Robot Planning

Drew McDermott

YALEU/CSD/RR #861
August, 1991

Based on an invited talk at AAAI, July, 1991.

This work was supported by the Defense Advanced Research Projects Agency, contract number DAAA15-87-K-0001, administered by the Ballistic Research Laboratory.

Robot Planning *

Drew McDermott
Yale University Computer Science Department
P.O. Box 2158 Yale Station
New Haven, CT 06520
mcdermott@cs.yale.edu

Abstract

Research on planning for robots is in a state of flux, to such an extent 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 program are combined at run time. There are several strands of research in the field; six are surveyed: (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) learning 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.

1 What is Robot Planning?

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 like "All the green blocks must be off

*This report is based on an invited talk given at the AAAI meeting in July, 1991. Some of the work described here was supported by DARPA/BRL grant DAA15-87-K-0001.

the table," a classical planner was supposed to generate a plan like, "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 any more. The problem as stated 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 planning 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 pretty crude. Perhaps we should build robots that have their own "need to plan" before we think about robot planning again. I will 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 reprogram them completely. As soon as we tell one 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 has been true since Grey Walter's turtles (Walter 1950); I leave aside the question whether it is true of robots controlled by neural nets. Virtually all other robots are under the control of formal objects which we may call their "programs," even though in some cases the object is compiled down to hardware.

This may seem like a trivial observation, but I have three reasons to make it:

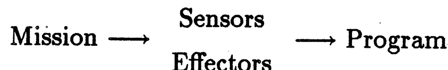
1. There is an odd current of mysticism in the AI/robotics community. Some practitioners like to pretend that their machines are "being in the world" or "living a life" rather than being controlled by something as anal retentive as a program. I wish to put distance between my views and theirs.

2. Some robot programmers call attention to the fact that their robots continually react to the current sensory input, rather than following a list of instructions without heed to the

world around them. The specifications of how to react are nonetheless still a program.

3. 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 of programming. As in the rest of computer science, robotics can't escape the advantages of expressing behaviors first as textual objects, and worrying about mapping them to hardware later.

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, development of a robot system proceeds thus:



We identify what we want the robot for (e.g., 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 may 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 (Cf. Lyons and Arbib 1989):

- 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.

- Programs are inherently concurrent. Sensors and effectors run in parallel. Many tasks demand only intermittent attention.

- 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.

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 fits into the world of programmed robots, the answer is clear: this 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 may or may 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 graphical user interface, and the planner itself. But then how 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 may be some residual philosophical issues about what counts as "explicit reasoning," but we can, I hope, take a commonsense position on that question.

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 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.

That's my 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 may be other operations that should be included.

This is tidy, but it has the unfortunate consequence that we've 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, in that 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; or to devise the most efficient plan for a certain task; or to react to a type of recurring world state; or some combination of all of 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 $\ldots A_n$) into a particular ordering of the A_i, with the object of minimizing total expected time to completion. A classical planning problem might be thought as transforming the abstract plan (**ACHIEVE** p) into a sequence of actions such that p is true when the sequence is executed starting in 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. But it is useful to keep the framework in mind in comparing the different approaches to planning that will be surveyed in what follows.

2 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.

Much 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 that in mind.

2.1 Minimalism

We 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 that moment. Elaborate mechanisms for building world models and and making inferences from them are to be avoided. This approach is associated most prominently with the research group of Brooks at MIT (Brooks 1986). It

also includes the work on situated automata by Kaelbling and Rosenschein (1990), and on 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 (from 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 is a description 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 like 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 instead must track sensory data moment by moment, small errors cannot accumulate, but get washed away by new data. "Use the world as its own best model" is a slogan of this movement.

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

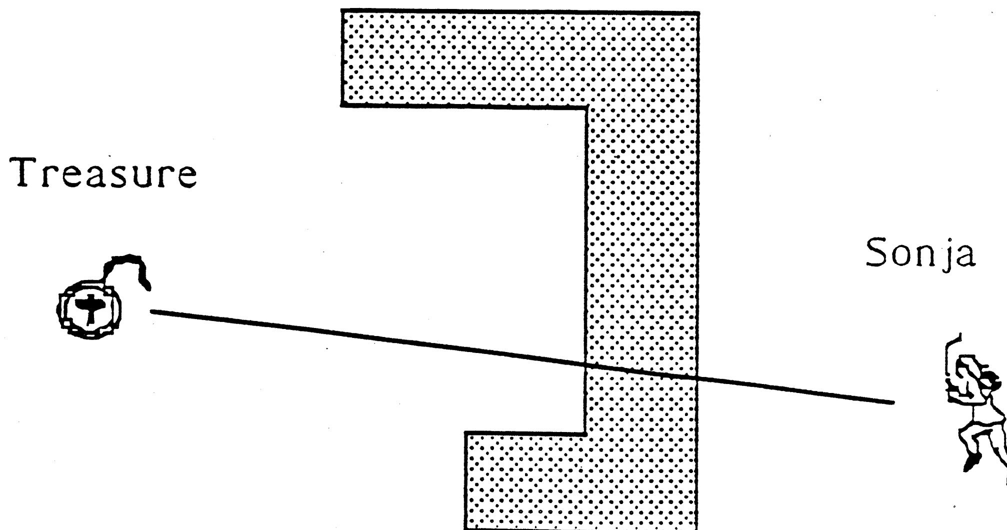


Figure 1: Navigation in Sonja (Chapman 1990)

An example of approach (1) is the module in Chapman's (1990) Sonja program that 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 to go counterclockwise around the obstacle, else clockwise.

If this 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.

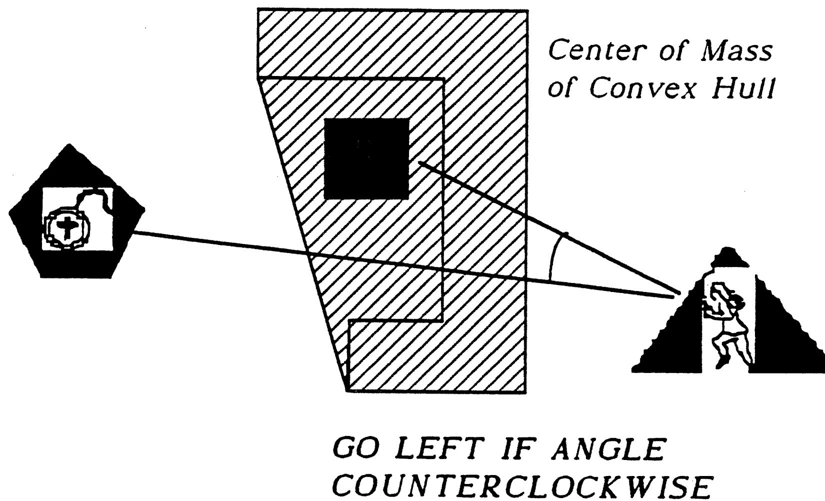


Figure 2: Sonja's decision procedure

So, 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 to discard the plan and reconstruct it? One answer is this: If computation costs nothing, then the plan should be discarded as soon as it begins to be executed! That's 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 may 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 will allow it to find a better plan. See Sections 2.3 and 2.4.)

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 in order to generate a program, which need not be executed in its entirety. It may seem odd to generate an entire program and then use only an initial segment of it, but (a) the agent is going to have to discard the plan eventually; and (b) the generation of the entire plan legitimizes the intial segment by showing a plausible way it could 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. But the program just finds the best first move, and never actually attempts to remember the whole tree of board positions it examined. It is always more cost-effective to regenerate the tree after receiving the opponent's next move than to try to keep the old version around. The same principle applies to planning in general, except when planning is too expensive or too litle 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 specification a set of states of affairs to be achieved or to be maintained, and compiles them down into programs in a minimalist formalism called Rex (Kaelbling and Wilson 1988). For our purposes[1] we can take a Rex program to be of the form

```
(WHENEVER CLOCK-TICK*
   (TRY-ONE (cond1 action1)
            (cond2 action2)
            ...
            (condn actionn)))
```

where one *action* whose *cond* is satisfied is selected each time TRY-ONE is executed. Each $cond_i$ and 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 very long). In parallel, all the conditions are checked, and one action corresponding to true conditions is executed. I will call such a plan *synchronous*, by analogy with logic circuitry. By contrast, an *asynchronous* plan is one whose actions are not yoked to a global cycle in this way, but which

[1]There is actually more to Rex than I am implying here. See Kaelbling and Wilson 1988 for the whole story.

poll the world when necessary (or get interrupted by incoming sensory information). A robot's program may 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-actions pairs. Conjunction is slightly more complicated. Wherever program 1 specifies a_1 as a response to c_1, and program specifies a_2 as a response to c_2, the conjunction will recommend the "merge" of a_1 and a_2 as the response to $c_1 \wedge c_2$, provided a_1 and a_2 are mergeable. Two actions are *mergeable* if it makes sense to send their effector messages simultaneously. (E.g., "Right-wheel motor on" and "Front sonar on" can be merged, but "Right-wheel motor on and "Right-wheel motor off" cannot be.)

