plishments in modeling the semantic complexities of a natural language such as English have also been very modest. Even fundamental issues such as organizing knowledge or fully managing the complexity and correctness of very large computer programs (such as large knowledge bases) require considerable further research. Expert systems, while they have achieved marketable engineering successes, still have many limitations in the quality and generality of their reasoning. These include their inability to perform *commonsense reasoning* or to exhibit knowledge of rudimentary physical reality, such as how things change over time.

But we must maintain a reasonable perspective. It is easy to overlook the accomplishments of artificial intelligence when honestly facing the work that remains. In the next section, we establish this perspective through an overview of several areas of artificial intelligence research and development.

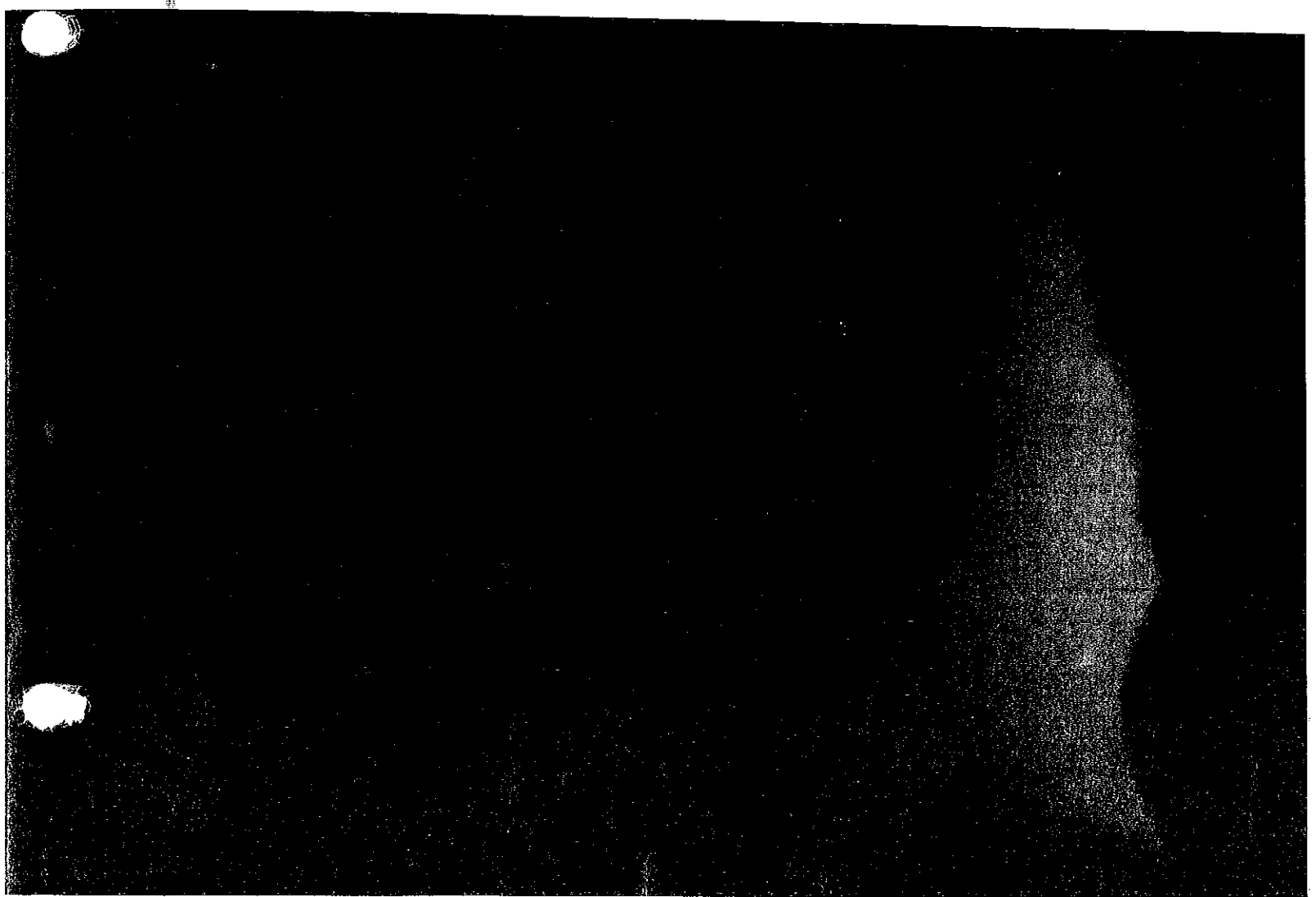
1.2 Overview of AI Application Areas

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.

—ADA BYRON, *Countess of Lovelace*

I'm sorry Dave; I can't let you do that.

—HAL 9000 in *2001: A Space Odyssey* by Arthur C. Clarke



We now return to our stated goal of defining artificial intelligence through an examination of the ambitions and accomplishments of workers in the field. The two most fundamental concerns of AI researchers are *knowledge representation* and *search*. The first of these addresses the problem of capturing the full range of knowledge required for intelligent behavior in a formal language, i.e., one suitable for computer manipulation. Chapter 2 introduces predicate calculus as a language for describing the properties and relationships among objects in problem domains that require qualitative reasoning rather than arithmetic calculations for their solutions. Later chapters (9, 10, 11, and 14) discuss other languages that artificial intelligence has developed for better capturing the ambiguities and complexities of areas such as commonsense reasoning and natural language understanding.

Search is a problem-solving technique that systematically explores a space of *problem states*, i.e., successive and alternative stages in the problem-solving process. Examples of problem states might include the different board configurations in a game or intermediate steps in a reasoning process. This space of alternative solutions is then searched to find a final answer. Newell and Simon (1976) have argued that this is the essential basis of human problem solving. Indeed, when a chess player examines the effects of different moves or a doctor considers a number of alternative diagnoses, they are searching among alternatives. The implications of this model and techniques for its implementation are discussed in chapters 3, 4, and 5.

Like most sciences, AI is decomposed into a number of subdisciplines that, while sharing an essential approach to problem solving, have concerned themselves with different applications. In this section we outline several of these major application areas and their contributions to artificial intelligence as a whole.

1.2.1 Game Playing

Much of the early research in state space search was done using common board games such as checkers, chess, and the 16-puzzle. In addition to their inherent intellectual appeal, board games have certain properties that made them ideal subjects for this early work. Most games are played using a well-defined set of rules: this makes it easy to generate the search space and frees the researcher from many of the ambiguities and complexities inherent in less structured problems. The board configurations used in playing these games are easily represented on a computer, requiring none of the complex formalisms needed to capture the semantic subtleties of real-world problem domains. Since games can be easily played, testing a game-playing program presents no financial or ethical burden. State space search, the paradigm underlying most game-playing research, is presented in chapters 3 and 4.

Games can generate extremely large search spaces. These are large and complex enough to require powerful techniques for determining what alternatives to explore in the problem space. These techniques are called *heuristics* and constitute a major area of AI research. A heuristic is a useful but potentially fallible problem-solving strategy, such as checking to make sure that an unresponsive appliance is plugged in before assuming that it is broken, or trying to protect your queen from capture in a chess game. Much of what we call intelligence resides in the heuristics used by humans to solve problems.

Because most of us have some experience with these simple games, it is possible to devise and test the effectiveness of our own heuristics. We do not need to find and consult an expert in some esoteric problem area such as medicine or mathematics (chess is an obvious exception to this rule). For these reasons, games provide a rich domain for the study of heuristic search. Chapter 5 introduces heuristics using these simple games; Chapter 8 extends their application to expert systems.

Game-playing programs, in spite of their simplicity, offer their own challenges, including an opponent whose moves may not be reliably anticipated. This presence of the opponent further complicates program design by adding an element of unpredictability and the need to consider psychological as well as tactical factors in game strategy.

1.2.2 Automated Reasoning and Theorem Proving

We could argue that automatic theorem proving is the oldest branch of artificial intelligence, tracing its roots back through Newell and Simon's Logic Theorist (Newell and Simon 1963*a*) and General Problem Solver (Newell and Simon 1963*b*) to its origins in Russell and Whitehead's efforts to treat all of mathematics as the purely formal derivation of theorems from basic axioms. In any case, it has certainly been one of the most fruitful branches of the field. Theorem-proving research was responsible for much of the early work in formalizing search algorithms and developing formal representation languages such as predicate calculus (Chapter 2) and the logic programming language PROLOG (Chapter 12).

Most of the appeal of automated theorem proving lies in the rigor and generality of logic. Because it is a formal system, logic lends itself to automation. A wide variety of problems can be attacked by representing the problem description and relevant background information as logical axioms and treating problem instances as theorems to be proved. This insight is the basis of work in automatic theorem proving and mathematical reasoning systems (Chapter 11).

