representation contains argument indices and predicate indices which can be extremely helpful in the inference process.

A very simple example illustrates the foregoing points. Suppose that S consists of the set of clauses

$$\text{SouthOf(river2,}x\text{), NorthOf(river1,}x\text{)} \rightarrow \text{Between(river1, river2,}\ x\text{)} \quad (12.4)$$
$$\rightarrow \text{SouthOf (}u\text{, silo30)} \quad (12.5)$$
$$\rightarrow \text{NorthOf (river1, silo30)} \quad (12.6)$$

Clause (12.5) might arise when it is determined that "silo30" is south of some feature in the image whose identity is not known. *Bottom up inference* derives new assertions from old ones. Thus in the example above the variable substitutions

$$u = \text{river2} \qquad x = \text{silo30}$$

match assertion (12.5) with the general clause (12.4) and allow the inference

$$\text{NorthOf(river1, silo30)}$$
$$\rightarrow \text{Between(river1, river2, silo30)} \quad (12.7)$$

Consequently, use (12.6) and (12.7) to assert

$$\rightarrow \text{Between(river1, river2, silo 30)} \quad (12.8)$$

Suppose that this was not the case: that is, that

$$\text{Between(river1, river2, silo30)} \rightarrow \quad (12.9)$$

and that $S = \{(12.4), (12.9)\}$. One could then use *top-down inference*, which infers new denials from old ones. In this case

$$\text{NorthOf(river1,silo30), SouthOf(river2,silo30)} \rightarrow \quad (12.10)$$

follows with the variable substitution x = silo30. This can be interpreted as follows: "If x is really silo30, then it is neither north of river1 or south of river2." Figure 12.2 shows two examples using the network notation.

Now suppose the goal is to prove that (12.8) logically follows from (12.4) through (12.6) and the substitutions. The strategy would be to negate (12.8), add it to the data base, and show that the empty clause can be derived. Negating an assertion produces a denial, in this case (12.9), and now the set of axioms (including the denial) consists of {(12.4), (12.5), (12.6), (12.9)}. It is easy to repeat the earlier steps to the point where the set of clauses includes (12.8) and (12.9), which resolve to produce the empty clause. Hence the theorem is proved.

12.1.6 Predicate Calculus And Knowledge Representation

Pure predicate calculus has strengths and weaknesses as a knowledge representation system. Some of the seeming weaknesses can be overcome by technical "tricks." Some are not inherent in the representation but are a property of the common interpreters used on it (i.e., on state-of-the-art theorem provers). Some problems are relatively basic, and the majority opinion seems to be that first order

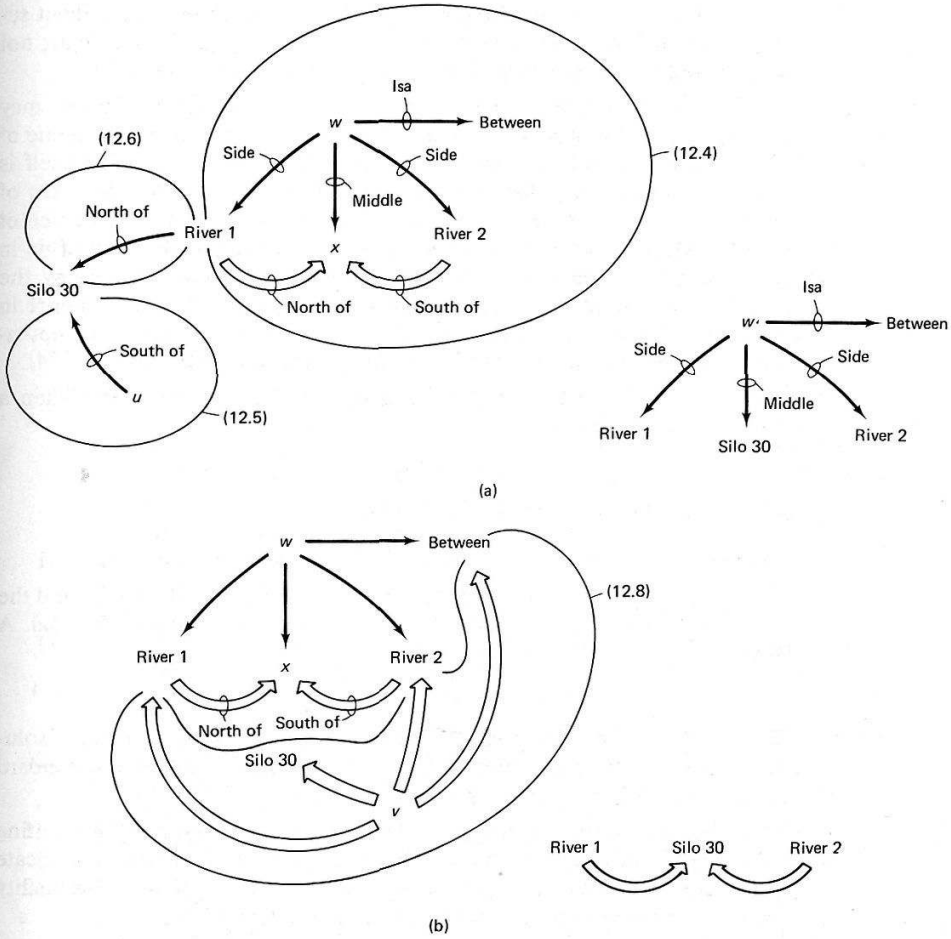


Fig. 12.2 Resolution using networks. (a) Bottom-up inference as a result of substitutions u = river2, x = silo30. (b) Top-down inference as a result of substitutions $w = v$, x = silo30.

predicate logic must be extended in order to become a representation scheme that is satisfactorily matched to the power of the deductive methods applied by human beings. Opinion is divided on the technical aspects of such enhancements. Predicate calculus has several strengths, some of which we list below.

1. Predicate logic is a well-polished gem, having been refined and studied for several generations. It was designed to represent knowledge and inference. One knows what it means. Its model theory and proof theory are explicit and lucid [Hayes 1977; 1980].

2. Predicate logic can be considered a language with a machine-independent semantics; the meaning of the language is determined by the laws of logic, not the actual programming system upon which the logic is "executed."
3. Predicate calculus clauses with only one conclusion atom (Horn clauses) may be considered as "procedures," with the single conclusion being the name of the procedure and the conditions being the procedure body, which itself is made up of procedure calls. This view of logic leads to the development of predicate logic-based programming languages (such as PROLOG [Warren et al. 1977; McDermott 1980]). These programs exhibit nondeterminism in several interesting ways; the order of computations is not specified by the proof procedure (and is not restricted by it, either). Several resolutions are in general possible for any clause; the combinations determine many computations and several distinguishable forms of nondeterminism [Kowalski 1974].
4. Predicate logic may be interpreted as a problem-reduction system. Then a (Horn) clause of the form

$$\rightarrow B$$

represents a solved problem. One of the form

$$A_1, \ldots, A_n \rightarrow$$

with variables $x_1, \ldots, x_k$ is a goal statement, or command, which is to find the x's that solve the problems $A_1, \ldots, A_n$. Finding the x's solves the goal. A clause

$$A_1, \ldots, A_n \rightarrow B$$

is a solution method, which reduces the solution of B to a combination of solutions of A's. This interpretation of Horn clauses maps cleanly into a standard and–or goal tree formulation of problem solving.
5. Resolutions may be performed on the left or right of clauses, and the resulting derivation trees correspond, in the problem-solving interpretation of predicate calculus, to top-down and bottom-up versions of problem solving. This duality is very important in conceptualizing aspects of problem solving.
6. There is a uniform proof procedure for logic which is guaranteed to prove in finite time any true theorem (logic is semidecidable and complete). No false theorems are provable (logic is correct). These and other good formal properties are important when establishing formally the properties of a knowledge representation system.

Predicate calculus is not a favorite of everyone, however; some of the (perceived) disadvantages are given below, together with ways they might be countered.

1. Sometimes the axioms necessary to implement relatively common concepts are not immediately obvious. A standard example is "equality." These largely technical problems are annoying but not basic.

2. The "first order" in first order predicate calculus means that the system

does not allow clauses with variables ranging over an infinite number of predicates, functions, assertions and sentences (e.g., "All unary functions are boring" cannot be stated directly). This problem may be ameliorated by a notational trick; the situations under which predicates are true are indicated with a Holds predicate. Thus instead of writing On(block1, surface, situation1), write Holds (On(block1,surface), situation1). This notation allows inferences about many situations with only one added axiom. The "situational calculus" reappears in Section 12.3.1. Another useful notational trick is a Diff relation, which holds between two terms if they are syntactically different. There are infinitely many axioms asserting that terms are different; the actual system can be made to incorporate them implicitly in a well-defined way. The Diff relation is also used in Section 12.3.1.

3. The *frame problem* (so called for historical reasons and not related to the frames described in Section 10.3.1) is a classic bugbear of problem-solving methods including predicate logic. One aspect of this problem is that for technical reasons, it must be explicitly stated in axioms that describe actions (in a general sense a visual test is an action) that almost all assertions were true in a world state *remain* true in the new world state after the action is performed. The addition of these new axioms causes a huge increase in the "bureaucratic overhead" necessary to maintain the state of the world. Currently, no really satisfactory way of handling this problem has been devised. The most common way to attack this aspect of the frame problem is to use explicit "add lists" and "delete lists" ([Fikes 1977], Chapter 13) which attempt to specify exactly what changes when an action occurs. New true assertions are added and those that are false after an action must be deleted. This device is useful, but examples demonstrating its inadequacy are readily constructed. More aspects of the frame problem are given in Chapter 13.

4. There are several sorts of reasoning performed by human beings that predicate logic does not pretend to address. It does not include the ability to describe its own formulae (a form of "quotation"), the notion of defaults, or a mechanism for plausible reasoning. Extensions to predicate logic, such as modal logic, are classically motivated. More recently, work on extensions addressing the topics above have begun to receive attention [McCarthy 1978; Reiter 1978; Hayes 1977]. There is still active debate as to whether such extensions can capture many important aspects of human reasoning and knowledge within the model-theoretic system. The contrary view is that in some reasoning, the very *process* of reasoning itself is an important part of the semantics of the representation. Examples of such extended inference systems appear in the remainder of this chapter, and the issues are addressed in more detail in the next section.

12.2 COMPUTER REASONING

Artificial intelligence in general and computer vision in particular must be concerned with *efficiency* and *plausibility* in inference [Winograd 1978]. Computer-based knowledge representations and their accompanying inference processes often sacrifice classical formal properties for gains in control of the inference process and for flexibility in the sorts of "truth" which may be inferred.

does not allow clauses with variables ranging over an infinite number of predicates, functions, assertions and sentences (e.g., "All unary functions are boring" cannot be stated directly). This problem may be ameliorated by a notational trick; the situations under which predicates are true are indicated with a Holds predicate. Thus instead of writing On(block1, surface, situation1), write Holds (On(block1,surface), situation1). This notation allows inferences about many situations with only one added axiom. The "situational calculus" reappears in Section 12.3.1. Another useful notational trick is a Diff relation, which holds between two terms if they are syntactically different. There are infinitely many axioms asserting that terms are different; the actual system can be made to incorporate them implicitly in a well-defined way. The Diff relation is also used in Section 12.3.1.

3. The *frame problem* (so called for historical reasons and not related to the frames described in Section 10.3.1) is a classic bugbear of problem-solving methods including predicate logic. One aspect of this problem is that for technical reasons, it must be explicitly stated in axioms that describe actions (in a general sense a visual test is an action) that almost all assertions were true in a world state *remain* true in the new world state after the action is performed. The addition of these new axioms causes a huge increase in the "bureaucratic overhead" necessary to maintain the state of the world. Currently, no really satisfactory way of handling this problem has been devised. The most common way to attack this aspect of the frame problem is to use explicit "add lists" and "delete lists" ([Fikes 1977], Chapter 13) which attempt to specify exactly what changes when an action occurs. New true assertions are added and those that are false after an action must be deleted. This device is useful, but examples demonstrating its inadequacy are readily constructed. More aspects of the frame problem are given in Chapter 13.

4. There are several sorts of reasoning performed by human beings that predicate logic does not pretend to address. It does not include the ability to describe its own formulae (a form of "quotation"), the notion of defaults, or a mechanism for plausible reasoning. Extensions to predicate logic, such as modal logic, are classically motivated. More recently, work on extensions addressing the topics above have begun to receive attention [McCarthy 1978; Reiter 1978; Hayes 1977]. There is still active debate as to whether such extensions can capture many important aspects of human reasoning and knowledge within the model-theoretic system. The contrary view is that in some reasoning, the very *process* of reasoning itself is an important part of the semantics of the representation. Examples of such extended inference systems appear in the remainder of this chapter, and the issues are addressed in more detail in the next section.

12.2 COMPUTER REASONING

Artificial intelligence in general and computer vision in particular must be concerned with *efficiency* and *plausibility* in inference [Winograd 1978]. Computer-based knowledge representations and their accompanying inference processes often sacrifice classical formal properties for gains in control of the inference process and for flexibility in the sorts of "truth" which may be inferred.

Automated inference systems usually have inference methods that achieve efficiency through implementational, computation-based, inference criteria. For example, truth may be defined as a successful lookup in a data base, falsity as the failure to find a proof with a given allocation of computational resources, and the establishment of truth may depend on the order in which deductions are made.

The semantics of computer knowledge representations is intimately related to the inference process that acts on them. Therefore, it is possible to define knowledge representations and interpreters in computers whose properties differ fairly radically from those of classical representations and proof procedures, such as the first-order predicate calculus. For instance, although the systems are deterministic, they may not be formally consistent (loosely, they may contain contradictory information). They may not be complete (they cannot derive all true theorems from the givens); it may be possible to prove P from Q but $\neg P$ from Q and R. The set of provable theorems may not be recursively enumerable [Reiter 1978]. Efforts are being made to account for the "extended inference" needed by artificial intelligence using more or less classical logic [McCarthy 1978; Reiter 1978; Hayes 1977; 1978a; 1978b; Kowalski 1974, 1979]. In each case, the classical view of logic demands that the deductive process and the deducible truths be independent. On the other hand, it is reasonable to devote attention to developing a nonclassical semantics of these inference processes; this topic is in the research stage at this writing.

Several knowledge representations and inference methods using them are "classical" in the artificial intelligence world; that is, they provide paradigmatic methods of dealing with the issues of computational inference. They include STRIPS [Fikes and Nilsson 1971], the situational calculus [McCarthy and Hayes 1969], PLANNER and CONNIVER [Hewitt 1972; Sussman and McDermott 1972], and semantic net representations [Hendrix 1979; Brachman 1979].

To illustrate the issue of consistency, and to illustrate how various sorts of propositions can be represented in semantic nets, we address the question of how the order of inference can affect the set of provable theorems in a system.

Consider the semantic net of Fig. 12.3. The idea is that in the absence of specific information to the contrary, one should assume that railroad bridges are narrow. There are exceptions, however, such as Bridge02 (which has a highway bridge above the rail bridge, say). The network is clearly inconsistent, but trouble is avoided if inferences are made "from specific to general." Such ordering implies that the system is incomplete, but in this case incompleteness is an advantage.

Simple ordering constraints are possible only with simple inferential powers in the system [Winograd 1978]. Further, there is as yet little formal theory on the effects of ordering rules on computational inference, although this has been an active topic [Reiter 1978].

12.3 PRODUCTION SYSTEMS

The last section explored why the process of inference itself could be an important part of the semantics of a knowledge representation system. This idea is an impor-

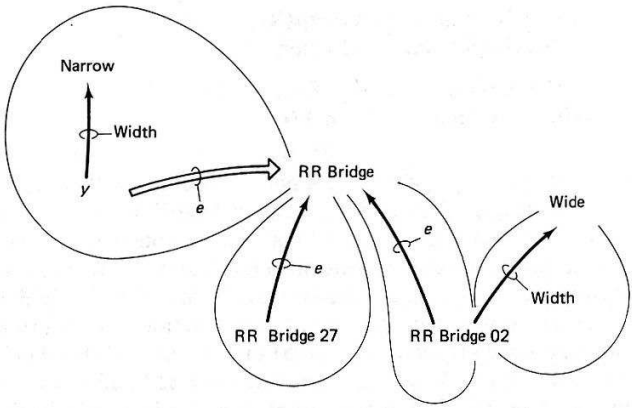


Fig. 12.3 An inconsistent network.

tant part of production systems. Perceived limitations in logic inference mechanisms and the seductive power of arbitrary algorithmic processes for inference has spawned the development of *rule-based* systems which differ from first-order logic in the following respects:

- Arbitrary additions and deletions to the clausal data base are allowed.
- An interpreter that controls the inference process in special ways is usually an integral part of the system.

Early examples of systems with the first addition are STRIPS [Fikes and Nilsson 1971] and PLANNER [Hewitt 1972]. Later examples of systems with both additions are given in [Waterman and Hayes-Roth 1978]. The virtues of trying to control inferences may be appreciated after our brief introduction to clausal automatic theorem proving, where there are no very good semantic heuristics to guide inferences. However, the price paid for restricting the inference process is the loss of formal properties of consistency and correctness of the system, which are not guaranteed in rule-based systems. We shall look in some detail at a particular form of rule-based inference system called production systems.

A *production system* supports a general sort of "inference." It has in common with resolution that matching is needed to identify which inference to make. It is different in that the action upon finding a matching data item is less constrained. Actions of arbitrary complexity are allowed. A production system consists of an explicit set of situation–action nodes, which can be applied against a data base of situations. For example, in a very constrained visual domain the rule

$$(\text{Green } (\text{Region } X)) \rightarrow (\text{Grass } (\text{Region } X)) \tag{12.11}$$

could infer directly the interpretation of a given region. Segmentation rules can also be developed; the following example merges two adjacent green regions into a single region.

(Green(Region X))$\wedge$(Green(Region Y))$\wedge$
(Adjacent(Region X), (Region Y))

$\rightarrow$ (Green(Region Z))$\wedge$((Region Z) :=
(Union(Region X, Region Y)))

These examples highlight several points. The first is that basic idea of production systems is simple. The rules are easy to "read" by both the programmer and his program and new rules are easily added. Although it is imaginable that "situations" could extend down to the pixel level, and production systems could be used (for instance) to find lines, the system overhead would render such an approach impractical. In the visual domain, the production system usually operates on the segmented image (Chapters 4 and 5) or with the high-level internal model. In the rules above, X and Y are variables that must be bound to specific instances of regions in a data base. This process of binding variables or matching can become very complex, and is one of the two central issues of this kind of inference. The other is how to choose rules from a set all of whose situations match the current situation to some degree.

12.3.1 *Production System Details*

In its simplest form a production system has three basic components:

1. A data base
2. A set of rules
3. An interpreter for the rules

The vision data base is usually a set of facts that are known about the visual environment. Often the rules are considered to be themselves a manipulable part of the data base. Examples of some visual facts may be

(ABOVE (Region 5) (Region 10))
(SIZE (Region 5) 300)
(SKY (Region 5)) (12.12)
(TOP (Region 5) 255)

The data base is the sole storage medium for all state variables of the system. In particular, unlike procedurally oriented languages, there is no provision for separate storage of control state information—no separate program counter, push-down stack, and so on [Davis and King 1975].

