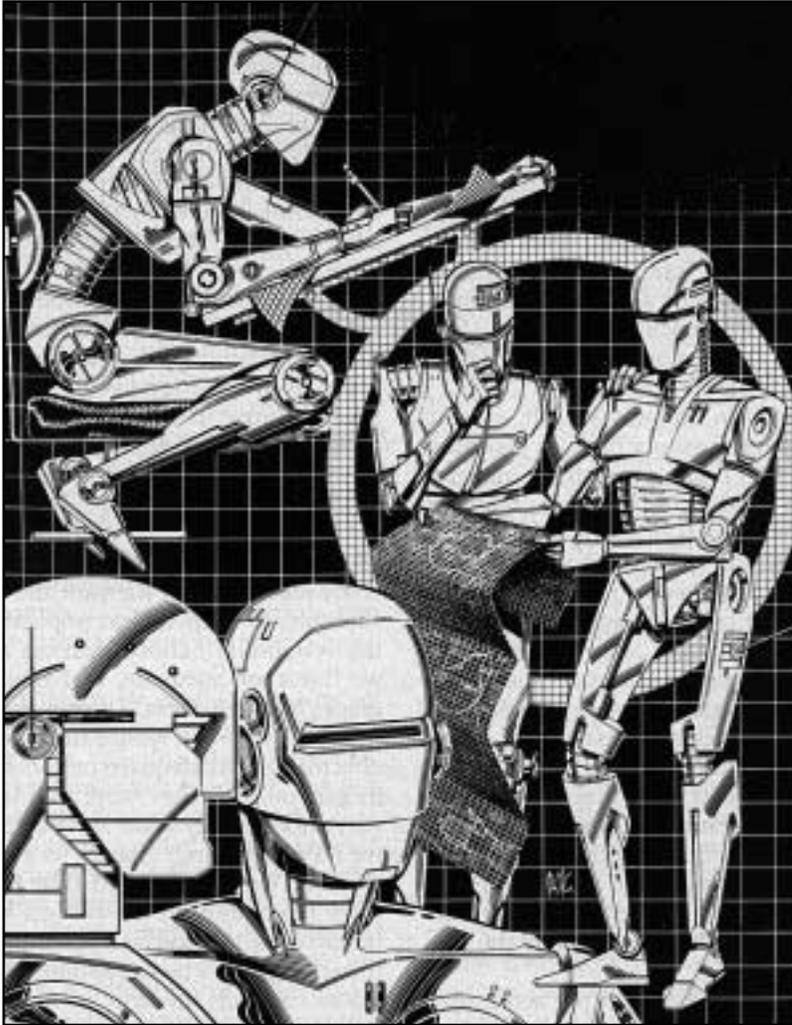


# Robot Planning

*Drew McDermott*



*Research on planning for robots is in such a state of flux that there is disagreement about what planning is and whether it is necessary. We can take planning to be the optimization and debugging of a robot's program by reasoning about possible courses of execution. It is necessary to the extent that fragments of robot programs are combined at run time. There are several strands of research in the field; I survey six: (1) attempts to avoid planning; (2) the design of flexible plan notations; (3) theories of time-constrained planning; (4) planning by projecting and repairing faulty plans; (5) motion planning; and (6) the learning of optimal behaviors from reinforcements. More research is needed on formal semantics for robot plans. However, we are already beginning to see how to mesh plan execution with plan generation and learning.*

*The plan is that part of the robot's program whose future execution the robot reasons about explicitly.*

We used to know what *planning* was. It was the automatic generation of an action sequence to bring about a desired state of affairs. Given a goal such as "All the green blocks must be off the table," a classical planner was supposed to generate a plan such as "Put block 33 on block 12, put block 16 on block 33, put block 45 on block 16," where blocks 16, 33, and 45 were the only green blocks on the table.

Nowadays nobody works on this problem any more. As stated, the problem turned out to be too hard and too easy. It was too hard because it was intractable. (As stated, the problem is so open ended that it has to be intractable, but even when severely simplified, it is still pretty bad [Chapman 1987; Gupta and Nau 1991].) It was too easy because action sequences are not adequate as a representation of a real robot's program. As often happens in AI, we are now trying to redefine the problem or do away with it.

The temptation to do away with it, at least for the time being, is strong because planning the behavior of an organism seems to depend crucially on a model of what the organism can do, and our models of robot behavior are still crude. Perhaps we should build robots that have their own need to plan before we think about robot planning again. I say more about this line of thought later. For now, let's just make the obvious assumption that there are robots whose actions need to be planned to avoid inefficiencies and disasters. A robot that pursued its own immediate agenda without regard to the plans of its makers would be of limited value. Presumably, we will want to tell our robots to do things without having to completely reprogram them. As soon as we tell a robot to do two things at once, it will have to reason about how to combine the two directives.

All robots have programs in the sense that their behavior is under the control of a formally representable object. This statement has been true since Grey Walter's (1950) turtles; I leave aside the question of whether it is true of robots controlled by neural nets. Virtually all other robots are under the control of formal objects that we can call their programs, even though in some cases, the object is compiled down to hardware.

This observation might seem trivial, but I have three reasons for making it:

First, there is an odd current of mysticism in the AI-robotics community. Some practitioners like to pretend that their machines are "being in the world" or "living a life" rather than being controlled by something as anal retentive as a program. I want to put distance

between my views and theirs.

Second, some robot programmers call attention to the fact that their robots continually react to the current sensory input rather than follow a list of instructions without heed to the world around them. The specifications of how to react are nonetheless still a program.

Third, there is a tendency in the field to praise hardware and condemn software. Hardware is speedy and reactive; software spends its time swapping and garbage collecting. This preference is a temporary aberration, I believe, based on a misunderstanding about programming. As in the rest of computer science, robotics can't escape the advantages of first expressing behaviors as textual objects and later worrying about mapping them to hardware.

Once we focus on the robot's program, we realize that the robot's intentions and capabilities are its program. To change how the robot reacts to a situation, you change its formal specifications. This claim is not strictly true. In reality, the development of a robot system proceeds as follows:



We identify what we want the robot for (for example, to keep the rat population down at the warehouse), choose sensors and effectors we think are adequate, and then devise programs to drive them. During program development, we might realize that the sensors and effectors are inadequate or too complex, but in general, it is inevitable that hardware will vary more slowly than software. (That's why we have software.) Hence, to a first approximation, we can pretend that the hardware stays fixed, and all the robot's behavior is a function of the software. If the robot moves toward light, it's because some part of its controller connects the light sensor to the motion effector; without this programmed connection, there would be no tropism.

Robot programming presents several peculiarities (compare Lyons and Arbib [1989]):

First, programs must provide real-time response to changing environmental conditions. This response must be based on a model of how the world reacts to the robot, often stated in control-theoretic terms.

Second, programs are inherently concurrent. Sensors and effectors run in parallel. Many tasks demand only intermittent attention.

Third, some of the objects manipulated by a program lie outside the computer. The program cannot simply bind a variable to such an object but must spend time tracking and reacquiring it.

## What Is Robot Planning?

Now that we've explained robot programming, where does planning fit in? When we started, a plan was a sequence of actions, such as "First put block 33 on block 12, then put block 45 on block 12." If we ask how this sequence of actions fits into the world of programmed robots, the answer is clear: It is just a simple form of program. It's not plausible as a complete program. If it is to actually control a real robot, we must implement "put" as a subroutine that actually moves blocks. This subroutine might or might not be part of the plan itself.

Here I must address a technical point. If we insist that plans actually control behavior, then the plan is usually just a small fragment of a much larger system for controlling behavior, including the put routine but also including the garbage collector, the graphic user interface, and the planner itself. How then do we tell when a robot actually does any planning? How do we find the plans? Surely, it's a trivialization to use the label plan for an arbitrary piece of the robot's software.

Here's the answer: The *plan* is that part of the robot's program whose future execution the robot reasons about explicitly. There might be some residual philosophical issues about what counts as explicit reasoning, but we can, I hope, take a commonsense position on this matter.

A robot that learns by debugging parts of its program is an interesting special case. Here, the system reasons about how to improve the program it just executed, but the reasoning is about how to generalize the current lesson to future similar circumstances. This situation counts as reasoning about the future execution of the program, so the part of the program that is improved by learning is the robot's plan.

Thus, we have a definition of robot plan. The definition of robot planning is a natural corollary: *Robot planning* is the automatic generation, debugging, or optimization of robot plans. This list of three operations is merely a gloss on the phrase "explicit reasoning about future execution." There might be other operations that should be included.

This approach is tidy, but it has the unfortunate consequence of having defined planning to be a form of automatic programming of robots, and automatic programming has not been a hugely successful endeavor. In fact, its evolution parallels that of robot planning in a way: It has progressed from elegant, intractable formulations to more task-oriented and modest techniques.

Another issue raised by my formulation is

that it is more difficult to state the general form of a planning problem. The classical format is as descriptions of an initial situation and a goal situation. In robot planning, we sometimes want to devise a plan that keeps a certain state true, devise the most efficient plan for a certain task, react to a type of recurring world state, or some combination of all these things. Perhaps the most general formulation of the planning problem is as follows:

Given an abstract plan  $P$  and a description of the current situation, find an executable realization  $Q$  that maximizes some objective function  $V$ .

For example, a scheduling problem might be thought of as transforming (IN-SOME-ORDER  $A_1 A_2 \dots A_n$ ) into a particular ordering of  $A_i$  with the object of minimizing total expected time to completion. A classical planning problem might be thought of as transforming the abstract plan (ACHIEVE  $p$ ) into a sequence of actions such that  $p$  is true when the sequence is executed, starting with the current situation.

No existing planner can accept plans stated in this extremely general form, and it is not likely that such a general planner will ever be written. However, it is useful to keep the framework in mind when comparing the different approaches to planning that are surveyed in what follows.

## A Survey of Robot Planners

My goal in this section is to give you an impression of the current state of the field, including solid results and dreamy aspirations. You can be the judge of the ratio of the mass of the former to the volume of the latter. Another goal is to try to fit much of the current work in this field into a single notational framework to make comparison easier among a bewildering variety of approaches.

Much of the work on robot planning involves simulated robots because experimentation with real robots is difficult. Even researchers who use the real thing must inevitably do most of their work on software imitations. We hope that the results we get are not warped too much by having been generated in this context, but we know they are warped to some extent. Keep this point in mind.

### Minimalism

I start with attempts to treat robotic behavior as a matter of stimuli and responses. Behavior at a given moment is to be controlled by the situation at the moment. Elaborate mechanisms for building world models and making

*...planning  
is... a matter  
of deliberating  
about the  
future to  
generate a  
program...*

inferences from them are to be avoided. This approach is associated most prominently with the research group of Brooks (1986) at the Massachusetts Institute of Technology. It also includes the work on situated automata by Kaelbling and Rosenschein (1990) and Kaelbling (1987) and video game players by Agre and Chapman (1987).

For example, here is the specification in Brooks's notation for a module to make a robot run away from a disturbing force (Connell [1990]):

```
(defmodule runaway
  :inputs (force)
  :outputs (command)
  :states
    ((nil (event-dispatch force decide))
     (decide (conditional-dispatch
              (significant-force-p force)
              runaway
              nil))
     (runaway
      (output command (follow-
                       force force))
      nil))) .
```

This description is of a finite-state machine with three states: nil, decide, and runaway. The machine waits in state nil until an event is detected on the input line named force. It then enters state decide, where it checks whether the force is large enough to cause a branch to state runaway or a return to the quiescent state nil. In state runaway, an output is issued to the command output line.

Notations such as these do not make it impossible to write programs that maintain complicated world models, but they discourage the practice. The idea is that a complex world model is just an opportunity for error. If the robot must instead track sensory data moment by moment, small errors cannot accumulate; they get washed away by new data. A slogan of this movement is, Use the world as its own best model.

Some of the robots designed with these notations do cute things, but so do robots designed using Pascal. Because my topic is robot planning, the relevant question here is, To what extent do the notations support planning? They could in two ways: (1) they could be used to write planners and plan interpreters or (2) they could be notations for plans to be manipulated by a planner. The two alternatives are not necessarily exclusive (Nilsson 1988).

An example of the first approach is the module in Chapman's (1990) SONJA program, which is responsible for deciding which way to navigate around obstacles. SONJA is a reactive program whose task is to play a video

game of the usual sort, involving fighting with monsters and ghosts. It does almost no planning, but we can occasionally see glimmers. In figure 1, we see a simplified version of a typical SONJA navigation problem. The hero, on the right, is trying to get to the treasure, on the left, but there is an obstacle in between. To find the shortest path, the system uses visual routines to color the obstacle, find its convex hull, and find and mark the centroid of the convex hull. It also marks the hero (SONJA) and the treasure. If the angle shown in figure 2 is positive, then the planner chooses a counterclockwise path around the obstacle; otherwise, it chooses a clockwise path.

If this approach is planning, then it's as minimalist as you can get. The ingredients are there: Two alternative courses of action are compared with respect to their consequences (a rough estimate of the travel times). The action with the best consequences is then chosen. What's interesting is that one can do even this level of reasoning with the visual-marking hardware Chapman provides.

Thus, in the case of SONJA, plans form only a tiny and evanescent piece of the program. SONJA plots a course, takes the first step, and thinks again, regenerating the plan if need be (or a variant).

The structure of the SONJA navigator raises another issue. Suppose an agent has constructed a plan and begins to execute it. What is the optimal point at which to discard the plan and reconstruct it? If computation costs nothing, then the plan should be discarded as soon as it begins to be executed because the agent can only have gained information as time passes, which the new plan will reflect. Under such circumstances, the optimal strategy is to plan, take a step, replan, take a step, and so forth. It might seem as though the conditions under which this strategy makes sense would be rare but consider an agent that has a processor dedicated to planning. Such an agent should adopt a strategy of installing a new plan whenever the specialized processor generates one. (I am neglecting the cost of plan switching, the option of gathering more information, and the possibility that giving the planner more time allows it to find a better plan. See the sections Theories of Time-Constrained Planning and Transformational Planning).