Unfortunately, early efforts at writing theorem provers failed to develop a system that could consistently solve complicated problems. This was due to the ability of any reasonably complex logical system to generate an infinite number of provable theorems: without powerful techniques (heuristics) to guide their search, automated theorem provers proved large numbers of irrelevant theorems before stumbling onto the correct one. In response to this inefficiency, many argue that purely formal, syntactic methods of guiding search are inherently incapable of handling such a huge space and that the only alternative is to rely on the informal, ad hoc strategies that humans seem to use in solving problems. This is the approach underlying the development of expert systems, and it has proved to be a fruitful one.

Still, the appeal of reasoning based in formal mathematical logic is too strong to ignore. Many important problems such as the design and verification of logic circuits, verification of the correctness of computer programs, and control of complex systems seem to respond to such an approach. In addition, the theorem-proving community has enjoyed increasing success in devising powerful heuristics that rely solely on an evaluation of the syntactic form of a logical expression, reducing the complexity of the search space without resorting to the ad hoc techniques used by human problem solvers.

Another reason for the continued interest in automatic theorem provers is the realization that such a system does not have to be capable of independently solving extremely complex problems without human assistance. Many modern theorem provers function as intelligent assistants, letting humans perform the more demanding tasks of decomposing a large problem into subproblems and devising heuristics for searching the space of possible proofs. The theorem prover then performs the simpler but still demanding task of proving lemmas, verifying smaller conjectures, and completing the formal aspects of a proof outlined by its human associate (Boyer and Moore 1979).

1.2.3 Expert Systems

One major insight gained from early work in problem-solving was the importance of domain-specific knowledge. A doctor, for example, is not effective at diagnosing illness solely because she possesses some innate general problem-solving skill; she is effective because she knows a lot about medicine. Similarly, a geologist is effective at discovering mineral deposits because he is able to apply a good deal of theoretical and empirical knowledge about geology to the problem at hand. Expert knowledge is a combination of a theoretical understanding of the problem and a collection of heuristic problem-solving rules that experience has shown to be effective in the domain. Expert systems are constructed by obtaining this knowledge from a human expert and coding it into a form that a computer may apply to similar problems.

This reliance on the knowledge of a human domain expert for the system's problem-solving strategies is a major feature of expert systems. Although some programs are written in which the designer is also the source of the domain knowledge, it is far more typical to see such programs growing out of a collaboration between a domain expert such as a doctor, chemist, geologist, or engineer and a separate artificial intelligence specialist. The domain expert provides the necessary knowledge of the problem domain through a general discussion of her problem-solving methods and by demonstrating those skills on a carefully chosen set of sample problems. The AI specialist, or *knowledge engineer*, as expert systems designers are often known, is responsible for implementing this knowledge in a program that is both effective and seemingly intelligent in its behavior. Once such a program has been written, it is necessary to refine its expertise through a process of giving it example problems to solve, letting the domain expert criticize its behavior, and making any required changes or modifications to the program's knowledge. This process is repeated until the program has achieved the desired level of performance.

EXAMPLES:

(A)

One of the earliest systems to exploit domain-specific knowledge in problem solving was DENDRAL, developed at Stanford in the late 1960s (Lindsay et al. 1980). DENDRAL was designed to infer the structure of organic molecules from their chemical formulas and mass spectrographic information about the chemical bonds present in the molecules. Because organic molecules tend to be very large, the number of possible structures for these molecules tends to be huge. DENDRAL addresses the problem of this large search space by applying the heuristic knowledge of expert chemists to the structure elucidation problem. DENDRAL's methods proved remarkably effective, routinely finding the correct structure out of millions of possibilities after only a few trials.

The approach has proved so successful that descendants of the system are used in chemical laboratories throughout the country.

While DENDRAL was one of the first programs to effectively use domain-specific knowledge to achieve expert level problem-solving performance, MYCIN established the methodology of contemporary expert systems (Buchanan 1984). MYCIN uses expert medical knowledge to diagnose and prescribe treatment for spinal meningitis and bacterial infections of the blood.

MYCIN, developed at Stanford in the mid-1970s, was one of the first programs to address the problems of reasoning with uncertain or incomplete information. MYCIN provided clear and logical explanations of the reasoning, used a control structure appropriate to the specific problem domain, and identified criteria to reliably evaluate its performance. Many of the expert system development techniques currently in use were first developed in the MYCIN project (Chapter 8).

Other classic expert systems include the PROSPECTOR program for determining the probable location and type of ore deposits based on geological information about a site (Duda et al. 1979*a*, 1979*b*), the INTERNIST program for performing diagnosis in the area of internal medicine, the Dipmeter Advisor developed to interpret the results of oil well drilling logs (Smith and Baker 1983), and XCON for configuring VAX computers. XCON has been in use since 1981; every VAX sold by Digital Equipment Corporation is now configured by XCON. Numerous other expert systems are currently solving problems in areas such as medicine, education, business, design, and science (Waterman 1986).

It is interesting to note that most expert systems have been written for relatively specialized, expert level domains. These domains are generally well studied and have clearly defined problem-solving strategies. Problems that depend on a more loosely defined notion of "common sense" are much more difficult to solve by these means (Part IV).

In spite of the promise of expert systems, it would be a mistake to overestimate the ability of this technology. Current deficiencies include:

1. Difficulty in capturing "deep" knowledge of the problem domain. MYCIN, for example, lacks any real knowledge of human physiology. It does not know what blood does or the function of the spinal cord. Folklore has it that once, when selecting a drug for treatment of meningitis, MYCIN asked if the patient was pregnant, even though it had been told that the patient was male. Whether this actually occurred or not, it does illustrate the narrowness of current expert systems.
2. Lack of robustness and flexibility. If humans are presented with a problem instance that they cannot solve immediately, they can generally return to an examination of first principles and come up with some strategy for attacking the problem. Expert systems generally lack this ability.
3. Inability to provide deep explanations. Because expert systems lack deep knowledge of their problem domains, their explanations are generally restricted to a description of the steps they took in finding a solution. They cannot tell "why" a certain approach was taken.

4. Difficulties in verification. While the correctness of any large computer system is difficult to prove, expert systems are particularly difficult to verify. This is a serious problem, as expert systems technology is being applied to critical applications such as air traffic control, nuclear reactor operations, and weapons systems.
5. Little learning from experience. Current expert systems are handcrafted; once the system is completed, its performance will not improve without further attention from its programmers. This leads to severe doubts about the intelligence of such systems.

In spite of these limitations, expert systems are proving their value in a number of important applications. It is hoped that these limitations will only encourage the student to pursue study and research in this important new branch of computer science. Expert systems are a major topic in this text and are discussed in chapters 4, 8, 12, 13, and 14.

1.2.4 Natural Language Understanding and Semantic Modeling

One of the long-standing goals of artificial intelligence is the creation of programs that are capable of understanding human language. Not only does the ability to understand natural language seem to be one of the most fundamental aspects of human intelligence, but also its successful automation would have an incredible impact on the usability and effectiveness of computers themselves. Much effort has been put into writing programs that understand natural language; although these programs have achieved success within certain restricted contexts, systems that can use natural language with the flexibility and generality that characterize human speech are beyond current methodologies.

Understanding natural language involves much more than parsing sentences into their individual parts of speech and looking those words up in a dictionary. Real understanding depends on extensive background knowledge about the domain of discourse and the idioms used in that domain and an ability to apply general contextual knowledge to resolve the omissions and ambiguities that are a normal part of human speech.

Consider, for example, the difficulties in carrying on a conversation about baseball with an individual who understands English but knows nothing about the rules, players, or history of the game. Could this person possibly understand the meaning of the sentence: "With none down in the top of the ninth and the go-ahead run at second, the manager called his relief from the bull pen"? Even though all of the words in the sentence may be individually understood, this sentence would be gibberish to even the most intelligent non-baseball fan.

The task of collecting and organizing this background knowledge in such a way that it may be applied to language comprehension forms the major problem in automating natural language understanding. Responding to this need, researchers have developed many of the techniques for structuring semantic meaning used throughout artificial intelligence (chapters 9 and 10).

Because of the tremendous amounts of knowledge required for understanding natural language, most work is done in well-understood, specialized problem areas. One of the earliest programs to exploit this "micro world" methodology was Winograd's

SHRDLU, a natural language system that could "converse" about a simple configuration of blocks of different shapes and colors (Winograd 1973). SHRDLU could answer queries such as "what color block is on the blue cube?" and plan actions such as "move the red pyramid onto the green brick." Problems of this sort, involving the description and manipulation of simple arrangements of blocks, have appeared with surprising frequency in AI research and are known as "blocks world" problems.

In spite of SHRDLU's success in conversing about arrangements of blocks, its methods did not generalize from the blocks world. The representational techniques used in the program were too simple to capture the semantic organization of much richer real-world domains in a useful way. Much of the current work in natural language understanding is devoted to finding representational formalisms that are general enough to be used in a wide range of applications, yet adapt themselves well to the specific structure of a given domain. A number of different techniques (most of which are extensions or modifications of *semantic networks*) are explored for this purpose and used in the development of programs that can understand natural language in constrained but interesting knowledge domains. General natural language understanding, however, remains beyond the current state of the art.