This compositionality 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 spec 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 the achievement or maintenance of the *state*. (These ideas are based on the situated automata theory of Rosenschein 1989.) A collection of **DEFGOALR**s is a *plan library*, which spells out how to attack all the complex goals the agent knows about. We will see this idea again later. Kaelbling proves in (Kaelbling 1988) that the plans produced by Gapps actually do lead to the goals discussed, but this guarantee is weaker than it sounds. Given an impossible problem, Gapps may detect the impossibility, but it may also go into an infinite loop, and never find a solution, or produce a plan that loops forever, postponing achieving the impossible goal indefinitely. 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.

2.2 Plan Interpreters

There is a trade-off between the expressivity and "plannability" of a plan notation. The notations we looked at in Section 2.1 are at one end of the spectrum. They are sharply constrained

in order to be guaranteed to be easy to manipulate. In this section, I will look at plan notations that allow more complex plans to be stated more easily. I will 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 et al. 1991).

One of the most controversial features is a program counter, oddly enough. In a plan interpreter, it is the most natural thing in the world to write a plan of the form `(SEQ` $step_1$ ... $step_n$`)`, 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, the counter that keeps track of 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 might be:

```
(WITH-POLICY (WHENEVER (NOT P) step1)
     (WAIT-FOR P step2))
```

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

The plan skeleton given above 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 below.) And we have to make sure that if $step_1$ cannot actually achieve P that the robot gives up eventually. With these gaps filled, the `WITH-POLICY` version will be more robust and responsive than the version using `SEQ`.

Still, there are times when `SEQ` is awfully 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; there is instead a global rationale for doing the steps in this order rather than some other.

In the rest of this review, I will use a textual, Lisp-like notation for plans, and avoid graphical representations. The reason is simple: Graphical representations rapidly become clumsy as soon as we move beyond certain familiar types of plans. Hence I will avoid the standard "task network" or "procedural network," and emphasize that *plans are programs.* I reserve the term *task* for 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, everything that a robot programming language needs, plus features to support reasoning about the plan and the state of its execution. These features include:

1. Loops, conditionals, procedures
2. 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)
3. Bound variables
4. Explicit references to tasks (to allow explicit relations among them)
5. Constraints as explicit parts of plans (policies)
6. Annotations regarding the purposes of steps

A constraint is an active part of the plan, which acts to repair violations at run time. (E.g., "If speed exceeds 20KPH, slow down.") An annotation is a note to the planner about the purpose of a piece of the plan. (E.g., "Keep china you're carrying intact.") It may or may not be checkable at run time.

The need for bound variables requires further discussion. As I pointed out early on, 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)))
   (PLUG-INTO X))
```

We have apparently succeeded in binding the local variable `X` to an object in the world. We then pass this object to the `PLUG-INTO` plan. This may look absurd, but the classical plan `(SEQ (MOVE B C) (MOVE A B))` 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.* Such a description I will call an *effective designator.* The best a robot can do in general is to match up 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 in order 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 may call **EQUATE**, which takes two designators and declares that they refer to the same object. For example, suppose we have a plan that requires the robot to pick up an object whenever it is dropped. Enforcing this invariant depends upon the ability to find the object. The plan for restoring the invariant might look like this:

```
(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 **EQUATE**s the two designators, any belief about one is taken to be true of the other. Hence, assuming that **LOOK-FOR** returned enough information to allow **PICK-UP** to work, that information is now attached to **DESIGNATOR**, making it *effective* in the sense described above. If **LOOK-FOR** find 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 will 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: that 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 very

complicated.

In the plan shown, we made use of an explicit **EQUATE** operator. Another approach was explored by Firby in his (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 kept track of the status of such matches, and would discard information as actions were performed. A key idea was the *expectation set,* or set of objects that the robot expected to see again in some class of situations. When in such a situation, it would equate a perceived object with an expected object as soon as the perceived object matched one expected object better than all the rest.

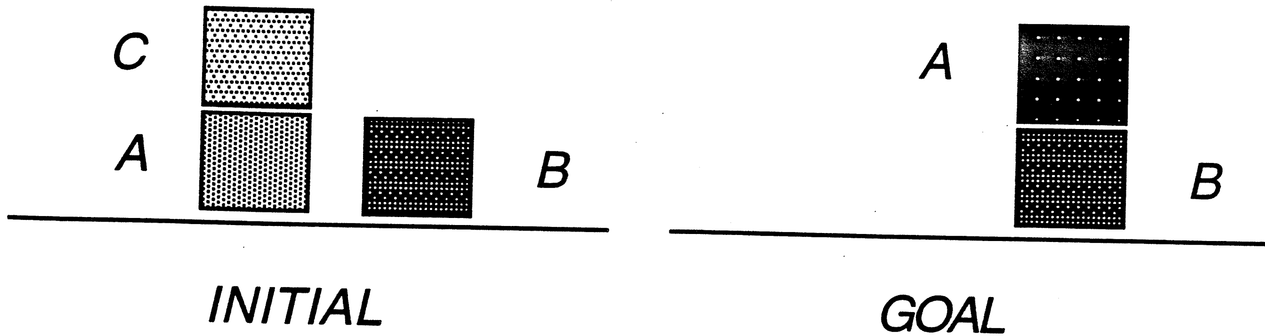
In what follows, I will introduce other notational features as we need them. If you begin to get the impression that the notation can be turned into an actual plan notation, that's because it has been. See (McDermott 1991) for a complete description of the Reactive Plan Language.

2.3 Theories of Time-constrained Planning

So far 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, e.g., a description of a stack of blocks. It then generates a sequence of actions that will bring that world state about, starting from the state obtaining now. 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 plannning* 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 graphical depiction of a typical planning problem, to get object A on B and change its color to black. Suppose further that actions require preconditions, but that otherwise their effects do not depend on what's true before they begin. In that 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



GIVEN GOALS: (ON A B)
(COLOR A BLACK)

Figure 3: Simple planning problem

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, connecting to one edge per 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

