CHAPTER
NINE

ROBOT PROGRAMMING LANGUAGES

Observers are not led by the same physical
evidence to the same picture of the
universe unless their linguistic backgrounds
are similar or can in some way be calibrated.
Benjamin Lee Whorf

9.1 INTRODUCTION

The discussion in the previous chapters focused on kinematics, dynamics, control, trajectory planning, sensing, and vision for computer-based manipulators. The algorithms used to accomplish these functions are usually embedded in the controlling software modules. A major obstacle in using manipulators as general-purpose assembly machines is the lack of suitable and efficient communication between the user and the robotic system so that the user can direct the manipulator to accomplish a given task. There are several ways to communicate with a robot, and three major approaches to achieve it are *discrete word recognition, teach and playback,* and *high-level programming languages.*

Current state-of-the-art speech recognition systems are quite primitive and generally speaker-dependent. These systems can recognize a set of discrete words from a limited vocabulary and usually require the user to pause between words. Although it is now possible to recognize discrete words in real time due to faster computer components and efficient processing algorithms, the usefulness of discrete word recognition to describe a robot task is quite limited in scope. Moreover, speech recognition generally requires a large memory or secondary storage to store speech data, and it usually requires a training period to build up speech templates for recognition.

Teach and playback, also known as *guiding,* is the most commonly used method in present-day industrial robots. The method involves teaching the robot by leading it through the motions the user wishes the robot to perform. Teach and playback is typically accomplished by the following steps: (1) leading the robot in slow motion using manual control through the entire assembly task and recording the joint angles of the robot at appropriate locations in order to replay the motion; (2) editing and playing back the taught motion; and (3) if the taught motion is correct, then the robot is run at an appropriate speed in a repetitive mode.

Leading the robot in slow motion usually can be achieved in several ways: using a joystick, a set of pushbuttons (one for each joint), or a master-slave mani-

pulator system. Presently, the most commonly used system is a manual box with pushbuttons. With this method, the user moves the robot manually through the space, and presses a button to record any desired angular position of the manipulator. The set of angular positions that are recorded form the set-points of the trajectory that the manipulator has traversed. These position set-points are then interpolated by numerical methods, and the robot is "played back" along the smoothed trajectory. In the edit-playback mode, the user can edit the recorded angular positions and make sure that the robot will not collide with obstacles while completing the task. In the run mode, the robot will run repeatedly according to the edited and smoothed trajectory. If the task is changed, then the above three steps are repeated. The advantages of this method are that it requires only a relatively small memory space to record angular positions and it is simple to learn. The main disadvantage is that it is difficult to utilize this method for integrating sensory feedback information into the control system.

High-level programming languages provide a more general approach to solving the human-robot communication problem. In the past decade, robots have been successfully used in such areas as arc welding and spray painting using guiding (Engelberger [1980]). These tasks require no interaction between the robot and the environment and can be easily programmed by guiding. However, the use of robots to perform assembly tasks requires high-level programming techniques because robot assembly usually relies on sensory feedback, and this type of unstructured interaction can only be handled by conditionally programmed methods.

Robot programming is substantially different from traditional programming. We can identify several considerations which must be handled by any robot programming method: The objects to be manipulated by a robot are three-dimensional objects which have a variety of physical properties; robots operate in a spatially complex environment; the description and representation of three-dimensional objects in a computer are imprecise; and sensory information has to be monitored, manipulated, and properly utilized. Current approaches to programming can be classified into two major categories: *robot-oriented programming* and *object-oriented*, or *task-level programming*.

In robot-oriented programming, an assembly task is explicitly described as a sequence of robot motions. The robot is guided and controlled by the program throughout the entire task with each statement of the program roughly corresponding to one action of the robot. On the other hand, task-level programming describes the assembly task as a sequence of positional goals of the objects rather than the motion of the robot needed to achieve these goals, and hence no explicit robot motion is specified. These approaches are discussed in detail in the following two sections.

9.2 CHARACTERISTICS OF ROBOT-LEVEL LANGUAGES

The most common approach taken in designing robot-level language is to extend an existing high-level language to meet the requirements of robot programming.

To a certain extent, this approach is ad hoc and there are no guidelines on how to implement the extension.

We can easily recognize several key characteristics that are common to all robot-oriented languages by examining the steps involved in developing a robot program. Consider the task of inserting a bolt into a hole (Fig. 9.1). This requires moving the robot to the feeder, picking up the bolt, moving it to the beam and inserting the bolt into one of the holes. Typically, the steps taken to develop the program are:

1. The workspace is set up and the parts are fixed by the use of fixtures and feeders.
2. The location (orientation and position) of the parts (*feeder*, *beam*, etc.) and their features (*beam_bore*, *bolt_grasp*, etc.) are defined using the data structures provided by the language.†
3. The assembly task is partitioned into a sequence of actions such as moving the robot, grasping objects, and performing an insertion.
4. Sensory commands are added to detect abnormal situations (such as inability to locate the bolt while grasping) and monitor the progress of the assembly task.

† The reader will recall that the use of the underscore symbol is a common practice in programming languages to provide an effective identity in a variable name and thus improve legibility.

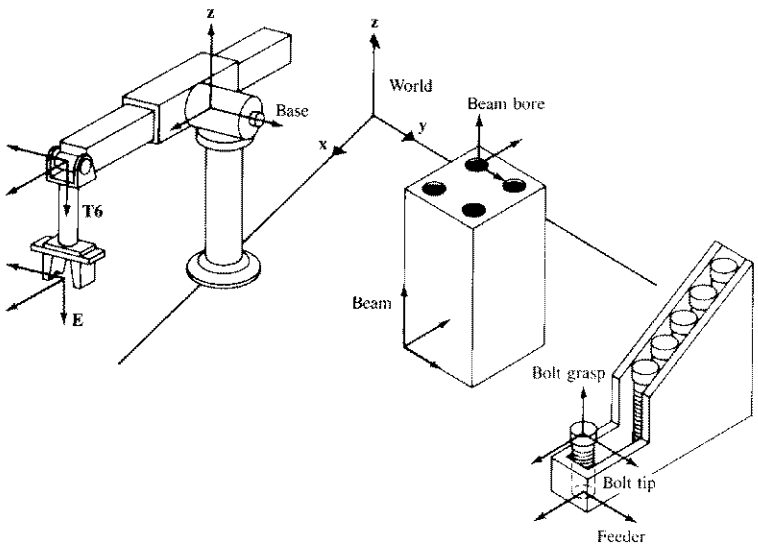


Figure 9.1 A simple robotic insertion task.

5. The program is debugged and refined by repeating steps 2 to 4.

The important characteristics we recognized are position specification (step 2), motion specification (step 3), and sensing (step 4). These characteristics are discussed in detail in this section.

We will use the languages AL (Mujtaba et al. [1982]) and AML (Taylor et al. [1983]) as examples. The choice of using these two languages is not arbitrary. AL has influenced the design of many robot-oriented languages and is still actively being developed. It provides a large set of commands to handle the requirements of robot programming and it also supports high-level programming features. AML is currently available as a commercial product for the control of IBM's robots and its approach is different from AL. Its design philosophy is to provide a system environment where different robot programming interfaces may be built. Thus, it has a rich set of primitives for robot operations and allows the users to design high-level commands according to their particular needs. These two languages represent the state of the art in robot-oriented programming languages. A brief description of the two languages is shown in Table 9.1.

Table 9.1 A brief summary of the AL and AML robot programming languages

AL was developed by Stanford University. Currently AL can be executed on a VAX computer and real-time control of the arms are performed on a stand alone PDP-11. Its characteristics are:

- High-level language with features of ALGOL and Pascal
- Supports both robot-level and task-level specification
- Compiled into low-level language and interpreted on a real time control machine
- Has real-time programming language constructs like synchronization, concurrent execution, and on-conditions
- ALGOL like data and control structure
- Support for world modeling

AML was developed by IBM. It is the control language for the IBM RS-1 robot. It runs on a Series-1 computer (or IBM personal computer) which also controls the robot. The RS-1 robot is a cartesian manipulator with 6 degrees of freedom. Its first three joints are prismatic and the last three joints are rotary. Its characteristics are:

- Provides an environment where different user-interface can be built
- Supports features of LISP-like and APL-like constructs
- Supports data aggregation
- Supports joint-space trajectory planning subject to position and velocity constraints
- Provides absolute and relative motions
- Provides sensor monitoring that can interrupt motion

Table 9.2 AL and AML definitions for base frames

AL:

base ← FRAME(nilrot, VECTOR(20, 0, 15)*inches);
beam ← FRAME(ROT(Z, 90*deg), VECTOR(20, 15, 0)*inches);
feeder ← FRAME(nilrot, VECTOR(25, 20, 0)*inches);

Notes: nilrot is a predefined frame which has value ROT(Z, 0*deg).
The "←" is the assignment operator in AL.
A semicolon terminates a statement.
The "*" is a type-dependent multiplication operator. Here, it is used to append units to the elements of the vector.

AML:

base = <<20, 0, 15>, EULERROT(<0, 0, 0>)>;
beam = <<20, 15, 0>, EULERROT(<0, 0, 90>)>;
feeder = <<25, 20, 0>, EULERROT(<0, 0, 0>)>;

Note: EULERROT is a subroutine which forms the rotation matrix given the angles.

9.2.1 Position Specification

In robot assembly, the robot and the parts are generally confined to a well-defined workspace. The parts are usually restricted by fixtures and feeders to minimize positional uncertainities. Assembly from a set of randomly placed parts requires vision and is not yet a common practice in industry.

The most common approach used to describe the orientation and the position of the objects in the workspace is by coordinate frames. They are usually represented as 4×4 homogeneous transformation matrices. A frame consists of a 3×3 submatrix (specifying the orientation) and a vector (specifying the position) which are defined with respect to some base frame. Table 9.2 shows AL and AML definitions for the three frames *base*, *beam*, and *feeder* shown in Fig. 9.1. The approach taken by AL is to provide predefined data structures for frames (FRAME), rotational matrices (ROT), and vectors (VECTOR), all of them in cartesian coordinates. On the other hand, AML provides a general structure called an *aggregate* which allows the user to design his or her own data structures. The AML frames defined in Table 9.2 are in cartesian coordinates and the format is <vector, matrix>, where vector is an aggregate of three scalars representing position and matrix is an aggregate of three vectors representing orientation.

In order to further explain the notation used in Table 9.2, the first statement in AL means the establishment of the coordinate frame *base*, whose principal axes are parallel (nilrot implies no rotation) to the principal axes of the reference frame and whose origin is at location (20, 0, 15) inches from the origin of the reference frame. The second statement in AL establishes the coordinate frame *beam*, whose principal axes are rotated 90° about the Z axis of the reference frame, and whose origin is at location (20, 15, 0) inches from the origin of the reference frame. The

third statement has the same meaning as the first, except for location. The meaning of the three statements in AML is exactly the same as for those in AL.

A convenient way of referring to the features of an object is to define a frame (with respect to the object's base frame) for it. An advantage of using a homogeneous transformation matrix is that defining frames relative to a base frame can be simply done by postmultiplying a transformation matrix to the base frame. Table 9.3 shows the AL and AML statements used to define the features *T6*, *E*, *bolt_tip*, *bolt_grasp*, and *beam_bore* with respect to their base frames, as indicated in Fig. 9.1. AL provides a matrix multiplication operator (*) and a data structure TRANS (a transformation which consists of a rotation and a translation operation) to represent transformation matrices. AML has no built-in matrix multiplication operator, but a system subroutine, DOT, is provided.

In order to illustrate the meaning of the statements in Table 9.3, the first AL statement means the establishment of the coordinate frame *T6*, whose principal axes are rotated 180° about the *X* axis of the *base* coordinate frame, and whose origin is at location (15, 0, 0) inches from the origin of the *base* coordinate frame. The second statement establishes the coordinate frame *E*, whose principal axes are parallel (nilrot implies no rotation) to the principal axes of the *T6* coordinate frame, and whose origin is at location (0, 0, 5) inches from the origin of the *T6* coordinate frame. Similar comments apply to the other three AL statements. The meaning of the AML statements is the same as those for AL.

Figure 9.2*a* shows the relationships between the frames we have defined in Tables 9.2 and 9.3. Note that the frames defined for the arm are not needed for AL because AL uses an implicit frame to represent the position of the end-effector and does not allow access to intermediate frames (*T6*, *E*). As parts are moved or

Table 9.3 AL and AML definitions for feature frames

AL:

```
T6 ← base * TRANS(ROT(X, 180*deg), VECTOR(15, 0, 0)*inches);
E ← T6 * TRANS(nilrot, VECTOR(0, 0, 5)*inches);
bolt_tip ← feeder * TRANS(nilrot, nilvect)
bolt_grasp ← bolt_tip * TRANS(nilrot, VECTOR(0, 0, 1)*inches);
beam_bore ← beam * TRANS(nilrot, VECTOR(0, 2, 3)*inches);
```

Note: nilvect is a predefined vector which has value VECTOR(0, 0, 0)*inches.

AML:

```
T6 = DOT(base, <<15, 0, 0>, EULERROT(<180, 0, 0>)>);
E = DOT(T6, <<0, 0, 5>, EULERROT(<0, 0, 0>)>);
bolt_tip = DOT(feeder, <<0, 0, 0>, EULERROT(<0, 0, 0>)>);
bolt_grasp = DOT(bolt_tip, <<0, 0, 1>, EULERROT(<0, 0, 0>)>);
beam_bore = DOT(beam, <<0, 2, 3>, EULERROT(<0, 0, 0>)>);
```

Note: DOT is a subroutine that multiplies two matrices.

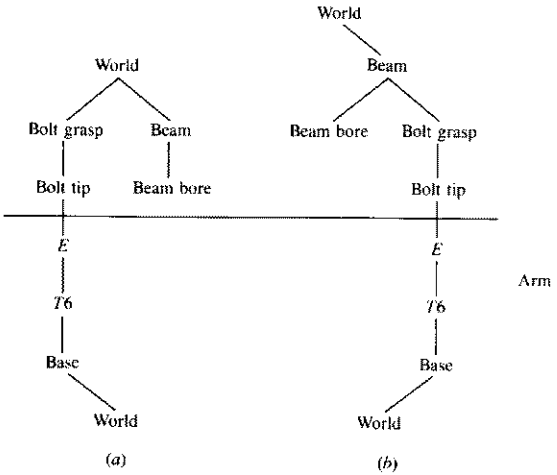


Figure 9.2 Relationships between the frames.

are attached to other objects, the frames are adjusted to reflect the current state of the world (see Fig. 9.2*b*).

Another way of acquiring the position and orientation of an object is by using the robot as a pointing device to gather the information interactively. POINTY (Grossman and Taylor [1978]), a system designed for AL, allows the user to lead the robot through the workspace (by hand or by a pendant) and, by pointing the hand (equipped with a special tool) to objects, it generates AL declarations similar to those shown in Tables 9.2 and 9.3. This eliminates the need to measure the distances and angles between frames, which can be quite tedious.

Although coordinate frames are quite popular for representing robot configurations, they do have some limitations. The natural way to represent robot configurations is in the joint-variable space rather than the cartesian space. Since the inverse kinematics problem gives nonunique solutions, the robot's configuration is not uniquely determined given a point in the cartesian space. As the number of features and objects increases, the relationships between coordinate frames become complicated and difficult to manage. Furthermore, the number of computations required also increases significantly.

9.2.2 Motion Specification

The most common operation in robot assembly is the pick-and-place operation. It consists of moving the robot from an initial configuration to a grasping configuration, picking up an object, and moving to a final configuration. The motion is usually specified as a sequence of positional goals for the robot to attain. However, only specifying the initial and final configurations is not sufficient. The

path is planned by the system without considering the objects in the workspace and obstacles may be present on the planned path. In order for the system to generate a collision-free path, the programmer must specify enough intermediate or *via* points on the path. For example, in Fig. 9.3, if a straight line motion were used from point *A* to point *C*, the robot would collide with the beam. Thus, intermediate point *B* must be used to provide a safe path.

The positional goals can be specified either in the joint-variable space or in the cartesian space, depending on the language. In AL, the motion is specified by using the MOVE command to indicate the destination frame the arm should move to. Via points can be specified by using the keyword "VIA" followed by the frame of the via point (see Table 9.4). AML allows the user to specify motion in the joint-variable space and the user can write his or her own routines to specify motions in the cartesian space. Joints are specified by joint numbers (1 through 6) and the motion can be either relative or absolute (see Table 9.4).

One disadvantage of this type of specification is that the programmer must preplan the entire motion in order to select the intermediate points. The resulting path may produce awkward and inefficient motions. Furthermore, describing a complex path as a sequence of points produces an unnecessarily long program.

As the robot's hand departs from its starting configuration or approaches its final configuration, physical constraints, such as an insertion, which require the hand to travel along an axis, and environmental constraints, such as moving in a crowded area, may prohibit certain movement of the robot. The programmer must have control over various details of the motion such as speed, acceleration, deceleration, approach and departure directions to produce a safe motion. Instead

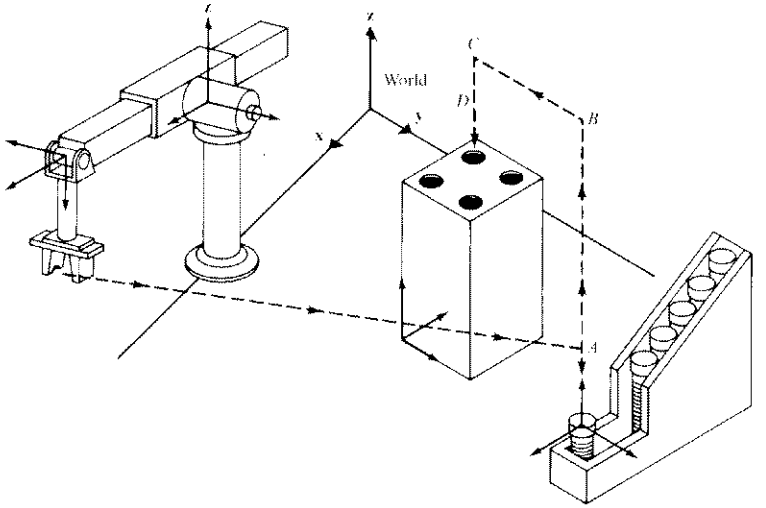


Figure 9.3 Trajectory of the robot.

Table 9.4 Examples of AL and AML motion statements

AL:

```
{ Move arm from rest to frame A and then to bolt_grasp }
MOVE barm TO A;
MOVE barm TO bolt_grasp;

{ Another way of specifying the above movement }
MOVE barm TO bolt_grasp VIA A;

{ Move along the current Z axis by 1 inch, i.e., move relative }
MOVE barm TO ⊗ − 1*Z*inches;
```

Notes: *barm* is the name of the robot arm.
⊗ indicates the current location of the arm which is equivalent to *base* * *T6* * *E*.
Statements inside brackets { · · · } are comments.

