PLANNING

In artificial intelligence (AI), *planning* is the activity of finding in advance some course of action that promises to make true or keep true some desirable features in the world, if and when executed by an agent. An agent here may be a human, a robot, a group of these, a technical process—any system that can change its environment in a well-defined way. The agent executing the plan may differ from the one generating the plan.

A planning algorithm or planning system, then, has to work on a problem of the following structure:

given a description of a world state in which some agent finds itself, descriptions of actions that the agent can execute in the world, and goals that should become true or remain true in the world,

find a plan, i.e., a specification of how to act in the world, that, when executed successfully, will fulfill the goals.

Depending on the precise syntax, semantics, and pragmatics of world states, actions, goals, and plans, there are a large variety of instances of planning. For example, the goals may be described by a set of ground facts or by a formula of propositional logic (syntactic difference), the available description of the current world state may be assumed to be accurate or not (semantic difference), and the quality of a plan may or may not depend on the time when it is found, that is, a mediocre plan in time may be better than a perfect one too late (pragmatic difference). All these differences—and their combinations, as far as they make sense—must be mirrored by differences in the respective algorithms and representation languages.

Planning algorithms and techniques are being used for a great variety of applications. Typical application areas are scheduling and logistics.

The following sketch of the field starts with a fairly comprehensive description of basic planning methods that make some strong assumptions about its application domains, thereby gaining simplicity of the representation formalisms and algorithms involved; then some more advanced planning methods are described; after that, we address some typical planning applications; we conclude with a summary of the history of AI planning and literature for further study.

BASIC PLANNING

The best-studied planning method—or set of methods, in fact—is so-called classical planning. As the name suggests, it is also a method that has been in use for quite some time, with the planning system STRIPS (Stanford Research Institute Problem Solver) (1) as a cornerstone laid in the late sixties. Classical planning is sometimes identified with certain planning systems, such as STRIPS, with algorithms, or representations that are used or avoided within a planner. From today's perspective, however, it is best described in terms of the central simplifying assumptions that it makes about its domains:

1. The relevant features of the world can be described in terms of static "snapshots" or states.
2. All relevant world features are known; all that is known about the world is accurate.
3. Only actions of the agent change world states; no two actions are executed in parallel.
4. Time only occurs as the transition from state to state by acting; there is no notion of duration.
5. An action is adequately described by its preconditions and postconditions, i.e., by the features that must be true about the world prior to action execution and by the features whose truth changes by executing the action.
6. The effect of an action is deterministic; it is context-free in the sense that it is not affected by what is true or false in the world other than the action preconditions.
7. On successful termination of planning, all actions in the plan have to be executable at their respective time slots within the plan; the plan has to fulfill all given goals.
8. Plan execution succeeds planning.
9. The time for computing a plan has no effect on plan quality.

While it is obvious that a description of the world that makes these assumptions may be somewhat simplistic, it does lead to a way of planning that is sometimes useful as a first, nontrivial approximation. Besides, classical planning is a good start to understand basic problems and techniques of planning in general. A number of advanced planning techniques are described further below, in which some of the classical assumptions are relaxed.

More comprehensive descriptions of classical planning include Refs. 2 and 3; it is also typically contained in AI textbooks, such as Ref. 4.

Basic Concepts

We start the description of classical planning by introducing some basic definitions and notation.

A state in the world is represented by a set of ground propositions of some given domain description language $\mathscr{L}$, each proposition representing a feature of the state. We call such a representation of a world state a *situation.* All propositions contained in a situation are assumed to be true in the corresponding world state; everything not contained in a situation is assumed to be false in the corresponding state (closed-world assumption).

Actions of the agent are represented by *operators.* An operator is a triple of the form $o = \langle P, D, A \rangle$, where P, D, A are sets of ground propositions from the language $\mathscr{L}$. P denotes o's preconditions, i.e., the state features that must be true in order to apply o. D and A describe the postconditions, D (the delete conditions) specifying what ceases to be true after o is applied, and A (the add conditions) specifying what executing o makes true. Operators of this $\langle P, D, A \rangle$ format are often

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering.* Copyright © 1999 John Wiley & Sons, Inc.

called STRIPS operators after the planner that first introduced them (1). The original STRIPS, like many other planners, does not make the restriction that all propositions involved in operators must be ground, but variables are allowed, which get bound to object constants whenever required; this may happen during planning or else immediately before execution. The version of classical planning described here is in fact *propositional* classical planning, and object variables are not handled here.

It may happen that instances of an operator occur more than once in a plan; for example, a plan for cleaning the house may contain the operator for getting fresh water several times. To denote uniquely different occurrences of an operator in a plan, every such operator occurrence is labeled uniquely. (In the rest of this article, these labels are skipped for brevity.)