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 thus:

```
(PARTIAL-ORDER ((TAG STEP1 (->TABLE C A))
                (TAG STEP2 (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

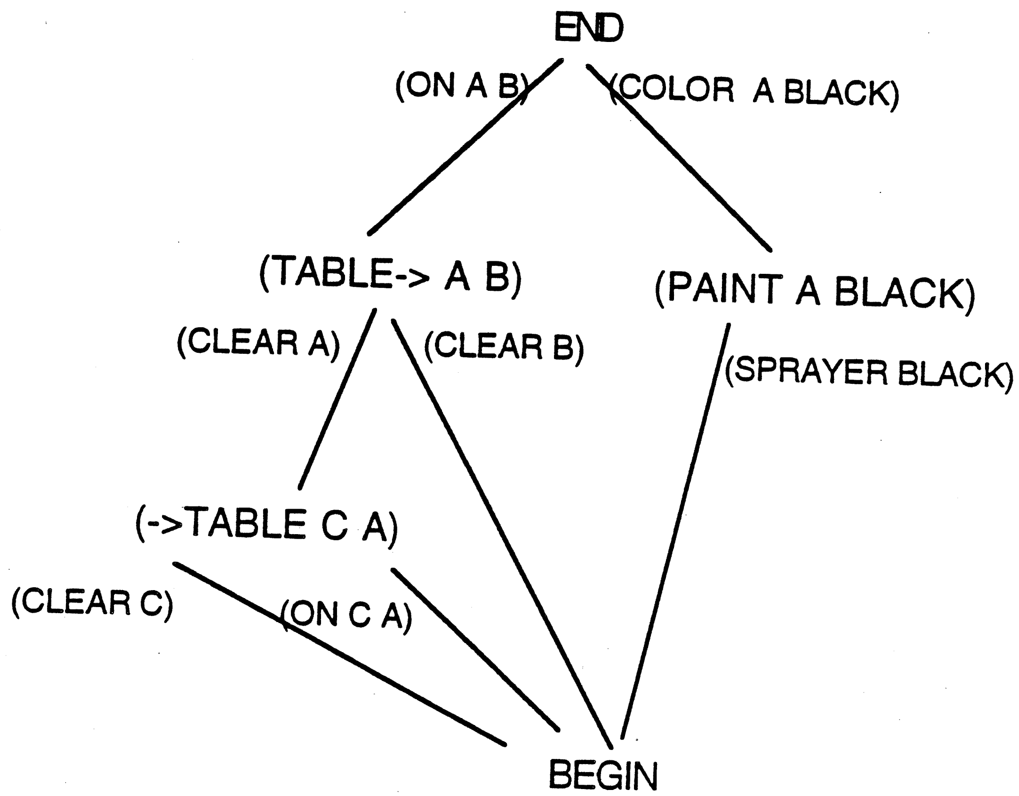


Figure 4: Plans as action graphs

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. That representation would be preferable if we did not want the plan executor to have 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 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 deleter is executed between A_1 and A_2 of the protection. The planner must insert extra edges to make sure that this 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 of 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, Chapter 9):

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. Same, except 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 achieve that goal.

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

The algorithm really becomes practical only when it is combined with techniques for postponing the choice of objects (e.g., a destination for object `C` in Figure 3) by allowing them to be represented as unknowns when first introduced into a plan, for which constraints accumulate 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

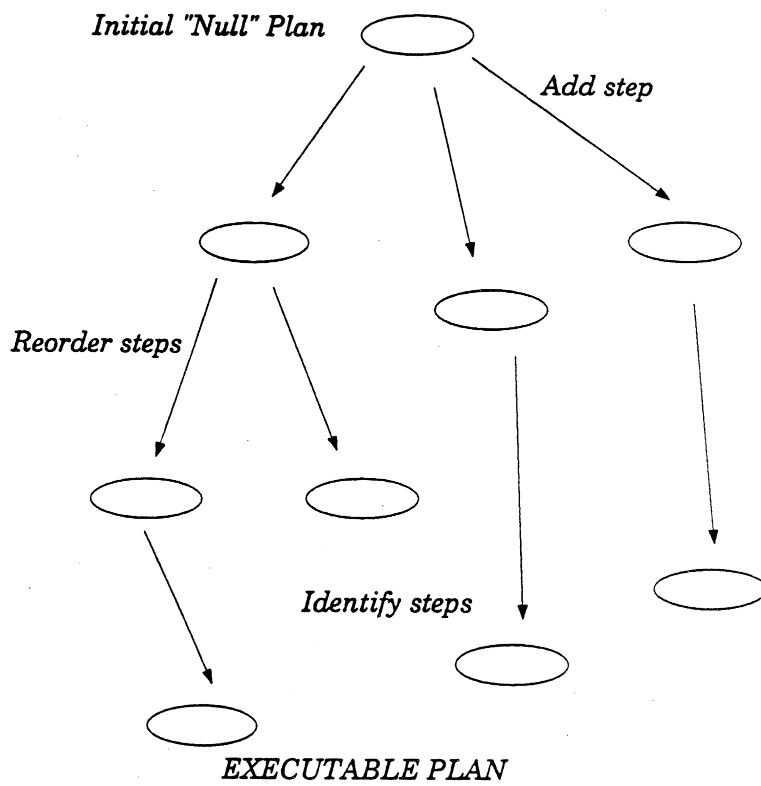


Figure 5: Planning search space

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. Then the planner can treat those effects as stable when reasoning about their impacts on other actions. This requirement may seem impossibly hard to meet, but we can often satisfy it by 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 introduces an instance of it into the plan, it must choose a y; here is where the ability to postpone such choices comes in handy.

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

The phrase "nonlinear planning" covers a lot of different techniques.[2] 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 those plans. (Kaelbling's (1988) **DEFGOALR**, discussed above, 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 may cause the robot to make a detour with a full bag of dirt. If there is a reason why that's a bad idea, then the planner could predict when the batteries would need recharging, and schedule recharging trips during periods when the bag was empty. (I will say more about such *bug projection* in Section 2.4.)

The technique of Nau, Yang, and Hendler (1990) is in the category of techniques for managing

[2]There is so much confusion about exactly what the distinction is between "linear" and "nonlinear" planning that the terms should really be replaced with a finer set of categories.

> *Idea: Combine "large" plans & run simultaneously*

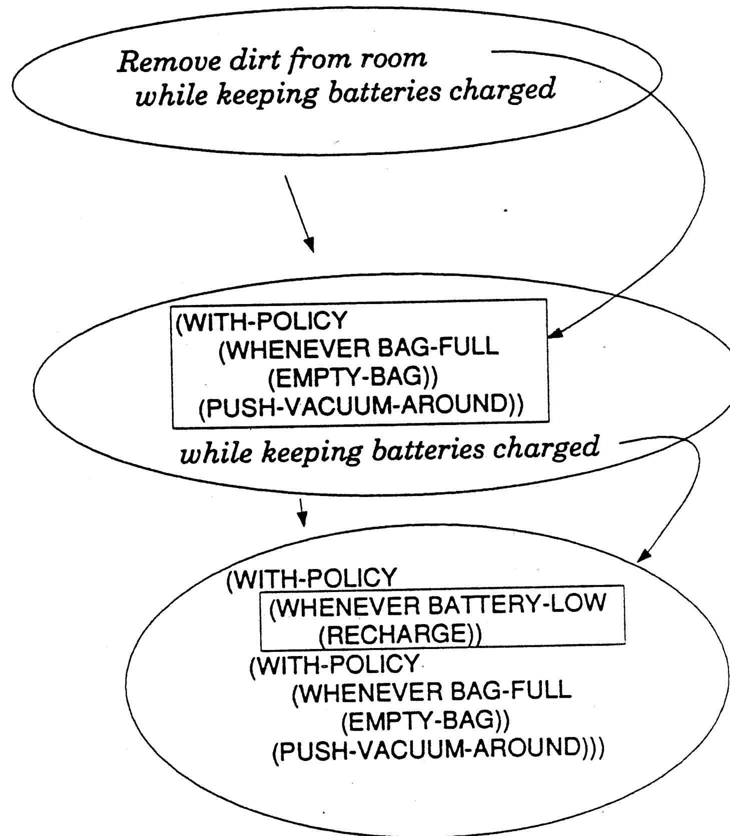


Figure 6: Plan reduction using libraries

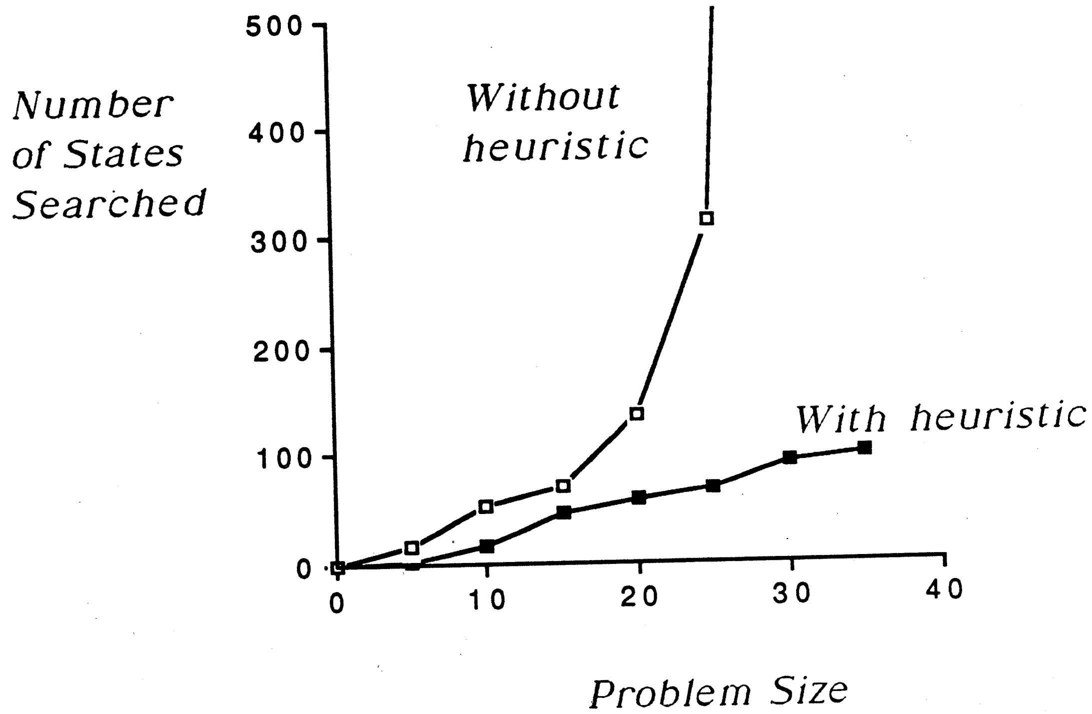
plan libraries. Suppose that the planner has derived a plan of the form