The point is that planning is not a matter of generating a program and then becoming a slave to it. It is a matter of deliberating about the future to generate a program, which need not be executed in its entirety. It might seem

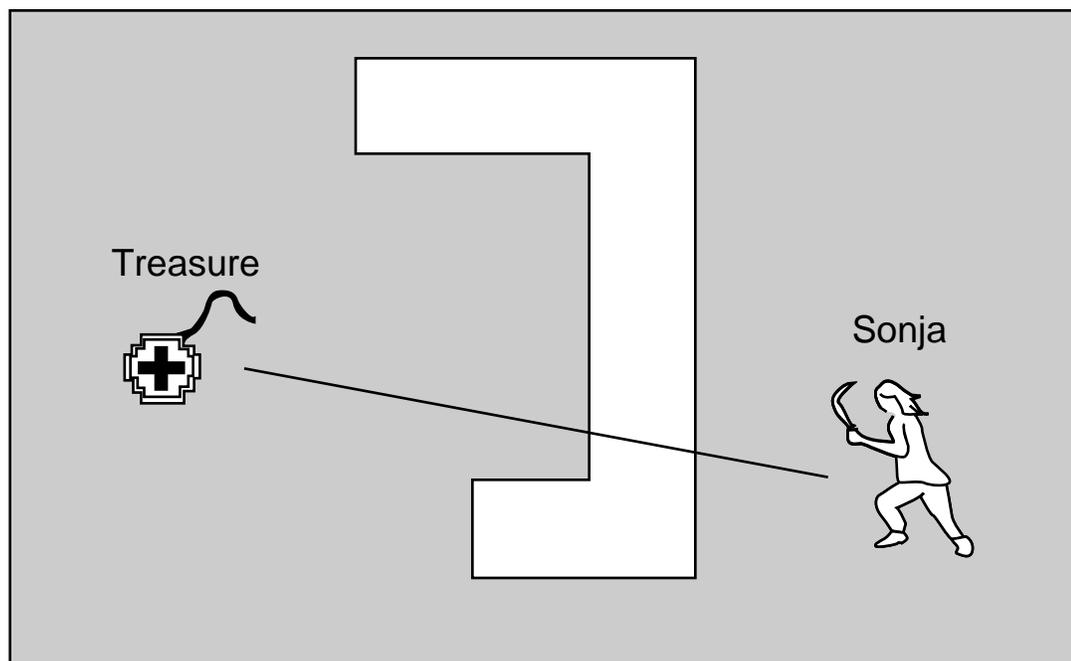


Figure 1. Navigation in SONJA (Chapman 1990).

odd to generate an entire program and then use only an initial segment of it, but the agent is going to have to discard the plan eventually, and the generation of the entire plan legitimizes the initial segment by showing a plausible way for it to continue. A chess-playing program illustrates the point well: Such a program explores a tree of board positions, which could be thought of as containing an elaborate plan for how to respond to the opponent's possible moves. However, the program just finds the best first move and never actually attempts to remember the whole tree of board positions. It is always more cost effective to regenerate the tree after receiving the opponent's next move than to keep the old version around. The same principle applies to planning in general, except when planning is too expensive, or too little information has come in since the plan was revised.

The other way planning can interact with minimalism is for a planner to generate a minimalist robot plan, starting from a more abstract specification. The most cited example of such a system is Kaelbling's (1988) GAPPS program. It takes as a specification a set of states of affairs to be achieved or maintained and compiles them down into programs in a minimalist formalism called REX (Kaelbling and Wilson 1988). For our purposes,<sup>1</sup> we can take a REX program to be of the form

$$\begin{aligned} &(\text{WHENEVER CLOCK-TICK}^* \\ & \quad (\text{TRY-ONE } (\text{cond}_1 \text{ action}_1) \\ & \quad \quad (\text{cond}_2 \text{ action}_2) \\ & \quad \quad \dots \\ & \quad \quad (\text{cond}_n \text{ action}_n))) , \end{aligned}$$

where one action whose cond is satisfied is selected each time TRY-ONE is executed. Each  $\text{cond}_i$  and  $\text{act}_i$  is implemented as a simple combinational logic circuit connected to sensors or effectors. CLOCK-TICK\* is a condition set by an internal clock that alternates between true and false over an interval long enough to allow all the condition and action circuits to settle (that is, not long). In parallel, all the conditions are checked, and one action corresponding to a true condition is executed. I call such a plan *synchronous*, using an analogy with logic circuitry. By contrast, an *asynchronous plan* is one whose actions are not yoked to a global cycle in this way but that poll the world when necessary (or get interrupted by incoming sensory information). A robot's program can have both synchronous and asynchronous components, even within its plan.

One nice feature of the (synchronous) REX formalism is that it permits a natural definition of conjoining and disjoining programs. To disjoin two programs, we simply concatenate their lists of condition-action pairs. Conjunction is slightly more complicated.

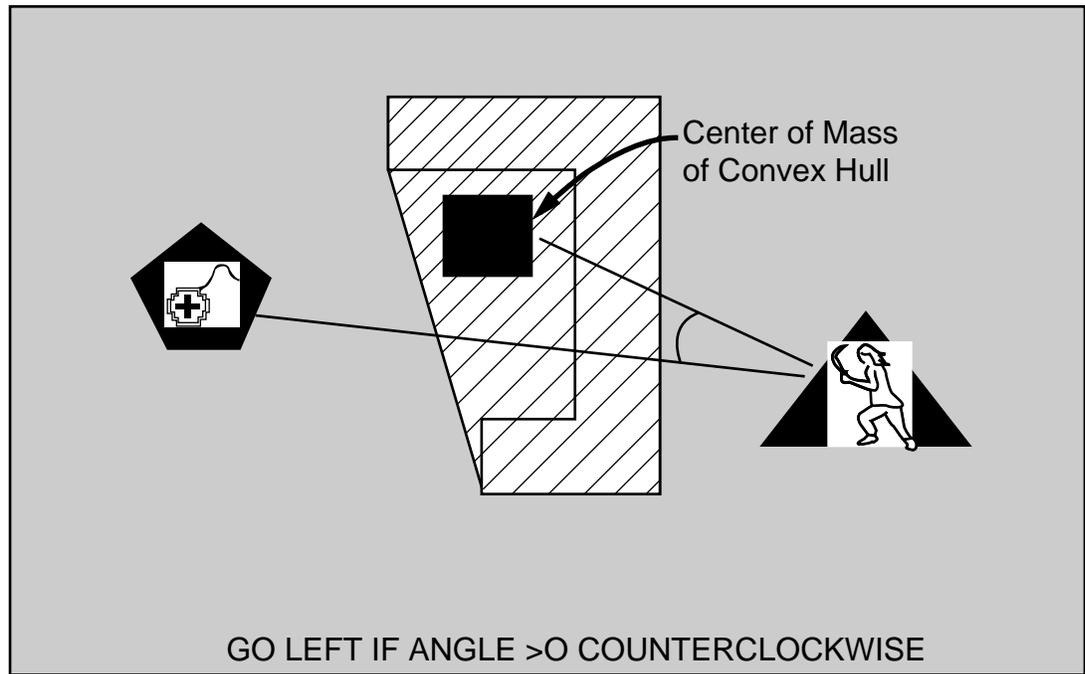


Figure 2. SONJA's Decision Procedure.

Wherever program 1 specifies  $a_1$  as a response to  $c_1$ , and the program specifies  $a_2$  as a response to  $c_2$ , the conjunction recommends the merge of  $a_1$  and  $a_2$  as the response to  $c_1 \wedge c_2$ , provided  $a_1$  and  $a_2$  can be merged. Two actions can be merged if it makes sense to send their effector messages simultaneously (for example, "Right-wheel motor on" and "Front sonar on" can be merged, but "Right-wheel motor on" and "Right-wheel motor off" cannot).

The fact that REX programs compose is important to the definition of Kaelbling's (1988) GAPPS compiler, which transforms programs of the form (AND (ACHIEVE  $p$ ) (ACHIEVE  $q$ )) into executable programs of the sort described. It does so by walking through the goal specification and applying transformation rules. To compile an AND, GAPPS simply applies the definition just discussed. To compile an ACHIEVE, it makes use of rules of the form

(DEFGOALR ([ACHIEVE | MAINTAIN] *state*)  
*subgoal*) ,

which means that any action that leads to the subgoal will lead to the achievement or maintenance of the state. (These ideas are based on the situated automata theory of Rosenzchein [1989].) A collection of DEFGOALRS is a *plan library*, which spells out how to attack all the complex goals the agent knows about. We see this idea again later. Kaelbling (1988) proves that the plans produced by GAPPS actu-

ally do lead to the goals discussed, but this guarantee is weaker than it sounds. Given an impossible problem, GAPPS can detect the impossibility, but it can also go into an infinite loop and never find a solution, or it can produce a plan that loops forever, indefinitely postponing achieving the impossible goal. The DEFGOALR library is not enough of a formal theory of how the world reacts to the robot's actions for GAPPS to do any better.

### Plan Interpreters

There is a trade-off between the expressivity and the manipulability of a plan notation. The notations we looked at in the section on minimalism are at one end of the spectrum. They are sharply constrained to be guaranteed easy to manipulate. In this section, I look at plan notations that allow more complex plans to be stated more easily. I use the term plan interpreter for the module that executes a plan written in one of these notations. There is no real need to draw a contrast here between interpretation and compilation, but thinking in terms of an interpreter makes it easier to experiment with more complex language features (Firby 1987, 1989; Georgeff and Lansky 1986, 1987; Simmons 1990, 1991; Lyons, Hendricks, and Mehta 1991).

Oddly enough, one of the most controversial features is a program counter. In a plan interpreter, it is the most natural thing in the

world to write a plan of the form (SEQ  $step_1 \dots step_n$ ), which means to do each of the steps in order. Indeed, classical plans were purely of this form. The danger with SEQ is that it encourages hiding a world model in an overly simple place—in the counter that tracks which step to do next. It is often the case that the next step is the right thing to do only if previous steps have had their intended effects. The plan should really check to see if the effects really happened, and SEQ makes it easy to omit the check. A better representation for a two-step plan whose second step depends on the first achieving  $P$  might be

```
(WITH-POLICY (WHENEVER (NOT  $P$ )  $step_1$ )
(WAIT-FOR  $P$   $step_2$ )) ,
```

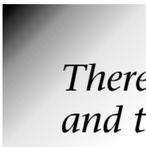
where (WITH-POLICY  $p a$ ) means to do  $a$  and  $p$  in parallel until  $a$  succeeds, giving  $p$  a higher priority.

This plan skeleton is only suggestive. In reality, we might have to spend some resources checking the truth of  $P$ , especially if it involves objects outside the robot (see later discussion). In addition, we have to make sure that if  $step_1$  cannot actually achieve  $P$ , then the robot eventually gives up. With these gaps filled, the WITH-POLICY version is more robust and responsive than the version using SEQ.

Still, there are times when SEQ is handy. Sometimes it is silly to check for the effects of a series of actions after every step. The robot might want to look for an object, move its hand to the location found, close the gripper, and then check to see if it grasped anything. If not, it should move away and repeat the whole sequence. Sometimes the order among plan steps arises for reasons of efficiency rather than correctness. There is no local condition to check between steps; instead, there is a global rationale for doing the steps in this order rather than some other.

In the rest of this article, I use a textual, Lisp-like notation for plans and avoid graphic representations. The reason is simple: Graphic representations become clumsy as soon as we move beyond certain familiar types of plans. Hence, I avoid the standard task network or procedural network and emphasize that plans are programs. I reserve the term *task* to refer to the execution of a particular piece of a plan. It is only in plans with no loops or subroutine calls that each piece of the plan will correspond to exactly one task.

Now, what other features does a plan notation need? Basically, a plan notation needs everything that a robot programming language needs plus features to support reasoning about the plan and the state of its execution. These features include loops, conditionals,



*There is a trade-off between the expressivity and the manipulability of a plan notation.*

and procedures; concurrency (the ability to read more than one sensor; run more than one more effector; and, in general, work on more than one task at once); bound variables; explicit references to tasks (to allow explicit relations among them); constraints as explicit parts of plans (policies); and annotations regarding the purposes of the steps. A *constraint* is an active part of the plan, which acts to repair violations at run time (for example, “If speed exceeds 20 KPH, slow down”). An *annotation* is a note to the planner about the purpose of a piece of the plan (for example, “Keep china you’re carrying intact”). It might or might not be checkable at run time.

The need for bound variables requires further discussion. As I pointed out earlier, a robot needs to be able to refer to objects outside itself. That is, it needs to be able to bind variables that apparently refer to objects in the world:

```
(LET ((X (FIND power-outlet)))
(PUG-INTO X)) .
```

We apparently succeed in binding the local variable  $X$  to an object in the world. We then pass this object to the PUG-INTO plan. This process might look absurd, but the classical plan (SEQ (MOVE B C) (MOVE B C)) more or less assumed that  $A$ ,  $B$ , and  $C$  denoted objects in this way. To back away from this unrealistic assumption, we must make clear that variables can at best be bound to descriptions of objects, that is, to information sufficient to manipulate and reacquire them. I call such a description an *effective designator*. The best a robot can do, in general, is to match a newly perceived object with a description of an object it expects to perceive and jump to the conclusion that they are the same. For example, in a juggling robot, there is a continual process of reading the sensors and assigning the resulting data to one puck or the other to fit a smooth trajectory.