A *plan,* then, is a pair $\Pi = \langle O, \prec \rangle$, where O is a set of operator occurrences, which we will briefly call the *operator set*. $\prec$ is an ordering on O, i.e., an irreflexive and transitive relation on O, which is to be interpreted as the operator execution order. If the order $\prec$ is total, then Π is called *linear* or *total-order;* otherwise it is called *nonlinear* or *partial-order*. If, in the nonlinear case, $o_1, o_2 \in O$ are not ordered by $\prec$, this is to be interpreted as saying that the respective actions may be executed in either order. (Note that it was assumed above that actions can only be executed one at a time.)

The effect of executing an action in the world is calculated for the corresponding operators and situations in the following way. If S is a situation, then the successor situation S' that results from applying o in S is defined by

$$S' = \begin{cases} S & \text{if} \quad P \not\subseteq S \\ (S \setminus D) \cup A & \text{else} \end{cases} \tag{1}$$

A classical planning *problem,* which can be given for a planner to solve, consists of a domain description language specifying all propositions, objects, and operators that exist in the domain, of an initial situation that describes the state of the world as is, and of a set of goal propositions. A *solution* of such a planning problem is a plan that, given the initial situation, yields a situation that includes all goal propositions. As there may be many solutions, usually a plan with a minimal operator set is preferred.

To exemplify all this, let us turn to an example domain that is classical in AI research: the blocks world. Note that this domain is chosen here for the didactic purpose of being easy to understand and to present and for its property of allowing a spectrum of difficulty grades from easy to very rich. Planning domains that are of practical relevance will be addressed below.

The blocks world consists of a flat surface, such as a table top; toy blocks; and agents that are able to manipulate the blocks, such as a robot arm that can grip and move blocks. A typical planning problem would specify a block building to be constructed. Instances of the blocks world may differ in the number of blocks, in block features such as size, shape or color, in the granularity of actions, in the number of agents guided by the plan, in the presence of malevolent other agents, in differences of cost and benefit of operators, in the possibility of malfunction of action execution, and many other features that would come along with relaxing the classical planning assumptions above.

In harmony with the strictness of these assumptions, a blocks world version for classical planning can only be simple in structure. Figure 1 gives a small example. Objects involved are the blocks A, B, C; the constant NIL denotes "nothing." In any state, some block x can be gripped [Hand(x)], be sitting on the table [Ontbl(x)], be stacked on another block y [On(x, y)], and/or be clear of other blocks [Clear(x)]. The gripper can hold only one block at a time, and only one block can sit on top of another block; all blocks can be sitting on the table at the same time; in any state, a block is either on the table, or on another block, or gripped.

The latter constraints are reflected in the preconditions and postconditions of the operators in Fig. 1(a). **PICK** and **PUT** represent the actions of moving a block x from or on the table, respectively. **STACK** and **REMOVE** represent stacking x on block y and taking it off, respectively. All these operators happen to delete all of their preconditions; this is not generally the case.

Figure 1(b) gives an example problem in this domain. The initial situation is given as a set of propositions (with a drawing of the corresponding world state); the set of goal propositions is given below. In the blocks world sketched here, intuition tells that there is exactly one world state in which these goal conditions are true, i.e., the goal state is uniquely characterized by the goal condition set. This is not generally true, i.e., there may be many different goal states for a planning problem.

Figure 2 shows a solution to the problem. The plan is linear; in fact, there is no nonlinear plan for solving this problem as defined in Fig. 1. It is also the shortest solution plan. There are infinitely many solutions, as it is possible to insert infinitely long detours, such as putting down and immediately picking up one block arbitrarily often.

PICK(x)

Pre: {Ontbl(x), Clear(x), Hand(NIL)}
Del: {Ontbl(x), Clear(x), Hand(NIL)}
Add: {Hand(x)}

PUT(x)

Pre: {Hand(x)}
Del: {Hand(x)}
Add: {Ontbl(x), Clear(x), Hand(NIL)}

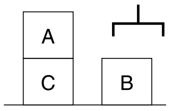
STACK(x, y)

Pre: {Hand(x), Clear(y)}
Del: {Hand(x), Clear(y)}
Add: {Hand(NIL), Clear(x), On(x, y)}

REMOVE(x, y)

Pre: {Hand(NIL), Clear(x), On(x, y)}
Del: {Hand(NIL), Clear(x), On(x, y)}
Add:{Hand(x), Clear(y)}

(a)



{Ontbl(C), Ontbl(B), Clear(B), Clear(A), On(A, C), Hand(NIL)}

$\leadsto$

{On(A, B) On(B, C)}

(b)

Figure 1. Planning problem example for classical planning in the blocks world. Operator schemata are given in (a), where the variables x, y have to be replaced by block names A, B, C in all possible ways to form operators from the schemata. Part (b) shows the start situation and the goal propositions.

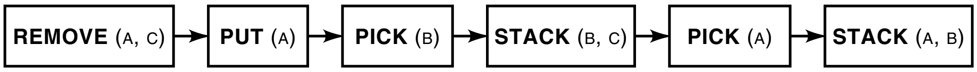


Figure 2. A plan to solve the problem defined in Fig. 1. Operators are drawn as boxes; the plan ordering is represented by the arrows.