```
(SEQ (DO-ONE-OF P11 P12 ... P1n1)
     (DO-ONE-OF P21 P22 ... P2n2)
     ...
     (DO-ONE-OF Pm1 Pm2 ... Pmnm))
```

That is, it has determined that any of the P_{ij} will accomplish step i. The only remaining question is which combination of the step plans will be the least expensive. It may happen that, say, P_{12} and P_{22} are a particularly felicitious combination, because their setup costs can be shared. Suppose that the cost sharing can be modeled in terms of *step mergeabilities* among the steps for each plan P_{ij}, and suppose these are all known. That is, for each pair of action types, we store a new action type that accomplishes both of them more cheaply, if there is one. 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 merging their mergeable steps, where ordering constraints allow. The algorithm of Nau et al. finds optimal plan choices by doing a best-first search through choices for 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 et al. works quite well in practice.

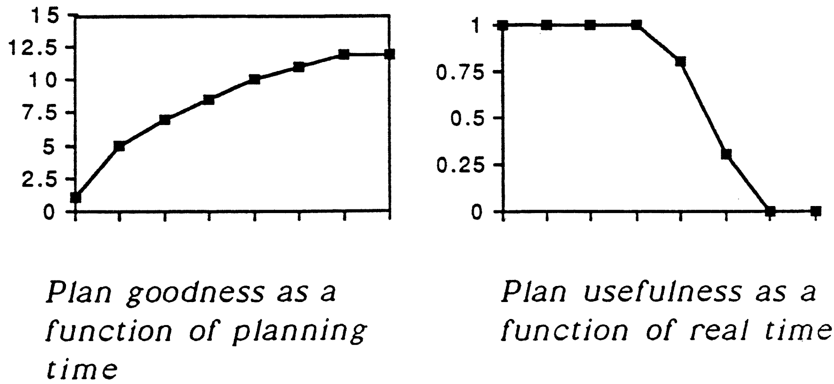
So far we have assumed that planning should be fast, but only because of the general principle that all computations should be fast. In the case of robot planning, there is a special urgency due to the fact that the robot must react to events quickly. Once the need for a plan is perceived, there is usually 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 will have to be computed while it is still within reach.

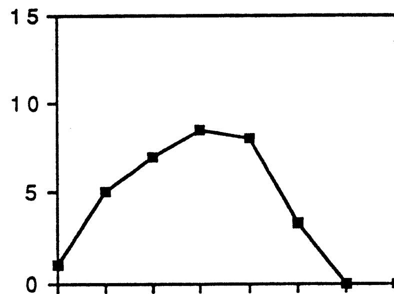
Why should more time help? It might be the case that the planning algorithm needs a certain amount of time to run. Giving it more time will give it a better chance of finding an answer before time runs out. A more pleasing possibility is that the algorithm satisfies the "Principle of Continuously Available Output" (Norman and Bobrow 1975), that it will return a plan no matter how little time it is given, but will return a better plan if given more time. Such an algorithm is called an *anytime* algorithm (Dean and Boddy 1988). (See also Horvitz et



(From Yang 1989)

Figure 7: Cost on random problems of algorithm of Nau, Yang, and Hendler





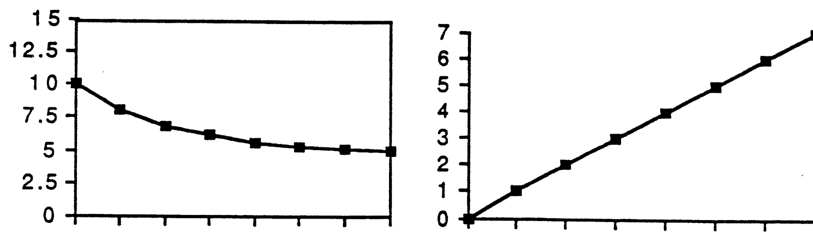
Overall value of planning time

Figure 8: The value of deliberation

al. 1989, Russell and Wefald 1991.)

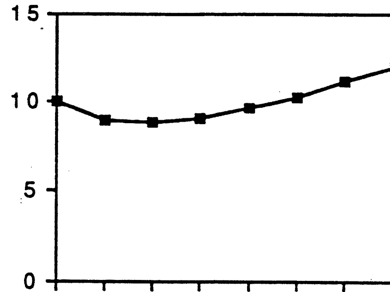
Suppose a robot has a model of how much better its plan will get per unit time. Suppose also it has a model of how fast the value of a plan will deteriorate as its time of delivery gets later. (A hard deadline is a special case where the deterioration happens all at once.) For examples, see the upper two graphs of Figure 8. If we take the product of the two graphs, we get a graph that plots represents the overall value of deliberation, shown at the bottom of Figure 8. 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 that many time units, then execute the plan it has found.

This idea is based on a few assumptions. The first 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



Path travel time as a function of planning time

Delay



Total time to completion

Figure 9: Performance of an incremental "Traveling Salesman" algorithm

considering only fast algorithms for "meta-deliberation."

The second assumption is that the only cost of deliberation is loss of effectiveness due to delay. This assumption is plausible if deliberation requires only CPU cycles. We can impose that assumption by requiring all "overt robot acts" to be planned. The best plan may be of the form:

$$\textit{Seek information} \longrightarrow \textit{plan some more}$$

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 it is possible (from Boddy and Dean 1989): Suppose the robot is expected to visit a long list of locations, in classical traveling-salesman style. The Lin-Kernighan algorithm (Lin and Kernighan 1973) works by mutating a legal but suboptimal

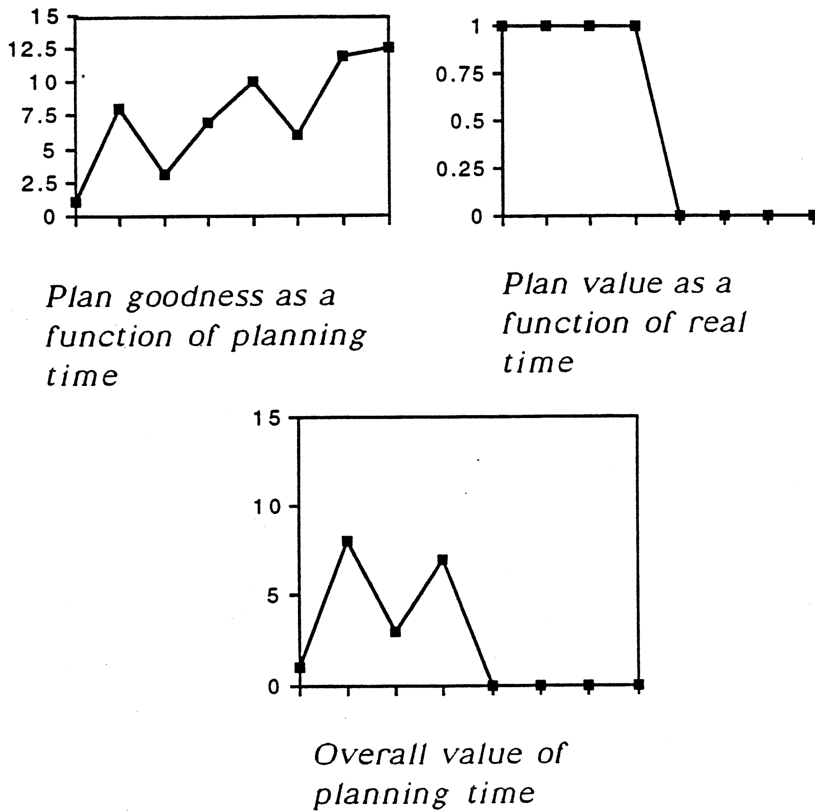


Figure 10: Typical time-constrained planning scenario

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 the upper-left graph of Figure 9. However, every unit of time spent deliberating must be added to the overall execution time, so there is a delay penalty associated with further planning, as shown in the upper-right graph of Figure 9. The sum of the two curves, shown at the bottom of the figure, 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, there are cases where further planning might 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. In the upper left 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 planner just remember the plan it had at time 1, and use it instead of the inferior one it has just generated? The answer is that the planner does not always have an accurate model of how good its plan are. The plan after time 1 might have a bug, which it repairs at time 2. (See Section 2.4.) Unbeknownst to the planner, this repair has introduced a worse bug, which it will require two more units of plan time to detect and correct. Meanwhile, as shown in the upper-right graph of Figure 10, there is a deadline looming, after which it will be 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 the lower graph of Figure 10 illustrates, this tactic does not necessarily produce the best plan. However, it should *on average* be better than not planning at all (or the planner should be decommissioned).

2.4 Transformational Planning

Many of the planning methods we have examined have in common that they operate on fully-formed plans rather than generating 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.
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 AM. 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 these 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.
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 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. Any partial plan could in principle be completed in several ways, and a plan operator simply discarded some of those 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. But this space of refinement decisions will not in general 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 plan space of an abstract plan which included the correct solution as a possible refinement (Collins 1987). The pitfall is that in revising a plan it may get worse, or cease to solve the original problem at all.

One way to cope with such pitfalls was devised by Hammond (Hammond 1988, 1990), and by Simmons (1988a,1988b). I will 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 like *Monitor a nuclear test at location B and time 12 PM*). It uses a plan library to reduce it to executable steps (like *Wait until 9 AM, then travel along Highway 24 to the test site. Stay there until noon*). Such plan fragments are likely to work, perhaps 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 may get several scenarios. Cf. Hanks 1990a, 1990b, 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 produces instead an explanation of what went wrong, in the form of a proof tree whose conclusion is of the form "Wrong thing W occurs." (E.g., "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 speed to 20; it can change "Not padded" by introducing a padding step. These are exactly the repairs that I listed before. (The speed repair will introduce a new bug, that the robot arrives late the to

The china breaks

It was rattled

  Traveled along a bumpy road at speed > 20 KPH

    Highway 24 is bumpy

    Traveled along Highway 24

    Speed = 40KPH

  Not padded

Figure 11: Explanation for plan bug

the test, which is repaired by starting earlier.) The repairs are effected by adding steps and changing argument values.

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 those 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 Section 2.3.) 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 may 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, because "the bomb" is not an effective designator (Section 2.2), and so can't be used to give commands to the gripper to pick the bomb up. Classical refinement planning provides ways of verifying that the plan "Put both objects in the toilet" will solve the the problem, but no way of generating that 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 distractor objects.
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 thus:

```
(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 foolproof, 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 question how plan revision fits in with the need to do planning under time constraints. There are two issues: How fast is transformational planning? and 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 will grow, 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 like 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 tarpit is in searching through the space of transformed plans. After every revision, the planner must reproject and recritique the plan. Some revisions turn out to introduce more bugs than they fix, so the planner must backtrack in revision space. The success of the whole paradigm turns on two assumptions:

1. Plans are already almost correct. (Cf. 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.
2. The planning process will have paid for itself when the planner finds a better plan than it started with, not necessarily the optimal plan. When that happens, it can switch to the improved version.

I know of one case history in which these assumptions have been observed empirically to be correct, the transformational planner of Zweben et al. (1990), which 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 (e.g., 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. Violations of constraints on start and end times are handled by rippling shifts through the set of tasks, using a technique due to Ow et al. (1988). Other constraint violations are then handled by specialized repair strategies. These 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 et al. 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 above was, How can transformational planning be combined with plan execution? In Zweben et al.'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 will cost 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 be executing the plan while the planner thinks about improving it (McDermott 1990). The planner should be active whenever (a) a new command has come from the robot's supervisor to be added to the current plan; (b) an old command has failed; (c) the planner has not yet tried every trick it knows 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 may sound as if this could cause the agent to burst into flames, but if plans are written in a robust enough way they should be able to cope with this situation. 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). (Cf. 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 100KPH. 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. (E.g., 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 keep track of all assumptions about the world state that are made by the planner as it makes them. If the agent knows that the world has diverged from those assumptions (based on information it has been acquiring as it goes), it can abort the planning process and then either attempt to get into a more quiescent state, or 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.

2.5 Robot Motion Planning

When real roboticists (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 — there's not much else that a robot does except move. The classical approach to motion planning (Lozano 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 will move along that trajectory. The trajectory can be thought of as the plan: "Move while staying close to this curve." See Figure 12.

This whole approach is surprisingly problematic. The main problem is that the space 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 there are degrees of freedom in the system, and the path planning problem becomes intractable as the degrees of freedom grow (Canny 1988). I will 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 will be hard to come by in most circumstances.

