path* π (i. e., a path from s_0 to a goal state).

```
1.  State-space-search(s0; O)
2.    Active ← {⟨s0⟩}
3.    while Active ≠ ∅ do
4.      choose a path π = ⟨s0,…, sk⟩ ∈ Active and remove it from Active
5.      if sk is a goal state then return π
6.      Successors ← {⟨s0,…, sk, o(sk)⟩ : o ∈ O is applicable to sk}
7.      optional pruning step: remove unpromising paths from Successors
8.      Active ← Active ∪ Successors
9.    repeat
10.   return failure
```

Fig. 14.2 An abstract model of state-space search. In line 6, $o(s_k)$ is the state produced by applying the operator o to the state s_k

As discussed earlier, we would like the search algorithm to focus on those parts of the state space that will lead to optimal (or at least near-optimal) solution paths. For this purpose, we will use a *heuristic function* $f(\pi)$ that returns a numeric value giving an approximate idea of how good a solution can be found by extending π, i. e.,

$$f(\pi) \approx \min\{F(\pi') : \pi' \text{ is a solution path that is an extension of } \pi\} .$$

It is hard to give foolproof guidelines for writing heuristic functions. Often they can be very ad hoc: in the worst case, $f(\pi)$ may just be an arbitrary function that the user *hopes* will give reasonable estimates. However, often it works well to define an easy-to-solve *relaxation* of the original problem, i. e., a modified problem in which some of the constraints are weakened or removed. If π is a partial solution for the original problem, then we can compute $f(\pi)$ by extending π into a solution π' for the relaxed problem, and returning $F(\pi')$; for example, in the famous traveling-salesperson problem, $f(\pi)$ can be computed by solving a simpler problem called the assignment problem [14.5]. Here are several procedures that can make use of such a heuristic function:

- *Best-first search* means that at line 4 of the algorithm in Fig. 14.2, we always choose a path $\pi = \langle s_0, \ldots, s_k \rangle$ that has the smallest value $f(\pi)$ of any path we have seen so far. Suppose that at least one solution exists, that there are no infinite paths of finite cost, and that the heuristic function f has the following *lower-bound* property

$$f(\pi) \le \min\{F(\pi') : \pi' \text{ is a solution path that is an extension of } \pi\} . \quad (14.1)$$

  Then best-first search will always return a solution π^* that minimizes $F(\pi^*)$. The well-known A*

search procedure [14.6] is a special case of best-first search, with some modifications to handle situations where there are multiple paths to the same state.
Best-first search has the advantage that, if it chooses an obviously bad state s to explore next, it will not spend much time exploring the subtree below s. As soon as it reaches successors of s whose f-values exceed those of other states on the *Active* list, best-first search will go back to those other states. The biggest drawback is that best-first search must remember every state it has ever visited, hence its memory requirement can be huge. Thus, best-first search is more likely to be a good choice in cases where the state space is relatively small, and the difficulty of solving the problem arises for some other reason (e.g., a costly-to-compute heuristic function, as in [14.7]).

- In *depth-first branch and bound*, at line 4 the algorithm always chooses the longest path in *Active*; if there are several such paths then the algorithm chooses the one that has the smallest value for $f(\pi)$. The algorithm maintains a variable π^* that holds the best solution seen so far, and the pruning step in line 7 removes a path π iff $f(\pi) \geq F(\pi^*)$. If the state space is finite and acyclic, at least one solution exists, and (14.1) holds, then depth-first branch and bound is guaranteed to return a solution π^* that minimizes $F(\pi^*)$.
  The primary advantage of depth-first search is its low memory requirement: the number of nodes in *Active* will never exceed bd, where d is the length of the current path. The primary drawback is that, if it chooses the wrong state to look at next, it will explore the entire subtree below that state before returning and looking at the state's siblings. Depth-first search does better in cases where the likelihood of choosing the wrong state is small or the time needed to search the incorrect subtrees is not too great.
- *Greedy search* is a state-space search without any backtracking. It is accomplished by replacing line 8 with $Active \leftarrow \{\pi_1\}$, where π_1 is the path in *Successors* that minimizes $\{f(\pi') \mid \pi' \in Successors\}$. *Beam search* is similar except that, instead of putting just one successor π_1 of π into *Active*, we put k successors $\pi_1, \ldots, \pi_k$ into *Active*, for some fixed k.
  Both greedy search and beam search will return very quickly once they find a solution, since neither of them will spend any time looking for better solutions. Hence they are good choices if the state space is large, most paths lead to solutions, and we are more interested in finding a solution quickly than in finding an optimal solution. However, if most paths do not lead to solutions, both algorithms may fail to find a solution at all (although beam search is more robust in this regard, since it explores several paths rather than just one path). In this case, it may work well to do a modified greedy search that backtracks and tries a different path every time it reaches a dead end.

Hill-Climbing

A *hill-climbing* problem is a special kind of search problem in which *every* state is a goal state. A hill-climbing procedure is like a greedy search, except that *Active* contains a single state rather than a single path; this is maintained in line 6 by inserting a single successor of the current state s_k into *Active*, rather than all of s_k's successors. In line 5, the algorithm terminates when none of s_k's successors looks better than s_k itself, i. e., when s_k has no successor s_{k+1} with $f(s_{k+1}) > f(s_k)$. There are several variants of the basic hill-climbing approach:

- *Stochastic hill-climbing and simulated annealing.* One difficulty with hill-climbing is that it will terminate in cases where s_k is a local minimum but not a global minimum. To prevent this from happening, a *stochastic hill-climbing* procedure does not always return when the test in line 5 succeeds. Probably the best known example is *simulated annealing*, a technique inspired by annealing in metallurgy, in which a material is heated and then slowly cooled. In simulated annealing, this is accomplished as follows. At line 5, if none of s_k's successors look better than s_k then the procedure will not necessarily terminate as in ordinary hill-climbing; instead it will terminate with some probability p_i, where i is the number of loop iterations and p_i grows monotonically with i.
- *Genetic algorithms.* A genetic algorithm is a modified version of hill-climbing in which successor states are generated not using the normal successor function, but instead using operators reminiscent of genetic recombination and mutation. In particular, *Active* contains k states rather than just one, each state is a string of symbols, and the operators O are computational analogues of genetic recombination and mutation. The termination criterion in line 5 is generally ad hoc; for example, the algorithm may terminate after a specified number of iterations, and return the best one of the states currently in *Active*.

Hill-climbing algorithms are good to use in problems where we want to find a solution very quickly, then continue to look for a better solution if additional time is available. More specifically, genetic algorithms are useful in situations where each solution can be represented as a string whose substrings can be combined with substrings of other solutions.

Constraint Satisfaction and Constraint Optimization

A constraint-satisfaction problem is a special kind of search problem in which each state is a set of assignments of values to variables $\{X_i\}_{i=1}^n$ that have finite domains $\{D_i\}_{i=1}^n$, and the objective is to assign values to the variables in such a way that some set of constraints is satisfied.

In the search space for a constraint-satisfaction problem, each state at depth i corresponds to an assignment of values to i of the n variables, and each branch corresponds to assigning a specific value to an unassigned variable. The search space is finite: the maximum length of any path from the root node is n since there are only n variables to assign values to. Hence a depth-first search works quite well for constraint-satisfaction problems. In this context, some powerful techniques have been formulated for choosing which variable to assign next, detecting situations where previous variable assignments will make it impossible to satisfy the remaining constraints, and even restructuring the problem into one that is easier to solve [14.8, Chap. 5].

A *constraint-optimization* problem combines a constraint-satisfaction problem with an objective function that one wants to optimize. Such problems can be solved by combining constraint-satisfaction techniques with the optimization techniques mentioned in *Heuristic Search.*

Applications of Search Procedures

Software using AI search techniques has been developed for a large number of commercial applications. A few examples include the following:

- Several universities routinely use constraint-satisfaction software for course scheduling.
- *Airline ticketing.* Finding the best price for an airline ticket is a constraint-optimization problem in which the constraints are provided by the airlines' various rules on what tickets are available at what prices under what conditions [14.9]. An example of software that works in this fashion is the ITA software (itasoftware.com) system that is used by several airline-ticketing web sites, e.g., Orbitz (orbitz.com) and Kayak (kayak.com).
- *Scheduling and routing.* Companies such as ILOG (ilog.com) have developed software that uses search and optimization techniques for scheduling [14.10], routing [14.11], workflow composition [14.12], and a variety of other applications.
- *Information retrieval from the web.* AI search techniques are important in the web-searching software used at sites such as Google News [14.2].

Additional reading. For additional reading on search algorithms, see *Pearl* [14.13]. For additional details about constraint processing, see *Dechter* [14.14].

14.1.2 Logical Reasoning

A *logic* is a formal language for representing information in such a way that one can reason about what things are true and what things are false. The logic's *syntax* defines what the *sentences* are; and its *semantics* defines what those sentences mean in some *world*. The two best-known logical formalisms, propositional logic and first-order logic, are described briefly below.