Before turning to algorithms for plan generation, a statement can be made about the computational complexity of planning, independent of which algorithm is used. Simple as it may seem, the problem of determining the existence of a solution of propositional classical planning as defined above is PSPACE-complete (5). That means that, in general, the run time of a planning algorithm is likely to grow exponentially with the size of its input, which is determined by the number of operator types and objects. Plan existence in a slightly more general, nonpropositional classical planning variant is even undecidable (6). So there are hard fundamental efficiency limits for planners in general. That does not mean, of course, that no practical planners exist that are efficient for their application domains. Moreover, certain planning algorithms may still be better than most others in most cases, so empirical complexity analyses certainly play a role.

The Algorithmics of Classical Planning

A variety of algorithms exist for classical planning. This subsection presents the basic concepts and ideas for many of them and explains a simple one in some detail; the following two subsections deal with how to enhance efficiency and expressivity in this framework.

General Views of Planning. Let us start with some general considerations concerning the abstract view of planning. In AI, there has been the debate whether problem solving should best be seen as search or as deduction, and this debate is also alive for planning. Good arguments exist for both views, a strong one for planning-as-deduction being that it allows an agent to generate plans and do general reasoning about the domain within the same representation. Mainstream planning research has followed the planning-as-search view, around which most of this article is centered. Before continuing with this view, let us sketch deductive planning.

Deductive Planning. The planning-as-deduction view sees planning as the problem of finding a formal, deductive proof that a sequence of operators exists that would transform the present state into a goal state. This proof has to be constructive, so that having found it implies having found a plan. The influential paper by Green (7) proposed to represent the problem in first-order predicate calculus in a straightforward way: All world features are represented as predicates, which get an additional situation argument; operators would be represented as implications from preconditions to postconditions; operator application is represented by a situation term made of a logical function that corresponds to the operator. A standard theorem prover can then be used for finding the proof/plan. Along these lines, an analog of the **PUT** operator of Fig. 1 would be

$$\begin{aligned}\forall x, s.\quad & [\mathsf{Hand}(x, s) \rightarrow \\ & \neg\mathsf{Hand}(x, \mathrm{PUT}(x, s)) \wedge \\ & \mathsf{Ontbl}(x, \mathrm{PUT}(x, s)) \wedge \mathsf{Clear}(x, \mathrm{PUT}(x, s)) \wedge \\ & \mathsf{Hand}(\mathrm{NIL}, \mathrm{PUT}(x, s))]\end{aligned} \tag{2}$$

However, this specification is not yet complete for specifying the operator, as this representation suffers from the infamous *frame problem.* Its essence is that each and every operator formulated in the straightforward logical notation of Eq. (2) needs to represent in its postcondition not only what it changes (i.e., adds or deletes), but also what it leaves untouched. Specifying this is prohibitively cumbersome for all practical purposes; for example, to complete the description of the **PUT** operator in Eq. (2), one has to specify additionally that all blocks different from x that were Ontbl in s are still Ontbl in the situation $\mathrm{PUT}(x, s)$, that all clear blocks remain clear, that all blocks on other blocks remain on them, and so on.

Some have taken the frame problem as an argument against deductive planning in general, but this is not warranted. Other logics (typically, leading to planning variants that are more powerful than classical planning) and other, more effective ways of deductive planning exist in which the frame problem does not arise. Reference 8 presents arguments for the deduction view of planning as well as further references.

Search Spaces. If planning is seen in the abstract as search, then the search space needs to be made explicit. Again, two views have been prominent: situation-space planning and plan-space planning.

Situation-space planning sees the problem space of planning like this: Its nodes correspond to the situations, and its transitions correspond to the operators applied in the respective situations. Planning, then, means to find a path through this search space, typically by forward search, from the start situation to a goal situation; this path represents a sequence of operators, i.e., a plan. Figure 3 shows a part of the corresponding problem space that contains the start situation of the planning problem in Fig. 1.

Situation-space planning is intuitive, but it poses some practical difficulties. For example, as there may be many different goal states, it is harder than in plan-space planning to search goal-directedly without explicitly dealing with the many different paths in the space that might eventually lead to different goal states. Without postprocessing, situation-space planning returns only linear plans, which may overconstrain the operator sequence. Elements of situation-space planning have recently reappeared, though, as a fast preprocessing method for classical planning, which is followed by a plan-space planning pass. The idea, which was first used in GRAPHPLAN (9), will be sketched in the subsection on enhancing efficiency.