AML:

```
-- Move joint 1 and 4 to 10 inches and 20 degrees, respectively (absolute move)
MOVE(<1, 4>, <10, 20>);

-- Move joints 1, 3 and 6 by 1 inch, 2 inches, and 5 degrees, respectively (relative move)
DMOVE(<1, 3, 6>, <1, 2, 5>);
```

Notes: Statements preceeded by "--" are comments.

of separate commands, the usual approach is to treat them as constraints to be satisfied by the move command. AL provides a keyword "WITH" to attach constraint clauses to the move command. The constraints can be an approach vector, departure vector, or a time limit. Table 9.5 shows the AL statements for moving the robot from *bolt_grasp* to *A* with departure direction along $+Z$ of *feeder* and time duration of 5 seconds (i.e., move slowly). In AML, aggregates of the form <speed, acceleration, deceleration> can be added to the MOVE statement to specify speed, acceleration, and deceleration of the robot.

In general, gripper motions have to be tailored according to the environment and the task. Most languages provide simple commands on gripper motion so that sophisticated motions can be built using them. For a two-fingered gripper, one can either move the fingers apart (open) or move them together (close). Both AML and AL use a predefined variable to indicate the gripper (*bhand* corresponds to *barm* for AL and GRIPPER for AML). Using the OPEN (for AL) and MOVE (for AML) primitives, the gripper can be programmed to move to a certain opening (see Table 9.5).

9.2.3 Sensing and Flow of Control

The location and the dimension of the objects in the workspace can be identified only to a certain degree of accuracy. For the robot to perform tasks in the presence of these uncertainties, sensing must be performed. The sensory information gathered also acts as a feedback from the environment, enabling the robot to exam-

Table 9.5 Examples of AL and AML motion statements

AL:

```
{ Move arm from bolt_grasp to A }
MOVE barm TO A
WITH DEPARTURE = Z WRT feeder
WITH DURATION = 5*seconds;

{ Open the hand to 2.5 inches }
OPEN bhand TO 2.5*inches;
```

Note: WRT (means with respect to) generates a vector in the specified frame.

AML:

```
-- Move joint 1 and 4 to 10 inches and 20 degrees, respectively, with speed
1 inch/second,
-- Acceleration and deceleration 1 inch/second^2
MOVE(<1, 4>, <10, 20>, <1, 1, 1>);

-- Open the hand to 2.5 inches
MOVE(GRIPPER, 2.5);
```

ine and verify the state of the assembly. Sensing in robot programming can be classified into three types:

1. *Position sensing* is used to identify the current position of the robot. This is usually done by encoders that measure the joint angles and compute the corresponding hand position in the workspace.
2. *Force* and *tactile sensing* can be used to detect the presence of objects in the workspace. Force sensing is used in compliant motion to provide feedback for force-controlled motions. Tactile sensing can be used to detect slippage while grasping an object.
3. *Vision* is used to identify objects and provide a rough estimate of their position.

There is no general consensus on how to implement sensing commands, and each language has its own syntax. AL provides primitives like FORCE(axis) and TORQUE(axis) for force sensing. They can be specified as conditions like FORCE(Z) > 3*ounces in the control commands. AML provides a primitive called MONITOR which can be specified in the motion commands to detect asynchronous events. The programmer can specify the sensors to monitor and, when the sensors are triggered, the motion is halted (see Table 9.6). It also has position-sensing primitives like QPOSITION (joint numbers†) which returns the current position of the joints. Most languages do not explicitly support vision, and the user has to provide modules to handle vision information.

† Specified as an aggregate like <1, 5> which specifies joints 1 and 5.

Table 9.6 Force sensing and compliant motion

AL:

```
{ Test for presence of hole with force sensing }
MOVE barm TO ⊗ −1*Z*inches ON FORCE(Z) > 10*ounces
    DO ABORT("No Hole");

{ Insert bolt, exert downward force while complying side forces }
MOVE barm TO beam_bore
WITH FORCE(Z) = −10*ounces  WITH FORCE(X) = 0*ounces
WITH FORCE(Y) = 0*ounces  WITH DURATION = 3*seconds;
```

AML:

```
-- Define a monitor for the force sensors SLP and SLR; Monitor triggers if the sensor
-- values exceed the range 0 and F
fmons = MONITOR(<SLP, SRP>, 1, 0, F);

-- Move joint 3 by 1 inch and stop if fmons is triggered
DMOVE(<3>, <1>, fmons);
```

Note: The syntax for monitor is MONITOR(sensors, test type, limit1, limit2).

One of the primary uses of sensory information is to initiate or terminate an action. For example, a part arriving on a conveyor belt may trip an optical sensor and activate the robot to pick up the part, or an action may be terminated if an abnormal condition has occurred. Table 9.6 illustrates the use of force sensing information to detect whether the hand has positioned correctly above the hole. The robot arm is moved downward slightly and, as it descends, the force exerted on the hand along the Z axis of the hand coordinate frame is returned by FORCE(Z). If the force exceeds 10 ounces, then this indicates that the hand missed the hole and the task is aborted.

The flow of a robot program is usually governed by the sensory information acquired. Most languages provide the usual decision-making constructs like "if_then_else_", "case_", "do_until_", and "while_do_" to control the flow of the program under different conditions.

Certain tasks require the robot to comply with external constraints. For example, insertion requires the hand to move along one direction only. Any sideward forces may generate unwanted friction which would impede the motion. In order to perform this compliant motion, force sensing is needed. Table 9.6 illustrates the use of AL's force sensing commands to perform the insertion task with compliance. The compliant motion is indicated by quantifying the motion statement with the amount of force allowed in each direction of the hand coordinate frame. In this case, forces are applied only along the Z axis of this frame.

9.2.4 Programming Support

A language without programming support (editor, debugger, etc.) is useless to the user. A sophisticated language must provide a programming environment that

allows the user to support it. Complex robot programs are difficult to develop and can be difficult to debug. Moreover, robot programming imposes additional requirements on the development and debugging facilitates:

1. *On-line modification and immediate restart.* Since robot tasks requires complex motions and long execution time, it is not always feasible to restart the program upon failure. The robot programming system must have the ability to allow programs to be modified on-line and restart at any time.
2. *Sensor outputs and program traces.* Real-time interactions between the robot and the environment are not always repeatable; the debugger should be able to record sensor values along with program traces.
3. *Simulation.* This feature allows testing of programs without actually setting up robot and workspace. Hence, different programs can be tested more efficiently.

The reader should realize by now that programming in a robot-oriented language is tedious and cumbersome. This is further illustrated by the following example.

Example: Table 9.7 shows a complete AL program for performing the insertion task shown diagramatically in Fig. 9.1. The notation and meaning of the statements have already been explained in the preceding discussion. Keep in mind that a statement is not considered terminated until a semicolon is encountered. □

Table 9.7 An AL program for performing an insertion task

```
BEGIN insertion

{ set the variables }
bolt_diameter ← 0.5*inches;
bolt_height ← 1*inches;
tries ← 0;
grasped ← false;
{ Define base frames }
beam ← FRAME(ROT(Z, 90*deg), VECTOR(20, 15, 0)*inches);
feeder ← FRAME(nilrot, VECTOR(25, 20, 0)*inches);

{ Define feature frames }
bolt_grasp ← feeder * TRANS(nilrot, nilvect);
bolt_tip ← bolt_grasp * TRANS(nilrot, VECTOR(0, 0, 0.5)*inches);
beam_bore ← beam * TRANS(nilrot, VECTOR(0, 0, 1)*inches);

{ Define via points frames }
A ← feeder * TRANS(nilrot, VECTOR(0, 0, 5)*inches);
B ← feeder * TRANS(nilrot, VECTOR(0, 0, 8)*inches);
C ← beam_bore * TRANS(nilrot, VECTOR(0, 0, 5)*inches);
D ← beam_bore * TRANS(nilrot, bolt_height*Z);
```

Table 9.7 (continued)

```
{ Open the hand }
OPEN bhand TO bolt_diameter + 1*inches;

{ Position the hand just above the bolt }
MOVE barm TO bolt_grasp VIA A
    WITH APPROACH = -Z WRT feeder;

{ Attempt to grasp the bolt }
DO
    CLOSE bhand TO 0.9*bolt_diameter;
    IF bhand < bolt_diameter THEN BEGIN{ failed to grasp the bolt, try again }
    OPEN bhand TO bolt_diameter + 1*inches;
    MOVE barm TO ⊗ - 1*Z*inches;
    END ELSE grasped ← true;
    tries ← tries + 1;
UNTIL grasped OR (tries > 3);

{ Abort the operation if the bolt is not grasped in three tries. }
IF NOT grasped THEN ABORT("failed to grasp bolt");

{ Move the arm to B }
MOVE barm TO B
    VIA A
    WITH DEPARTURE = Z WRT feeder;

{ Move the arm to D }
MOVE barm TO D VIA C
    WITH APPROACH = -Z WRT beam_bore;

{ Check whether the hole is there }
MOVE barm TO ⊗ - 0.1*Z*inches ON FORCE(Z) > 10*ounces
    DO ABORT("No hole");

{ Do insertion with compliance }
MOVE barm TO beam_bore DIRECTLY
    WITH FORCE(Z) = -10*ounces
    WITH FORCE(X) = 0*ounces
    WITH FORCE(Y) = 0*ounces
    WITH DURATION = 5*seconds;

END insertion.
```

9.3 CHARACTERISTICS OF TASK-LEVEL LANGUAGES

A completely different approach in robot programming is by task-level programming. The natural way to describe an assembly task is in terms of the objects

being manipulated rather than by the robot motions. Task-level languages make use of this fact and simplify the programming task.

A task-level programming system allows the user to describe the task in a high-level language (task specification); a task planner will then consult a database (world models) and transform the task specification into a robot-level program (robot program synthesis) that will accomplish the task. Based on this description, we can conceptually divide task planning into three phases: world modeling, task specification, and program synthesis. It should be noted that these three phases are not completely independent, in fact, they are computationally related.

Figure 9.4 shows one possible architecture for the task planner. The task specification is decomposed into a sequence of subtasks by the task decomposer and information such as initial state, final state, grasping position, operand, specifications, and attachment relations are extracted. The subtasks then pass through the subtask planner which generates the required robot program.

The concept of task planning is quite similar to the idea of automatic program generation in artificial intelligence. The user supplies the input-output requirements of a desired program, and the program generator then generates a program that will produce the desired input-output behavior (Barr et al. [1981, 1982]).

Task-level programming, like automatic program generation, is, in the research stage with many problems still unsolved. In the remaining sections we will discuss the problems encountered in task planning and some of the solutions that have been proposed to solve them.

9.3.1 World Modeling

World modeling is required to describe the geometric and physical properties of the objects (including the robot) and to represent the state of the assembly of objects in the workspace.

Geometric and Physical Models. For the task planner to generate a robot program that performs a given task, it must have information about the objects and the robot itself. These include the geometric and physical properties of the objects which can be represented by models.

A geometric model provides the spatial information (dimension, volume, shape) of the objects in the workspace. As discussed in Chap. 8, numerous techniques exist for modeling three-dimensional objects (Baer et al. [1979], Requicha [1980]). The most common approach is constructive solid geometry (CSG), where objects are defined as constructions or combinations, using regularized set operations (such as union, intersection), of primitive objects (such as cube, cylinder). The primitives can be represented in various ways:

1. A set of edges and points
2. A set of surfaces
3. Generalized cylinders
4. Cell decomposition

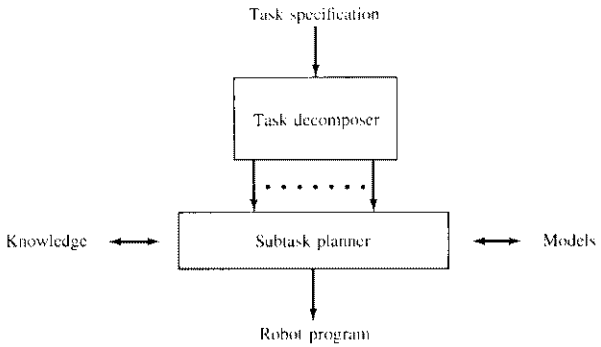


Figure 9.4 Task planner.

In the AUTOPASS system (Lieberman and Wesley [1977]), objects are modeled by utilizing a modeling system called GDP (geometric design processor) (Wesley et al. [1980]) which uses a procedural representation to describe objects. The basic idea is that each object is represented by a procedure name and a set of parameters. Within this procedure, the shape of the object is defined by calls to other procedures representing other objects or set operations.

GDP provides a set of primitive objects (all of them are polyhedra) which can be cuboid, cylinder, wedge, cone, hemisphere, laminum, and revolute. These primitives are internally represented as a list of surfaces, edges, and points which are defined by the parameters in the corresponding procedure. For example,

CALL SOLID(CUBOID, "Block", xlen, ylen, zlen);

will invoke the procedure SOLID to define a rectangular box called Block with dimensions xlen, ylen, and zlen. More complicated objects can then be defined by calling other procedures and applying the MERGE subroutine to them. Table 9.8 shows a description of the bolt used in the insertion task discussed in Sec. 9.2.

Physical properties such as inertia, mass, and coefficient of friction may limit the type of motion that the robot can perform. Instead of storing each of the properties explicitly, they can be derived from the object model. However, no model can be 100 percent accurate and identical parts may have slight differences in their physical properties. To deal with this, tolerances must be introduced into the model (Requicha [1983]).

Representing World States. The task planner must be able to stimulate the assembly steps in order to generate the robot program. Each assembly step can be succinctly represented by the current state of the world. One way of representing these states is to use the configurations of all the objects in the workspace.

AL provides an attachment relation called AFFIX that allows frames to be attached to other frames. This is equivalent to physically attaching a part to

Table 9.8 GDP description of a bolt

```
Bolt: PROCEDURE(shaft_height, shaft_radius, shaft_nfacets, head_height, head_radius,
                head_nfacets);

/* define parameters */
DECLARE
    shaft_height,     /* height of shaft */
    head_height,      /* height of head */
    shaft_radius,     /* radius of shaft */
    head_radius,      /* radius of head */
    shaft_nfacets,    /* number of shaft faces */
    head_nfacets,     /* number of head faces */

/* specify floating point for above variables */
FLOAT;

/* define shape of the shaft */
CALL SOLID(CYLIND, "Shaft", shaft_height, shaft_radius, shaft_nfacets);

/* define shape of head */
CALL SOLID(CYLIND,"Head", head_height, head_radius, head_nfacets);

/* perform set union to get bolt */
CALL MERGE("Shaft", "Head", union);

END Bolt.
```

Note: The notation /* · · · */ indicates a comment.

another part and if one of the parts moves, the other parts attached will also move. AL automatically updates the locations of the frames by multiplying the appropriate transformations. For example,

```
AFFIX beam_bore TO beam RIGIDLY;
beam_bore = FRAME(nilrot, VECTOR(1,0,0)*inches);
```

describes that the frame *beam_bore* is attached to the frame *beam*.

AUTOPASS uses a graph to represent the world state. The nodes of the graph represents objects and the edges represent relationships. The relations can be one of:

1. *Attachment*. An object can be rigidly, nonrigidly, or conditionally attached to another object. The first two of these have a function similar to the AFFIX statement in AL. Conditionally attachment means that the object is supported by the gravity (but not strictly attached).
2. *Constraints*. Constraint relationships represent physical constraints between objects which can be translational or rotational.
3. *Assembly component*. This is used to indicate that the subgraph linked by this edge is an assembly part and can be referenced as an object.

As the assembly proceeds, the graph is updated to reflect the current state of the assembly.

9.3.2 Task Specification

Task specification is done with a high-level language. At the highest level one would like to have natural languages as the input, without having to give the assembly steps. An entire task like building a water pump could then be specified by the command "build water pump." However, this level of input is still quite far away. Not even omitting the assembly sequence is possible. The current approach is to use an input language with a well-defined syntax and semantics, where the assembly sequence is given.

An assembly task can be described as a sequence of states of the world model. The states can be given by the configurations of all the objects in the workspace, and one way of specifying configurations is to use the spatial relationships between the objects. For example, consider the block world shown in Fig. 9.5. We define a spatial relation AGAINST to indicate that two surfaces are touching each other. Then the statements in Table 9.9 can be used to describe the two situations depicted in Fig. 9.5. If we assume that state A is the initial state and state B is the goal state, then they can be used to represent the task of picking up *Block3* and placing it on top of *Block2*. If state A is the goal state and state B is the initial state, then they would represent the task of removing *Block3* from the stack of blocks and placing it on the table. The advantage of using this type of representation is that they are easy to interpret by a human, and therefore, easy to specify and modify. However, a serious limitation of this method is that it does not specify all the necessary information needed to describe an operation. For example, the torque required to tighten a bolt cannot be incorporated into the state description.

An alternate approach is to describe the task as a sequence of symbolic operations on the objects. Typically, a set of spatial constraints on the objects are also given to eliminate any ambiguity. This form of description is quite similar to those used in an industrial assembly sheet. Most robot-oriented languages have adopted this type of specification.

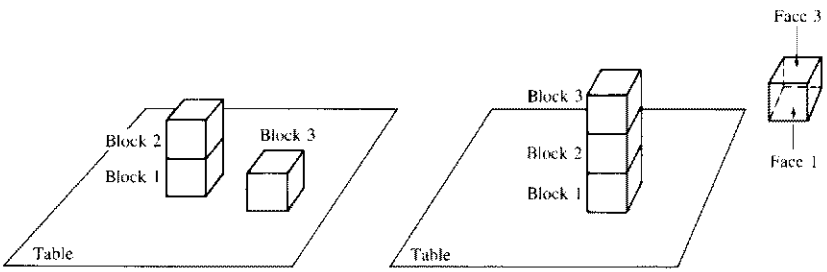


Figure 9.5 Block world.

Table 9.9 State description of block world

State A:	State B:
(*Block1_face1* AGAINST *table*)	(*Block1_face1* AGAINST *Table*)
(*Block1_face3* AGAINST *Block2_face1*)	(*Block1_face3* AGAINST *Block2_face1*)
(*Block3_face1* AGAINST *Table*)	(*Block2_face3* AGAINST *Block3_face1*)

AL provides a limited way of describing a task using this method. With the AFFIX statements, an object frame can be attached to *barm* to indicate that the hand is holding the object. Then moving the object to another point can be described by moving the object frame instead of the arm. For example, the inserting process in Fig. 9.1 can be specified as

AFFIX *bolt_tip* TO *barm;*
MOVE *bolt_tip* TO *beam_bore;*

Popplestone et al. [1978] have proposed a language called RAPT which uses contact relations AGAINST, FIT, and COPLANAR to specify the relationship between object features. Object features, which can be planar or spherical faces, cylindrical shafts and holes, edges, and vertices, are defined by coordinate frames similar to those used in AL. For example, the two operations in the block world example can be described as:

PLACE *Block3* SO THAT (*Block2_face3* AGAINST *Block3_face1*)
PLACE *Block3* SO THAT (*Block3_face1* AGAINST *Table*)

The spatial relationships are then extracted and solved for the configuration constraints on the objects required to perform the task.

