FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

CHAPTER

10

STRONG SLOT-AND-FILLER STRUCTURES

In the 1960s and 1970s, students frequently asked, "Which kind of representation is best?" and I usually replied that we'd need more research. . . But now I would reply: To solve really hard problems, we'll have to use several different representations. This is because each particular kind of data structure has its own virtues and deficiencies, and none by itself would seem adequate for all the different functions involved with what we call common sense.

—Minsky, Marvin
(1927-), American cognitive scientist

The slot-and-filler structures described in the previous chapter are very general, Individual semantic networks and frame systems may have specialized links and inference procedures, but there are no hard and fast rules about what kinds of objects and links are good in general for knowledge representation. Such decisions are left up to the builder of the semantic network or frame system.

The three structures discussed in this chapter, *conceptual dependency, scripts,* and *CYC,* on the other hand, embody specific notions of what types of objects and relations are permitted. They stand for powerful theories of how AI programs can represent and use knowledge about common situations.

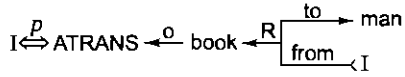
10.1 CONCEPTUAL DEPENDENCY

Conceptual dependency (often nicknamed CD) is a theory of how to represent the kind of knowledge about events that is usually contained in natural language sentences. The goal is to represent the knowledge in a way that

- Facilitates drawing inferences from the sentences.
- Is independent of the language in which the sentences were originally stated.

Because of the two concerns just mentioned, the CD representation of a sentence is built not out of primitives corresponding to the words used in the sentence, but rather out of conceptual primitives that can be combined to form the meanings of words in any particular language. The theory was first described in Schank [1973] and was further developed in Schank [1975]. It has since been implemented in a variety of programs that read and understand natural language text. Unlike semantic nets, which provide only a structure into which nodes

representing information at any level can be placed, conceptual dependency provides both a structure and a specific set of primitives, at a particular level of granularity, out of which representations of particular pieces of information can be constructed.



where the symbols have the following meanings:

- Arrows indicate direction of dependency.
- Double arrow indicates two way link between actor and action.
- p indicates past tense.
- ATRANS is one of the primitive acts used by the theory. It indicates transfer of possession.
- o indicates the object case relation.
- R indicates the recipient case relation.

Fig. 10.1 *A Simple Conceptual Dependency Representation*

As a simple example of the way knowledge is represented in CD, the event represented by the sentence

I gave the man a book.

would be represented as shown in Fig. 10.1.

In CD, representations of actions are built from a set of primitive acts. Although there are slight differences in the exact set of primitive actions provided in the various sources on CD, a typical set is the following, taken from Schank and Abelson [1977]:

ATRANS	Transfer of an abstract relationship (e.g., give)
PTRANS	Transfer of the physical location of an object (e.g., go)
PROPEL	Application of physical force to an object (e.g., push)
MOVE	Movement of a body part by its owner (e.g., kick)
GRASP	Grasping of an object by an actor (e.g., clutch)
INGEST	Ingestion of an object by an animal (e.g.. eat)
EXPEL	Expulsion of something from the body of an animal (e.g., cry)
MTRANS	Transfer of mental information (e.g., tell)
MBUILD	Building new information out of old (e.g., decide)
SPEAK	Production of sounds (e.g., say)
ATTEND	Focusing of a sense organ toward a. stimulus (e.g., listen)

A second set of CD building blocks is the set of allowable dependencies among the conceptualizations described in a sentence. There are four primitive conceptual categories from which dependency structures can be built. These are

ACTs	Actions
PPs	Objects (picture producers)
AAs	Modifiers of actions (action aiders)
PAs	Modifiers of PPs (picture aiders)

In addition, dependency structures are themselves conceptualizations and can serve as components of larger dependency structures.

The dependencies among conceptualizations correspond to semantic relations among the underlying concepts. Figure 10.2 lists the most important ones allowed by CD.[1] The first column contains the rules; the second contains examples of their-useLand the third contains an English version of each example. The rules shown in the Fig. can be interpreted as follows:

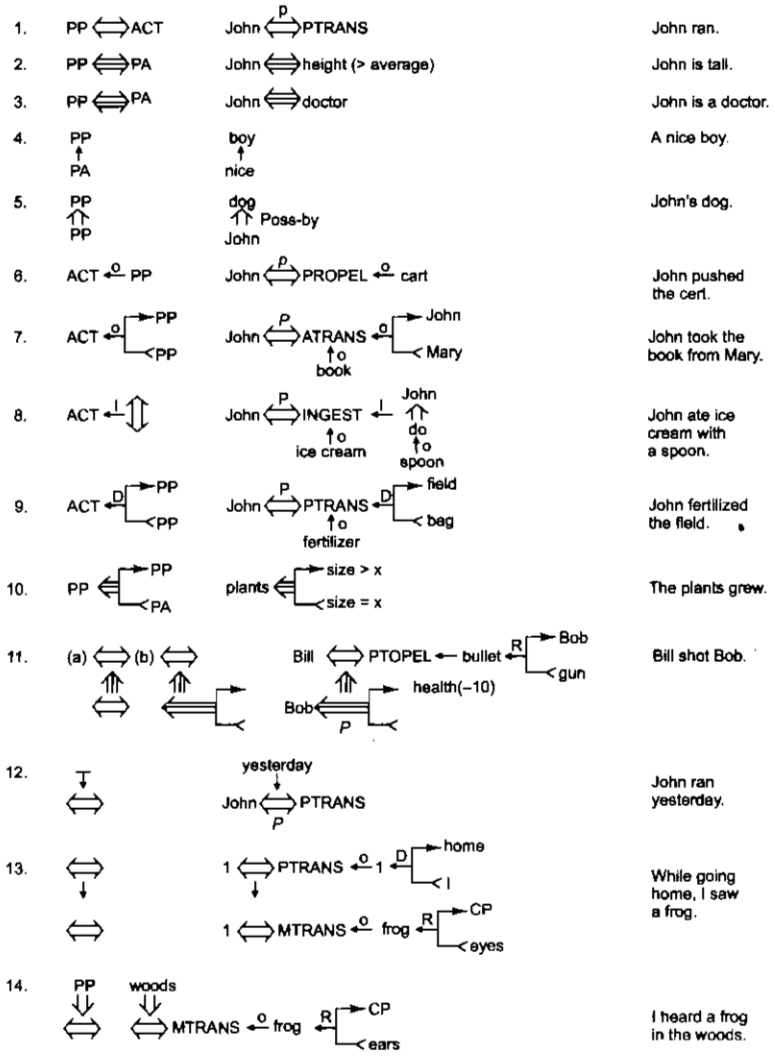


Fig. 10.2 *The Dependencies of CD*

- Rule 1 describes the relationship between an actor and the event he or she causes. This is a two-way dependency since neither actor nor event can be considered primary. The letter p above the dependency link indicates past tense.
- Rule 2 describes the relationship between a PP and a PA that is being asserted to describe it. Many state descriptions, such as height, are represented in CD as numeric scales.

[1]The lable shown in the figure is adapted from several tables in Schank [1973].

- Rule 3 describes the relationship between two PPs, one of which belongs to the set defined by the other.
- Rule 4 describes the relationship between a PP and an attribute that has already been predicated of it. The direction of the arrow is toward the PP being described.
- Rule 5 describes the relationship between two PPs, one of which provides a particular kind of information about the other. The three most common types of information to be provided in this way are possession (shown as POSS-BY), location (shown as LOC), and physical containment (shown as CONT). The direction of the arrow is again toward the concept being described.
- Rule 6 describes the relationship between an ACT and the PP that is the object of that ACT. The direction of the arrow is toward the ACT since the context of the specific ACT determines the meaning of the object relation.
- Rule 7 describes the relationship between an ACT and the source and,the recipient of the ACT.
- Rule 8 describes the relationship between an ACT and the instrument with which it is performed. The instrument must always be a full conceptualization (i.e.. it must contain an ACT), not just a single physical object.
- Rule 9 describes the relationship between an ACT and its physical source and destination.
- Rule 10 represents the relationship between a PP and a state in which it started and another in which it ended.
- Rule 11 describes the relationship between one conceptualization and another that causes it. Notice that the arrows indicate dependency of one conceptualization on another and so point in the opposite direction of the implication arrows. The two forms of the rule describe the cause of an action and the cause of a state change.
- Rule 12 describes the relationship between a conceptualization and the time at which the event it describes occurred.
- Rule 13 describes the relationship between one conceptualization and another that is the time of the first. The example for this rule also shows how CD exploits a model of the human information processing system; *see* is represented as the transfer of information between the eyes and the conscious processor.
- Rule 14 describes the relationship between a conceptualization and the place at which it occurred.

Conceptualizations representing events can be modified in a variety of ways to supply information normally indicated in language by the tense, mood, or aspect of a verb form. The use of the modifier p to indicate past tense has already been shown. The set of conceptual tenses proposed by Schank [1973] includes

p	Past
f	Future
t	Transition
t_s	Start transition
t_f	Finished transition
k	Continuing
?	Interrogative
/	Negative
nil	Present
delta	Timeless
c	Conditional

As an example of the use of these tenses, consider the CD representation shown in Fig. 10.3 (taken from Schank [1973]) of the sentence

Since smoking can kill you, I stopped.

The vertical causality link indicates that smoking kills one. Since it is marked c, however, we know only that smoking can kill one, not that it necessarily does. The horizontal causality link indicates that it is that first causality that made me stop smoking. The qualification t_{fp} attached to the dependency between I and INGEST indicates that the smoking (an instance of INGESTING) has stopped and that the stopping happened in the past.

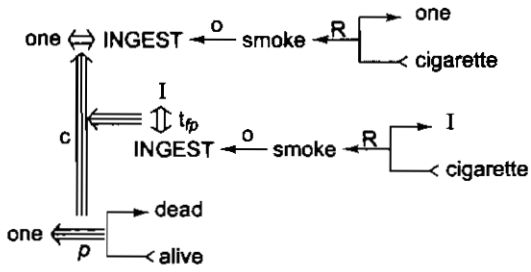


Fig. 10.3 *Using Conceptual Tenses*

There are three important ways in which representing knowledge using the conceptual dependency model facilitates reasoning with the knowledge:

1. Fewer inference rules are needed than would be required if knowledge were not broken down into primitives.
2. Many inferences are already contained in the representation itself.
3. The initial structure that is built to represent the information contained in one sentence will have holes that need to be filled. These holes can serve as an attention focuser for the program that must understand ensuing sentences.