In a planning context, we focus on reasoning about plans that acquire and manipulate objects in this way. The fundamental inference here is an operation we can call EQUATE, which takes two designators and declares that they refer to the same object. For example,

*True planning requires generating, optimizing, or, at least, choosing among plans.*

suppose we have a plan that requires the robot to pick up an object whenever it is dropped. Enforcing this invariant depends on the ability to find the object. The plan for restoring the invariant might look like the following:

```
(DEF-PLAN FIND-AND-PICK-UP (DESIGNATOR)
  (LET((NEW (LOOK-FOR DESIGNATOR)))
    ; NEW is a list of things that look like
    ; DESIGNATOR
    (IF (= (LENGTH NEW) 1)
      (SEQ (EQUATE DESIGNATOR (CAR NEW))
           (PICK-UP DESIGNATOR))
      (FAIL)))) .
```

At the point where the plan equates the two designators, any belief about one is taken to be true of the other. Hence, based on the assumption that LOOK-FOR returned enough information to allow PICK-UP to work, this information is now attached to DESIGNATOR, making it effective in the sense previously described. If LOOK-FOR finds no object of the correct sort—or too many—the plan fails. *Failure* means that the planner must step in and try to find a way to work around the problem. I say more about this particular sort of failure later.

I go into this scenario at such length because there has been a lot of controversy and confusion about the role of names in plans. Agre and Chapman (1990) have gone so far as to suggest that there is a need to rethink the entire institution of representation. In fact, a more modest conclusion is appropriate: Tarskian semantics has nothing to say about how descriptions of objects in plans relate to the objects in the world. Fortunately, the full story is not complicated.

In the plan shown, we made use of an explicit EQUATE operator. Another approach was explored by Firby (1989). He viewed the process of object tracking as an autonomous filter between the planner and the sensors. Every time a set of objects was noticed or examined, this filter would perform the most restrictive match it could manage between the newly sensed objects and objects with descriptions stored in memory. The plan interpreter tracked the status of such matches and would discard information as actions were performed. A key idea was the *expectation set*, or the set of objects that the robot expected to see again in some class of situations. In such a situation, the robot would equate a perceived object with an expected object as soon as the perceived object matched one expected object better than all the rest.

In what follows, I introduce other notational features as we need them. If you get the impression that the notation can be

turned into an actual plan notation, that's because it has been. See McDermott (1991) for a complete description of the reactive plan language.

### Theories of Time-Constrained Planning

We have looked at systems that execute plans. True planning requires generating, optimizing, or, at least, choosing among plans.

Classically, planning was supposed to operate as follows: The planner is given a description of a world state, for example, a description of a stack of blocks. It then generates a sequence of actions that bring this world state about, starting from the current state. Unfortunately, this problem tends to be too hard, especially the general versions. If we take an arbitrary logical formula as the goal specification, then it is not even decidable whether the goal state is consistent, let alone achievable.

However, if we specialize the problem, then the technique of nonlinear planning becomes useful. Suppose that the initial state of the world is known completely, and the goal-state description is just a set of ground literals. In figure 3, we see a graphic depiction of a typical planning problem: Get object *A* on *B*, and change its color to black. Suppose further that actions require preconditions; otherwise, their effects do not depend on what's true before they begin. In this case, we can consider a plan to be (almost) a tree whose nodes are actions and whose edges are goals. An edge joining two action nodes  $A_1$  and  $A_2$  means that the goal labeling the edge is achieved by  $A_1$  as a precondition for  $A_2$ . The actions are to be executed in postorder, that is, children before parents. The root of the tree is labeled END and corresponds to an artificial no-op at the end of the plan. The edges into the END node correspond to the goals originally given to the system. The planner can produce a plan by starting with a degenerate tree consisting solely of an END node that connects to one edge for each given goal. An edge corresponding to an as-yet-unsatisfied goal has a dummy goal node on its lower end, which needs to be replaced by a legal action. Planning occurs by repeatedly replacing a goal node with an action node, which, in turn, connects to goal edges and nodes corresponding to its preconditions. The process ends when all goal edges are labeled with goals true in the initial state. All such edges can be completed with an artificial BEGIN action. An example is shown in figure 4. (The action (TO-TABLE  $x$   $y$ ) moves  $x$  from  $y$  to the table; (FROM-TABLE  $x$   $y$ ) moves  $x$  from the table to  $y$ .)

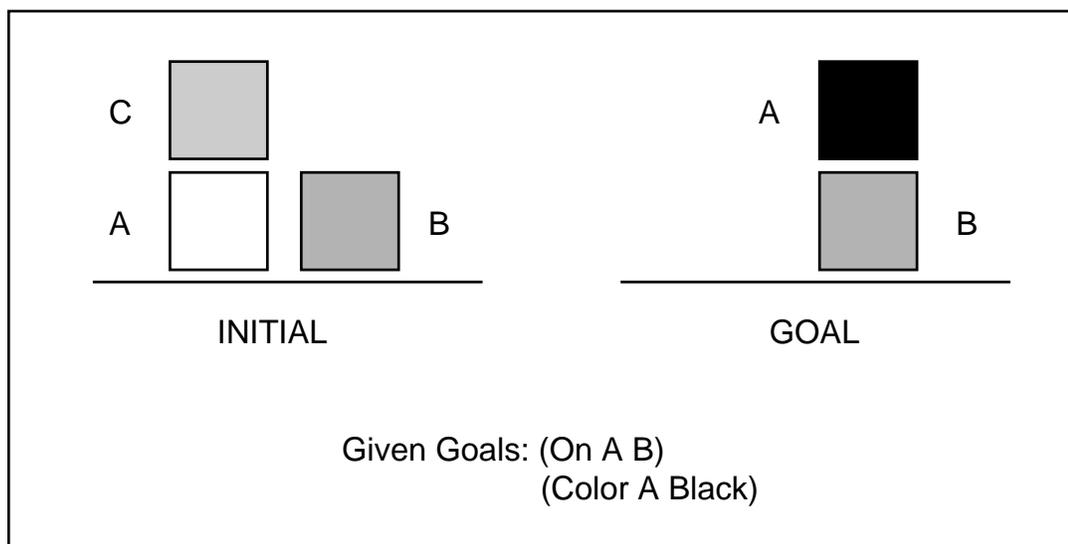


Figure 3. Simple Planning Problem.

To convert the graph of figure 4 back to a piece of text, we use the PARTIAL-ORDER construct. Each action node (except BEGIN and END) is given a label using the notation (:TAG label action). The resulting code fragments are bundled as follows:

```
(PARTIAL-ORDER ((:TAG STEP1 (TO-TABLE C A))
                (:TAG STEP2 (FROM-TABLE A B))
                (:TAG STEP3 (PAINT A BLACK)))
  (:ORDER STEP1 STEP2)) .
```

The locution (PARTIAL-ORDER (-steps-) -orderings-) means to do the steps as constrained by the orderings but with no other constraints. An alternative representation would be to choose an arbitrary total ordering that extends the partial ordering specified by the plan graph. This representation would be preferable if we did not want the plan executor to worry about concurrency.

Of course, it's not really this easy to generate plans, even with our stringent assumptions about the representation of actions and plans. My tree picture is too simple; in general, an action can achieve more than one goal as preconditions for one or more later actions, so the plan graph is really a directed acyclic graph (DAG). The figure displays this phenomenon for the artificial BEGIN action, but it can occur at any node in the plan graph. If one node in the tree deletes a state protected elsewhere, then the possibility exists of a protection violation in which the deleted state is executed between  $A_1$  and  $A_2$  of the protection. The planner must insert extra edges to ensure that this execution can't happen. (An edge in the plan DAG is traditionally called a *protection* because it records that the achieved

state must persist—be protected—until it is needed.)

All these considerations lead to choice points and, hence, a search process (schematized in figure 5), which is exponential in the worst case (Chapman 1987). The nodes in the search space are partial plans, DAGs with some number of unsatisfied goal nodes and threatened protections. The planner moves through the plan space using these operators (Tate 1977; Charniak and McDermott 1985):

1. For some protection of  $G$  across  $A_1 \rightarrow A_2$  and potential violator  $A_v$ , add an edge  $A_v \rightarrow A_1$ .
2. For some protection of  $G$  across  $A_1 \rightarrow A_2$  and potential violator  $A_v$ , add the edge  $A_2 \rightarrow A_v$ .
3. Replace a goal node with a new action node plus goal edges and nodes for its preconditions.
4. Replace a goal node with an existing action node that achieves this goal.

Recently, McAllester and Rosenblitt (1991) showed that if these rules are codified, the result is a search algorithm that is *complete* in that it is guaranteed to find every plan and *systematic* in that it never looks at the same plan twice.

The algorithm really becomes practical only when it is combined with techniques for

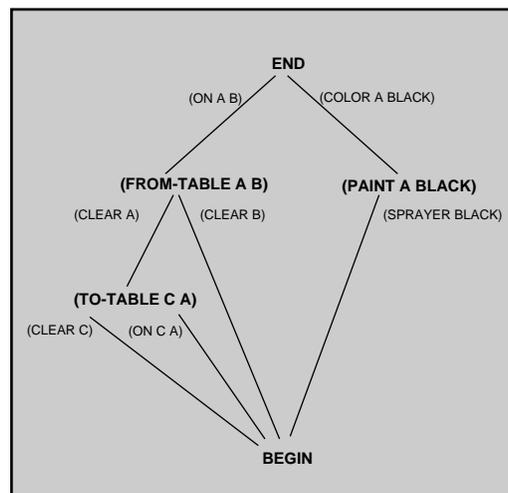


Figure 4. Plans as Action Graphs.

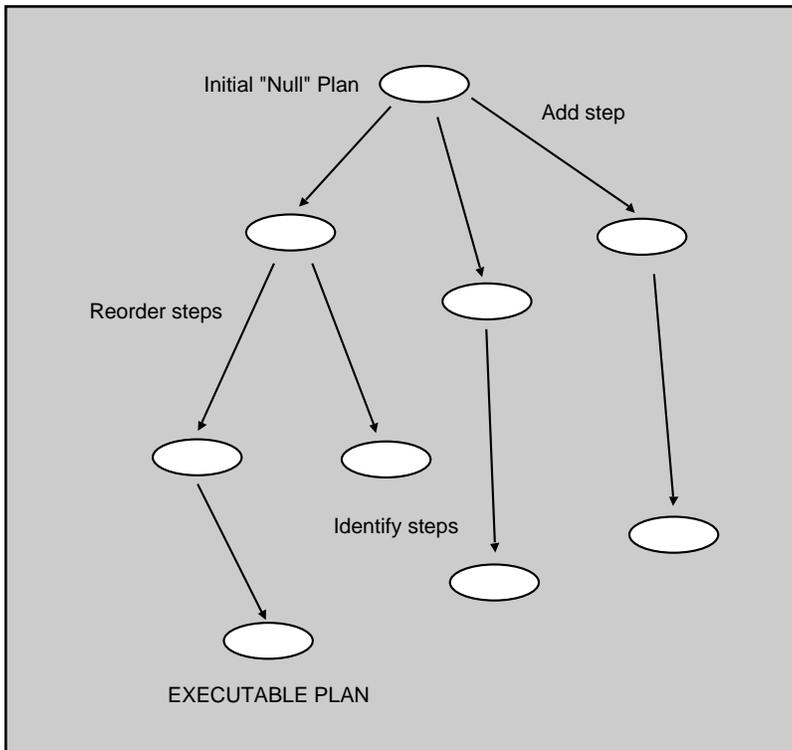


Figure 5. Planning Search Space.

postponing the choice of objects (for example, a destination for object C in figure 3) by allowing the objects to be represented as unknowns when first introduced into a plan. Constraints accumulate for the unknowns until particular objects can be chosen (Sussman 1975; McAllester and Rosenblitt 1991).

Nonlinear planning is effective to the degree that plan steps do not interact, thus making it possible to think about different sections of the plan tree in isolation. We remove one source of interaction by requiring that actions' effects depend only on their arguments. The planner can then treat these effects as stable when reasoning about their impacts on other actions. This requirement might seem impossible to meet, but we can often satisfy it with the trick of including an action's context in its arguments. For example, in the domain of figure 3, we might on first analysis want to include an action (TABLE  $x$ ), which moves object  $x$  to the table. However, this action has a context-dependent effect: It creates empty space on top of the object that used to hold  $x$ . We can patch around this problem by using the action (TABLE  $x$   $y$ ), where  $y$  is the object holding  $x$  before the move. This action always creates empty space on  $y$ . When the planner intro-

duces an instance of it in the plan, it must choose a  $y$ ; here is where the ability to postpone such choices comes in handy.

The amount of literature on heuristic extensions to this basic idea is large. See Hendler, Tate, and Drummond (1990) for a survey of this literature and Allen, Hendler, and Tate (1990) for a sample. See Wilkins (1988) for a description of a large implemented system.

The term nonlinear planning has been used to refer to the technique of searching through a space of plans represented as DAGs. Unfortunately, it has also been used in at least two other ways.<sup>2</sup> Rich (1983) uses it to refer to any algorithm that can find a plan for achieving  $p \wedge p_2 \wedge \dots \wedge p_n$  that does not consist of a sequence of  $n$  subplans, each of which achieves one of the  $p_i$  but leaves the previously achieved conjuncts undisturbed.<sup>3</sup> Another rather different use of the term assumes that the planner is given a library of plans at the outset and not just a description of the effects of each type of action. Given a set of goals, the planner retrieves plans for accomplishing them and then runs these plans, in general, concurrently.<sup>4</sup> (Kaelbling's [1988] DEFGOALR fits this paradigm.) If the plans interfere with each other, then the agent must cope with the resulting problems, either by foreseeing and forestalling them or by correcting them after they happen. For example (figure 6), a robot might have the goal of removing dirt from a room by repeatedly vacuuming it up and emptying the vacuum cleaner bag, and it might have a contemporaneous goal of recharging its batteries whenever they need it. These two plans can simply be run simultaneously. The second is normally dormant, but when active, it takes priority and can cause the robot to make a detour with a full bag of dirt. If there is a reason why this idea is bad, the planner could predict when the batteries would need recharging and could schedule recharging trips during periods when the bag is empty. (I say more about such bug projection in the section on transformational planning.)