A rule is an ordered pair of *patterns* with a left-hand side and a right-hand side. A pattern may involve only data base primitives but usually will have variables and special forms as subpatterns which are matched against the data base by the interpreter. For example, applying the following rule to a data base which includes (12.12),

(TOP (Region X) (GreaterThan 200))

→ (12.13)

(SKY (Region X))

region 5 can be inferred to be sky. The left-hand side matches a set of data-base facts and this causes (SKY (Region 5)) to be added to the data base. This example shows the kinds of matching that the interpreter must do: (1) the primitive TOP in the data base fact matches the same symbol in the rule, (2) (Region X) matched (Region 5) and X is bound to 5 as a side effect, and (3) (GreaterThan 200) matches 255. Naturally, the user must design his own interpreter to recognize the meaning of such operational subpatterns.

However, even the form of the rules outlined so far is relatively restrictive. There is no reason why the right-hand side cannot do almost arbitrary things. For instance, the application of a rule may result in various productions being deleted or added from the set of productions; the data base of productions and assertions thus can be adaptive [Waterman and Hayes-Roth 1978]. Also, the right-hand side may specify programs to be run which can result in facts being asserted into the data base or actions performed.

Control in a basic production system is relatively simple: Rules are applied until some condition in the data base is reached. Rules may be applied in two distinct ways: (1) a match on the left-hand side of a rule may result in the addition of the consequences on the right-hand side to the data base, or (2) a match on the right-hand side may result in the addition of the antecedents in the left-hand side to the data base. The order of application of rules in the first case is termed *forward chaining* reasoning, where the objective is to see if a set of consequences can be derived from a given set of initial facts. The second case is known as *backward chaining*; the objective is to determine a set of facts that could have produced a particular consequence.

12.3.2 Pattern Matching

In the process of matching rules against the data base, several problems occur:

- Many rule situations may match data base facts
- Rules designed for a specific context may not be appropriate for larger context
- The pattern matching process may become very expensive
- The data base or set of rules may become unmanageably large.

The problem of multiple matches is important. Early systems simply resolved it by scanning the data base in a linear fashion and choosing the first match, but this is an ineffective strategy for large data bases, and has conceptual problems as well. Accordingly, strategies have evolved for dealing with these conflicts. Like most inference-controlling heuristics, their effectiveness can be domain-dependent, they can introduce incompleteness into the system, and so on.

On the principle of *least commitment*, when there are many chances of errors, one strategy is to apply the most general rule, defined by some metric on the com-

ponents of the pattern. One simple such metric is the number of elements in a pattern. Antithetical to this strategy is the heuristic of applying the *most specific pattern.* This may be appropriate where the likelihood of making a false inference is small, and where specific actions may be indicated (match (MAD DOG) with (MAD DOG), not with (DOG)). Another popular but inelegant technique is to exercise control over the *order of production application* by using state markers which are inserted into the data base by right-hand sides and looked for by left-hand sides.

1. $A \rightarrow B \wedge <\text{marker } 1>$.
2. $A \rightarrow B \wedge <\text{marker } 2>$.
3. $B \wedge <\text{marker } 1> \rightarrow C$.
4. $B \wedge <\text{marker } 2> \rightarrow D$.

Here if rule 1 is executed, "control goes to rule 3," i.e., rule 3 is now executable, whereas if rule 2 is applied, "control goes to rule 4." Similarly, such control paradigms as subroutining, iteration and co-routining may be implemented with production sytems [Rychner 1978].

The use of connectives and special symbols can make matching become arbitrarily complex. Rules might be interpreted as allowing all partial matches in their antecedent clauses [Bajcsy and Joshi 1978]. Thus

$$(A\ B\ C) \rightarrow (D)$$

is interpreted as

$$(ABC) \vee (BC) \vee (AB) \vee (AC) \vee (A) \vee (B) \vee (C) \rightarrow (D)$$

where the leftmost actual match is used to compare the rule to others in the case of conflicts.

The problem of large data bases is usually overcome by structuring them in some way so that the interpreter applies the rules only to a subset of the data base or uses a subset of the rules. This structuring undermines a basic principle of pure rule-based systems: Control should be dependent on the contents of the data base alone. Nevertheless, many systems divide the data base into two parts: an active smaller part which functions like the original data base but is restricted in size, and a larger data base which is inaccessible to the rule set in the active smaller part. "Meta-rules" have actions that move situation-action rules and facts from the smaller data base to the larger one and vice versa. The incoming set of rules and facts is presumably that which is applicable in the context indicated by the situation triggering the meta-rule. This two-level organization of rules is used in "blackboard" systems, such as Hearsay for speech-understanding [Erman and Lesser 1975]. The meta-rules seem to capture our idea of "mental set," or "context," or "frame" (Section 10.3.1, [Minsky 1975]). The two data bases are sometimes referred to as short-term memory and long term memory, in analogy with certain models of human memory.

12.3.3 An Example

We shall follow the actions of a production system for vision [Sloan 1977; Sloan and Bajcsy 1979]. The intent here is to avoid a description of all the details (which may be found in the References) and concentrate on the performance of the system as reflected by a sample of its output. The program uses a production system architecture in the domain of outdoor scenes. The goal is to determine basic features of the scene, particularly the separation between sky and ground. The interpreter is termed the "observer" and the memory has a two-tiered structure: (1) short term memory (STM) and (2) long term memory (LTM), a data base of all facts ever known or established, structured to prefer access to the most recently used facts. The image to be analyzed is shown in Fig. 12.4, and the action may be followed in Fig. 12.5. The analysis starts with the initialization command

*(look 100000 100 nil)

This command directs the Observer to investigate all regions that fall in the size range 100 to 100000, in decreasing order of size. The LTM is initialized to NIL.

our first look at (region 11)

x	*y*	*r-g*	*y-b*	*w-b*	size	top	bottom	left	right
35	2	24	29	6	2132	35	97	2	127

This report is produced by an image-processing procedure that produces assertions about (region 11). This region is shown highlighted in Fig. 12.5c.

_____ Progress Report _____

regions on this branch:
(11)
context stack:

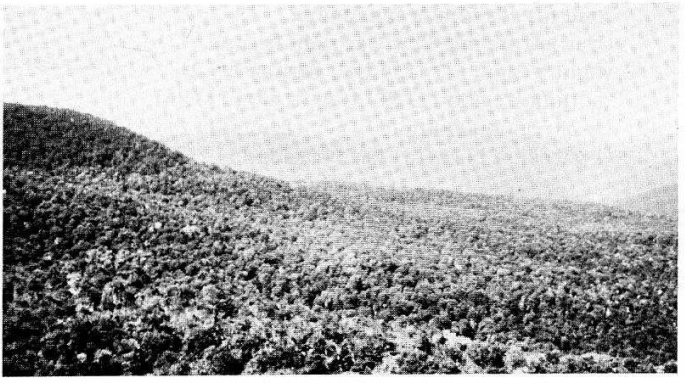


Fig. 12.4 Outdoor scene to be analyzed with production system.

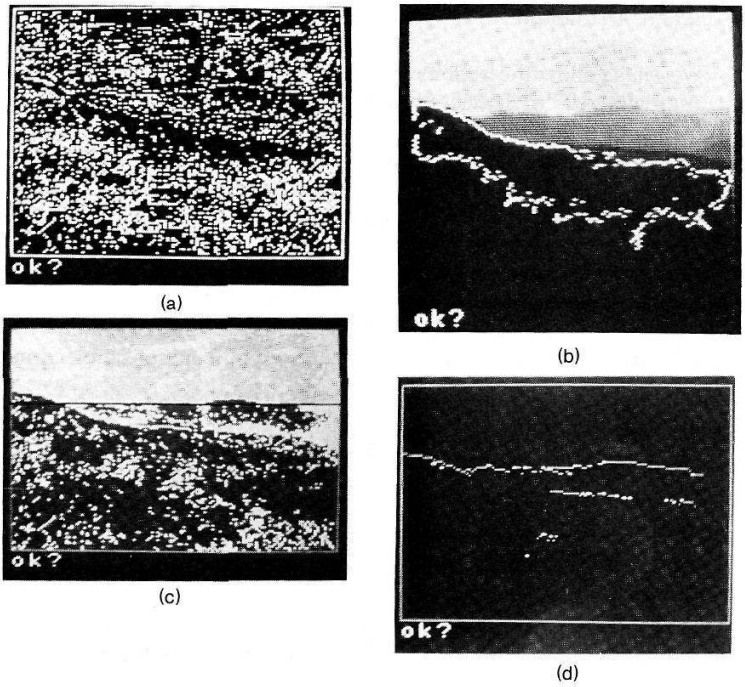


Fig. 12.5 Images corresponding to steps in production system analysis. (a) Texture in the scene. (b) Region 11 outlined. (c) Sky-Ground separation. (d) Skyline.

nil
contents of short term memory:
((far-left (region 11)) (far-right (region 11))
(right (region 11) 127) (left (region 11) 2)
(bottom (region 11) 97) (top (region 11) 35)
(*w-b* (region 11) minus) (*y-b* (region 11) zero)
(*r-g* (region 11) zero) (size (region 11) 2132))

———— end of progress report ————

Note that gray-level information is represented as a vector in opponent color space (Chapter 2), where the components axes are WHITE-BLACK (*w-b*), RED-GREEN (*r-g*), and YELLOW-BLUE (*y-b*). Three values (plus, zero, minus) are used for each component. The display above is generated once after every iteration of the Observer. The report shows that (REGION 11) is being investigated; there is no known context for this investigation; the information about (REGION 11) created by the image-processing apparatus has been placed in STM. The context stack is for information only, and shows a trace of activated sets of rules.

i think that (far-left (region 11))
i think that (far-right (region 11))
i think that (right (region 11) 127)
i think that (left (region 11) 2)
i think that (bottom (region 11) 97)
i think that (top (region 11) 35)
i think that (size (region 11) 2132)

This portion of the trace shows assertions moving from STM to LTM. They are reported because this is the first time they have been REMEMBERed (a special procedure in the Observer).

_____ Progress Report _____

regions on this branch:
(11)
context stack:
nil
contents of short term memory:
((color (region 11) black))

_____ end of progress report _____

The assertions created from the region data structure have been digested, and lead only to the conclusion that (REGION 11) is BLACK, based on a production that looks like:

(w-b (region x) minus) $\wedge$ (r-w (region x) zero)
$\wedge$ (b-w (region x) zero) $\rightarrow$ (color (region x) black)

_____ Progress Report _____

regions on this branch:
(11)
context stack:
nil
contents of short term memory:
((ground (region 11)) (shadow (region 11)))

_____ end of progress report _____

The observer knows that things that are black are GROUND and SHADOW. The facts it deduces about region 11 are again stored in the LTM.

Having discovered a piece of ground, the Observer has activated the GROUND-RULES, and changed context. It now investigates the neighbors of (REGION 11).

our first look at (region 16)

x	y	r-g	y-b	w-b	size	top	bottom	left	right
58	2	23	30	3	1833	57	119	2	127

(REGION 16) is a neighbor of (REGION 11), and the observer is trying to determine whether or not they are sufficiently similar, in both color and texture, to justify merging them.

_____ Progress Report _____

regions on this branch:
(16 11)
context stack:
(ground)
contents of short term memory:
((texture-difference (region 16) (region 11)))
(color-similar (region 16) (region 11))
(distance (region 16) near) (ground (region 16))
(color (region 16 black))

_____ end of progress report _____

The Observer decides that (REGION 16) is ground because it is at the bottom of the picture.

The ground-growing process continues, until finally one of the neighbors of a ground region is a piece of sky. The Observer will not immediately recognize this region as sky, but will see that a depth discontinuity exists and that the border between these two regions represents a section of three dimensional skyline.

our first look at (region 8)

x	*y*	*r-g*	*y-b*	*w-b*	size	top	bottom	left	right
27	2	13	13	33	394	15	38	2	57

_____ Progress Report _____

regions on this branch:
(8 13 16 11)
context stack:
(ground ground ground)
contents of short term memory:
((new-neighbor (region 800) (far-left (region 8))
(right (region 8) 57) (left (region 8) 2) (bottom (region 8) 38)
(top (region 8) 15) (*w-b* (region 8) zero) (*y-b* (region 8) minus)
(*r-g* (region 8) minus) (size (region 8) 394))

_____ end of progress report _____

texture descriptors for (region 8) are (54 50)
texture descriptors for (region 13) are (44 51)

Texture measurement is appropriate in the context of ground areas.

_____ Progress Report _____

regions on this branch:
(8 13 16 11)
context stack:
(ground ground ground)
contents of short term memory:
((texture-similar (region 8) (region 13)) (color-difference
(region 8) (region 13)) (color (region 8) blue-green))

_____ end of progress report _____

(REGION 8) passes the texture similarity test, but fails the color match.

_____ Progress Report _____

regions on this branch:
(8 13 16 11)
context stack:
(ground ground ground)
contents of short term memory:
((darker (region 13) (region 8)) (brighter (region 8) (region
13))
(yellower (region 13) (region 8)) (bluer (region 8) (region 13))
(redder (region 13) 13)
(below (region 13) (region 8)) (above (region 8) (region 13)))

_____ end of progress report _____

checking the border between (region 13) and (region 8)

_____ Progress Report _____

regions on this branch:
(8 13 16 11)
context stack:
(skyline ground ground ground)
contents of short term memory:
((segments built) (skyline-segment ((117 42)) (region 13)
(region 8)) (skyline-segment ((14 40) (13 40)) (region 13)
(region 8)))

_____ end of progress report _____

_____ Progress Report _____

regions on this branch:
(8 13 16 11)
context stack:
(skyline ground ground ground)

contents of short term memory:
((peak (14 40)) (peak (17 42)))

_____ end of progress report _____

Two local maxima have been discovered in the skyline. On the basis of a depth judgment, these peaks are correctly identified as treetops.

The analysis continues until all the major regions have been analyzed. The sky-ground separation is shown in Fig. 12.5a and skyline in Fig. 12.5e.

In most cases, complete analysis of the image follows from the context established by the first (largest) region. This implies that initial scanning of such scenes can be quite coarse, and very simple ideas about gross context are enough to get started. Once started, inferences about local surroundings lead the Observer's attention over the entire scene, often returning many times to the same part of the image, each time with a bit more knowledge.

12.3.4 Production System Pros and Cons

In their pure form, the productions of production systems are completely "modular," and are themselves independent of the control process. The data base of facts, or situations, is unordered set accessed in undetermined order to find one matching some rule. The rule is applied, and the system reports the search for a matching situation and situation-action pair (rule). This completely unstructured organization of knowledge could be a model for the human learning of "facts" which become available for use by some associative mechanism that finds relevant facts in our memories. The hope for pure production systems is that performance will degrade noncatastrophically from the deletion of rules or facts, and that the rules can interact in synergistic and surprising ways. A learning curve may be simulated by the addition of productions. Thus one is encouraged to experiment with how knowledge may best be broken up into disjoint fragments that interact to produce intelligent behavior.

Together with the modularity of productions in a simple system, there is a corresponding simplicity in the overall control program. The pure controller simply looks at the data base and somehow finds a matching situation (left-hand side) among the productions, applies the rule, and cycles. This simple structure remains constant no matter how the rules change, so any nondeterminism in the performance arises from the matcher, which may find different left hand side matches for sets of assertions in the data base.

The productions usually have a syntax that is machine-readable. Their semantics is similarly constrained, and so it begins to seem hopeful that a program (perhaps fired up by a production) could reason about the rules themselves, add them, modify them, or delete them. This is in contrast to the situation with *procedurally embedded knowledge* (Section 10.1.3), because it is difficult or impossible for programs to answer general questions about other programs. Thus the claim is that a production system can more easily reason about itself than can many other knowledge representation systems.

Productions often interact in ways that are not foreseen. This can be an advantage or a drawback, depending on the behavior desired. The pattern-matching control structure allows knowledge to be used whenever it is relevant, not only when the original designer thought that it might be. Symbiotic interaction of knowledge may also produce unforeseen insights. Production systems are a primary tool of *knowledge engineering*, an enterprise that attempts to encode and use expert knowledge at such tasks as medical diagnosis and interpretation of mass spectrograms [Lindsay et al. 1980; Buchanan and Mitchell 1978; Buchanan and Feigenbaum 1978; Shortliffe 1976; Aikins 1980].

There are many who are not convinced that production systems really offer the advantages they initially seem to. They use the following sorts of arguments.

The pure form of production system is almost never seen doing anything useful. In particular, the production system is most naturally a forward-chaining inference system, and one must exercise restraints and guidelines on it to keep it from running away and deducing lots of irrelevant facts instead of doing useful work. Of course, production systems may be written to do backward chaining by hypothesizing a RHS and seeing which LHS must be true for the desired RHS to occur (the process may be iterated to any depth). In practical systems based on production systems, there is implicit or explicit ordering of production rules so the matcher tries them in some order. Often the ordering is determined in a rather complex and dynamic manner, with groups of related rules being more likely to be applied together, the most recently used rule not allowed to be reapplied immediately, and so on. In fact, many production systems's controllers have all the control structure tricks mentioned above (and more) built into them; the simple and elegant "bag of rules" ideal is inadequate for realistic examples. When the rules are explicitly written with an idiosyncratic control structure in mind, the system can become unprincipled and inexplicable.

On the same lines, notice how difficult it is to specify a time-ordered sequence of actions by a completely modular set of rewriting rules. It is unnatural to force knowledge about processes that may contain iteration, tests, and recursion into the form of independent situation–action rules. A view that is more easily defensible is that knowledge about procedures for perception should be encoded as (embedded in) computer procedures, not assertions or rules. The causal chain that dictates that some actions are best performed before others is implicit in the sequential execution of procedures, and the language constraints, such as iterate and test, test and branch, or subroutine invocation, are all fairly natural ways to think about solving certain problems. Production systems can in fact be made to perform all these procedural-like functions, but only through an abrogation of the ideal of modular, unordered, matching-oriented rule invocation which is the production system ideal. The question turns into one of aesthetics; how to use productions in a good style, and to work with their philosophy instead of against it.

To summarize the previous two objections: Production-based knowledge systems may in practice be no more robust, easily modified, modular, extensible, understandable, or self-understanding than any other (say, procedural) system unless great care is taken. After a certain level of complexity is reached, they are

likely to be as opaque as any other scheme because of the control-structuring methods that must be imposed on the pure production system form.

12.4 SCENE LABELING AND CONSTRAINT RELAXATION

The general computational problem of assigning labels consistently to objects is sometimes called the "labeling problem," and arises in many contexts, such as graph and automata homomorphism, graph coloring, Latin square generation, and of course, image understanding [Davis and Rosenfeld 1976; Zucker 1976; Haralick and Shapiro 1979]. "Relaxation labeling," "constraint satisfaction," and "cooperative algorithms" are natural implementations for labeling, and their potential parallelism has been a very influential development in computer vision. As should any important development, the relaxation paradigm has had an impact on the conceptualization as well as on the implementation of processes.

Cooperating algorithms to solve the labeling problem are useful in low level vision (e.g., line finding, stereopsis) and in intermediate-level vision (e.g., line-labeling, semantics-based region growing). They may also be useful for the highest-level vision programs, those that maintain a consistent set of beliefs about the world to guide the vision process.