AUTOPASS also uses this type of specification but it has a more elaborate syntax. It divides its assembly related statements into three groups:

1. *State change statement*: Describes an assembly operation such as placement and adjustment of parts.
2. *Tools statement*: Describes the type of tools to use.
3. *Fastener statement*: Describes a fastening operation.

The syntax of these statements is complicated (see Table 9.10). For example,

PLACE *bolt* ON *beam* SUCH THAT *bolt_tip* IS ALIGNED WITH *beam_bore*;
DRIVE IN *bolt* AT *bolt_grasp* SUCH THAT TORQUE IS EQ 12.0 IN-LBS USING *air_driver*;

would be used to describe the operation of inserting a bolt and tightening it.

Table 9.10 The syntax of the state change and tool statements in AUTOPASS

State change statement

PLACE <object> <preposition> <object> <grasping> <final-condition> <constraint> <then-hold>

where

<object>	Is a symbolic name for the object.
<preposition>	Is either IN or ON; it is used to determine the type of operation.
<grasping>	Specifies how the object should be grasped.
<constraint>	Specifies the constraints to be met during the execution of the command.
<then-hold>	Indicates that the hand is to remain in position on completion of the command.

Tool statement

OPERATE <tool> <load-list> <at-position> <attachment> <final-condition> <tool-parameters> <then-hold>

where

<tool>	Specifies the tool to be used.
<load-list>	Specifies the list of accessories.
<at-position>	Specifies where the tool is to be operated.
<attachment>	Specifies new attachment.
<final-condition>	Specifies the final condition to be satisfied at the completion of the command.
<tool-parameters>	Specifies tool operation parameters such as direction of rotation and speed.
<then-hold>	Indicates that the hand is to remain in position on completion of the command.

9.3.3 Robot Program Synthesis

The synthesis of a robot program from a task specification is one of the most important and most difficult phases of task planning. The major steps in this phase are grasping planning, motion planning, and plan checking. Before the task planner can perform the planning, it must first convert the symbolic task specification into a usable form. One approach is to obtain configuration constraints from the symbolic relationships. The RAPT interpreter extracts the symbolic relationships and forms a set of matrix equations with the constraint parameters of the objects as unknowns. These equations are then solved symbolically by using a set of rewrite rules to simplify them. The result obtained is a set of constraints on the configurations of each object that must be satisfied to perform the operation.

Grasping planning is probably the most important problem in task planning because the way the object is grasped affects all subsequent operations. The way

the robot can grasp an object is constrained by the geometry of the object being grasped and the presence of other objects in the workspace. A usable grasping configuration is one that is reachable and stable. The robot must be able to reach the object without colliding with other objects in the workspace and, once grasped, the object must be stable during subsequent motions of the robot.

Typically, the method used to choose a grasp configuration is a variation of the following procedure:

1. A set of candidate grasping configurations are chosen based on:
   Object geometry (e.g., for a parallel-jaw gripper, a good place to grasp is on either side of parallel surfaces).
   Stability (one heuristic is to have the center of mass of the object lie within the fingers).
   Uncertainty reduction.
2. The set is then pruned according to whether they:
   Are reachable by the robot.
   Would lead to collisions with other objects.
3. The final configuration is selected among the remaining configurations (if any) such that:
   It would lead to the most stable grasp.
   It would be the most unlikely to have a collision in the presence of position errors.

Most of the current methods for grasp planning focus only on finding reachable grasping positions, and only a subset of constraints are considered. Grasping in the presence of uncertainties is more difficult and often involves the use of sensing.

After the object is grasped, the robot must move the object to its destination and accomplish the operation. This motion can be divided into four phases:

1. A guarded departure from the current configuration
2. A free motion to the desired configuration without collision
3. A guarded approach to the destination
4. A compliant motion to achieve the goal configuration

One of the important problems here is planning the collision-free motion. Several algorithms have been proposed for planning collision-free path and they can be grouped into three classes:

1. *Hypothesis and test.* In this method, a candidate path is chosen and the path is tested for collision at a set of selected configurations. If a collision occurs, a correction is made to avoid the collision (Lewis and Bejczy [1973]). The main advantage of this method is its simplicity and most of the tools needed are already available in the geometric modeling system. However, generating the correction is difficult, particularly when the workspace is clustered with obstacles.

2. *Penalty functions.* This method involves defining penalty functions whose values depend on the proximity of the obstacles. These functions have the characteristic that, as the robot gets closer to the obstacles, their values increase. A total penalty function is computed by adding all the individual penalty functions and possibly a penalty term relating to minimum path. Then the derivatives of the total penalty function with respect to the configuration parameters are estimated and the collision-free path is obtained by following the local minima of the total penalty function. This method has the advantage that adding obstacles and constraints is easy. However, the penalty functions generally are difficult to specify.
3. *Explicit free space.* Several algorithms have been proposed in this class. Lozano-Perez [1982] proposed to represent the free space (space free of obstacles) in terms of the robot's configuration (configuration space). Conceptually, the idea is equivalent to transforming the robot's hand holding the object into a point, and expanding the obstacles in the workspace appropriately. Then, finding a collision-free path amounts to finding a path that does not intersect any of the expanded obstacles. This algorithm performs reasonably well when only translation is considered. However, with rotation, approximations must be made to generate the configuration space and the computations required increase significantly. Brooks [1983*a*, 1983*b*] proposed another method by representing the free space as overlapping generalized cones and the volume swept by the moving object as a function of its orientation. Then, finding the collision-free path reduces to comparing the swept volume of the object with the swept volume of the free space.

Generating the compliant motion is another difficult and important problem. Current work has been based on using the task kinematics to constraint the legal robot configurations to lie on a C-surface† (Mason [1981]) in the robot's configuration space. Then generating compliant motions is equivalent to finding a hybrid position/force control strategy that guarantees the path of the robot to stay on the C surface.

9.4 CONCLUDING REMARKS

We have discussed the characteristics of robot-oriented languages and task-level programming languages. In robot-oriented languages, an assembly task is explicitly described as a sequence of robot motions. The robot is guided and controlled by the program throughout the entire task with each statement of the program roughly corresponding to one action of the robot. On the other hand, task-level

† A C-surface is defined on a C-frame. It is a task configuration which allows only partial freedom in position. Along its tangent is the positional freedom and along its normal is the force freedom. A C-frame is an orthogonal coordinate system in the cartesian space. The frame is so chosen that the task freedoms are defined to be translation along and rotation about each of the three principal axes.

Table 9.11*a* Comparison of various existing robot control languages

Language	AL	AML	AUTOPASS	HELP	JARS	MAPLE
Institute	Stanford	IBM	IBM	GE	JPL	IBM
Robot controlled	PUMA Stanford Arm	IBM Arm	IBM	Allegro	PUMA Stanford Arm	IBM
Robot-or object-level	Mix	Robot	Object	Robot	Robot	Robot
Language basis	Concurrent Pascal	Lisp, APL, Pascal	PL/I	Pascal	Pascal	PL/I
Compiler or interpreter	Both	Interpreter	Both	Interpreter	Compiler	Interpreter
Geometric data type	Frame	Aggregate	Model	None	Frame	None
Motion specified by	Frame	Joints	Implicit	Joints	Joints, frame	Translation, rotation
Control structure	Pascal	Pascal	PL/I	Pascal	Pascal	PL/I
Sensing command	Position, force	Position	Force, tactile	Force, vision	Proximity, vision	Force, proximity
Parallel processing	COBEGIN, semaphores	None	IN PARALLEL	Semaphores	None	IN PARALLEL
Multiple robot	Yes	No	No	Yes	No	Yes
References	1	2	3	4	5	6

1. Mujtaba et al. [1982].
2. Taylor et al. [1983].
3. Lieberman and Wesley [1977].
4. *Automation Systems A12 Assembly Robot Operator's Manual*, P50VE025, General Electric Co., Bridgeport, Conn., February 1982.
5. Craig [1980].
6. Darringer and Blasgen [1975].

languages describe the assembly task as a sequence of positional goals of the objects rather than the motion of the robot needed to achieve these goals, and hence no explicit robot motion is specified. Two existing robot programming languages, AL and AML, were used to illustrate the characteristics of robot-oriented languages. We conclude that a robot-oriented language is difficult to use because it requires the user to program each detailed robot motion in completing a task. Task-level languages are much easier to use. However, many problems in task-level languages, such as task planning, object modeling, obstacle avoidance,

Table 9.11*b*

Language	MCL	PAL	RAIL	RPL	VAL
Institute	McDonnell Douglas	Purdue	Automatix	SRI	Unimate
Robot controlled	Cincinnati Milacron T-3	Stanford Arm	Custom-designed Cartesian arm	PUMA	PUMA
Robot-or object-level	Robot	Robot	Robot	Robot	Robot
Language basis	APT	Transform base	Pascal	Fortran, Lisp	Basic
Complier or interpreter	Compiler	Interpreter	Interpreter	Both	Interpreter
Geometric data type	Frame	Frame	Frame	None	Frame
Motion specified by	Translation, rotation	Frame	Joints, frame	Joints	Joints, frame
Control structure	If-then-else while-do	If-then-else	Pascal	Fortran	If-then
Sensing command	Position	Force	Force, vision	Position, vision	Position, force
Parallel processing	INPAR	None	None	None	Semaphores
Multiple robot	Yes	No	No	No	No
References	1	2	3	4	5

1. Oldroyd [1981].
2. Takase et al. [1981].
3. Franklin and Vanderbrug [1982].
4. Park [1981].
5. *User's Guide to VAL*, version 11, second edition, Unimation, Inc., Danbury, Conn., 1979.

trajectory planning, sensory information utilization, and grasping configurations, must be solved before they can be used effectively. We conclude this chapter with a comparison of various languages, as shown in Table 9.11*a* and *b*.

REFERENCES

Further reading in robot-level programming can be found in Bonner and Shin [1982], Geschke [1983], Gruver et al. [1984], Lozano-Perez [1983a], Oldroyd [1981], Park [1981], Paul [1976, 1981], Popplestone et al. [1978, 1980], Shimano [1979], Synder [1985], Takase et al. [1981], and Taylor et al. [1983]. Further

reading in task-level programming can be found in Binford [1979], Darringer and Blasgen [1975], Finkel et al. [1975], Lieberman and Wesley [1977], and Mujtaba et al. [1982]. Languages for describing objects can be found in Barr et al. [1981, 1982], Grossman and Taylor [1978], Lieberman and Wesley [1977], and Wesley et al. [1980]. Takase et al. [1981] presented a homogeneous transformation matrix equation in describing a task sequence to a manipulator.

Various obstacle avoidance algorithms embedded in the programming languages can be found in Brooks [1983*a*, 1983*b*], Brooks and Lozano-Perez [1983], Lewis and Bejczy [1973], Lozano-Perez [1983*a*], Lozano-Perez and Wesley [1979]. In task planning, Lozano-Perez [1982, 1983*b*] presented a configuration space approach for moving an object through a crowded workspace.

Future robot programming languages will incorporate techniques in artificial intelligence (Barr et al. [1981, 1982]) and utilize "knowledge" to perform reasoning (Brooks [1981]) and planning for robotic assembly and manufacturing.

PROBLEMS

9.1 Write an AL statement for defining a coordinate frame *grasp* which can be obtained by rotating the coordinate frame *block* through an angle of 65° about the Y axis and then translating it 4 and 6 inches in the X and Y axes, respectively.

9.2 Repeat Prob. 9.1 with an AML statement.

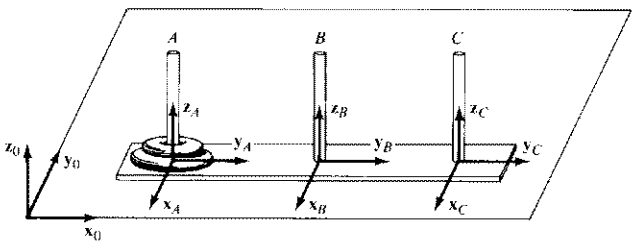
9.3 Write an AL program to palletize nine parts from a feeder to a tray consisting of a 3 × 3 array of bins. Assume that the locations of the feeder and tray are known. The program has to index the location for each pallet and signal the user when the tray is full.

9.4 Repeat Prob. 9.3 with an AML program.

9.5 Repeat Prob. 9.3 with a VAL program.

9.6 Repeat Prob. 9.3 with an AUTOPASS program.

9.7 *Tower of Hanoi problem.* Three pegs, A, B, and C, whose coordinate frames are, respectively, (x_A, y_A, z_A), (x_B, y_B, z_B), and (x_C, y_C, z_C), are at a known location from the reference coordinate frame (x_0, y_0, z_0), as shown in the figure below. Initially, peg A has two disks of different sizes, with disks having smaller diameters always on the top of disks with larger diameters. You are asked to write an AL program to control a robot equipped with a special suction gripper (to pick up the disks) to move the two disks from peg A to peg C so that at any instant of time disks of smaller diameters are always on the top of disks with larger diameters. Each disk has an equal thickness of 1 inch.



9.8 Repeat Prob. 9.7 with an AML program.

CHAPTER
TEN

ROBOT INTELLIGENCE AND TASK PLANNING

That which is apprehended by intelligence and
reason is always in the same state; but
that which is conceived by opinion with the
help of sensation and without reason, is
always is a process of becoming and
perishing and never really is.
Timaeus, in the "Dialogues of Plato"

10.1 INTRODUCTION

A basic problem in robotics is *planning* motions to solve some prespecified task, and then *controlling* the robot as it executes the commands necessary to achieve those actions. Here, planning means deciding on a course of action before acting. This action synthesis part of the robot problem can be solved by a problem-solving system that will achieve some stated goal, given some initial situation. A plan is, thus, a representation of a course of action for achieving the goal.

Research on robot problem solving has led to many ideas about problem-solving systems in artificial intelligence. In a typical formulation of a robot problem we have a robot that is equipped with sensors and a set of primitive actions that it can perform in some easy-to-understand world. Robot actions change one state, or configuration, of the world into another. In the "blocks world," for example, we imagine a world of several labeled blocks resting on a table or on each other and a robot consisting of a TV camera and a moveable arm and hand that is able to pick up and move blocks. In some problems the robot is a mobile vehicle with a TV camera that performs tasks such as pushing objects from place to place through an environment containing other objects.

In this chapter, we briefly introduce several basic methods in problem solving and their applications to robot planning.

10.2 STATE SPACE SEARCH

One method for finding a solution to a problem is to try out various possible approaches until we happen to produce the desired solution. Such an attempt involves essentially a trial-and-error search. To discuss solution methods of this sort, it is helpful to introduce the notion of problem states and operators. *A prob-*

lem state, or simply *state*, is a particular problem situation or configuration. The set of all possible configurations is the space of problem states, or the *state space*. An *operator*, when applied to a state, transforms the state into another state. A *solution* to a problem is a sequence of operators that transforms an initial state into a goal state.

It is useful to imagine the space of states reachable from the initial state as a graph containing nodes corresponding to the states. The nodes of the graph are linked together by arcs that correspond to the operators. A solution to a problem could be obtained by a search process that first applies operators to the initial state to produce new states, then applies operators to these, and so on until the goal state is produced. Methods of organizing such a search for the goal state are most conveniently described in terms of a graph representation.

10.2.1 Introductory Examples

Before proceeding with a discussion of graph search techniques, we consider briefly some basic examples as a means of introducing the reader to the concepts discussed in this chapter.

Blocks World. Consider that a robot's world consists of a table T and three blocks, A, B, and C. The initial state of the world is that blocks A and B are on the table, and block C is on top of block A (see Fig. 10.1). The robot is asked to change the initial state to a goal state in which the three blocks are stacked with block A on top, block B in the middle, and block C on the bottom. The only operator that the robot can use is MOVE X from Y to Z, which moves object X from the top of object Y onto object Z. In order to apply the operator, it is required that (1) X, the object to be moved, be a block with nothing on top of it, and (2) if Z is a block, there must be nothing on it.

We can simply use a graphical description like the one in Fig. 10.1 as the state representation. The operator MOVE X from Y to Z is represented by MOVE(X,Y,Z). A graph representation of the state space search is illustrated in Fig. 10.2. If we remove the dotted lines in the graph (that is, the operator is not to be used to generate the same operation more than once), we obtain a state space

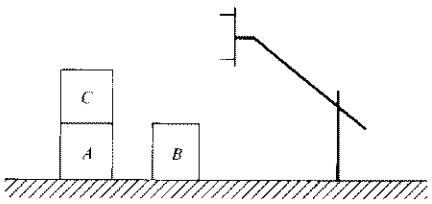


Figure 10.1 A configuration of robot and blocks.

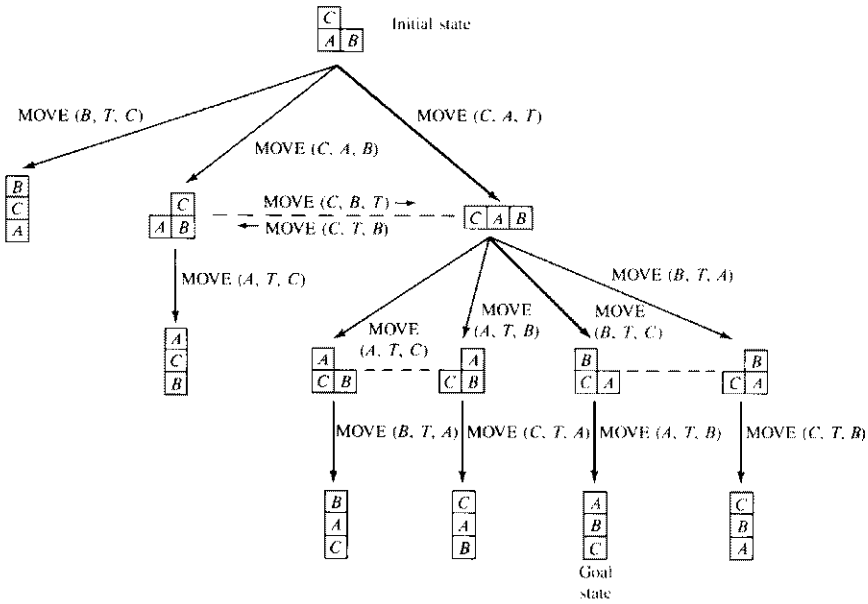


Figure 10.2 State space search graph.

search tree. It is easily seen from Fig. 10.2 that a solution that the robot can obtain consists of the following operator sequence:

$$\text{MOVE}(C,A,T),\ \text{MOVE}(B,T,C),\ \text{MOVE}(A,T,B)$$

Path Selection. Suppose that we wish to move a long thin object A through a crowded two-dimensional environment as shown in Fig. 10.3. To map motions of the object once it is grasped by a robot arm, we may choose the state space representation (x, y, α) where

$$x = \text{horizontal coordinate of the object} \qquad 1 \leqslant x \leqslant 5$$

$$y = \text{vertical coordinate of the object} \qquad 1 \leqslant y \leqslant 3$$

$$\alpha = \text{orientation of the object}$$

$$\alpha = \begin{cases} 0 & \text{if object } A \text{ is parallel to } x \text{ axis} \\ 1 & \text{if object } A \text{ is parallel to } y \text{ axis} \end{cases}$$

Both position and orientation of the object are quantized. The operators or robot

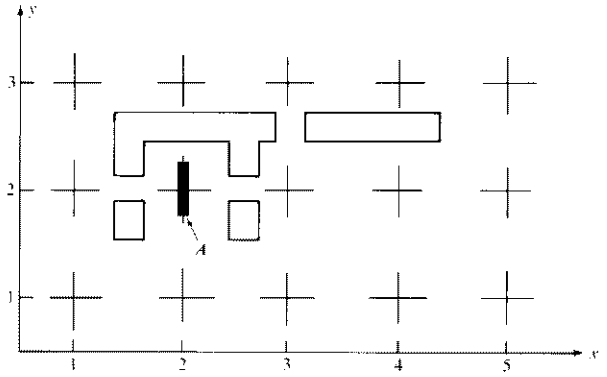


Figure 10.3 Physical space for Example 2.

commands are:

MOVE ± x direction one unit
MOVE ± y direction one unit
ROTATE 90°

The state space appears in Fig. 10.4. We assume for illustration that each "move" is of length 2, and each "rotate" of length 3. Let the object A be initially at location (2,2), oriented parallel to the y axis, and the goal is to move A to (3,3) and oriented parallel to the x axis. Thus the initial state is (2,2,1) and the goal state is (3,3,0).