Each of these points merits further discussion.

The first argument in favor of representing knowledge in terms of CD primitives rather than in the higher-level terms in which it is normally described is that using the primitives makes it easier to describe the inference rules by which the knowledge can be manipulated. Rules need only be represented once for each primitive ACT rather than once for every word that describes that ACT. For example, all of the following verbs involve a transfer of ownership of an object:

- Give
- Take
- Steal
- Donate

If any of them occurs, then inferences about who now has the object and who once had the object (and thus who may know something about it) may be important. In a CD representation, those possible inferences can be stated once and associated with the primitive ACT ATRANS.

A second argument in favor of the use of CD representation is that to construct it, we must use not only the information that is stated explicitly in a sentence but also a set of inference rules associated with the specific information. Having applied these: rules once, we store these results as part of the representation and they can be used repeatedly without the rules being reapplied. For example, consider the sentence

Bill threatened John With a broken nose.

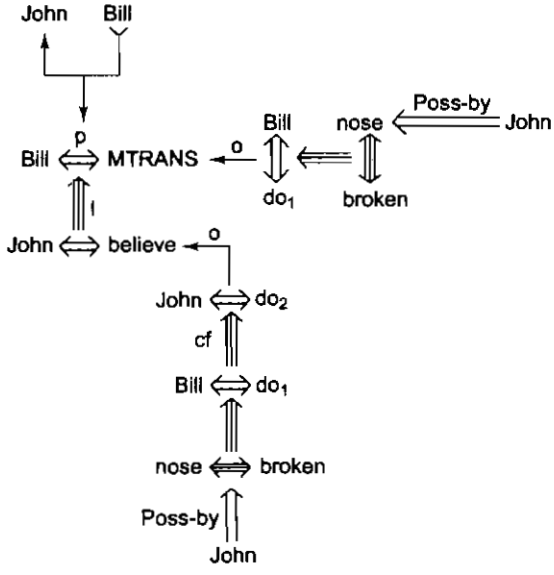


Fig. 10.4 *The CD Representation of a Threat*

The CD representation of the information contained in this sentence is shown in Fig. 10.4. (For simplicity, *believe* is shown as a single unit. In fact, it must be represented in terms of primitive ACTs and a model of the human information processing system.) It says that Bill informed John that he (Bill) will do something to

break John's nose. Bill did this so that John will believe that if he (John) does some other thing (different from what Bill will do to break his nose), then Bill will break John's nose. In this representation, the word "believe" has been used to simplify the example. But the idea behind *believe* can be represented in CD as an MTRANS of a fact into John's memory. The actions do_1 and do_2 are dummy placeholders that refer to some as yet unspecified actions.

A third argument for the use of the CD representation is that unspecified elements of the representation of one piece of, information can be used as a focus for the understanding of later events as they are encountered. So, for example, after hearing that

> Bill threatened John with a broken nose.

we might expect to find out what action Bill was trying to prevent John from performing. That action could then be substituted for the dummy action represented in Fig. 10.4 as do_2. The presence of such dummy objects provides clues as to what other events or objects are important for the understanding of the known event.

Of course, there are also arguments against the use of CD as a representation formalism. For one thing, it requires that all knowledge be decomposed into fairly low-level primitives. In Section 4.3.3 we discussed how this may be inefficient or perhaps even impossible in some situations. As Schank and Owens [1987] put it,

> CD is a theory of representing fairly simple actions. To express, for example, "John bet Sam fifty dollars that the Mets would win the World Series" takes about two pages of CD forms. This does not seem reasonable.

Thus, although there are several arguments in favor of the use of CD as a model for representing events, it is not always completely appropriate to do so, and jt may be worthwhile to seek out higher-level primitives.

Another difficulty with the theory of conceptual dependency as a general model for the representation of knowledge is that it is only a theory of the representation of events. But to represent all the information that a complex program may need, it must be able to represent other things besides events. There have been attempts to define a set of primitives, similar to those of CD for actions, that can be used to describe other kinds of knowledge. For example, physical objects, which in CD are simply represented as atomic units, have been analyzed in Lehnert [1978]. A similar analysis of social actions is provided in Schank and Carbonell [1979]. These theories continue the style of representation pioneered by CD, but they have not yet been subjected to the same amount of empirical investigation (i.e,, use in real programs) as CD.

We have discussed the theory of conceptual dependency in some detail in order to illustrate the behavior of a knowledge representation system built around a fairly small set of specific primitive elements. But CD is not the only such theory to have been developed and used in AI programs. For another example of a primitive-based system, see Wilks [1972].

10.2 SCRIPTS

CD is a mechanism for representing and reasoning about events. But rarely do events occur in isolation. In this section, we present a mechanism for representing knowledge about common sequences of events.

A *script* is a structure that describes a stereotyped sequence of events in a particular context. A script consists of a set of slots. Associated with each slot may be some information about what kinds of values it may contain as well as a default value to be used if no other information is available. So far, this definition of a script looks very similar to that of a frame given in Section 9.2, and at this level of detail, the two structures are identical. But now, because of the specialized role to be played by a script, we can make some more precise statements about its structure.

Figure 10.5 shows part of a typical script, the restaurant script (taken from Schank and Abelson [1977]). It illustrates the important components of a script:

Entry conditions	Conditions that must, in general, be satisfied before the events described in the script can occur.
Result	Conditions that will, in general, be true after the events described in the script have occurred.
Props	Slots representing objects that are involved in the events described in the script. The presence of these objects can be inferred even if they are not mentioned explicitly.
Roles	Slots representing people who are involved in the events described in the script. The presence of these people, too, can be inferred even if they are not mentioned explicitly. If specific individuals are mentioned, they can be inserted into the appropriate slots.
Track	The specific variation on a more general pattern that is represented by this particular script. Different tracks of the same script will share many but not all components.
Scenes	The actual sequences of events that occur. The events are represented in conceptual dependency formalism.

Scripts are useful because, in the real world, there are patterns to the occurrence of events. These patterns arise because of causal relationships between events. Agents will perform one action so that they will then be able to perform another. The events described in a script form a giant *causal chain*. The beginning of the chain is the set of entry conditions which enable the first events of the script to occur. The end of the chain is the set of results which may enable later events or event sequences (possibly described by other scripts) to occur. Within the chain, events are connected both to earlier events that make them possible and to later events that they enable.

If a particular script is known to be appropriate in a given situation, then it can be very useful in predicting the occurrence of events that were not explicitly mentioned. Scripts can also be useful by indicating how events that were mentioned relate to each other. For example, what is the connection between someone's ordering steak and someone's eating steak? But before a particular script can be applied, it must be activated (i.e., it must be selected as appropriate to the current situation). There are two ways in which it may be useful to activate a script, depending on how important the script is likely to be:

- For fleeting scripts (ones that are mentioned briefly and may be referred to again but are not central to the situation), it may be sufficient merely to store a pointer to the script so that it can be accessed later if necessary. This would be an appropriate strategy to take with respect to the restaurant script when confronted with a story such as

  Susan passed her favorite restaurant on her way to the museum. She really enjoyed the new Picasso exhibit.

- For nonfleeting scripts it is appropriate to activate the script fully and to attempt to fill in its slots with particular objects and people involved in the current situation.

The headers of a script (its preconditions, its preferred locations, its props, its roles, arid its events) can all serve as indicators that the script should be activated. In order to cut down on the number of times a spurious script is activated, it has proved useful to require that a situation contain at least two of a script's headers before the script will be activated.

Once a script has been activated, there are, as we have already suggested, a variety of ways in which it can be useful in interpreting a particular situation. The most important of these is the ability to predict events that have not explicitly been observed. Suppose, for example, that you are told the following story:

John went out to a restaurant last night. He ordered steak. When he paid for it, he noticed that he was running out of money. He hurried home since it had started to rain.

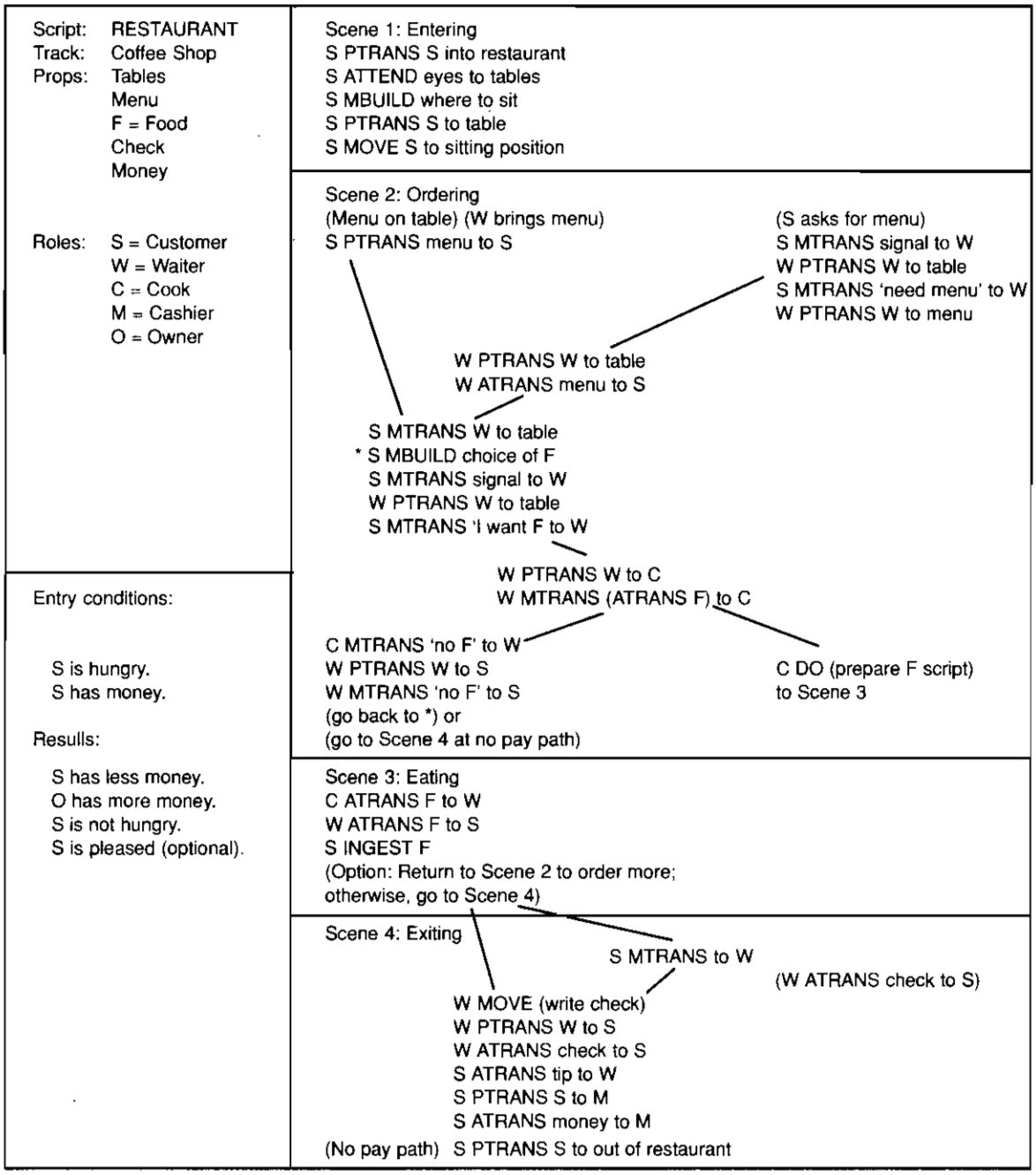


Fig. 10.5 *The Restaurant Script*

If you were then asked the question

> Did John eat dinner last night?