The majority of classical planners search the plan-space. Nodes in this search space are plans, i.e., $\langle O, \prec \rangle$ pairs. A goal node in this search space is a plan that is "O.K." in the sense of being executable and leading to a goal situation when executed; most nodes in the search space correspond to plans that are "not O.K." A transition in the search space from a node n is effected by a change to the plan that n represents: for example, an operator can be added, or two operators in

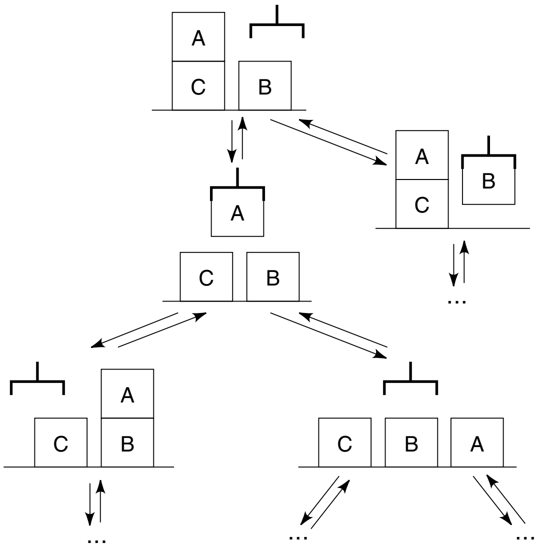


Figure 3. Part of the situation space for the blocks world of Fig. 1. The arrows are labeled by the operators that they represent (unique labeling in this blocks world version; omitted in the drawing to enhance clearness). Planning in situation space means finding a path in the graph from the node representing the start state to a note representing a goal state.

the plan can be ordered. The solution to the planning problem is the plan with which the goal node is labeled that is eventually found.

The basic planning algorithm that is developed in the following section is an example for plan-space planning.

Generating Nonlinear Plans

Basic Concepts and Definitions. Although it is part of the assumptions for classical planning that only one operator can be executed at a given time, a least-commitment strategy with respect to the operator order is often desirable: the plan should contain only the ordering restrictions that are absolutely necessary, leaving it for decision at plan execution time which one of possibly several executable operators is actually executed next. Moreover, a plan with a nonlinear order represents a family of linear plans, namely, all those whose linear order is compatible with the given nonlinear one; in consequence, the planning effort for a nonlinear plan is in fact performed on the whole family of linear plans that it represents. In the abstract, that sounds computationally efficient.

On the other hand, working with nonlinear plans requires some conceptual and computational overhead for determining whether a given plan is "O.K." Before describing a simple algorithm for generating nonlinear plans, we have to introduce some basic concepts: dependencies and conflicts in a plan.

As a necessary criterion for applicability of an operator, all of its preconditions must have been established before its position in the plan according to the operator order. That is, a precondition fact must have been contained in the initial situation, or been added by an earlier operator. As a representation convention, we represent the initial situation by a special operator $\mathcal{I} \in O$ that has no preconditions, deletes nothing, and adds all features of the initial state. The goal features are represented by a special operator $\mathcal{G} \in O$ whose preconditions are the goals and whose postconditions are empty. $\mathcal{I}$ precedes all other operators in the plan; all other operators precede $\mathcal{G}$.

The dependency structure of a plan describes which operator produces which condition for which other operator. Intuitively, a plan contains a dependency from an operator p (the "producer") to an operator o (the "beneficiary") with respect to (wrt) a condition c if p adds c, c is a precondition of o, and no other operator q adds it in between:

Definition 1 (Dependency). Let $\Pi = \langle O, \prec \rangle$ be a plan and $o, p \in O$. Then $\delta_\Pi = \langle p, c, o \rangle$ is a *dependency* between p and o wrt c in Π if and only if p adds c, c is a precondition of o, $p \prec o$, and no other $q \in O$ adds c such that $p \prec q \prec o$.

A finished plan must have all preconditions of all its operators *resolved* in the sense that it contains a dependency wrt each and every precondition. That is not sufficient for a plan to be "OK," though: Once added, a condition might be deleted by another operator (the *destroyer*) between the producer and the beneficiary of a dependency, if the operator order allows for executing the destroyer in between. In this case, the plan contains a conflict:

Definition 2 (Conflict). Let $\Pi = \langle O, \prec \rangle$ contain a dependency $\delta_\Pi = \langle p, c, o \rangle$; let $d \in O$ be an operator that deletes c. Then Π contains a conflict between δ_Π and d if and only if there is an ordering relation $\prec'$ on O that extends $\prec$, that is, $\prec \subseteq \prec'$ such that $p \prec' d \prec' o$, and c does not get reestablished between d and o, that is, there is no $p' \in O$ adding c such that $d \prec' p' \prec' o$.

A precondition of some operator in a plan is called *unresolved* if either there is no dependency with respect to it or the plan contains a conflict with respect to its dependency.

A Basic Planning Algorithm. Building on the concepts of dependency and conflict, a basic planning algorithm can work like this. As an input, it gets a plan, which is initially the plan consisting of just the operators $\mathcal{I}$ and $\mathcal{G}$. Whenever the recent plan contains no unresolved preconditions, then this plan is the result. Else the algorithm nondeterministically chooses among a small number of options for resolving a precondition, which will be discussed next, and continues with the modified plan. It is obvious that this algorithm works in the plan space; given that there are several choice points, it is also obvious that it has to perform search. The CPP algorithm in Fig. 4 formulates this explicitly. The choice points may simply be implemented by backtracking; sophisticated search strategies are possible, but they are of no interest for the moment.