1.2.5 Modeling Human Performance

Although much of the above discussion uses human intelligence as a reference point in considering artificial intelligence, it does not follow that programs should pattern themselves after the organization of the human mind. Indeed, many AI programs are engineered to solve some useful problem without regard for their similarities to human mental architecture. Even expert systems, while deriving much of their knowledge from human experts, do not really attempt to simulate human internal mental processes. If performance is the only criterion by which a system will be judged, there may be little reason to attempt to simulate human problem-solving methods; in fact, programs that take nonhuman approaches to solving problems are often more successful than their human counterparts. Still, the design of systems that explicitly model some aspect of human problem solving has been a fertile area of research in both artificial intelligence and psychology.

Human performance modeling, in addition to providing AI with much of its basic methodology, has proved to be a very powerful tool for formulating and testing theories of human cognition. The problem-solving methodologies developed by computer scientists have given psychologists a new metaphor for exploring the human mind. Rather than casting theories of cognition in the vague language used in early research, or abandoning the problem of describing the inner workings of the human mind entirely (as suggested by the behaviorists), many psychologists have adopted the language and theory of computer science to formulate models of human intelligence. Not only do these techniques provide a new vocabulary for describing human intelligence, but also computer implementations of these theories offer psychologists an opportunity to empirically test, critique, and refine their ideas. Further discussion of the relationship between artificial intelligence and efforts to understand human intelligence is found throughout the text and summarized in Chapter 15.

1.2.6 Planning and Robotics

Planning is an important aspect of the effort to design robots that perform their task with some degree of flexibility and responsiveness to the outside world. Briefly, planning assumes a robot that is capable of performing certain atomic actions. It attempts to find a sequence of those actions that will accomplish some higher-level task such as moving across an obstacle-filled room.

Planning is a difficult problem for a number of reasons, not the least of which is the size of the space of possible sequences of moves. Even an extremely simple robot is capable of generating a vast number of potential move sequences. Imagine, for example, a robot that can move forward, backward, right, or left, and consider how many different ways that robot can possibly move around a room. Assume also that there are obstacles in the room and that the robot must select a path that moves around them in some efficient fashion. Writing a program that can intelligently discover the best path under these circumstances, without being overwhelmed by the huge number of possibilities, requires sophisticated techniques for representing spatial knowledge and controlling search.

One method that human beings use in planning is *hierarchical problem decomposition*. If you are planning a trip to London, you will generally treat the problems of arranging a flight, getting to the airport, making airline connections, and finding ground transportation in London separately, even though they are all part of a bigger overall plan. Each of these may be further decomposed into smaller subproblems such as finding a map of the city, negotiating the subway system, and finding a decent pub. Not only does this approach effectively restrict the size of the space that must be searched, but also you can save frequently used subplans for future use.

While humans plan effortlessly, creating a computer program that can do the same is a difficult challenge. A seemingly simple task such as breaking a problem into independent subproblems actually requires sophisticated heuristics and extensive knowledge about the planning domain. Determining what subplans should be saved and how they may be generalized for future use is an even harder problem, which is currently explored in research situations (Chapter 15).

A robot that blindly performs a sequence of actions without responding to changes in its environment or being able to detect and correct errors in its own plan could hardly be considered intelligent. Often, a robot will have to formulate a plan based on incomplete information and correct its behavior as it executes the plan. A robot may not have adequate sensors to locate all obstacles in the way of a projected path. Such a robot must begin moving through the room based on what it has "perceived" and correct its path as other obstacles are detected. Organizing plans in a fashion that allows response to environmental conditions is another major problem for planning.

1.2.7 Languages and Environments for AI

Some of the most important by-products of artificial intelligence research are advances in programming languages and software development environments. For a number of reasons, including the sheer size of most AI application programs, the tendency of search algorithms to generate huge spaces, and the difficulty of predicting the behavior of heu-

ristically driven programs, AI programmers have been forced to develop a powerful set of programming methodologies.

Programming environments include knowledge-structuring techniques such as object-oriented programming and expert systems frameworks (these are discussed in Part IV). High-level languages, such as LISP and PROLOG, which strongly support modular development, help manage program size and complexity. Trace packages allow a programmer to reconstruct the execution of a complex algorithm and make it possible to unravel the complexities of heuristically guided search. Without such tools and techniques, it is doubtful that many significant AI systems could have been built.

Many of these techniques are now standard tools for software engineering and have little relationship to the core of AI theory. Others, such as object-oriented programming, are of significant theoretical and practical interest (Chapters 9, 12, 13, and 14).

The languages developed for artificial intelligence programming are intimately bound to the theoretical structure of the field. We cover both LISP and PROLOG in this text and prefer to remain apart from religious debates over the relative merits of these languages. Rather, we adhere to the adage "a good workman knows all of his (or her) tools." The language chapters (6, 7, 12, and 13) discuss the advantages of each language for specific programming tasks.

1.2.8 Machine Learning

Learning has remained a difficult problem for AI programs, in spite of their success as problem solvers. This shortcoming seems severe, particularly since the ability to learn is one of the most important components of intelligent behavior. An expert system may perform extensive and costly computations to solve a problem. Unlike a human being, however, if it is given the same or a similar problem a second time, it will not remember the solution. It performs the same sequence of computations again. This is true the second, third, fourth, and every time it solves that problem—hardly the behavior of an intelligent problem solver.

Most expert systems are hindered by the inflexibility of their problem-solving strategies and the difficulty of modifying large amounts of code. The obvious solution to these problems is for programs to learn on their own, either from experience, analogy, and examples or by being "told" what to do.

Learning is a difficult area of research. Several programs have been written that suggest that this is not an impossible goal. Perhaps the most striking such program is AM, the Automatic Mathematician, designed to discover mathematical laws (Lenat 1977, 1982). Initially given the concepts and axioms of set theory, AM was able to induce such important mathematical concepts as cardinality and integer arithmetic and many of the results of number theory. AM conjectured new theorems by modifying its current knowledge base and used heuristics to pursue the most "interesting" of a number of possible alternatives.

Other influential work includes Patrick Winston's research on the induction of structural concepts such as "arch" from a set of examples in the blocks world (Winston 1975*a*). Meta-DENDRAL learns rules for interpreting mass spectrographic data in organic chemistry from examples of data on known compounds of known structure.

Teiresias, an intelligent "front end" for expert systems, converts high-level advice into new rules for its knowledge base (Davis 1982). Hacker devises plans for performing blocks world manipulations through an iterative process of devising a plan, testing it, and correcting any flaws discovered in the candidate plan (Sussman 1975).

The success of these and other machine learning programs suggests the existence of a set of general learning principles that will allow the construction of programs with the ability to learn in realistic domains. We discuss machine learning in more detail in Chapter 15.

1.3 Artificial Intelligence—A Summary

We have attempted to define artificial intelligence through a discussion of the major areas of research and application in the field. This survey reveals a young and promising field of study whose primary concern is finding an effective way to understand and apply intelligent problem-solving, planning, and communication skills to a wide range of practical problems. In spite of the variety of problems addressed in artificial intelligence research, a number of important features emerge that seem common to all divisions of the field; these include:

1. The use of computers to do symbolic reasoning.
2. A focus on problems that do not respond to algorithmic solutions. This underlies the reliance on heuristic search as an AI problem-solving technique.
3. A concern with problem solving using inexact, missing, or poorly defined information and the use of representational formalisms that enable the programmer to compensate for these problems.
4. An effort to capture and manipulate the significant qualitative features of a situation rather than relying on numeric methods.
5. An attempt to deal with issues of semantic meaning as well as syntactic form.
6. Answers that are neither exact nor optimal, but are in some sense "sufficient." This is a result of the essential reliance on heuristic problem-solving methods in situations where optimal or exact results are either too expensive or not possible.
7. The use of large amounts of domain-specific knowledge in solving problems. This is the basis of expert systems.
8. The use of meta-level knowledge to effect more sophisticated control of problem-solving strategies. Although this is a very difficult problem, addressed in relatively few current systems, it is emerging as an essential area of research.

We hope that this introduction provides some feel for the overall structure and significance of the field of artificial intelligence. We also hope that the brief discussions of such technical issues as search and representation were not excessively cryptic and obscure; they are developed in proper detail throughout the remainder of the text. They were included here to demonstrate their significance in the more general organization of the field.

As we mentioned in the discussion of knowledge representation, objects take on meaning through their relationships with other objects. This is equally true of the facts, theories, and techniques that constitute a field of scientific study. We have intended to give a sense of those interrelationships, so that when the separate technical themes of artificial intelligence are presented, they will find their place in a developing understanding of the overall substance and directions of the field. In writing this text we were guided by an observation made by Gregory Bateson, the psychologist and systems theorist (Bateson 1979):

> Break the pattern which connects the items of learning and you necessarily destroy all quality.