The technique of Nau, Yang, and Hendler (1990) is in the category of techniques for managing plan libraries. Suppose that the planner has derived a plan of the form

$$\begin{aligned}
 &(\text{SEQ} \quad (\text{DO-ONE-OF } P_{11} P_{12} \dots P_{1n_1} \\
 &\quad (\text{DO-ONE-OF } P_{21} P_{22} \dots P_{2n_2} \\
 &\quad \dots \\
 &\quad (\text{DO-ONE-OF } P_{m1} P_{m2} \dots P_{mn_m})) \cdot
 \end{aligned}$$

That is, it has determined that any  $P_{ij}$  accomplishes step  $i$ . The only remaining question is which combination of the step plans is the

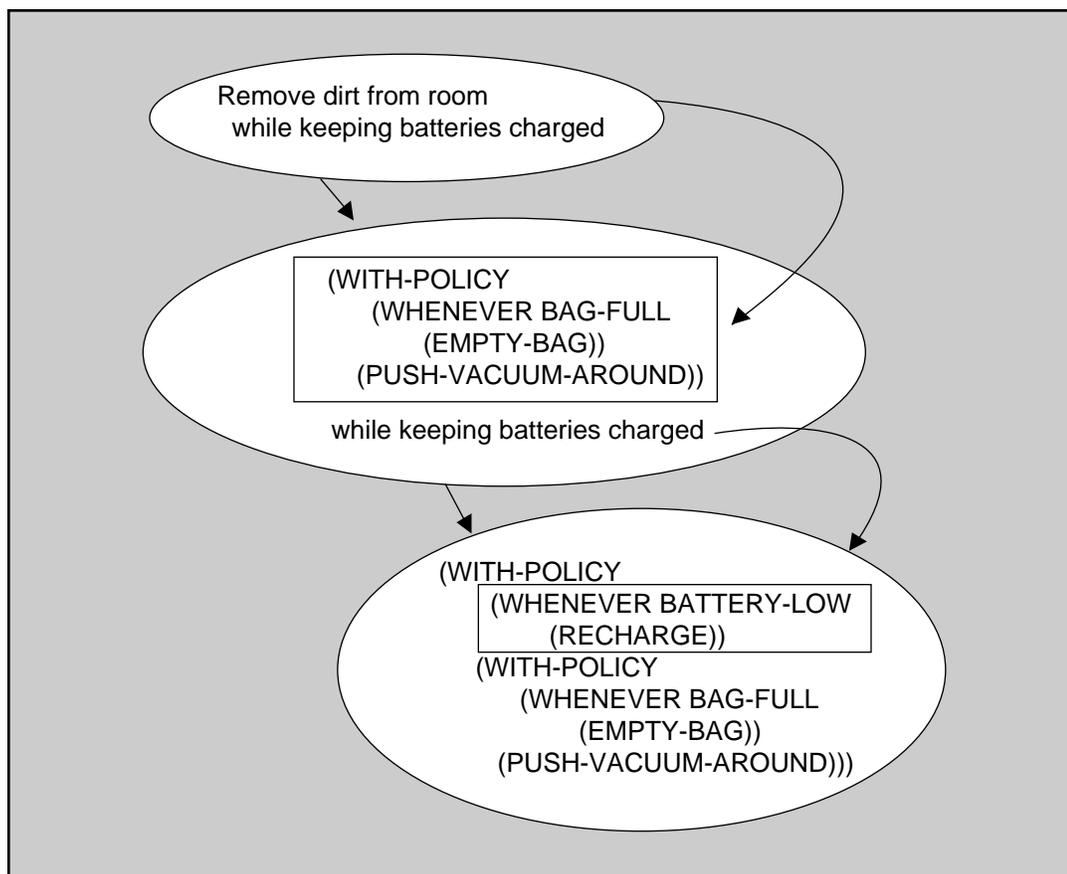


Figure 6. Plan Reduction Using Libraries.

least expensive. It might happen that, say,  $P_{12}$  and  $P_{22}$  are a particularly felicitous combination because their setup costs can be shared. Suppose that the cost sharing can be modeled in terms of the abilities of the steps for each plan  $P_{ij}$  to merge, and suppose these gains from merging are all known. That is, for each pair of action types, we store a new action type, if there is one, that accomplishes both of them more cheaply. The most obvious example is the pair  $\langle A, A \rangle$ , which can be merged to  $A$  in many cases. Two plans,  $P_{ij}$  and  $P_{kl}$ , can be merged by performing all steps consistent with ordering constraints. The algorithm of Nau, Yang, and Hendler (1990) finds optimal plan choices by doing a best-first search through choices of which  $P_{ij}$  to include at step  $i$ . It makes the choices in order of increasing  $i$ , choosing at each stage the  $P_{ij}$  that minimizes the estimated total plan cost. The performance of this kind of algorithm depends on the heuristic cost estimator, which must predict all possible savings from merges to be made later. Empirical results

(figure 7) show that the estimator of Nau, Yang, and Hendler (1990) works well in practice.

To this point, we have assumed that planning should be fast but only because of the general principle that all computations should be fast. In the case of robot planning, there is a special urgency because the robot must react to events quickly. Once the need for a plan is perceived, there is usually a bounded amount of time after which there is no point in continuing to search for a plan. For example (Dean and Boddy 1988), an object might be coming down a conveyor belt, and a grasping plan has to be computed while the object is still within reach.

Why should more time help? It might be that the planning algorithm needs a certain amount of time to run. Giving it more time allows it a better chance of finding an answer before time runs out. A more pleasing possibility is that the algorithm satisfies the principle of continuously available output (Norman and Bobrow 1975): It returns a plan no matter how little time it is given but returns a better

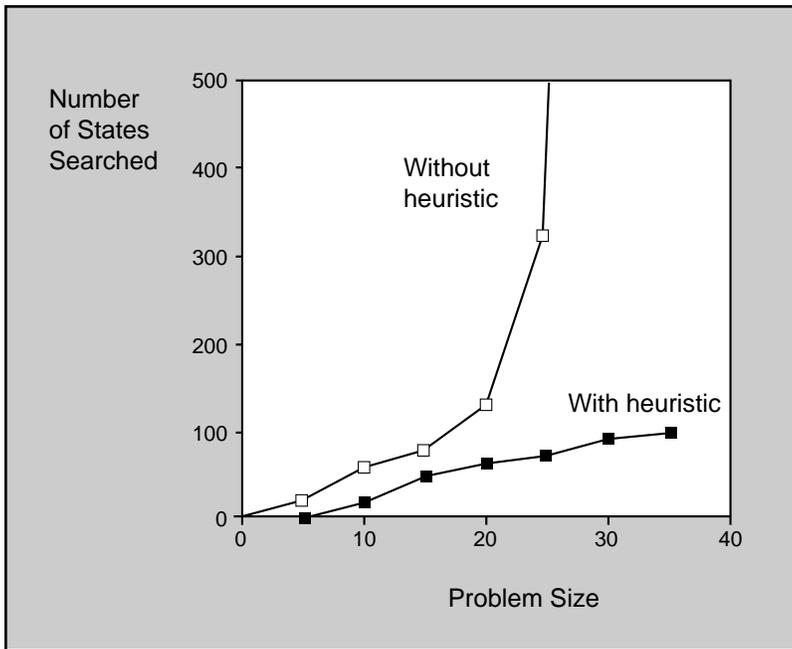


Figure 7. Cost on Random Problems of Algorithm of Nau, Yang, and Hendler (1989).

plan if given more time. Such an algorithm is called an *anytime algorithm* (Dean and Boddy 1988). (See also Horvitz, Cooper, and Heckerman [1989]; and Russell and Wefald [1991].)

Suppose a robot has a model of how much better its plan will get for each time unit. Suppose also that it has a model of how fast the value of a plan deteriorates as its time of delivery gets later. (A hard deadline is a special case where the deterioration happens all at once.) For examples, see figures 8a and 8b. If we take the product of the two graphs, we get a graph that plots the overall value of deliberation (figure 8c). The graph peaks at the optimal time to deliberate, past which the enhanced value of the plan found is offset by its staleness. The planner should plan for this many time units, then execute the plan it has found.

This idea is based on a few assumptions. The first assumption is that the time required to decide how much time to spend deliberating is negligible. An exponential algorithm that finds the absolute best amount of time to spend planning is obviously useless. We make this assumption true by considering only fast algorithms for metadeliberation.

The second assumption is that the only cost of deliberation is the loss of effectiveness because of delay. This assumption is plausible if deliberation requires only central processing unit cycles. We can impose this assump-

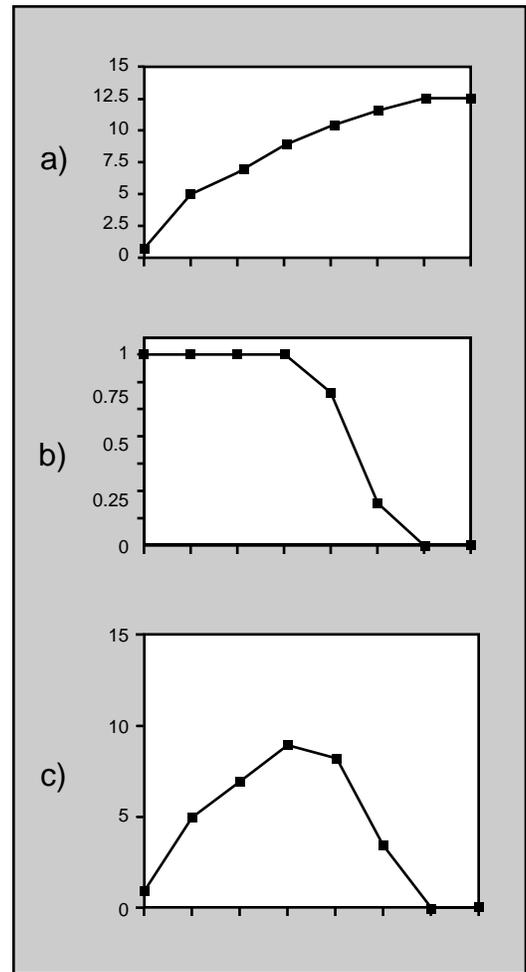


Figure 8. The Value of Deliberation.

(a) Plan goodness as a function of planning time. (b) Plan usefulness as a function of real time. (c) Overall value of planning time.

tion by requiring all overt robot acts to be planned. The best plan might be of the form

Seek information —> plan some more , whose value is the expected value of the best plan obtained after getting the information.

The third assumption is that it is possible to quantify the expected gain from further computation. Figure 9 shows an example where the quantification is possible (Boddy and Dean 1989): Suppose the robot is expected to visit a long list of locations in classical traveling salesman style. The Lin and Kernighan (1973) algorithm works by mutating a legal but suboptimal tour. The more time it is given to mutate, the better the result gets, and we can fit a curve to the degree of improvement. This curve is shown in figure 9a. However, every unit of time spent deliberating must be added to the overall execution time, so a

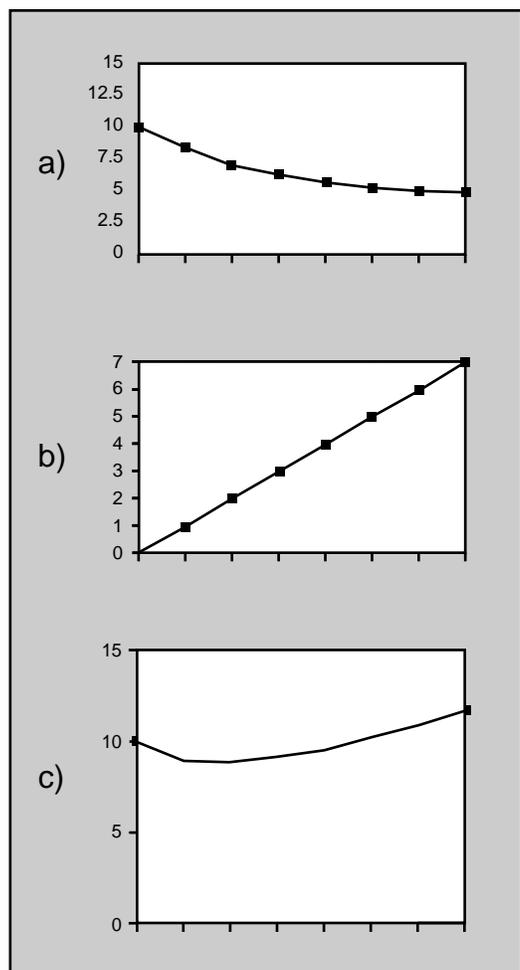


Figure 9. Performance of an Incremental Traveling Salesman Algorithm.

Path travel time (a), delay (b), and total time to completion (c) as functions of planning time.