Section 12.4.1 presents the main concepts in the labeling problem. Section 12.4.2 outlines some basic forms that "discrete labeling" algorithms can take. Section 12.4.3 introduces a continuing example, that of labeling lines in a line drawing, and gives a mathematically well-behaved probabilistic "linear operator" labeling method. Section 12.4.4 modifies the linear operator to be more in accord with our intuitions, and Section 12.4.5 describes relaxation as linear programming and optimization, thereby gaining additional mathematical rigor.

12.4.1 Consistent and Optimal Labelings

All labeling problems have the following notions.

1. A set of *objects*. In vision, the objects usually correspond to entities to be labeled, or assigned a "meaning."
2. A finite set of *relations* between objects. These are the sorts of relations we saw in Chapter 10; in vision, they are often geometric or topological relations between segments in a segmented image. Properties of objects are simply unary relations. An input scene is thus a relational structure.
3. A finite set of *labels*, or symbols associated with the "meanings" mentioned above. In the simplest case, each object is to be assigned a single label. A *labeling* assigns one or more labels to (a subset of) the objects in a relational structure. Labels may be weighted with "probabilities"; a (label, weight) pair can indicate something like the "probability of an object having that label."
4. *Constraints*, which determine what labels may be assigned to an object and what sets of labels may be assigned to objects in a relational structure.

A basic labeling problem is then: Given a finite input scene (relational structure of objects), a set of labels, and a set of constraints, find a "consistent labeling." That is, assign labels to objects without violating the constraints. We saw this problem in Chapter 11, where it appeared as a matching problem. Here we shall start with the discrete labeling of Chapter 11 and proceed to more general labeling schemes.

As a simple example, consider the indoor scene of Fig. 12.6. The segmented office scene is to have its regions labeled as Door, Wall, Ceiling, Floor, and Bin, with the obvious interpretation of the labels. Here are some possible constraints, informally stated. Note that these particular constraints are in terms of the input relational structure, not the world from which the structure arose. A more complex (but reasonable) situation arises if scene constraints must be derived from rules about the three dimensional domain of the scene and the imaging process. Unary constraints use object properties to constrain labels; n-ary constraints force sets of label assignments to be compatible.

Unary constraints

1. The Ceiling is the single highest region in the image.
2. The Floor must be checkered.

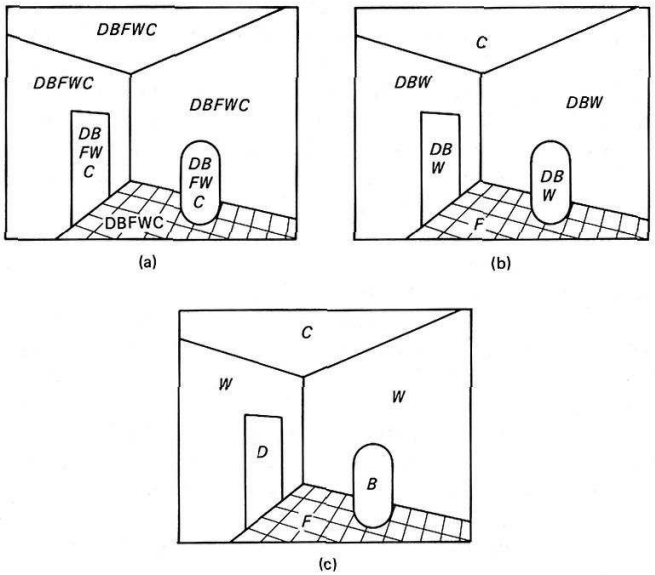


Fig. 12.6 A stylized "segmented office scene." The regions are the objects to be assigned labels D, B, F, W, C (Door, Bin, Floor, Wall, Ceiling). In (a), each object is assigned all labels. In (b) unary constraints have been applied (see text). In (c), relational constraints have been applied, and a unique label for each region results.

N-ary constraints

3. A Wall is adjacent to the Floor and Ceiling.
4. A Door is adjacent to the Floor and a Wall.
5. A Bin is adjacent to a Floor.
6. A Bin is smaller than a Door.

Obviously, there are many constraints on the appearance of segments in such a scene; which ones to use depends on the available sensors, the ease of computation of the relations and their power in constraining the labeling. Here the application of the constraints (Fig. 12.6) results in a unique labeling. Although the constraints of this example are purely for illustration, a system that actually performs such labeling on real office scenes is described in [Barrow and Tenenbaum 1976].

Labelings may be characterized as *inconsistent* or *consistent.* A weaker notion is that of an *optimal* labeling. Each of these adjectives reflects a formalizable property of the labeling of a relational structure and the set of constraints. If the constraints admit of only completely compatible or absolutely incompatible labels, then a labeling is consistent if and only if all its labels are mutually compatible, and inconsistent otherwise. One example is the line labels of Section 9.5; line drawings that could not be consistently labeled were declared "impossible." Such a black-and-white view of the scene interpretation problem is convenient and neat, but it is sometimes unrealistic. Recall that one of the problems with the line-labeling approach of Chapter 9 is that it does not cope gracefully with missing lines; strictly, missing lines often mean "impossible" line drawings. Such an uncompromising stance can be modified by introducing constraints that allow more degrees of compatibility than two (wholly compatible or strictly incompatible). Once this is done, both consistent and inconsistent labelings may be ranked on compatibility and likelihood. It is possible that a formally inconsistent labeling may rank better than a consistent but unlikely labeling.

Some examples are shown in Fig. 12.7. In 12.7b, the "inconsistent" labels are not nonsensical, but can only arise from (a very unlikely) accidental alignment of convex edges with three of the six vertices of a hexagonal hole in an occluding surface. The vertices that arise are not all included in the traditional catalog of legal vertices, hence the "inconsistent" labeling. The "floating cube" interpretation is consistent, but the "sitting cube" interpretation may be more likely if support and gravity are important concepts in the system. In Fig.12.7c, the scene with a missing line cannot be consistent according to the traditional vertex catalog, but the "inconsistent" labels shown are still the most likely ones. Labelings are only "consistent," "inconsistent," or "optimal" with respect to a given relational structure of objects (an input scene) and a set of constraints. These examples are meant to be illustrative only.

12.4.2 Discrete Labeling Algorithms

Let us consider the problem of finding a consistent set of labels, taken from a discrete finite set. This problem may be placed in an abstract algebraic context [Haralick and Kartus 1978; Haralick 1978; Haralick et al. 1978]. Perhaps the sim-

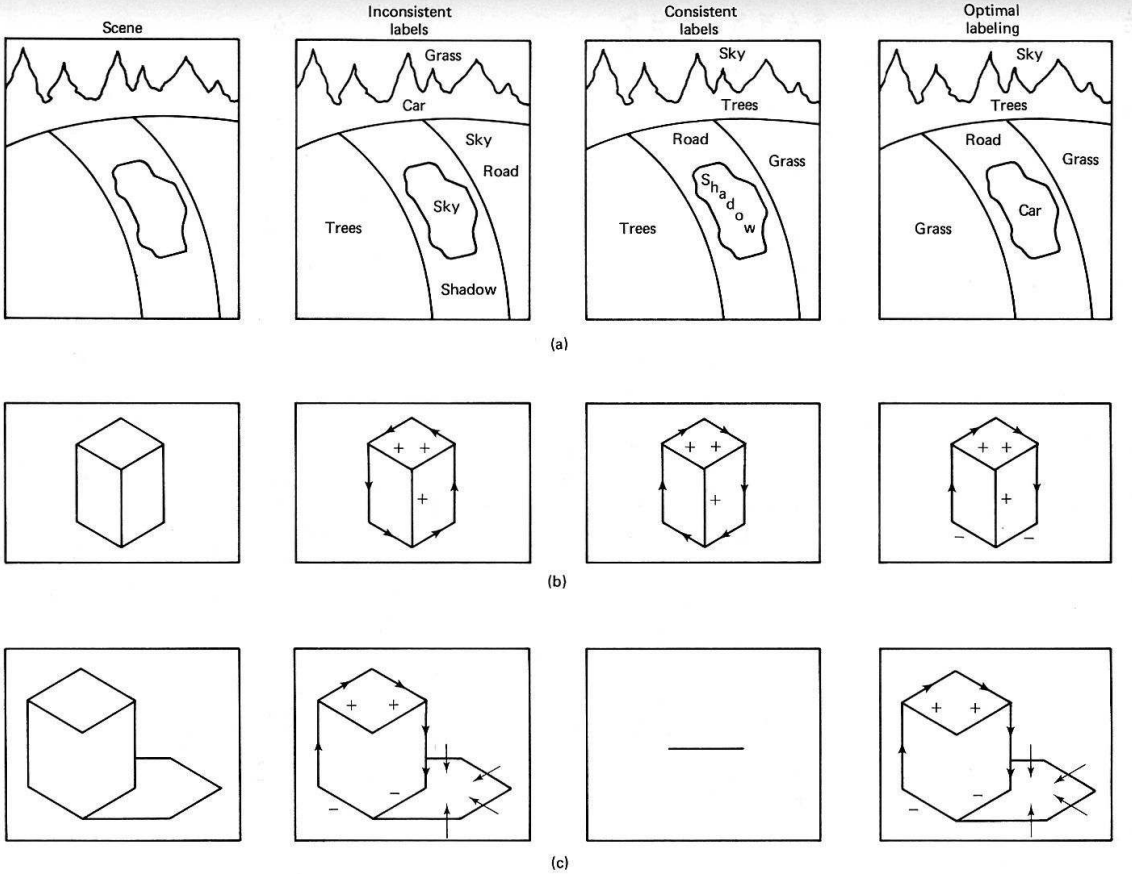


Fig. 12.7 Three scenes (A, B, C) and their labelings. Labelings are only "consistent," "inconsistent," or "optimal" with respect to a given relational structure of objects (an input scene) and a set of constraints. These examples are meant to be illustrative only.

plest way to find a consistent labeling of a relational structure (we shall often say "labeling of a scene") is to apply a depth-first *tree search* of the labeling possibilities, as in the backtracking algorithm (11.1).

Label an object in accordance with unary constraints.

Iterate until a globally consistent labeling is found:

Given the current labeling, label another object consistently—in accordance with all constraints.

If the object cannot be labeled consistently, backtrack and pick a new label for a previously labeled object.

This labeling algorithm can be computationally inefficient. First, it does not prune the search tree very effectively. Second, if it is used to generate all consistent labelings, it does not recognize important independences in the labels. That is, it does not notice that conclusions reached (labels assigned) in part of the tree search are usable in other parts without recomputation.

In a *serial relaxation*, the labels are changed one object at a time. After each such change, the new labeling is used to determine which object to process next. This technique has proved useful in some applications [Feldman and Yakimovsky 1974].

Assign all possible labels to each object in accordance with unary constraints.

Iterate until a globally consistent labeling is found:

Somehow pick an object to be processed.

Modify its labels to be consistent with the current labeling.

A *parallel iterative* algorithm adjusts all object labels at once; we have seen this approach in several places, notably in the "Waltz filtering algorithm" of Section 9.5.

Assign all possible labels to each object in accordance with unary constraints.

Iterate until a globally consistent labeling is found:

In parallel, eliminate from each object's label set those labels that are inconsistent with the current labels of the rest of the relational structure.

A less structured version of relaxation occurs when the iteration is replaced with an *asynchronous interaction* of labeled objects. Such interaction may be implemented with multiple cooperating processes or in a data base with "demons" (Ap-

pendix 2). This method of relaxation was used in MSYS [Barrow and Tenenbaum 1976]. Here imagine that each object is an active process that knows its own label set and also knows about the constraints, so that it knows about its relations with other objects. The program of each object might look like this.

> If I have just been activated, and my label set is not consistent with the labels of other objects in the relational structure, then I change my label set to be consistent, else I suspend myself.
>
> Whenever I change my label set, I activate other objects whose label set may be affected, then I suspend myself.

To use such a set of active objects, one can give each one all possible labels consistent with the unary constraints, establish the constraints so that the objects know where and when to pass on activity, and activate all objects.

Constraints involving arbitrarily many objects (i.e., constraints of arbitrarily high *order*) can efficiently be relaxed by recording acceptable labelings in a graph structure [Freuder 1978]. Each object to be labeled initially corresponds to a node in the graph, which contains all legal labels according to unary constraints. Higher order constraints involving more and more nodes are incorporated successively as new nodes in the graph. At each step the new node constraint is *propagated*; that is, the graph is checked to see if it is consistent with the new constraint. With the introduction of more constraints, node pairings that were previously consistent may be found to be inconsistent. As an example consider the following graph coloring problem: color the graph in Fig. 12.8 so that neighboring nodes have different colors. It is solved by building constraints of increasingly higher order and propagating them. The node constraints are given explicitly as shown in Fig. 12.8a, but the higher-order constraints are given in functional implicit form; prospective colorings must be tested to see if they satisfy the constraints. After the node constraints are given, order two constraints are synthesized as follows: (1) make a node for each node pairing; (2) add all labelings that satisfy the constraint. The result is shown in Fig. 12.8b. The single constraint of order three is synthesized in the same way, but now the graph is inconsistent: the match "Y,Z: Red,Green" is ruled out by the third order legal label set (RGY,GRY). To restore consistency the constraint is propagated through node (Y,Z) by deleting the inconsistent labelings. This means that the node constraint for node Z is now inconsistent. To remedy this, the constraint is propagated again by deleting the inconsistency, in this case the labeling $(Z{:}G)$. The change is propagated to node (X,Z) by deleting $(X,Z$: Red,Green) and finally the network is consistent.

In this example constraint propagation did not occur until constraints of order three were considered. Normally, some constraint propagation occurs after every order greater than one. Of course it may be impossible to find a consistent graph. This is the case when the labels for node Z in our example are changed from (G,Y) to (G,R). Inconsistency is then discovered at order three.

It is quite possible that a discrete labeling algorithm will not yield a unique label for each object. In this case, a consistent labeling exists using each label for the

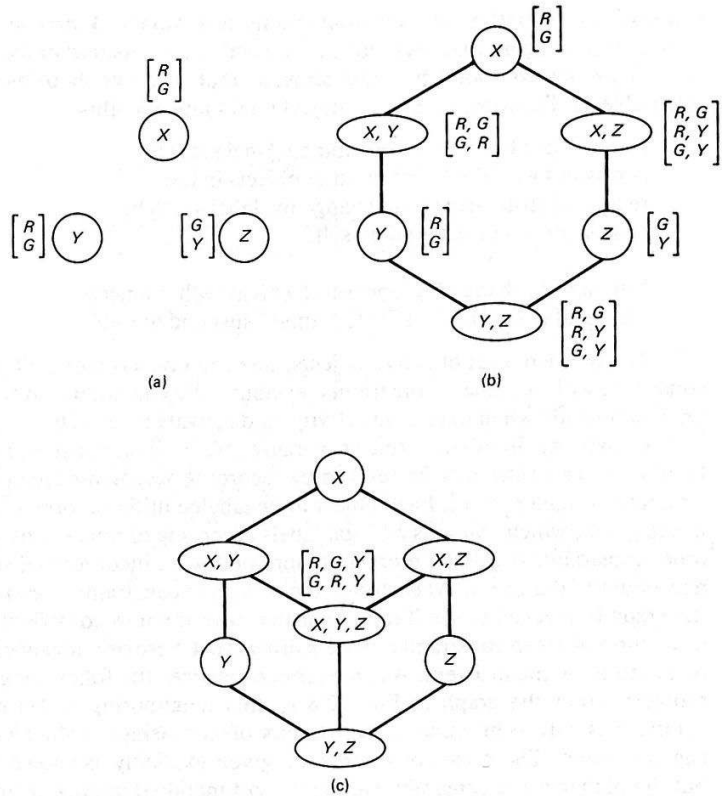


Fig. 12.8 Coloring a graph by building constraints of increasingly higher order.

object. However, which of an object's multiple labels goes with which of another object's multiple labels is not determined. The final enumeration of consistent labelings usually proceeds by tree search over the reduced set of possibilities remaining after the relaxation.

Convergence properties of relaxation algorithms are important; convergence means that in some finite time the labeling will "settle down" to a final value. In discrete labeling, constraints may often be written so that the label adjustment phase always reduces the number of labels for an object (inconsistent ones are eliminated). In this case the algorithm clearly must converge in finite time to a consistent labeling, since for each object the label set must either shrink or stay stable. In schemes where labels are added, or where labels have complex structure (such as real number "weights" or "probabilities"), convergence is often not guaranteed mathematically, though such schemes may still be quite useful. Some probabilistic labeling schemes (Section 12.4.3) have provably good convergence properties.

It is possible to use relaxation schemes without really considering their mathematical convergence properties, their semantics (What is the semantics of weights attached to labels—are they probabilities?), or a clear definition of what exactly the relaxation is to achieve (What is a good set of labels?). The fact that some schemes can be shown to have unpleasant properties (such as assigning nonzero weights to each of two inconsistent hypotheses, or not always converging to a solution), does not mean that they cannot be used. It only means that their behavior is not formally characterizable or possibly even predictable. As relaxation computations become more common, the less formalizable, less predictable, and less conceptually elegant forms of relaxation computations will be replaced by better behaved, more thoroughly understood schemes.

12.4.3 A Linear Relaxation Operator and a Line Labeling Example

The Formulation

We now move away from discrete labeling and into the realm of continuous *weights* or *supposition values* on labels. In Sections 12.4.3 and 12.4.4 we follow closely the development of [Rosenfeld et al. 1976]. Let us require that the sum of label weights for each object be constrained to sum to unity. Then the weights are reminiscent of probabilities, reflecting the "probability that the label is correct." When the labeling algorithm converges, a label emerges with a high weight if it occurs in a probable labeling of the scene. Weights, or supposition values, are in fact hard to interpret consistently as probabilities, but they are suggestive of likelihoods and often can be manipulated like them.

In what follows p refers to probability-like weights (supposition values) rather than to the value of a probability density function. Let a relational structure with n objects be given by a_i, $i = 1, \ldots, n$, each with m discrete labels $\lambda_1, \ldots, \lambda_m$. The shorthand $p_i(\lambda)$ denotes the weight, or (with the above caveats) the "probability" that the label λ (actually λ_k for some k) is correct for the object a_i. Then the probability axioms lead to the following constraints,

$$0 \leqslant p_i(\lambda) \leqslant 1 \tag{12.14}$$

$$\sum_{\lambda} p_i(\lambda) = 1 \tag{12.15}$$

The labeling process starts with an initial assignment of weights to all labels for all objects [consistent with Eqs. (12.14) and (12.15)]. The algorithm is parallel iterative: It transforms all weights at once into a new set conforming to Eqs. (12.14) and (12.15), and repeats this transformation until the weights converge to stable values.

Consider the transformation as the application of an operator to a vector of label weights. This operator is based on the *compatibilities* of labels, which serve as constraints in this labeling algorithm. A compatibility p_{ij} looks like a conditional probability.

$$\sum_{\lambda} p_{ij}(\lambda | \lambda') = 1 \qquad \text{for all } i, j, \lambda' \tag{12.16}$$

$$p_{ii}(\lambda|\lambda') = 1 \qquad iff\ \lambda = \lambda', \quad \text{else } 0. \tag{12.17}$$

The $p_{ij}(\lambda|\lambda')$ may be interpreted as the conditional probability that object a_i has label λ given that another object a_j has label λ'. These compatibilities may be gathered from statistics over a domain, or may reflect a priori belief or information.