1.4 Epilogue and References

The challenging field of AI reflects some of the oldest concerns of Western civilization in the light of the modern computational model. The notions of rationality, representation, and reason are now under scrutiny as perhaps never before, since we computer scientists demand to understand them operationally, even algorithmically! At the same time, the political, economic, and ethical situation of our species forces us to confront our responsibility for the effects of our artifices. The interplay between applications and the more humanistic inspirations for much of AI continues to inspire hosts of rich, challenging questions. We hope you will glean from the following chapters both a familiarity with the contemporary concepts and techniques of AI and an appreciation for the timelessness of the problems they address.

Several excellent sources available on the topics raised in this chapter are *Mind Design* (Haugheland 1981), *Artificial Intelligence: The Very Idea* (Haugheland 1985), *Brainstorms* (Dennett 1978), and *Elbow Room* (Dennett 1984). Several of the primary sources are also readily available, including Aristotle's *Physics, Metaphysics,* and *Logic,* papers by Frege, and the writings of Babbage, Boole, and Russell and Whitehead. Turing's papers are also very interesting, especially his discussions of the nature of intelligence and the possibility of designing intelligent programs (Turing, 1950). Turing's biography, *Alan Turing: The Enigma* (Hodges 1983), also makes excellent reading.

Computer Power and Human Reason (Weizenbaum 1976) and *Understanding Computers and Cognition* (Winograd and Flores 1986) offer some sobering comments on the limitations of and ethical issues in AI. *The Sciences of the Artificial* (Simon 1981) is a positive statement on the possibility of artificial intelligence and its role in society.

The AI applications mentioned in Section 1.2 are intended to introduce the reader to the broad interests of AI researchers and outline many of the important questions under investigation. The *Handbook of Artificial Intelligence* (Barr and Feigenbaum 1981) offers an introduction to each of these areas. Besides the *Handbook* we recommend, for extended treatment of game playing, *Principles of Artificial Intelligence* (Nilsson 1980) and *Heuristics* (Pearl 1984); this is also an important topic in our chapters 2 through 5. We discuss automated reasoning in chapters 2, 3, and 11; some of the highlights in the literature of automated reasoning are *Automated Reasoning* (Wos et al. 1984), "Non-

SYMBOLIC A.I.

8 RULE-BASED EXPERT SYSTEMS

(ALMOST ALWAYS RULE-BASED)

The first principle of knowledge engineering is that the problem solving power exhibited by an intelligent agent's performance is primarily the consequence of its knowledge base, and only secondarily a consequence of the inference method employed. Expert systems must be knowledge-rich even if they are methods-poor. This is an important result and one that has only recently become well understood in AI. For a long time AI has focused its attentions almost exclusively on the development of clever inference methods; almost any inference method will do. The power resides in the knowledge.

—EDWARD FEIGENBAUM, Stanford University

I hate to criticize, Dr Davis, but did you know that
most rules about what the category of
an organism might be
that mention:
the site of a culture and
the infection
also mention:
the portal of entry of the organism?
Shall I try to write a clause to account for this?

—The program Teiresias helping a Stanford physician improve the MYCIN knowledge base

8.0 Introduction

An *expert system* is a knowledge-based program that provides "expert quality" solutions to problems in a specific domain. Generally, its knowledge is extracted from human experts in the domain and it attempts to emulate their methodology and performance. As with skilled humans, expert systems tend to be specialists, focusing on a narrow set of problems. Also, like humans, their knowledge is both theoretical and practical, having been perfected through experience in the domain. Unlike a human being,

however, current programs cannot learn from their own experience; their knowledge must be extracted from humans and encoded in a formal language. This is the major task facing expert system builders.

Expert systems should not be confused with cognitive modeling programs, which attempt to simulate human mental architecture in detail. These are discussed in Chapter 15. Expert systems neither copy the structure of the human mind nor are mechanisms for general intelligence. They are practical programs that use heuristic strategies developed by humans to solve specific classes of problems.

Because of the heuristic, knowledge-intensive nature of expert level problem solving, expert systems are generally:

E.S. ARE:

1. Open to inspection, both in presenting intermediate steps and in answering questions about the solution process.
2. Easily modified, both in adding and in deleting skills from the knowledge base.
3. Heuristic, in using (often imperfect) knowledge to obtain solutions.

1)

An expert system is "open to inspection" in that the user may, at any time during program execution, inspect the state of its reasoning and determine the specific choices and decisions that the program is making. There are several reasons why this is desirable: first, if a human expert such as a doctor or an engineer is to accept a recommendation from the computer, they must be satisfied the solution is correct. "The computer did it" is not sufficient reason to follow its advice. Indeed, few human experts will accept advice from another human without understanding the reasons for that advice. This need to have answers explained is more than mistrust on the part of users; explanations help people relate the advice to their existing understanding of the domain and apply it in a more confident and flexible manner. (NOT PPS)

2)

Second, when a solution is open to inspection, we can evaluate every aspect and decision taken during the solution process, allowing for partial agreement and the addition of new information or rules to improve performance. This plays an essential role in the refinement of a knowledge base.

The exploratory nature of AI and expert system programming requires that programs be easily prototyped, tested, and changed. These abilities are supported by AI programming languages and environments and the use of good programming techniques by the designer. In a pure production system, for example, the modification of a single rule has no global syntactic side effects. Rules may be added or removed without requiring further changes to the larger program. Expert system designers have commented that easy modification of the knowledge base is a major factor in producing a successful program (McDermott 1981). (EASILY TRUE PPS)

3)

The third feature of expert systems is their use of heuristics to problem-solving methods. As expert system designers have discovered, informal "tricks of the trade" and "rules of thumb" are often more important than the standard theory presented in textbooks and classes. Sometimes these rules augment theoretical knowledge; occasionally they are simply shortcuts that seem unrelated to the theory but have been shown to work.

The heuristic nature of expert problem-solving knowledge creates problems in the evaluation of program performance. Although we know that heuristic methods will

occasionally fail, it is not clear exactly how often a program must be correct in order to be accepted: 98% of the time? 90%? 80%? Perhaps the best way to evaluate a program is to compare its results to those obtained by human experts in the same area. This suggests a variation of the Turing test (Chapter 1) for evaluating the performance of expert systems: a program has achieved expert level performance if people working in the area could not differentiate, in a blind evaluation, between the best human efforts and those of the program. In evaluating the MYCIN program for diagnosing meningitis infections, Stanford researchers had a number of infectious-disease experts blindly evaluate the performance of both MYCIN and human specialists in infectious diseases. Similarly, Digital Equipment Corporation decided that XCON, a program for configuring VAX computers, was ready for commercial use when its performance was comparable to that of human engineers.

Expert systems have been built to solve a range of problems in domains such as medicine, mathematics, engineering, chemistry, geology, computer science, business, law, defense, and education. These programs have addressed a wide range of problem types; the following list, adapted from Waterman (1986), is a useful summary of general expert system problem categories. Another excellent survey of expert system applications is Smart and Langeland-Knudsen (1986).

1. *Interpretation*—forming high-level conclusions or descriptions from collections of raw data.
2. *Prediction*—projecting probable consequences of given situations.
3. *Diagnosis*—determining the cause of malfunctions in complex situations based on observable symptoms.
4. *Design*—determining a configuration of system components that meets certain performance goals while satisfying a set of constraints.
5. *Planning*—devising a sequence of actions that will achieve a set of goals given certain starting conditions.
6. *Monitoring*—comparing the observed behavior of a system to its expected behavior.
7. *Debugging and Repair*—prescribing and implementing remedies for malfunctions.
8. *Instruction*—detecting and correcting deficiencies in students' understanding of a subject domain.
9. *Control*—governing the behavior of a complex environment.

8.1 Overview of Expert System Technology

8.1.1 Design of Rule-Based Expert Systems

Figure 8.1 shows the most important modules that make up a rule-based expert system. The user interacts with the expert system through a user interface that makes access more comfortable for the human and hides much of the system complexity (e.g., the

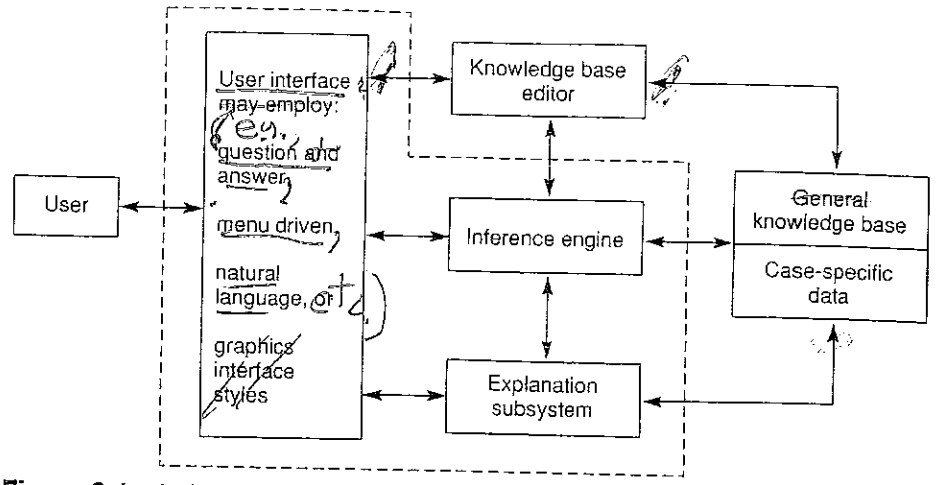


Figure 8.1 Architecture of a typical expert system.

internal structures of the rule base). Expert systems employ a variety of interface styles, including question and answer, menu-driven, natural language, or graphics interfaces.