There are two equal-length solution paths, shown in Fig. 10.5 and visualized on a sketch of the task site in Fig. 10.6. These paths may not look like the most direct route. Closer examination, however, reveals that these paths, by initially moving the object away from the goal state, are able to save two rotations by utilizing a little more distance.

Monkey-and-Bananas Problem. A monkey† is in a room containing a box and a bunch of bananas (Fig. 10.7). The bananas are hanging from the ceiling out of reach of the monkey. How can the monkey get the bananas?
The four-element list (W,x,Y,z) can be selected as the state representation, where

W = horizontal position of the monkey

x = 1 or 0, depending on whether the monkey is on top of the box or not, respectively

† It is noted that the monkey could be a mobile robot.

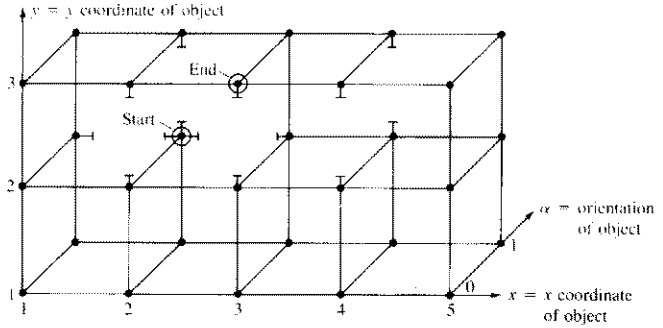


Figure 10.4 Graph of the problem in Fig. 10.3.

Y = horizontal position of the box

z = 1 or 0, depending on whether the monkey has grasped the bananas or not, respectively

The operators in this problem are:

1. goto(U). Monkey goes to horizontal position U, or in the form of a *production rule*,

$$(W,0,Y,z) \xrightarrow{\text{goto}(U)} (U,0,Y,z)$$

That is, state $(W,0,Y,z)$ can be transformed into $(U,0,Y,z)$ by the applying operator goto(U).

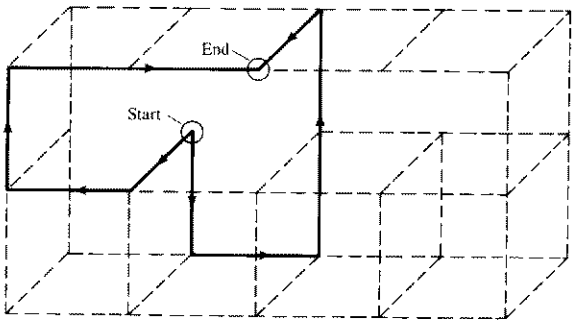


Figure 10.5 Solution to the graph of Fig. 10.4.

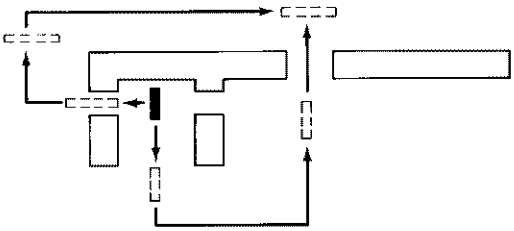


Figure 10.6 Visualization of solution path for Fig. 10.5.

2. pushbox(V). Monkey pushes the box to horizontal position V, or

$$(W,0,W,z) \xrightarrow{\text{pushbox}(V)} (V,0,V,z)$$

It should be noted from the left side of the production rule that, in order to apply the operator pushbox(V), the monkey should be at the same position W as the box, but not on top of it. Such a condition imposed on the applicability of an operator is called the *precondition* of the production rule.

3. climbbox. Monkey climbs on top of the box, or

$$(W,0,W,z) \xrightarrow{\text{climbbox}} (W,1,W,z)$$

It should be noted that, in order to apply the operator climbbox, the monkey must be at the same position W as the box, but not on top of it.

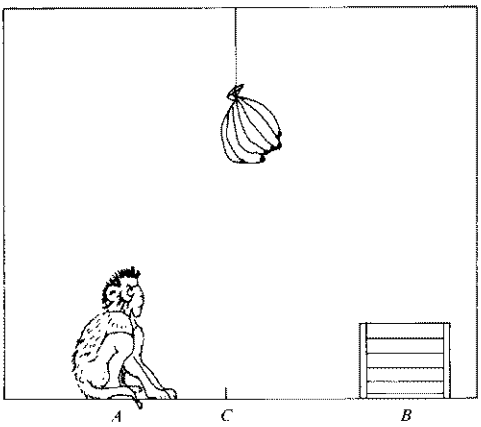


Figure 10.7 Monkey-and-bananas problem.

4. grasp. Monkey grasps the bananas, or

$$(C,1,C,0) \xrightarrow{\text{grasp}} (C,1,C,1)$$

where C is the location on the floor directly under the bananas. It should be noted that in order to apply the operator grasp, the monkey and the box should both be at position C and the monkey should already be on the top of the box.

It is noted that both the applicability and the effects of the operators are expressed by the production rules. For example, in rule 2, the operator pushbox(V) is only applicable when its precondition is satisfied. The effect of the operator is that the monkey has pushed the box to position V. In this formulation, the set of goal states is described by any list whose last element is 1.

Let the initial state be $(A,0,B,0)$. The only operator that is applicable is goto (U), resulting in the next state $(U,0,B,0)$. Now three operators are applicable; they are goto(U), pushbox(V) and climbbox (if $U = B$). Continuing to apply all operators applicable at every state, we produce the state space in terms of the graph representation shown in Fig. 10.8. It can be easily seen that the sequence of operators that transforms the initial state into a goal state consists of goto(B), pushbox(C), climbbox, and grasp.

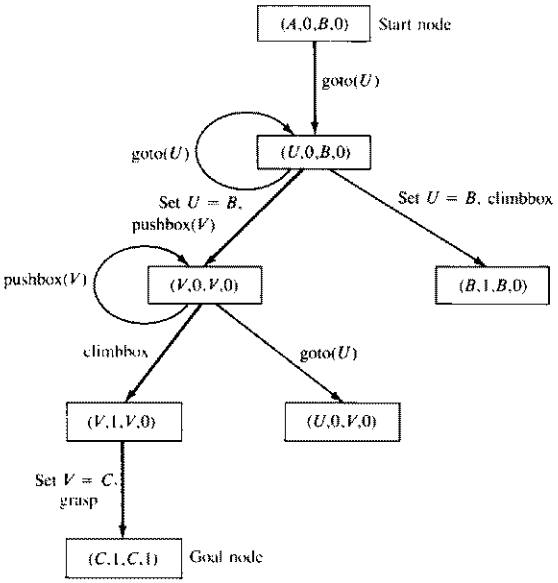


Figure 10.8 Graph representation for the monkey-and-bananas problem.

10.2.2 Graph-Search Techniques

For small graphs, such as the one shown in Fig. 10.8, a solution path from the initial state to a goal state can be easily obtained by inspection. For a more complicated graph a formal search process is needed to move through the state (problem) space until a path from an initial state to a goal state is found. One way to describe the search process is to use *production systems*. A production system consists of:

1. A *database* that contains the information relevant to the particular task. Depending on the application, this database may be as simple as a small matrix of numbers or as complex as a large, relational indexed file structure.
2. A set of *rules* operating on the database. Each rule consists of a left side that determines the applicability of the rule or precondition, and a right side that describes the action to be performed if the rule is applied. Application of the rule changes the database.
3. A *control strategy* that specifies which rules should be applied and ceases computation when a termination condition on the database is satisfied.

In terms of production system terminology, a graph such as the one shown in Fig. 10.8 is generated by the control strategy. The various databases produced by rule applications are actually represented as nodes in the graph. Thus, a graph-search control strategy can be considered as a means of finding a path in a graph from a (start) node representing the initial database to one (goal node) representing a database that satisfies the termination (or goal) condition of the production system.

A general graph-search procedure can be described as follows.

Step 1. Create a search graph G consisting solely of the start node s. Put s on a list called OPEN.

Step 2. Create a list called CLOSED that is initially empty.

Step 3. LOOP: if OPEN is empty, exit with failure.

Step 4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this mode n.

Step 5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G (pointers are established in step 7).

Step 6. Expand node n, generating the set M of its successors that are not ancestors of n. Install these members of M as successors of n in G.

Step 7. Establish a pointer to n from those members of M that were not already in OPEN or CLOSED. Add these members of M to OPEN. For each member of M that was already on OPEN or CLOSED, decide whether or

not to redirect its pointer to n. For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer†.

Step 8. Reorder the list OPEN, either according to some arbitrary criterion or according to heuristic merit.

Step 9. Go to LOOP.

If no heuristic information from the problem domain is used in ordering the nodes on OPEN, some arbitrary criterion must be used in step 8. The resulting search procedure is called *uninformed* or *blind*. The first type of blind search procedure orders the nodes on OPEN in increasing order of their depth in the search tree.‡ The search that results from such an ordering is called breadth-first search. It has been shown that breadth-first search is guaranteed to find a shortest-length path to a goal node, providing that a path exists. The second type of blind search orders the nodes on OPEN in descending order of their depth in the search tree. The deepest nodes are put first in the list. Nodes of equal depth are ordered arbitrarily. The search that results from such an ordering is called depth-first search. To prevent the search process from running away along some fruitless path forever, a depth bound is set. No node whose depth in the search tree exceeds this bound is ever generated.

The blind search methods described above are exhaustive search techniques for finding paths from the start node to a goal node. For many tasks it is possible to use task-dependent information to help reduce the search. This class of search procedures is called *heuristic* or *best-first* search, and the task-dependent information used is called heuristic information. In step 8 of the graph search procedure, heuristic information can be used to order the nodes on OPEN so that the search expands along those sectors of the graph thought to be the most promising. One important method uses a real-valued evaluation function to compute the "promise" of the nodes. Nodes on OPEN are ordered in increasing order of their values of the evaluation function. Ties among the nodes are resolved arbitrarily, but always in favor of goal nodes. The choice of evaluation function critically determines search results. A useful best-first search algorithm is the so-called A* algorithm described below.

Let the evaluation function f at any node n be

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

where $g(n)$ is a measure of the cost of getting from the start node to node n, and

† If the graph being searched is a tree, then none of the successors generated in step 6 has been generated previously. Thus, the members of M are not already on either OPEN or CLOSED. In this case, each member of M is added to OPEN and is installed in the search tree as successors of n. If the graph being searched is not a tree, it is possible that some of the members of M have already been generated, that is, they may already be on OPEN or CLOSED.

‡ To promote earlier termination, goal nodes should be put at the very beginning of OPEN.

$h(n)$ is an estimate of the additional cost from node n to a goal node. That is, $f(n)$ represents an estimate of the cost of getting from the start node to a goal node along the path constrained to go through node n.

The A* Algorithm

Step 1. Start with OPEN containing only the start node. Set that node's g value to 0, its h value to whatever it is, and its f value to $h + 0$, or h. Set CLOSED to the empty list.

Step 2. Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise, pick the node on OPEN with the lowest f value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution (either BESTNODE if all we want is the node, or the path that has been created between the start node and BESTNODE if we are interested in the path). Otherwise, generate the successors of BESTNODE, but do not set BESTNODE to point to them yet. (First we need to see if any of them have already been generated.) For each such SUCCESSOR, do the following:

a. Set SUCCESSOR to point back to BESTNODE. These back links will make it possible to recover the path once a solution is found.

b. Compute g(SUCCESSOR) = g(BESTNODE) + cost of getting from BESTNODE to SUCCESSOR.

c. See if SUCCESSOR is the same as any node on OPEN (i.e., it has already been generated but not processed). If so, call that node OLD. Since this node already exists in the graph, we can throw SUCCESSOR away, and add OLD to the list of BESTNODE's successors. Now we must decide whether OLD's parent link should be reset to point to BESTNODE. It should be if the path we have just found to SUCCESSOR is cheaper than the current best path to OLD (since SUCCESSOR and OLD are really the same node). So see whether it is cheaper to get to OLD via its current parent or to SUCCESSOR via BESTNODE, by comparing their g values. If OLD is cheaper (or just as cheap), then we need do nothing. If SUCCESSOR is cheaper, then reset OLD's parent link to point to BESTNODE, record the new cheaper path in g(OLD), and update f(OLD).

d. If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD, and add OLD to the list of BESTNODE's successors. Check to see if the new path or the old path is better just as in step 2*c*, and set the parent link and g and f values appropriately. If we have just found a better path to OLD, we must propagate the improvement to OLD's successors. This is a bit tricky. OLD points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that

either is still on OPEN or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node's g value (and thus also its f value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found. This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

e. If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute f(SUCCESSOR) = g(SUCCESSOR) + h(SUCCESSOR).

It is easy to see that the A* algorithm is essentially the graph search algorithm using $f(n)$ as the evaluation function for ordering nodes. Note that because $g(n)$ and $h(n)$ must be added, it is important that $h(n)$ be a measure of the cost of getting from node n to a goal node.

The objective of a search procedure is to discover a path through a problem space from an initial state to a goal state. There are two directions in which such a search could proceed: (1) forward, from the initial states, and (2) backward, from the goal states. The rules in the production system model can be used to reason forward from the initial states and to reason backward from the goal states. To reason forward, the left sides or the preconditions are matched against the current state and the right side (the results) are used to generate new nodes until the goal is reached. To reason backward, the right sides are matched against the current state and the left sides are used to generate new nodes representing new goal states to be achieved. This continues until one of these goal states is matched by a initial state.

By describing a search process as the application of a set of rules, it is easy to describe specific search algorithms without reference to the direction of the search. Of course, another possibility is to work both forward from the initial state and backward from the goal state simultaneously until two paths meet somewhere in between. This strategy is called *bidirectional search.*

10.3 PROBLEM REDUCTION

Another approach to problem solving is *problem reduction.* The main idea of this approach is to reason backward from the problem to be solved, establishing sub-

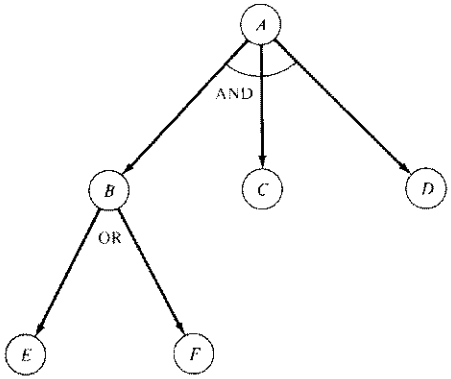


Figure 10.9 An AND/OR graph.

problems and sub-subproblems until, finally, the original problem is reduced to a set of trivial primitive problems whose solutions are obvious. A problem-reduction operator transforms a problem description into a set of reduced or successor problem descriptions. For a given problem description there may be many reduction operators that are applicable. Each of these produces an alternative set of subproblems. Some of the subproblems may not be solvable, however, so we may have to try several operators in order to produce a set whose members are all solvable. Thus it again requires a search process.

The reduction of problem to alternative sets of successor problems can be conveniently expressed by a graphlike structure. Suppose problem A can be solved by solving all of its three subproblems B, C, and D; an AND arc will be marked on the incoming arcs of the nodes B, C, and D. The nodes B, C, and D are called AND nodes. On the other hand, if problem B can be solved by solving any one of the subproblems E and F, an OR arc will be used. These relationships can be shown by the AND/OR graph shown in Fig. 10.9. It is easily seen that the search methods discussed in Sec. 10.2 are for OR graphs through which we want to find a single path from the start node to a goal node.

Example: An AND/OR graph for the monkey-and-bananas problem is shown in Fig. 10.10. Here, the problem configuration is represented by a triple (S,F,G), where S is the set of starting states, F is the set of operators, and G the set of goal states. Since the operator set F does not change in this problem and the initial state is $(A,0,B,0)$, we can suppress the symbol F and denote the problem simply by $(\{(A,0,B,0)\},G)$. One way of selecting problem-reduction operators is through the use of a *difference*. Loosely speaking, the difference for (S,F,G) is a partial list of reasons why the goal test defining the set G is failed by the member of S. (If some member of S is in G, the problem is solved and there is no difference.)

From the example in Sec. 10.2.1, F = $\{f_1, f_2, f_3, f_4\}$ = {goto(U), pushbox(V), climbbox, grasp}. First, we calculate the difference for the initial problem. The reason that the list $(A,0,B,0)$ fails to satisfy the goal test is that the last element is not 1. The operator relevant to reduce this difference is f_4 = grasp. Using f_4 to reduce the initial problem, we obtain the following pair of subproblems: $(\{(A,0,B,0)\}, G_{f_4})$ and $(\{f_4(s_1)\}G)$, where G_{f_4} is the set of state descriptions to which the operator f_4 is applicable and s_1 is that state in G_{f_4} obtained as a consequence of solving $(\{(A,0,B,0)\}, G_{f_4})$.

To solve the problem $(\{(A,0,B,0)\}, G_{f_4})$, we first calculate its difference. The state described by $(A,0,B,0)$ is not in G_{f_4} because (1) the box is not at C, (2) the monkey is not at C, and (3) the monkey is not on the box. The operators relevant to reduce these differences are, respectively, f_2 = pushbox(C), f_1 = goto(C), and f_3 = climbbox. Applying operator f_2 results in the subproblems $(\{(A,0,B,0)\}, G_{f_2})$ and $(f_2(s_{11}), G_{f_3})$, where $s_{11} \in G_{f_2}$ is obtained as a consequence of solving the first subproblem.