Propositional Logic and Satisfiability

Propositional logic, also known as *Boolean algebra*, includes sentences such as $A \wedge B \Rightarrow C$, where A, B, and C are variables whose domain is $\{true, false\}$. Let w_1 be a world in which A and C are *true* and B is *false*, and let w_2 be a world in which all three of the Boolean variables are *true*. Then the sentence $A \wedge B \Rightarrow C$ is false in w_1 and true in w_2. Formally, we say that w_2 is a *model* of $A \wedge B \Rightarrow C$, or that it *entails* S_1. This is written symbolically as

$$w_2 \models A \wedge B \Rightarrow C \,.$$

The *satisfiability problem* is the following: given a sentence S of propositional logic, does there exist a world (i. e., an assignment of truth values to the variables in S) in which S is true? This problem is central to the theory of computation, because it was the very first computational problem shown to be NP-complete. Without going into a formal definition of NP-completeness, NP is, roughly, the set of all computational problems such that, if we are given a purported solution, we can check *quickly* (i. e., in a polynomial amount of computing time) whether the solution is correct. An *NP-complete* problem is a problem that is one of the hardest problems in NP, in the sense that

solving any NP-complete problems would provide a solution to every problem in NP. It is conjectured that no NP-complete problem can be solved in a polynomial amount of computing time. There is a great deal of evidence for believing the conjecture, but nobody has ever been able to prove it. This is the most famous unsolved problem in computer science.

First-Order Logic

A much more powerful formalism is *first-order logic* [14.15], which uses the same logical connectives as in propositional logic but adds the following syntactic elements (and semantics, respectively): constant symbols (which denote the objects), variable symbols (which range over objects), function symbols (which represent functions), predicate symbols (which represent relations among objects), and the quantifiers $\forall x$ and $\exists x$, where x is any variable symbol (to specify whether a sentence is true for every value x or for at least one value of x).

First-order logic includes a standard set of *logical axioms*. These are statements that must be true in every possible world; one example is the transitive property of equality, which can be formalized as

$$\forall x\, \forall y\, \forall z\, (x = y \wedge y = z) \Rightarrow x = z\,.$$

In addition to the logical axioms, one can add a set of *nonlogical axioms* to describe what is true in a particular kind of world; for example, if we want to specify that there are exactly two objects in the world, we could do this by the following axioms, where a and b are constant symbols, and x, y, z are variable symbols

$$a \neq b\,, \tag{14.2a}$$

$$\forall x\, \forall y\, \forall z\, x = y \vee y = z \vee x = z\,. \tag{14.2b}$$

The first axiom asserts that there are at least two objects (namely a and b), and the second axiom asserts that there are no more than two objects.

First-order logic also includes a standard set of *inference rules*, which can be used to infer additional true statements. One example is *modus ponens*, which allows one to infer a statement Q from the pair of statements $P \Rightarrow Q$ and P.

The logical and nonlogical axioms and the rules of inference, taken together, constitute a *first-order theory*. If $\mathcal{T}$ is a first-order theory, then a *model* of $\mathcal{T}$ is any world in which $\mathcal{T}$'s axioms are true. (In science and engineering, a *mathematical model* generally means a formalism for some real-world phenomenon; but in mathematical logic, *model* means something very different: the formalism is called a theory, and the the real-world phenomenon *itself* is a model of the theory.) For example, if $\mathcal{T}$ includes the nonlogical axioms given above, then a model of $\mathcal{T}$ is any world in which there are exactly two objects.

A *theorem* of $\mathcal{T}$ is defined recursively as follows: every axiom is a theorem, and any statement that can be produced by applying inference rules to theorems is also a theorem; for example, if $\mathcal{T}$ is any theory that includes the nonlogical axioms (14.2a) and (14.2b), then the following statement is a theorem of $\mathcal{T}$

$$\forall x\; x = a \vee x = b\,.$$

A fundamental property of first-order logic is *completeness*: for every first-order theory $\mathcal{T}$ and every statement S in $\mathcal{T}$, S is a theorem of $\mathcal{T}$ if and only if S is true in all models of $\mathcal{T}$. This says, basically, that first-order logical reasoning does exactly what it is supposed to do.

Nondeductive Reasoning

Deductive reasoning – the kind of reasoning used to derive theorems in first-order logic – consists of deriving a statement y as a consequence of a statement x. Such an inference is *deductively valid* if there is no possible situation in which x is true and y is false. However, several other kinds of reasoning have been studied by AI researchers. Some of the best known include *abductive reasoning* and *nonmonotonic reasoning*, which are discussed briefly below, and *fuzzy logic*, which is discussed later.

Nonmonotonic Reasoning. In most formal logics, deductive inference is *monotone*; i. e., adding a formula to a logical theory never causes something *not* to be a theorem that was a theorem of the original theory. Nonmonotonic logics allow deductions to be made from *beliefs* that may not always be true, such as the *default assumption* that birds can fly. In nonmonotonic logic, if b is a bird and we know nothing about b then we may conclude that b can fly; but if we later learn that b is an ostrich or b has a broken wing, then we will retract this conclusion.

Abductive Reasoning. This is the process of inferring x from y when x entails y. Although this can produce results that are incorrect within a formal deductive system, it can be quite useful in practice, especially when something is known about the probability of different causes of y; for example, the Bayesian reasoning described later can be viewed as a combina-

tion of deductive reasoning, abductive reasoning, and probabilities.

Applications of Logical Reasoning

The satisfiability problem has important applications in hardware design and verification; for example, electronic design automation (EDA) tools include satisfiability checking algorithms to check whether a given digital system design satisfies various criteria. Some EDA tools use first-order logic rather than propositional logic, in order to check criteria that are hard to express in propositional logic.

First-order logic provides a basis for automated reasoning systems in a number of application areas. Here are a few examples:

- *Logic programming*, in which mathematical logic is used as a programming language, uses a particular kind of first-order logic sentence called a Horn clause. Horn clauses are implications of the form $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow P_{n+1}$, where each P_i is an *atomic formula* (a predicate symbol and its argument list). Such an implication can be interpreted logically, as a statement that P_{n+1} is true if $P_1, \ldots, P_n$ are true, or procedurally, as a statement that a way to show or solve P_{n+1} is to show or solve $P_1, \ldots, P_n$. The best known implementation of logic programming is the programming language Prolog, described further below.
- *Constraint programming*, which combines logic programming and constraint satisfaction, is the basis for ILOG's CP Optimizer (http://www.ilog.com/products/cpoptimizer).
- The web ontology language (OWL) and DAML + OIL languages for semantic web markup are based on description logics, which are a particular kind of first-order logic.
- Fuzzy logic has been used in a wide variety of commercial products including washing machines, refrigerators, automotive transmissions and braking systems, camera tracking systems, etc.

14.1.3 Reasoning About Uncertain Information

Earlier in this chapter it was pointed out that AI systems often need to reason about discrete sets of states, and the relationships among these states are often nonnumeric. There are several ways in which uncertainty can enter into this picture; for example, various events may occur spontaneously and there may be uncertainty about whether they will occur, or there may be uncertainty about what things are currently true, or the degree to which they are true. The two best-known techniques for reasoning about such uncertainty are Bayesian probabilities and fuzzy logic.

Bayesian Reasoning

In some cases we may be able to model such situations probabilistically, but this means reasoning about discrete random variables, which unfortunately incurs a combinatorial explosion. If there are n random variables and each of them has d possible values, then the joint probability distribution function (PDF) will have d^n entries. Some obvious problems are (1) the worst-case time complexity of reasoning about the variables is $\Theta(d^n)$, (2) the worst-case space complexity is also $\Theta(d^n)$, and (3) it seems impractical to suppose that we can acquire accurate values for all d^n entries.

The above difficulties can be alleviated if some of the variables are known to be independent of each other; for example, suppose that the n random variables mentioned above can be partitioned into $\lceil n/k \rceil$ subsets, each containing at most k variables. Then the joint PDF for the entire set is the product of the PDFs of the subsets. Each of those has d^k entries, so there are only $\lceil n/k \rceil n^k$ entries to acquire and reason about.

Absolute independence is rare; but another property is more common and can yield a similar decrease in time and space complexity: *conditional* independence [14.16]. Formally, a is *conditionally independent of b given c* if $P(ab|c) = P(a|c)P(b|c)$.