you would almost certainly respond that he did, even though you were not told so explicitly. By using the restaurant script, a computer question-answerer would also be able to infer that John ate dinner, since the restaurant script could have been activated. Since all of the events in the story correspond to the sequence of events predicted by the script, the program could infer that the entire sequence predicted by the script occurred normally. Thus it could conclude, in particular, that John ate. In their ability to predict unobserved events, scripts are similar to frames and to other knowledge structures that represent stereotyped situations. Once one of these structures is activated in a particular situation, many predictions can be made.

A second important use of scripts is to provide a way of building a single coherent interpretation from a collection of observations. Recall that a script can be viewed as a giant causal chain. Thus it provides information about how events are related to each other. Consider, for example, the following story:

> Susan went out to lunch. She sat down at a table and called the waitress. The waitress brought her a menu and she ordered a hamburger.

Now consider the question

> Why did the waitress bring Susan a menu?

The script provides two possible answers to that question:

- Because Susan asked her to. (This answer is gotten by going backward in the causal chain to find out what caused her to do it.)
- So that Susan could decide what she wanted to eat. (This answer is gotten by going forward in the causal chain to find out what event her action enables.)

A third way in which a script is useful is that it focuses attention on unusual events. Consider the following story:

> John went to a restaurant. He was shown to his table. He ordered a large steak. He sat there and waited for a long time. He got mad and left.

The important part of this story is the place in which it departs from the expected sequence of events in a restaurant. John did not get mad because he was shown to his table. He did get mad because he had to wait to be served. Once the typical sequence of events is interrupted, the script can no longer be used to predict other events. So, for example, in this story, we should not infer that John paid his bill. But we can infer that he saw a menu, since reading the menu would have occurred before the interruption. For a discussion of SAM, a program that uses scripts to perform this kind of reasoning, see Cullingford [1981].

From these examples, we can see how information about typical sequences of events, as represented in scripts, can be useful in interpreting a particular, observed sequence of events. The usefulness of a script in some of these examples, such as the one in which unobserved events were predicted, is similar to the usefulness of other knowledge structures, such as frames. In other examples, we have relied on specific properties of the information stored in a script, such as the causal chain represented by the events it contains. Thus although scripts are less general structures than are frames, and so are not suitable for representing all kinds of knowledge, they can be very effective for representing the specific kinds of knowledge for which they were designed.

10.3 CYC

CYC [Lenat and Guha, 1990] is a very large knowledge base project aimed at capturing human commonsense knowledge. Recall that in Section 5.1, our first attempt to prove that Marcus was not loyal to Caesar failed because we were missing the simple fact that all men are people. The goal of CYC is to encode the large body of knowledge that is so obvious that it is easy to forget to state it explicitly. Such a knowledge base could then be combined with specialized knowledge bases to produce systems that are less brittle than most of the ones available today.

Like CD, CYC represents a specific theory of how to describe the world, and like CD, it can be used for AI tasks such as natural language understanding. CYC, however, is more comprehensive; while CD provided a specific theory of representation for events, CYC contains representations of events, objects, attitudes, and so forth. In addition, CYC is particularly concerned with issues of scale, that is, what happens when we build knowledge bases that contain millions of objects.

10.3.1 Motivations

Why should we want to build large knowledge bases at all? There are many reasons, among them:

- Brittleness—Specialized knowledge-based systems are brittle. They cannot cope with novel situations, and their performance degradation is not graceful. Programs built on top of deep, commonsense knowledge about the world should rest on firmer foundations.
- Form and Content—The techniques we have seen so far for representing and using knowledge may or may not be sufficient for the purposes of AI. One good way to find out is to start coding large amounts of commonsense knowledge and see where the difficulties crop up. In other words, one strategy is to focus temporarily on the content of knowledge bases rather than on their form.
- Shared Knowledge—Small knowledge-based systems must make simplifying assumptions about how to represent things like space, time, motion, and structure. If these things can be represented once at a very high level, then domain-specific systems can gain leverage cheaply. Also, systems that share the same primitives can communicate easily with one another.

Building an immense knowledge base is a staggering task, however We should ask whether there are any methods for acquiring this knowledge automatically. Here-are two possibilities:

1. Machine Learning—In Chapter 17, we discuss some techniques for automated learning. However, current techniques permit only modest extensions of a program's knowledge. In order for a system to learn a great deal, it must already know a great deal. In particular, systems with a lot of knowledge will be able to employ powerful analogical reasoning.
2. Natural Language Understanding—Humans extend their own knowledge by reading books and talking with other humans. Since we now have on-line versions of encyclopedias and dictionaries, why not feed these texts into an AI program and have it assimilate all the information automatically? Although there are many techniques for building language understanding systems (see Chapter 15), these methods are themselves very knowledge-intensive. For example, when we hear the sentence

   John went to the bank and withdrew $50.

   we easily decide that "bank" means a financial institution, and not a river bank. To do this, we apply fairly deep knowledge about what a financial institution is, what it means to withdraw money, etc. Unfortunately, for a program to assimilate the knowledge contained in an encyclopedia, that program must already know quite a bit about the world.

The approach taken by CYC is to hand-code (what its designers consider to be) the ten million or so facts that make up commonsense knowledge. It may then be possible to bootstrap into more automatic methods.

10.3.2 CYCL

CYC's knowledge is encoded in a representation language called CYCL. CYCL is a frame-based system that incorporates most of the techniques described in Chapter 9 (multiple inheritance, slots as full-fledged objects, *transfers-through, mutually-disjoint-with,* etc). CYCL generalizes the notion of inheritance so that properties can be inherited along any link, not just *isa* and *instance*. Consider the two statements:

```
Mary
  likes:                ???
  constraints:          (LispConstraint)
LispConstraint
  slotConstrained:      (likes)
  slotValueSubsumes:
    (TheSetOf X (Person allInstances)
       (And (programsIn X LispLanguage)
          (Not (ThereExists Y (Languages all Instances)
             (And (Not (Equal Y LispLanguage))
                (programsIn X Y))))))
  propagationDirection: forward
Bob
  programsIn:           (LispLanguage)
Jane
  programsIn:           (LispLanguage CLanguage)
```

Fig. 10.6 *Frames and Constraint Expressions in CYC*

1. All birds have two legs.
2. All of Mary's friends speak Spanish.

We can easily encode the first fact using standard inheritance—any frame with *Bird* on its *instance* slot inherits the value 2 on its *legs* slot. The second fact can be encoded in a similar fashion if we allow inheritance to proceed along the *friend* relation—any frame with *Mary* on its *friend* slot inherits the value *Spanish* on its *languagesSpoken* slot. CYC further generalizes inheritance to apply to a chain of relations, allowing us to express facts like, "All the parents of Mary's friends are rich," where the value *Rich* is inherited through a composition of the *friend* and *parentOf* links.

In addition to frames, CYCL contains a *constraint language* that allows the expression of arbitrary first-order logical expressions. For example, Fig. 10.6 shows how we can express the fact "Mary likes people who program solely in Lisp." *Mary* has a constraint called *lispConstraint,* which restricts the values of her *likes* slot. The *slotValueSubsumes* attribute of *lispConstraint* ensures that Mary's *likes* slot will be filled with at least those individuals that satisfy the logical condition, namely that they program in *LispLanguage* and no others.

The time at which the default reasoning is actually performed is determined by the direction of the *slotValueSubsumes* rules. If the direction is *backward,* the rule is an if-needed rule, and it is invoked whenever someone inquires as to the value of Mary's *likes* slot. (In this case, the rule infers that Mary likes Bob but not Jane.) If the direction is *forward,* the rule is an if-added rule, and additions are automatically propagated to Mary's *likes* slot. For example, after we place LISP on Bob's *programsIn* slot, then the system quickly places Bob on Mary's *likes* slot for us. A truth maintenance system (see Chapter 7) ensures that if Bob ceases to be a Lisp programmer (or if he starts using Pascal), then he will also cease to appear on Mary's *likes* slot.

While forward rules can be very useful, they can also require substantial time and space to propagate their values. If a rule is entered as backward, then the system defers reasoning until the information is specifically requested. CYC maintains a separate background process for accomplishing forward propagations. A

knowledge engineer can continue entering knowledge while its effects are propagated during idle keyboard time.[2]

Now let us return to the constraint language itself. Recall that it allows for the expression of facts as arbitrary logical expressions. Since first-order logic is much more powerful than CYC's frame language, why does CYC maintain both? The reason is that frame-based inference is very efficient, while general logical reasoning is computationally hard. CYC actually supports about twenty types of efficient inference mechanisms (including inheritance and transfers-through), each with its own truth maintenance facility. The constraint language allows for the expression of facts that are too complex for any of these mechanisms to handle.