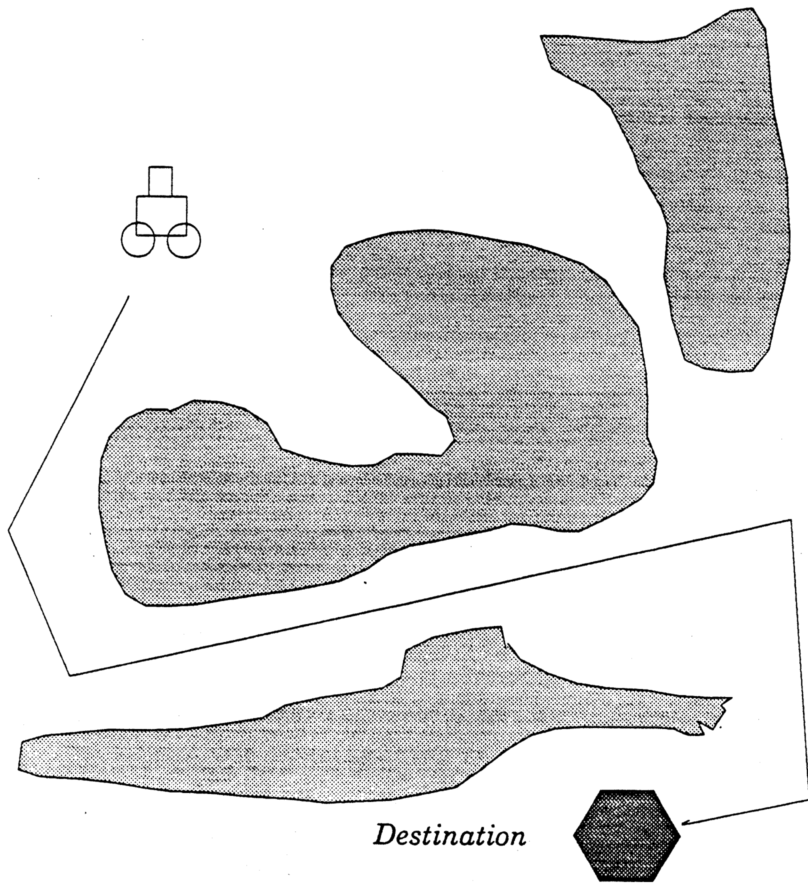


Figure 12: Motion planning

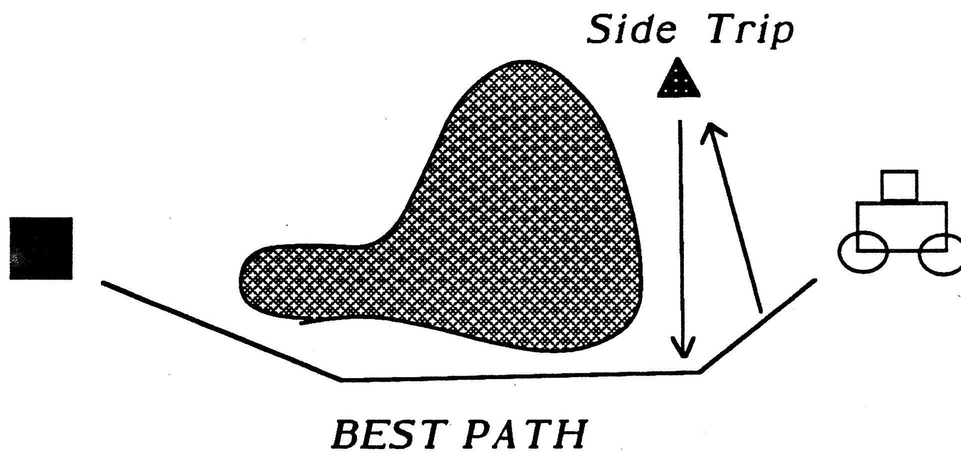


Figure 13: Sticking to a path can be a bad idea (Payton 1990)

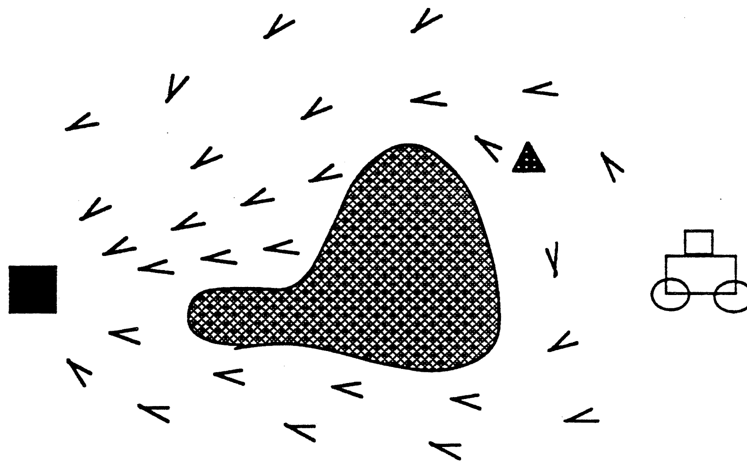
A third problem is that the method breaks the problem into two chunks: Find a path; stay near it. The path is generated using geometric criteria, and may not be suitable for a real robot to follow. If it has sharp corners, it will be almost impossible for the robot to track it accurately. Of course, it's usually unnecessary to track it very 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 may interrupt the task to go to the destination in order to make a side trip, or to attend to a high-priority interrupt. When the interruption is over, there's no particular reason to return to the previous path. See 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 thus:

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

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

This is an attractive idea at first glance, but it is not quite right. It requires computing



After side trip, follow local vector

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

Figure 14: Vector field shows travel directions

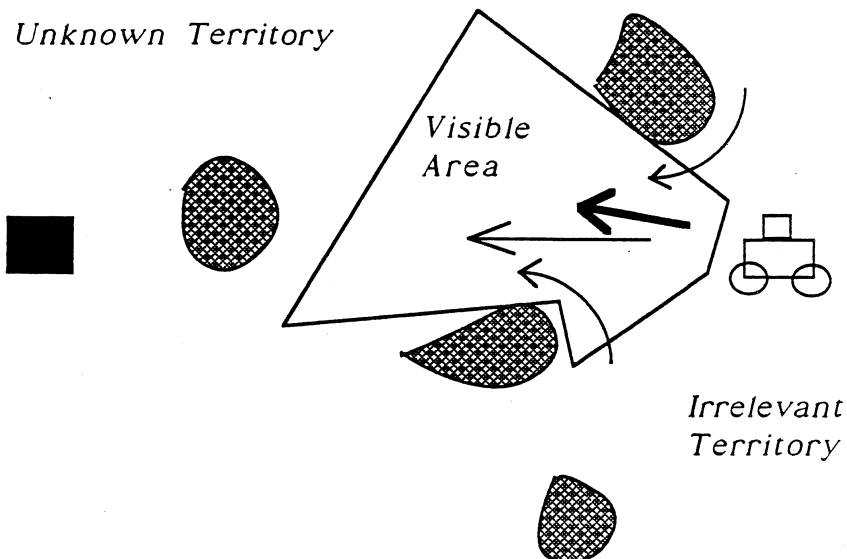


Figure 15: Local vector fields (Miller and Slack 1991)

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 will be meaningless if there are inaccuracies in the robot's map. The stored values will not be retrievable unless the robot is able to compute its X,Y location quite accurately 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) have designed and implemented such a scheme. The robot's destination generates a vector field which is then modified by vector fields associated with obstacles (Figure 15). Obstacles are recognized by transforming range data into elevation data. (Cf. 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 in order to maneuver around obstacles it can't see, and (in most cases) in order to know the direction of its destination. Locally visible obstacles can be dealt with directly. At any instant, a set of obstacles are selected as relevant to path planning. A decision is made to go around each obstacle either clockwise or counterclockwise. That decison 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 now. 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 will look like a reasonably attractive place to be for just about any potential-field approach.[3]

Clearly, what's needed is a model of how a robot (a) builds a world map as it moves through the world; (b) uses that map for long-range path planning (in spite of its incompleteness); (c) makes use of local cues (as in Figure 15) to adjust as it goes; (d) switches to an entirely different path whenever unknown areas turn out to be full of obstacles. Job (a) is beyond the scope of this paper (but see Kuipers and Byun 1988, Thorpe et al. 1988, Mataric 1990, and the survey in McDermott 1992). The algorithm for building the world map is presumably not part of the plan, but an autonomous black box as far as the planner is concerned. (The act of exploring in order 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 Section 2.4.

2.6 Learning

So far 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

[3]Koditschek (1987) has proven 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.