Bayesian networks are graphical representations of conditional independence in which the network topology reflects knowledge about which events *cause* other events. There is a large body of work on these networks, stemming from seminal work by Judea Pearl. Here is a simple example due to *Pearl* [14.17]. Figure 14.3 represents the following hypothetical situation:

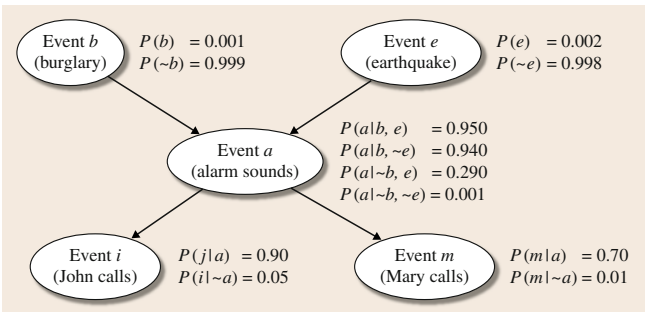

Fig. 14.3 A simple Bayesian network

My house has a burglar alarm that will usually go off (event a) if there's a burglary (event b), an earthquake (event e), or both, with the probabilities shown in Fig. 14.3. If the alarm goes off, my neighbor John will usually call me (event j) to tell me; and he may sometimes call me by mistake even if the alarm has not gone off, and similarly for my other neighbor Mary (event m); again the probabilities are shown in the figure.

The joint probability for each combination of events is the product of the conditional probabilities given in Fig. 14.3

$$\begin{aligned} P(b, e, a, j, m) &= P(b)P(e)P(a|b, e) \\ &\quad \times P(j|a)P(m|a)\,, \\ P(b, e, a, j, \neg m) &= P(b)P(e)P(a|b, e)P(j|a) \\ &\quad \times P(\neg m|a)\,, \\ P(b, \neg e, \neg a, j, \neg m) &= P(b)P(\neg e)P(\neg a|b, \neg e) \\ &\quad \times P(j|\neg a)P(\neg m|\neg a)\,, \\ &\quad \ldots \end{aligned}$$

Hence, instead of reasoning about a joint distribution with $2^5 = 32$ entries, we only need to reason about products of the five conditional distributions shown in the figure.

In general, probability computations can be done on Bayesian networks much more quickly than would be possible if all we knew was the joint PDF, by taking advantage of the fact that each random variable is conditionally independent of most of the other variables in the network. One important special case occurs when the network is acyclic (e.g., the example in Fig. 14.3), in which case the probability computations can be done in low-order polynomial time. This special case includes decision trees [14.8], in which the network is both acyclic and rooted. For additional details about Bayesian networks, see *Pearl* and *Russell* [14.16].

Applications of Bayesian Reasoning. Bayesian reasoning has been used successfully in a variety of applications, and dozens of commercial and freeware implementations exist. The best-known application is spam filtering [14.18, 19], which is available in several mail programs (e.g., Apple Mail, Thunderbird, and Windows Messenger), webmail services (e.g., gmail), and a plethora of third-party spam filters (probably the best-known is spamassassin [14.20]). A few other examples include medical imaging [14.21], document classification [14.22], and web search [14.23].

Fuzzy Logic

Fuzzy logic [14.24, 25] is based on the notion that, instead of saying that a statement P is true or false, we can give P a *degree of truth*. This is a number in the interval [0, 1], where 0 means false, 1 means true, and numbers between 0 and 1 denote partial degrees of truth.

As an example, consider the action of moving a car into a parking space, and the statement *the car is in the parking space*. At the start, the car is not in the parking space, hence the statement's degree of truth is 0. At the end, the car is completely in the parking space, hence the statement's degree of is 1. Between the start and end of the action, the statement's degree of truth gradually increases from 0 to 1.

Fuzzy logic is closely related to fuzzy set theory, which assigns degrees of truth to set membership. This concept is easiest to illustrate with sets that are intervals over the real line; for example, Fig. 14.4 shows a set S having the following set membership function

$$\text{truth}(x \in S) = \begin{cases} 1\,, & \text{if } 2 \le x \le 4\,, \\ 0\,, & \text{if } x \le 1 \text{ or } x \ge 5\,, \\ x - 1\,, & \text{if } 1 < x < 2\,, \\ 5 - x\,, & \text{if } 4 < x < 5\,. \end{cases}$$

The logical notions of conjunction, disjunction, and negation can be generalized to fuzzy logic as follows

$$\begin{aligned} \text{truth}(x \wedge y) &= \min[\text{truth}(x), \text{truth}(y)]\,; \\ \text{truth}(x \vee y) &= \max[\text{truth}(x), \text{truth}(y)]\,; \\ \text{truth}(\neg x) &= 1 - \text{truth}(x)\,. \end{aligned}$$

Fuzzy logic also allows other operators, more linguistic in nature, to be applied. Going back to the example of a full gas tank, if the degree of truth of *g is full* is d, then one might want to say that the degree of truth of *g is very full* is d^2. (Obviously, the choice of d^2 for *very* is subjective. For different users or different applications, one might want to use a different formula.) Degrees of truth are semantically distinct from probabilities, although the two concepts are often confused;

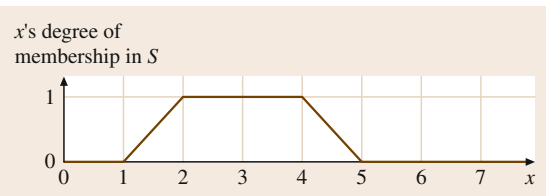

Fig. 14.4 A degree-of-membership function for a fuzzy set

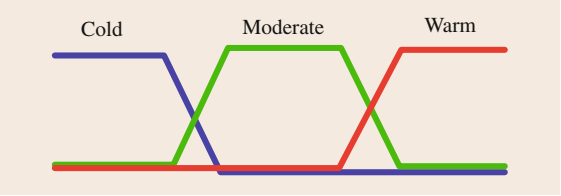

Fig. 14.5 Degree-of-membership functions for three overlapping temperature ranges

for example, we could talk about the probability that someone would say the car is in the parking space, but this probability is likely to be a different number than the degree of truth for the statement that the car is in the parking space.

Fuzzy logic is controversial in some circles; e.g., many statisticians would maintain that probability is the only rigorous mathematical description of uncertainty. On the other hand, it has been quite successful from a practical point of view, and is now used in a wide variety of commercial products.

Applications of Fuzzy Logic. Fuzzy logic has been used in a wide variety of commercial products. Examples include washing machines, refrigerators, dishwashers, and other home appliances; vehicle subsystems such as automotive transmissions and braking systems; digital image-processing systems such as edge detectors; and some microcontrollers and microprocessors.

In such applications, a typical approach is to specify fuzzy sets that correspond to different subranges of a continuous variable; for instance, a temperature measurement for a refrigerator might have degrees of membership in several different temperature ranges, as shown in Fig. 14.5. Any particular temperature value will correspond to three degrees of membership, one for each of the three temperature ranges; and these degrees of membership could provide input to a control system to help it decide whether the refrigerator is too cold, too warm, or in the right temperature range.

14.1.4 Planning

In ordinary English, there are many different kinds of plans: project plans, floor plans, pension plans, urban plans, floor plans, etc. AI planning research focuses specifically on *plans of action*, i. e., [14.26]:

> *... representations of future behavior ... usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.*

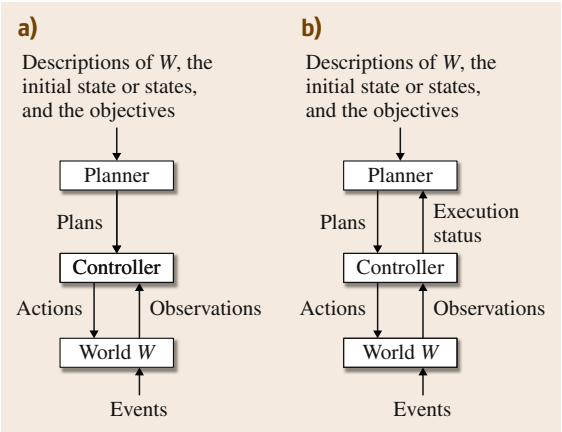

Fig. 14.6a,b Simple conceptual models for **(a)** offline and **(b)** online planning

Figure 14.6 gives an abstract view of the relationship between a planner and its environment. The planner's input includes a description of the world W in which the plan is to be executed, the initial state (or set of possible initial states) of the world, and the objectives that the plan is supposed to achieve. The planner produces a plan that is a set of instructions to a *controller*, which is the system that will execute the plan. In *offline planning*, the planner generates the entire plan, gives it to the controller, and exits. In *online planning*, plan generation and plan execution occur concurrently, and the planner gets feedback from the controller to aid it in generating the rest of the plan. Although not shown in the figure, in some cases the plan may go to a *scheduler* before going to the controller. The purpose of the scheduler is to make decisions about when to execute various parts of the plan and what resources to use during plan execution.

Examples. The following paragraphs include several examples of offline planners, including the sheet-metal bending planner in *Domain-Specific Planners*, and all of the planners in *Classical Planning* and *Domain-Configurable Planners.* One example of an online planner is the planning software for the Mars rovers in *Domain-Specific Planners.* The planner for the Mars rovers also incorporates a scheduler.

Domain-Specific Planners

A *domain-specific* planning system is one that is tailor-made for a given planning domain. Usually the design of the planning system is dictated primarily by the detailed requirements of the specific domain, and the

Fig. 14.7 One of the Mars rovers

Fig. 14.8 A sheet-metal bending machine

system is unlikely to work in any domain other other than the one for which it was designed.