The program must keep track of *case-specific data*, the facts, conclusions, and other relevant information of the case under consideration. This includes the data given in a problem instance, partial conclusions, confidence measures of conclusions, and dead ends in the search process. This information is separate from the general knowledge base.

The *explanation subsystem* allows the program to explain its reasoning to the user. These explanations include justifications for the system's conclusions (*how queries*, as discussed in Section 8.2.2), explanations of why the system needs a particular piece of data (*why queries*), and, in some experimental systems, tutorial explanations or deeper theoretical justifications of the program's actions.

Many systems also include a *knowledge base editor*. Knowledge base editors can access the explanation subsystem and help the programmer locate bugs in the program's performance. They also may assist in the addition of new knowledge, help maintain correct rule syntax, and perform consistency checks on the updated knowledge base. An example of the Teiresias knowledge base editor is presented in Section 8.4.5.

The heart of the expert system is the general knowledge base, which contains the problem-solving knowledge of the particular application. In a rule-based expert system this knowledge is represented in the form of *if . . . then* rules, as in the auto battery/cable example of Section 8.2. (Other ways of representing this knowledge are discussed in chapters 9 and 14.)

The *inference engine* applies the knowledge to the solution of actual problems. It is the interpreter for the knowledge base. In the production system, the inference engine performs the recognize-act control cycle. The procedures that implement the control cycle are separate from the production rules themselves. It is important to maintain this separation of knowledge base and inference engine for several reasons:

1. The separation of the problem-solving knowledge and the inference engine makes it possible to represent knowledge in a more natural fashion. "If... then" rules, for example, are closer to the way in which human beings describe their own problem-solving techniques than a program that embeds this knowledge in lower-level computer code.
2. Because the knowledge base is separated from the program's lower-level control structures, expert system builders can focus directly on capturing and organizing problem-solving knowledge rather than on the details of its computer implementation.
3. The separation of knowledge and control, along with the modularity of rules and other representational structures used in building knowledge bases, allows changes to be made in one part of the knowledge base without creating side effects in other parts of the program code.
4. The separation of the knowledge and control elements of the program allows the same control and interface software to be used in a variety of systems. The *expert system shell* has all the components of Figure 8.1 except that the knowledge base (and, of course, the case-specific data) contains no information. Programmers can use the "empty shell" and create a new knowledge base appropriate to their application. The broken lines of Figure 8.1 indicate the shell modules.
5. As illustrated in the discussion of production systems (Chapter 4), this modularity allows us to experiment with alternative control regimes for the same rule base.

The use of an expert system shell can reduce the design and implementation time of a program considerably. As may be seen in Figure 8.4, MYCIN was developed in about 20 person-years. EMYCIN (Empty MYCIN) is a general expert system shell that was produced by removing the specific domain knowledge from the MYCIN program. Using EMYCIN, knowledge engineers implemented PUFF, a program to analyze pulmonary problems in patients, in about 5 person-years. This is a remarkable saving and an important aspect of the commercial viability of expert system technology. Expert system shells have become increasingly common, with commercially produced shells available for all classes of computers.

It is important that the programmer choose the proper expert system shell. Different problems often require different reasoning processes: goal-driven rather than data-driven search, for instance. The control strategy provided by the shell must be appropriate to the new application. The medical reasoning in the PUFF application was much like that of the original MYCIN work; this made the use of the EMYCIN shell appropriate. If the shell does not support the appropriate reasoning processes, its use can be a mistake and worse than starting from nothing. As we shall see, part of the responsibility of the expert system builder is to correctly characterize the reasoning processes required for a given problem domain and to either select or construct an inference engine that implements these structures.

Unfortunately, shell programs do not solve all of the problems involved in building expert systems. While the separation of knowledge and control, the modularity of the

production system architecture, and the use of an appropriate knowledge representation language all help with the building of an expert system, the acquisition and organization of the knowledge base remain difficult tasks.

8.1.2 Selecting a Problem for Expert System Development

Expert systems tend to involve a considerable investment in money and human effort. Attempts to solve a problem that is too complex, poorly understood, or otherwise unsuited to the available technology can lead to costly and embarrassing failures. Researchers have developed an informal set of guidelines for determining whether a problem is appropriate for expert system solution:

1. *The need for the solution justifies the cost and effort of building an expert system.* For example, Digital Equipment Corporation had experienced considerable financial expense because of errors in configurations of VAX and PDP-11 computers. If a computer is shipped with missing or incompatible components, the company is obliged to correct this situation as quickly as possible, often incurring added shipping expense or absorbing the cost of parts not taken into account when the original price was quoted. Because this expense was considerable, DEC was extremely interested in automating the configuration task; the resulting system, XCON, has paid for itself in both financial savings and customer goodwill. Similarly, many expert systems have been built in domains such as mineral exploration, business, defense, and medicine where there is a large potential for savings in either money, time, or human life. In recent years, the cost of building expert systems has gone down as software tools and expertise in AI have become more available. The range of potentially profitable applications has grown correspondingly.

2. *Human expertise is not available in all situations where it is needed.* Much expert system work has been done in medicine, for example, because the specialization and technical sophistication of modern medicine have made it difficult for doctors to keep up with advances in diagnostics and treatment methods. Specialists with this knowledge are rare and expensive, and expert systems are seen as a way of making their expertise available to a wider range of doctors. In geology, there is a need for expertise at remote mining and drilling sites. Often, geologists and engineers find themselves traveling large distances to visit sites, with resulting expense and wasted time. By placing expert systems at remote sites, many problems may be solved without needing a visit by a human expert. Similarly, loss of valuable expertise through employee turnover or pending retirement may justify building an expert system.

3. *The problem may be solved using symbolic reasoning techniques.* This means that the problem should not require physical dexterity or perceptual skill. Although robots and vision systems are available, they currently lack the sophistication and flexibility of human beings. Expert systems are generally restricted to problems that humans can solve through symbolic reasoning.

4. *The problem domain is well structured and does not require commonsense reasoning.* Although expert systems have been built in a number of areas requiring specialized technical knowledge, more mundane commonsense reasoning is well beyond our current capabilities. Highly technical fields have the advantage of being well studied

and formalized; terms are well defined and domains have clear and specific conceptual models. Most significantly, however, the amount of knowledge required to solve such problems is small in comparison to the amount of knowledge used by human beings in commonsense reasoning.

5. *The problem may not be solved using traditional computing methods.* Expert system technology should not be used to "reinvent the wheel." If a problem can be solved satisfactorily using more traditional techniques such as numerical, statistical, or operations research techniques, then it is not a candidate for an expert system. Because expert systems rely on heuristic approaches, it is unlikely that an expert system will outperform an algorithmic solution if such a solution exists.

6. *Cooperative and articulate experts exist.* The knowledge used by expert systems is often not found in textbooks but comes from the experience and judgment of humans working in the domain. It is important that these experts be both willing and able to share that knowledge. This implies that the experts should be articulate and believe that the project is both practical and beneficial. If, on the other hand, the experts feel threatened by the system, fearing that they may be replaced by it or that the project can't succeed and is therefore a waste of time, it is unlikely that they will give it the necessary time and effort. It is also important that management be supportive of the project and allow the domain experts to take time away from their usual responsibilities to work with the program implementers.

7. *The problem is of proper size and scope.* It is important that the problem not exceed the capabilities of current technology. For example, a program that attempted to capture all of the expertise of a medical doctor would not be feasible; a program that advised MDs on the use of a particular piece of diagnostic equipment would be more appropriate. As a rule of thumb, problems that require days or weeks for human experts to solve are probably too complex for current expert system technology.

Although a large problem may not be amenable to expert system solution, it may be possible to break it into smaller, independent subproblems that are. Or we may be able to start with a simple program that solves a portion of the problem and gradually increase its functionality to handle more of the problem domain. This was done successfully in the creation of XCON: initially the program was designed only to configure VAX 780 computers; later it was expanded to include the full VAX and PDP-11 series (Bachant and McDermott 1984).

8.1.3 Overview of "Knowledge Engineering"

The primary people involved in building an expert system are the *knowledge engineer*, the *domain expert*, and the *end user*.

The knowledge engineer is the AI language and representation expert. His or her main task is to select the software and hardware tools for the project, help extract the necessary knowledge from the domain expert, and implement that knowledge in a correct and efficient knowledge base. The knowledge engineer may initially be ignorant of the application domain.