In its steps 3 and 4, CPP uses four different ways of resolving open preconditions. In the case that the condition c is unresolved for lack of an appropriate producer (step 4), one may either insert a new operator at the right place, or employ one that is already in the plan by ordering it before the beneficiary. As formulated in Fig. 4, this ordering restriction is executed without further check of whether it is allowable. In general, that may result in an *inconsistent* ordering $\prec$, that is, an ordering that contains a cycle of the sort $q \prec q$ for some operator q. Such an ordering leads to failure (and hence backtracking) in step 0.

Inserting a fresh operator and employing an existing one are also ways to resolve a conflict as in step 3. Alternatively, the destroyer may be promoted (put before the producer) or demoted (put after the beneficiary); this resolves the conflict, but may lead to an inconsistent ordering and also to backtracking in step 0.

In principle, there is another way to resolve a conflict, namely, withdraw the destroyer from the plan. This is usually not done in algorithms for classical planning: they rely on monotonic growth of the operator set and on an appropriate search strategy for finding a plan. It has even been shown (10) that monotonicity of the dependency set in a plan can, under certain additional conditions, lead to *systematicity* of a planning algorithm, that is, the property that during its search in the plan space the planning algorithm will generate the same plan at most once. This requires a different plan definition that mentions explicitly and keeps under control the set of dependencies. In these algorithms, conflict (or *threat,* as it is commonly called in the literature) arises not only if the condition of a dependency may be deleted, but also if it may be produced by a different operator than originally intended.

Note that the way in which new operators are inserted in step 3 or 4 leads to a *backward-planning* behavior of the algorithm: operators are chosen that produce unresolved preconditions, and their preconditions, in turn, may lead to new unresolved preconditions, or subgoals. This contrasts with the forward-planning strategy that is natural for situation-space planning.

Note further that the algorithm, by definition, supports a simple form of *incremental* planning, that is, the strategy of starting planning from an existing plan that is deficient in some respect. In the case of the CPP algorithm, the only deficits possible are unresolved preconditions; other variants of planning and other planning algorithms generalize incrementality. This feature is often useful for a planner in applications if the real-world problem to be solved changes frequently, but not so much that a fresh planning pass is necessary.

Enhancing Efficiency

Making CPP or similar basic planning algorithms efficient for a given problem means constraining its search appropriately. Effort pays back that is invested in choosing deliberately among the ways to resolve an unresolved condition and in choosing one among the possibly many operators in the recent plan to work on that have some of their preconditions unresolved. Many algorithmic or heuristic variants of CPP-style planning have been described. Many of these ideas are addressed by Yang (3, Part I).

More recently, planning algorithms have been described that move away from plan-space planning towards the situation space and that have been shown to outperform by orders of magnitude standard plan-space planners such as UCPOP (11) on many examples. Two archetypes of these algorithms, GRAPHPLAN (9) and SATPLAN (12), will be sketched here.

The naive version of situation-space planning as depicted in Fig. 3 would traverse this space starting from the initial situation by applying operators to situations that have already been generated, until a goal situation is found. The new algorithms are different in that they allow one to make leaps in the state space that correspond to applying sets of *compatible* operators, that is, operators that can be executed in either order without affecting the overall result. Note that this leads to nonlinear plans of the structure $O_1, O_2, \ldots, O_n$ where every *time step* O_i is a set of operators that can be executed in any order, and all operators of O_i must have been executed before execution of O_{i+1} starts. In consequence, the situation immediately after a time step is defined and unique, whereas the situations that occur within a time step depend on the concrete execution order—just as in any nonlinear plan.

GRAPHPLAN and SATPLAN differ in the way that the time steps are generated, and SATPLAN is more general in that it can mimic GRAPHPLAN's procedure. SATPLAN's basic idea is to describe in terms of logical formulas constraints that are true about individual states of the domain as well as constraints that must be true about transitions between states. The latter point is similar to formulating operators in logic as in Eq. (2) above; however, as the point is describing state transitions rather than operators, it turns out that straightforward formalizations can be found that need no frame axioms. If x, i are variables standing for a block and a time step, respectively, then an example for a state constraint axiom is the one expressing that at most one block may be held at a time:

$$\forall x, i.[\mathsf{Hand}(x, i) \rightarrow \neg\exists y \neq x.\mathsf{Hand}(y, i)]$$

```
Input:   $\Pi = \langle O, \prec \rangle$: plan
Output:  plan
------------------------------------------------------------
do forever
0. if $\prec$ is inconsistent then return fail;
1. if $\Pi$ contains an operator $o$ with an unresolved precondition $c$
2. then if   $c$ unresolved by conflict between $d \in O$ and dependency $\langle p, c, o \rangle$
3.       then choose one of
             Promote: $\Pi := (O, \prec \cup\{(d, p)\})$
             Demote: $\Pi := (O, \prec \cup\{(o, d)\})$
             Employ: Choose a $c$-producer $p' \in O$ that is unordered wrt. $d$ and $p \neq p'$;
                 $\Pi := (O, \prec \cup\{(d, p'), (p', o)\})$
             Insert: Choose a $c$-producer $p' \notin O$;
                 $\Pi := (O \cup \{p'\}, \prec \cup\{(d, p'), (p', o)\})$
4.       else choose a $c$-producer $p'$ by one of
             Employ: $p' \in O, p' \not\prec o$;
                 $\Pi := (O, \prec \cup\{(p', o)\})$
             Insert: $p' \notin O$;
                 $\Pi := (O \cup \{p'\}, \prec \cup\{(p', o), (\mathscr{I}, p')\})$
5. else return $\Pi$
```

Figure 4. CPP, a nondeterministic algorithm for classical propositional planning.

Here is an example axiom for a state transition, stating that a block x that is on y in i can only be still on y or held in the next time step $i + 1$:

$$\forall x, y, i.[\mathsf{On}(x, y, i) \rightarrow [\mathsf{On}(x, y, i+1) \vee \mathsf{Hand}(x, i+1)]]$$

Using appropriate normalizations of these formulas and very efficient—for some documented problems, stochastic—algorithms for constructing ground models for first-order logical formulas, SATPLAN "guesses" a consistent sequence of situations from the start to a goal situation, and then constructs locally the time steps between the successive situations.

GRAPHPLAN does not use a logical domain axiomatization, but develops and exploits a special data structure, the *planning graph.* This is a leveled, directed, acyclic graph with two types of nodes: proposition nodes and operator nodes. At the front of the graph is a level of proposition nodes with one node per proposition in the start situation. Then comes a level of operator nodes with one node per operator that is applicable in the proposition level before. Then comes the next proposition node level with one node per proposition that was added by an operator in the previous level, and so on. Three types of arcs connect appropriately preconditions to operators and operators to add/delete conditions. (For technical reasons, there is a special type of *no-op* operators that just copy single propositions from one proposition level to the next.)

Once a proposition level has been generated that includes the goal predicates, GRAPHPLAN tries to extract a sequence of compatible time steps from the planning graph. The exact definition of compatibility is purely technical; for example, it needs to be checked that the preconditions and delete conditions of operators within one time step are disjoint. If no such sequence can be found, the planning graph gets extended. The process is finite, as the set of propositions is finite and proposition levels grow strongly monotonically. In consequence, GRAPHPLAN—as well as SATPLAN—is guaranteed to terminate. Contrast this with CPP, which would run forever for unsolvable planning problems.

GRAPHPLAN and SATPLAN exploit better than CPP-style plan-space planners the structural constraints of propositional classical planning—hence their considerable, sometimes dramatic, saving in run time in many problems. Time steps are just a slight restriction of partial orders of operators in general, yet they make it possible to find very compact representations in propositional classical planning. On the other hand, it may be hard or even impossible to modify these algorithms appropriately in the process of moving the interpretation of planning to variants that relax the tight classical planning assumptions.

Enhancing Expressivity

When formulating domains for a planner, it is often inconvenient to use a purely propositional representation language. For example, one may wish to use variable objects in operators. In addition, more structured representations may help a planner plan faster. Finally, more expressive languages will become necessary as we turn to advanced planning techniques. All this motivates a tendency to enhance the expressivity of domain description languages and plan formats. This subsection sketches two orthogonal ways for doing this. Obviously, the planning algorithms have to be changed in reaction to these enhancements. Space does not permit us to give more than sketches of them; for details, see the original papers. More ways to enhance expressivity are summarized, for example, in Refs. 2 and 3.

Enriching the Operator Language: ADL. ADL (Action Description Language) (13) is a formalism that has been used in several planners, most prominently the planner UCPOP (11). ADL allows operators to be formulated using the following constructs:

Preconditions are—with slight restrictions—sets of first-order formulas. For example, a new blocks world operator **MOVE**(x, y) for moving block x from somewhere to location y, which may be either a different block or the table, might specify as preconditions

$$x \neq y, \qquad x \neq \text{TABLE}, \qquad \forall z.\neg\mathsf{On}(z, x),$$
$$[y = \text{TABLE} \vee \forall z.\neg\mathsf{On}(z, y)]$$

Add sets: Let P be a predicate symbol, $\bar{t}$ a fitting list of arguments, $z_1, \ldots, z_k$ variables appearing in $\bar{t}$, and Ψ a first-order formula. An *add set* consists of elements of the following forms:

- $P(\bar{t})$
- $P(\bar{t})$ if Ψ
- $P(\bar{t})$ for all $z_1, \ldots, z_k$
- $P(\bar{t})$ for all $z_1, \ldots, z_k$ such that Ψ