Many successful planners for real-world applications are domain specific. Two examples are the autonomous planning system that controlled the Mars rovers [14.27] (Fig. 14.7), and the software for planning sheet-metal bending operations [14.28] that is bundled with Amada Corporation's sheet-metal bending machines (Fig. 14.8).

Classical Planning

Most AI planning research has been guided by a desire to develop principles that are *domain independent*, rather than techniques specific to a single planning domain. However, in order to make any significant headway in the development of such principles, it has proved necessary to make restrictions on what kinds of planning domains they apply to.

In particular, most AI planning research has focused on *classical* planning problems. In this class of planning problems, the world W is finite, fully observable, deterministic, and static (i.e., the world never changes except as a result of our actions); and the objective is to produce a finite sequence of actions that takes the world from some specific *initial* state to any of some set of *goal* states. There is a standard language, planning domain definition language (PDDL) [14.29], that can represent planning problems of this type, and there are dozens (possibly hundreds) of classical planning algorithms.

One of the best-known classical planning algorithms is GraphPlan [14.30], an iterative-deepening algorithm that performs the following steps in each iteration i:

1. Generate a *planning graph* of depth i. Without going into detail, the planning graph is basically the search space for a greatly simplified version of the planning problem that can be solved very quickly.
2. Search for a solution to the original unsimplified planning problem, but restrict this search to occur solely within the planning graph produced in step 1. In general, this takes much less time than an unrestricted search would take.

GraphPlan has been the basis for dozens of other classical planning algorithms.

Domain-Configurable Planners

Another important class of planning algorithms are the *domain-configurable planners*. These are planning systems in which the planning engine is domain independent but the input to the planner includes domain-specific information about how to do planning in the problem domain at hand. This information serves to constrain the planner's search so that the planner searches only a small part of the search space. There are two main types of domain-configurable planners:

- *Hierarchical task network* (HTN) planners such as O-Plan [14.31], SIPE-2 (system for interactive planning and execution) [14.32], and SHOP2 (simple hierarchical ordered planner 2) [14.33]. In these planners, the objective is described not as a set of goal states, but instead as a collection of *tasks* to perform. Planning proceeds by decomposing tasks into subtasks, subtasks into sub-subtasks, and so forth in a recursive manner until the planner reaches *primitive* tasks that can be performed using actions

similar to those used in a classical planning system. To guide the decomposition process, the planner uses a collection of *methods* that give ways of decomposing tasks into subtasks.

- *Control-rule* planners such as temporal logic planner (TLPlan) [14.34] and temporal action logic planner (TALplanner) [14.35]. Here, the domain-specific knowledge is a set of rules that give conditions under which nodes can be pruned from the search space; for example, if the objective is to load a collection of boxes into a truck, one might write a rule telling the planner "do not pick up a box unless (1) it is not on the truck and (2) it is supposed to be on the truck." The planner does a forward search from the initial state, but follows only those paths that satisfy the control rules.

Planning with Uncertain Outcomes

One limitation of classical planners is that they cannot handle uncertainty in the outcomes of the actions. The best-known model of uncertainty in planning is the Markov decision process (MDP) model. MDPs are well known in engineering, but are generally defined over continuous sets of states and actions, and are solved using the tools of continuous mathematics. In contrast, the MDPs considered in AI research are usually discrete, with the relationships among the states and actions being symbolic rather than numeric (the latest version of PDDL [14.29] incorporates the ability to represent planning problems in this fashion):

- There is a set of states S and a set of actions A. Each state s has a reward $R(s)$, which is a numeric measure of the desirability of s. If an action a is applicable to s, then $C(a, s)$ is the cost of executing a in s.
- If we execute an action a in a state s, the outcome may be any state in S. There is a probability distribution over the outcomes: $P(s'|a, s)$ is the probability that the outcome will be s', with $\sum_{s' \in S} P(s'|a, s) = 1$.
- Starting from some *initial state* s_0, suppose we execute a sequence of actions that take the MDP from s_0 to some state s_1, then from s_1 to s_2, then from s_2 to s_3, and so forth. The sequence of states $h = \langle s_0, s_1, s_2, \ldots \rangle$ is called a *history*. In a *finite-horizon* problem, all of the MDP's possible histories are finite (i. e., the MDP ceases to operate after a finite number of state transitions). In an *infinite-horizon* problem, the histories are infinitely long (i. e., the MDP never stops operating).
- Each history h has a utility $U(h)$ that can be computed by summing the rewards of the states minus the costs of the actions

$$U(h) = \begin{cases} \sum_{i=0}^{n-1} R(s_i) - C(s_i, \pi(s_i)) + R(s_n) \,, \\ \quad \text{for finite-horizon problems} \,, \\ \sum_{i=0}^{\infty} \gamma^i R(s_i) - C(s_i, \pi(s_i)) \\ \quad \text{for infinite-horizon problems} \,. \end{cases}$$

  In the equation for infinite-horizon problems, γ is a number between 0 and 1 called the *discount factor*. Various rationales have been offered for using discount factors, but the primary purpose is to ensure that the infinite sum will converge to a finite value.
- A *policy* is any function $\pi : S \to A$ that returns an action to perform in each state. (More precisely, π is a *partial* function from S to A. We do not need to define π at a state $s \in S$ unless π can actually generate a history that includes s.) Since the outcomes of the actions are probabilistic, each policy π induces a probability distribution over MDP's possible histories

$$P(h|\pi) = P(s_0)P(s_1|\pi(s_0), s_0)P(s_1|\pi(s_1), s_1) \times P(s_2|\pi(s_2), s_2) \ldots$$

  The *expected utility* of π is the sum, over all histories, of h's probability times its utility: $\mathrm{EU}(\pi) = \sum_h P(h|\pi)U(h)$. Our objective is to generate a policy π having the highest expected utility.

Traditional MDP algorithms such as value iteration or policy iteration are difficult to use in AI planning problems, since these algorithms iterate over the entire set of states, which can be huge. Instead, the focus has been on developing algorithms that examine only a small part of the search space. Several such algorithms are described in [14.36]. One of the best-known is real-time dynamic programming (RTDP) [14.37], which works by repeatedly doing a forward search from the initial state (or the set of possible initial states), extending the frontier of the search a little further each time until it has found an acceptable solution.

Applications of Planning

The paragraph on *Domain-Specific Planners* gave several examples of successful applications of domain-specific planners. Domain-configurable HTN planners such as O-Plan, SIPE-2, and SHOP2 have been deployed in hundreds of applications; for example

a system for controlling unmanned aerial vehicles (UAVs) [14.38] uses SHOP2 to decompose high-level objectives into low-level commands to the UAV's controller.

Because of the strict set of restrictions required for classical planning, it is not directly usable in most application domains. (One notable exception is a cybersecurity application [14.39].) On the other hand, several domain-specific or domain-configurable planners are based on generalizations of classical planning techniques. One example is the domain-specific Mars rover planning software mentioned in *Domain-Specific Planners*, which involved a generalization of a classical planning technique called *plan-space planning* [14.40, Chap. 5]. Some of the generalizations included ways to handle action durations, temporal constraints, and other problem characteristics. For additional reading on planning, see *Ghallab* et al. [14.40] and *LaValle* [14.41].

14.1.5 Games

One of the oldest and best-known research areas for AI has been classical games of strategy, such as chess, checkers, and the like. These are examples of a class of games called *two-player perfect-information zero-sum turn-taking games*. Highly successful decision-making algorithms have been developed for such games: Computer chess programs are as good as the best grandmasters, and many games – including most recently checkers [14.42] – are now completely solved.

A *strategy* is the game-theoretic version of a policy: a function from states into actions that tells us what move to make in any situation that we might encounter. Mathematical game theory often assumes that a player chooses an entire strategy in advance. However, in a complicated game such as chess it is not feasible to construct an entire strategy in advance of the game. Instead, the usual approach is to choose each move at the time that one needs to make this move.

In order to choose each move intelligently, it is necessary to get a good idea of the possible future consequences of that move. This is done by searching a *game tree* such as the simple one shown in Fig. 14.9. In this figure, there are two players whom we will call Max and Min. The square nodes represent states where it is Max's move, the round nodes represent states where it is Min's move, and the edges represent moves. The terminal nodes represent states in which the game has ended, and the numbers below the terminal nodes are the payoffs. The figure shows the payoffs for both Max and Min; note that they always sum to 0 (hence the name *zero-sum* games).