The domain expert provides the knowledge of the problem area. The domain expert is generally someone who has worked in the domain area and understands its problem-

solving techniques, including the use of shortcuts, handling imprecise data, evaluating partial solutions, and all the other skills that mark a person as an expert. The domain expert is primarily responsible for spelling out these skills to the knowledge engineer.

As in most applications, the end user determines the major design constraints. Unless the user is happy, the development effort is by and large wasted. The skills and needs of the user must be considered throughout the design cycle: What level of explanation does the user need? Can the user provide correct information to the system? Is the user interface appropriate? Are there environmental restrictions on the program's use?

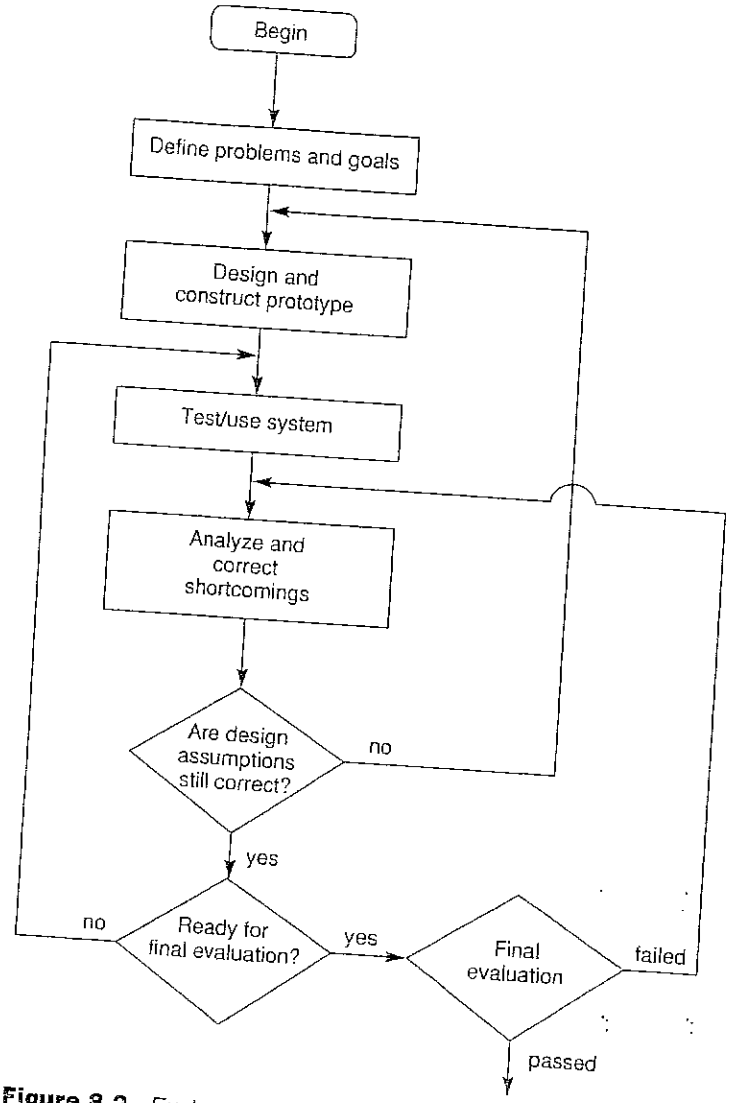


Figure 8.2 Exploratory development cycle for rule-based expert systems.

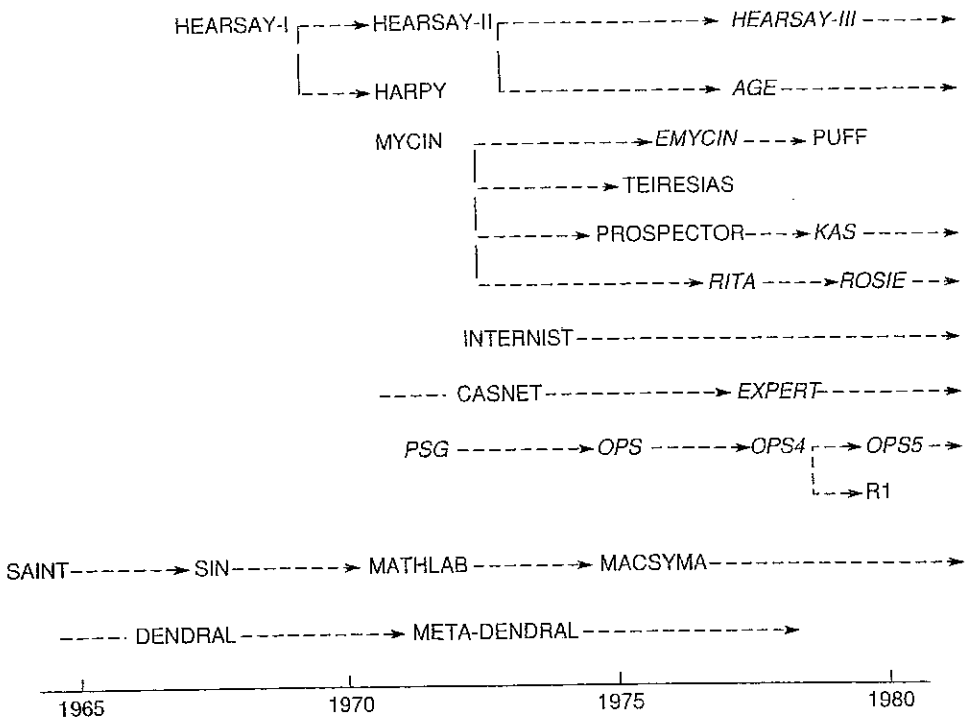


Figure 8.3 Development history for some classic expert systems (Winston and Prendergast 1984). Expert system shells in italics.

(An interface that required typing, for example, would not be appropriate for use in the cockpit of a jet fighter.) Will the program make the user's work easier, quicker, more comfortable?

Like most AI programming, expert system development requires a nontraditional development cycle based on early prototyping and incremental revision of the code. This exploratory programming methodology (described in the introduction to Part III) is complicated by the interaction between the knowledge engineer and the domain expert.

Generally, work on the system begins with the knowledge engineer attempting to gain some familiarity with the problem domain. This helps in communicating with the domain expert. This is done in initial interviews with the expert or by reading introductory texts in the domain area. Next, the knowledge engineer and expert begin the process of extracting the expert's problem-solving knowledge. This is often done by giving the domain expert a series of sample problems and having him or her explain the techniques used in their solution.

Here, it is often useful for the knowledge engineer to be a novice in the problem domain. Human experts are notoriously unreliable in explaining exactly what goes on in solving a complex problem. Often they forget to mention steps that have become obvious or even automatic to them after years of work in their field. Knowledge engineers, by virtue of their relative naïveté in the domain, should be able to spot these conceptual jumps and ask for clarification.

Once the knowledge engineer has obtained a general overview of the problem domain and gone through several problem-solving sessions with the domain expert, he or she is ready to begin actual design of the system: selecting a way to represent the knowledge (such as rules or frames), determining the basic search strategy (forward, backward, etc.), and designing the user interface. After making these design commitments, the knowledge engineer builds a prototype.

This prototype should be able to solve problems in a small area of the domain and provides a test bed for preliminary design assumptions. Once the prototype has been implemented, the knowledge engineer and domain expert test and refine its knowledge by giving it problems to solve and correcting its shortcomings. Should the assumptions made in designing the prototype prove correct, it can be built into a complete system.

Expert systems are built by progressive approximations, with the program's mistakes leading to corrections or additions to the knowledge base. In a sense, the knowledge base is "grown" rather than constructed. This approach to programming was proposed by Seymour Papert with his LOGO language (Papert 1980). The LOGO philosophy argues that watching the computer respond to the improperly formulated ideas represented by the code leads to their correction (being debugged) and clarification with more precise code. Other researchers have verified this notion that design proceeds through a process of trying and correcting candidate designs, rather than by such neatly hierarchical processes as top-down design.

It is also understood that the prototype may be thrown away if it becomes too cumbersome or if the designers decide to change their basic approach to the problem. The prototype lets program builders explore the problem and its important relationships by actually constructing a program to solve it. After this progressive clarification is complete, they can often write a cleaner version, usually with fewer actual rules.

The second major feature of expert system programming is that the program need never be considered "finished." A large heuristic knowledge base will always have limitations. The modularity and ease of modification available in the production system model make it natural to add new rules or make up for the shortcomings of the present rule base at any time. DEC, for example, has continued to add new rules to the XCON program to extend its capabilities to the rest of their product line. In 1981, XCON had about 500 rules and could configure the VAX 780; it was progressively refined until, in 1984 with about 4000 rules, it configured most of the DEC product line.

Furthermore, as DEC changed the specifications for its computers, previously correct rules needed updating. One report noted that up to 50% of the XCON rules were changed each year just to keep up with changes in the product line (Soloway et al. 1987). Figure 8.2 presents a schematic overview of the development cycle of expert system software.