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 behaver. What gets learned is therefore 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. The former occurs at plan time, when the planner anticipates and fixes a bug arising from a novel combination of plans. The latter 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 does this 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 those statistics. We can assume our plans look like

```
(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 the its $action_i$. The condition may include coin flips as well as tests of sensory inputs. The learner does ***not*** attempt to construct new actions. In our current context, it would be quite useful to have a theory of constructing or adapting new plans for the plan library, but there hasn't been

much work on that. (But see Hammond 1988.)

The learner gets feedback about how appropriate its past actions have been. There are various alternative assumptions we can make about this feedback signal:

1. It can be *deterministic* or *stochastic*. In our context, we will always assume the latter. Reinforcements are correlated with actions, but even the "correct" action can get a "bad" feedback signal some of the time.
2. It can be *stationary* or *nonstationary*, depending on whether the world's behavior changes over time.
3. It can be *binary* or *graded*. In the former case, all we know is that previous actions were "good" or "bad." In the latter, we have some notion of how far they were from right. (The extra information helps only if an action is parametrized in such a way that we know how to change it "a little bit.")
4. It can be *immediate* or *delayed*.
5. It can be *memoryless* or *state-dependent*.

The last two items are closely related. Figure 16 will clarify 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 asychronous model), and the signal depends only on the pair $\langle I_d, A_d \rangle$, that's *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. That's delayed feedback. Now suppose that the feedback signal depends, not just on $\langle I_d, A_d \rangle$, but on the past sequence of inputs $\ldots, I_{d-2}, I_{d_1}, I_d$. That's state-dependent feedback, so-called because the feedback depends on the state the robot has been driven into by prior inputs.

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

Unfortunately, this scenario assumes that there are only a few distinct inputs, and that what to do when one input is seen is independent of what to do when any other input is seen. But

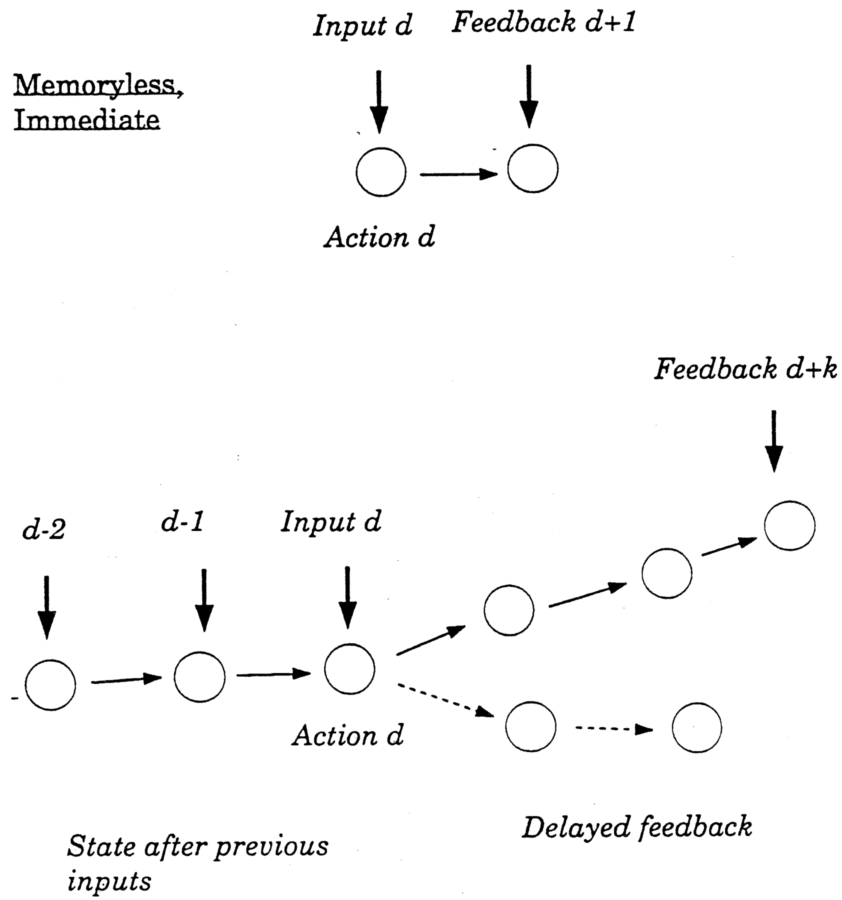


Figure 16: Feedback regimes

suppose the input is a pattern of N bits. Then there are 2^N possible inputs, and those that share bits are likely to indicate the same action. At this point we need to introduce the idea of a ***hypothesis space*** (Haussler 1988) that describes the possible hypotheses defined on the bit vector, and we need to introduce algorithms that search through this space. At this point all of computational learning theory opens up before us.

I will confine myself to examining one algorithm, due to Kaelbling (1990), called GTRL. (It's based on Schlimmer's STAGGER algorithm (Schlimmer 1987).) For the purpose of exposition, I will 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 inputs, and it keeps track of 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 that 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

$$\lfloor 1000\, er(h) \rfloor + er\text{-}ub(h)$$

where (very roughly) $er(h)$ is the expected reinforcement for a hypothesis h, based on comparison of past behavior with what h recommended, and $er\text{-}ub(h)$ is an upper bound e such that the chance of the true value of $er(h)$ being higher than e is less than 5%. As more data are gathered, $er\text{-}ub$ and er will converge, but initially the $er\text{-}ub$ figure is significantly higher, and biases 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, up 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. When creating conjunctions, it uses "necessity," a measure of how well a hypothesis does when it recommends action 0.

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 involving 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 like *Swing leg K forward* or *Swing leg K backward,* and the reinforcement signal was whether the belly stayed off the floor and 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 kept track of its current best hypothesis regarding which conjunction of conditions should trigger that action. New sensor inputs 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.

So far I have been assuming that feedback is immediate and memoryless. 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 *temporal-difference (TD) methods* for learning (Sutton 1988). The TD 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'll use the current best guess regarding eventual reinforcement as the reinforcement signal for the action learner. Those 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, and 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

γ = discount rate for future reinforcement. That is, the total lifetime reinforcement as measured at time d is

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

So that reinforcements in the remote future are counted as less important.

Assuming the robot experiences all inputs infinitely often (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 $\hat{V}(i_d, a_d)$:

$$\begin{aligned}\Delta\hat{V}(i_d, a_d) &= \alpha(R(d+1) + \gamma\hat{V}(i_{d+1}, a_{d+1}) \\ &\quad - \hat{V}(i_d, a_d))\end{aligned}$$

Note that the quantity

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

will be 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 α.

Temporal-difference methods are appealing for their simplicity, theoretical properties, and practicality. (See Sutton 1988, Kaelbling 1990.) However, they do not as yet solve some of the hard problems. One of the hardest such problems is the problem of state-dependent feedback. One way to model it is to broaden the set of inputs to include input sequences up to some length. However, this option is obviously combinatorially explosive. As usual, it is not hard to think of situations that will baffle any learning algorithm. Learning makes the most sense when it is thought of as filling in the details in an algorithm that is close to complete as written. An example is map learning, in which the agent's techniques are specialized to the case of figuring out the shape of its environment. This problem is virtually impossible to solve when cast in terms of maximizing reinforcement, unless the world is shaped very simply, and the same sorts of rewards 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.

3 Conclusions

The main conclusion I wish 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

1. Limitations on quantity and 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.

2. 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 will 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, computational learning theory, and many others. [4]

One topic that has really been 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 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
- Control theory

[4] I have a feeling that my choice of topics may mislead the less informed reader. So let me make it clear that in a practical system the code required to do domain-specific planning (e.g., to assemble a set of parts) is likely to dwarf the code required to do the kinds of projection and transformation that I have discussed.

This may 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 may 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 will anticipate that 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, interruptable 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.

4 Bibliography

1 Philip E. Agre and David Chapman 1987 Pengi: an implementation of a theory of activity. *Proc. AAAI* **6**, pp. 268–272

2 Philip E. Agre and David Chapman 1990 What are plans for? In Maes 1990

3 James Allen, James Hendler, and Austin Tate 1990 *Readings in planning.* San Mateo, CA: Morgan Kaufmann Publishers, Inc.

4 Mark Boddy and Thomas Dean 1989 Solving time-dependent planning problems. *Proc. Ijcai* **11**

5 Rodney A. Brooks 1982 Solving the find-path problem by good representation of space. *Proc. AAAI* **2**

6 Rodney A. Brooks 1986 A robust layered control system for a mobile robot. *IEEE J. of Robotics and Automation* **RA-2**, pp. 14–23

7 Rodney A. Brooks 1991 Intelligence without representation. *Artificial Intelligence* **47**, special issue on foundations of AI, edited by David Kirsh, pp. 139–159