From von Neuman and Morgenstern's famous Minimax theorem, it follows that Max's *dominant* (i. e., best) strategy is, on each turn, to move to whichever state s has the highest *minimax value* $m(s)$, which is defined as follows

$$m(s) = \begin{cases} \text{Max's payoff at } s \text{ ,} \\ \quad \text{if } s \text{ is a terminal node ,} \\ \max\{m(t) : t \text{ is a child of } s\} \text{ ,} \\ \quad \text{if it is Max's move at } s \text{ ,} \\ \min\{m(t) : t \text{ is a child of } s\} \text{ ,} \\ \quad \text{if it is Min's move at } s \text{ ,} \end{cases} \tag{14.3}$$

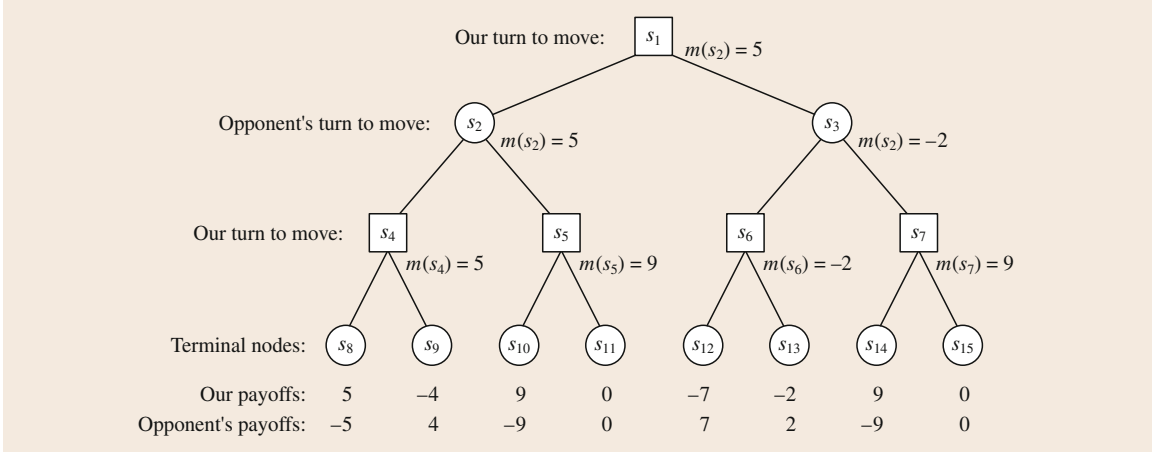

Fig. 14.9 A simple example of a game tree

where *child* means any immediate successor of s; for example, in Fig. 14.9,

$$m(s_2) = \min(\max(5, -4), \max(9, 0)) = \min(5, 9) = 5 ; \quad (14.4)$$

$$m(s_3) = \min(\max(s_{12}), \max(s_{13}), \max(s_{14}), \max(s_{15})) = \min(7, 0) = 0 . \quad (14.5)$$

Hence Max's best move at s_1 is to move to s_2.

A brute-force computation of (14.3) requires searching every state in the game tree, but most nontrivial games have so many states that it is infeasible to explore more than a small fraction of them. Hence a number of techniques have been developed to speed up the computation. The best known ones include:

- *Alpha–beta pruning*, which is a technique for deducing that the minimax values of certain states cannot have any effect on the minimax value of s, hence those states and their successors do not need to be searched in order to compute s's minimax value. Pseudocode for the algorithm can be found in [14.8, 43], and many other places.
  In brief, the algorithm does a modified depth-first search, maintaining a variable α that contains the minimax value of the best move it has found so far for Max, and a variable β that contains the minimax value of the best move it has found so far for Min. Whenever it finds a move for Min that leads to a subtree whose minimax value is less than α, it does not search this subtree because Max can achieve at least α by making the best move that the algorithm found for Max earlier. Similarly, whenever the algorithm finds a move for Max that leads to a subtree whose minimax value exceeds β, it does not search this subtree because Min can achieve at least β by making the best move that the algorithm found for Min earlier.
  The amount of speedup provided by alpha–beta pruning depends on the order in which the algorithm visits each node's successors. In the worst case, the algorithm will do no pruning at all and hence will run no faster than a brute-force minimax computation, but in the best case, it provide an exponential speedup [14.43].
- *Limited-depth search*, which searches to an arbitrary *cutoff depth*, uses a *static evaluation function* $e(s)$ to estimate the utility values of the states at that depth, and then uses these estimates in (14.3) as if those states were terminal states and their estimated utility values were the exact utility values for those states [14.8].

Games with Chance, Imperfect Information, and Nonzero-Sum Payoffs

The game-tree search techniques outlined above do extremely well in perfect-information zero-sum games, and can be adapted to perform well in perfect-information games that include chance elements, such as backgammon [14.44]. However, game-tree search does less well in imperfect-information zero-sum games such as bridge [14.45] and poker [14.46]. In these games, the lack of imperfect information increases the effective branching factor of the game tree because the tree will need to include branches for all of the moves that the opponent *might* be able to make. This increases the size of the tree exponentially.

Second, the minimax formula implicitly assumes that the opponent will always be able to determine which move is best for them – an assumption that is less accurate in games of imperfect information than in games of perfect information, because the opponent is less likely to have enough information to be able to determine which move is best [14.47].

Some imperfect-information games are *iterated* games, i. e., tournaments in which two players will play the same game with each other again and again. By observing the opponent's moves in the previous *iterations* (i. e., the previous times one has played the game with this opponent), it is often possible to detect patterns in the opponent's behavior and use these patterns to make probabilistic predictions of how the opponent will behave in the next iteration. One example is Roshambo (rock–paper–scissors). From a game-theoretic point of view, the game is trivial: the best strategy is to play purely at random, and the expected payoff is 0. However, in practice, it is possible to do much better than this by observing the opponent's moves in order to detect and exploit patterns in their behavior [14.48]. Another example is poker, in which programs have been developed that play nearly as well as human champions [14.46]. The techniques used to accomplish this are a combination of probabilistic computations, game-tree search, and detecting patterns in the opponent's behavior [14.49].

Applications of Games

Computer programs have been developed to take the place of human opponents in so many different games of strategy that it would be impractical to list all of them here. In addition, game-theoretic techniques have application in several of the behavioral and social sciences, primarily in economics [14.50].

Highly successful computer programs have been written for chess [14.51], checkers [14.42, 52], bridge [14.45], and many other games of strategy [14.53]. AI game-searching techniques are being applied successfully to tasks such as business sourcing [14.54] and to games that are models of social behavior, such as the iterated prisoner's dilemma [14.55].

14.1.6 Natural-Language Processing

Natural-language processing (NLP) focuses on the use of computers to analyze and understand human (as opposed to computer) languages. Typically this involves three steps: part-of-speech tagging, syntactic parsing, and semantic processing. Each of these is summarized below.

Part-of-Speech Tagging

Part-of-speech tagging is the task of identifying individual words as nouns, adjectives, verbs, etc. This is an important first step in parsing written sentences, and it also is useful for *speech recognition* (i. e., recognizing spoken words) [14.56].

A popular technique for part-of-speech tagging is to use hidden Markov models (HMMs) [14.57]. A hidden Markov model is a finite-state machine that has states and probabilistic state transitions (i. e., at each state there are several different possible *next* states, with a different probability of going to each of them). The states themselves are not directly observable, but in each state the HMM emits a symbol that we can observe.

To use HMMs for part-of-speech tagging, we need an HMM in which each state is a pair (w, t), where w is a word in some finite lexicon (e.g., the set of all English words), and t is a part-of-speech tag such as *noun*, *adjective*, or *verb*. Note that, for each word w, there may be more than one possible part-of-speech tag, hence more than one state that corresponds to w; for example, the word *flies* could either be a plural noun (the insect), or a verb (the act of flying).

In each state (w, t), the HMM emits the word w, then transitions to one of its possible next states. As an example (adapted from [14.58]), consider the sentence, *Flies like a flower.* First, if we consider each of the words separately, every one of them has more than one possible part-of-speech tag:

Flies could be a plural noun or a verb;
like could be a preposition, adverb, conjunction, noun or verb;
a could be an article or a noun, or a preposition;
flower could be a noun or a verb;

Here are two sequences of state transitions that could have produced the sentence:

- Start, (Flies, noun), (like, verb), (a, article), (flower, noun), End

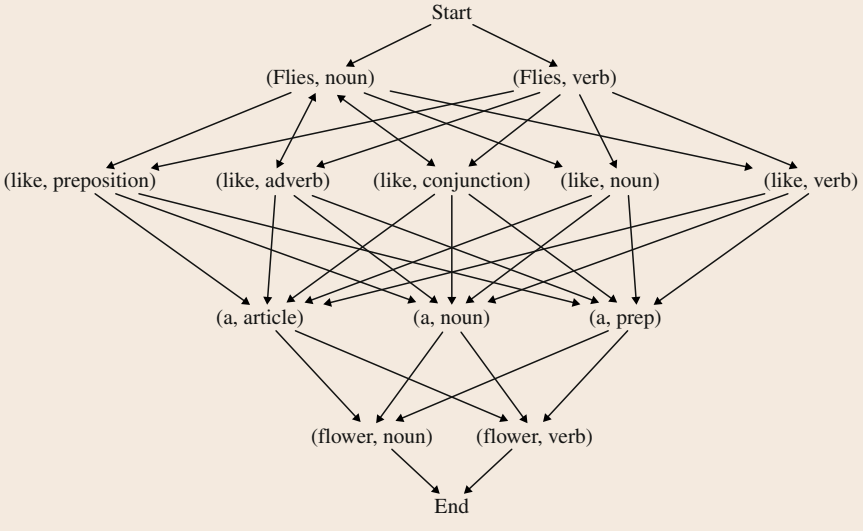

Fig. 14.10 A graphical representation of the set of all state transitions that might have produced the sentence *Flies like a flower.*

- Start, (Flies, verb), (like, preposition), (a, article), (flower, noun), End.