Since $(\{(A,0,B,0)\}, G_{f_2})$ must be solved first, we calculate its difference. The difference is that the monkey is not at B, and the relevant operator is f_1 = goto(B). This operator is then used to reduce the problem to a pair of subproblems $(\{(A,0,B,0)\}, G_{f_1})$ and $(f_1(s_{111}), G_{f_2})$. Now the first of these problems is primitive; its difference is zero since $(A,0,B,0)$ is in the domain of f_1 and f_1 is applicable to solve this problem. Note that $f_1(s_{111}) = (B,0,B,0)$ so the second problem becomes $(\{(B,0,B,0)\}, G_{f_2})$. This problem is also primitive since $(B,0,B,0)$ is in the domain of f_2, and f_2 is applicable to solve this problem. This process of completing the solution of problems generated earlier is continued until the initial problem is solved. □

In an AND/OR graph, one of the nodes, called the *state node*, corresponds to the original problem description. Those nodes in the graph corresponding to primitive problem descriptions are called *terminal nodes*. The objective of the search process carried out on an AND/OR graph is to show that the start node is solved. The definition of a solved node can be given recursively as follows:

1. The terminal nodes are solved nodes since they are associated with primitive problems.
2. If a nonterminal node has OR successors, then it is a solved node if and only if at least one of its successors is solved.
3. If a nonterminal node has AND successors, then it is a solved node if and only if all of its successors are solved.

A *solution graph* is the subgraph of solved nodes that demonstrates that the start node is solved. The task of the production system or the search process is to find a solution graph from the start node to the terminal nodes. Roughly speaking, a solution graph from node n to a set of nodes N of an AND/OR graph is analo-

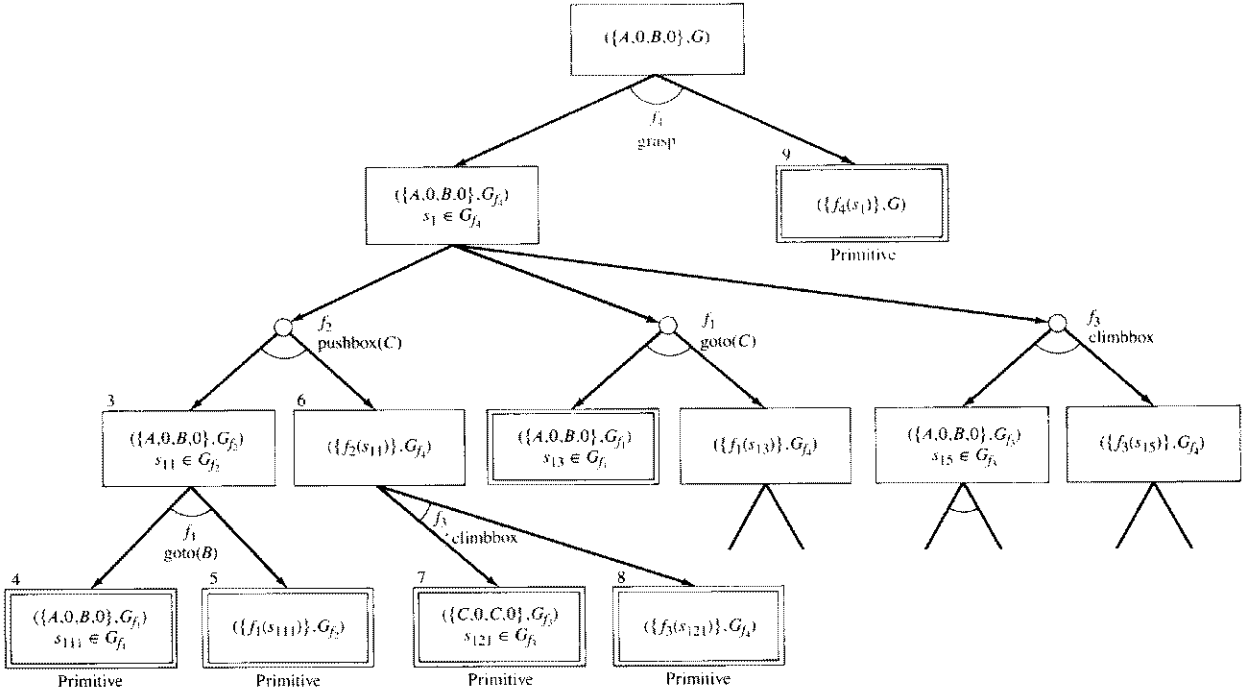


Figure 10.10 AND/OR graph for monkey-and-bananas problem.

gous to a path in an ordinary graph. It can be obtained by starting with node n and selecting exactly one outgoing arc. From each successor node to which this arc is directed, we continue to select one outgoing arc, and so on until eventually every successor thus produced is an element of N.

In order to find solutions in an AND/OR graph, we need an algorithm similar to A*, but with the ability to handle the AND arc appropriately. Such an algorithm for performing heuristic search of an AND/OR graph is the so-called AO* algorithm.

The AO* Algorithm

Step 1. Let G consist only of the node representing the initial state. (Call this node INIT.) Compute h(INIT).

Step 2. Until INIT is labeled SOLVED or until INIT's h value becomes greater than FUTILITY, repeat the following procedure:

a. Trace the marked arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node NODE.

b. Generate the successors of NODE. If there are none, then assign FUTILITY as the h value of NODE. This is equivalent to saying that NODE is not solvable. If there are successors, then for each one (called SUCCESSOR) that is not also an ancestor of NODE do the following:

(1) Add SUCCESSOR to G.

(2) If SUCCESSOR is a terminal node, label it SOLVED and assign it an h value of 0.

(3) If SUCCESSOR is not a terminal node, compute its h value.

c. Propagate the newly discovered information up the graph by doing the following: Let S be a set of nodes that have been marked SOLVED or whose h values have been changed and so need to have values propagated back to their parents. Initialize S to NODE. Until S is empty, repeat the following procedure:

(1) Select from S a node none of whose descendants in G occurs in S. (In other words, make sure that for every node we are going to process, we process it before processing any of its ancestors.) Call this node CURRENT, and remove it from S.

(2) Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the h values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as CURRENT's new h value the minimum of the costs just computed for the arcs emerging from it.

(3) Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.

(4) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked arc have been labeled SOLVED.

(5) If CURRENT has been marked SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph. So add to S all the ancestors of CURRENT.

It is noted that rather than the two lists, OPEN and CLOSED, that were used in the A* algorithm, the AO* algorithm uses a single structure G, representing the portion of the search graph that has been explicitly generated so far. Each node in the graph points both down to its immediate successors and up to its immediate predecessors. Each node in the graph is associated with an h value, an estimate of the cost of a path from itself to a set of solution nodes. The g value (the cost of getting from the start node to the current node) is not stored as in the A* algorithm, and h serves as the estimate of goodness of a node. A quantity FUTILITY is needed. If the estimated cost of a solution becomes greater than the value of FUTILITY, then the search is abandoned. FUTILITY should be selected to correspond to a threshold such that any solution with a cost above it is too expensive to be practical, even if it could ever be found.

A breadth-first algorithm can be obtained from the AO* algorithm by assigning $h \equiv 0$.

10.4 USE OF PREDICATE LOGIC

Robot problem solving requires the capability for representing, retrieving, and manipulating sets of statements. The language of logic or, more specifically, the first-order predicate calculus, can be used to express a wide variety of statements. The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old (i.e., mathematical deduction). In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known to be true. Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.†

Let us first explore the use of propositional logic as a way of representing knowledge. Propositional logic is appealing because it is simple to deal with and there exists a decision procedure for it. We can easily represent real-world facts as logical *propositions* written as *well-formed formulas* (wffs) in propositional logic, as shown in the following:

It is raining.
RAINING

† At this point, readers who are unfamiliar with propositional and predicate logic may want to consult a good introductory logic text before reading the rest of this chapter. Readers who want a more complete and formal presentation of the material in this section should consult the book by Chang and Lee [1973].

It is sunny.
SUNNY
It is foggy.
FOGGY
If it is raining then it is not sunny.
RAINING → ~SUNNY

Using these propositions, we could, for example, deduce that it is not sunny if it is raining. But very quickly we run up against the limitations of propositional logic. Suppose we want to represent the obvious fact stated by the sentence

John is a man

We could write

JOHNMAN

But if we also wanted to represent

Paul is a man

we would have to write something such as

PAULMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between John and Paul. It would be much better to represent these facts as

MAN(JOHN)
MAN(PAUL)

since now the structure of the representation reflects the structure of the knowledge itself. We are in even more difficulty if we try to represent the sentence

All men are mortal

because now we really need quantification unless we are willing to write separate statements about the mortality of every known man.

So we appear to be forced to move to predicate logic as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in propositional logic. In predicate logic, we can represent real-world facts as *statements* written as wffs. But a major motivation for choosing to use logic at all was that if we used logical statements as a way of representing knowledge, then we had available a good way of reasoning with that knowledge. In this section, we briefly introduce the language and methods of predicate logic.

The elementary components of predicate logic are *predicate symbols*, *variable symbols*, *function symbols*, and *constant symbols*. A predicate symbol is used to represent a relation in a domain of discourse. For example, to represent the fact "Robot is in Room r_1," we might use the simple *atomic formula*:

INROOM(ROBOT,r_1)

In this atomic formula, ROBOT and r_1 are constant symbols. In general, atomic formulas are composed of predicate symbols and terms. A constant symbol is the

simplest kind of term and is used to represent objects or entities in a domain of discourse. Variable symbols are terms also, and they permit us to be indefinite about which entity is being referred to, for example, INROOM(x,y). Function symbols denote functions in the domain of discourse. For example, the function symbol *mother* can be used to denote the mapping between an individual and his or her female parent. We might use the following atomic formula to represent the sentence "John's mother is married to John's father."

$$\text{MARRIED[father(JOHN), mother(JOHN)]}$$

An atomic formula has value T (true) just when the corresponding statements about the domain are true and it has the value F (false) just when the corresponding statement is false. Thus INROOM(ROBOT,r_1) has value T, and INROOM(ROBOT,r_2) has value F. Atomic formulas are the elementary building blocks of predicate logic. We can combine atomic formulas to form more complex wffs by using connectives such as $\wedge$ (and), $\vee$ (or), and $\Rightarrow$ (implies). Formulas built by connecting other formulas by $\wedge$'s are called *conjunctions*. Formulas built by connecting other formulas by $\vee$'s are called *disjunctions*. The connective "$\Rightarrow$" is used for representing "if-then" statements, e.g., as in the sentence "If the monkey is on the box, then the monkey will grasp the bananas":

$$\text{ON(MONKEY, BOX)} \Rightarrow \text{GRASP(MONKEY, BANANAS)}$$

The symbol "$\sim$" (not) is used to negate the truth value of a formula; that is, it changes the value of a wff from T to F and vice versa. The (true) sentence "Robot is not in Room r_2" might be represented as

$$\sim \text{INROOM(ROBOT,}r_2\text{)}$$

Sometimes an atomic formula, $P(x)$, has value T for all possible values of x. This property is represented by adding the universal quantifier $(\forall x)$ in front of $P(x)$. If $P(x)$ has value T for at least one value of x, this property is represented by adding the existential quantifier $(\exists x)$ in front of $P(x)$. For example, the sentence "All robots are gray" might be represented by

$$(\forall x)[\text{ROBOT}(x) \Rightarrow \text{COLOR}(x,\text{GRAY})]$$

The sentence "There is an object in Room r_1" might be represented by

$$(\exists x)\text{INROOM}(x,r_1)$$

If P and Q are two wffs, the truth values of composite expressions made up of these wffs are given by the following table:

P	Q	$P \vee Q$	$P \wedge Q$	$P \Rightarrow Q$	$\sim P$
T	T	T	T	T	F
F	T	T	F	T	T
T	F	T	F	F	F
F	F	F	F	T	T

If the truth values of two wffs are the same regardless of their interpretation, these two wffs are said to be equivalent. Using the truth table, we can establish the following equivalences:

$\sim(\sim P)$	is equivalent to	P
$P \vee Q$	is equivalent to	$\sim P \Longrightarrow Q$

deMorgan's laws:

$\sim(P \vee Q)$	is equivalent to	$\sim P \wedge \sim Q$
$\sim(P \wedge Q)$	is equivalent to	$\sim P \vee \sim Q$

Distributive laws:

$P \wedge (Q \vee R)$	is equivalent to	$(P \wedge Q) \vee (P \wedge R)$
$P \vee (Q \wedge R)$	is equivalent to	$(P \vee Q) \wedge (P \vee R)$

Commutative laws:

$P \wedge Q$	is equivalent to	$Q \wedge P$
$P \vee Q$	is equivalent to	$Q \vee P$

Associative laws:

$(P \wedge Q) \vee R$	is equivalent to	$P \wedge (Q \wedge R)$
$(P \vee Q) \vee R$	is equivalent to	$P \vee (Q \vee R)$

Contrapositive law:

$P \Longrightarrow Q$	is equivalent to	$\sim Q \Longrightarrow \sim P$

In addition, we have

$\sim(\exists x)P(x)$	is equivalent to	$(\forall x)[\sim P(x)]$
$\sim(\forall x)P(x)$	is equivalent to	$(\exists x)[\sim P(x)]$

In predicate logic, there are rules of inference that can be applied to certain wffs and sets of wffs to produce new wffs. One important inference rule is *modus ponens*, that is, the operation to produce the wff W_2 from wffs of the form W_1 and $W_1 \Longrightarrow W_2$. Another rule of inference, *universal specialization*, produces the wff $W(A)$ from the wff $(\forall x)W(x)$, where A is any constant symbol. Using modus ponens and universal specialization together, for example, produces the wff $W_2(A)$ from the wffs $(\forall x)[W_1(x) \Longrightarrow W_2(x)]$ and $W_1(A)$.

Inference rules are applied to produce derived wffs from given ones. In the predicate logic, such derived wffs are called *theorems*, and the sequence of inference rule applications used in the derivation constitutes a *proof* of the theorem. In artificial intelligence, some problem-solving tasks can be regarded as the task of finding a proof for a theorem. The sequence of inferences used in the proofs gives a solution to the problem.

Example: The state space representation of the monkey-and-bananas problem can be modified so that the states are described by wffs. We assume that, in this example, there are three operators—grasp, climbbox, and pushbox.

Let the initial state s_0 be described by the following set of wffs:

$$\sim \text{ONBOX}$$

$$\text{AT(BOX,}B\text{)}$$
$$\text{AT(BANANAS,}C\text{)}$$
$$\sim\text{HB}$$

The predicate ONBOX has value T only when the monkey is on top of the box, and the predicate HB has value T only when the monkey has the bananas.

The effects of the three operators can be described by the following wffs:

1. grasp

$$(\forall s)\{\text{ONBOX}(s) \wedge \text{AT(BOX,}C\text{,}s) \Rightarrow \text{HB(grasp}(s))\}$$

meaning "For all s, if the monkey is on the box and the box is at C in state s, then the monkey will have the bananas in the state attained by applying the operator grasp to state s." It is noted that the value of grasp(s) is the new state resulting when the operator is applied to state s.

2. climbbox

$$(\forall s)\{\text{ONBOX(climbbox}(s))\}$$

meaning "For all s, the monkey will be on the box in the state attained by applying the operator climbbox to state s."

3. pushbox

$$(\forall x \forall s)\{\sim\text{ONBOX}(s) \Rightarrow \text{AT(BOX,}x\text{,pushbox(}x\text{,}s))\}$$

meaning "For all x and s, if the monkey is not on the box in state s, then the box will be at position x in the state attained by applying the operator pushbox(x) to state s."

The goal wff is

$$(\exists s)HB(s)$$

This problem can now be solved by a theorem-proving process to show that the monkey can have the bananas (Nilsson [1971]). □

10.5 MEANS-ENDS ANALYSIS

So far, we have discussed several search methods that reason either forward or backward but, for a given problem, one direction or the other must be chosen. Often, however, a mixture of the two directions is appropriate. Such a mixed strategy would make it possible to solve the main parts of a problem first and then go back and solve the small problems that arise in connecting the big pieces together.

A technique known as *means-ends analysis* allows us to do that. The technique centers around the detection of the difference between the current state and the goal state. Once such a difference is determined, an operator that can reduce the difference must be found. It is possible that the operator may not be applicable to the current state. So a subproblem of getting to a state in which it can be applied is generated. It is also possible that the operator does not produce exactly the goal state. Then we have a second subproblem of getting from the state it does produce to the goal state. If the difference was determined correctly, and if the operator is really effective at reducing the difference, then the two subproblems should be easier to solve than the original problem. The means-ends analysis is applied recursively to the subproblems. From this point of view, the means-ends analysis could be considered as a problem-reduction technique.

In order to focus the system's attention on the big problems first, the differences can be assigned priority levels. Differences of higher priority can then be considered before lower priority ones. The most important data structure used in the means-ends analysis is the "goal." The goal is an encoding of the current problem situation, the desired situation, and a history of the attempts so far to change the current situation into the desired one. Three main types of goals are provided:

Type 1. *Transform* object A into object B.
Type 2. *Reduce* a difference between object A and object B by modifying object A.

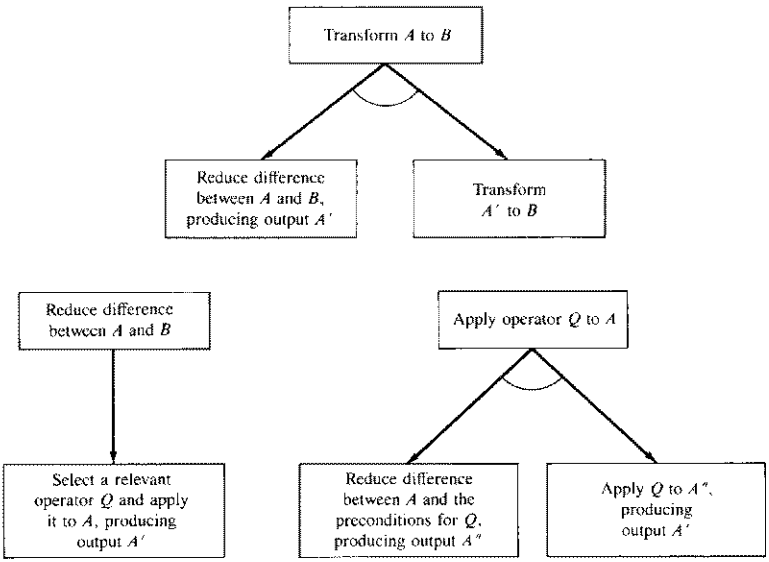


Figure 10.11 Methods for means-ends analysis.

Type 3. *Apply* operator Q to object A.

Associated with the goal types are methods or procedures for achieving them. These methods, shown in a simplified form in Fig. 10.11, can be interpreted as problem-reduction operators that give rise either to AND nodes, in the case of *transform* or *apply*, or to OR nodes, in the case of a *reduce* goal.

The first program to exploit means-ends analysis was the general problem solver (GPS). Its design was motivated by the observation that people often use this technique when they solve problems. For GPS, the initial task is represented as a *transform* goal, in which A is the initial object or state and B the desired object or the goal state. The recursion stops if, for a *transform* goal, there is no difference between A and B, or for an *apply* goal the operator Q is immediately applicable. For a *reduce* goal, the recursion may stop, with failure, when all relevant operators have been tried and have failed.