The constraint language also provides an elegant, abstract layer of representation. In reality, CYC maintains two levels of representation: the *epistemological level* (EL) and the *heuristic level* (HL). The EL contains facts stated in the logical constraint language, while the HL contains the same facts stored using efficient inference templates. There is a translation program for automatically converting an EL statement into an efficient HL representation. The EL provides a clean, simple functional interface to CYC so that users and computer programs can easily insert and retrieve information, from the knowledge base. The EL/HL distinction represents one way of combining the formal neatness of logic with the computational efficiency of frames.

In addition to frames, inference mechanisms, and the constraint language, CYCL performs consistency checking (e.g., detecting when an illegal value is placed on a slot) and conflict resolution (e.g., handling cases where multiple inference procedures assign incompatible values to a slot).

10.3.3 Control and Meta-Knowledge

Recall our discussion of control knowledge in Chapter 6, where we saw how to take information about control out of a production system interpreter and represent it declar-atively using rules. CYCL strives to accomplish the same thing with frames. We have already seen how to specify whether a fact is propagated in the forward or backward direction—this is a type of control information. Associated with each slot is a set of inference mechanisms that can be used to compute values for it. For any given problem, CYC's reasoning is constrained to a small range of relevant, efficient procedures. A query in CYCL can be tagged with a level of effort. At the lowest level of effort, CYC merely checks whether the fact is stored in the knowledge base. At higher levels, CYC will invoke backward reasoning and even entertain metaphorical chains of inference. As the knowledge base grows, it will become necessary to use control knowledge to restrict reasoning to the most relevant portions of the knowledge base. This control knowledge can, of course, be stored in frames.

In the tradition of its predecessor RLL (Representation Language Language) [Greiner and Lenat, 1980], many of the inference mechanisms used by CYC are stored explicitly as EL templates in the knowledge base. These templates can be modified like any other frames, and a user can create a new inference template by copying and editing an old one. CYC generates LISP code to handle the various aspects of an inference template. These aspects include recognizing when an EL statement can be transformed into an instance of the template, storing justifications of facts that are deduced (and retracting those facts when the justifications disappear), and applying the inference mechanism efficiently. As with production systems, we can build a more flexible, reflective system by moving inference procedures into a declarative representation.

It should be clear that many of the same control issues exist for frames and rules. Unlike numerical heuristic evaluation functions, control knowledge often has a commonsense, "knowledge about the world" flavor to it. It therefore -begins to bridge the gap between two usually disparate types of knowledge: knowledge that is typically used for search control and knowledge that is typically used for natural language disambiguation.

[2] Another idea is to have the system do forward propagation of knowledge during periods of infrequent use, such as at night.

10.3.4 Global Ontology

Ontology is the philosophical study of what exists. In the AI context, ontology is concerned with which categories we can usefully quantify over and how those categories relate to each other. All knowledge-based systems refer to entities in the world, but in order to capture the breadth of human knowledge, we need a well-designed *global ontology* that specifies at a very high level what kinds of things exist and what their general properties are. As mentioned above, such a global ontology should provide a more solid foundation for domain-specific AI programs and should also allow them to communicate with each other.

The highest level concept in CYC is called *Thing*. Everything is an instance of *Thing*. Below this top-level concept, CYC makes several distinctions, including:

- *IndividualObject* versus *Collection*—The CYCL concept *Collection* corresponds to the class CLASS described in Chapter 9. Here are some examples of frames that are instances of *Collection: Person, Nation, Nose*. Some instances of *In dividualObject* are *Fred, Greece, Fred'sNose*. These two sets share no common instances, and any instance of *Thing* must be an instance of one of the two sets. Anything that is an instance of *Collection* is a subset of *Thing*. Only *Collections* may have supersets and subsets; only *IndividualObjects* may have parts.
- *Intangible, Tangible,* and *Composite*—Instances of *Intangible* are things without mass, e.g., sets, numbers, laws, and events. Instances of *TangibleObject are* things with mass that have no intangible aspect, e.g., a person's body, an orange, and dirt. Every instance of *TangibleObject* is also an instance of *IndividualObject* since sets have no mass. Instances of *CompositeObject* have two key slots, *physicalExtent* and *intangibleExtent*. For example, a person is a *CompositeObject* whose *physicalExtent* is his body and whose *intangibleExtent* is his mind.
- *Substance*—*Substance* is a subclass of *IndividualObject*. Any subclass of *Substance* is something that retains its properties when it is cut up into smaller pieces. For example, *Wood* is a *Substance*.[3] A concept like *Table34* can be an instance of both *Wood* (a *Substance)* and *Table* (an *IndividualObject)*.
- *Intrinsic* versus *Extrinsic* properties—A property is intrinsic if when an object has that property all parts of the object also have that property. For example, *color* is an intrinsic property. Objects tend to inherit their intrinsic properties from *Substances*. Extrinsic properties include things like *number-of-legs*. Objects tend to inherit their extrinsic properties from *IndividualObjects*.
- *Event* and *Process*—An *Event* is anything with temporal extent, e.g., *Walking*. *Process* is a subclass of *Event*. If every temporal slice of an *Event* is essentially the same as the entire *Event*, then that *Event* is also a *Process*. For example, *Walking* is a *Process*, but *WalkingTwoMiles* is not. This relationship is analogous to *Substance* and *IndividualObject*.
- Slots—*Slot* is a subclass of *Intangible*. There are many types of *Slot*. *BookkeepingSlots* record such information as when a frame was created and by whom. *DefiningSlots* refer not to properties of the frame but to properties of the object represented by the frame. *DefiningSlots* are further divided into intensional, taxonomic, and extensional categories. *QuantitativeSlots* are those which take on a scalar range of values, e.g., *height*, as opposed to *gender*.
- Time—*Events* can have temporal properties, such as *duration* and *startsBefore*. CYC deals with two basic types of temporal measures: intervals, and sets of intervals. A number of basic interval properties, such as *endsDuring*, are denned from the property *before*, which applies to starting and ending times

[3] Of course, if we cut a substance up *too* finely, itceases to be the same substance. For each substance type, CYC stores its *granule* size, e.g., *Wood.granule* = *PlantCell.Crowd.granule* = *Person*, etc.

for events. Sets of intervals are built up from basic intervals through operations like union and intersection. Thus, it is possible to state facts like "John goes to the movies at three o'clock every Sunday."

- *Agent*—An important subset of *CompositeObject* is *Agent,* the collection of in telligent beings. *Agents* can be collective (e.g., corporations) owndividual (e.g., people). *Agents* have a number of properties, one of which is *beliefs.* Agents often ascribe their own beliefs to other agents in order to facilitate communication. An agent's beliefs may be incorrect, so CYC must be able to distinguish between facts in its own knowledge base (CYC's beliefs) and "facts" that are possibly inconsistent with the knowledge base.

These are but a few of the ontological decisions that the builders of a large knowledge base must make. Other problems arise in the representation of space, causality, structures, and the persistence of objects through time. We return to some of these issues in Chapter 19.

10.3.5 Tools

CYC is a multi-user system that provides each knowledge enterer with a textual and graphical interface to the knowledge base. Users' modifications to the knowledge base are transmitted to a central server, where they are checked and then propagated to other users.

We do not yet have much experience with the engineering problems of building and maintaining very large knowledge bases. In the future, it will be necessary to have tools that check consistency in the knowledge base, point out areas of incompleteness, and ensure that users do not step on each others' toes.

EXERCISES

1. Show a conceptual dependency representation of the sentence

   John begged Mary for a pencil.

   How does this representation make it possible to answer the question

   Did John talk to Mary?

2. One difficulty with representations that rely on a small set of semantic primitives, such as conceptual dependency, is that it is often difficult to represent distinctions between fine shades of meaning. Write CD representations for each of the following sentences. Try to capture the differences in meaning between the two sentences of each pair.

   John slapped Bill.
   John punched Bill.

   Bill drank his Coke.
   Bill slurped his Coke.

   Sue likes Dickens.
   Sue adores Dickens.

3. Construct a script for going to a movie from the viewpoint of the movie goer.
4. Consider the following paragraph:

   Jane was extremely hungry. She thought about going to her favorite restaurant for dinner, but it was the day before payday. So instead she decided to go home and pop a frozen pizza in the oven. On the way, though, she ran into her friend, Judy. Judy invited Jane to go out to dinner with her and Jane instantly agreed. When they got to their favorite place, they found a good table and relaxed over their meal.

How could the restaurant script be invoked by the contents of this story? Trace the process throughout the story. Might any other scripts also be invoked? For example, how would you answer the question, "Did Jane pay for her dinner?"

5. Would conceptual dependency be a good way to represent the contents of a typical issue *of National Geographic*"?
6. State where in the CYC ontology following concepts should fall:
   - cat
   - court case
   - New York Times
   - France
   - glass of water

PART III

ADVANCED TOPICS

CHAPTER

12

GAME PLAYING

Every game of skill is susceptible of being played by an automaton.

—**Charles Babbage**
(1791-1871), English mathematician, philosopher, inventor and mechanical engineer

12.1 OVERVIEW

Games hold an inexplicable fascination for many people, and the notion that computers might play games has existed at least as long as computers. Charles Babbage, the nineteenth-century computer architect, thought about programming his Analytical Engine to play chess and later of building a machine to play tic-tac-toe [Bowden, 1953]. Two of the pioneers of the science of information and computing contributed to the fledgling computer game-playing literature. Claude Shannon [1950] wrote a paper in which he described mechanisms that could be used in a program to play chess. A few years later, Alan Turing described a chess-playing program, although he never built it. (For a description, see Bowden [1953].) By the early 1960s, Arthur Samuel had succeeded in building the first significant, operational game-playing program. His program played checkers and, in addition to simply playing the game, could learn from its mistakes and improve its performance [Samuel, 1963].

There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine. Unfortunately, the second is not true for any but the simplest games. For example, consider chess.

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.

Thus it is clear that a program that simply does a straightforward search of the game tree will not be able to select even its first move during the lifetime of its opponent. Some kind of heuristic search procedure is necessary.

One way of looking at all the search procedures we have discussed is that they are essentially generate-and-test procedures in which the testing is done after varying amounts of work by the generator. At one extreme, the generator generates entire proposed solutions, which the tester then evaluates. At the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the, tester and the most promising one is chosen. Looked at this way, it is clear that to improve the effectiveness of a search-based problem-solving program two things can be done:

- Improve the generate procedure so that only good moves (or paths) are generated.
- Improve the test procedure so that the best moves (or paths) will be recognized and explored first.