8 John Canny 1988 *The complexity of robot motion planning.* ACM Doctoral Dissertation Series. Cambridge: MIT Press

9 David Chapman 1987 Planning for conjunctive goals. *Artificial Intelligence* **32**, pp. 333–377

10 David Chapman 1990 Vision, instruction, and action. MIT AI Lab Tech Report 1204

11 Eugene Charniak and Drew McDermott 1985 *Introduction to Artificial Intelligence.* Reading, Mass.: Addison-Wesley

12 Gregg C. Collins 1987 *Plan creation: using strategies as blueprints.* Yale University Ph.D. Dissertation

13 Jonathan H. Connell 1990 *Minimalist mobile robots.* Boston: Academic Press, Inc.

14 Thomas Dean and Mark Boddy 1988 An analysis of time-dependent planning. *Proc. AAAI* **7**, pp. 49–54

15 Mark Drummond and John Bresina 1990 Anytime synthetic projection: maximizing the probability of goal satisfaction. *Proc. AAAI* **8**, pp. 138–144

16 R.J. Firby 1987 An investigation into reactive planning in complex domains. *Proc. AAAI* **6**, pp. 202–206

17 R.J. Firby 1989 *Adaptive execution in complex dynamic worlds.* Yale University CS Dept. TR 672

18 Michael Georgeff and Amy Lansky 1986 Procedural knowledge. *Proc. IEEE Special Issue on Knowledge Representation* **74**, pp. 1383–1398

19 Michael Georgeff and Amy Lansky 1987 Reactive reasoning and planning. *Proc. AAAI* **7**, pp. 677–682

20 Naresh Gupta and Dana Nau 1991 Complexity results for blocks-world planning. *Proc. AAAI* **9**

21 Kristian Hammond 1988 *Case-based planning: an integrated theory of planning, learning, and memory.* New York: Academic Press.

22 Kris Hammond 1990 Explaining and repairing plans that fail. *Artificial Intelligence* **45**, no. 1–2, pp. 173–228

23 Steven Hanks 1990 *Projecting plans for uncertain worlds.* Yale Yale Computer Science Department Technical Report 756.

24 Steven Hanks 1990 Practical temporal projection. *Proc. AAAI* **8**, pp. 158–163

25 David Haussler 1988 Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence* **36***(2)*, pp. 177–222

26 James Hendler, Austin Tate, and Mark Drummond 1990 AI planning: systems and techniques. *AI Magazine* **11** (2), pp. 61–77

27 Eric Horvitz, G.F. Cooper, and D.E. Heckerman 1989 Reflection and action under scarce resources: theoretical principles and empirical study. *Proc. Ijcai* **11**, pp. 1121–1127

28 Leslie Pack Kaelbling 1987 An architecture for intelligent reactive systems. In Michael Georgeff and Amy Lansky (eds.) *Reasoning about Plans and Actions*, Morgan Kaufmann

29 Leslie Pack Kaelbling 1988 Goals as parallel program specifications. *Proc. AAAI* **7**, pp. 60–65

30 Leslie Pack Kaelbling 1990 *Learning in embedded systems.* Stanford University Ph.D. thesis. Teleos Research Report 90-04.

31 Leslie Pack Kaelbling and Stanley J. Rosenschein 1990 Action and planning in embedded agents. In Maes 1990, pp. 35–48

32 Leslie Pack Kaelbling and Nathan J. Wilson 1988 *REX programmer's manual.* SRI Technical Note 381R

33 Osama Khatib 1986 Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. of Robotics Res.* **5** (1), pp. 90-98

34 Scott Kirkpatrick, C.D. Gelatt, and M.P. Vecchi 1983 Optimization by simulated annealing. *Science* **220**, p. 4598

35 Daniel Koditschek 1987 Exact robot navigation by means of potential functions: some topological considerations. *Proc. IEEE Int. Conf. on Robotics and Automation*, Raleigh, N.C., pp. 1–6

36 Yoram Koren and Johann Borenstein 1991 Potential field methods and their inherent limitations for mobile robot navigation. *Proc. IEEE Conf. on Robotics and Automation*, Sacramento, pp. 1398–1404

37 Benjamin Kuipers and Yung-tai Byun 1988 A robust, qualitative method for robot spatial reasoning. *Proc. AAAI* **7**, pp. 774–779.

38 Jean-Claude Latombe 1991 *Robot motion planning.* Boston: Kluwer Academic Publishers

39 S. Lin and B.W. Kernighan 1973 An effective heuristic algorithms for the traveling salesman problem. *Operations Res.* **21**, pp. 498–516

40 Tomás Lozano-Pérez 1983 Spatial planning: a configuration space approach. *IEEE Trans. on Computers* **C-32***(2)*, pp. 108–120

41 D.M. Lyons 1990 A process-based approach to task-plan representation. *Proc. IEEE Conf. on Robotics and Automation*

42 D.M. Lyons and M.A. Arbib 1989 A formal model of computation for sensory-based robotics. *IEEE Trans. on Robotics and Automation* **5**, no. 3, pp. 280–293

43 D.M. Lyons, A.J. Hendriks, and S. Mehta 1991 Achieving robustness by casting planning as adaptation of a reactive system. *Proc. IEEE Conf. on Robotics and Automation*, pp. 198–203

44 Patti Maes (ed.) 1990 *New architectures for autonomous agents: task-level decomposition and emergent functionality*, Cambridge: MIT Press

45 Pattie Maes and Rodney A. Brooks 1990 Learning to coordinate behaviors. *Proc. AAAI* **8**, pp. 796–802

46 Maja J. Mataric 1990 *A distributed model for mobile robot environment-learning and navigation.* MIT AI Lab Report 1228

47 David McAllester and David Rosenblitt 1991 Systematic nonlinear planning. *Proc. AAAI* **9**

48 Drew McDermott 1987 A critique of pure reason. *Computational Intelligence* **3**, no. 3, pp. 151–160.

49 Drew McDermott 1990 Planning reactive behavior: a progress report. *Proc. Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pp. 450–458

50 Drew McDermott 1991 A reactive plan language. Yale Computer Science Report 864

51 Drew McDermott 1992 Spatial reasoning. Article in *Encyclopedia of Artificial Intelligence*, second edition, S. Shapiro, editor.

52 David Miller and Marc G. Slack 1991 Global symbolic maps from local navigation. *Proc. AAAI* **9**

53 Tom M. Mitchell 1990 Becoming increasingly reactive. *Proc. AAAI* **8**, pp. 1051–1058

54 Dana S. Nau 1982 An investigation of the causes of pathology in games. *Artificial Intelligence* **19**, pp. 257–258

55 Dana S. Nau, Qiang Yang, and James Hendler 1990 Optimization of multiple-goal plans with limited interactions. In Sycara 1990, pp. 160–165

56 Nils J. Nilsson 1988 Action networks. In *Proc. Rochester Planning Workshop*, pp. 20–51

57 Donald A. Norman and Daniel G. Bobrow 1975 On data-limited and resource-limited processes. *Cognitive Psychology* **7**, pp. 44–64

58 Peng Si Ow, Steven Smith, and A. Thiriez 1988 Reactive plan revision. *Proc. AAAI* **8**

59 David Payton 1990 Exploiting plans as resources for action. In Sycara 1990, pp. 175–180

60 Richard Pelavin 1988 A formal approach to planning with concurrent actions and external events. University of Rochester Department of CS Ph.D. thesis

61 Stanley J. Rosenschein 1989 Synthesizing information-tracking automata from environment descriptions. *Proc. Int. Conf. on Principles of Knowledge Representation and Reasoning* **1**, Toronto

62 Stuart Russell and Eric Wefald 1991 Principles of metareasoning. *Artificial Intelligence* **49** (1–3), pp. 361–395

63 Jeffrey C. Schlimmer 1987 Learning and representation change. *Proc. AAAI* **6**, pp. 511–515

64 Marcel Schoppers 1987 Universal plans for reactive robots in unpredictable environments. *Proc. Ijcai* **10**, pp. 1039–1046

65 Reid Gordon Simmons 1988a A theory of debugging plans and interpretations. *Proc. AAAI* **7**, pp. 94–99.

66 Reid Gordon Simmons 1988b *Combining associational and causal reasoning to solve interpretation and planning problems.* MIT AI Laboratory TR 1048.

67 Reid Gordon Simmons 1990 An architecture for coordinating planning, sensing, and action. In Sycara 1990, pp. 292–297

68 Reid Gordon Simmons 1991 Concurrent planning and execution for a walking robot. *Proc. IEEE Conf. on Robotics and Automation*, pp. 300–305

69 Richard Sutton 1988 Learning to predict by the method of temporal differences. *Machine Learning* **3**, pp. 9–44

70 Gerald J. Sussman 1975 *A computer model of skill acquisition.* American Elsevier Publishing Company

71 Katia Sycara (ed.) 1990 *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control.* San Mateo: Morgan Kaufmann Publishers, Inc.

72 Austin Tate 1977 Generating project networks. *Proc. IJCAI* **5**, pp. 888-893

73 Charles Thorpe, Martial H. Hebert, Takeo Kanade, and Steven Shafer 1988 Vision and navigation for the Carnegie-Mellon Navlab. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **10**, no. 3, pp. 362–373

74 W. Grey Walter 1950 An imitation of life. *Scientific American* **182**(5), p. 42.

75 David Wilkins 1988 *Practical planning: extending the classical AI planning paradigm.* San Mateo, CA: Morgan Kaufmann Publishers, Inc.

76 Qiang Yang 1989 Improving the efficiency of planning. University of Waterloo Mathematics Department Report CS-89-34. (Also a Ph.D. thesis from the University of Maryland)

77 Monte Zweben, Michael Deale, and Robert Gargan 1990 Anytime rescheduling. In Sycara 1990, pp. 251–259