The operator iteratively adjusts label weights in accordance with other weights and the compatibilities. A new weight $p_i(\lambda)$ is computed from old weights and compatibilities as follows.

$$p_i(\lambda) := \sum_j c_{ij} \{ \sum_{\lambda'} p_{ij}(\lambda|\lambda') p_j(\lambda') \} \tag{12.18}$$

The c_{ij} are coefficients such that

$$\sum_j c_{ij} = 1 \tag{12.19}$$

In Eq. (12.18), the inner sum is the expectation that object a_i has label λ, given the evidence provided by object a_j. $p_i(\lambda)$ is thus a weighted sum of these expectations, and the c_{ij} are the weights for the sum.

To run the algorithm, simply pick the p_{ij} and c_{ij}, and apply Eq. (12.18) repeatedly to the p_i until they stop changing. Equation (12.18) is in the form of a matrix multiplication on the vector of weights, as shown below; the matrix elements are weighted compatibilities, the $c_{ij}p_{ij}$. The relaxation operator is thus a matrix; if it is partitioned into several *component* matrices, one for each set of non-interacting weights, linear algebra yields proofs of convergence properties [Rosenfeld et al. 1976]. The iteration for the reduced matrix for each component does converge, and converges to the weight vector that is the eigenvector of the matrix with eigenvalue unity. This final weight vector is independent of the initial assignments of label weights; we shall say more about this later.

An Example

Let us consider the input line drawing scene of Fig. 12.9a used in [Rosenfeld et.al. 1976]. The line labels given in Section 9.5 allow several consistent labels as shown in Fig. 12.9b-e, each with a different physical interpretation.

In the discrete labelling "filtering" algorithm presented in Section 9.5 and outlined in the preceding section, the relational structure is imposed by the neighbor relation between vertices induced by their sharing a line. Unary constraints are imposed through a catalog of legal combinations of line labels at vertices, and the binary constraint is that a line must not change its label between vertices. The algorithm eliminates inconsistent labels.

Let us try to label the sides of the triangle a_1, a_2, and a_3 in Fig. 12.9 with the solid object edge labels $\{>, <, +, -\}$. To do this requires some "conditional probabilities" for compatibilities $p_{ij}(\lambda|\lambda')$, so let us use those that arise if all eight interpretations of Fig. 12.9 are equally likely. Remembering that

$$p(X|Y) = \frac{p(X,Y)}{p(Y)} \tag{12.20}$$

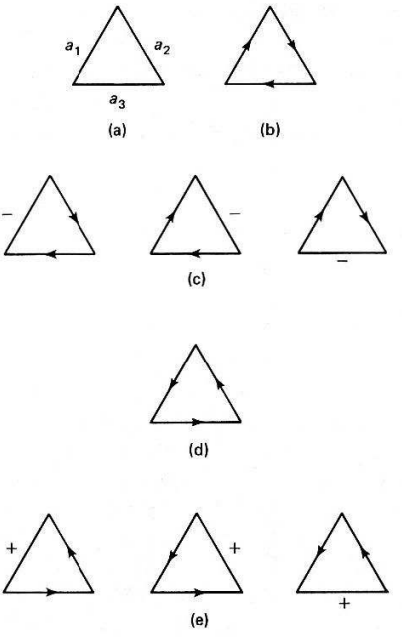


Fig. 12.9 A triangle and its possible labels. (a) Edge names. (b) Floating. (c) Flap folded up. (d) Triangular hole. (e) Flap folded down.

and taking $p(X, Y)$ to mean the probability that labels X and Y occur consecutively in clockwise order around the triangle, one can derive Table 12.2. Of course, we could choose other compatibilities based on any considerations whatever as long as Eqs. (12.16) and (12.17) are preserved.

Table 12.2 shows that there are two noninteracting components, $\{-,>\}$ and $\{+,<\}$. Consider the first component that consists of the weight vector

$$[p_1(>),\ p_1(-),\ p_2(>),\ p_2(-),\ p_3(>),\ p_3(-)] \tag{12.21}$$

The second is treated similarly. This vector describes weights for the subpopulation of labelings given by Fig. 12.9b and c. The matrix M of compatibilities has columns of weighted p_{ij}.

$$M = \begin{bmatrix} c_{11}p_{11}(>|>) & c_{21}p_{21}(>|>) & \cdots \\ c_{11}p_{11}(>|-) & c_{21}p_{21}(>|-) & \cdots \\ c_{12}p_{12}(>|>) & c_{22}p_{22}(>|>) & \cdots \\ c_{12}p_{12}(>|-) & c_{22}p_{22}(>|-) & \cdots \\ c_{13}p_{13}(>|>) & c_{23}p_{23}(>|>) & \cdots \\ c_{13}p_{13}(>|-) & c_{23}p_{23}(>|-) & \cdots \end{bmatrix} \tag{12.22}$$

Table 12.2

λ_1	λ_2	$p(\lambda_1, \lambda_2)$	$p(\lambda_1 \mid \lambda_2)$
>	>	1/4	2/3
>	−	1/8	1
−	>	1/8	1/3
−	−	0	0
>	<	0	0
>	+	0	0
−	<	0	0
−	+	0	0
<	>	0	0
+	>	0	0
<	−	0	0
+	−	0	0
<	<	1/4	2/3
<	+	1/8	1
+	<	1/8	1/3
+	+	0	0

If we let $c_{ij} = 1/3$ for all i, j, then

$$M = \frac{1}{3}\begin{bmatrix} 1 & 0 & 2/3 & 1/3 & 2/3 & 1/3 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 2/3 & 1/3 & 1 & 0 & 2/3 & 1/3 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 2/3 & 1/3 & 2/3 & 1/3 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{12.23}$$

An analytic eigenvector calculation (Appendix 1) shows that the M of Eq. (12.23) yields (for any initial weight vector) the final weight vector of

$$[3/4,\ 1/4,\ 3/4,\ 1/4,\ 3/4,\ 1/4] \tag{12.24}$$

Thus each line of the population in the component we chose (Fig. 12.9b and c) has label > with "probability" 3/4, —with "probability" 1/4. In other words, from an initial assumption that all labelings in Fig. 12.9b and c were equally likely, the system of constraints has "relaxed" to the state where the "most likely" labeling is that of Fig. 12.9b, the floating triangle.

This relaxation method is a crisp mathematical technique, but it has some drawbacks. It has good convergence properties, but it converges to a solution entirely determined by the compatibilities, leaving no room for preferences or local scene evidence to be incorporated and affect the final weights. Further, the algorithm perhaps does not exactly mirror the following intuitions about how relaxation should work.

1. Increase $p_i(\lambda)$ if high probability labels for other objects are compatible with assignment of λ to a_i.
2. Decrease $p_i(\lambda)$ if high probability labels are incompatible with the assignment of λ to a_i.
3. Labels with low probability, compatible or incompatible, should have little influence on $p_i(\lambda)$.

However, the operator of this section decreases $p_i(\lambda)$ the most when other labels have both low compatibility and low probability. Thus it accords with (1) above, but not with (2) or (3). Some of these difficulties are addressed in the next section.

12.4.4 A Nonlinear Operator

The Formulation

If compatibilities are allowed to take on both positive and negative values, then we can express strong incompatibility better and obtain behavior more like (1), (2), and (3) just above. Denote the compatibility of the event "label λ on a_i" with the event "label λ on a_j" by $r_{ij}(\lambda, \lambda')$. If the two events occur together often, r_{ij} should be positive. If they occur together rarely, r_{ij} should be negative. If they are independent, r_{ij} should be 0. The *correlation coefficient* behaves like this, and the compatibilities of this section are based on correlations (hence the the notation r_{ij} for compatibilities). The correlation is defined using the covariance.

$$\operatorname{cov}(X, Y) = p(X, Y) - p(X)p(Y)$$

Now define a quantity σ which is like the standard deviation

$$\sigma(X) = [p(X) - (p(X))^2]^{1/2} \tag{12.25}$$

then the correlation is the normalized covariance

$$\operatorname{cor}(X, Y) = \frac{\operatorname{cov}(X, Y)}{\sigma(X)\sigma(Y)} \tag{12.26}$$

This allows the formulation of an expression precisely analogous to Eq. (12.18), only that r_{ij} instead of p_{ij} is used to obtain a means of calculating the positive or negative change in weights.

$$q_i^{(k)}(\lambda) = \sum_j c_{ij} \left[\sum_{\lambda'} r_{ij}(\lambda, \lambda') p_j^{(k)}(\lambda')\right] \tag{12.27}$$

In Eqs. (12.27)–(12.29) the superscripts indicate iteration numbers. The weight change (Eq. 12.27) could be applied as follows,

$$p_i^{(k+1)}(\lambda) = p_i^{(k)}(\lambda) + q_i^{(k)}(\lambda) \tag{12.28}$$

but then the resultant label weights might not remain nonnegative. Fixing this in a straightforward way yields the iteration equation

$$p_i^{(k+1)}(\lambda) = \frac{p_i^{(k)}(\lambda)[1 + q_i^{(k)}(\lambda)]}{\sum_{\lambda} p_i^{(k)}(\lambda)[1 + q_i^{(k)}(\lambda)]} \tag{12.29}$$

The convergence properties of this operator seem to be unknown, and like the linear operator it can assign nonzero weights to maximally incompatible labelings. However, its behavior can accord with intuition, as the following example shows.

An Example

Computing the covariances and correlations for the set of labels of Fig. 12.9b-e yields Table 12.3.

Figure 12.10 shows the nonlinear operator of Eq. (12.29) operating on the example of Fig. 12.9. Figure 12.10 shows several cases.

1. Equal initial weights: convergence to apriori probabilities ($\frac{3}{8}, \frac{3}{8}, \frac{1}{8}, \frac{1}{8}$).
2. Equal weights in the component $\{>, -\}$: convergence to "most probable" floating triangle labeling.
3. Slight bias toward a flap labeling is not enough to overcome convergence to the "most probable" labeling, as in (2).
4. Like (3), but greater bias elicits the "improbable" labeling.
5. Contradicatory biases toward "improbable" labelings: convergence to "most probable" labeling instead.
6. Like (5), but stronger bias toward one "improbable" labeling elicits it.
7. Bias toward one of the components $\{>, -\}$, $\{<, +\}$ converges to most probable labeling in that component.
8. Like (7), only biased to less probable labelling in a component.

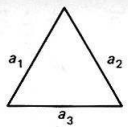
12.4.5 Relaxation as Linear Programming

The Idea

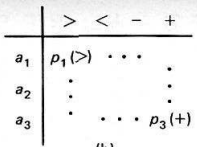
Linear programming (LP) provides some useful metaphors for thinking about relaxation computations, as well as actual algorithms and a rigorous basis [Hummel and Zucker 1980]. In this section we follow the development of [Hinton 1979].

Table 12.3

λ_1	λ_2	$\text{cov}(\lambda_1, \lambda_2)$	$\text{cor}(\lambda_1, \lambda_2)$
$>$	$>$	$\frac{7}{64}$	$\frac{7}{15}$
$>$	$-$	$\frac{5}{64}$	$5/\sqrt{105}$
$-$	$>$	$\frac{5}{64}$	$5/\sqrt{105}$
$-$	$-$	$-\frac{1}{64}$	$-\frac{1}{7}$
$>$	$<$	$-\frac{9}{64}$	$-\frac{3}{5}$
.	.	.	.
.	.	.	.
.	.	.	.



(a)



(b)

Case	Initial weights				After 2 to 3 iterations				After 20 to 30 iterations				Limit			
	0.25	0.25	0.25	0.25	0.3	0.3	0.2	0.2	0.33	0.33	0.17	0.17	0.37	0.37	0.13	0.13
(1)	0.25	0.25	0.25	0.25	0.3	0.3	0.2	0.2	0.33	0.33	0.17	0.17	0.37	0.37	0.13	0.13
	0.25	0.25	0.25	0.25	0.3	0.3	0.2	0.2	0.33	0.33	0.17	0.17	0.37	0.37	0.13	0.13
	0.5	0	0.5	0	0.8	0	0.2	0	0.98	0	0.2	0	1	0	0	0
(2)	0.5	0	0.5	0	0.8	0	0.2	0	0.98	0	0.2	0	1	0	0	0
	0.5	0	0.5	0	0.8	0	0.2	0	0.98	0	0.2	0	1	0	0	0
	0.5	0	0.5	0	0.62	0	0.37	0	1	0	0	0	1	0	0	0
(3)	0.4	0	0.6	0	0.49	0	0.51	0	0.97	0	0.03	0	1	0	0	0
	0.5	0	0.5	0	0.62	0	0.37	0	1	0	0	0	1	0	0	0
	0.5	0	0.5	0	0.64	0	0.36	0	1	0	0	0	1	0	0	0
(4)	0.3	0	0.7	0	0.36	0	0.64	0	0.07	0	0.93	0	0	0	1	0
	0.5	0	0.5	0	0.64	0	0.36	0	1	0	0	0	1	0	0	0
	0.3	0	0.7	0	0.5	0	0.5	0	0.95	0	0.05	0	1	0	0	0
(5)	0.3	0	0.7	0	0.5	0	0.5	0	0.95	0	0.05	0	1	0	0	0
	0.5	0	0.5	0	0.84	0	0.16	0	1	0	0	0	1	0	0	0
	0.2	0	0.8	0	0.3	0	0.7	0	0.06	0	0.94	0	0	0	1	0
(6)	0.3	0	0.7	0	0.51	0	0.49	0	1	0	0	0	1	0	0	0
	0.5	0	0.5	0	0.83	0	0.17	0	1	0	0	0	1	0	0	0
	0.3	0.2	0.3	0.2	0.41	0.13	0.32	0.14	0.98	0	0.02	0	1	0	0	0
(7)	0.3	0.2	0.3	0.2	0.41	0.13	0.32	0.14	0.98	0	0.02	0	1	0	0	0
	0.3	0.2	0.3	0.2	0.41	0.13	0.32	0.14	0.98	0	0.02	0	1	0	0	0
	0.3	0.2	0.3	0.2	0.38	0.17	0.29	0.16	1	0	0	0	1	0	0	0
(8)	0.25	0.25	0.25	0.25	0.35	0.20	0.25	0.20	1	0	0	0	1	0	0	0
	0.2	0.2	0.4	0.2	0.23	0.16	0.45	0.16	0.2	0	0.8	0	0	0	1	0

(c)

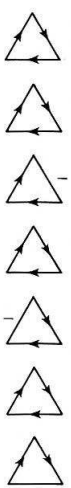


Fig. 12.10 The nonlinear operator produces labelings for the triangle in (a). (b) shows how the label weights are displayed, and (c) shows a number of cases (see text).

To put relaxation in terms of linear programming, we use the following translations.

- LABEL WEIGHT VECTORS $\Longrightarrow$ POINTS IN EUCLIDEAN N-SPACE. Each possible assignment of a label to an object is a *hypothesis*, to which a weight (supposition value) is to be attached. With N hypotheses, an N-vector of weights describes a labeling. We shall call this vector a (hypothesis or label) *weight vector.* For m labels and n objects, we need at most Euclidean nm-space.
- CONSTRAINTS $\Longrightarrow$ INEQUALITIES. *Constraints are mapped into linear inequalities* in hypothesis weights, by way of various identities like those of "fuzzy logic" [Zadeh 1965]. Each inequality determines an infinite half-space. The weight vectors within this half-space satisfy the constraint. Those outside do not. The convex solid that is the set intersection of all the half-spaces includes those weight vectors that satisfy all the constraints: each represents a "consistent" labeling. In linear programming terms, each such weight vector is a *feasible solution.* We thus have the usual geometric interpretation of the linear programming problem, which is to find the best (optimal) consistent (feasible) labeling (solution, or weight vector). Solutions should have *integer*-valued (1- or 0-valued) weights indicating convergence to actual labelings, not probabilistic ones such as those of Section 12.4.3, or the one shown in Fig. 12.10c, case 1.
- HYPOTHESIS PREFERENCES $\Longrightarrow$ PREFERENCE VECTOR. Often some hypotheses (label assignments) are preferred to others, on the basis of a priori knowledge, image evidence, and so on. To express this preference, make an N-dimensional *preference vector*, which expresses the relative importance (preference) of the hypotheses. Then
  - The *preference of a labeling* is the dot product of the preference vector and the weight vector (it is the sum for all hypotheses of the weight of each hypothesis times its preference).
  - The preference vector defines a *preference direction* in N-space. The optimal feasible solution is that one "farthest" in the preference direction. Let $\mathbf{x}$ and $\mathbf{y}$ be feasible solutions; they are N-dimensional weight vectors satisfying all constraints. If $\mathbf{z} = \mathbf{x} - \mathbf{y}$ has a component in the positive preference direction, then $\mathbf{x}$ is a better solution than $\mathbf{y}$, by the definition of the preference of a labeling.

  It is helpful for our intuition to let the preference direction define a "downward" direction in N-space as gravity does in our three-space. Then we wish to pick the lowest (most preferred) feasible solution vector.
- LABELING $\Longrightarrow$ OPTIMAL SOLUTION. The relaxation algorithm must solve the linear programming problem—find the best consistent labeling. Under the conditions we have outlined, the best solution vector occurs generally at a vertex of the N-space solid. This is so because usually a vertex will be the "lowest" part of the convex solid in the preference direction. It is a rare coincidence that the solid "rests on a face or edge," but when it does a whole edge or face of the solid contains equally preferred solutions (the preference direction is normal to

the edge or face). For integer solutions, the solid should be the convex hull of integer solutions and not have any vertices at noninteger supposition values.

The "simplex algorithm" is the best known solution method in linear programming. It proceeds from vertex to vertex, seeking the one that gives the optimal solution. The simplex algorithm is not suited to parallel computation, however, so here we describe another approach with the flavor of hill-climbing optimization. Basically, any such algorithm moves the weight vector around in N-space, iteratively adjusting weights. If they are adjusted one at a time, serial relaxation is taking place; if they are all adjusted at once, the relaxation is parallel iterative. The feasible solution solid and the preference vector define a "cost function" over all N-space, which acts like a potential function in physics. The algorithm tries to reach an optimum (minimum) value for this cost function. As with many optimization algorithms, we can think of the algorithm as trying to simulate (in N-space) a ball bearing (the weight vector) rolling along some path down to a point of minimum gravitational (cost) potential. Physics helps the ball bearing find the minimum; computer optimization techniques are sometimes less reliable.

Translating Constraints to Inequalities

The supposition values, or hypothesis weights, may be encoded into the interval [0, 1], with 0 meaning "false," 1 meaning "true." The extension of weights to the whole interval is reminiscent of "fuzzy logic," in which truth values may be continuous over some range [Zadeh 1965]. As in Section 12.4.3, we denote supposition values by $p(\cdot)$; H, A, B, and C are label assignment events, which may be considered as hypotheses that the labels are correctly assigned. $\tilde{}$, $\vee$, $\wedge$, $\Longrightarrow$ and $\Longleftrightarrow$ are the usual logical connectives relating hypotheses. The connectives allow the expression of complex constraints. For instance, a constraint might be "Label x as 'y' if and only if z is labeled 'w' or q is labelled 'v'." This constraint relates three hypotheses: h_1: (x is "y"), h_2: (z is "w"), h_3: (q is "v"). The constraint is then $h_1 \Longleftrightarrow (h_2 \vee h_3)$.

Inequalities may be derived from constraints this way.