In game-playing programs, it is particularly important that both these things be done. Consider again the problem of playing chess. On the average, there are about 35 legal moves available at each turn. If we use a simple legal-move generator, then the test procedure (which probably uses some combination of search and a heuristic evaluation function) will have to look at each of them. Because the test procedure must look at so many possibilities, it must be fast. So it probably cannot do a very accurate job. Suppose, on the other hand, that instead of a legal-move generator, we use a *plausible-move generator* in which only some small number of promising moves are generated. As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise. (So, for example, it is extremely important in programs that play the game of go [Benson *et al.*, 1979].) With a more selective move generator, the test procedure can afford to spend more time evaluating each of the moves it is given so it can produce a more reliable result. Thus by incorporating heuristic knowledge into both the generator and the tester, the performance of the overall system can be improved.

Of course, in game playing, as in other problem domains, search is not the only available technique. In some games, there are at least some times when more direct techniques are appropriate. For example, in chess, both openings and endgames are often highly stylized, so they are best played by table lookup into a database of stored patterns. To play an entire game then, we need to combine search-oriented and nonsearch-oriented techniques.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached. In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for interesting games such as chess, it is not usually possible, even with a good plausible-move generator, to search until a goal state is found. The depth of the resulting tree (or graph) and its branching factor are too great. In the amount of time available, it is usually possible to search a tree only ten or twenty moves (called *ply* in the game-playing literature) deep. Then, in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous. This is done using a *static evaluation function,* which uses whatever information it has to evaluate individual board positions by estimating how likely they are to lead eventually to a win. Its function is similar to that of the heuristic function h' In the A* algorithm: in the absence of complete information, choose the most promising position. Of course, the static evaluation function could simply be applied directly to the positions generated by the proposed moves. But since it is hard to produce a function like this that is very accurate, it is better to apply it as many levels down in the game tree as time permits.

A lot of work in game-playing programs has gone into the development of good static evaluation functions.[1] A very simple static evaluation function for chess based on piece advantage was proposed by Turing—simply add the values of black's pieces (B), the values of white's pieces (W), and then compute the quotient W/B. A more sophisticated approach was that taken in Samuel's checkers program, in which the static evaluation function was a linear combination of several simple functions, each of which appeared as though it might be

[1]See Berliner [1979b] for a discussion of some theoretical issues in the design of static evaluation functions.

significant. Samuel's functions included, in addition to the obvious one, piece advantage, such things as capability for advancement, control of the center, threat of a fork, and mobility. These factors were then combined by attaching to each an appropriate weight and then adding the terms together. Thus the complete evaluation function had the form:

$$c_1 \times pieceadvantage + c_2 \times advancement + c_3 \times centercontrol...$$

There were also some nonlinear terms reflecting combinations of these factors. But Samuel did not know the correct weights to assign to each of the components. So he employed a simple learning mechanism in which components that had suggested moves that turned out to lead to wins were given an increased weight, while the weights of those that had led to losses were decreased.

Unfortunately, deciding which moves have contributed to wins and which to losses is not always easy. Suppose we make a very bad move, but then, because the opponent makes a mistake, we ultimately win the game. We would not like to give credit for winning to our mistake. The problem of deciding which of a series of actions is actually responsible for a particular outcome is called the *credit assignment problem* [Minsky, 1963]. It plagues many learning mechanisms, not just those involving games. Despite this and other problems, though, Samuel's checkers program was eventually able to beat its creator. The techniques it used to acquire this performance are discussed in more detail in Chapter 17.

We have now discussed the two important knowledge-based components of a good game-playing program: a good plausible-move generator and a good static evaluation function. They must both incorporate a great deal of knowledge about the particular game being played. But unless these functions are perfect, we also need a search procedure that makes it possible to look ahead as many moves as possible to see what may occur. Of course, as in other problem-solving domains, the role of search can be altered considerably by altering the amount of knowledge that is available to it. But, so far at least, programs that play nontrivial games rely heavily on search.

What search strategy should we use then? For a simple one-person game or puzzle, the A* algorithm described in Chapter 3 can be used. It can be applied to reason forward from the current state as far as possible in the time allowed. The heuristic function h' can be applied at terminal nodes and used to propagate values back up the search graph so that the best next move can be chosen. But because of their adversarial nature, this procedure is inadequate for two-person games such as chess. As values are passed back up, different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses. There are several ways that this can be done. The most commonly used method is the *minimax* procedure, which is described in the next section. An alternative approach is the B* algorithm [Berliner, 1979a], which works on both standard problem-solving trees and on game trees.

12.2 THE MINIMAX SEARCH PROCEDURE

The *minimax search procedure is* a depth-first, depth-limited search procedure. It was described briefly in Section 1.3.1. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evaluation of it. The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to *maximize* the value of the static evaluation function of the next board position.

An example of this operation is shown in Fig. 12.1. It assumes a static evaluation function that returns values ranging from – 10 to 10, with 10 indicating a win for us, –10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

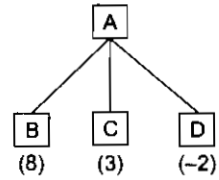


Fig. 12.1 *One-Ply Search*

But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply. This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed. So we would like to look ahead to see what will happen to each of the new game positions at-the next move which will be made by the opponent. Instead of applying the static evaluation function to each of the positions that we just generated, we apply the plausible-move generator, generating a set of successor positions for each position. If we wanted to stop here, at two-ply lookahead, we could apply the static evaluation function to each of these positions, as shown in Fig. 12.2.

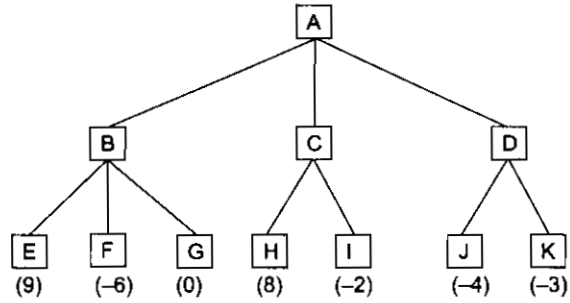


Fig. 12.2 *Two-Ply Search*

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level. Suppose we made move B. Then the opponent must choose among moves E, F, and G. The opponent's goal is to *minimize* the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 12.3 shows the result of propagating the new values up the tree. At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.

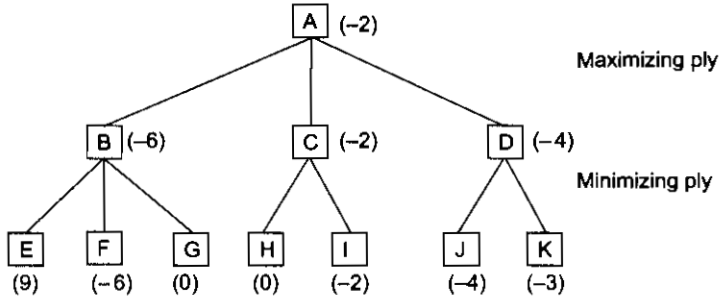


Fig. 12.3 *Backing Up the Values of a Two-Ply Search*

Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information we have available, is C, since there is nothing the opponent can do from there to produce a value worse than –2. This process can be repeated for as many ply as time allows, and

the more accurate evaluations that are produced can be used to choose the correct move at the top level. The alternation of maximizing and minimizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and gives this method the name minimax.

Having described informally the operation of the minimax procedure, we now describe it precisely. It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played:

1. MOVEGEN(*Position, Player*)—The plausible-move generator, which returns a list of nodes representing the moves that can be made by *Player* in *Position*. We call the two players PLAYER-ONE and PLAYER-TWO; in a chess program, we might use the names BLACK and WHITE instead.
2. STATIC(*Position, Player*)—The static evaluation function, which returns a num ber representing the goodness of *Position* from the standpoint of *Player*.[2]

As with any recursive program, a critical issue in the design of the MINIMAX procedure is when to stop the recursion and simply call the static evaluation function. There are a variety of factors that may influence this decision. They include:

- Has one side won?
- How many ply have we already explored?
- How promising is this path?
- How much time is left?
- How stable is the configuration?

For the general MINIMAX procedure discussed here, we appeal to a function, DEEP-ENOUGH, which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise. Our simple implementation of DEEP-ENOUGH will take two parameters, *Position* and *Depth*. It will ignore its *Position* parameter and simply return TRUE if its *Depth* parameter exceeds a constant cutoff value.

One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

We assume that MINIMAX returns a structure containing both results and that we have two functions, VALUE and PATH, that extract the separate components.

Since we define the MINIMAX procedure as a recursive function, we must also specify how it is to be called initially. It takes three parameters, a board position, the current depth of the search, and the player to move. So the initial call to compute the best move from the position CURRENT should be

```
MINIMAX(CURRENT, 0, PLAYER-ONE)
```

if PLAYER-ONE is to move, or

```
MINIMAX(CURRENT, 0, PLAYER-TWO)
```

if PLAYER-TWO is to move.

[2] This may be a bit confusing, but it need not be. In all the examples in this chapter so far (including Fig. 12.2 and 12.3), we have assumed that all values of STATIC are from the point of view of the initial (maximizing) player. It turns out to be easier when defining the algorithm, though, to let STATIC alternate perspectives so that we do not need to write separate procedures for the two levels. It is easy to modify STATIC for this purpose; we merely compute the value *of Position* from PLAYER-ONE's perspective, then invert the value if STATIC's parameter is PLAYER-TWO.

Algorithm: MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(*Position, Depth*), then return the structure

   VALUE = *STATIC*(*Position, Player*);
   PATH = nil

   This indicates that there is no path from this node and that its value is that determined by the static evaluation function.
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
   Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the hest score that can be achieved by an element of SUCCESSORS.
   For each element SUCC of SUCCESSORS, do the following:
   (a) Set RESULT-SUCC to
      MINIMAX(SUCC, *Depth* + 1, OPPOSITE(*Player*))
   This recursive call to MINIMAX will actually carry out the exploration of SUCC.
   (b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
   (c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
      (i) Set BEST-SCORE to NEW-VALUE.
      (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined hy the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
   VALUE = BEST-SCORE
   PATH = BEST-PATH

When the initial call to MINIMAX returns, the best move from CURRENT is the first element on PATH. To see how this procedure works, you should trace its execution for the game tree shown in Fig. 12.2.