Here, as an example, is the add set of **MOVE**(x, y), where the predicate $\mathsf{Above}(x, y)$ means x is on y or on another block above y:

$$\mathsf{On}(x, y), \mathsf{Above}(x, y), \mathsf{Above}(x, z) \text{ for all } z \text{ such that } \mathsf{Above}(y, z)$$

Delete sets are of the same form like add sets. For example, the delete set of **MOVE**(x, y) is

$\mathsf{On}(x, z)$ for all z such that $z \neq y$

$\mathsf{Above}(x, z)$ for all z such that $[z \neq y \wedge \neg\mathsf{Above}(y, z)]$

Obviously, the simple situation update scheme using set difference and union [Eq. (1)] no longer works with ADL add sets and delete sets. Instead, the individual components of the add and delete sets are schematically transformed into special logical formulas, which are then used in reasoning about what is true or false in individual situations and what changes or remains inert between situations. Details are beyond the scope of this article, as is the handling of changes of continuous values by operators that ADL allows (13).

Introducing Layers of Description: Hierarchical Task Network Planning. As described until here, an operator has served as an atomic element of the domain representation under two aspects. First, it must be an action unit to give the plan execution agent, i.e., a "command" that the agent can interpret and execute without further advice: an operator is an *atom for execution.* Second, an operator is an *atom for description:* The domain modeler has to specify in terms of operators the change that can be effected by the agent.

It is unnecessary that these two aspects coincide, and it is often undesirable: First, domain models with "flat" operator inventories tend to be hard to understand; second, through domain knowledge, the domain modeler often knows standard nonatomic procedures to apply in certain situations that can be used as subplans and that the planner need not generate from scratch again and again. In consequence, the idea of using *virtual,* not directly executable operators in planning has appeared in several variants; depending on whether the modeling aspect or a possible speedup for planners is focused, similar ideas have been given different names such as abstraction, hierarchical decomposition, and macro operators. The recent literature most often uses the term *hierarchical task network* (HTN) for a plan containing or made up from the respective operators. Yang (3) gives a more comprehensive description.

To give a simple blocks world example, consider Fig. 5. **STACK3** is a virtual operator for stacking the three blocks x, y, z. The only new ingredient of the description is the *plot:* It specifies a plan consisting of a mixture of virtual and elementary operators that must be used to refine the operator. As can be seen in the example, the plot in itself need not be finished; obviously, the precondition Hand(x) of **STACK**(x, y) is not true in the plot.

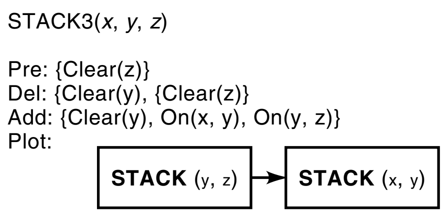
Virtual operators are to be used in planning in the following way: Whenever an open precondition is to be resolved, virtual operators can be inserted or employed just like elementary ones. Planning must continue, however, as long as the recent plan contains virtual operators. Such an operator must eventually get replaced by its plot; after that, planning proceeds by checking flaws that this replacement may have introduced.

A simple fix of the CPP algorithm (Fig. 4) is to replace its step 5 by

```
5. else if Π contains a virtual operator v
6. then replace v by its plot
7. else return Π
```

The resulting algorithm would expand virtual operators only after resolving all preconditions. This is an arguable strategy; other strategies may be useful, but would require a more complicated formulation of the algorithm. The same is true for operator selection: As virtual operators eventually lead to a larger expansion of the plan, it makes sense to insert them with special care—only if some other operator requires all or many of their postconditions. We do not go into these heuristic issues here, but keep in mind the general point: Making use of a more expressive domain language requires algorithms that make real use of the enhanced expressivity.

This leads to the question of how much is or may be gained by using virtual operators. The answer to this is twofold. First, as an empirical observation, practically all successful application planners (see the appropriate section below) are using them in one way or the other; the reason is that they may help enhance both planning performance and ease of domain description, as stated above. Second, virtual operators can make matters worse, as they are yet more operators, which blow up the search space.



STACK3(x, y, z)

Pre: {Clear(z)}
Del: {Clear(y), {Clear(z)}
Add: {Clear(y), On(x, y), On(y, z)}
Plot:

Figure 5. Schema of a virtual operator for stacking three blocks. A more detailed version of the plot could also specify which of the operators require or generate which of **STACK3**'s preconditions or postconditions, respectively.

To describe this more precisely, note that the following two properties are intuitively expected of an HTN plan:

Downward Solvability. If Π is a plan with all preconditions resolved and containing a virtual operator v, then expanding v eventually leads to a finished plan Π′.

Upward Solvability. If a planning problem has a solution Π consisting of elementary operators, then there is an abstract plan Π′ with all preconditions resolved that contains a virtual operator v, and expanding v eventually leads to Π.