In trying to transform object A into object B, the *transform* method uses a matching process to discover the differences between the two objects. The difference with the highest priority is the one chosen for reduction. A difference-operator table lists the operators relevant to reducing each difference.

Consider a simple robot problem in which the available operators are listed as follows:

	Preconditions	Operator	Results
1.	AT(ROBOT,OBJ) ∧ LARGE(OBJ) ∧ CLEAR(OBJ) ∧ HANDEMPTY	PUSH(OBJ,LOC) ⟶	AT(OBJ,LOC) ∧ AT(ROBOT,LOC)
2.	AT(ROBOT,OBJ) ∧ SMALL(OBJ)	CARRY(OBJ,LOC) ⟶	AT(OBJ,LOC) ∧ AT(ROBOT,LOC)
3.	None	WALK(LOC) ⟶	AT(ROBOT,LOC)
4.	AT(ROBOT,OBJ)	PICKUP(OBJ) ⟶	HOLDING(OBJ)
5.	HOLDING(OBJ)	PUTDOWN(OBJ) ⟶	~HOLDING(OBJ)
6.	AT(ROBOT,OBJ2) ∧ HOLDING(OBJ1)	PLACE(OBJ1,OBJ2) ⟶	ON(OBJ1,OBJ2)

Fig. 10.12 shows the difference-operator table that describes when each of the operators is appropriate. Notice that sometimes there may be more than one operator that can reduce a given difference, and a given operator may be able to reduce more than one difference.

Difference \ Operator	PUSH	CARRY	WALK	PICKUP	PUTDOWN	PLACE
Move object	√	√				
Move robot			√			
Clear object				√		
Get object on object						√
Get hand empty					√	√
Be holding object				√		

Figure 10.12 A difference-operator table.

Suppose that the robot were given the problem of moving a desk with two objects on it from one room to another. The objects on top must also be moved. The main difference between the initial state and the goal state would be the location of the desk. To reduce the difference, either PUSH or CARRY could be chosen. If CARRY is chosen first, its preconditions must be met. This results in two more differences that must be reduced: the location of the robot and the size of the desk. The location of the robot can be handled by applying operator WALK, but there are no operators that can change the size of an object. So the path leads to a dead end. Following the other possibility, operator PUSH will be attempted.

PUSH has three preconditions, two of which produce differences between the initial state and the goal state. Since the desk is already large, one precondition creates no difference. The robot can be brought to the correct location by using the operator WALK, and the surface of the desk can be cleared by applying operator PICKUP twice. But after one PICKUP, an attempt to apply the second time results in another difference—the hand must be empty. The operator PUTDOWN can be applied to reduce that difference.

Once PUSH is performed, the problem is close to the goal state, but not quite. The objects must be placed back on the desk. The operator PLACE will put them there. But it cannot be applied immediately. Another difference must be eliminated, since the robot must be holding the objects. The operator PICKUP can be applied. In order to apply PICKUP, the robot must be at the location of the objects. This difference can be reduced by applying WALK. Once the robot is at the location of the two objects, it can use PICKUP and CARRY to move the objects to the other room.

The order in which differences are considered can be critical. It is important that significant differences be reduced before less critical ones. Section 10.6 describes a robot problem-solving system, STRIPS, which uses the means-ends analysis.

10.6 PROBLEM SOLVING

The simplest type of robot problem-solving system is a production system that uses the state description as the database. State descriptions and goals for robot problems can be constructed from logical statements. As an example, consider the robot hand and configurations of blocks shown in Fig. 10.1. This situation can be represented by the conjunction of the following statements:

CLEAR(B)	Block B has a clear top
CLEAR(C)	Block C has a clear top
ON(C,A)	Block C is on block A
ONTABLE(A)	Block A is on the table
ONTABLE(B)	Block B is on the table
HANDEMPTY	The robot hand is empty

The goal is to construct a stack of blocks in which block B is on block C and block A is on block B. In terms of logical statements, we may describe the goal as ON(B,C) $\wedge$ ON(A,B).

Robot actions change one state, or configuration, of the world into another. One simple and useful technique for representing robot action is employed by a robot problem-solving system called STRIPS (Fikes and Nilsson [1971]). A set of rules is used to represent robot actions. Rules in STRIPS consist of three components. The first is the *precondition* that must be true before the rule can be applied. It is usually expressed by the left side of the rule. The second component is a list of predicates called the *delete list*. When a rule is applied to a state description, or database, delete from the database the assertions in the delete list. The third component is called the *add list*. When a rule is applied, add the assertions in the add list to the database. The MOVE action for the block-stacking example is given below:

MOVE(X,Y,Z)	Move object X from Y to Z
Precondition:	CLEAR(X), CLEAR(Z), ON(X,Y)
Delete list:	ON(X,Y), CLEAR(Z)
Add list:	ON(X,Z), CLEAR(Y)

If MOVE is the only operator or robot action available, the search graph (or tree) shown in Fig. 10.2 is generated.

Consider a more concrete example with the initial database shown in Fig. 10.1 and the following four robot actions or operations in STRIPS-form:

1. PICKUP(X)
   Precondition and delete list: ONTABLE(X), CLEAR(X), HANDEMPTY
   Add list: HOLDING(X)
2. PUTDOWN(X)
   Precondition and delete list: HOLDING(X)
   Add list: ONTABLE(X), CLEAR(X), HANDEMPTY

3. STACK(X,Y)
   Precondition and delete list: HOLDING(X), CLEAR(Y)
   Add list: HANDEMPTY, ON(X,Y), CLEAR(X)
4. UNSTACK(X,Y)
   Precondition and delete list: HANDEMPTY, CLEAR(X), ON(X,Y)
   Add list: HOLDING(X), CLEAR(Y)

Suppose that our goal is ON(B,C) $\wedge$ ON(A,B). Working forward from the initial state description shown in Fig. 10.1, we obtain the complete state space for this problem as shown in Fig. 10.13, with a solution path between the initial state and the goal state indicated by dark lines. The solution sequence of actions consists of: {UNSTACK(C,A), PUTDOWN(C), PICKUP(B), STACK(B,C), PICKUP(A), STACK(A,B)}. It is called a "plan" for achieving the goal.

If a problem-solving system knows how each operator changes the state of the world or the database and knows the preconditions for an operator to be executed, it can apply means-ends analysis to solve problems. Briefly, this technique involves looking for a difference between the current state and a goal state and trying to find an operator that will reduce the difference. A *relevant operator* is one whose add list contains formulas that would remove some part of the difference. This continues recursively until the goal state has been reached. STRIPS and most other planners use means-ends analysis.

We have just seen how STRIPS computes a specific plan to solve a particular robot problem. The next step is to generalize the specific plan by replacing constants by new parameters. In other words, we wish to elevate the particular plan to a *plan schema*. The need for a plan generalization is apparent in a learning system. For the purpose of saving plans so that portions of them can be used in a later planning process, the preconditions and effects of any portion of the plan need to be known. To accomplish this, plans are stored in a *triangle table* with rows and columns corresponding to the operators of the plan. The triangle table reveals the structure of a plan in a fashion that allows parts of the plan to be extracted later in solving related problems.

An example of a triangle table is shown in Fig. 10.14. Let the leftmost column be called the zeroth column; then the jth column is headed by the jth operator in the sequence. Let the top row be called the first row. If there are N operators in the plan sequence, then the last row is the $(N + 1)$th row. The entries in cell (i, j) of the table, for $j > 0$ and $i < N + 1$, are those statements added to the state description by the jth operator that survive as preconditions of the ith operator. The entries in cell $(i, 0)$ for $i < N + 1$ are those statements in the initial state description that survive as preconditions of the ith operator. The entries in the $(N + 1)$th row of the table are then those statements in the original state description, and those added by the various operators, that are components of the goal.

Triangle tables can easily be constructed from the initial state description, the operators in the sequence, and the goal description. These tables are concise and convenient representations for robot plans. The entries in the row to the left of the

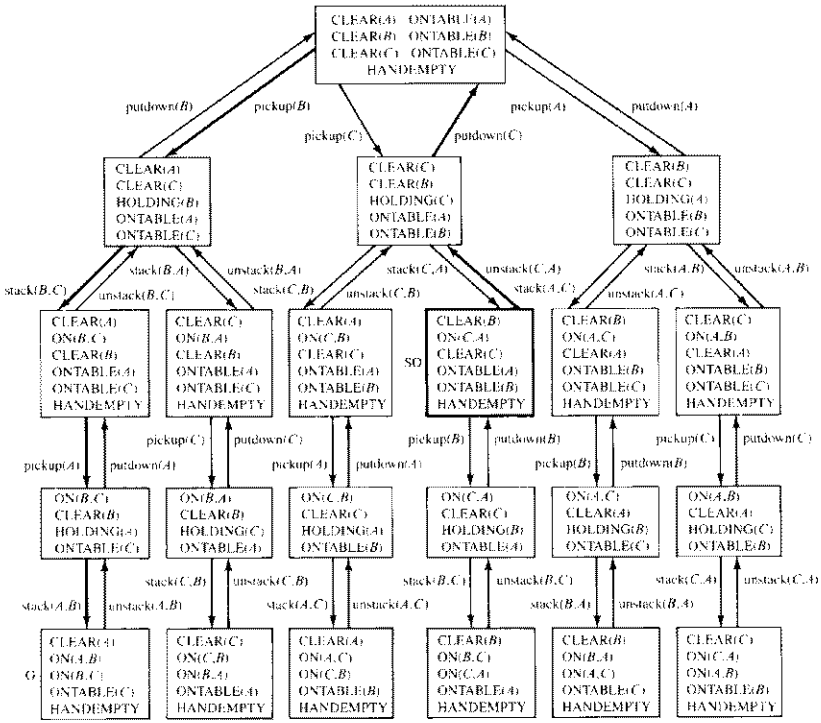


Figure 10.13 A state space for a robot problem.

*i*th operator are precisely the preconditions of the operator. The entries in the column below the *i*th operator are precisely the add formula statements of that operator that are needed by subsequent operators or that are components of the goal.

Let us define the *i*th *kernel* as the intersection of all rows below, and including, the *i*th row with all columns to the left of the *i*th column. The fourth kernel is outlined by double lines in Fig. 10.14. The entries in the *i*th kernel are then precisely the conditions that must be matched by a state description in order that the sequence composed of the *i*th and subsequent operators be applicable and achieve the goal. Thus, the first kernel (i.e., the zeroth column), contains those conditions of the initial state needed by subsequent operators and by the goal; the $(N + 1)$th kernel [i.e., the $(N + 1)$th row] contains the goal conditions themselves. These properties of triangle tables are very useful for monitoring the actual execution of robot plans.

Since robot plans must ultimately be executed in the real world by a mechanical device, the execution system must acknowledge the possibility that the actions

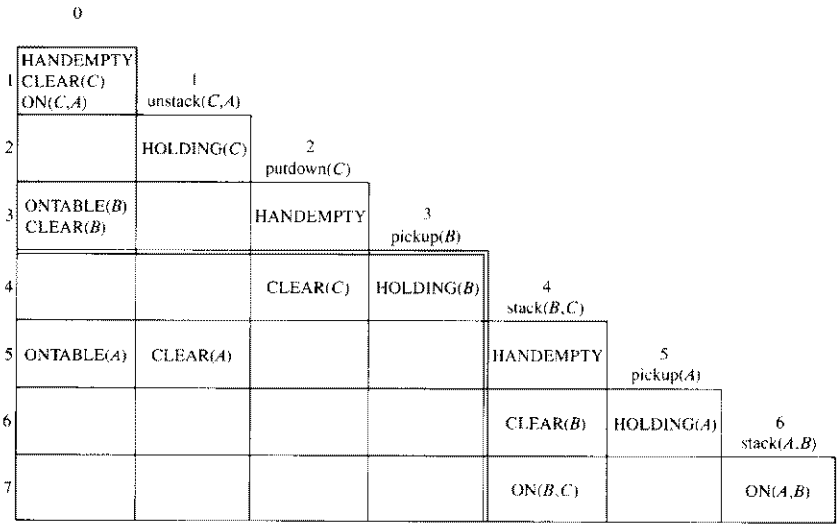


Figure 10.14 A triangle table.

in the plan may not accomplish their intended tasks and that mechanical tolerances may introduce errors as the plan is executed. As actions are executed, unplanned effects might either place us unexpectedly close to the goal or throw us off the track. These problems could be dealt with by generating a new plan (based on an updated state description) after each execution step, but obviously, such a strategy would be too costly, so we instead seek a scheme that can intelligently monitor progress as a given plan is being executed.

The kernels of triangle tables contain just the information needed to realize such a plan execution system. At the beginning of a plan execution, we know that the entire plan is applicable and appropriate for achieving the goal because the statements in the first kernel are matched by the initial state description, which was used when the plan was created. (Here we assume that the world is static; that is, no changes occur in the world except those initiated by the robot itself.) Now suppose the system has just executed the first $i - 1$ actions of a plan sequence. Then, in order for the remaining part of the plan (consisting of the ith and subsequent actions) to be both applicable and appropriate for achieving the goal, the statements in the ith kernel must be matched by the new current state description. (We assume that a sensory perception system continuously updates the state description as the plan is executed so that this description accurately models the current state of the world.) Actually, we can do better than merely check to see if the expected kernel matches the state description after an action; we can look for the highest numbered matching kernel. Then, if an unanticipated effect places us

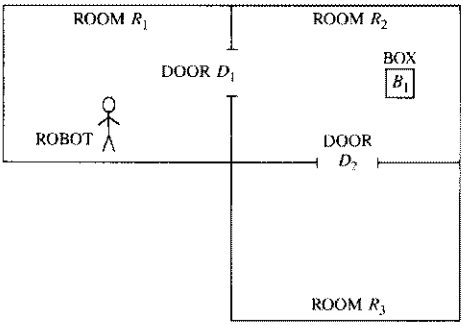
closer to the goal, we need only execute the appropriate remaining actions; and if an execution error destroys the results of previous actions, the appropriate actions can be reexecuted.

To find the appropriate matching kernel, we check each one in turn starting with the highest numbered one (which is the last row of the table) and work backward. If the goal kernel (the last row of the table) is matched, execution halts; otherwise, supposing the highest numbered matching kernel is the ith one, then we know that the ith operator is applicable to the current state description. In this case, the system executes the action corresponding to this ith operator and checks the outcome, as before, by searching again for the highest numbered matching kernel. In an ideal world, this procedure merely executes in order each action in the plan. In a real-world situation, on the other hand, the procedure has the flexibility to omit execution of unnecessary actions or to overcome certain kinds of failures by repeating the execution of appropriate actions. Replanning is initiated when there are no matching kernels.

As an example of how this process might work, let us return to our block-stacking problem and the plan represented by the triangle table in Fig. 10.14. Suppose that the system executes actions corresponding to the first four operators and that the results of these actions are as planned. Now suppose that the system attempts to execute the pickup block A action, but the execution routine (this time) mistakes block B for block A and picks up block B instead. [Assume again that the perception system accurately updates the state description by adding HOLDING(B) and deleting ON(B,C); in particular, it does not add HOLDING(A).] If there were no execution error, the sixth kernel would now be matched; the result of the error is that the highest numbered matching kernel is now kernel 4. The action corresponding to STACK(B,C) is thus reexecuted, putting the system back on track.

The fact that the kernels of triangle tables overlap can be used to advantage to scan the table efficiently for the highest numbered matching kernel. Starting in the bottom row, we scan the table from left to right, looking for the first cell that contains a statement that does not match the current state description. If we scan the whole row without finding such a cell, the goal kernel is matched; otherwise, if we find such a cell in column i, the number of the highest numbered matching kernel cannot be greater than i. In this case, we set a *boundary* at column i and move up to the next-to-bottom row and begin scanning this row from left to right, but not past column i. If we find a cell containing an unmatched statement, we reset the column boundary and move up another row to begin scanning that row, etc. With the column boundary set to k, the process terminates by finding that the kth kernel is the highest numbered matching kernel when it completes a scan of the kth row (from the bottom) up to the column boundary.

Example: Consider the simple task of fetching a box from an adjacent room by a robot vehicle. Let the initial state of the robot's world model be as shown in Fig. 10.15. Assume that there are two operators, GOTHRU and PUSHTHRU.



Initial data base M_0:
INROOM (ROBOT,R_1)
CONNECTS (D_1,R_1,R_2)
CONNECTS (D_2,R_2,R_3)
BOX (B_1)
INROOM (B_1,R_2)

$(\forall x\ \forall y\ \forall z)$ [CONNECTS $(x)(y)(z) \rightarrow$ CONNECTS $(x)(y)(z)$]

Goal G_0:
$(\exists x)$ [BOX$(x) \wedge$ INROOM (x,R_1)]

Figure 10.15 Initial world model.

GOTHRU(d,r_1,r_2) Robot goes through door d from room r_1 into room r_2
Precondition: INROOM(ROBOT,r_1) $\wedge$ CONNECTS(d,r_1,r_2) the robot is in room r_1 and door d connects room r_1 to room r_2
Delete list: INROOM(ROBOT,S) for any value of S
Add list: INROOM(ROBOT,r_2)

PUSHTHRU(b,d,r_1,r_2) Robot pushes object b through door d from room r_1 into room r_2
Precondition: INROOM(b,r_1) $\wedge$ INROOM(ROBOT,r_1) $\wedge$ CONNECTS(d,r_1,r_2)
Delete list: INROOM(ROBOT,S), INROOM(b,S)
Add list: INROOM(ROBOT,r_2), INROOM(b,r_2)

The difference-operator table is shown in Fig. 10.16. □

When STRIPS is given the problem, it first attempts to achieve the goal G_0 from the initial state M_0. This problem cannot be solved immediately. However, if the initial database contains a statement INROOM(B_1,R_1), the problem-solving process could continue. STRIPS finds the operator PUSHTHRU(B_1,d,r_1,R_1) whose effect can provide the desired statement. The precondition G_1 for PUSHTHRU is

G_1: INROOM(B_1,r_1) $\wedge$ INROOM(ROBOT,r_1) $\wedge$ CONNECTS(d,r_1,R_1)

Difference \ Operator	GOTHRU	PUSHTHRU
Location of box		√
Location of robot	√	
Location of box and robot		√

Figure 10.16 Difference-operator table.

From the means-ends analysis, this precondition is set up as a subgoal and STRIPS tries to accomplish it from M_0.