The MINIMAX procedure just described is very simple. But its performance can be improved significantly with a few refinements. Some of these are described in the next few sections.

12.3 ADDING ALPHA-BETA CUTOFFS

Recall that the minimax procedure is a depth-first process. One path is explored as far as time allows, the static evaluation function is applied to the game positions at the last step of the path, and the value can then be passed up the path one level at a time. One of the good things about depth-first procedures is that their efficiency can often be improved by using branch-and-bound techniques in which partial solutions that are clearly worse than known solutions can be abandoned early. We described a straightforward application of this technique to the traveling salesman problem in Section 2.2.1. For that problem, all that was required was storage of the length of the best path found so far. If a later partial path outgrew that bound, it was abandoned.

But just as it was necessary to modify our search procedure slightly to handle both maximizing and minimizing players, it is also necessary to modify the branch-and-bound strategy to include two bounds, one for each of the players. This modified strategy is called *alpha-beta pruning*. It requires the maintenance of two threshold values, one representing a lower bound on the value that a maximizing node may ultimately be assigned (we call this *alpha*) and another representing an upper bound on the value that a minimizing node may be assigned (this we call *beta*).

To see how the alpha-beta procedure works, consider the example shown in Fig. 12.4.[3] After examining node F, we know that the opponent is guaranteed a score of –5 or less at C (since the opponent is the minimizing player). But we also know that we are guaranteed a score of 3 or greater at node A, which we can achieve if we move to B. Any other move that produces a score of less than 3 is worse than the move to B, and we can ignore it. After examining only F, we are sure that a move to C is worse (it will be less than or equal to –5) regardless of the score of node G. Thus we need not bother to explore node G at all. Of course, cutting out one node may not appear to justify the expense of keeping track of the limits and checking them, but if we were exploring this tree to six ply, then we would have eliminated not a single node but an entire tree three ply deep.

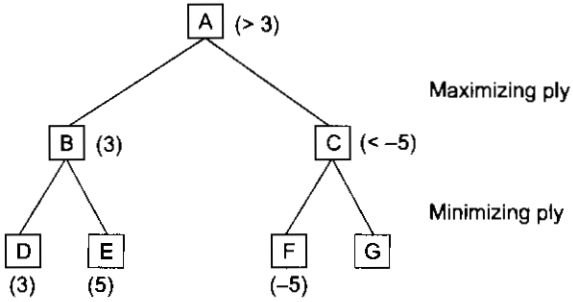


Fig. 12.4 *An Alpha Cutoff*

To see how the two thresholds, alpha and beta, can both be used, consider the example shown in Fig. 12.5 In searching this tree, the entire subtree headed by B is searched, and we discover that at A we can expect a score of at least 3. When this alpha value is passed down to F, it will enable us to skip the exploration of L. Let's see why. After K is examined, we see that I is guaranteed a maximum score of 0, which means that F is

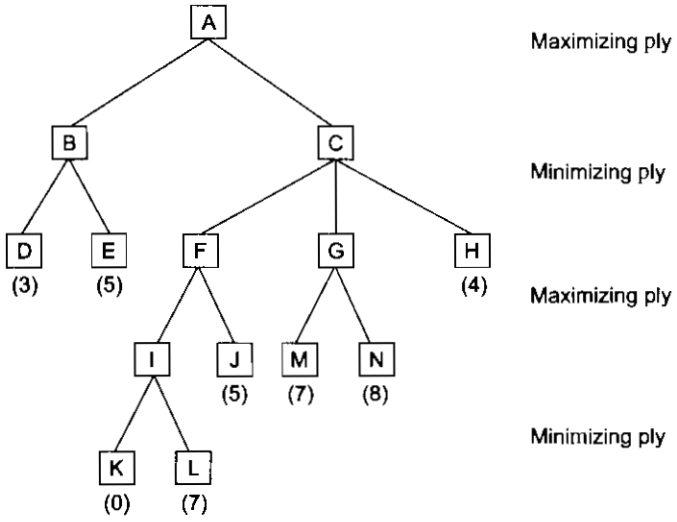


Fig. 12.5 *Alpha and Beta Cutoffs*

[3]In this figure, we return to the use of a single STATIC function from the point of view of the maximizing player.

guaranteed a minimum of 0. But this is less than alpha's value of 3, so no more branches of I need be considered. The maximizing player already knows not to choose to move to C and then to I since, if that move is made, the resulting score will be no better than 0 and a score of 3 can be achieved by moving to B instead. Now let's see how the value of beta can be used. After cutting off further exploration of I, J is examined, yielding a value of 5, which is assigned as the value of F (since it is the maximum of 5 and 0). This value becomes the value of beta at node C. It indicates that C is guaranteed to get a 5 or less. Now we must expand G. First M is examined and it has a value of 7, which is passed back to G as its tentative value. But now 7 is compared to beta (5). It is greater, and the player whose turn it is at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least 7, since there is an alternative move to F, which will lead to a score of 5. Thus it is not necessary to explore any of the other branches of G.

From this example, we see that at maximizing levels, we can rule out a move early if it becomes clear that its value will be less than the current threshold, while at minimizing levels, search will be terminated if values that are greater than the current threshold are discovered. But ruling out a possible move by a maximizing player actually means cutting off the search at a minimizing level. Look again at the example in Fig. 12.4. Once we determine that C is a bad move from A, we cannot bother to explore G, or any other paths, at the minimizing level below C. So the way alpha and beta are actually used is that search at a minimizing level can be terminated when a value less than alpha is discovered, while a search at a maximizing level can be terminated when a value greater than beta has been found. Cutting off search at a maximizing level when a high value is found may seem counterintuitive at first, but if you keep in mind that we only get to a particular node at a maximizing level if the minimizing player at the level above chooses it, then it makes sense.

Having illustrated the operation of alpha-beta pruning with examples, we can now explore how the MINIMAX procedure described in Section 12.2 can be modified to exploit this technique. Notice that at maximizing levels, only beta is used to determine whether to cut off the search, and at minimizing levels only alpha is used. But at maximizing levels alpha must also be known since when a recursive call is made to MINIMAX, a minimizing level is created, which needs access to alpha. So at maximizing levels alpha must be known not so that it can be used but so that it can be passed down the tree. The same is true of minimizing levels with respect to beta. Each level must receive both values, one to use and one to pass down for the next level to use.

The MINIMAX procedure as it stands does not need to treat maximizing and minimizing levels differently since it simply negates evaluations each time it changes levels. It would be nice if a comparable technique for handling alpha and beta could be found so that it would still not be necessary to write separate procedures for the two players. This turns out to be easy to do. Instead of referring to alpha and beta, MINIMAX uses two values, USE-THRESH and PASS-THRESH. USE-THRESH is used to compute cutoffs. PASS-THRESH is merely passed to the next level as its USE-THRESH. Of course, USE-THRESH must also be passed to the next level, but it will be passed as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth. Just as values had to be negated each time they were passed across levels, so too must these thresholds be negated. This is necessary so that, regardless of the level of the search, a test for greater than will determine whether a threshold has been crossed. Now there need still be no difference between the code required at maximizing levels and that required at minimizing ones.

We have now described how alpha and beta values are passed down the tree. In addition, we must decide how they are to be set. To see how to do this, let's return first to the simple example of Fig. 12,4. At a maximizing level, such as that of node A, alpha is set to be the value of the best successor that has yet been found. (Notice that although at maximizing levels it is beta that is used to determine cutoffs, it is alpha whose new value can be computed. Thus at any level, USE-THRESH will be checked for cutoffs and PASS-THRESH will be updated to be used later.) But if the maximizing node is not at the top of the tree, we must also consider

the alpha value that was passed down from a higher node. To see how this works, look again at Fig. 12.5 and consider what happens at node F. We assign the value 0 to node I on the basis, of examining node K. This is so far the best successor of F. But from an earlier exploration of the subtree headed by B, alpha was set to 3 and passed down from A to F. Alpha should not be reset to 0 on the basis of node I. It should stay as 3 to reflect the best move found so far in the entire tree. Thus we see that at a maximizing level, alpha should be set to either the value it had at the next-highest maximizing level or the best value found at this level, whichever is greater. The corresponding statement can be made about beta at minimizing levels. In fact, what we want to say is that at any level, PASS-THRESH should always be the maximum of the value it inherits from above and the best move found at its level. If PASS-THRESH is updated, the new value should be propagated both down to lower levels and back up to higher ones so that it always reflects the best move found anywhere in the tree.

At this point, we notice that we are doing the same thing in computing PASS-THRESH that we did in MINIMAX to compute BEST-SCORE. We might as well eliminate BEST-SCORE and let PASS-THRESH serve in its place.

With these observations, we are in a position to describe the operation of the function MINIMAX-A-B, which requires four arguments, *Position, Depth, Use-Thresh,* and *Pass-Thresh.* The initial call, to choose a move for PLAYER-ONE from the position CURRENT, should be

```
MINIMAX-A-B(CURRENT,
            0,
            PLAYER-ONE,
            maximum value STATIC can compute,
            minimum value STATIC can compute)
```

These initial values for *Use-Thresh* and *Pass-Thresh* represent the worst values that each side could achieve.

Algorithm: MINIM AX-A-B(Position, Depth, Player, Use-Thresh, Pass-Thresh)

1. If DEEP-ENOUGH(*Position, Depth*), then return the structure
   VALUE = *STATIC*(*Position, Player*);
   PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVE- GEN(Position, Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have been returned if DEEP-ENOUGH had returned TRUE.
4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.
   For each element SUCC of SUCCESSORS:
   (a) Set RESULT-SUCC to
   MINIMAX-A-B(SUCC, *Depth* + 1, OPPOSITE(P*layer*),
   –Pass-Thresh, –Use-Thresh).
   (b) Set NEW-VALUE to –VALUE(RESULT-SUCC).
   (c) If NEW-VALUE > *Pass-Thresh,* then we have found a successor that is better than any that have been examined so far. Record this by doing the following.
      (i) Set *Pass-Thresh* to NEW-VALUE.
      (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

(d) If *Pass-Thresh* (reflecting the current best value) is not better than *Use-Thresh*, then we should stop examining this branch. But both thresholds and values have been inverted. So if *Pass-Thresh* >= *Use-Thresh*, then return immediately with the value

VALUE = *Pass-Thresh*
PATH = BEST-PATH

5. Return the structure
VALUE = *Pass-Thresh*
PATH = BEST-PATH

The effectiveness of the alpha-beta procedure depends greatly on the order in which paths are examined. If the worst paths are examined first, then no cutoffs at all will occur. But, of course, if the best path were known in advance so that it could be guaranteed to be examined first, we would not need to bother with the search process. If, however, we knew how effective the pruning technique is in the perfect case, we would have an upper bound on its performance in other situations. It is possible to prove that if the nodes are perfectly ordered, then the number of terminal nodes considered by a search to depth d using alpha-beta pruning is approximately equal to twice the number of terminal nodes generated by a search to depth $d/2$ without alpha-beta [Knuth and Moore, 1975].