Even though expert systems of commercial quality have been available since 1980, research in the development of expert systems has gone on since the middle 1960s. In fact, the emergence of commercially viable expert systems in the present world market has been a direct result of this earlier research. Figure 8.3 shows the history of the programs that are now considered "classic" in this area.

The major development activity for these early programs took place at three universities: Stanford, MIT, and Carnegie Mellon. The main research and development thrust for the rule-based expert system was at Stanford, under the direction of Edward Feigen-

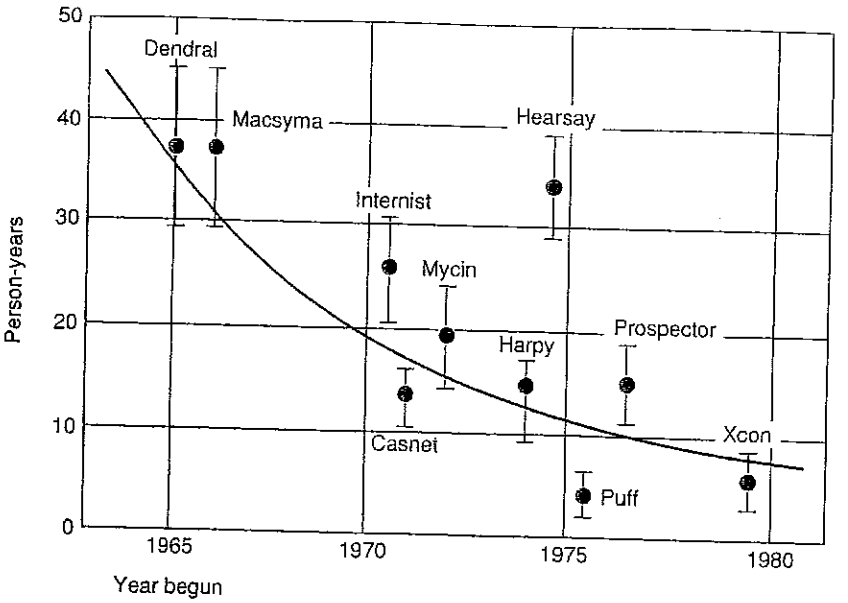


Figure 8.4 Development time for selected expert systems between 1965 and 1985 (Hayes-Roth et al. 1984).

baum, although other work in the area certainly took place at, for instance, Rutgers and the University of Pittsburgh. The early development of the production system model for problem solving, including the development of the OPS languages, took place primarily at Carnegie Mellon University.

Figure 8.4 presents another important aspect of the evolution of expert system programs: the average development time for expert systems has been drastically reduced across the decades of evolution. The emergence of expert system shells was instrumental in reducing the design time. This is perhaps the most important reason for current commercial successes.

8.2 A Framework for Organizing and Applying Human Knowledge

8.2.1 Production Systems, Rules, and the Expert System Architecture

The architecture of rule-based expert systems may be understood in terms of the production system model for problem solving presented in Part II. In fact, the parallel between the two entities is more than a simple analogy: the production system was the intellectual precursor of modern expert system architecture. This is not surprising; when Newell and

Simon began developing the production system model, their goal was to find a way to model human problem solving.

If we regard the expert system architecture in Figure 8.1 as a production system, the knowledge base is the set of production rules. The expertise of the problem area is represented by the productions. In a rule-based system, these condition-action pairs are represented as rules, with the premises of the rules (the if portion) corresponding to the condition and the conclusion (the then portion) corresponding to the action. Case-specific data are kept in the working memory. Finally, the inference engine is the recognize-act cycle of the production system. This control may be either data driven or goal driven.

In a goal-driven expert system, the goal expression is initially placed in working memory. The system matches rule conclusions with the goal, selecting one rule and placing its *premises* in the working memory. This corresponds to a decomposition of the problem into simpler subgoals. The process continues, with these premises becoming the new goals to match against rule conclusions. The system thus works back from the original goal until all the subgoals in working memory are known to be true, indicating that the hypothesis has been verified. Thus, backward search in an expert system corresponds roughly to the process of hypothesis testing in human problem solving.

In an expert system, subgoals are often solved by asking the user for information. Some expert system shells allow the system designer to specify which subgoals may be solved by asking the user. Other inference engines simply ask about all subgoals that fail to match with the conclusion of some rule in the knowledge base; i.e., if the program cannot infer the truth of a subgoal, it asks the user.

Many problem domains seem to lend themselves more naturally to forward search. In an interpretation problem, for example, most of the data for the problem are initially given and it is often difficult to formulate a hypotheses (goal). This suggests a forward reasoning process in which the facts are placed in working memory and the system searches for an interpretation in a forward fashion.

As a more detailed example, let us create a small expert system for diagnosing automotive problems:

Rule 1:
if
the engine is getting gas, and
the engine will turn over,
then
the problem is spark plugs.

Rule 2:
if
the engine does not turn over, and
the lights do not come on
then
the problem is battery or cables.

Rule 3:
if
the engine does not turn over, and
the lights do come on
then
the problem is the starter motor.

Rule 4:
if
there is gas in fuel tank, and
there is gas in carburetor
then
the engine is getting gas.

To run this knowledge base under a goal-directed control regime, place the top-level goal, the problem is X, in working memory as in Figure 8.5. X is a variable that can match with any phrase; it will become bound to the solution when the problem is solved.

There are three rules that match with the expression in working memory, rule 1, rule 2, and rule 3. If we resolve conflicts in favor of the lowest-numbered rule, then rule 1 will fire. This causes X to be bound to the value spark plugs and the premises of rule 1 to be placed in the working memory as in Figure 8.6. The system has thus chosen to explore the possible hypothesis that the spark plugs are bad. Another way to look at this is that the system has selected an or branch in an and/or graph (Chapter 3).

Note that there are two premises to rule 1, both of which must be satisfied to prove the conclusion true. These are and branches of the search graph representing a decomposition of the problem (finding if the spark plugs are bad) into two subproblems (finding

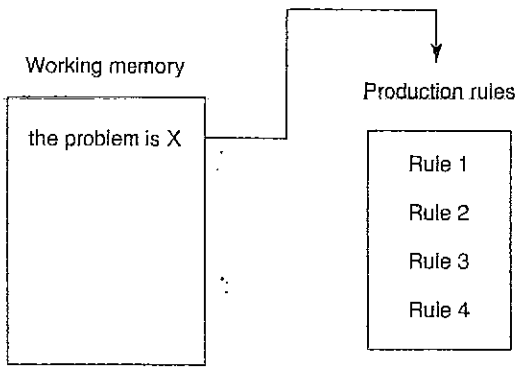


Figure 8.5 Working memory at the start of a consultation in the car diagnostic example.

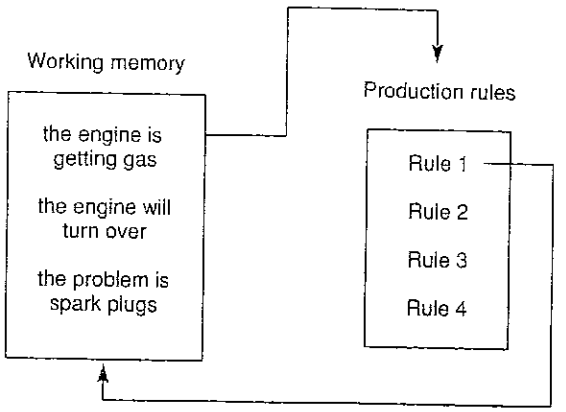


Figure 8.6 Working memory after Rule 1 has fired.

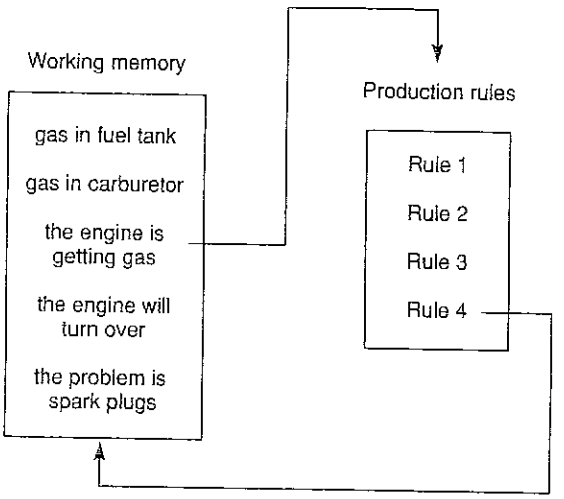


Figure 8.7 Working memory after Rule 4 has fired.

if the engine is getting gas and if the engine is turning over). We may then fire rule 4, whose conclusion matches with "engine is getting gas," causing its premises to be placed in working memory as in Figure 8.7.