But there are many other state transitions that could also produce it; Fig. 14.10 shows all of them. If we know the probability of each state transition, then we can compute the probability of each possible sequence – which gives us the probability of each possible sequence of part-of-speech tags.

To establish the transition probabilities for the HMM, one needs a source of data. For NLP, these data sources are language *corpora* such as the Penn Treebank (http://www.cis.upenn.edu/~treebank/).

Context-Free Grammars

While HMMs are useful for part-of-speech tagging, it is generally accepted that they are not adequate for parsing entire sentences. The primary limitation is that HMMs, being finite-state machines, can only recognize *regular* languages, a language class that is too restricted to model several important syntactical features of human languages. A somewhat more adequate model can be provided by using *context-free grammars* [14.59].

In general, a *grammar* is a set of rewrite rules such as the following:

$$\begin{aligned} \text{Sentence} &\rightarrow \text{NounPhrase}\ \ \text{VerbPhrase} \\ \text{NounPhrase} &\rightarrow \text{Article}\ \ \text{NounPhrase1} \\ \text{Article} &\rightarrow \text{the} \mid \text{a} \mid \text{an} \\ &\ldots \end{aligned}$$

The grammar includes both *nonterminal symbols* such as *NounPhrase*, which represents an entire noun phrase, and *terminal symbols* such as *the* and *an*, which represent actual words. A *context-free grammar* is a grammar in which the left-hand side of each rule is always a single nonterminal symbol (such as Sentence in the first rewrite rule shown above).

Context-free grammars can be used to *parse* sentences into parse trees such as the one shown in Fig. 14.11, and can also be used to generate sentences. A parsing algorithm (parser) is a procedure for searching through the possible ways of combining grammatical rules to find one or more parses (i. e., one or more trees similar to the one in Fig. 14.11) that match a given sentence.

Features. While context-free grammars are better at modeling the syntax of human languages than regular grammars, there are still important features of human languages that context-free grammars cannot handle well; for example, a pronoun should not be plural unless it refers to a plural noun. One way to handle these is to augment the grammar with a set of *features* that restrict the circumstances under which different rules can be used (e.g., to restrict a pronoun to be plural if its referent is also plural).

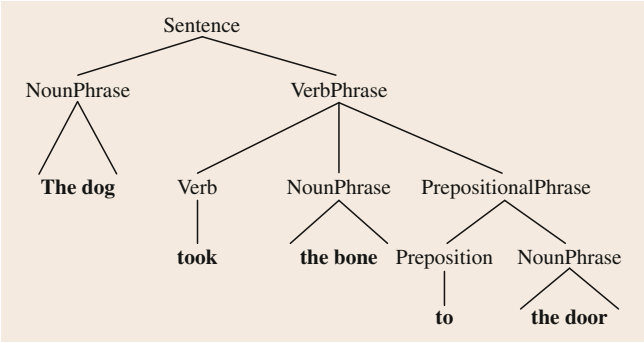

Fig. 14.11 A parse tree for the sentence *The dog took the bone to the door.*

PCFGs. If a sentence has more than one parse, one of the parses might be more likely than the others: for example, *time flies* is more likely to be a statement about time than about insects. A *probabilistic context-free grammar* (PCFG) is a context-free grammar that is augmented by attaching a probability to each grammar rule to indicate how likely different possible parses may be.

PCFGs can be learned from a parsed language corpora in a manner somewhat similar (although more complicated) than learning HMMs [14.60]. The first step is to acquire CFG rules by reading them directly from the parsed sentences in the corpus. The second step is to try to assign probabilities to the rules, test the rules on a new corpus, and remove rules if appropriate (e.g., if they are redundant or if they do not work correctly).

Applications

NLP has a large number of applications. Some examples include automated language-translation services such as Babelfish, Google Translate, Freetranslation, Teletranslator and Lycos Translation [14.61], automated speech-recogition systems used in telephone call centers, systems for categorizing, summarizing, and retrieving text (e.g., [14.62, 63]), and automated evaluation of student essays [14.64].

For additional reading on natural-language processing, see *Wu*, *Hsu*, and *Tan* [14.65] and *Thompson* [14.66].

14.1.7 Expert Systems

An *expert system* is a software system that performs, in some specialized field, at a level comparable to a human expert in the field. Most expert systems are *rule-based systems*, i. e., their *expert knowledge* consists of a set of logical inference rules similar to the Horn clauses discussed in Sect. 14.1.2.

Often these rules also have probabilities attached to them; for example, instead of writing

if A_1 and A_2 **then** conclude A_3

one might write

if A_1 and A_2 **then** conclude A_3 with probability p_0 .

Now, suppose A_1 and A_2 are known to have probabilities p_1 and p_2, respectively, and to be stochastically independent so that $P(A_1 \wedge A_2) = p_1 p_2$. Then the rule would conclude $P(C) = p_0 p_1 p_2$.

If A_1 and A_2 are not known to be stochastically independent, or if there are several rules that conclude A_3, then the computations can get much more complicated. If there are n variables $A_1, \ldots, A_n$, then the worst case could require a computation over the entire joint distribution $P(A_1, \ldots, A_n)$, which would take exponential time and would require much more information than is likely to be available to the expert system.

In some of the early expert systems, the above complication was circumvented by assuming that various events were stochastically independent even when they were not. This made the computations tractable, but could lead to inaccuracies in the results. In more modern systems, conditional independence (Sect. 14.1.3) is used to obtain more accurate results in a computationally tractable manner.

Expert systems were quite popular in the early and mid-1980s, and were used successfully in a wide variety of applications. Ironically, this very success (and the hype resulting from it) gave many potential industrial users unrealistically high expectations of what expert systems might be able to accomplish for them, leading to disappointment when not all of these expectations were met. This led to a backlash against AI, the so-called *AI winter* [14.67], that lasted for some years. but in the meantime, it became clear that simple expert systems were more elaborate versions of the decision logic already used in computer programming; hence some of the techniques of expert systems have become a standard part of modern programming practice.

Applications. Some of the better-known examples of expert-system applications include medical diagnosis [14.68], analysis of data gathered during oil exploration [14.69], analysis of DNA structure [14.70], configuration of computer systems [14.71], as well as a number of expert system *shell* (i. e., tools for building expert systems).

14.1.8 AI Programming Languages

AI programs have been written in nearly every programming language, but the most common languages for AI programming are Lisp, Prolog, C/C++, and Java.

Lisp

Lisp [14.72, 73] has many features that are useful for rapid prototyping and AI programming. These features include garbage collection, dynamic typing, functions as data, a uniform syntax, an interactive programming and debugging environment, ease of extensibility, and a plethora of high-level functions for both numeric and symbolic computations. As an example, Lisp has a built-in function, append, for concatenating two lists – but even if it did not, such a function could easily be written as follows:

```
(defun concatenate (x y)
 (if (null x)
   y
    (cons (first x)
     (concatenate (rest x) y))))
```

The above program is *tail-recursive*, i. e., the recursive call occurs at the very end of the program, and hence can easily be translated into a loop – a translation that most Lisp compilers perform automatically.

An argument often advanced in favor of conventional languages such as C++ and Java as opposed to Lisp is that they run faster, but this argument is largely erroneous. As of 2003, experimental comparisons showed compiled Lisp code to run nearly as fast as C++, and substantially faster than Java. (The speed comparison to Java might not be correct any longer, since a huge amount of work has been done since 2003 to improve Java compilers.) Probably the misconception about Lisp's speed arose from the fact that early Lisp systems ran Lisp code interpretively. Modern Lisp systems give users the option of running their code interpretively (which is useful for experimenting and

debugging) or compiling their code (which provides much higher speed).

See [14.74] for a discussion of other advantages of Lisp. One notable disadvantage of Lisp is that, if one has a computer program written in a conventional language such as C, C++ or Java, it is difficult for such a program to call a Lisp program as a subroutine: one must run the Lisp program as a separate process in order to provide the Lisp execution environment. (On the other hand, Lisp programs can quite easily invoke subroutines written in conventional programming languages.)

Applications. Lisp was quite popular during the expert-systems boom of the mid-1980s, and several *Lisp machine* computer architectures were developed and marketed in which the entire operating system was written in Lisp. Ultimately these machines did not meet with long-term commercial success, as they were eventually surpassed by less-expensive, less-specialized hardware such as Sun workstations and Intel x86 machines.