Although no immediate solution can be found to solve this problem, STRIPS finds that if $r_1 = R_2$, $d = D_1$ and the current database contains INROOM(ROBOT, R_2), the process could continue. Again STRIPS finds the operator GOTHRU(d,r_1,R_2) whose effect can produce the desired statement. Its precondition is the next subgoal, namely:

$$G_2\text{: INROOM(ROBOT,}r_1\text{)} \wedge \text{CONNECTS}(d,r_1,R_2)$$

Using the substitutions $r_1 = R_1$ and $d = D_1$, STRIPS is able to accomplish G_2. It therefore applies GOTHRU(D_1,R_1,R_2) to M_0 to yield

$$M_1\text{: INROOM(ROBOT,}R_2\text{), CONNECTS}(D_1,R_1,R_2)$$

$$\text{CONNECTS}(D_2,R_2,R_3),\ \text{BOX}(B_1)$$

$$\text{INROOM}(B_1,R_2),\ \ldots$$

$$(\forall x)(\forall y)(\forall z)\ [\text{CONNECTS}(x,y,z) \implies \text{CONNECTS}(x,z,y)]$$

Now STRIPS attempts to achieve the subgoal G_1 from the new database M_1. It finds the operator PUSHTHRU(B_1,D_1,R_2,R_1) with the substitutions $r_1 = R_2$ and $d = D_1$. Application of this operator to M_1 yields

$$M_2\text{: INROOM(ROBOT,}R_1\text{), CONNECTS}(D_1,R_1,R_2)$$

$$\text{CONNECTS}(D_1,R_2,R_3),\ \text{BOX}(B_1)$$

$$\text{INROOM}(B_1,R_1),\ \ldots$$

$$(\forall x \forall y \forall z)[\text{CONNECTS}(x,y,z) \implies \text{CONNECTS}(x,z,y)]$$

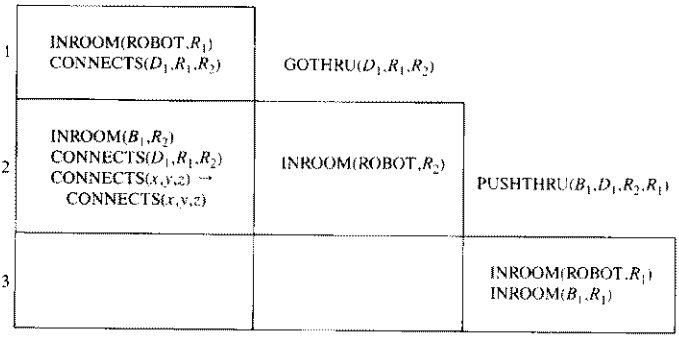


Figure 10.17 Triangle table.

Next, STRIPS attempts to accomplish the original goal G_0 from M_2. This attempt is successful and the final operator sequence is

$$\text{GOTHRU}(D_1,R_1,R_2),\ \text{PUSHTHRU}(B_1,D_1,R_2,R_1)$$

We would like to generalize the above plan so that it could be free from the specific constants D_1, R_1, R_2, and B_1 and used in situations involving arbitrary doors, rooms, and boxes. The triangle table for the plan is given in Fig. 10.17, and the triangle table for the generalized plan is shown in Fig. 10.18. Hence the plan could be generalized as follows:

$$\text{GOTHRU}(d_1,r_1,r_2)$$

$$\text{PUSHTHRU}(b,d_2,r_2,r_3)$$

and could be used to go from one room to an adjacent second room and push a box to an adjacent third room.

10.7 ROBOT LEARNING

We have discussed the use of triangle tables for generalized plans to control the execution of robot plans. Triangle tables for generalized plans can also be used by STRIPS to extract a relevant operator sequence during a subsequent planning process. Conceptually, we can think of a single triangle table as representing a family of generalized operators. Upon the selection by STRIPS of a relevant add list, we must extract from this family an economical parameterized operator achieving the add list. Recall that the $(i + 1)$th row of a triangle table (excluding the first cell) represents the add list, $A_{1,\ldots,i}$, of the ith head of the plan, i.e., of the sequence

$OP_1, \ldots, OP_i$. An n-step plan presents STRIPS with n alternative add lists, any one of which can be used to reduce a difference encountered during the normal planning process. STRIPS tests the relevance of each of a generalized plan's add lists in the usual fashion, and the add lists that provide the greatest reduction in the difference are selected. Often a given set of relevant statements will appear in more than one row of the table. In that case only the lowest-numbered row is selected, since this choice results in the shortest operator sequence capable of producing the desired statements.

Suppose that STRIPS selects the ith add list $A_{1,\ldots,i}$, $i < n$. Since this add list is achieved by applying in sequence $OP_1, \ldots, OP_i$, we will obviously not be interested in the application of $OP_{i+1}, \ldots, OP_n$, and will therefore not be interested in establishing any of the preconditions for these operators. In general, some steps of a plan are needed only to establish preconditions for subsequent steps. If we lost interest in a tail of a plan, then the relevant instance of the generalized plan need not contain those operators whose sole purpose is to establish preconditions for the tail. Also, STRIPS will, in general, have used only some subset of $A_{1,\ldots,i}$ in establishing the relevance of the ith head of the plan. Any of the first i operators that does not add some statement in this subset, or help establish the preconditions for some operator that adds a statement in the subset, is not needed in the relevant instance of the generalized plan.

In order to obtain a robot planning system that can not only speed up the planning process but can also improve its problem-solving capability to handle more complex tasks, one could design the system with a learning capability. STRIPS uses a generalization scheme for machine learning. Another form of learning would be the updating of the information in the difference-operator table from the system's experience.

Learning by analogy has been considered as a powerful approach and has been applied to robot planning. A robot planning system with learning, called PULP-I, has been proposed (Tangwongsan and Fu [1979]). The system uses an analogy between a current unplanned task and any known similar tasks to reduce the search

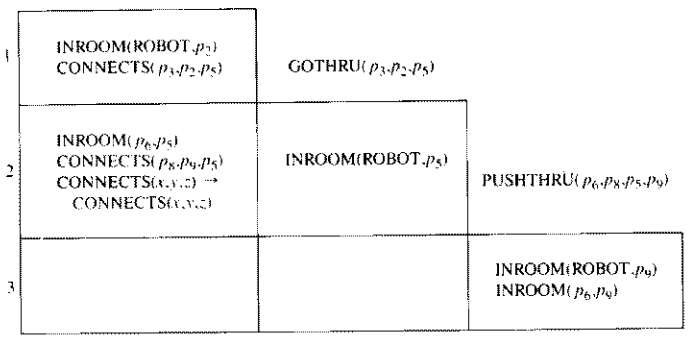


Figure 10.18 Triangle table for generalized plan.

for a solution. A semantic network, instead of predicate logic, is used as the internal representation of tasks. Initially a set of basic task examples is stored in the system as knowledge based on past experience. The analogy of two task statements is used to express the similarity between them and is determined by a semantic matching procedure. The matching algorithm measures the semantic "closeness"; the smaller the value, the closer the meaning. Based on the semantic matching measure, past experience in terms of stored information is retrieved and a candidate plan is formed. Each candidate plan is then checked by its operators' preconditions to ensure its applicability to the current world state. If the plan is not applicable, it is simply dropped out of the candidacy. After the applicability check, several candidate plans might be found. These candidate plans are listed in ascending order according to their evaluation values of semantic matching. The one with the smallest value of semantic matching has the top priority and must be at the beginning of the candidate list. Of course, if no candidate is found, the system terminates with failure.

Computer simulation of PULP-I has shown a significant improvement of planning performance. This improvement is not merely in the planning speed but also in the capability of forming complex plans from the learned basic task examples.

10.8 ROBOT TASK PLANNING

The robot planners discussed in the previous section require only a description of the initial and final states of a given task. These planning systems typically do not specify the detailed robot motions necessary to achieve an operation. These systems issue robot commands such as: PICKUP(A) and STACK(X,Y) without specifying the robot path. In the foreseeable future, however, robot task planners will need more detailed information about intermediate states than these systems provide. But they can be expected to produce a much more detailed robot program. In other words, a task planner would transform the task-level specifications into manipulator-level specifications. To carry out this transformation, the task planner must have a description of the objects being manipulated, the task environment, the robot carrying out the task, the initial state of the environment, and the desired final (goal) state. The output of the task planner would be a robot program to achieve the desired final state when executed in the specified initial state.

There are three phases in task planning: modeling, task specification, and manipulator program synthesis. The world model for a task must contain the following information: (1) geometric description of all objects and robots in the task environment; (2) physical description of all objects; (3) kinematic description of all linkages; and (4) descriptions of robot and sensor characteristics. Models of task states also must include the configurations of all objects and linkages in the world model.

10.8.1 Modeling

The geometric description of objects is the principal component of the world model. The major sources of geometric models are computer-aided design (CAD) systems and computer vision. There are three major types of three-dimensional object representation schemes (Requicha and Voelcker [1982]):

1. Boundary representation
2. Sweep representation
3. Volumetric representation

There are three types of volumetric representations: (1) spatial occupancy, (2) cell decomposition, and (3) constructive solid geometry (CSG). A system based on constructive solid geometry has been suggested for task planning. In CSG, the basic idea is that complicated solids are constructed by performing set operations on a few types of primitive solids. The object in Fig. 10.19*a* can be described by the structure given in Fig. 10.19*b*.

The legal motions of an object are constrained by the presence of other objects in the environment, and the form of the constraints depends on the shapes of the objects. This is the reason why a task planner needs geometric descriptions of objects. There are additional constraints on motion imposed by the kinematic structure of the robot itself. The kinematic models provide the task planner with the information required to plan manipulator motions that are consistent with external constraints.

Many of the physical characteristics of objects play important roles in planning robot operations. The mass and inertia of parts, for example, determine how fast they can be moved or how much force can be applied to them before they fall over. Another important aspect of a robot system is its sensing capabilities. For task planning, vision enables the robot to obtain the configuration of an object to some specified accuracy at execution time; force sensing allows the use of compliant motions; touch information could serve in both capacities. In addition to sensing, there are many individual characteristics of manipulators that must be described; for example, velocity and acceleration bounds, and positioning accuracy of each of the joints.

10.8.2 Task Specification

A model state is given by the configurations of all the objects in the environment; tasks are actually defined by sequences of states of the world model. There are three methods for specifying configurations: (1) using a CAD system to position models of the objects at the desired configurations, (2) using the robot itself to specify robot configurations and to locate features of the objects, and (3) using symbolic spatial relationships among object features to constrain the configurations of objects. Methods 1 and 2 produce numerical configurations which are difficult

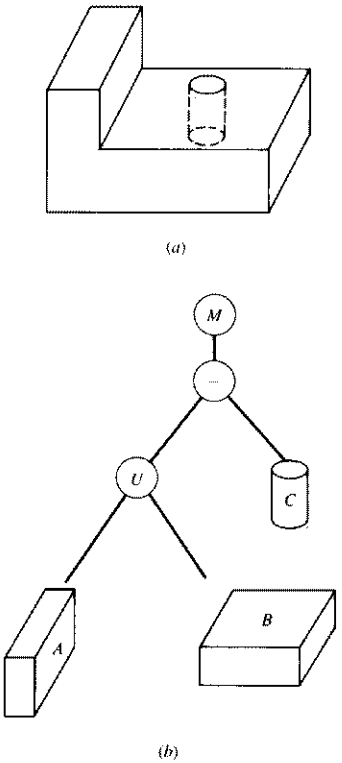


Figure 10.19 Constructive solid geometry (CSG). Attributes of A and B: length, width, height; attributes of C: radius, height. Set relational operators: $\cup$, union, $\cap$, intersection, $-$, difference.

to interpret and modify. In the third method, a configuration is described by a set of symbolic spatial relationships that are required to hold between objects in that configuration.

Since model states are simply sets of configurations and task specifications are sequences of model states, given symbolic spatial relationships for specifying configurations, we should be able to specify tasks. Assume that the model includes names for objects and object features. The first step in the task planning process is transforming the symbolic spatial relationships among object features to equations on the configuration parameters of objects in the model. These equations must then be simplified as much as possible to determine the legal ranges of configurations of all objects. The symbolic form of the relationships is also used during program synthesis.

10.8.3 Manipulator Program Synthesis

The synthesis of a manipulator program from a task specification is the crucial phase of task planning. The major steps involved in this phase are grasp planning, motion planning, and error detection. The output of the synthesis phase is a program composed of grasp commands, several kinds of motion specifications, and error tests. This program is generally in a manipulator-level language for a particular manipulator and is suitable for repeated execution without replanning.

10.9 BASIC PROBLEMS IN TASK PLANNING

10.9.1 Symbolic Spatial Relationships

The basic steps in obtaining configuration constraints from symbolic spatial relationships are: (1) defining a coordinate system for objects and object features, (2) defining equations of object configuration parameters for each of the spatial relationships among features, (3) combining the equations for each object, and (4) solving the equations for the configuration parameters of each object. Consider the following specification, given in the state depicted in Fig. 10.20:

$$\text{PLACE Block1}(f_1 \text{ against } f_3) \text{ and } (f_2 \text{ against } f_4)$$

The purpose is to obtain a set of equations that constrain the configuration of Block1 relative to the known configuration of Block2. That is, the face f_1 of Block2 must be against the face f_3 of Block1 and the face f_2 of Block2 must be against the face f_4 of Block1.

Each object and feature has a set of axes embedded in it, as shown in Fig. 10.21. Configurations of entities are the 4×4 transformation matrices:

$$f_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \qquad f_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

$$f_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad f_4 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Let $\text{twix}(\theta)$ be the transformation matrix for a rotation of angle θ around the x axis, $\text{trans}(x,y,z)$ the matrix for a translation x, y, and z, and let M be the matrix for the rotation around the y axis that rotates the positive x axis into the negative x axis, with $M = M^{-1}$. Each *against* relationship between two faces, say, face f on

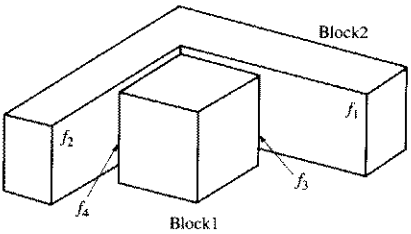


Figure 10.20 Illustration for spatial relationships among objects.

object A and face g on object B, generates the following constraint on the configuration of the two objects:

$$A = f^{-1}M \text{ twix}(\theta)\ (0,\ y,\ z)gB \tag{10.9-1}$$

The two *against* relations in the example of Fig. 10.15 generate the following equations:

$$\begin{aligned} \text{Block1} &= f_3^{-1}M \text{ twix}(\theta_1) \text{ trans}(0, y_1, z_1) f_1 \text{Block2} \\ \text{Block1} &= f_4^{-1}M \text{ twix}(\theta_2) \text{ trans}(0, y_2, z_2) f_2 \text{Block2} \end{aligned} \tag{10.9-2}$$

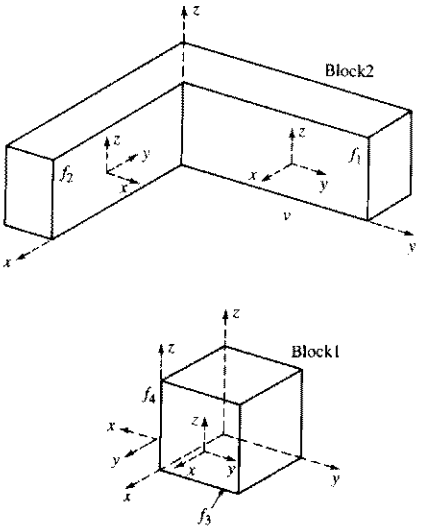


Figure 10.21 Axes embedded in objects and features from Fig. 10.20.

Equation (10.9-2) consists of two independent constraints on the configuration of Block1 that must be satisfied simultaneously. Setting the two expressions equal to each other and removing the common term, Block2, we get

$$f_3^{-1}M\ \text{twix}(\theta_1)\ \text{trans}(0,y_1,z_1)\ f_1 = f_4^{-1}M\ \text{twix}(\theta_2)\ \text{trans}(0,y_2,z_2)\ f_2 \qquad (10.9\text{-}3)$$

Applying the rewrite rules, (10.9-3) is transformed to

$$f_3^{-1}M\ \text{twix}(\theta_1)\ \text{trans}(0,y_1+1,z_1+1)(f'_2)^{-1} \times \text{trans}(0,-y_1,-z_2)\ \text{twix}(-\theta_2)M^{-1}\ f_4 = \text{I} \qquad (10.9\text{-}4)$$

where the primed matrix denotes the rotational component of the transformation, obtained by setting the last row of the matrix to [0,0,0,1].

It can be shown that the rotational and translational components of this type of equation can be solved independently. The rotational equation can be obtained by replacing each of the trans matrices by the identity and only using the rotational components of other matrices. The rotational equation for Eq. (10.9-4) is

$$(f'_3)^{-1}M\ \text{twix}(\theta_1)(f'_2)^{-1}\ \text{twix}(-\theta_2)M(f'_4) = \text{I} \qquad (10.9\text{-}5)$$

Since $f'_3 = \text{I}$, (10.9-5) can be rewritten as

$$\text{twix}(\theta_1)(f'_2)^{-1}\ \text{twix}(-\theta_2) = M(f'_4)^{-1}M \qquad (10.9\text{-}6)$$

Also, since

$$(f'_2) = M(f'_4)^{-1}M = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Eq. (10.9-6) is satisfiable and we can choose $\theta_2 = 0$. Letting $\theta_2 = 0$ in Eq. (10.9-6), we obtain

$$\text{twix}(\theta_1) = M(f'_4)^{-1}M(f'_2) = \text{I} \qquad (10.9\text{-}7)$$

From Eq. (10.9-7) we conclude that $\theta_1 = 0$. Thus, Eq. (10.9-2) becomes

$$\text{Block1} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & y_1 & 2+z_1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 2-y_2 & 0 & 2+z_2 & 1 \end{bmatrix} \qquad (10.9\text{-}8)$$

Equating the corresponding matrix terms, we obtain

$$2 - y_2 = 1$$

$$y_1 = 0$$

$$2 + z_1 = 2 + z_2$$

Hence, $y_2 = 1$, $y_1 = 0$, and $z_1 = z_2$; that is, the position of Block1 has 1 degree of freedom corresponding to translations along the z axis.

The method used in the above example was proposed by Ambler and Popplestone [1975]. The contact relationships treated there include *against*, *fits*, and *coplanar* among features that can be planar or spherical faces, cylindrical shafts and holes, edges and vertices. Taylor [1976] extended this approach to noncontact relationships such as for a peg in a hole of diameter greater than its own, an object in a box, or a feature in contact with a region of another feature. These relationships give rise to inequality constraints on the configuration parameters. They can be used to model, for example, the relationship of the position of the tip of a screwdriver in the robot's gripper to the position errors in the robot joints and the slippage of the screwdriver in the gripper. After simplifying the equalities and inequalities, a set of linear constraints are derived by using differential approximations for the rotations around a nominal configuration. The values of the configuration parameters satisfying the constraints can be bounded by applying linear programming techniques to the linearized constraint equations.

10.9.2 Obstacle Avoidance

The most common robot motions are transfer movements for which the only constraint is that the robot and whatever it is carrying should not collide with objects in the environment. Therefore, an ability to plan motions that avoid obstacles is essential to a task planner. Several obstacle avoidance algorithms have been proposed in different domains. In this section, we briefly review those algorithms that deal with robot obstacle avoidance in three dimensions. The algorithms for robot obstacle avoidance can be grouped into the following classes: (1) hypothesize and test, (2) penalty function, and (3) explicit free space.