At this point, there are three entries in working memory that do not match with any rule conclusions. Our expert system will query the user directly about these subgoals. If the user confirms all three of these as true, the expert system will have successfully determined that the car will not start because the spark plugs are bad. In finding this solution, the system has searched the and/or graph presented in Figure 8.8.

This is, of course, a very simple example. Not only is its automotive knowledge limited at best, but also a number of important aspects of real implementations are

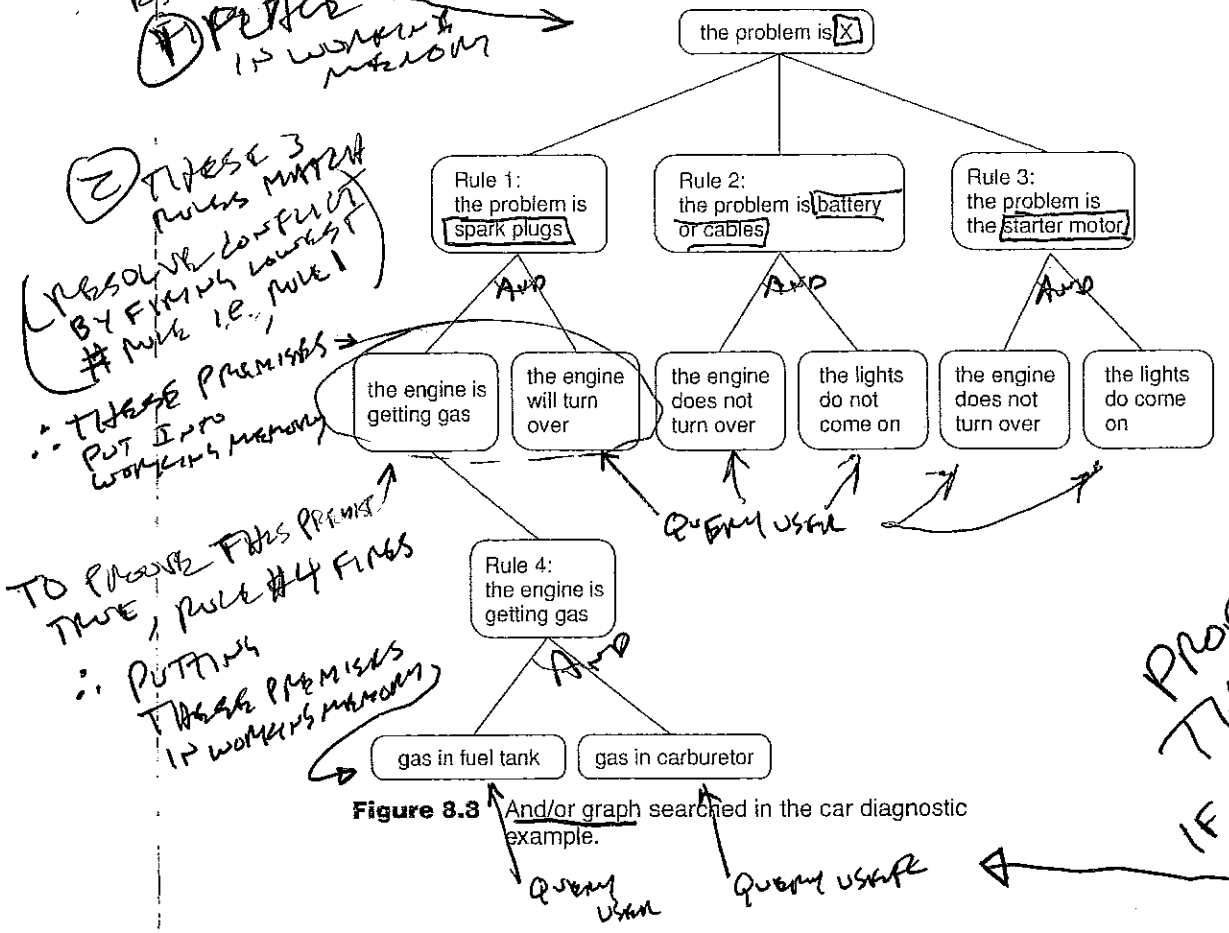


Figure 8.3 And/or graph searched in the car diagnostic example.

ignored. The rules are phrased in English, rather than a formal language. On finding a solution, a real expert system will tell the user its diagnosis (our model simply stops). Also, we should maintain enough of a trace of the reasoning to allow backtracking if necessary; in our example, had we failed to determine that the spark plugs were bad, we would have needed to back up to the top level and try rule 2 instead. Notice that this information is implicit in the ordering of subgoals in working memory of Figure 8.7 and in the graph of Figure 8.8. This example of expert systems underscores their foundation in the production system and and/or graph search models of earlier chapters.

At the beginning of this chapter, we emphasized that an expert system needed to be open to inspection, easily modified, and heuristic in nature. The production system architecture is an important factor in each of these requirements. Ease of modification, for example, is made possible by the syntactic independence of production rules: each rule is a chunk of knowledge that can be independently modified. The heuristic nature of these rules is in the knowledge obtained from the domain expert and, as seen in Section 8.3, from use of confidence measures. The next section discusses the way in which this architecture supports explanations and inspection of the reasoning process.

8.2.2 Explanation and Transparency

The production system implements a graph search: the explanation routines of Figure 8.1 keep track of the graph search and use a trace of the search to answer user queries. The production rules themselves document each step of the reasoning process.

In general, the two questions supported by rule-based expert systems are why? and how? In a why query, the user has been asked for information by the program and responds with why, meaning "why did you ask that?" The answer is the current rule that the production system is attempting to fire. The answer to "how did you get that result?" is the sequence of rules that were used to conclude a goal.

Suppose our automotive expert system has an explanation facility and assume that the consultation has reached the stage represented by the working memory configuration in Figure 8.7. The following dialogue begins with the computer asking the user about the goals present in working memory. User responses are in this type.

gas in fuel tank?
yes

gas in carburetor?
yes

engine will turn over?
why

Here the user enters a why query, i.e., why do you want to know that?

It has been established that:

1. engine is getting gas,

therefore if

2. engine will turn over,

then the problem is spark plugs.

Since the user answered yes to the first two questions, rule 4 was able to fire, verifying the first premise of rule 1. When the why query was made, the system was trying to solve the second premise of rule 1. Note that the explanation is just a restatement of the current rule, but it provides an adequate explanation for why the program wants to know if the engine will turn over. Continuing with the dialogue:

how engine is getting gas

Here the user asks how the program knew that the engine is getting gas. The response is a trace of the reasoning that led to this conclusion, working back from the goal along the rules that support it to the user responses:

This follows from Rule 4:

if

gas in fuel tank, and

gas in carburetor

then

engine is getting gas.

gas in fuel tank was given by the user

gas in carburetor was given by the user

This simple example illustrates the explanations that expert systems provide the user. Why queries are answered by showing the current rule that the system is attempting to fire. How queries are answered by giving a trace of the reasoning that led to a goal. Although the mechanisms are conceptually simple, they can exhibit remarkable explanatory power if the knowledge base is organized in a logical fashion.

The production system architecture provides an essential basis for these explanations. Each cycle of the control loop selects and fires another rule. The program may be stopped after each cycle and inspected. Since each rule represents a complete chunk of problem-solving knowledge, the current rule provides a context for the explanation. Contrast this with more traditional program architectures: if we stop a Pascal or FORTRAN program in midexecution, it is doubtful that the current statement will have much meaning.

If explanations are to behave logically, it is important not only that the knowledge base get the correct answer but also that each rule correspond to a single logical step in the problem-solving process. If a knowledge base combines several steps into a single rule or if it breaks up the rules in an arbitrary fashion, it may get correct answers but seem arbitrary and illogical in responding to how and why queries. This not only undermines the user's faith in the system but also makes the program much more difficult for its builders to understand and modify.

8.2.3 Heuristics and Control in Expert Systems

Because of the separation of the knowledge base and the inference engine and the fixed control regimes provided by the inference engine, the only way that a programmer can control the search is through the structure of the rules in the knowledge base. This is an advantage, since the control strategies required for expert level problem solving tend to be domain specific and knowledge intensive. Although a rule of the form if p, q, and r then s resembles a logical expression, it may also be interpreted as a series of procedures for solving a problem: to do s, first do p, then do q, then do r. This dual semantics of rules was already discussed in the introductory PROLOG chapter.

This procedural interpretation allows us to control search through the structure of the rules. For example, we may order the premises of a rule so that the premise that is most likely to fail or is easiest to confirm will be tried first. This gives the opportunity of eliminating a rule (and hence a portion of the search space) as early in the search as possible. Rule 1 in the automotive example tries to determine if the engine is getting gas before it asks if the engine turns over. This is inefficient, in that trying to determine if the engine is getting gas invokes another rule and eventually asks the user two questions. By reversing the order of the premises, a negative response to the query "engine will turn over?" eliminates this rule from consideration before the more involved condition is examined.

Also, from another point of view, it makes more sense to determine whether the engine is turning over before checking to see if it is getting gas; if the engine won't turn