1. *Negation.* $p(H) = 1 - p(\tilde{}(H))$.
2. *Disjunction.* The sums of weights of the disjunct are greater than or equal to one. $p(A \vee B \vee \ldots \vee C)$ gives the inequality $p(A) + p(B) + \ldots + p(C) \geqslant 1$.
3. *Conjunction.* These are simply separate inequalities, one per conjunct. In particular, a conjunction of disjunctions may be dealt with conjunct by conjunct, producing one disjunctive inequality per conjunct.
4. *Arbitrary expressions.* These must be put into conjunctive normal form (Chapter 10) by rewriting all connectives as $\wedge$'s and $\vee$'s. Then (3) applies.

As an example, consider the simple case of two hypotheses A and B, with the single constraint that $A \Longrightarrow B$. Applying rules 1 through 4 results in the following five inequalities in $p(A)$ and $p(B)$; the first four assure weights in [0, 1]. The fifth arises from the logical constraint, since $A \Longrightarrow B$ is the same as $B \vee \tilde{}(A)$.

$$0 \leqslant p(A)$$
$$p(A) \leqslant 1$$
$$0 \leqslant p(B)$$
$$p(B) \leqslant 1$$
$$p(B) + (1 - p(A)) \geqslant 1 \qquad \text{or} \qquad p(B) \geqslant p(A)$$

These inequalities are shown in Fig. 12.11. As expected from the $\Longrightarrow$ constraint, optimal feasible solutions exist at: (1,1) or (A,B); (0,1) or $(\tilde{}(A),B)$; (0,0) or $(\tilde{}(A), \tilde{}(B))$. Which of these is preferred depends on the preference vector. If both its components are positive, (A,B) is preferred. If both are negative, $(\tilde{}(A), \tilde{}(B))$ is preferred, and so on.

A Solution Method

Here we describe (in prose) a search algorithm that can find the optimal feasible solution to the linear programming problem as described above. The description makes use of the mechanical analogy of an N-dimensional solid of feasible solutions, oriented in N-space so that the preference vector induces a "downward" direction in space. The algorithm attempts to move the vector of hypothesis weights to the point in space representing the feasible solution of maximum preference. It should be clear that this is a point on the surface of the solid, and unless the preference vector is normal to a face or edge of the solid, the point is a unique "lowest" vertex.

To establish a potential that leads to feasible solutions, one needs a measure of the infeasibility of a weight vector for each constraint. Define the amount a vector violates a constraint to be zero if it is on the feasible side of the constraint hyperplane. Otherwise the violation is the normal distance of the vector to the hyperplane. If $\mathbf{h}_i$ is the coefficient vector of the i^{th} hyperplane (Appendix 1) and $\mathbf{w}$ the weight vector, this distance is

$$d_i = \mathbf{w} \cdot \mathbf{h}_i \tag{12.30}$$

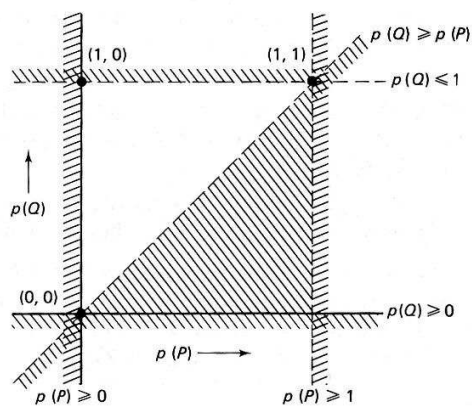


Fig. 12.11 The feasible region for two hypotheses A and B and the constraint A B. Optimal solutions may occur at the three vertices. The preferred vertex will be that one farthest in the direction of the preference vector, or lowest if the preference vector defines "down."

If we then define the infeasibility as

$$I = \sum_i \frac{d_i^2}{2} \tag{12.31}$$

then $\partial I/\partial d_i = d_i$ is the rate the infeasibility changes for changes in the violation. The force exerted by each constraint is proportional to the normal distance from the weight vector to the feasible region defined by that constraint, and tends to pull the weight vector onto the surface of the solid.

Now add a weak "gravity-like" force in the preference direction to make the weight vector drift to the optimal vertex. At this point an optimization program might perform as shown in Fig. 12.12.

Figure 12.12 illustrates a problem: The forces of preference and constraints will usually dictate a minimum potential outside the solid (in the preference direction). Fixes must be applied to force the weight vector back to the closest (presumably the optimum) vertex. One might round high weights to 1 and low ones to 0, or add another local force to draw vectors toward vertices.

Examples

An algorithm based on the principles outlined in the preceeding section was successfully used to label scenes of "puppets" such as Fig. 12.13 with body parts [Hinton 1979].

The discrete, consistency-oriented version of line labeling may be extended to incorporate the notion of optimal labelings. Such a system can cope with the explosive increase in consistent labelings that occurs if vertex labels are included for cases of missing lines, accidental alignment, or "two-dimensional" objects such as folded paper. It allows modeling of the fact that human beings do not "see" all possible interpretations of scenes with accidental alignments. If labelings are given

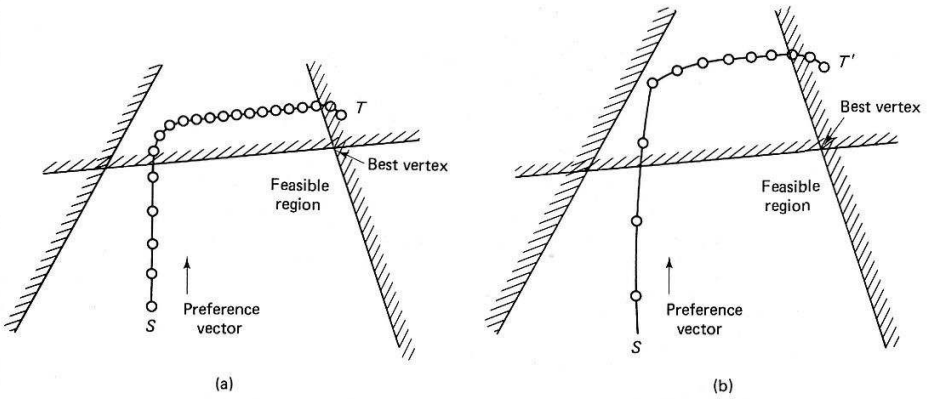
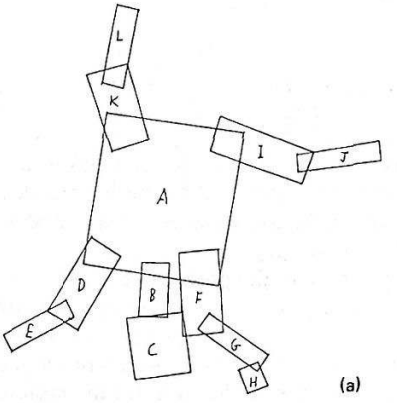


Fig. 12.12 In (a), the weight vector moves from S to rest at T, under the combined influence of the preferences and the violated constraints. In (b), convergence is speeded by making stronger preferences, but the equilibrium is farther away from the optimal vertex.



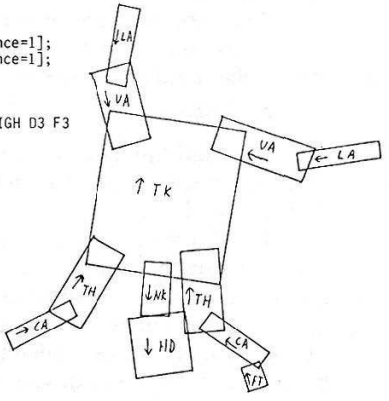

```
!.bestset;
A1  BOT TRUNK  NECK B1  UPPERARM D2 F2  THIGH 13 K2
B1  BOT NECK  HEAD C1  TRUNK A1
C1  BOT HEAD  NECK B1
D2  TOP UPPERARM  TRUNK A1  LOWERARM E4
E4  TOP LOWERARM  UPPERARM D2  HAND -
F2  TOP UPPERARM  TRUNK A1  LOWERARM G2
G2  TOP LOWERARM  UPPERARM F2  HAND H2
H2  TOP HAND  LOWERARM G2
I3  TOP THIGH  TRUNK A1  CALF J4
J4  BOT CALF  THIGH I3  FOOT -
K2  BOT THIGH  TRUNK A1  CALF L4
L4  BOT CALF  THIGH K2  FOOT -
```

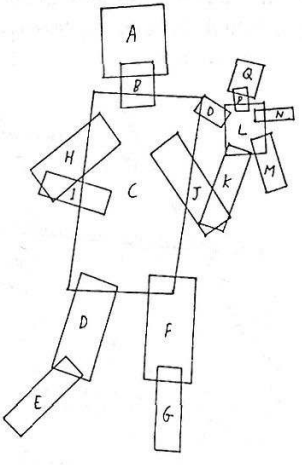
(a)

```
!trytointerpret [trunk as upright importance=1];
!trytointerpret [thigh as upright importance=1];

!.bestset;
A2  TOP TRUNK  NECK -  UPPERARM I2 K1  THIGH D3 F3
B1  BOT NECK  HEAD C1  TRUNK -
C1  BOT HEAD  NECK B1
D3  TOP THIGH  TRUNK A2  CALF E3
E3  TOP CALF  THIGH D3  FOOT -
F3  TOP THIGH  TRUNK A2  CALF E3
G3  TOP CALF  THIGH F3  FOOT H1
H1  TOP FOOT  CALF G3
I2  TOP UPPERARM  TRUNK A2  LOWERARM J3
J3  BOT LOWERARM  UPPERARM I2  HAND -
K1  BOT UPPERARM  TRUNK A2  LOWERARM L3
L3  BOT LOWERARM  UPPERARM K1  HAND -
```

(b)






```
!.bestset;
A1  TOP HEAD  NECK B1
B1  TOP NECK  HEAD A1  TRUNK C2
C2  TOP TRUNK  NECK B1  UPPERARM H1 J1  THIGH D3 F3
D3  TOP THIGH  TRUNK C2  CALF E3
E3  TOP CALF  THIGH D3  FOOT -
F3  TOP THIGH  TRUNK C2  CALF G3
G3  TOP CALF  THIGH F3  FOOT-
H1  TOP UPPERARM  TRUNK C2  LOWERARM I1
I1  TOP LOWERARM  UPPERARM H1  HAND -
J1  TOP LOWERARM  TRUNK C2  LOWERARM K4
K4  BOT LOWERARM  UPPERARM J1  HAND L6
L6  BOT HAND  LOWERARM K4
```

(c)

Fig. 12.13 Puppet scenes interpreted by linear programming relaxation. (a) shows an upside down puppet. (b) is the same input along with preferences to interpret the trunk and thighs as upright; these result in an interpretation with trunk and neck not connected. In (c), the program finds only the "best" puppet, since it was only expecting one.

costs, then one can include labels for missing lines and accidental alignment as high-cost labels, rendering them usable but undesirable. Also, in a scene-analysis system using real data, local evidence for edge appearance can enhance the a priori likelihood that a line should bear a particular label. If such preferences can be extracted along with the lines in a scene, the evidence can be used by the line labeling algorithm.

The inconsistency constraints for line labels may be formalized as follows. Each line and vertex has one label in a consistent labeling; thus for each line L and vertex J,

$$\sum_{\text{all line labels}} p(L \text{ has label LLABEL}) = 1 \tag{12.32}$$

$$\sum_{\text{all vertex labels}} p(J \text{ has label VLABEL}) = 1 \tag{12.33}$$

Of course, the VLABELS and LLABELS in the above constraints must be forced to be compatible (if L has LLABEL, JLABEL must agree with it). For a line L and a vertex J at its end,

$$p(L \text{ has LLABEL}) = \sum_{\substack{\text{all VLABELS} \\ \text{giving LLABEL to } L}} p(J \text{ has label VLABEL}) \tag{12.34}$$

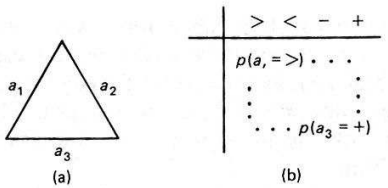
This constraint also enforces the coherence rule (a line may not change its label betwen vertices).

Using these constraints, linear programming relaxation labeled the triangle example of Fig. 12.7 as shown in Fig. 12.14, which shows three cases.

1. Preference 0.5 for each of the three junction label assignments (hypotheses) corresponding to the floating triangle, 0 preference for all other junction and line label hypotheses: converges to floating triangle.
2. Like (1), but with equal preferences given to the junction labels for the triangular hole interpretation, 0 to all other preferences.
3. Preference 3 to the convex edge label for a 2 overrides the three preferences of 1/2 for the floating triangle of case (1). All preferences but these four were 0.

Some Extensions

The translation of constraints to inequalities described above does not guarantee that they produce a set of half-spaces whose intersection is the convex hull of the feasible integer solutions. They can produce "noninteger optima," for which supposition values are not forced to 1 or 0. This is reminiscent of the behavior of the linear relaxation operator of Section 12.4.3, and may not be objectionable. If it is, some effort must be expended to cope with it. Here is an example



Case	After 10 iterations				After 20 iterations				After 30 to 40 iterations				
(1)	0.65	0.22	0.01	0.14	0.90	0.07	0	0.04	0.99	0	0	0	
	0.65	0.22	0.01	0.14	0.90	0.07	0	0.04	0.99	0	0	0	
	0.65	0.22	0.01	0.14	0.90	0.07	0	0.04	0.99	0	0	0	
(2)	0.39	0.89	0	0	0.14	0.95	0	0	0	0.99	0	0	
	0.39	0.89	0	0	0.14	0.95	0	0	0	0.99	0	0	
	0.39	0.89	0	0	0.14	0.95	0	0	0	0.99	0	0	
(3)	0.56	0.48	0	0.05	0.81	0.23	0	0	0.99	0	0	0	
	0	0.34	0	0.99	0	0.15	0	0.99	0	0	0	0.99	+
	0.56	0.48	0	0.05	0.81	0.23	0	0	0.99	0	0	0	

(c)

Fig. 12.14 As in Fig. 12.10, the triangle of (a) is to be assigned labels, and the changing label weights are shown for three cases in (c) using the format of (b). Supposition values for junction labels were used as well, but are not shown. All initial supposition values were 0.

of the problem. Assume three logical constraints, $\tilde{}(A \wedge B)$, $\tilde{}(B \wedge C)$, and $\tilde{}(C \wedge A)$. Suppose A, B, and C have equal preferences of unity (the preference vector is $(1, 1, 1)$). Translating the constraints yields

$$\begin{aligned} p(A) + p(B) &\leqslant 1 \\ p(B) + p(C) &\leqslant 1 \\ p(C) + p(A) &\leqslant 1 \end{aligned} \tag{12.35}$$

The best feasible solution has a total preference of 1½, and is

$$p(A) = p(B) = p(C) = \tfrac{1}{2} \tag{12.36}$$

Here the "best" solution is outside the convex hull of the integer solutions (Fig. 12.15).

The basic way to ensure integer solutions is to use stronger constraints than those arising from the simple rules given above. These may be introduced at first, or when some noninteger optimum has been reached. These stronger constraints are called *cutting planes*, since they cut off the noninteger optima vertices. In the example above, the obvious stronger constraint is

$$p(A) + p(B) + p(C) \leqslant 1 \tag{12.37}$$

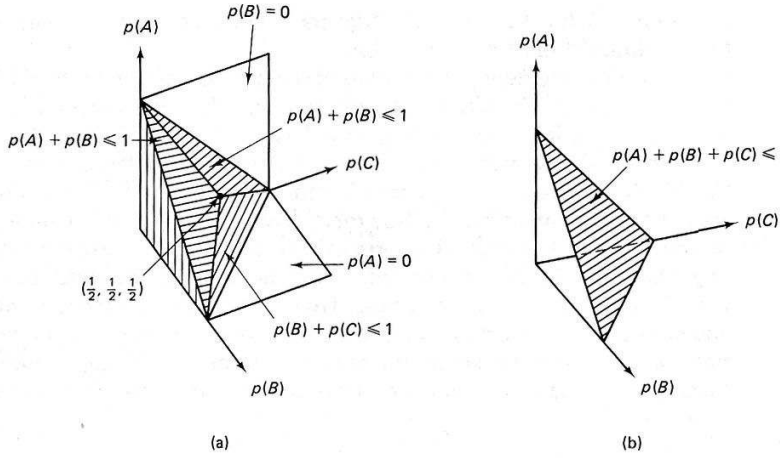


Fig. 12.15 (a) shows part of the surface of the feasible solid with constraints $\neg (A \ \& \ B)$, $\neg (B \ \& \ C)$, $\neg (C \ \& \ A)$, and the non-integer vertex where the three halfspaces intersect. (b) shows a cutting plane corresponding to the constraint "at most one of A, B, or C" that removes the non-integer vertex.

which says that at most one of A, B, and C is true (this is a logical consequence of the logical constraints). Such cutting planes can be derived as needed, and can be guaranteed to eliminate all noninteger optimal vertices in a finite number of cuts [Gomory 1968; Garfinkel and Nemhauser 1972]. Equality constraints may be introduced as two inequality constraints in the obvious way: This will constrain the feasible region to a plane.

Suppose that one desires "weak rules," which are usually true but which can be broken if evidence demands it? For each constraint arising from such a rule, add a hypothesis to represent the situation where the rule is broken. This hypothesis is given a negative preference depending on the strength of the rule, and the constraint enhanced to include the possibility of the broken rule. For example, if a weak rule gives the constraint $P \vee Q$, create a hypothesis H equivalent to $\tilde{}(P \vee Q) = (\tilde{}(P) \wedge \tilde{}(Q))$, and replace the constraint with $P \vee Q \vee H$. Then by "paying the cost" of the negative preference for H, we can have neither P nor Q true.

Hypotheses can be created as the algorithm proceeds by having demon-like "generator hypotheses." The demon watches the supposition value of the generator, and when it becomes high enough, runs a program that generates explicit hypotheses. This is clearly useful; it means that all possible hypotheses do not need to be generated in advance of any scene investigation. The generator can be given a preference equal to that of the best hypotheses that it can generate.

Relaxation sometimes should determine a real number (such as the slope of a line) instead of a truth value. A generator-like technique can allow the method to refine the value of real-valued hypotheses. Basically, the idea is to assign a (Boolean-valued) generator hypothesis to a range of values for the real value to be

determined. When this generator triggers, more hypotheses are generated to get a finer partition of the range, and so on.

The enhancements to the linear programming paradigm of relaxation give some idea of the flexibility of the basic idea, but also reveal that the method is not at all cut-and-dried, and is still open to basic investigation. One of the questions about the method is exactly how to take advantage of parallel computation capabilities. Each constraint and hypothesis can be given its own processor, but how should they communicate? Also, there seems little reason to suppose that the optimization problems for this form of relaxation are any easier than they are for any other multidimensional search, so the method will encounter the usual problems inherent in such optimization. However, despite all these technical details and problems of implementation, the linear programming paradigm for the relaxation computation is a coherent formalization of the process. It provides a relatively "classical" context of results and taxonomy of problems [Hummel and Zucker 1980].

12.5 ACTIVE KNOWLEDGE

Active knowledge systems [Freuder 1975] are characterized by the use of procedures as the elementary units of knowledge (as opposed to propositions or data base items, for instance). We describe how active knowledge might work, because it is a logical extreme of the procedural implementation of propositions. In fact, this style of control has not proven influential; some reasons are given below.