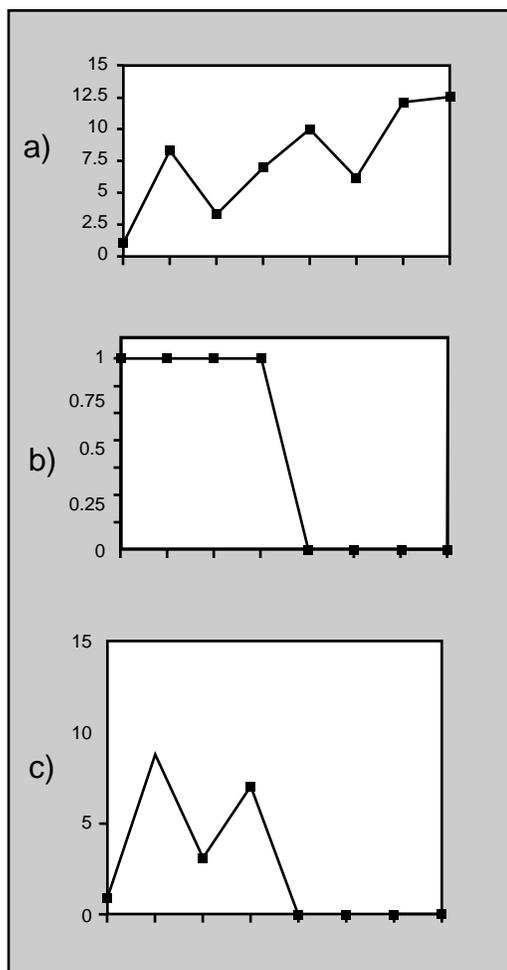


Figure 10. Typical Time-Constrained Planning Scenario.

(a) Plan goodness as a function of planning time. (b) Plan value as a function of real time. (c) Overall value of planning time.

delay penalty is associated with further planning, as shown in figure 9b. The sum of the two curves, shown in figure 9c, is the estimated total execution time as a function of deliberation time. The deliberation time that minimizes total execution time is the one to pick.

Unfortunately, the assumption that we can model the benefits of deliberation is often unrealistic. In fact, further planning can make a plan worse (Nau 1982), although it normally makes it better to some degree. Figure 10 shows a case that exemplifies the limitations of anytime planning. Figure 10a is a graph of plan improvement as a function of plan time. It might seem odd that taking two units of planning time could make the plan worse than taking one unit; why couldn't the plan-

ner just remember the plan it had at time 1 and use it instead of the inferior one it just generated? The answer is that the planner does not always have an accurate model of how good its plans are. The plan after time 1 might have a bug, which it repairs at time 2 (see Transformational Planning). Unknown to the planner, this repair has introduced a worse bug, which requires two more units of plan time to detect and correct. Meanwhile, as shown in figure 10b, there is a deadline looming, after which it is too late to execute any plan at all. In such cases, the best we can do is run the planning algorithm until a deadline looms, then install the current plan and hope for the best. As figure 10c illustrates, this tactic does not necessarily produce the best

Unfortunately  
... we do not  
have a general  
theory of plan  
revision as  
opposed to  
plan  
generation.

```

The china breaks
It was rattled
  Traveled along a bumpy road at speed
  > 20 KPH
    Highway 24 is bumpy
    Traveled along Highway 24
    Speed = 40 KPH
  Not padded
  
```

Figure 11. Explanation of Time Bug.

plan. However, it should, on average, be better than not planning at all (or the planner should be decommissioned).

### Transformational Planning

Many of the planning methods we have examined have in common that they operate on fully formed plans rather than generate a plan from scratch each time. This feature is almost a necessity in a domain where planning involves modifying ongoing behavior. Unfortunately, however, we do not have a general theory of plan revision as opposed to plan generation.

Let's look at an example of why revision is such a good idea. Suppose a robot has to carry out several tasks, including (1) hold onto some fragile china and protect it from breaking until ordered to relinquish it and (2) monitor a nuclear test at location B that takes place at noon. The plan for the second task requires starting for the test site at 9 A.M. The plan for the first task requires picking up the china and avoiding moving anywhere without making sure it is held. Unfortunately, the road from here to the nuclear test observation post is bumpy, so if both these plans are executed simultaneously, the china is likely to get broken. Fortunately, the plan can be repaired in one of the following ways: (1) start for the test site earlier, and avoid going fast enough to break the china; (2) pack the china in a more protective package; or (3) travel by a smoother road. These are all *revisions* of the plan. That is, they are incompatible with the first draft of the plan. The alternative to revising a plan is backtracking to a relevant decision point in the plan space and trying something else. In figure 5, the nodes in the search space were intended to be partial plans, which became further instantiated and constrained by planning operations. In principle, any partial plan could be com-

pleted in several ways, and a plan operator simply discarded some of these completions. If a blind alley was encountered, the planner could switch to a different part of the search space where the correct possibility was still open and try something else. However, in general, this space of refinement decisions will not be represented explicitly, especially if the current version of the plan was produced by composing several large entries in a plan library. It is potentially simpler and more efficient just to provide methods for pushing a plan in the right direction, without having to find a representation in the plan space of an abstract plan that included the correct solution as a possible refinement (Collins 1987). The pitfall is that in revising a plan, it might get worse or cease to solve the original problem at all.

One way to cope with such pitfalls was devised by Hammond (1988, 1990) and Simmons (1988a, 1988b). I abstract their method a little bit and adapt it to the notational framework I have been using. The planner starts with an abstract plan (including pieces such as "Monitor a nuclear test at location B and time 12 P.M."). It uses a plan library to reduce it to executable steps (such as "Wait until 9 A.M., then travel along Highway 24 to the test site. Stay there until noon."). Such plan fragments are likely to work, perhaps are even guaranteed to work, if executed in isolation. However, when they are combined, various interactions can occur. To detect them, the planner projects the plan, yielding an execution scenario that specifies what will happen. (In general, you might get several scenarios. Compare Hanks [1990a, 1990b] and Drummond and Bresina [1990].) At this point, the planner tries to prove that the plan projection represents a successful execution of the original abstract plan. If it fails, then it instead produces an explanation of what went wrong in the form of a proof tree whose conclusion is of the form "Wrong thing *W* occurs" (for example, "The china breaks."). The proof tree might be as shown in figure 11. To fix the problem, the planner looks for leaves of the tree that can be changed. It has no way of changing the bumpiness of Highway 24, but it can change "Traveled along Highway 24" by taking another route; it can change "Speed = 40" by reducing the speed to 20; it can change "Not padded" by introducing a padding step. These repairs are exactly the ones that I listed before. (The speed repair introduces a new bug—that the robot arrives late to the test—which is repaired by starting earlier.) The repairs are affected by adding steps and changing argument values.

## *Ideally... the robot executes the plan while the planner thinks about improving it...*

The work of Hammond and especially Simmons goes a long way toward a general theory of plan revision. For robot planning, we need to think about transformations on arbitrary plans, including those with subroutines and loops. Furthermore, we need to broaden the class of transformations to include optimizations as well as bug repairs. Suppose that a plan critic wants to propose an optimized schedule for a set of errands to be undertaken by a mobile robot. This revision is not naturally construed as a bug repair. Instead, the critic must walk through the current plan, make a list of all the locations to be visited, then alter the plan so that the locations are visited in an efficient order. The PARTIAL-ORDER construct we introduced in connection with nonlinear planning can express the required ordering constraints when used in conjunction with the :TAG notation for referring to pieces of a plan.

Another such transformation is the classic protection-violation removal (see previous section). Suppose one step of a plan requires a fact to be true over an interval, and projection shows that another step can make it false, thereby violating a protection. The standard way to eliminate this possibility is to install ordering constraints to ensure that the violator comes before or after the protection interval. This transformation is easy to express in this language.

Other transformations might require more drastic program revision. Consider the famous bomb in the toilet problem (McDermott 1987). The planner is given two objects, one of which is a bomb. The plan "Put the bomb in the toilet" would save the robot, but it can't be executed: The bomb is not an effective designator (see Plan Interpreters), so it can't be used to give commands to the gripper to pick up the bomb. Classical refinement planning provides ways of verifying that the plan "Put both objects in the toilet" solves the problem but no way of generating the plan.

From a transformational perspective, we can see how such plans can be generated and why humans need the ability to do so. The planner could try projecting the plan

```
(LET ((B (LOOK-FOR-ONE Bomb-shaped object)))
  ;; B is now an effective designator for
```

```
gripper manipulation
(DISARM B))
```

and encounter the bug "More than one object expected to meet description." We can catalog possible repairs just as for other kinds of bugs: (1) find an earlier point in the plan where there was an effective designator for the object, and insert a step to mark it distinctively; (2) find an earlier point in the plan where there was an effective designator, and insert steps to clear the area of distracting objects; or (3) alter the plan after the attempted acquisition of the object so that it applies to every object meeting the description. The first two repairs are not applicable in this situation, but the third is and suggests disarming both objects. In other words, the plan should be revised as follows:

```
(LET ((BL (LOOK-FOR bomb-shaped objects)))
  ;; At this point BL is a list of effective
  designators
  (LOOP FOR ((B IN BL))
    (DISARM B))) .
```

Of course, this transformation is not fool-proof, but then revisions never are. That's why the projection step is so crucial—to test by simulation whether the revised plan actually works. I should also point out that this discussion is a little speculative; no one has implemented this particular transformation.

Let us now turn to the matter of how plan revision fits in with the need to do planning under time constraints. There are two issues: How fast is transformational planning? How can it be combined with plan execution?

The individual steps of transformational planning are not expensive. Plan projection is considerably faster than plan execution. (It depends on processor speed, not the speed of effector motions.) If the planner needs to generate several scenarios, the time taken grows, but if less time is available, the number of projections can be trimmed. Plan critics typically scan a portion of a plan near a bug or, at most, work through a projection, counting things such as resources. Hence, each critic is unlikely to do a huge amount of computation. Perhaps I am hand waving here, but it doesn't matter; the real tar pit is in searching the space of transformed plans. After every revision, the planner must reproject and recri-

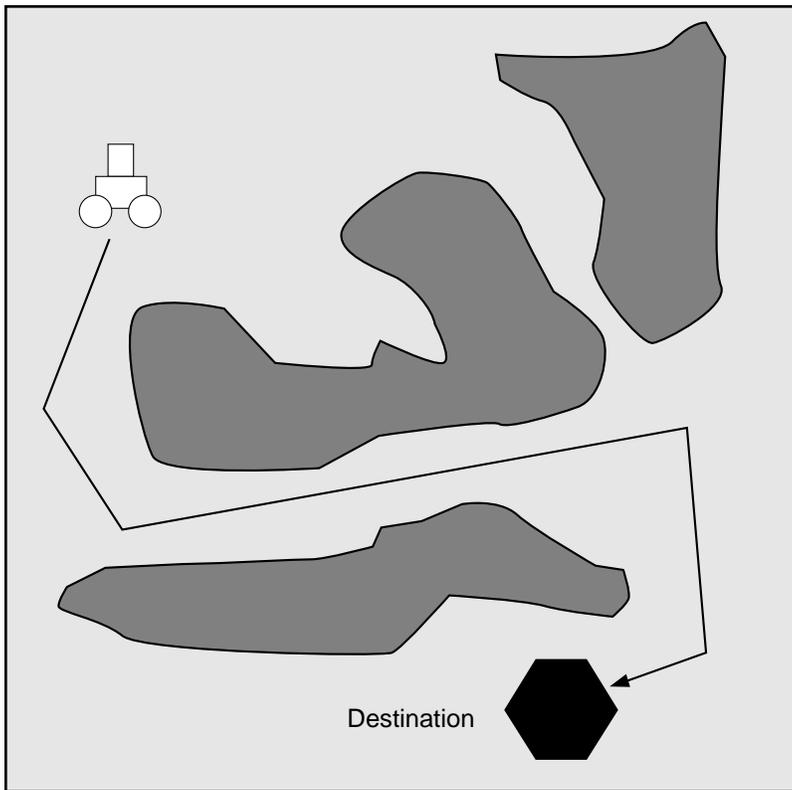


Figure 12. Motion Planning.

tique the plan. Some revisions introduce more bugs than they fix, so the planner must backtrack in revision space. The success of the whole paradigm turns on two assumptions:

First, plans are already almost correct (Sussman 1975). By constructing plans out of large, robust pieces, we don't have to worry that without an intricate debugging process, they will collapse.

Second, the planning process will have paid for itself when the planner finds a better plan than it started with but not necessarily the optimal plan. When this situation happens, it can switch to the improved version.

I know of one case history in which these assumptions were observed empirically to be correct—the transformational planner of Zweben, Deal, and Gargan (1990). This planner is based on an algorithm for rescheduling large-scale plans after revision. The problem is to take a correct schedule for a large set of activities, make a few changes (for example, add some tasks, change some time windows, delete some resources), and find a correct schedule for the new problem. The algorithm treats temporal constraints among steps differently from other constraints. Constraint violations on start and end times are handled

by rippling shifts through the set of tasks, using a technique owed to Ow, Smith, and Thiriez (1988). Other constraint violations are then handled by specialized repair strategies. These strategies are not guaranteed to improve matters, but the algorithm can decide (randomly) to accept an attempted repair that makes the plan worse in the usual simulated-annealing way (Kirkpatrick, Gelatt, and Vecchi 1983). The algorithm has not actually been applied in the context of robot planning, but there is no reason it couldn't be.

The second question I raised earlier was, How can transformational planning be combined with plan execution? In Zweben, Deal, and Gargan's program, as soon as the planner has produced an improved version of the plan, it can tell the interpreter to begin executing it. In this case, the interpreter is a human organization that can see how to make the transition to the new plan. It might be considerably harder to tell a robot to forget its current plan and start work on a new one.

In many cases, it is possible to duck this issue by assuming that the agent does nothing while it plans, then switches to executing the plan. The robot standing by the conveyor belt has no other goals but to plan a grasp as it waits for the target object to get closer. It might be a good strategy to have a robot always react to a new planning problem by discarding its current plan and getting itself into a quiescent state, to pull over to the side of the road as it were.