The hypothesize and test method was the earliest proposal for robot obstacle avoidance. The basic method consists of three steps: first, hypothesize a candidate path between the initial and final configuration of the robot manipulator; second, test a selected set of configurations along the path for possible collisions; third, if a possible collision is found, propose an avoidance motion by examining the obstacle(s) that would cause the collision. The entire process is repeated for the modified motion.

The main advantage of the hypothesize and test technique is its simplicity. The method's basic computational operations are detecting potential collisions and modifying proposed paths to avoid collisions. The first operation, detecting poten-

tial collisions, amounts to the ability to detect nonnull geometric intersections between the manipulator and obstacle models. This capability is part of the repertoire of most geometric modeling systems. We have pointed out in Sec. 10.8 that the second operation, modifying a proposed path, can be very difficult. Typical proposals for path modification rely on approximations of the obstacles, such as enclosing spheres. These methods work fairly well when the obstacles are sparsely located so that they can be dealt with one at a time. When the space is cluttered, however, attempts to avoid a collision with one obstacle will typically lead to another collision with a different obstacle. Under such conditions, a more accurate detection of potential collisions could be accomplished by using the information from vision and/or proximity sensors.

The second class of algorithms for obstacle avoidance is based on defining a penalty function on manipulator configurations that encodes the presence of objects. In general, the penalty is infinite for configurations that cause collisions and drops off sharply with distance from obstacles. The total penalty function is computed by adding the penalties from individual obstacles and, possibly, adding a penalty term for deviations from the shortest path. At any configuration, we can compute the value of the penalty function and estimate its partial derivatives with respect to the configuration parameters. On the basis of this local information, the path search function must decide which sequence of configurations to follow. The decision can be made so as to follow local minima in the penalty function. These minima represent a compromise between increasing path length and approaching too close to obstacles. The penalty function methods are attractive because they provide a relatively simple way of combining the constraints from multiple objects. This simplicity, however, is achieved only by assuming a circular or spherical robot; only in this case will the penalty function be a simple transformation of the obstacle shape. For more realistic robots, such as two-link manipulator, the penalty function for an obstacle would have to be defined as a transformation of the configuration space obstacle. Otherwise, motions of the robot that reduce the value of the penalty function will not necessarily be safe. The distinction between these two types of penalty functions is illustrated in Fig. 10.22. It is noted that in Fig. 10.22*a* moving along decreasing values of the penalty function is safe, whereas in Fig. 10.22*b* moving the tip of the manipulator in the same way leads to a collision.

An approach proposed by Khatib [1980] is intermediate between these two extremes. The method uses a penalty function which satisfies the definition of a potential field; the gradient of this field at a point on the robot is interpreted as a repelling force acting on that point. In addition, an attractive force from the destination is added. The motion of the robot results from the interaction of these two forces, subject to kinematic constraints. By using many points of the robot, rather than a single one, it is possible to avoid many situations such as those depicted in Fig. 10.22.

The key drawback of using penalty functions to plan safe paths is the strictly local information that they provide for path searching. Pursuing the local minima of the penalty function can lead to situations where no further progress can be

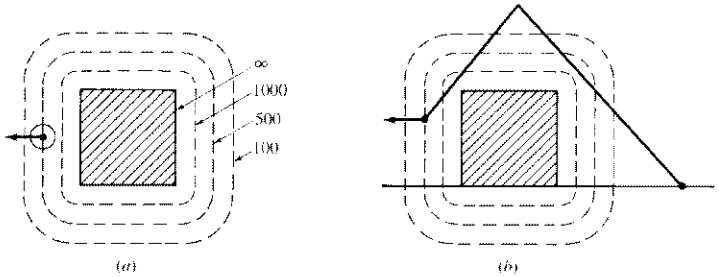


Figure 10.22 Illustration of penalty function for (*a*) simple circular robot and (*b*) the two-link manipulator. (Numbers in the figure indicate values of the penalty function.)

made. In these cases, the algorithm must choose a previous configuration where the search is to be resumed, but in a different direction from the previous time. These backup points are difficult to identify from local information. This suggests that the penalty function method might be combined profitably with a more global method of hypothesizing paths. Penalty functions are more suitable for applications that require only small modifications to a known path.

The third class of obstacle avoidance algorithms builds explicit representations of subsets of robot configurations that are free of collisions, the *free space*. Obstacle avoidance is then the problem of finding a path, within these subsets, that connects the initial and final configurations. The algorithms differ primarily on the basis of the particular subsets of free-space which they represent and in the representation of these subsets. The advantage of free space methods is that their use of an explicit characterization of free space allows them to define search methods that are guaranteed to find paths if one exists within the known subset of free space. Moreover, it is feasible to search for short paths, rather than simply finding the first path that is safe. The disadvantage is that the computation of the free space may be expensive. In particular, other methods may be more efficient for uncluttered spaces. However, in relatively cluttered spaces other methods will either fail or expend an undue amount of effort in path searching.

10.9.3 Grasp Planning

A typical robot operation begins with the robot grasping an object; the rest of the operation is influenced by choices made during grasping. Several proposals for choosing collision-free grasp configurations on objects exist, but other aspects of the general problem of planning grasp motions have received little attention. In this section, *target object* refers to the object to be grasped. The surfaces on the robot used for grasping, such as the inside of the fingers, are *gripping surfaces*. The manipulator configuration which has it grasping the target object at that object's initial configuration is the *initial-grasp configuration*. The manipulator

configuration that places the target object at its destination is the *final-grasp configuration*.

There are three principal considerations in choosing a grasp configuration for objects whose configuration is known. The first is *safety*: the robot must be safe at the initial and final grasp configurations. The second is *reachability*: the robot must be able to reach the initial grasp configuration and, with the object in the hand, to find a collision-free path to the final grasp configuration. The third is *stability*: the grasp should be stable in the presence of forces exerted on the grasped object during transfer motions and parts mating operations. If the initial configuration of the target object is subject to substantial uncertainty, an additional consideration in grasping is *certainty*: the grasp motion should reduce the uncertainty in the target object's configuration.

Choosing grasp configurations that are safe and reachable is related to obstacle avoidance; there are significant differences, however. The first difference is that the goal of grasp planning is to identify a single configuration, not a path. The second difference is that grasp planning must consider the detailed interaction of the manipulator's shape and that of the target object. Note that candidate grasp configurations are those having the gripping surfaces in contact with the target object while avoiding collisions between the manipulator and other objects. The third difference is that grasp planning must deal with the interaction of the choice of grasp configuration and the constraints imposed by subsequent operations involving the grasped object. Because of these differences, most existing methods for grasp planning treat it independently from obstacle avoidance.

Most approaches to choosing safe grasps can be viewed as instances of the following method:

1. Choose a set of candidate grasp configurations. The choice can be based on considerations of object geometry, stability, or uncertainty reduction. For parallel-jaw grippers, a common choice is grasp configurations that place the grippers in contact with a pair of parallel surfaces of the target object. An additional consideration in choosing the surfaces is to minimize the torques about the axis between the grippers.
2. The set of candidate grasp configurations is then pruned by removing those that are not reachable by the robot or lead to collisions. Existing approaches to grasp planning differ primarily on the collision-avoidance constraints, for example:
   a. Potential collisions of gripper and neighboring objects at initial-grasp configuration.
   b. *p* convexity (indicates that all the matter near a geometric entity lies to one side of a specified plane).
   c. Existence of collision-free path to initial-grasp configuration.
   d. Potential collisions of whole manipulator and neighboring objects at initial-grasp configuration, potential collisions of gripper and neighboring objects at final-grasp configuration, potential collisions of whole manipulator and neighboring objects at final-grasp configuration, and existence of collision-free paths from initial to final-grasp configuration.

3. After pruning, a choice is made among the remaining configurations, if any. One possibility is choosing the configuration that leads to the most stable grasp; another is choosing the one least likely to cause a collision in the presence of position error or uncertainty.

It is not difficult to see that sensory information (vision, proximity, torque, or force) should be very useful in determining a stable and collision-free grasp configuration.

10.10 EXPERT SYSTEMS AND KNOWLEDGE ENGINEERING

Most techniques in the area of artificial intelligence fall far short of the competence of humans or even animals. Computer systems designed to see images, hear sounds, and understand speech can only claim limited success. However, in one area of artificial intelligence—that of reasoning from knowledge in a limited domain—computer programs can not only approach human performance but in some cases they can exceed it.

These programs use a collection of facts, rules of thumb, and other knowledge about a given field, coupled with methods of applying those rules, to make inferences. They solve problems in such specialized fields as medical diagnosis, mineral exploration, and oil-well log interpretation. They differ substantially from conventional computer programs because their tasks have no algorithmic solutions and because often they must make conclusions based on incomplete or uncertain information. In building such expert systems, researchers have found that amassing a large amount of knowledge, rather than sophisticated reasoning techniques, is responsible for most of the power of the system. Such high-performance expert systems, previously limited to academic research projects, are beginning to enter the commercial marketplace.

10.10.1 Construction of an Expert System

Not all fields of knowledge are suitable at present for building expert systems. For a task to qualify for "knowledge engineering," the following prerequisites must be met:

1. There must be at least one human expert who is acknowledged to perform the task well.
2. The primary sources of the expert's abilities must be special knowledge, judgment, and experience.
3. The expert must be able to articulate that special knowledge, judgement, and experience and also explain the methods used to apply it to a particular task.
4. The task must have a well-bounded domain of application.

Sometimes an expert system can be built that does not exactly match these prerequisites; for example, the abilities of several human experts, rather than one, might be brought to bear on a problem.

The structure of an expert system is modular. Facts and other knowledge about a particular domain can be separated from the inference procedure—or control structure—for applying those facts, while another part of the system—the global database—is the model of the "world" associated with a specific problem, its status, and its history. It is desirable, though not yet common, to have a natural-language interface to facilitate the use of the system both during development and in the field. In some sophisticated systems, an explanation module is also included, allowing the user to challenge the system's conclusions and to examine the underlying reasoning process that led to them.

An expert system differs from more conventional computer programs in several important respects. In a conventional computer program, knowledge pertinent to the problem and methods for using this knowledge are intertwined, so it is difficult to change the program. In an expert system there is usually a clear separation of general knowledge about the problem (the knowledge base) from information about the current problem (the input data) and methods (the inference machine) for applying the general knowledge to the problem. With this separation the program can be changed by simple modifications of the knowledge base. This is particularly true of rule-based systems, where the system can be changed by the simple addition or subtraction of rules in the knowledge base.

10.10.2 Rule-Based Systems

The most popular approach to representing the domain knowledge (both facts and heuristics) needed for an expert system is by production rules (also referred to as SITUATION-ACTION rules or IF-THEN rules). A simple example of a production rule is: IF the power supply on the space shuttle fails, AND a backup power supply is available, AND the reason for the first failure no longer exists, THEN switch to the backup power supply. Rule-based systems work by applying rules, noting the results, and applying new rules based on the changed situation. They can also work by directed logical inference, either starting with the initial evidence in a situation and working toward a solution, or starting with hypotheses about possible solutions and working backward to find existing evidence—or a deduction from existing evidence—that supports particular hypothesis.

One of the earliest and most often applied expert systems is Dendral (Barr et al. [1981, 1982]). It was devised in the late 1960s by Edward A. Feigenbaum and Joshua Lederberg at Stanford University to generate plausible structural representations of organic molecules from mass spectrogram data. The approach called for:

1. Deriving constraints from the data
2. Generating candidate structures
3. Predicting mass spectrographs for candidates
4. Comparing the results with data

This rule-based system, chaining forward from the data, illustrates the very common AI problem-solving approach of "generation and test." Dendral has been used as a consultant by organic chemists for more than 15 years. It is currently recognized as an expert in mass-spectral analysis.

One of the best-known expert systems is MYCIN (Barr et al. [1981,*1982]), design by Edward Shortliffe at Stanford University in the mid-1970s. It is an interactive system that diagnoses bacterial infections and recommends antibiotic therapy. MYCIN represents expert judgmental reasoning as condition-conclusions rules, linking patient data to infection hypotheses, and at the same time it provides the expert's "certainty" estimate for each rule. It chains backward from hypothesized diagnoses, using rules to estimate the certainty factors of conclusions based on the certainty factors of their antecedents, to see if the evidence supports a diagnosis. If there is not enough information to narrow the hypotheses, it asks the physician for additional data, exhaustively evaluating all hypotheses. When it has finished, MYCIN matches treatments to all diagnoses that have high certainty values..

Another rule-based system, R1, has been very successful in configuring VAX computer systems from a customer's order of various standard and optional components. The initial version of R1 was developed by John McDermott in 1979 at Carnegie-Mellon University, for Digital Equipment Corp. Because the configuration problem can be solved without backtracking and without undoing previous steps, the system's approach is to break the problem up into the following subtasks and do each of them in order:

1. Correct mistakes in order.
2. Put components into CPU cabinets.
3. Put boxes in Unibus cabinets and put components in boxes.
4. Put panels in Unibus cabinets.
5. Lay out system floor plan.
6. Do the cabling.

At each point in the configuration development, several rules for what to do next are usually applicable. Of the applicable rules, R1 selects the rule having the most IF clauses for its applicability, on the assumption that that rule is more specialized for the current situation. (R1 is written in OPS 5, a special language for executing production rules.) The system now has about 1200 rules for VAXs, together with information about some 1000 VAX components. The total system has about 3000 rules and knowledge about PDP-11 as well as VAX components.

10.10.3 Remarks

The application areas of expert systems include medical diagnosis and prescription, medical-knowledge automation, chemical-data interpretation, chemical and biological synthesis, mineral and oil exploration, planning and scheduling, signal interpretation, military threat assessment, tactical targeting, space defense, air-traffic con-

trol, circuit analysis, VLSI design, structure damage assessment, equipment fault diagnosis, computer-configuration selection, speech understanding, computer-aided instruction, knowledge-base access and management, manufacturing process planning and scheduling, and expert-system construction.

There appear to be few constraints on the ultimate use of expert systems. However, the nature of their design and construction is changing. The limitations of rule-based systems are becoming apparent: not all knowledge can be structured as empirical associations. Such associations tend to hide causal relationships, and they are also inappropriate for highlighting structure and function. The newer expert systems contain knowledge about causality and structure. These systems promise to be considerably more robust than current systems and may yield correct answers often enough to be considered for use in autonomous systems, not just as intelligent assistants.

Another change is the increasing trend toward non-rule-based systems. Such systems, using semantic networks, frames, and other knowledge-representation structures, are often better suited for causal modeling. By providing knowledge representations more appropriate to the specific problem, they also tend to simplify the reasoning required. Some expert systems, using the "blackboard" approach, combine rule-based and non-rule-based portions which cooperate to build solutions in an incremental fashion, with each segment of the program contributing its own particular expertise.

10.11 CONCLUDING REMARKS

The discussion in this chapter emphasizes the problem-solving or planning aspect of a robot. A robot planner attempts to find a path from our initial robot world to a final robot world. The path consists of a sequence of operations that are considered primitive to the system. A solution to a problem could be the basis of a corresponding sequence of physical actions in the physical world. Planning should certainly be regarded as an intelligent function of a robot.

In late 1971 and early 1972, two main approaches to robot planning were proposed. One approach, typified by the STRIPS system, is to have a fairly general robot planner which can solve robot problems in a great variety of worlds. The second approach is to select a specific robot world and, for that world, to write a computer program to solve problems. The first approach, like any other general problem-solving process in artificial intelligence, usually requires extensive computing power for searching and inference in order to solve a reasonably complex real-world problem, and, hence, has been regarded computationally infeasible. On the other hand, the second approach lacks generality, in that a new set of computer programs must be written for each operating environment and, hence, significantly limits the robot's flexibility in real-world applications.

In contrast to high-level robot task planning usually requires more detailed and numerical information describing the robot world. Existing methods for task planning are considered computationally infeasible for real-time practical applications.

Powerful and efficient task planning algorithms are certainly in demand. Again, special-purpose computers can be used to speed up the computations in order to meet the real-time requirements.

Robot planning, which provides the intelligence and problem-solving capability to a robot system, is still a very active area of research. For real-time robot applications, we still need powerful and efficient planning algorithms that will be executed by high-speed special-purpose computer systems.

REFERENCES

Further general reading for the material in this chapter can be found in Barr et al. [1981, 1982], Nilsson [1971, 1980], and Rich [1983]. The discussion in Secs. 10.2 and 10.3 is based on the material in Whitney [1969], Nilsson [1971], and Winston [1984]. Further basic reading for Sec. 10.4 may be found in Chang and Lee [1973]. Complementary reading for the material in Secs. 10.5 and 10.6 may be found in Fikes and Nilsson [1971] and Rich [1983]. Additional reading and references for the material in Sec. 10.7 can be found in Tangwongsan and Fu [1979].

Early representative references on robot task planning (Secs. 10.8 and 10.9) are Doran [1970], Fikes et al. [1972], Siklossy and Dreussi [1973], Ambler and Popplestone [1975], and Taylor [1976]. More recent work may be found in Khatib [1980], Requicha and Voelcher [1982], and Davis and Comacho [1984]. Additional reading for the material in Sec. 10.10 may be found in Nau [1983], Hayer-Roth et al. [1983], and Weiss and Allanheld [1984].

PROBLEMS

10.1 Suppose that three missionaries and three cannibals seek to cross a river from the right bank to the left bank by boat. The maximum capacity of the boat is two persons. If the missionaries are outnumbered at any time by the cannibals, the cannibals will eat the missionaries. Propose a computer program to find a solution for the safe crossing of all six persons. *Hint*: Using the state-space representation and search methods described in Sec. 10.2, one can represent the state description by (N_m, N_c), where N_m, N_c are the number of missionaries and cannibals in the left bank, respectively. The initial state is (0,0), i.e., no missionary and cannibal are on the left bank, the goal state is (3,3) and the possible intermediate states are (0,1), (0,2), (0,3), (1,1), (2,2), (3,0), (3,1), (3,2).

10.2 Imagine that you are a high school geometry student and find a proof for the theorem: "The diagonals of a parallelogram bisect each other." Use an AND/OR graph to chart the steps in your search for a proof. Indicate the solution subgraph that constitutes a proof of the theorem.

10.3 Represent the following sentences by predicate logic wffs. (*a*) A formula whose main connective is a $\Rightarrow$ is equivalent to some formula whose main connective is a $\vee$. (*b*) A robot is intelligent if it can perform a task which, if performed by a human, requires intelligence. (*c*) If a block is on the table, then it is not also on another block.

10.4 Show how the monkey-and-bananas problem can be represented so that STRIPS would generate a plan consisting of the following actions: go to the box, push the box under the bananas, climb the box, grasp the bananas.

10.5 Show, step by step, how means-ends analysis could be used to solve the robot planning problem described in the example at the end of Sec. 10.4.

10.6 Show how the monkey-and-bananas problem can be represented so that STRIPS would generate a plan consisting of the following actions: go to the box, push the box under the bananas, climb the box, grab the bananas. Use means-ends analysis as the control strategy.