Active knowledge is notionally parallel and heterarchical. Many different procedures can be active at the same time depending on the input. For this reason active knowledge is more easily applied to belief maintenance than to planning; it is very difficult to organize sequential activity within this discipline. Basically, each procedure is responsible for a "chunk" of knowledge, and knows how to manage it with respect to different visual inputs. Control in an active knowledge system is completely distributed. Active knowledge can also be viewed as an extension of the constraint relaxation problem; powerful procedures can make arbitrary detailed tests of the consistency between constraints.

Each piece of active knowledge (program module) knows which other modules it depends on, which depend on it, which it can complain to, and so forth. Thus the choice of "what to do next" is contained in the modules and is not made by an exterior executive.

We describe HYPER, a particular active knowledge system design which illustrates typical properties of active knowledge [Brown 1975]. HYPER provides a less structured mechanism for construction and exploration of hypotheses than does LP-relaxation. Using primitive control functions of the system, the user may write programs for establishing hypotheses and for using the conclusions so reached. The programs are "procedurally embedded" knowledge about a problem domain (e.g. how events relate one to another, what may be conjectured or inferred from a clue, or how one might verify a hypothesis).

When HYPER is in use on a particular task in a domain, hypotheses are created, or instantiated, on the basis of low-level input, high-level beliefs, or any

reason in between. The process of establishing the initial hypotheses leads to a propagation of activity (creation, verification, and disconfirmation of hypotheses). Activation patterns will generally vary with the particular task, in heterarchical fashion. A priority mechanism can rank hypotheses in importance depending on the data that contribute to them. Generally, the actions that occur are conditioned by previous assumptions, the data, the success of methods, and other factors. HYPER can be used for planning applications and for multistep vision processing as well as inference (procedures then should generate parallel activity only under tight control). We shall thus allow HYPER to make use of a context-oriented data base (Section 13.1.1). It will use the context mechanism to implement "alternative worlds" in which to reason.

12.5.1 Hypotheses

A HYPER hypothesis is the attribution of a predicate to some arguments; its name is always of the form (PREDICATE ARGUMENTS). Sample hypothesis names could be (HEAD-SHAPED REGION1), (ABOVE A B), (TRIANGLE (X1,Y1) (X2,Y2) (X3,Y3)). A hypothesis is represented as a data structure with four components; the *status*, *contents*, *context*, and *links* of the hypothesis.

The *status* represents the state of the HYPER's knowledge of the truth of the hypothesis; it may be T(rue), F(alse), (in either case the hypothesis has been *established*) or P(ending). The *contents* are arbitrary; hypotheses are not just truth-valued assertions. The hypothesis was asserted in the data-base context given in *context*. The *links* of a hypothesis H are pointers to other hypotheses that have asked that H be established because they need H's contents to complete their own computations.

12.5.2 HOW-TO and SO-WHAT Processes

Two processes are associated with every predicate P which appears as the predicate of a hypothesis. Their names are (HOW-TO P) and (SO-WHAT P). In them is embedded the procedural knowledge of the system which remains compiled in from one particular task to another in a problem domain. (HOW-TO P) expresses how to establish the hypothesis (P arguments). It knows what other hypotheses must be established first, the computations needed to establish (P arguments), and so forth. It has a backward-chaining flavor. Similarly, (SO-WHAT P) expresses the consequences of knowing P: what hypotheses could possibly now be established using the contents of (P arguments), what alternative hypotheses should be explored if the status of (P arguments) is *F*, and so on. The feeling here is of forward chaining.

12.5.3 Control Primitives

HYPER hypotheses interact through *primitive control statements*, which affect the investigation of hypotheses and the ramification of their consequences. The primi-

tives are used in HOW-TO and SO-WHAT programs together with other general computations. Most primitives have an argument called priority, which expresses the reliability, urgency, or importance of the action they produce, and is used to schedule processes in a nonparallel computing environment (implemented as a priority job queue [Appendix 2]). The primitives are GET, AFFIRM, DENY, RETRACT, FAIL, WONDERIF, and NUDGE.

GET is to ascertain or establish the status and contents of a hypothesis. It takes a hypothesis H and priority PRI as arguments and returns the status and contents of the hypothesis. If H's status is T or F at the time of execution of the statement, the status and contents are returned immediately. If the status is P (pending), or if H has not been created yet, the current HOW-TO or SO-WHAT program calling GET (call it CURPROG) is exited, the proper HOW-TO job (i.e., the one that deals with H's predicate) is run at priority PRI with argument H, and a link is planted in H back to CURPROG. When H is established, CURPROG will be reactivated through the link mechanism.

AFFIRM is to assert a hypothesis as true with some contents. AFFIRM(H,CONT,PRI) sets H's status to T, its contents to CONT, activates its linked programs and then executes the proper SO-WHAT program on it. The newly activated SO-WHAT programs are performed with priority PRI.

DENY is to assert that a hypothesis with some contents is false. DENY(H,CONT,PRI) is like AFFIRM except that no activation though links occurs, and the status of H is of course set to F.

ASSUME is to assert a hypothesis as true hypothetically. ASSUME(H,CONT,PRI) uses the data base context mechanism to create a new context in which H is AFFIRMED; the original context in which the ASSUME command is given is preserved in the context field of H. H itself is stored into a context-dependent item named LASTASSUMED; this corresponds to remembering a decision point in PLANNER. By using the information in LASTASSUMED and the primitive FAIL (see below), simple backtracking can take place in a tree of contexts.

RETRACT(H) establishes as false a hypothesis that was previously ASSUMEd. RETRACT is always carried out at highest priority, on the principle that it is good to leave the context of a mistaken assumption as quickly as possible. Information (including the name of the context being exited) is transmitted back to the original context in which H was ASSUMEd by passing it back in the fields of H.

FAIL just RETRACTs the hypothesis that is the value of the item LASTASSUMED in the present context.

WONDERIF is to pass suggested contents to HOW-TO processes for verification. It can be useful if verifying a value is easier than computing it from scratch, and is the primitive that passes substantive suggestions. WONDERIF(H1, CONT, H2, PRI) approximates the notion "H2 wonders if H1 has contents CONT."

NUDGE is to wake up HOW-TO programs. NUDGE(H,PRI) runs the HOW-TO program on H with priority PRI. It is used to awaken hypotheses that might be able to use information just computed. Typically it is a SO-WHAT pro-

gram that NUDGEs others, since the SO-WHAT program is responsible for using the fact that a hypothesis is known.

12.5.4 Aspects of Active Knowledge

The active knowledge style of computation raises a number of questions or problems for its users.

A hypothesis whose contents may attain a large range can be established for some contents and thus express a perfectly good fact (e.g., that a given location of an x-ray does not contain evidence for a tumor) but such a fact is usually of little help when we want to reason about the predicate (about the location of tumors). The SO-WHAT program for a predicate should be written so as to draw conclusions from such negative facts if possible, and from the conclusions endeavor to establish the hypothesis as true for some contents. Usually, therefore, it would set the status of the hypothesis back to P and initiate a new line of attack, or at its discretion abandon the effort and start an entirely new line of reasoning.

Priorities

A major worry with the scheme as described is that priorities are used to schedule running of HOW-TO and SO-WHAT processes, not to express the importance (or supposition value) of the hypotheses. The hypothesis being investigated has no way to communicate how important it is to the program that operates on it, so it is impossible to accumulate importance through time. A very significant fact may lie ignored because it was given to a self-effacing process that had no way of knowing it had been handed something out of the ordinary.

The obvious answer is to make a supposition value a field of the hypothesis, like its status or contents—a hypothesis should be given a measure of its importance. This value may be used to compute execution priorities for jobs involving it. This solution is used in some successful systems [Turner 1974].

Structuring Knowledge

One has a wide choice in how to structure the "theory" of a complex problem in terms of HYPER primitives, predicates, arguments, and HOW-TO and SO-WHAT processes. The set of HOW-TO and SO-WHAT processes specify the complete theory of the tasks to be performed; HYPER encourages one to consider the interrelations between widely separated and distinct-sounding facts and conjectures about a problem, and the structure it imposes on a problem is minimal.

Since HOW-TO and SO-WHAT processes make explicit references to one another via the primitives, they are not "modular" in the sense that they can easily be plugged in and unplugged. If HOW-TO and SO-WHAT processes are invoked by patterns, instead of by names, some of the edge is taken off this criticism. Removing a primitive from a program could modify drastically the avenues of activation, and the consequences of such a modification are sometimes hard to foresee in a program that logically could be running in parallel.

Writing a large and effective program for one domain may not help to write a program for another domain. New problems of segmenting the theory into predicates, and quantifying their interactions via the primitives, setting up a priority

structure, and so forth will occur in the new domain, and it seems quite likely that little more than basic utility programs will carry over between domains.

EXERCISES

12.1 In the production system example, write a production that specifies that blue regions are sky using the opponents color notation. How would you now deal with blue regions that are lakes (a) in the existing color-only system; (b) in a system which has surface orientation information?

12.2 This theorem was posed as a challenge for a clausal automatic theorem prover [Henschen et al. 1980]. It is obviously true: what problems does it present?

$$\{[(\exists x)(\forall y)(P(x) \iff P(y))]$$
$$\iff [[(\exists x)Q(x)] \iff [(\forall y)(P(y))]]\} \iff$$
$$\{[(\exists x)(\forall y)(Q(x) \iff Q(y))]$$
$$\iff [[(\exists x)P(x)] \iff [(\forall y)(Q(y))]]\}$$

12.3 Prove that the operator of Eq. (12.18) takes probability vectors into probability vectors, thus deriving the reason for Eq. (12.19).

12.4 Verify (12.23).

12.5 How do the c_{ij} of (12.18) affect the labeling? What is their semantics?

12.6 If events X and Y always co-occur, then $p(X, Y) = p(X) = p(Y)$. What is the correlation in this case? If X and Y never co-occur, what values of $p(X)$ and $p(Y)$ produce a minimum correlation? If X and Y are independent, how is $p(X, Y)$ related to $p(X)$ and $p(Y)$? What is the value of the correlation of independent X and Y?

12.7 Complete Table 12.3.

12.8 Use only the labels of Fig. 12.9b and c to compute covariances in the manner of Table 12.3. What do you conclude?

12.9 Show that Eq. (12.29) preserves the important properties of the weight vectors.

12.10 Think of some rival normalization schemes to Eq. (12.29) and describe their properties.

12.11 Implement the linear and nonlinear operators of Section 12.4.3 and 12.4.4 and investigate their properties. Include your ideas from Exercise 12.10.

12.12 Show a case that the nonlinear operator of Eq. (12.29) assigns nonzero weights to maximally incompatible labels (those with $r_{ij} = -1$).

12.13 How can a linear programming relaxation such as the one outlined in sec. 12.4.5 cope with faces or edges of the feasible solution solid that are normal to the preference direction, yielding several solutions of equal preference?

12.14 In Fig. 12.11, what (P, Q) solution is optimal if the preference vector is $(1, 4)$? $(4, 1)$? $(-1, 1)$? $(1, -1)$?

REFERENCES

AIKINS, J. S. "Prototypes and production rules: a knowledge representation for computer consultations." Ph.D. dissertation, Computer Science Dept., Stanford Univ., 1980.

BAJCSY, R. and A. K. JOSHI. "A partially ordered world model and natural outdoor scenes." In *CVS*, 1978.

BARROW, H. G. and J. M. TENENBAUM. "MSYS: a system for reasoning about scenes." Technical Note 121, AI Center, SRI International, March 1976.

BRACHMAN, R. J. "On the epistemological status of semantic networks." In *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler (Ed.). New York: Academic Press, 1979, 3–50.

BROWN, C. M. "The HYPER system." DAI Working Paper 9, Dept. of Artificial Intelligence, Univ. Edinburgh, July 1975.

BUCHANAN, B. G. and E. A. FEIGENBAUM. "DENDRAL and meta-DENDRAL: their applications dimensions." *Artificial Intelligence 11*, 2, 1978, 5–24.

BUCHANAN, B. G. and T. M. MITCHELL. "Model-directed learning of production rules." In *Pattern Directed Inference Systems,* D. A. Waterman and F. Hayes-Roth (Eds.). New York: Academic Press, 1978.

COLLINS, A. "Fragments of a theory of human plausible reasoning." *Theoretical Issues in Natural Language Processing-2*, Univ. Illinois at Urbana-Champaign, July 1978, 194–201.

DAVIS, R. and J. KING. "An overview of production systems." AIM-271, Stanford AI Lab, October 1975.

DAVIS, L. S. and A. ROSENFELD. "Applications of relaxation labelling 2. Spring-loaded template matching." Technical Report 440, Computer Science Center, Univ. Maryland, 1976.

DELIYANNI, A. and R. A. KOWALSKI. "Logic and semanatic networks." *Comm. ACM 22*, 3, March 1979, 184–192.

ERMAN, L. D. and V. R. LESSER. "A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge." *Proc.*, 4th IJCAI, September 1975, 483–490.

FELDMAN, J. A. and Y. YAKIMOVSKY. "Decision theory and artificial intelligence: I. A semantics-based region analyser." *Artificial Intelligence 5*, 4, 1974, 349–371.

FIKES, R. E. "Knowledge representation in automatic planning systems." In *Perspectives on Computer Science*, A. Jones (Ed). New York: Academic Press, 1977.

FIKES, R. E. and N. J. NILSSON. "STRIPS: a new approach to the application of theorem proving to problem solving." *Artificial Intelligence 2*, 3/4, 1971, 189–208.

FREUDER, E. C. "A computer system for visual recognition using active knowledge." Ph.D. dissertation, MIT, 1975.

FREUDER, E. C. "Synthesizing constraint expressions." *Comm. ACM 21*, 11, November 1978, 958–965.

GARFINKEL, R. S. and G. L. NEMHAUSER. *Integer Programming.* New York: Wiley, 1972.

GOMORY, R. E. "An algorithm for integer solutions to linear programs." *Bull. American Mathematical Society 64*, 1968, 275–278.

HARALICK, R. M. "The characterization of binary relation homomorphisms." *International J. General Systems 4*, 1978, 113–121.

HARALICK, R. M. and J. S. KARTUS. "Arrangements, homomorphisms, and discrete relaxation." *IEEE Trans. SMC 8*, 8, August 1978, 600–612.

HARALICK, R. M. and L. G. SHAPIRO. "The consistent labeling problem: Part I." *IEEE Trans. PAMI 1*, 2, April 1979, 173–184.

HARALICK, R. M., L. S. DAVIS, and A. ROSENFELD. "Reduction operations for constraint satisfaction." *Information Sciences 14*, 1978, 199–219.

HAYES, P. J. "In defense of logic." *Proc.*, 5th IJCAI, August 1977, 559–565.

HAYES, P. J. "Naive physics: ontology for liquids." Working paper, Institute for Semantic and Cognitive Studies, Geneva, 1978a.

HAYES, P. J. "The naive physics manifesto." Working paper, Institute for Semantic and Cognitive Studies, Geneva, 1978b.

HAYES, P. J. "The logic of frames." *The Frame Reader.* Berlin: DeGruyter, in press, 1981.

HENDRIX, G. G. "Encoding knowledge in partitioned networks." In *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler (Ed.). New York: Academic Press, 1979, 51–92.

HENSCHEN, L., E. LUSK, R. OVERBEEK, B. SMITH, R. VEROFF, S. WINKER, and L. WOS. "Challenge Problem 1." *SIGART Newsletter 72*, July 1980, 30–31.

HERBRAND, J. "Recherches sur la théorie de la démonstration." *Travaux de la Socíeté des Sciences et des Lettres de Varsovie, Classe III, Sciences Mathématiques et Physiques, 33,* 1930.

HEWITT, C. "Description and theoretical analysis (using schemata) of PLANNER" (Ph.D. dissertation). AI-TR-258, AI Lab, MIT, 1972.

HINTON, G. E. "Relaxation and its role in vision." Ph.D. dissertation, Univ. Edinburgh, December 1979.

HUMMEL, R. A. and S. W. ZUCKER. "On the foundations of relaxation labelling processes." TR-80-7, Computer Vision and Graphics Lab, Dept. of Electrical Engineering, McGill Univ., July 1980.

KOWALSKI, R. A. "Predicate logic as a programming language." *Information Processing 74.* Amsterdam: North-Holland, 1974, 569–574.

KOWALSKI, R. A. *Logic for Problem Solving.* New York: ElsevierNorth-Holland (AI Series), 1979.

LINDSAY, R. K., B. G. BUCHANAN, E. A. FEIGENBAUM, and J. LEDERBERG. *Applications of Artificial Intelligence to Chemistry: The DENDRAL Project.* New York: McGraw-Hill, 1980.

LOVELAND, D. "A linear format for resolution." *Proc.*, IRIA 1968 Symp. on Automatic Demonstration, Versailles, France. New York: Springer-Verlag, 1970.

LOVELAND, D. *Automated Theorem Proving: A Logical Basis.* Amsterdam: North-Holland, 1978.

MCCARTHY, J. "Circumscription induction—a way of jumping to conclusions." Unpublished report, Stanford AI Lab, 1978.

MCCARTHY, J. and P. J. HAYES. "Some philosophical problems from the standpoint of artificial intelligence." In *MI4*, 1969.

MCDERMOTT, D. "The PROLOG phenomenon." *SIGART Newsletter 72*, July 1980, 16–20.

MENDELSON, E. *Introduction to Mathematical Logic.* Princeton, NJ: D. Van Nostrand, 1964.

MINSKY, M. L. "A framework for representing knowledge." In *PCV*, 1975.

NEWELL, A., J. SHAW, and H. SIMON. "Empirical explorations of the logic theory machine." In *Computers and Thought*, E. Feigenbaum and J. Feldman (Eds.). New York: McGraw-Hill, 1963.

NILSSON, N. J. *Problem-Solving Methods in Artificial Intelligence.* New York: McGraw-Hill, 1971.

NILSSON, N. J. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga, 1980.

REITER, R. "On reasoning by default." *Theoretical Issues in Natural Language Processing-2*, Univ. Illinois at Urbana-Champaign, July 1978, 210–218.

ROBINSON, J. A. "A machine-oriented logic based on the resolution principle." *J. ACM 12*, 1, January *1965*, 23–41.

ROSENFELD, A., R. A. HUMMEL and S. W. ZUCKER. "Scene labelling by relaxation operations." *IEEE Trans. SMC 6*, 1976, 420.

RYCHNER, M. "An instructable production system: basic design issues." In *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds.). New York: Academic Press, 1978.

SHORTLIFFE, E. H. *Computer-Based Medical Consultations: MYCIN.* New York: American Elsevier, 1976.

SLOAN, K. R. "World model driven recognition of natural scenes." Ph.D. dissertation, Moore School of Electrical Engineering, Univ. Pennsylvania, June 1977.

SLOAN, K. R. and R. BAJCSY. "World model driven recognition of outdoor scenes." TR40, Computer Science Dept., Univ. Rochester, September 1979.

SUSSMAN, G. J. and D. MCDERMOTT. "Why conniving is better than planning." AI Memo 255, AI Lab, MIT, 1972.

TURNER, K. J. "Computer perception of curved objects using a television camera." Ph.D. dissertation, School of Artificial Intelligence, Univ. Edinburgh, 1974.