I see several problems with this view. First, it costs resources to get into a quiescent state, and some planning problems do not require spending them. Second, it often requires planning to choose a quiescent state and then get to it. An autonomous airplane might have to decide whether to circle while planning or find an airport to land at, and if it picks the airport option, it would have to plan how to get there. Third, even quiescent states require behavior. A robot will have to continue to manage power consumption and might have to go looking for fuel in the midst of planning.

Ideally, what we want to happen is that the robot executes the plan while the planner thinks about improving it (McDermott 1990). The planner should be active whenever (1) a new command comes from the robot's supervisor to be added to the current plan, (2) an old command fails, or (3) the planner still has tricks for improving the plan. When the planner has a new plan version, it should instruct the plan interpreter to discard the old one and begin executing the new. It might sound as if this approach could cause

the agent to burst into flames, but if plans are written robustly enough, they should be able to cope. For example, a plan for transporting an object from one place to another must be prepared to discover that the object is already halfway there (or already in the robot's gripper) (compare Schoppers [1987]).

The only thing left out of this sketch is how to cope with a world that is changing too rapidly for the planner to keep up. Suppose the planner takes five minutes to plot a series of errands but meanwhile is cruising along a highway at 100 KPH. The plan it comes up with might apply to the situation at the beginning of the planning process but be useless by the time it is generated (for example, the plan might say "Get off the highway at Exit 10," and the agent might have passed Exit 10 four minutes ago). One possible solution is to track all assumptions about the world state as the planner makes them. If the agent knows that the world has diverged from these assumptions (based on information it has been acquiring as it goes), it can abort the planning process and then attempt to get into a more quiescent state, try giving the planner fewer time resources, or both.

In the old days, the topic of combining planning and execution was called *execution monitoring*. The idea was that the run-time interpreter would track how well the world was conforming to the expectations in the plan and would react to deviations by replanning. This idea is still around but in a modified form. Now that plans themselves continually track the world, we can assume that they explicitly fail when they detect excessive deviations. We do not require a general-purpose theory that tells how to track the progress of an arbitrary plan. What we do require is a theory of how to switch to a new plan when the planner produces one.

### Robot Motion Planning

When real robotics people (as opposed to AI people) use the word *planning*, they usually mean *motion planning*, the computation of trajectories for robots to follow. Actually, this point of view makes sense—a robot doesn't do much more than move. The classical approach to motion planning (Lozano-Perez 1983; Brooks 1982; Latombe 1991) is to start with a complete description of the environment, generate a trajectory through it, then issue commands to the robot wheels or arm so that it moves along this trajectory. The trajectory can be thought of as the plan: "Move while staying close to this curve" (figure 12).

This whole approach is surprisingly problematic. The main problem is that the space

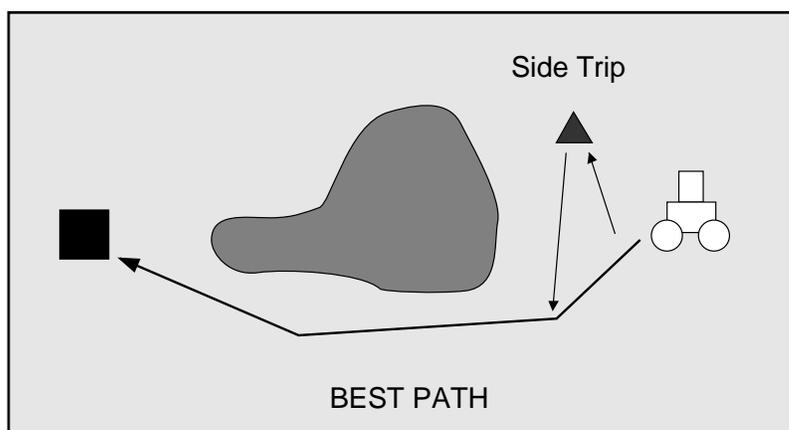


Figure 13. Sticking to a Path Can Be a Bad Idea (Payton 1990).

the path goes through is best thought of as *configuration space*, the space of possible joint positions of the robot, rather than physical space. Configuration space has as many dimensions as degrees of freedom in the system, and the path-planning problem becomes intractable as the degrees of freedom grow (Canny 1988). I more or less ignore this issue and focus on large-scale path planning for mobile robots, where the dimensionality remains a comfortable two.

Another problem is that the method appears to require accurate knowledge of the physical layout before execution begins. Obviously, such knowledge is hard to come by in most circumstances.

A third problem is that the method breaks the problem into two chunks: Find a path; stay near it. The path is generated using geometric criteria and might not be suitable for a real robot to follow. If it has sharp corners, it will be almost impossible for the robot to track it accurately. Of course, it's usually unnecessary to track it accurately, but there will be some tight squeezes.

In fact, there are times when it is inappropriate to track the path at all. If a robot is executing several plans concurrently, then it might interrupt its main travel task to make a side trip or attend to a high-priority interrupt. When the interruption is over, there's no particular reason to return to the previous path (figure 13). Instead, you want to start again toward the destination from the current point. One way to represent the information required is as a field of vectors pointing the correct travel direction at every point in space (Payton 1990), as in figure 14. The vector field is the plan; think of it as follows:

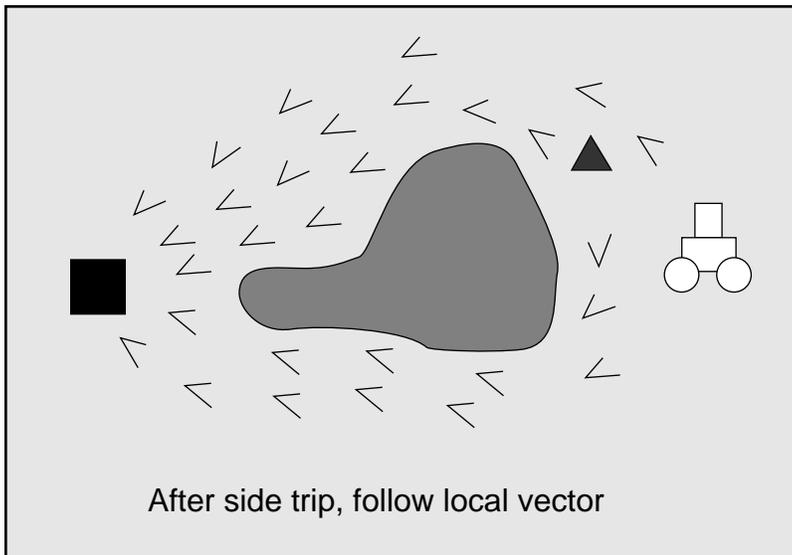


Figure 14. Vector Field Shows Travel Directions.

```
(WHENEVER ROBOT-MOVED*
  (LET ((X Y) (CURRENT-LOCATION)))
    (MOVE Direction = (VECTOR-
      FIELD X Y)))) .
```

If this plan is interrupted, it goes in whatever direction VECTOR-FIELD specifies when it resumes.

This idea is attractive at first glance, but it is not without problems. It requires computing the appropriate travel direction at all points in advance, whether they will be used or not, which seems excessive. (This feature is reminiscent of universal plans [Schoppers 1987].) Such precomputation is meaningless if there are inaccuracies in the robot's map. The stored values are not retrievable unless the robot is able to accurately compute its X,Y location at run time. (In some contexts, it is reasonable to assume that it can do so.)

Fortunately, we can fix the idea simply by allowing the robot to compute the path dynamically. Miller and Slack (1991) designed and implemented such a scheme. The robot's destination generates a vector field that is then modified by vector fields associated with obstacles (figure 15). Obstacles are recognized by transforming range data into elevation data (compare Thorpe et al. [1988]). A global map is gradually constructed by locating the obstacles in a global coordinate system. The robot must know its location fairly well with respect to this system to maneuver around obstacles it can't see and (in most cases) to know the direction of its destination. Locally visible obstacles can be dealt with directly. At any instant, a set of obstacles is selected as

relevant to path planning. A decision is made to go around each obstacle either clockwise or counterclockwise. This decision sets up a local vector field around each relevant obstacle. The fields are then combined to generate a motion direction for the robot at the current point. The computation is fast enough that it can be repeated before every robot movement.

This approach is known in the robot trade as the *artificial potential field approach* to motion planning because you can think of the vectors as sums of repulsions from obstacles and attractions toward goals. However, the phrase potential field is subject to several interpretations. Some researchers use it as a synonym for vector field even if the field does not have an associated potential (Miller and Slack's does not). Some use it as a device for generating paths, which are then tracked in the traditional way (Khatib 1986). Some use it literally as a potential field, treating the resulting artificial force as a virtual control input for the robot (Koditschek 1987). For any interpretation, the approach has difficulties (Koren and Borenstein 1991). The main problem is that because of local minima in the potential field corresponding to the vectors, it doesn't really address the path-planning problem at all without substantial modification. A cul-de-sac near the destination looks like a reasonably attractive place to be for just about any potential field approach.<sup>5</sup>

Clearly, what's needed is a model of how a robot (1) builds a world map as it moves through the world, (2) uses this map for long-range path planning (in spite of its incompleteness), (3) makes use of local cues (as in figure 15) to adjust as it goes, and (4) switches to an entirely different path whenever unknown areas turn out to be full of obstacles. The first job is beyond the scope of this article (see Kuipers and Byun [1988]; Thorpe et al. [1988]; Mataric [1990]; and McDermott [1992]). The algorithm for building the world map is presumably not part of the plan but is an autonomous black box as far as the planner is concerned. (The act of exploring to feed this black box with information might be planned, however.) We can picture the planner generating an alternative path whenever the world map changes. If the new path is substantially different from the old, it can become the new plan, as discussed in Transformational Planning.

## Learning

To this point, we have focused on the case where the planner has time to reason about alternative futures before guiding itself toward one of them. Such reasoning is likely

to be error prone, but the errors are often not fatal. Hence, the robot has an opportunity to learn from its mistakes and try something different the next time it's in a similar situation. All the usual learning techniques are adaptable to the robot setting. Mitchell (1990) discusses explanation-based generalization for robots; Hammond (1988) discusses case-based reasoning. However, for practical purposes, the most plausible application of learning is to learning the statistics of the domain: what to do under what circumstances or what is likely to happen under what circumstances, which is closely related. Furthermore, it is implausible to assume that the learner has a much richer perceptual model than the behavior. Therefore, what gets learned is likely to be a mapping from low-level perceptions to recommended low-level actions or to predictions of future low-level perceptions.

If it were not for such constraints on modeling, we could draw a parallel between transformational planning and learning. Transformational planning occurs at plan time when the planner anticipates and fixes a bug arising from a novel combination of plans. Learning occurs after a run when a bug in a stored plan has been experienced, and the fix gets written back into the plan. The only problem with this idea is that it depends on being able to generate good patches for bugs when the agent lacks either a good theory of the world or a good perceptual model of the world. If a wall-following plan goes astray, it is unlikely that the agent will be able to generate a good enough explanation to propose a fix to the plan. (The human plan writer creates such explanations all the time, of course.)

Hence, we fall back on a strategy of compiling statistics regarding the conditions under which things work or don't work without trying to explain these statistics. We can assume our plans look like the following:

```
(LOOP ; (Asynchronous version)
  (TRY-ONE (cond1 action1)
    (cond2 action2)
    ...
    (condn actionn))) .
```

The learning problem is to adjust each  $cond_i$  so that it is the best-attainable filter for its  $action_i$ . The condition can include coin flips as well as tests of sensory input. The learner does not attempt to construct new actions. In our current context, it would be useful to have a theory of constructing or adapting new plans for the plan library, but there hasn't been much work in this area (however, see Hammond [1988]).

The learner gets feedback about how appro-

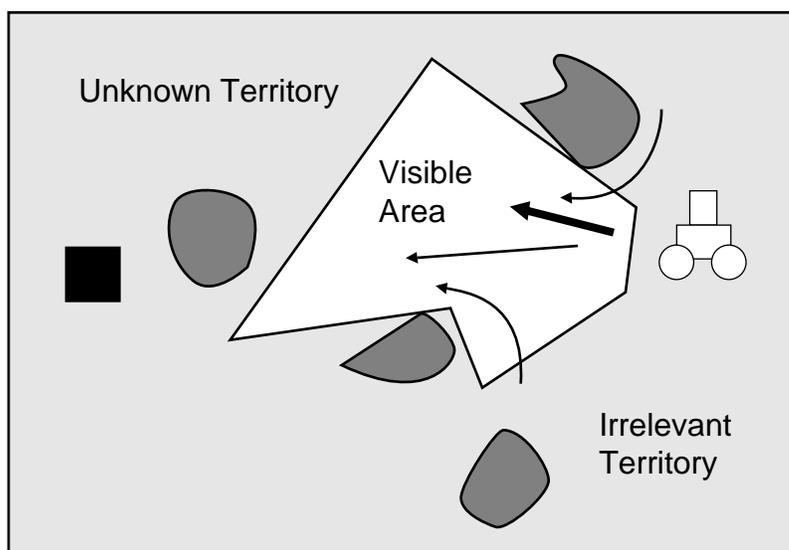


Figure 15. Local Vector Fields (Miller and Slack 1991).

prate its past actions have been. We can make various alternative assumptions about this feedback signal:

First, it can be deterministic or stochastic. In our context, we always assume the latter. Reinforcements are correlated with actions, but even the correct action can get a bad feedback signal some of the time.

Second, it can be stationary or nonstationary, depending on whether the world's behavior changes over time.

Third, it can be binary or graded. In the former case, all we know is that previous actions were good or bad. In the latter case, we have some notion of how far they were from right. (The extra information helps only if an action is parameterized in such a way that we know how to change it a little bit.)

Fourth, it can be immediate or delayed.

Fifth, it can be memoryless or state dependent.