On the other hand, development of software systems in Lisp has continued, and there are many current examples of Lisp applications. A few of them include the visual lisp extension language for the AutoCAD computer-aided design system (autodesk.com), the Elisp extension language for the Emacs editor (http://en.wikipedia.org/wiki/Emacs_Lisp)the Script-Fu plugins for the GNU Image Manipulation Program (GIMP), the Remote Agent software deployed on NASA's *Deep Space 1* spacecraft [14.75], the airline fare shopping engine used by Orbitz [14.9], the SHOP2 planning system [14.38], and the Yahoo Store e-commerce software. (As of 2003, about 20 000 Yahoo stores used this software. The author does not have access to more recent statistics.)

Prolog

Prolog [14.76] is based on the notion that a general theorem-prover can be used as a programming environment in which the program consists of a set of logical statements. As an example, here is a Prolog program for concatenating lists, analogous to the Lisp program given earlier

```
concatenate([],Y,Y).
concatenate([First|Rest],Y,[First|Z]) :-
concatenate(Rest,Y,Z).
```

To concatenate two lists [a,b] and [c], one asks the theorem prover if there exists a list Z that is their concatenation; and the theorem prover returns Z if it exists

```
?- concatenate([a,b],[c],Z)
Z=[a,b,c].
```

Alternatively, if one asks whether there are lists X and Y whose concatenation is a given list Z, then there are several possible values for X and Y, and the theorem prover will return *all* of them

```
?- concatenate(X,Y,[a,b])
X = []; Y = [a,b]
X = [a]; Y = [b]
X = [a,b]; Y = []
```

One of Prolog's biggest drawbacks is that several aspects of its programming style – for example, the lack of an assignment statement, and the automated backtracking – can require workarounds that feel unintuitive to most programmers. However, Prolog can be good for problems in which logic is intimately involved, or whose solutions have a succinct logical characterization.

Applications. Prolog became popular during the the expert-systems boom of the 1980s, and was used as the basis for the Japanese Fifth Generation project [14.77], but never achieved wide commercial acceptance. On the other hand, an extension of Prolog called constraint logic programming is important in several industrial applications (see *Constraint Satisfaction and Constraint Optimization*).

C, C++, and Java

C and C++ provide much less in the way of high-level programming constructs than Lisp, hence developing code in these languages can require much more effort. On the other hand, they are widely available and provide fast execution, hence they are useful for programs that are simple and need to be both portable and fast; for example, neural networks need very fast execution in order to achieve a reasonable learning rate, and a back-propagation procedure can be written in just a few pages of C or C++ code.

Java is a lower-level language than Lisp, but is higher-level than C or C++. It uses several ideas from Lisp, most notably garbage collection. As of 2003 it ran much more slowly than Lisp, but its speed has improved in the interim and it has the advantages of being highly portable and more widely known than Lisp.

14.2 Emerging Trends and Open Challenges

AI has gone through several periods of optimism and pessimism. The most recent period of pessimism was the *AI winter* mentioned in Sect. 14.1.7. AI has emerged from this period in recent years, primarily because of the following trends. First is the exponential growth improvement in computing power: computations that used to take days or weeks can now be done in minutes or seconds. Consequently, computers have become much better able to support the intensive computations that AI often requires. Second, the pervasive role of computers in everyday life is helping to erase the apprehension that has often been associated with AI in popular culture. Third, there have been huge advances in AI research itself. AI concepts such as search, planning, natural-language processing, and machine learning have developed mature theoretical underpinnings and extensive practical histories.

AI technology is widely expected to become increasingly pervasive in applications such as data mining, information retrieval (especially from the web), and prediction of human events (including anything from sports forecasting to economics to international conflicts).

During the next decade, it appears quite likely that AI will be able to make contributions to the behavioral and social sciences analogous to the contributions that computer science has made to the biological sciences during the past decade. To make this happen, one of the biggest challenges is the huge diversity among the various research fields that will be involved. These include behavioral and social sciences such as economics, political science, psychology, anthropology, and sociology, and technical disciplines such as AI, robotics, computational linguistics, game theory, and operations research. Researchers from these fields will need to forge a common understanding of principles, techniques, and objectives. Research laboratories are being set up to foster this goal (one example is the University of Maryland's Laboratory for Computational Cultural Dynamics (http://www.umiacs.umd.edu/research/LCCD/), which is co-directed by the author of this chapter), and several international conferences and workshops on the topic have been recently established [14.78, 79].

One of the biggest challenges that currently faces AI research is its fragmentation into a bewilderingly diverse collection of subdisciplines. Unfortunately, these subdisciplines are becoming rather insular, with their own research conferences and their own (sometimes idiosyncratic) notions of what constitutes a worthwhile research question or a significant result. The achievement of human-level AI will require integrating the best efforts among many different subfields of AI, and this in turn will require better communication amongst the researchers from these subfields. I believe that the field is capable of overcoming this challenge, and that human-level AI will be possible by the middle of this century.

References

14.1 A. Newell, H.A. Simon: Computer science as empirical inquiry: Symbols and search, Assoc. Comput. Mach. Commun. **19**(3), 113–126 (1976)

14.2 S. Hedberg: Proc. Int. Conf. Artif. Intell. (IJCAI) 03 conference highlights, AI Mag. **24**(4), 9–12 (2003)

14.3 C. Kuykendall: Analyzing solitaire, Science **283**(5403), 791 (1999)

14.4 R. Bjarnason, P. Tadepalli, A. Fern: Searching solitaire in real time, Int. Comput. Games Assoc. J. **30**(3), 131–142 (2007)

14.5 E. Horowitz, S. Sahni: *Fundamentals of Computer Algorithms* (Computer Science, Potomac 1978)

14.6 N. Nilsson: *Principles of Artificial Intelligence* (Morgan Kaufmann, San Francisco 1980)

14.7 D. Navinchandra: The recovery problem in product design, J. Eng. Des. **5**(1), 67–87 (1994)

14.8 S. Russell, P. Norvig: *Artificial Intelligence, A Modern Approach* (Prentice Hall, Englewood Cliffs 1995)

14.9 S. Robinson: Computer scientists find unexpected depths in airfare search problem, SIAM News **35**(1), 1–6 (2002)

14.10 C. Le Pape: Implementation of resource constraints in ilog schedule: a library for the development of constraint-based scheduling systems, Intell. Syst. Eng. **3**, 55–66 (1994)

14.11 P. Shaw: Using constraint programming and local search methods to solve vehicle routing problems, Proc. 4th Int. Conf. Princ. Pract. Constraint Program. (1998) pp. 417–431

14.12 P. Albertand, L. Henocque, M. Kleiner: Configuration based workflow composition, IEEE Int. Conf. Web Serv., Vol. 1 (2005) pp. 285–292

14.13 J. Pearl: *Heuristics* (Addison-Wesley, Reading 1984)

14.14 R. Dechter: *Constraint Processing* (Morgan Kaufmann, San Francisco 2003)

14.15 J. Shoenfield: *Mathematical Logic* (Addison-Wesley, Reading 1967)

14.16 J. Pearl, S. Russell: Bayesian networks. In: *Handbook of Brain Theory and Neural Networks*, ed. by M.A. Arbib (MIT Press, Cambridge 2003) pp. 157–160

14.17 J. Pearl: *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* (Morgan Kaufmann, San Fransisco 1988)

14.18 M. Sahami, S. Dumais, D. Heckerman, E. Horvitz: A bayesian approach to filtering junk e-Mail. In: *Learning for Text Categorization: Papers from the 1998 Workshop* AAAI Technical Report WS-98-05 (Madison 1998)

14.19 J.A. Zdziarski: *Ending Spam: Bayesian Content Filtering and the Art of Statistical Language Classification* (No Starch, San Francisco 2005)

14.20 D. Quinlan: BayesInSpamAssassin, http://wiki.apache.org/spamassassin/BayesInSpamAssassin (2005)

14.21 K.M. Hanson: Introduction to bayesian image analysis. In: *Medical Imaging: Image Processing*, Vol. 1898, ed. by M.H. Loew, (Proc. SPIE, 1993) pp. 716–732

14.22 L. Denoyer, P. Gallinari: Bayesian network model for semistructured document classification, Inf. Process. Manage. **40**, 807–827 (2004)

14.23 S. Fox, K. Karnawat, M. Mydland, S. Dumais, T. White: Evaluating implicit measures to improve web search, ACM Trans. Inf. Syst. **23**(2), 147–168 (2005)

14.24 L. Zadeh, G.J. Klir, Bo Yuan: *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi Zadeh* (World Scientific, River Edge 1996)

14.25 G.J. Klir, U.S. Clair, B. Yuan: *Fuzzy Set Theory: Foundations and Applications* (Prentice Hall, Englewood Cliffs 1997)