WARREN, H. D., L. PEREIRA, and F. PEREIRA. "PROLOG: The language and its implementation compared with LISP." *Proc.*, Symp. on Artificial Intelligence and Programming Languages, SIGPLAN/SIGART, 1977; *SIGPLAN Notices 12*, 8, August 1977, 109–115.

WATERMAN, D. A. and F. HAYES-ROTH (Eds.). *Pattern-Directed Inference Systems.* New York: Academic Press, 1978.

WINOGRAD, T. "Extended inference modes in reasoning by computer systems." *Proc.*, Conf. on Inductive Logic, Oxford Univ., August 1978.

ZADEH, L. "Fuzzy sets." *Information and Control 8*, 1965, 338–353.

ZUCKER, S. W. "Relaxation labelling and the reduction of local ambiguities." Technical Report 451, Computer Science Dept., Univ. Maryland, 1976.

Goal Achievement 13

Goal Achievement and Vision

Goals and plans are important for visual processing.

- Some skilled vision actually is like problem solving.
- Vision for information gathering can be part of a planned sequence of actions.
- Planning can be a useful and efficient way to guide many visual computations, even those that are not meant to imply "conscious" cognitive activity.

The artificial intelligence activity often called *planning* traditionally has dealt with "robots" (real or modeled) performing actions in the real world. Planning has several aspects.

- Avoid nasty "subgoal interactions" such as getting painted into a corner.
- Find the plan with optimal properties (least risk, least cost, maximized "goodness" of some variety).
- Derive a sequence of steps that will achieve the goal from the starting situation.
- Remember effective action sequences so that they may be applied in new situations.
- Apply planning techniques to giving advice, presumably by simulating the advisee's actions and making the next step from the point they left off.
- Recover from errors or changes in conditions that occur in the middle of a plan.

Traditional planning research has not concentrated on plans with information gathering steps, such as vision. The main interest in planning research has been the expensive and sometimes irrevocable nature of actions in the world. Our goal is to give a flavor of the issues that are pursued in much more detail in the planning

literature [Nilsson 1980; Tate 1977; Fahlman 1974; Fikes and Nilsson 1971; Fikes et al. 1972a; 1972b; Warren 1974; Sacerdoti 1974; 1977; Sussman 1975].

Planning concerns an active agent and its interaction with the world. This conception does not fit with the idea of vision as a passive activity. However, one claim of this book is that much of vision is a constructive, active, goal-oriented process, replete with uncertainty. Then a model of vision as a sequence of decisions punctuated by more or less costly information gathering steps becomes more compelling. Vision often is a sequential (recursive, cyclical) process of alternating information gathering and decision making. This paradigm is quite common in computer vision [Shirai 1975; Ballard 1978; Mackworth 1978; Ambler et al. 1975]. However, the formalization of the process in terms of minimizing cost or maximizing utility is not so common [Feldman and Sproull 1977; Ballard 1978; Garvey 1976]. This section examines the paradigms of planning, evaluating plans with costs and utilities, and how plans may be applied to vision processing.

13.1 SYMBOLIC PLANNING

In artificial intelligence, planning is usually a form of problem-solving activity involving a formal "simulation" of a physical world. (Planning, theorem proving, and state-space problem solving are all closely related.) There is an agent (the "robot") who can perform actions that transform the state of the simulated world. The robot planner is confronted with an initial world state and a set of goals to be achieved. Planning explores world states resulting from actions, and tries to find a sequence of actions that achieves the goals. The states can be arranged in a tree with initial state as the root, and branches resulting from applying different actions in a state. Planning is a search through this tree, resulting in a path or sequence of actions, from the root to a state in which the goals are achieved. Usually there is a metric over action sequences; the simplest is that there be as few actions as possible. More generally (Section 13.2), actions may be assigned some cost which the planner should minimize.

13.1.1 Representing the World

This section illustrates planning briefly with a classical example—block stacking. In one simple form there are three blocks initially stacked as shown on the left in Fig. 13.1, to be stacked as shown.

This task may be "formalized" [Bundy 1978] using only the symbolic objects Floor, A, B, and C. (A formalization suitable for a real automated planner must be much more careful about details than we shall be). Assume that only a single block can be picked up at a time. Necessary predicates are CLEAR(X) which is true if a block may be put directly on X and which must be true before X may be picked up, and ON(X, Y), which is true if X is resting directly on Y. Let us stipulate that the Floor is always CLEAR, but otherwise if ON(X, Y) is true, Y is not CLEAR. Then the initial situation in Fig. 13.1 is characterized by the following assertions.

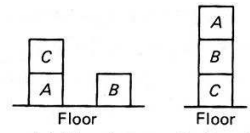


Fig. 13.1 A simple block stacking task.

INITIAL STATE: ON(C,A), ON(A, Floor), ON(B, Floor),
CLEAR(C), CLEAR(B), CLEAR(Floor)

The goal state is one in which the following two assertions are true.

GOAL ASSERTIONS: ON(A,B), ON(B,C)

With only these rules, the formalization of the block stacking world yields a very "loose" semantics. (The task easily translates to sorting integers with some restrictions on operations, or to the "seriation" task of arranging blocks horizontally in order of size, or a host of others.)

Actions transform the set of assertions describing the world. For problems of realistic scale, the representation of the tree of world states is a practical problem. The issue is one of maintaining several coexisting "hypothetical worlds" and reasoning about them. This is another version of the frame problem discussed in sec. 12.1.6. One way to solve this problem is to give each assertion an extra argument, naming the hypothetical world (usually called a situation [Nilsson 1980; McCarthy and Hayes 1969]) in which the assertion holds. Then actions map situations to situations as well as introducing and changing assertions.

An equivalent way to think about (and implement) multiple, dependent, hypothetical worlds is with a tree-structured *context-oriented data base.* This idea is a general one that is useful in many artificial intelligence applications, not just symbolic planning. Such data bases are included in many artificial intelligence languages and appear in other more traditional environments as well. A context-oriented data base *acts* like a tree of data bases; at any node of the tree is a set of assertions that makes up the data base. A new data base (context) may be spawned from any context (data base) in the tree. All assertions that are true in the spawning (ancestor) context are initially true in the spawned (descendant) context. However, new assertions added in any context or deleted from it do not affect its ancestor. Thus by going back to the ancestor, all data base changes performed in the descendent context disappear.

Implementing such a data base is an interesting exercise. Copying all assertions to each new context is possible, but very wasteful if only a few changes are made in each context. The following mechanism is much more efficient. The root or initial context has some set of assertions in it, and each descendant context is merely an *add list* of assertions to add to the data base and a *delete list* of assertions to delete. Then to see if an assertion is true in a context, do the following.

1. If the context is the root context, look up "as usual."
2. Otherwise, if the assertion is on the *add list* of this context, return *true.* If the assertion is on the *delete list* of this context, return *false.*

3. Otherwise, recursively apply this procedure to the ancestor of this context.

In a general programming environment, contexts have names, and there is the facility of executing procedures "in" particular contexts, moving around the context tree, and so forth. However, in what follows, only the ability to look up assertions in contexts is relevant.

13.1.2 Representing Actions

Represent an action as a triple.

ACTION ::= [PATTERN, PRECONDITIONS, POSTCONDITIONS].

Here the pattern gives the name of the action and names for the objects with which it deals—its "formal parameters." Preconditions and postconditions may use the formal variables of the pattern. In a sense, the preconditions and postconditions are the "body" of the action, with subroutine-like "variable bindings" taking place when the action is to be performed. The preconditions give the world states in which the action may be applied. Here the preconditions are assumed simply to be a list of assertions all of which must be true. The postconditions describe the world state that results from performing the action. The context-oriented data base of hypothetical worlds can be used to implement the postconditions.

POSTCONDITIONS ::= [ADD-LIST, DELETE-LIST].

An action is then performed as follows.

1. Bind the pattern variables to entities in the world, thus binding the associated variables in the preconditions and postconditions.
2. If the preconditions are met (the bound assertions exist in the data base), do the next step, else exit reporting failure.
3. Delete the assertions in the delete list, add those in the add list, and exit reporting success.

Here is the *Move* action for our block-stacking example.

Move Object X from Y to Z			
PATTERN	*PRECONDITIONS*	*DELETE-LIST*	*ADD-LIST*
Move(X,Y,Z)	CLEAR(X) CLEAR(Z) ON(X,Y)	ON(X,Y) CLEAR(Z)	ON(X,Z) CLEAR(Y)

Here X, Y, and Z are all variables bound to world entities. In the initial state of Fig. 13.1, Move(C,A,Floor) binds X to C, Y to A, Z to Floor, and the preconditions are satisfied; the action may proceed.

However, notice two things.

1. The action given above deletes the CLEAR(Floor) assertion that always should be true. One must fix this somehow; putting CLEAR(Floor) in the add-list does the job, but is a little inelegant.
2. What about an action like Move(C,A,C)? It meets the preconditions, but causes trouble when the add and delete lists are applied. One fix here is to keep in the data base ("world model") a set of assertions such as Different (A,B), Different(A,Floor), . . . , and to add assertions such as Different (X,Z) to the preconditions of Move.

Such housekeeping chores and details of axiomatization are inherent in applying basically syntactic, formal solution methods to problem solving. For now, let us assume that CLEAR(Floor) is never deleted, and that Move(X,Y,Z) is applied only if Z is different from X and Y.

13.1.3 Stacking Blocks

In the block-stacking example, the goal is two simultaneous assertions, ON(A,B) and ON(B,C). One solution method proceeds by repeatedly picking a goal to work on, finding an operator that moves closer to the goal, and applying it. In this case of only one action the question is how to apply it—what to move where. This is answered by looking at the postconditions of the action in the light of the goal. The reasoning might go like this: ON(B,C) can be made true if X is B and Z is C. That is possible in this state if Y is A; all preconditions are satisfied, and the goal ON(B,C) can be achieved with one action.

Part of the world state (or context) tree the planner must search is shown in Fig. 13.2, where states are shown diagrammatically instead of through sets of assertions. Notice the following things in Fig. 13.2.

1. Trying to achieve ON(B,C) first is a mistake (Branch 1).
2. Trying to achieve ON(A,B) first is also a mistake for less obvious reasons (Branch 2).
3. Branches 1 and 2 show "subgoal interaction." The goals as stated are not independent. Branch 3 must be generated somehow, either through backtracking or some intelligent way of coping with interaction. It will never be found by the single-minded approach of (1) and (2). However, if ON(C,Floor) were one of the goal assertions, Branch 3 could be found.

Clearly, representing world and actions is not the whole story in planning. Intelligent search of the context is also necessary. This search involves subgoal selection, action selection, and action argument selection. Bad choices anywhere can mean inefficient or looping action sequences, or the generation of impossible subgoals. "Intelligent" search implies a meta-level capability: the ability of a program to reason about its own plans. "Plan critics" are often a part of sophisticated planners; one of their main jobs is to isolate and rectify unwanted subgoal interaction [Sussman 1975].

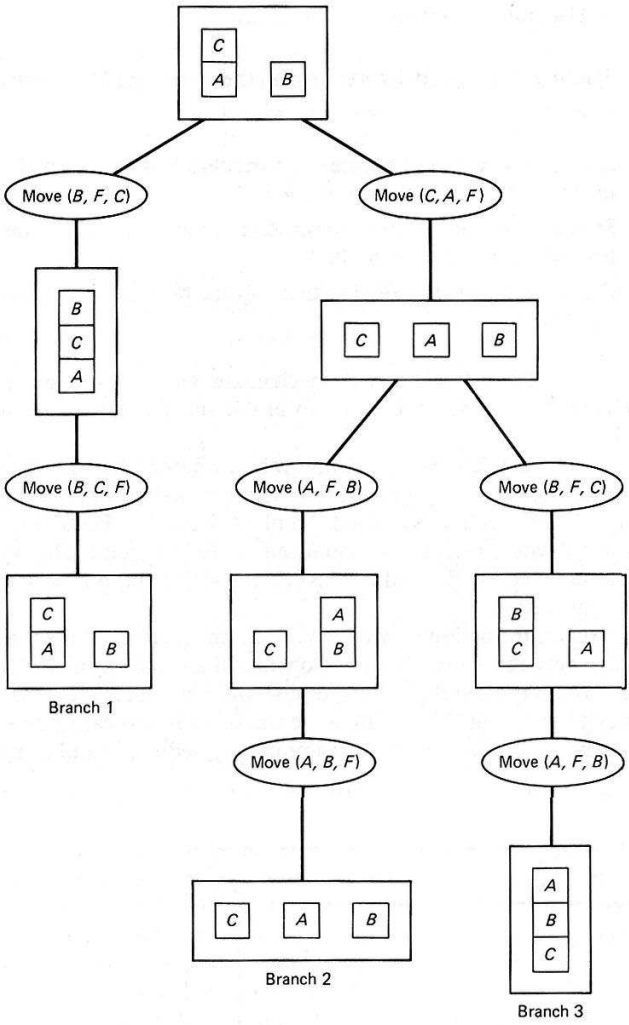


Fig. 13.2 A state tree generated in planning how to stack three blocks.

Intelligent choice of actions is the crux of planning, and is a major research issue. Several avenues have been and are being tried. Perhaps subgoals may be ordered by difficulty and achieved in that order. Perhaps planning should proceed at various levels of detail (like multiresolution image understanding), where the strategic skeleton of a plan is derived without details, then the details are filled in by applying the planner in more detail to the subgoals in the low-resolution plan.

13.1.4 The Frame Problem

All planning is plagued by aspects of the *frame problem* (introduced in Section 12.1.6).

1. It is impractical (and boring) to write down in an action all the things that stay the same when an action is applied.
2. Similarly, it is impractical to reassert in the data base all the things that remain true when an action is implied.
3. Often an action has effects that cannot be represented with simple add and delete lists.

The add and delete list mechanism and the context-oriented data base mechanism addressed the first two problems. The last problem is more troublesome.

Add and delete lists are simple ideas, whereas the world is a complex place. In many interesting cases, the add and delete lists depend on the current state of the world when the action is applied. Think of actions TURNBY(X) and MOVEBY(Z) in a world where orientation and location are important. The orientation and location after an action depend not just on the action but on the state of the world just before the action.

Again, the action may have very complex effects if there are complex dependencies between world objects. Consider the problem of the "monkey and bananas," where the monkey plans to push the box under the bananas and climb on it to reach them (Fig. 13.3). Implementation of realistically powerful add and delete lists may in fact require arbitrary amounts of deduction and computation.

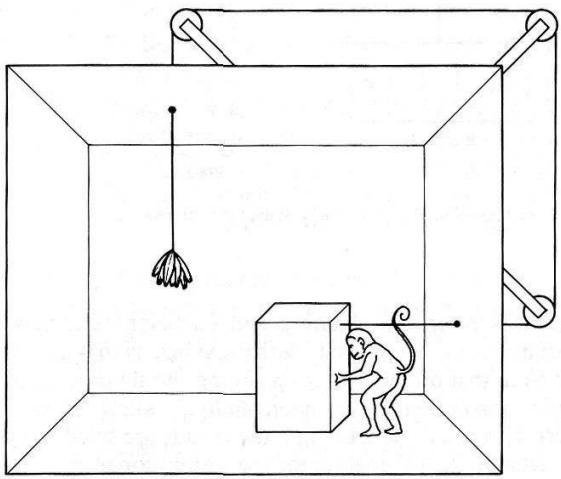


Fig. 13.3 Actions may have complex effects.

This quick précis of symbolic planning does not address many "classical" topics, such as learning or remembering useful plans. Also not discussed are: planning at varying levels of abstraction, plans with uncertain information, or plans with costs. The interested reader should consult the References for more information. The next section addresses plans with costs since they are particularly relevant to vision; some of the other issues appear in the Exercises.

13.2 PLANNING WITH COSTS

Decision making under uncertainty is an important topic in its own right, being of interest to policymakers and managers [Raiffa 1968]. Analytic techniques that can derive the strategy with the "optimal expected outcome" or "maximal expected utility" can be based on Bayesian models of probability.

In [Feldman and Sproull 1977] these techniques are explored in the context of action planning for real-world actions and vision. As an example of the techniques, they are used to model an extended version of the "monkey and bananas" problem of the last section, with multiple boxes but without the maddening pulley arrangement. In the extended problem, there are boxes of different weights which may or may not support the monkey, and he can apply tests (e.g., vision) at some cost to determine whether they are usable. Pushing weighted boxes costs some effort, and the gratification of eating the bananas is "worth" only some finite amount of effort. This extended set of considerations is more like everyday decision making in the number of factors that need balancing, in the uncertainty inherent in the universe, and in the richness of applicable tests. In fact, one might make the claim that human beings always "maximize their expected utility," and if one knew a person's utility functions, his behavior would become predictable. The more intuitive claim that humans beings plan only as far as "sufficient expected utility" can be cast as a maximization operation with nonzero "cost of planning."

The sequential decision-making model of planning with the goal of maximizing the goodness of the expected outcome was used in a travel planner [Sproull 1977]. Knowledge of schedules and costs of various modes of transportation and the attendant risks could be combined with personal prejudices and preferences to produce an itinerary with the maximum expected utility. If unexpected situations (canceled flights, say) arose *en route,* replanning could be initiated; this incremental plan ramification is a natural extension of sequential decision making.

This section is concerned with measuring the expected performance of plans using a single number. Although one might expect one number to be inadequate, the central theorem of decision theory [DeGroot 1970] shows essentially that one number is enough. Using a numerical measure of goodness allows comparisons between normally incomparable concepts to be made easily. Quite frequently numerical scores are directly relevant to the issues at stake in planning, so they are not obnoxiously reductionistic. Decision theory can also help in the process of applying a plan—the basic plan may be simple, but its application to the world may be complex, in terms of when to declare a result established or an action unsuccessful. The decision-theoretic approach has been used in several artificial intelligence and

vision programs [Feldman and Yakimovsky 1974; Bolles 1977; Garvey 1976; Ballard 1978; Sproull 1977].

13.2.1 Planning, Scoring, and Their Interaction

For didactic purposes, the processes of plan generation and plan scoring are considered separately. In fact, these processes may cooperate more or less intimately. The planner produces "sequences" of *actions* for evaluation by the scorer. Each action (computation, information gathering, performing a real-world action) has a *cost*, expressing expenditure of resources, or associated unhappiness. An action has a set of possible *outcomes*, of which only one will really occur when the action is performed. A *goal* is a state of the world with an associated "happiness" or *utility*. For the purposes of uniformity and formal manipulation, goals are treated as (null) actions with no outcomes, and negative utilities are used to express costs. Then the plan has only actions in it; they may be arranged in a strict sequence, or be in loops, be conditional on outcomes of other actions, and so forth.

The *scoring* process evalutes the *expected utility* of a plan. In an uncertain world, a plan prior to execution has only an expected goodness—something might go wrong. Such a scoring process typically is not of interest to those who would use planners to solve puzzles or do proofs; what is interesting is the result, not the effort. But plans that are "optimal" in some sense are decidedly of interest in real-world decision making. In a vision context, plans are usually useful only if they can be evaluated for efficiency and efficacy.