The last two items are closely related. Figure 16 clarifies the distinction. Suppose the agent has to make a decision at time  $t_d$  based on input  $I_d$ , and it opts for action  $A_d$ . If it gets a feedback signal at time  $t_d+1$  (or  $t_d + \epsilon$  in an asynchronous model), and the signal depends only on the pair  $\langle I_d, A_d \rangle$ , it is *immediate*, memoryless, feedback. Now suppose  $A_d$  has no immediate effects but sends the robot into a zone whose future feedback history is inferior or superior to the average. We have *delayed feedback*. Now suppose that the feedback signal depends not just on  $\langle I_d, A_d \rangle$  but on the past sequence of input ...  $I_{d-2}, I_{d-1}, I_d$ . Now we

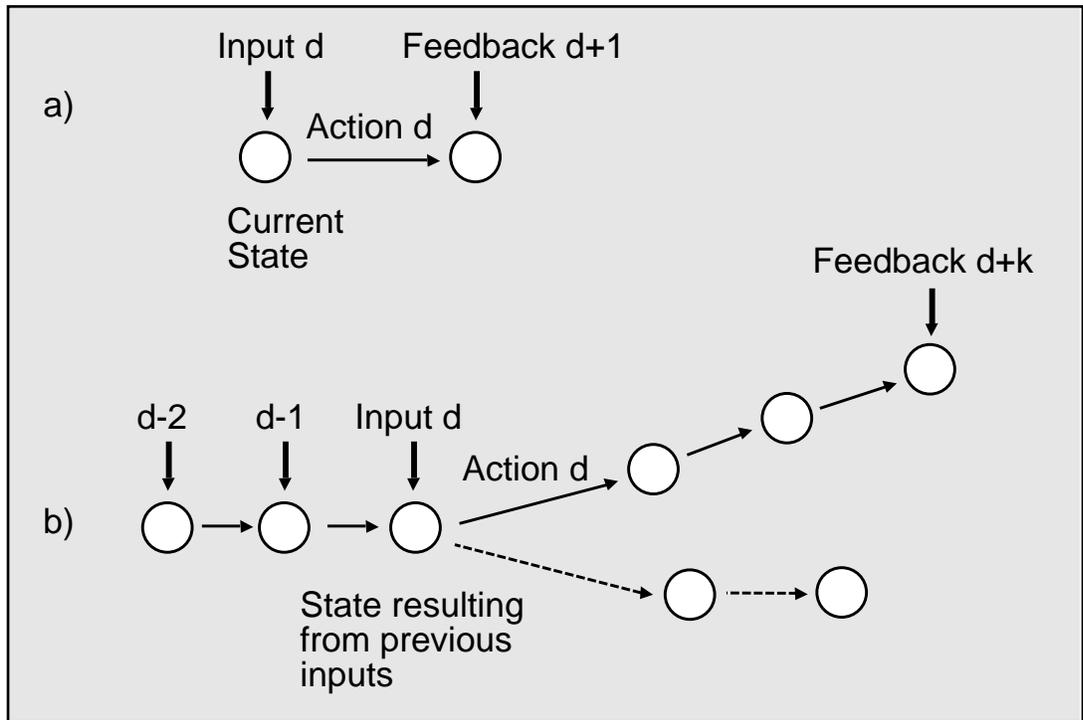


Figure 16. Feedback Regimes.

(a) Memoryless, immediate. (b) Delayed, state-dependent feedback.

have *state-dependent feedback*, so called because the feedback depends on the state the robot has been driven into by prior input.

For the moment, let's focus on the case of stochastic, stationary, binary, immediate, and memoryless learning. In this paradigm, what the planner is trying to learn can be thought of as a table giving the expected reward of each combination of actions for each possible condition. Once it learns the table, it should always pick the action combination with the highest possible value (because "being stationary" means that the rules never change).

Unfortunately, this scenario assumes that there are only a few distinct input and that what to do when one input is seen is independent of what to do when any other input is seen. However, suppose the input is a pattern of  $N$  bits. There are then  $2^N$  possible input, and those that share bits are likely to indicate the same action. If there are  $k$  actions, then there are  $k^{2^N}$  different hypotheses about how input map to actions. It's unrealistic to search all of them, so we need to *bias* the learning algorithm by focusing on a particular subset, or *hypothesis space*, at the outset (Mitchell 1980; Haussler 1988). At this point, all computational learning theory opens up before us.

I confine myself to examining one algo-

rithm, owed to Kaelbling (1990), called GTRL. (It's based on Schlimmer's [1987] STAGGER algorithm.) For the purpose of exposition, I assume that there are just two actions, and the problem is to find a Boolean combination of the input bits that tells us when to pick action 1 as opposed to action 0. The algorithm keeps a set of candidate hypotheses around, each a Boolean combination of input, and it tracks how well each candidate has done. That is, every time it takes an action and gets back a yes or a no, the algorithm updates the success and failure counts of every candidate hypothesis based on whether the hypothesis would have recommended the same action. The hypothesis that gets to control the action recommendation on the next cycle is the one with the highest value of

$$K \times er(h) + er-ub(h) ,$$

where (roughly)  $er(h)$  is the expected reinforcement for a hypothesis  $h$  based on comparison of past behavior with what  $h$  recommended, and  $er-ub(h)$  is an upper bound (arrived at using statistical methods)  $e$  such that the chance of the true value of  $er(h)$  being higher than  $e$  is less than 5 percent.  $K$  is chosen to be  $\gg 1$ . As more data are gathered,  $er-ub$  and  $er$  converge, and the  $K \times er(h)$  term dominates. Initially, however, the  $er-ub$  figure

is significantly higher and drives the agent toward attempting to gather more data about the hypothesis in question.

As statistics are gathered, some hypotheses do poorly and are pruned. To take their place, the algorithm generates new hypotheses by creating disjunctions and conjunctions of old ones to some size limit. When creating disjunctions, GTRL tries to make use of hypotheses that have high *sufficiency*, meaning that they tend to do well when they recommend action 1 (that is, evaluate to true). When creating conjunctions, it uses *necessity*, a measure of how well a hypothesis does when it recommends action 0 (that is, evaluates to false).

The GTRL algorithm works fairly well, especially in nonstationary environments, where its tendency to keep trying new things helps it track the currently correct hypothesis. Consult Kaelbling (1990) for details of its performance on test cases of both artificial tasks and tasks involving a simulated robot learning to find a beacon.

A similar approach was used by Maes and Brooks (1990) to get a six-legged robot to learn to walk. The robot could take actions such as “Swing leg K forward” or “Swing leg K backward,” and the reinforcement signal was whether the belly stayed off the floor and whether forward movement of the whole robot occurred. Input sensors told the robot whether each leg was in contact with the ground. For every action, the learning algorithm tracked its current best hypothesis regarding which conjunction of conditions should trigger the action. New sensor input are added to a conjunction if they seem to be correlated with positive reinforcement. This algorithm was able to learn within a few minutes to keep the robot moving forward.

To this point, I have assumed that feedback is immediate and without memory. If you want to relax the immediacy assumption, then you get the classical *temporal credit-assignment problem*, where actions taken at time  $d$  have consequences at some later time when the learner won't know which past actions were responsible. Lately, most of the attention in this area has gone to approaches related to Sutton's (1988) temporal difference methods for learning. The temporal difference approach is to separate the job of reinforcing the current action and the job of predicting what the eventual reinforcement will actually be. Each such prediction is a function of the current input (we're still assuming state is irrelevant). On each iteration, we use the current best guess regarding eventual reinforcement as the reinforcement signal for the action learner. These guesses

will eventually be reasonable if the reinforcement-prediction learner does its job.

Now the only problem is to learn good estimates of future reinforcement. Let  $d$  be the time of decision. Define the value of action  $a$  in response to input  $i_d$  to be

$V(i_d, a) = R(d + 1) + \gamma(\text{Expected } V(i_{d+1}, a_{d+1}))$ , where  $R(t)$  = reinforcement at time  $t$ ; Expected  $V(i_{d+1}, a_{d+1})$  = expected value given likely  $i_{d+1}$  and optimal behavior thereafter; and  $\gamma$  = discount rate for future reinforcement, a number between 0 and 1. A reinforcement  $r$  attained  $\Delta t$  time units in the future is counted as currently worth only  $\gamma^{\Delta t+1}r$  to the agent. Thus, the total lifetime reinforcement as measured at time  $d$  is

$$\sum_{t=d+1}^{\infty} \gamma^{t-d-1} R(t) .$$

Without this kind of discounting, many decisions would have infinite reward (or negatively infinite), and there would be no basis for choice among them.

Assuming the robot experiences all input repeatedly (and keeping in mind that there is no state beyond the current input), we can learn  $V$  by keeping a table  $\hat{V}(i, a)$  of estimates of  $V$ . After taking action  $a_d$ , getting reinforcement  $R(d + 1)$ , and choosing the next action  $a_{d+1}$ , add the following to  $\hat{V}(i_d, a_d)$ :

$$\Delta \hat{V}(i_d, a_d) = \alpha(R(d + 1) + \gamma \hat{V}(i_{d+1}, a_{d+1}) - \hat{V}(i_d, a_d)) .$$

This equation might look imposing but note that the quantity after the  $\alpha(\cdot)$  factor

$$R(d + 1) + \gamma \hat{V}(i_{d+1}, a_{d+1}) - \hat{V}(i_d, a_d)$$

is 0 when the table of estimates is correct. Otherwise, it's a measure of the discrepancy between  $\hat{V}(i_d, a_d)$  and

$$R(d + 1) + \gamma(\text{Expected } V(i_{d+1}, a_{d+1})) ,$$

using  $\hat{V}(i_{d+1}, a_{d+1})$  to estimate this expected value. The idea is to push  $\hat{V}$  in the right direction at a speed governed by  $\alpha$ .

Temporal difference methods are appealing for their simplicity, theoretical properties, and practicality (Sutton 1988; Kaelbling 1990). However, they do not as yet solve some of the hard problems. One of the hardest problems is state-dependent feedback. One way to model it is to broaden the set of input to include input sequences to some length. However, this option is obviously combinatorially explosive. As usual, it is not hard to think of situations that baffle any learning algorithm. Learning makes the most sense when it is thought of as filling in the details in an algorithm that is already nearly right. An example is map learning, where the agent's techniques are specialized to the case of figuring out the shape of its environment. This problem is vir-

*Temporal difference methods are appealing for their simplicity, theoretical properties, and practicality.*

tually impossible to solve when cast in terms of maximizing reinforcement unless the world is shaped simply, and the same sorts of reward are found in the same places all the time. (Feedback is strongly state dependent.) A much better approach is to wire in the presupposition that the world has a stable topology and geometry.

## Conclusions

The main conclusion I want to draw is that there is such a thing as robot planning, an enterprise at the intersection of planning and robotics. It consists of attempts to find fast algorithms for generating, debugging, and optimizing robot programs, or plans, by reasoning about the consequences of alternative plans. Its methods are influenced by two elements.

First are limitations on the quantity and the quality of sensory information. Most actions are taken on the basis of continually updated vectors of bits. Complex symbolic descriptions of the robot's surroundings are not available. The robot's model of world physics is often weak and needs to be supplemented by learning.

Second is the need to make decisions fast. Because of the need for speed, there is unlikely to be a general algorithm for robot planning. Instead, we assemble a suite of techniques for different parts of the job. Such an assembly is already forming. In my opinion, we will do a better job of understanding this assembly if we agree on terms; concepts; and, to the extent possible, notations.

There are a lot of topics I could not cover in this survey, including map learning, assembly planning, decision and estimation theory, adaptive control, and computational learning theory.<sup>6</sup>

One topic that was conspicuously absent is logical formalism. The reason for this omission is that practice has outstripped logic. Formal theories are still focusing on classical planning and the projection of simple action sequences. What we really need is a formal framework that embraces programming language semantics, including concurrency; reasoning about abstract intentions before knowing how or if they will be fulfilled; temporal reasoning, including probability; and control theory. This framework might sound like a tall order. Actually, these four elements have been well studied, and to some extent, all we need is for someone to put them together. We don't need the theory to be computationally tractable; we don't expect a robot to prove theorems in real time about what it is about to do. Instead,

we need the theory as a tool for robot designers to use in proving that their plans and planning algorithms can bring about results in the world. Some encouraging preliminary results can be found in Pelavin (1988) and Lyons and Arbib (1989).

Finally, a comment on where I expect the field to go. I expect to see the development of formal methods, as I just discussed. I also expect to see a strengthening of the tugs on robot planning from the two disciplines it relates to. In the short run, the pull toward robotics is definitely stronger, but eventually planning will pull back. I anticipate this development by making some architectural recommendations for future robot planners:

**Adopt explicit plans:** It will pay to have a simple, uniform plan notation at all levels of the robot program. Software is better than hardware. It's more flexible and encourages run-time plan manipulation, which is easier than conventional wisdom might suggest.

**Always behave:** The plan runs even while the planner thinks. If parts of the plan fail, the rest can usually continue while the planner fixes it.

**Treat planning as anytime plan transformation:** Make use of fast, interruptible algorithms (Zweben, Yang, Boddy, et al. have given us some examples). When the planner finds a better plan, swap it in.

**Use learning—judiciously:** Don't be afraid to let the robot learn the behavior of the world if it can be characterized statistically.

## Acknowledgments

This article is based on an invited talk given at the July 1991 National Conference on Artificial Intelligence. Some of the work described here was supported by DARPA/BRL grant DAA15-87-K-0001. A referee supplied some helpful comments.