14.26 A. Tate: Planning. In: *MIT Encyclopedia of the Cognitive Sciences*, (1999) pp. 652–653

14.27 T. Estlin, R. Castano, B. Anderson, D. Gaines, F. Fisher, M. Judd: Learning and planning for Mars rover science, Proc. Int. Joint Conf. Artif. Intell. (IJCAI) (2003)

14.28 S.K. Gupta, D.A. Bourne, K. Kim, S.S. Krishanan: Automated process planning for sheet metal bending operations, J. Manuf. Syst. **17**(5), 338–360 (1998)

14.29 A. Gerevini, D. Long: Plan constraints and preferences in pddl3: The language of the fifth international planning competition. Technical Report (University of Brescia, 2005), available at http://cs-www.cs.yale.edu/homes/dvm/papers/pddl-ipc5.pdf

14.30 A.L. Blum, M.L. Furst: Fast planning through planning graph analysis, Proc. Int. Joint Conf. Artif. Intell. (IJCAI) (1995) pp. 1636–1642

14.31 A. Tate, B. Drabble, R. Kirby: *O-Plan2: An Architecture for Command, Planning and Control* (Morgan Kaufmann, San Francisco 1994)

14.32 D.E. Wilkins: *Practical Planning: Extending the Classical AI Planning Paradigm* (Morgan Kaufmann, San Mateo 1988)

14.33 D. Nau, T.-C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, F. Yaman: SHOP2: An HTN planning system, J. Artif. Intell. Res. **20**, 379–404 (2003)

14.34 F. Bacchus, F. Kabanza: Using temporal logics to express search control knowledge for planning, Artif. Intell. **116**(1/2), 123–191 (2000)

14.35 J. Kvarnström, P. Doherty: TALplanner: A temporal logic based forward chaining planner, Ann. Math. Artif. Intell. **30**, 119–169 (2001)

14.36 M. Fox, D. E. Smith: Special track on the 4th international planning competition. J. Artif. Intell. Res. (2006), available at http://www.jair.org/specialtrack.html

14.37 B. Bonet, H. Geffner: Labeled RTDP: Improving the convergence of real-time dynamic programming, Proc. 13th Int. Conf. Autom. Plan. Sched. (ICAPS) (AAAI, 2003), pp 12–21

14.38 D. Nau, T.-C. Au, O. Ilghami, U. Kuter, H. Muñoz-Avila, J.W. Murdock, D. Wu, F. Yaman: Applications of SHOP and SHOP2, IEEE Intell. Syst. **20**(2), 34–41 (2005)

14.39 M. Boddy, J. Gohde, J.T. Haigh, S. Harp: Course of action generation for cyber security using classical planning, Proc. 15th Int. Conf. Autom. Plan. Sched. (ICAPS) (2005)

14.40 M. Ghallab, D. Nau, P. Traverso: *Automated Planning: Theory and Practice* (Morgan Kaufmann, San Francisco 2004)

14.41 S.M. Lavalle: *Planning Algorithms* (Cambridge University Press, Cambridge 2006)

14.42 J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen: Checkers is solved, Science **317**(5844), 1518–1522 (2007)

14.43 D.E. Knuth, R.W. Moore: An analysis of alpha-beta pruning, Artif. Intell. **6**, 293–326 (1975)

14.44 G. Tesauro: *Programming Backgammon Using Self-Teaching Neural Nets* (Elsevier, Essex 2002)

14.45 S.J.J. Smith, D.S. Nau, T. Throop: Computer bridge: A big win for AI planning, AI Mag. **19**(2), 93–105 (1998)

14.46 M. Harris: Laak-Eslami team defeats Polaris in man-machine poker championship, Poker News (2007), Available at www.pokernews.com/news/2007/7/laak-eslami-team-defeats-polaris-man-machine-poker-championship.htm

14.47 A. Parker, D. Nau, V.S. Subrahmanian: Overconfidence or paranoia? Search in imperfect-information games, Proc. Natl. Conf. Artif. Intell. (AAAI) (2006)

14.48 D. Billings: Thoughts on RoShamBo, Int. Comput. Games Assoc. J. **23**(1), 3–8 (2000)

14.49 B. Johanson: Robust strategies and counter-strategies: Building a champion level computer poker player. Master Thesis (University of Alberta, 2007)

14.50 S. Hart, R.J. Aumann (Eds.): *Handbook of Game Theory with Economic Applications 2* (North Holland, Amsterdam 1994)

14.51 F.-H. Hsu: Chess hardware in deep blue, Comput. Sci. Eng. **8**(1), 50–60 (2006)

14.52 J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers* (Springer, Berlin, Heidelberg 1997)

14.53 J. Schaeffer: A gamut of games, AI Mag. **22**(3), 29–46 (2001)

14.54 T. Sandholm: Expressive commerce and its application to sourcing, Proc. Innov. Appl. Artif. Intell. Conf. (IAAI) (AAAI Press, Menlo Park 2006)

14.55 T.-C. Au, D. Nau: Accident or intention: That is the question (in the iterated prisoner's dilemma), Int. Joint Conf. Auton. Agents and Multiagent Syst. (AAMAS) (2006)

14.56 C. Chelba, F. Jelinek: Structured language modeling for speech recognition, Conf. Appl. Nat. Lang. Inf. Syst. (NLDB) (1999)

14.57 B.H. Juang, L.R. Rabiner: Hidden Markov models for speech recognition, Technometrics **33**(3), 251–272 (1991)

14.58 S. Lee, J. Tsujii, H. Rim: Lexicalized hidden Markov models for part-of-speech tagging, Proc. 18th Int. Conf. Comput. Linguist. (2000)

14.59 N. Chomsky: *Syntactic Structures* (Mouton, The Hague 1957)

14.60 E. Charniak: A maximum-entropy-inspired parser, Technical Report CS-99-12 (Brown University 1999)

14.61 F. Gaspari: *Online MT Services and Real Users Needs: An Empirical Usability Evaluation* (Springer, Berlin, Heidelberg 2004), pp 74–85

14.62 P. Jackson, I. Moulinier: *Natural Language Processing for Online Applications: Text Retrieval, Extraction, and Categorization* (John Benjamins, Amsterdam 2002)

14.63 M. Sahami, T.D. Heilman: A web-based kernel function for measuring the similarity of short text snippets, WWW '06: Proc. 15th Int. Conf. World Wide Web (New York 2006) pp. 377–386

14.64 J. Burstein, M. Chodorow, C. Leacock: Automated essay evaluation: the criterion online writing service, AI Mag. **25**(3), 27–36 (2004)

14.65 Zhi Biao Wu, Loke Soo Hsu, Chew Lim Tan: A survey of statistical approaches to natural language processing, Technical Report TRA4/92 (National University of Singapore 1992)

14.66 C. A. Thompson: A brief introduction to natural language processing for nonlinguists. In: *Learning Language in Logic*, Lecture Notes in Computer Science (LNCS) Ser., Vol. 1925, ed. by J. Cussens, S. Dzeroski (Springer, New York 2000) pp. 36–48

14.67 H. Havenstein: Spring comes to AI winter, Computerworld (2005), available at http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=99691

14.68 B.G. Buchanan, E.H. Shortliffe (Eds.): *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project* (Addison-Wesley, Reading 1984)

14.69 R.G. Smith, J.D. Baker: The dipmeter advisor system – a case study in commercial expert system development, Proc. Int. Joint Conf. Artif. Intell. (IJCAI) (1983) pp. 122–129

14.70 M. Stefik: Inferring DNA structures from segmentation data, Artif. Intell. **11**(1/2), 85–114 (1978)

14.71 J.P. McDermott: R1 ("XCON") at age 12: Lessons from an elementary school achiever, Artif. Intell. **59**(1/2), 241–247 (1993)

14.72 G. Steele: *Common Lisp: The Language*, 2nd edn. (Digital, Woburn 1990)

14.73 P. Graham: *ANSI Common Lisp* (Prentice Hall, Englewood Cliffs 1995)

14.74 P. Graham: *Hackers and Painters: Big Ideas from the Computer Age* (O'Reilly Media, Sebastopol 2004) pp. 165–180, also available at http://www.paulgraham.com/avg.html

14.75 N. Muscettola, P. Pandurang Nayak, B. Pell, B.C. Williams: Remote agent: To boldly go where no AI system has gone before, Artif. Intell. **103**(1/2), 5–47 (1998)

14.76 W. Clocksin, C. Mellish: *Programming in Prolog* (Springer, Berlin, Heidelberg 1981)

14.77 E. Feigenbaum, P. McCorduck: *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World* (Addison-Wesley Longman, Boston 1983)

14.78 D. Nau, J. Wilkenfeld (Ed): Proc. 1st Int. Conf. Comput. Cult. Dyn. (ICCCD-2007) (AAAI Press, Menlo Park 2007)

14.79 H. Liu, J. Salerno, M. Young (Ed): Proc. 1st Int. Workshop Soc. Comput. Behav. Model. Predict. (Springer, Berlin, Heidelberg 2008)