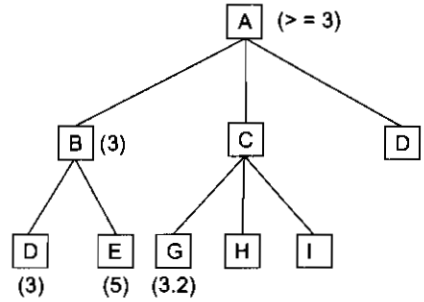


Fig. 12.6 *A Futility Cutoff*

A doubling of the depth to which the search can be pursued is a significant gain. Even though all of this improvement cannot typically be realized, the alpha-beta technique is a significant improvement to the minimax search procedure. For a more detailed study of the average branching factor of the alpha-beta procedure, see Baudet [1978] and Pearl [1982].

The idea behind the alpha-beta procedure can be extended to cut off additional paths that appear to be at best only slight improvements over paths that have already been explored. In step 4(*d*), we cut off the search if the path we were exploring was not better than other paths already found. But consider the situation shown in Fig. 12.6. After examining node G, we see that the best we can hope for if we make move C is a score of 3.2. We know that if we make move B we are guaranteed a score of 3. Since 3.2 is only very slightly better than 3, we should perhaps terminate our exploration of C now. We could then devote more time to exploring other parts of the tree where there may be more to gain. Terminating the exploration of a subtree that offers little possibility for improvement over other known paths is called a *futility cutoff*.

12.4 ADDITIONAL REFINEMENTS

In addition to alpha-beta pruning, there are a variety of other modifications to the minimax procedure that can also improve its performance. Four of them are discussed briefly in this section, and we discuss one other important modification in the next section.

12.4.1 Waiting for Quiescence

As we suggested above, one of the factors that should sometimes be considered in determining when to stop going deeper in the search tree is whether the situation is relatively stable. Consider the tree shown in Fig. 12.7. Suppose that when node B is expanded one more level, the result is that shown in Fig. 12.8. When we looked one move ahead, our estimate of the worth of B changed drastically. This might happen, for example, in the middle of a piece exchange. The opponent has significantly improved the immediate appearance of his or her position by initiating a piece exchange. If we stop exploring the tree at this level, we assign the value −4 to B and therefore decide that B is not a good move.

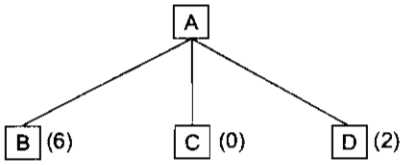


Fig. 12.7 *The Beginning of a Search*

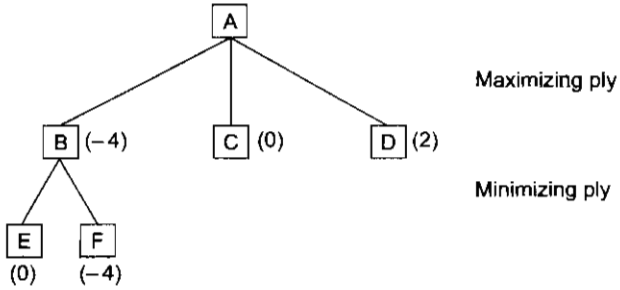


Fig. 12.8 *The Beginning of an Exchange*

To make sure that such short-term measures do not unduly influence our choice of move, we should continue the search until no such drastic change occurs from one level to the next. This is called waiting for *quiescence*. If we do that, we might get the situation shown in Fig. 12.9, in which the move to B again looks like a reasonable move for us to make since the other half of the piece exchange has occurred. A very general algorithm for quiescence can be found in Beal [1990].

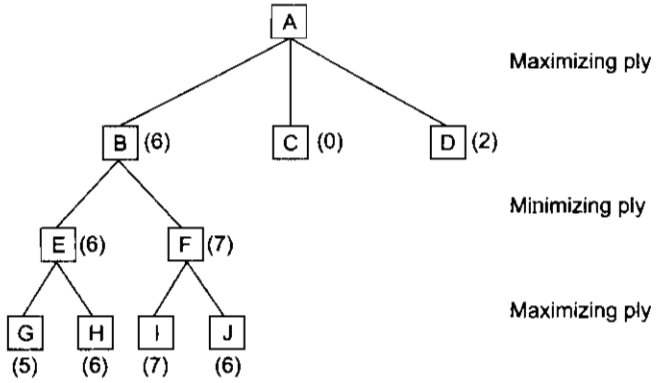


Fig. 12.9 *The Situation Calms Down*

Waiting for quiescence helps in avoiding the *horizon effect*, in which an inevitable bad event can be delayed by various tactics until it does not appear in the portion of the game tree that minimax explores. The horizon effect can also influence a program's perception of good moves. The effect may make a move look good despite the fact that the move might be better if delayed past the horizon. Even with quiescence, all fixed-depth search programs are subject to subtle horizon effects.

12.4.2 Secondary Search

One good way of combating the horizon effect is to double-check a chosen move to make sure that a hidden pitfall does not exist a few moves farther away than the original search explored. Suppose we explore a game tree to an average depth of six ply and, on the basis of that search, choose a particular move. Although it would have been too expensive to have searched the entire tree to a depth of eight, it is not very expensive to search the single chosen branch an additional two levels to make sure that it still looks good. This technique is called *secondary search*.

One particularly successful form of secondary search is called *singular extensions*. The idea behind singular extensions is that if a leaf node is judged to be far superior to its siblings and if the value of the entire search

depends critically on the correctness of that node's value, then the node is expanded one extra ply. This technique allows the search program to concentrate on tactical, forcing combinations. It employs a purely syntactic criterion, choosing interesting lines of play without recourse to any additional domain knowledge. The DEEP THOUGHT chess computer [Anantharaman *et al.*, 1990] has used singular extensions to great advantage, finding midgame mating combinations as long as thirty-seven moves, an impossible feat for fixed-depth minimax.

12.4.3 Using Book Moves

For complicated games taken as wholes, it is, of course, not feasible to select a move by simply looking up the current game configuration in a catalogue and extracting the correct move. The catalogue would be immense and no one knows how to construct it. But for some segments of some games, this approach is reasonable. In chess, for example, both opening sequences and endgame sequences are highly stylized. In these situations, the performance of a program can often be considerably enhanced if it is provided with a list of moves (called *book moves*) that should be made. The use of book moves in the opening sequences and endgames, combined with the use of the minimax search procedure for the midgame, provides a good example of the way that knowledge and search can be combined in a single program to produce more effective results than could either technique on its own.

12.4.4 Alternatives to Minimax

Even with the refinements above, minimax still has some problematic aspects. For instance, it relies heavily on the assumption that the opponent will always choose the optimal move. This assumption is acceptable in winning situations where a move that is guaranteed to be good for us can be found. But, as suggested in Berliner [1977], in a losing situation it might be better to take the risk that the opponent will make a mistake. Suppose we must choose between two moves, both of which, if the opponent plays perfectly, lead to situations that are very bad for us, but one is slightly less bad than the other. But further suppose that the less promising move could lead to a very good situation for us if the opponent makes a single mistake. Although the minimax procedure would choose the guaranteed bad move, we ought instead to choose the other one, which is probably slightly worse but possibly a lot better. A similar situation arises when one move appears to be only slightly more advantageous than another, assuming that the opponent plays perfectly. It might be better to choose the less advantageous move if it could lead to a significantly superior situation if the opponent makes a mistake. To make these decisions well, we must have access to a model of the individual opponent's playing style so that the likelihood of various mistakes can be estimated. But this is very hard to provide.

As a mechanism for propagating estimates of position strengths up the game tree, minimax stands on shaky theoretical grounds. Nau. [1980] and Pearl [1983] have demonstrated that for certain classes of game trees, e.g., uniform trees with random terminal values, the deeper the search, the *poorer* the result obtained by minimaxing. This "pathological" behavior of amplifying/error-prone heuristic estimates has not been observed in actual game-playing programs, however. It seems that game trees containing won positions and nonrandom distributions of heuristic estimates provide environments that are conducive to minimaxing.

12.5 ITERATIVE DEEPENING

A number of ideas for searching two-player game trees have led to new algorithms for single-agent heuristic search, of the type described in Chapter 3. One such idea is *iterative deepening*, originally used in a program called CHESS 4.5 [Slate and Atkin, 1977]. Rather than searching to a fixed depth in the game tree, CHESS 4.5 first searched only a single ply, applying its static evaluation function to the result of each of its possible moves. It then initiated a new minimax search, this time to a depth of two ply. This was followed by a three-

ply search, then a four-ply search, etc. The name "iterative deepening" derives from the fact that on each iteration, the tree is searched one level deeper. Figure 12.10 depicts this process.