Scoring can take place on "complete" plans, but it can also be used to guide plan generation. The usual artificial intelligence problem-solving techniques of progressive deepening search and branch-and-bound pruning may be applied to planning if scoring happens as the plan is generated [Nilsson 1980]. Scoring can be used to assess the cost of planning and to monitor planning horizons (how far ahead to look and how detailed to make the plan). Scoring will penalize plans that loop without producing results. Plan improvements, such as replanning upon failure, can be assessed with scores, and the contribution of additional steps (say for extra information gathering) can be assessed dynamically by scoring. Scoring can be arbitrarily complex utility functions, thus reflecting such concepts as "risk aversion" and nonlinear value of resources [Raiffa 1968].

13.2.2 Scoring Simple Plans

Scoring and an Example

A *simple plan* is a tree of nodes (there are no loops). The nodes represent actions (and goals). Outcomes are represented by labeled arcs in the tree. A probability of occurrence is associated with each possible outcome; since exactly one outcome actually occurs per action, the probabilities for the possible outcomes of any action sum to unity.

The *score* of a plan is its *expected utility*. The expected utility of any node is recursively defined as its utility times the probability of reaching that node in the

plan, plus the expected utilities of the actions at its (possible) outcomes. The probability of reaching any "goal state" in the plan is the product of probabilities of outcomes forming a path from the root of the plan to the goal state.

As an example, consider the plan shown in Fig. 13.4. If the plan of Fig. 13.4

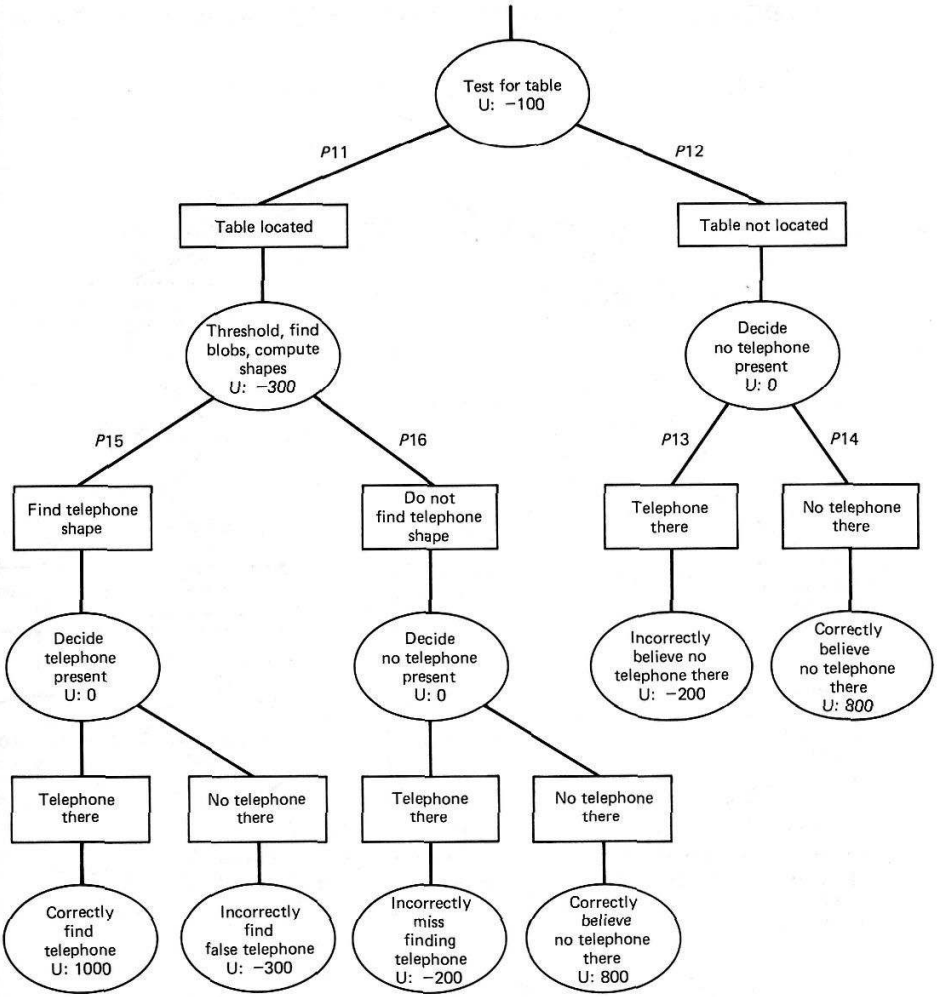


Fig. 13.4 This plan to find a telephone in an office scene involves finding a table first and looking there in more detail. The actions and outcomes are shown. The probabilities of outcomes are assigned symbols (P10, etc.). Utilities (denoted by U:) are given for the individual actions. Note that negative utilities may be considered costs. In this example, decision-making takes no effort, image processing costs vary, and there are various penalties and rewards for correct and incorrect finding of the telephone.

has probabilities assigned to its outcomes, we may compute its expected utility. Figure 13.5 shows the calculation. The probability of correctly finding the telephone is 0.34, and the expected utility of the plan is 433.

Although the generation of a plan may not be easy, scoring a plan is a trivial exercise once the probabilities and utilities are known. In practice, the assignment of probabilities is usually a source of difficulty. The following is an example using

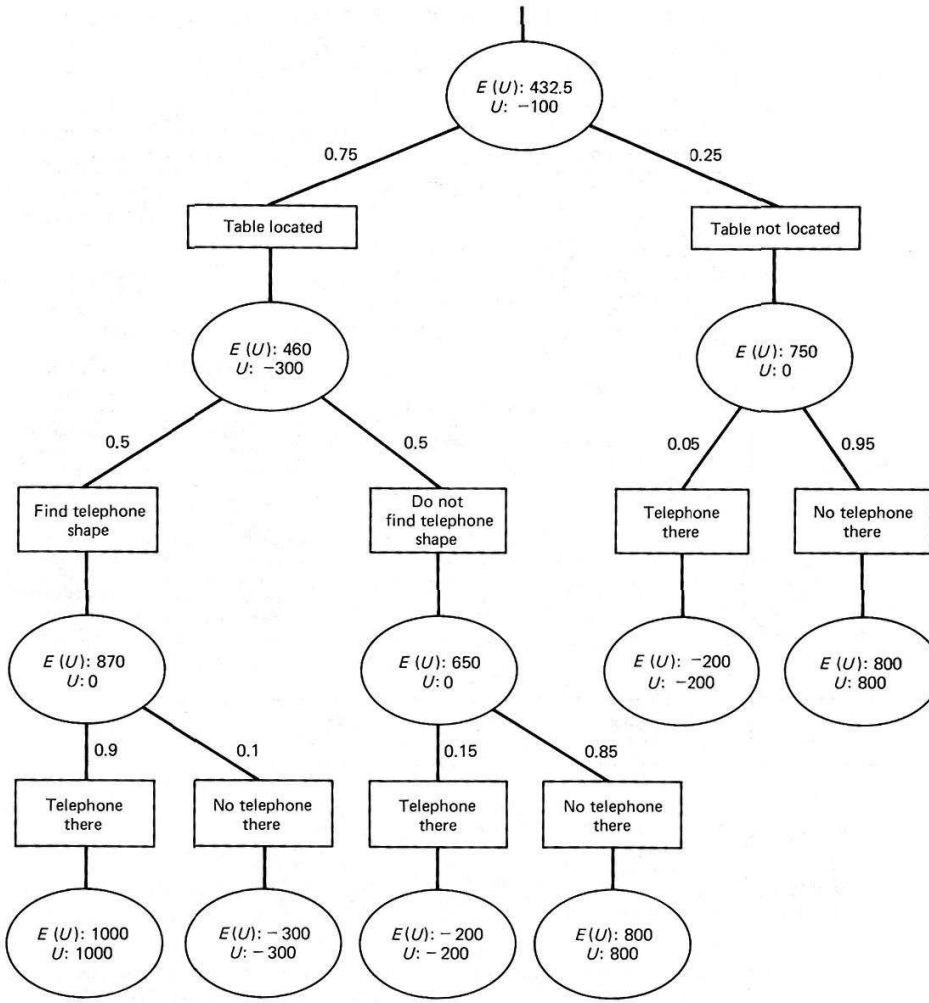


Fig. 13.5 As for Fig. 13.4. U: gives the utility of each action. E(U): gives the expected utillity of the action, which depends on the outcomes below it. Values for outcome probabilities are given on the outcome arcs.

the telephone-finding plan and some assumptions about the tests. Different assumptions yield different scores.

Computing Outcome Probabilities: An Example

This example relies heavily on Bayes' rule:

$$P(B|A)P(A) = P(A \wedge B) = P(A|B)P(B). \tag{13.1}$$

Let us assume a specific a priori probability that the scene contains a telephone.

$$P_1 = \text{apriori probability of Telephone} \tag{13.2}$$

Also assume that something is known about the behavior of the various tests in the presence of what they are looking for. This knowledge may accrue from experiments to see how often the table test found tables when telephones (or tables) were and were not present. Let us assume that the following are known probabilities.

$$P_3 = \text{P(table located|telephone in scene)} \tag{13.3}$$

$$P_5 = \text{P(table located|no telephone in scene)} \tag{13.4}$$

Either there is a telephone or there is not, and a table is located or it is not, so

$$P_2 = \text{a priori probability of no telephone} = 1 - P_1 \tag{13.5}$$

$$P_4 = \text{P(no table located|telephone in scene)} = 1 - P_3 \tag{13.6}$$

$$P_6 = \text{P(no table located|no telephone in scene)} = 1 - P_5 \tag{13.7}$$

Similarly with the "shape test" for telephones: assume probabilities

$$P_7 = \text{P(telephone shape located|telephone)} \tag{13.8}$$

$$P_9 = \text{P(telephone shape located|no telephone)} \tag{13.9}$$

with

$$P_8 = 1 - P_7, \qquad P_{10} = 1 - P_9 \tag{13.10}$$

as above.

There are a few points to make: First, it is not necessary to know exactly these probabilities in order to score the plan; one could use related probabilities and Bayes' rule. Other useful probabilities are of the form

$$\text{P(telephone|telephone shape located)}.$$

In some systems [Garvey 1976] these are assumed to be available directly. This section shows how to derive them from known conditional probabilities that describe the behavior of detectors given certain scene phenomena.

Second, notice the assumption that although both the outcome of the table test and the shape test depend on the presence of telephones, they are taken to be independent of each other. That is, having found a table tells us nothing about the likelihood of finding a telephone shape. Independence assumptions such as this are

useful to limit computations and data gathering, but can be somewhat unrealistic. To account for the dependence, one would have to measure such quantities as

$$\text{P(telephone shape found}\,|\,\text{table located)}.$$

Now to compute some outcome probabilities: Consider the probability

$$P_{11} = \text{P(table located)} \tag{13.11}$$

Let us write

TL for Table Located
TNL for Table Not Located.

A table may be located whether or not a telephone is in the scene. In terms of known probabilities, Bayes' rule yields

$$P_{11} = P_3\,P_1 + P_5\,P_2 \tag{13.12}$$

Then

$$P_{12} = P(\text{TNL}) = 1 - P_{11} \tag{13.13}$$

Calculating P_{13} shows a neat trick using Bayes' Rule:

$$P_{13} = P(\text{telephone}\,|\,\text{TNL}) \tag{13.14}$$

That is, P_{13} is the probability that there is a telephone in the scene given that search for a table was unsuccessful. This probability is not known directly, but

$$\begin{aligned} P_{13} &= \frac{P(\text{telephone and TNL})}{P(\text{TNL})} \\ &= \frac{P(\text{TNL and telephone})}{P_{12}} \\ &= \frac{[P(\text{TNL}\,|\,\text{telephone})\,P(\text{telephone})]}{P_{12}} \\ &= \frac{[P_4\,P_1]}{P_{12}} \end{aligned} \tag{13.15}$$

Then, of course

$$P_{14} = 1 - P_{13} \tag{13.16}$$

Reasoning in this way using the conditional probabilities and assumptions about their independence allows the completion of the calculation of outcome probabilities (see the Exercises). One possibly confusing point occurs in calculation of P_{15}, which is

$$P_{15} = P(\text{telephone shape found}\,|\,\text{table located}) \tag{13.17}$$

By assumption, these events are only indirectly related. By the simplifying assumptions of independence, the shape operator and the table operator are independent in their operation. (Such assumptions might be false if they used common image processing subroutines, for example.) Of course, the probability of success of each

depends on the presence of a telephone in the scene. Therefore their performance is linked in the following way (see the Exercises). (Write TSL for Telephone Shape Located.)

$$P_{15} = P(\text{TSL}|\text{TL})P(\text{TSL}|\text{telephone})P(\text{telephone}|\text{TL}) + P(\text{TSL}|\text{no telephone})P(\text{no telephone}|\text{TL}) \quad (13.18)$$

13.2.3 Scoring Enhanced Plans

The plans of Section 13.2.2 were called "simple" because of their tree structure, complete ordering of actions, and the simple actions of their nodes. With a richer output from the symbolic planner, the plans may have different structure. For example, there may be *OR* nodes, any one of whose sons will achieve the action at the node; *AND* nodes, all of which must be satisfied (in any order) for the action to be satisfactorily completed; *SEQUENCE* nodes, which specify a set of actions and a particular order in which to achieve them. The plan may have loops, shared subgoal structure, or goals that depend on each other. How enhanced plans are interpreted and executed depends on the scoring algorithms, the possibilities of parallel execution, whether execution and scoring are interleaved, and so forth. This treatment ignores parallelism and limits discussion to expanding enhanced plans into simple ones.

It should be clear how to go about converting many of these enhanced plans to simple plans. For instance, sequence nodes simply go to a unique path of actions. Alternatively, depending on assumptions about outcomes of such actions (say whether they can fail), they may be coalesced into one action, as was the "threshold, find blobs, and compute shapes" action in the telephone-finding plan.

Rather more interesting are the OR and AND nodes, the order of whose subgoals is unspecified. Each such node yields many simple plans, depending on the order in which the subgoals are attacked. One way to score such a plan is to generate all possible simple plans and score each one, but perhaps it is possible to do better. For example, loops and mutual dependencies in plans can be dealt with in various ways. A loop can be analyzed to make sure that it contains an exit (such as a branch of an OR node that can be executed). One can make ad hoc assumptions that the cost of execution is always more than the cost of planning [Garvey 1976], and score the loop by its executable branch. Another idea is to plan incrementally with a finite horizon, expanding the plan through some progressive deepening, heuristic search, or pruning strategy. The accumulated cost of going around a loop will soon remove it from further consideration.

Recall (Figs. 13.4 and 13.5) that the expected utility of a plan was defined as the sum of the utility of each leaf node times the probability of reaching that node. However, the utilities need not combine linearly in scoring. Different monotonic functions of utility express such different conceptions as "aversion to risk" or "gambling addiction." These considerations are real ones, and nonlinear utilities are the rule rather than the exception. For instance, the value of money is notoriously nonlinear. Many people would pay \$5 for an even chance to win \$15; not so many people would pay \$5,000 for an even chance to win \$15,000.

One common way to compute scores based on utilities is the "cost/benefit" ratio. This, in the form "cost/confidence" ratio, is used by Garvey in his planning vision system. This measure is examined in Section 13.2.5; roughly, his "cost" was the effort in machine cycles to achieve goals, and his "confidence" approximated the probability of a goal achieving the correct outcome. The utility of correct outcomes was not explicitly encoded in his planner.

Sequential plan elaboration or partial plan elaboration can be interleaved with execution and scoring. Most practical planning is done in interaction with the world, and the plan scoring approach lends itself well to assessing such interactions. In Section 13.2.5 considers a planning vision system that uses enhanced plans and a limited replanning capability.

A thorny problem for decision making is to assess the cost of planning itself. The planning process is given its own utility (cost), and is carried only out as far as is indicated. Of course, the problem is in general infinitely recursive, since there is also the cost of assessing the cost of planning, etc. If, however, there is a known upper bound on the utility of the best achievable plan, then it is known that infinite planning could not improve it. This sort of reasoning is weaker than that needed to give the expected benefits of planning; it measures only the cost and maximum value of planning.

Another more advanced consideration is that the results of actions can be continuous and multidimensional, and discrete probabilities can be extended to probability distribution functions. Such techniques can reflect the precision of measurements.

An obviously desirable extension to a planner is a "learner," that can abstract rules for action applicability and remember successful plans. One approach would be to derive and remember ranges of planning parameters arising during execution; a range could be associated with a rule specifying appropriate action. This problem is difficult and the subject of current research.

13.2.4 Practical Simplifications

The expected utility calculations allow plans to be evaluated in a more or less "realistic" manner. However, in order to complete the calculations certain probabilities are necessary, and many of these reflect detailed knowledge about the interaction of phenomena in the world. It is thus often impractical to go about a full-blown treatment of scoring in the style of Section 13.2.2. This section presents some possible simplifications.

Of course, in many planning problems, such as those whose costs are nil or irrelevant, or all of whose goals are equally valuable, there is no need to address utility of plans at all. Such plans are typically not concerned with expenditure of real-world or planning resources.

Independence of various probabilities is one of the most helpful and pervasive assumptions in the calculation of probabilities. An example appeared in Section 13.2.2 with the table and telephone shape detectors.

Certain information can be ignored. Garvey [Garvey 1976] ignores failure information. His planning parameters include the "cost" of an action (strictly nega-

tive utilities reflecting effort), the probability of the action "succeeding," and the conditional probability that the state of the world is correctly indicated, given success. Related to ignoring some information is the assumption that certain outcomes are more reliable than others. For instance, the decision not to plan past "failure" reports means that they are assumed reliable.

Non-Bayesian rules of inference abound in planners [Shortliffe 1976]; the idea of assigning a single numerical utility score to plans is by no means the only way to make decisions.

13.2.5 A Vision System Based on Planning

Overview

This section outlines some features of a working vision system whose actions are controlled by the planning paradigm [Garvey 1976]. As with all large vision systems, more issues are addressed in this work than with the planning paradigm as a control mechanism. For one thing, the system uses multisensory input, including range and color information. An interactive facility aids in developing and testing low-level operators and "strategies" for object location. The machine-usable representation of knowledge about the objects in the scene domains and how they could be located is of course a central component.

The domain is office scenes (Fig. 13.6). For the task of locating different objects in such scenes, a "uniform strategy" is adopted. That is, the vision task is always broken down into a sequence of major goals to be performed in order. Such uniform strategies, if they are imposed on a system at all, tend to vary with different tasks, with different sensors or domain, or with different research goals.

Garvey's uniform strategy consists of the following steps.

1. *Acquire* some pixels thought to be in the desired region (the area of scene making up the image of the desired object).

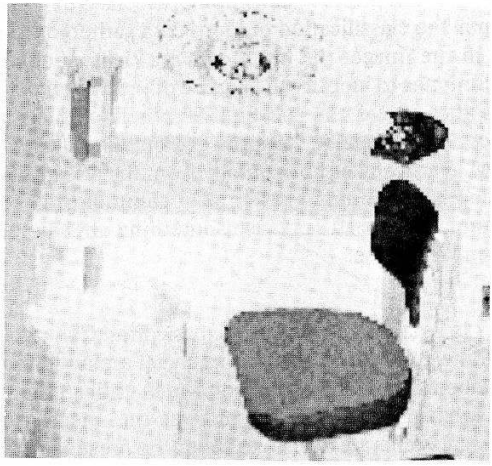


Fig. 13.6 The planning vision system uses input scenes such as these, imaged in different wavelengths and with a rangefinder.