## Notes

1. There is actually more to REX than I imply here. See Kaelbling and Wilson (1988) for the whole story.
2. It would be better if we changed terminology so that the technique just described were known as "search through partially ordered plans." However, getting people to change terminology can be a challenge.
3. We could call it "search through interleaved plans."
4. We could call it "search through concurrent plans." If these terminological suggestions were adopted, let me deprecate the tendency for the word plan to transmute to "planning," the process that has led to the use of the phrase reactive planning to mean "search through reactive plans." The last thing we need is for, for example, "search

through concurrent plans" to become known as concurrent planning.

5. Koditschek (1987) proved that there exists a field with no such local minima, but there is no reason for this field to be easily computable or to bear much resemblance to the usual locally generated field.

6. I have a feeling that my choice of topics might mislead the less informed reader. Let me make it clear that in a practical system, the code required to do domain-specific planning (for example, to assemble a set of parts) is likely to dwarf the code required to do the kinds of projection and transformation that I discussed.

## References

- Agre, P. E., and Chapman, D. 1990. What Are Plans For? In *New Architectures for Autonomous Agents: Task-Level Decomposition and Emergent Functionality*, ed. P. Maes. Cambridge, Mass.: MIT Press.
- Agre, P. E., and Chapman, D. 1987. PENG: An Implementation of a Theory of Activity. In Proceedings of the Sixth National Conference on Artificial Intelligence, 268–272. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Allen, J.; Hendler, J.; and Tate, A. 1990. *Readings in Planning*. San Mateo, Calif.: Morgan Kaufmann.
- Boddy, M., and Dean, T. 1989. Solving Time-Dependent Planning Problems. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 979–984. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.
- Brooks, R. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation* RA-2(1): 14–23.
- Brooks, R. 1982. Solving the Find-Path Problem by Good Representation of Space. In Proceedings of the Second National Conference on Artificial Intelligence, 381–386. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Canny, J. 1988. *The Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Series. Cambridge, Mass.: MIT Press.
- Chapman, D. 1990. Vision, Instruction, and Action, Technical Report, 1204, AI Laboratory, Massachusetts Inst. of Technology.
- Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32(3): 333–377.
- Charniak, E., and McDermott, D. 1985. *Introduction to Artificial Intelligence*. Reading, Mass.: Addison-Wesley.
- Collins, G. 1987. Plan Creation: Using Strategies as Blueprints. Ph.D. diss., Dept. of Computer Science, Yale Univ.
- Connell, J. 1990. *Minimalist Mobile Robots*. Boston: Academic.
- Dean, T., and Boddy, M. 1988. An Analysis of Time-Dependent Planning. In Proceedings of the Seventh National Conference on Artificial Intelligence, 49–54. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Drummond, M., and Bresina, J. 1990. Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction. In Proceedings of the Eighth National Conference on Artificial Intelligence, 138–144. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Firby, R. J. 1989. Adaptive Execution in Complex Dynamic Worlds, Technical Report, 672, Dept. of Computer Science, Yale Univ.
- Firby, R. J. 1987. An Investigation into Reactive Planning in Complex Domains. In Proceedings of the Sixth National Conference on Artificial Intelligence, 202–206. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Georgeff, M., and Lansky, A. 1987. Reactive Reasoning and Planning. In Proceedings of the Seventh National Conference on Artificial Intelligence, 677–682. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Georgeff, M., and Lansky, A. 1986. Procedural Knowledge. In Proceedings of the IEEE (Special Issue on Knowledge Representation), 1383–1398. Washington, D.C.: IEEE Computer Society.
- Gupta, N., and Nau, D. 1991. Complexity Results for Blocks-World Planning. In Proceedings of the Ninth National Conference on Artificial Intelligence, 629–633. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Hammond, K. 1990. Explaining and Repairing Plans That Fail. *Artificial Intelligence* 45(1–2): 173–228.
- Hammond, K. 1988. *Case-Based Planning: An Integrated Theory of Planning, Learning, and Memory*. New York: Academic.
- Hanks, S. 1990a. Practical Temporal Projection. In Proceedings of the Eighth National Conference on Artificial Intelligence, 158–163. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Hanks, S. 1990b. Projecting Plans for Uncertain Worlds, Technical Report, YALEU/DCS/RR-756, Dept. of Computer Science, Yale Univ.
- Hausler, D. 1988. Quantifying Inductive Bias: AI Learning Algorithms and VALIANT's Learning Framework. *Artificial Intelligence* 36(2): 177–222.
- Hendler, J.; Tate, A.; and Drummond, M. 1990. AI Planning: Systems and Techniques. *AI Magazine* 11(2): 61–77.
- Horvitz, G. F.; Cooper, G. F.; and Heckerman, D. E. 1989. Reflection and Action under Scarce Resources: Theoretical Principles and Empirical Study. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1121–1127. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.
- Kaelbling, L. P. 1990. Learning in Embedded Systems. Ph.D. diss., Teleos Research Report, 90-04, Stanford Univ.
- Kaelbling, L. P. 1988. Goals as Parallel Program Specifications. In Proceedings of the Seventh National Conference on Artificial Intelligence, 60–65. Menlo Park, Calif.: American Association for

Artificial Intelligence.

Kaelbling, L. P. 1987. An Architecture for Intelligent Reactive Systems. In *Reasoning about Plans and Actions*, eds. M. Georgeff and A. Lansky, 395–410. San Mateo, Calif.: Morgan Kaufmann.

Kaelbling, L. P., and Rosenschein, S. J. 1990. Action and Planning in Embedded Agents. In *New Architectures for Autonomous Agents: Task-Level Decomposition and Emergent Functionality*, ed. P. Maes, 35–48. Cambridge, Mass.: MIT Press.

Kaelbling, L., and Wilson, N. J. 1988. REX Programmer's Manual, Technical Note 381R, SRI International, Menlo Park, California.

Khatib, O. 1986. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *International Journal of Robotics Research* 5(1): 90–98.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science* 220(4598): 671–680.

Koditschek, D. 1987. Exact Robot Navigation by Means of Potential Functions: Some Topological Considerations. In Proceedings of the IEEE International Conference on Robotics and Automation, 1–6. Washington, D.C.: IEEE Computer Society.

Koren, Y., and Borenstein, J. 1991. Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation. In Proceedings of the IEEE International Conference on Robotics and Automation, 1398–1404. Washington, D.C.: IEEE Computer Society.

Kuipers, B., and Byun, Y. 1988. A Robust, Qualitative Method for Robot Spatial Reasoning. In Proceedings of the Seventh National Conference on Artificial Intelligence, 774–779. Menlo Park, Calif.: American Association for Artificial Intelligence.

Latombe, J. 1991. *Robot Motion Planning*. Boston: Kluwer Academic.

Lin, S., and Kernighan, B. W. 1973. An Effective Heuristic Algorithm for the Traveling Salesman Problem. *Operations Research* 21: 498–516.

Lozano-Perez, T. 1983. Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers* C-32(2): 108–120.

Lyons, D. M., and Arbib, M. A. 1989. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Transactions on Robotics and Automation* 5(3): 280–293.

Lyons, D. M.; Hendriks, A. J.; and Mehta, S. 1991. Achieving Robustness by Casting Planning as Adaptation of a Reactive System. In Proceedings of the IEEE Conference on Robotics and Automation, 198–203. Washington, D.C.: IEEE Computer Society.

McAllester, D., and Rosenblitt, D. 1991. Systematic Nonlinear Planning. In Proceedings of the Ninth National Conference on Artificial Intelligence, 634–639. Menlo Park, Calif.: American Association for Artificial Intelligence.

McDermott, D. 1992. Spatial Reasoning. In *Encyclopedia of Artificial Intelligence*, 2d ed., ed. S. Shapiro, 1322–1334. New York: Wiley.

McDermott, D. 1991. A Reactive Plan Language,

Technical Report, YALEU/DCS/RR-864, Dept. of Computer Science, Yale Univ.

McDermott, D. 1990. Planning Reactive Behavior: A Progress Report. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 450–458. San Mateo, Calif.: Morgan Kaufmann.

McDermott, D. 1987. A Critique of Pure Reason. *Computational Intelligence* 3(3): 151–160.

Maes, P., and Brooks, R. A. 1990. Learning to Coordinate Behaviors. In Proceedings of the Eighth National Conference on Artificial Intelligence, 796–802. Menlo Park, Calif.: American Association for Artificial Intelligence.

Mataric, M. J. 1990. A Distributed Model for Mobile Robot Environment-Learning and Navigation, Technical Report, 1228, AI Laboratory, Massachusetts Inst. of Technology.

Miller, D., and Slack, M. G. 1991. Global Symbolic Maps from Local Navigation. In Proceedings of the Ninth National Conference on Artificial Intelligence, 750–755. Menlo Park, Calif.: American Association for Artificial Intelligence.

Mitchell, T. 1990. Becoming Increasingly Reactive. In Proceedings of the Eighth National Conference on Artificial Intelligence, 1051–1058. Menlo Park, Calif.: American Association for Artificial Intelligence.

Nau, D. S. 1982. An Investigation of the Causes of Pathology in Games. *Artificial Intelligence* 19: 257–258.

Nau, D. S.; Yang, Q.; and Hendler, J. 1990. Optimization of Multiple-Goal Plans with Limited Interactions. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 160–165. San Mateo, Calif.: Morgan Kaufmann.

Nilsson, N. J. 1988. Action Networks. In Proceedings of the Rochester Planning Workshop: From Formal Systems to Practical Systems, eds. J. Tenenber, J. Weber, and J. Allen, 20–51. Rochester, N.Y.: University of Rochester Department of Computer Science.

Norman, D. A., and Bobrow, D. G. 1975. On Data-Limited and Resource-Limited Processes. *Cognitive Psychology* 7(1): 44–64.

Ow, P. S.; Smith, S.; and Thiriez, A. 1988. Reactive Plan Revision. In Proceedings of the Seventh National Conference on Artificial Intelligence, 77–82. Menlo Park, Calif.: American Association for Artificial Intelligence.

Payton, D. 1990. Exploiting Plans as Resources for Action. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 175–180. San Mateo, Calif.: Morgan Kaufmann.

Pelavin, R. 1988. A Formal Approach to Planning with Concurrent Actions and External Events. Ph.D. diss., Dept. of Computer Science, Univ. of Rochester.

Rich, E. 1983. *Artificial Intelligence*. New York: McGraw-Hill.

Rosenschein, S. J. 1989. Synthesizing Information-

Tracking Automata from Environment Descriptions. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning, Toronto*, 386–393. San Mateo, Calif.: Morgan Kaufmann.

Russell, S., and Wefald, E. 1991. Principles of Metareasoning. *Artificial Intelligence* 49 (1–3): 361–395.

Schlimmer, J. C. 1987. Learning and Representation Change. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 511–515. Menlo Park, Calif.: American Association for Artificial Intelligence.

Schoppers, M. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1039–1046. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Simmons, R. G. 1991. Concurrent Planning and Execution for a Walking Robot. In *Proceedings of the IEEE Conference on Robotics and Automation*, 300–305. Washington, D.C.: IEEE Computer Society.

Simmons, R. G. 1990. An Architecture for Coordinating Planning, Sensing, and Action. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 292–297. San Mateo, Calif.: Morgan Kaufmann.

Simmons, R. G. 1988a. A Theory of Debugging Plans and Interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 94–99. Menlo Park, Calif.: American Association for Artificial Intelligence.

Simmons, R. G. 1988b. Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems, Technical Report, TR 1048, AI Laboratory, Massachusetts Inst. of Technology.

Sussman, G. 1975. *A Computer Model of Skill Acquisi-*

*tion*. New York: American Elsevier.

Sutton, R. 1988. Learning to Predict by the Method of Temporal Differences. *Machine Learning* 3(1): 9–44.

Tate, A. 1977. Generating Project Networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 888–893. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Thorpe, C.; Hebert, M. H.; Kanade, T.; and Shafer, S. 1988. Vision and Navigation for the Carnegie-Mellon Navlab. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10(3): 362–373.

Walter, W. G. 1950. An Imitation of Life. *Scientific American* 182(5): 42.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, Calif.: Morgan Kaufmann.

Yang, Q. 1989. Improving the Efficiency of Planning, Technical Report, CS-89-34, Dept. of Mathematics, Univ. of Waterloo.

Zweben, M.; Deal, M.; and Gargan, R. 1990. Any-time Rescheduling. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 251–259. San Mateo, Calif.: Morgan Kaufmann.



Drew McDermott received all his education at the Massachusetts Institute of Technology, culminating in a Ph.D. in 1976. In 1976, he moved to Yale University, where he is currently. He recently became chairman of the Computer Science Department at Yale.

## Readings from AI Magazine

The First Five Years: 1980-1985  
Edited with a Preface by Robert Englemore

AAAI is pleased to announce publication of *Readings from AI Magazine*, the complete collection of all the articles that appeared during AI Magazine's first five years. Within this 650-page indexed volume, you will find articles on AI written by the foremost practitioners in the field—articles that earned AI Magazine the title “journal of record for the artificial intelligence community.” This collection of classics from the premier publication devoted to the entire field of artificial intelligence is available in one large, paperbound desktop reference.

### Subjects Include:

- Automatic Programming • Distributed Artificial Intelligence • Games • Learning
- Infrastructure • Natural Language Understanding • Problem Solving • Robotics • Education
- General Artificial Intelligence • Knowledge Acquisition • Legal Issues • Object Oriented Programming • Programming Language • Simulation • Technology Transfer • Discovery
- Expert Systems • Historical Perspectives • Knowledge Representation • Logic • Partial Evaluation • Computer Architectures • Reasoning with Uncertainty

\$74.95 plus \$2 postage and handling. 650 pages, illus., appendix, index. ISBN 0-929280-01-6.  
Send prepaid orders to American Association for Artificial Intelligence,  
445 Burgess Drive, Menlo Park, California 94025.