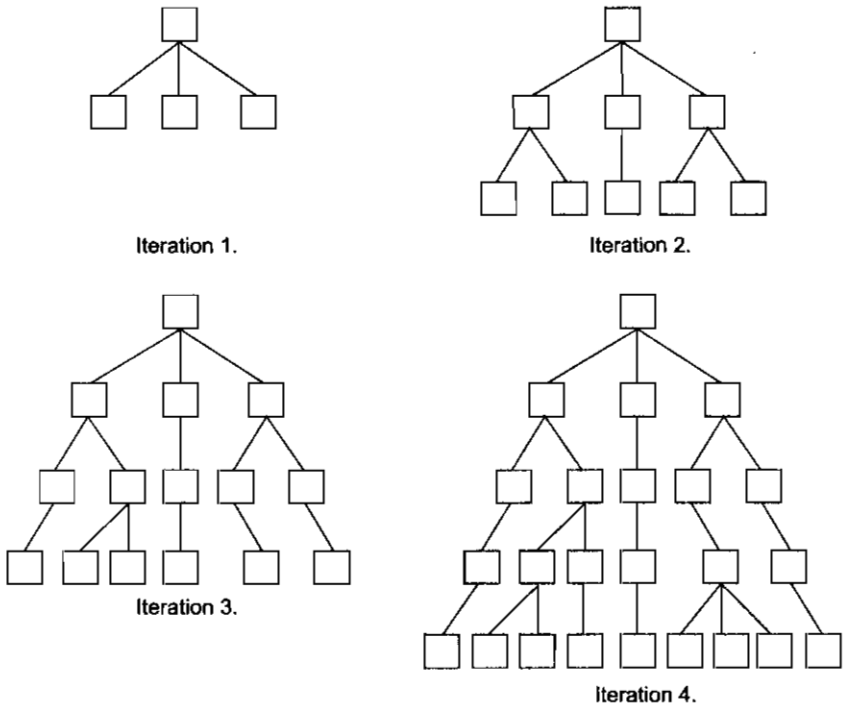


Fig. 12.10 *Iterative Deepening*

On the face of it, this process seems wasteful. Why should we be interested in any iteration except the final one? There are several reasons. First, game-playing programs are subject to time constraints. For example, a chess program may be required to complete all its moves within two hours. Since it is impossible to know in advance how long a fixed-depth tree search will take (because of variations in pruning efficiency and the need for selective search), a program may find itself running out of time. With iterative deepening, the current search can be aborted at any time and the best move found by the previous iteration can be played. Perhaps more importantly, previous iterations can provide invaluable move-ordering constraints. If one move was judged to be superior to its siblings in a previous iteration, it can be searched first in the next iteration. With effective ordering, the alpha-beta procedure can prune many more branches, and total search time can be decreased drastically. This allows more time for deeper iterations.

Years after CHESS 4.5's success with iterative deepening, it was noticed [Korf, 1985a] that the technique could also be applied effectively to single-agent search to solve problems like the 8-puzzle. In Section 2.2.1, we compared two types of uninformed search, depth-first search and breadth-first search. Depth-first search was efficient in terms of space but required some cutoff depth in order to force backtracking when a solution was not found. Breadth-first search was guaranteed to find the shortest solution path but required inordinate amounts of space because all leaf nodes had to be kept in memory. An algorithm called depth-first iterative deepening (DFID) combines the best aspects of depth-first and breadth-first search.

Algorithm: Depth-First Iterative Deepening

1. Set SEARCH-DEPTH = 1.
2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution path is found, then return it.
3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2.

Clearly, DFID will find the shortest solution path to the goal state. Moreover, the maximum amount of memory used by DFID is proportional to the number of nodes in that solution path. The only disturbing fact is that all iterations but the final one are essentially wasted. However, this is not a serious problem. The reason is that most of the activity during any given iteration occurs at the leaf-node level. Assuming a complete tree, we see that there are as many leaf nodes at level n as there are total nodes in levels 1 through n. Thus, the work expended during the nth iteration is roughly equal to the work expended during all previous iterations. This means that DFID is only slower than depth-first search by a constant factor. The problem with depth-first search is that there is no way to know in advance how deep the solution lies in the search space. DFID avoids the problem of choosing cutoffs without sacrificing efficiency, and, in fact, DFID is the optimal algorithm (in terms of space and time) for uninformed search.

But what about informed, heuristic search? Iterative deepening can also be used to improve the performance of the A* search algorithm [Korf, 1985a]. Since the major practical difficulty with A* is the large amount of memory it requires to maintain the search node lists, iterative deepening can be of considerable service.

Algorithm: Iterative-Deepening-A*

1. Set THRESHOLD = the heuristic evaluation of the start state.
2. Conduct a depth-first search, pruning any branch when its total cost function ($g + h'$) exceeds THRESHOLD.[4] If a solution path is found during the search, return it.
3. Otherwise, increment THRESHOLD by the minimum amount it was exceeded during the previous step, and then go to Step 2.

Like A*, Iterative-Deepening-A* (IDA*) is guaranteed to find an optimal solution, provided that h' is an admissible heuristic. Because of its depth-first search technique, IDA* is very efficient with respect to space. IDA* was the first heuristic search algorithm to find optimal solution paths for the 15-puzzle (a 4x4 version of the 8-puzzle) within reasonable time and space constraints.

12.6 REFERENCES ON SPECIFIC GAMES

In this chapter we have discussed search-based techniques for game playing. We discussed the basic minimax algorithm and then introduced a series of refinements to it. But even with these refinements, it is still difficult to build good programs to play difficult games. Every game, like every AI task, requires a careful combination of search and knowledge.

Chess

Research on computer chess actually predates the field we call artificial intelligence. Shannon [1950] was the first to propose a method for automating the game, and two early chess programs were written by Greenblatt *et al.* [1967] and Newell and Simon [1972].

Chess provides a well-defined laboratory for studying the trade-off between knowledge and search. The more knowledge a program has, the less searching it needs to do. On the other hand, the deeper the search, the less knowledge is required. Human chess players use a great deal of knowledge and very little search—they

[4] Recal g stands for the cost so far in reaching the current node, and h' stands for the heuristic estimate of the distance from the node to the goal.

typically investigate only 100 branches or so in deciding a move. A computer, on the other hand, is capable of evaluating millions of branches. Its chess knowledge is usually limited to a static evaluation function. Deep-searching chess programs have been calibrated on exercise problems in the chess literature and have even discovered errors in the official human analyses of the problems.

A chess player, whether human or machine, carries a numerical rating that tells how well it has performed in competition with other players. This rating lets us evaluate in an absolute sense the relative trade-offs between search and knowledge in this domain. The recent trend in chess-playing programs is clearly away from knowledge and toward faster brute force search. It turns out that deep, full-width search (with pruning) is sufficient for competing at very high levels of chess. Two examples of highly rate chess machines are HITECH [Berliner and Ebeling, 1989] and DEEP THOUGHT [Anantharaman *et al.*, 1990], both of which have beaten human grandmasters and both of which use custom-built parallel hardware to speed up legal move generation and heuristic evaluation.

Checkers

Work on computer checkers began with Samuel [1963]. Samuel's program had an interesting learning component which allowed its performance to improve with experience. Ultimately, the program was able to beat its author. We look more closely at the learning mechanisms used by Samuel in Chapter 17.

Go

Go is a very difficult game to play by machine since the average branching factor of the game tree is very high. Brute force search, therefore, is not as effective as it is in chess. Human go players make up for their inability to search deeply by using a great deal of knowledge about the game. It is probable that go-playing programs must also be knowledge-based, since today's brute-force programs cannot compete with humans. For a discussion of some of the issues involved, see Wilcox [1988].

Backgammon

Unlike chess, checkers, and go, a backgammon program must choose its moves with incomplete information about what may happen. If all the possible dice rolls are considered, the number of alternatives at each level of the search is huge. With current computational power, it is impossible to search more than a few ply ahead. Such a search will not expose the strengths and weaknesses of complex blocking positions, so knowledge-intensive methods must be used. One program that uses such methods is BKG Berliner [1980]. BKG actually does no searching at all but relies instead on positional understanding and understanding of how its goals should change for various phases of play. Like its chess-playing cousins, BKG has reached high levels of play, even beating a human world champion in a short match.

NEUROGAMMON [Tesauro and Sejnowski, 1989] is another interesting backgammon program. It is based on a neural network model that learns from experience. Neurogammon is one of the few competitive game-playing programs that relies heavily on automatic learning.

Othello

Othello is a popular board game that is played on an 8x8 grid with bi-colored pieces. Although computer programs have already achieved world-championship level play [Rosenbloom, 1982; Lee and Mahajan, 1990], humans continue to study the game and international tournaments are held regularly. Computers are not permitted to compete in these tournaments, but it is believed that the best programs are stronger than the best humans. High-performance Othello programs rely on fast brute-force search and table lookup.

The Othello experience may shed some light on the future of computer chess. Will top human players in the future study chess games between World Champion computers in the same way that they study classic human grandmaster matches today? Perhaps it will turn out that the different search versus knowledge trade-

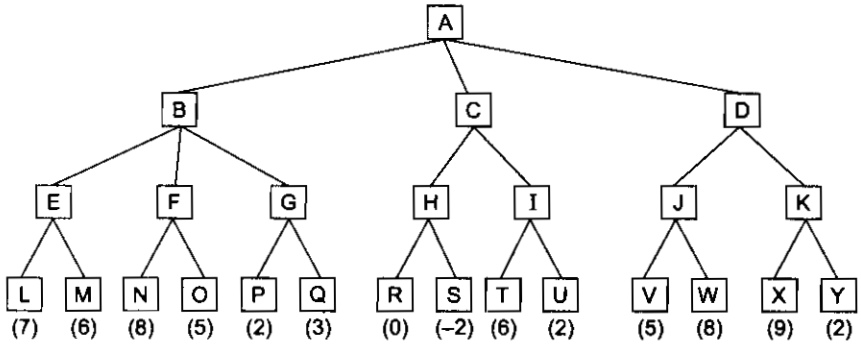
offs made by humans and computers will make it impossible for either of them to benefit from the experiences of the other.

Others

Levy [1988] contains a number of classic papers on computer game playing. The papers cover the games listed above as well as bridge, scrabble, dominoes, go-moku, hearts, and poker.

EXERCISES

1. Consider the following game tree in which static scores are all from the first player's point of view:



   Suppose the first player is the maximizing player. What move should be chosen?
2. In the game tree shown in the previous problem, what nodes would not need to be examined using the alpha-beta pruning procedure?
3. Why does the search in game-playing programs always proceed forward from the current position rather than backward from a goal state?
4. Is the minimax procedure a depth-first or breadth-first search procedure?
5. The minimax algorithm we have described searches a game tree. But for some games, it might be better to search a graph and to check, each time a position is generated, if it has been generated and evaluated before. Under what circumstances would this be a good idea? Modify the minimax procedure to do this.
6. How would the minimax procedure have to be modified to be used by a program playing a three- or four-person game rather than a two-person one?
7. In the context of the search procedure described in Section 12.3, does the ordering of the list of successor positions created by MOVEGEN matter? Why or why not? If it does matter, how much does it matter (i.e., how much effort is reasonable for ordering it)?
8. Implement the alpha-beta search procedure. Use it to play a simple game such as tic-tac-toe.
9. Apply DFID to the water jug problem of Section 2.1.