The theoretical problem with HTN planning is that neither of these properties holds in general. In consequence, effort may in theory be expended in vain on expanding an abstract solution that actually has no elementary refinement (no downward solution) or on trying to construct first a nonexistent abstract solution for a problem that has an elementary one (no upward solution). Practical application domains often allow sharp criteria to be formulated that lead to a highly selective operator choice. In consequence, the general lack of upward and downward solvability need not be practically relevant.

ADVANCED PLANNING

When designing algorithms, generality is both a virtue and a burden. It is a virtue in that a more general algorithm allows more problems to be tackled. It is a burden in that a more general algorithm has less structural clues to exploit and is therefore likely to be less efficient. That is true in particular for planning algorithms. Whenever it makes sense or is tolerable as an approximation for a planning domain to make the simplifying assumptions described for classical planning, they should be made and a corresponding algorithm chosen.

Sometimes, though, it is not tolerable. Since its early days, AI planning has included work on *nonclassical* planning, as it is often called now, that is, on planning that allows some of the classical assumptions to be relaxed. This section gives a brief introduction into a few different conceptions of planning, centering on three topics: richer models of time, handling uncertainty, and reactivity.

Variants of nonclassical planning differ in that they cope with different basic assumptions. In consequence, the respective techniques are mostly different and orthogonal. Therefore, comprehensive survey texts can hardly be expected to describe nonclassical planning as a whole; they normally focus on coherent parts of it. Stressing the connections between planning and control theory, Dean and Wellman (14) deal with the topics of time and uncertainty; centering on planning for autonomous mobile robots, McDermott (15) touches upon uncertainty and reactivity.

Time

The strobelike time model of classical planning abstracts away from two main aspects of time that may be important for the question of how to act in the world: quantization (for how long is a fact valid?) and intervals (which actions, external events, or facts overlap?). Planning methods exist for handling time in both aspects individually and in combination.

Adding Duration to Classical Plans. A basic—but sometimes sufficient—way to inject numerical time information is to consider a basic time unit, associate situations with discrete "clock ticks" in terms of this time unit, and specify durations of operators and facts in terms of the resulting *system time.* As this time model still has discrete situations as its ontological time primitive, it can (at least conceptually) be merged into classical planning in a straightforward way by appropriately splitting up the original situations into intermediate ones.

The basic idea is that a duration is specified for an operator in terms of time units, that a time must be specified for each and every precondition until which it must be true, counted from the operator start, and a time must be specified for each and every add/delete condition when it begins/ceases to be true. For example, let the **STACK**(x, y) operator have a duration of 4 units; then its precondition may specify that Hand(x) is required to be true at least until and including 2 units after the operator starts; we use Hand(x)@2 as the notation in preconditions. In the add list, assume the hand gets free at time 3, that is, Hand(NIL)@3; at the same time, x is no longer being held, i.e., Hand(x)@3 is in the delete set. (Note the interpretation difference of the @ sign in preconditions and postconditions.) Assuming **STACK**(A, B) gets scheduled for absolute system time 4711, that means Hand(A) must be true (and will be true, once the plan is finished) until 4713 and ceases to be true from 4714, at which time Hand(NIL) becomes true.

As a practical variation of this scheme, imprecision of knowledge about exact execution times and holding periods may be handled by providing lower and upper bounds of the respective values. For example, it may be specified for the **STACK**(x, y) operator that Hand(x) starts to be true no earlier than 3 and no later than 4 units after the operator start—or Hand(x)@[3, 4] to use a standard notation. Vere's (16) planner DEVISER has used this type of information. Moreover, it can handle *scheduled events,* that is, events that are known to happen and change certain facts without further action at absolute time points, such as sunrise or shop closing hours. IXTET by Ghallab and coworkers (17) is a temporal planner that integrates the planning process into a more general view of temporal reasoning.

Handling numerical time information in the way just sketched allows a planner to generate plans that not only specify a feasible order of actions, but also make a schedule that specifies exactly when to execute some action. Moreover, goals can be given deadlines or durations, and both the sequential plan and the schedule can be generated to meet them. This planning-plus-scheduling functionality is attractive for a large number of applications in manufacturing or logistics. Some examples will be given in the section below on planning applications.

Reasoning about Time Intervals. A different type of temporal reasoning comes into play if time *intervals* are considered—be it that they are assumed to be the ontological "time primitives" or that they are defined by their border time points. The new feature is that concurrency or overlap of operators may have special effects.

In the previous discrete numerical time model, operators may in fact be scheduled for execution in parallel by setting their execution times appropriately; roughly speaking, that may be done for operators and plans whose dependency structure is such that the operators are unordered in a classical plan. When considering time intervals, it can be expressed (and indeed, must be handled) that executing two operators yields different effects depending on whether they are executed sequentially or concurrently. For example, the effect of pressing the ⟨SHIFT⟩ key and the ⟨A⟩ key on a computer keyboard depends on whether and how these two actions overlap in time.

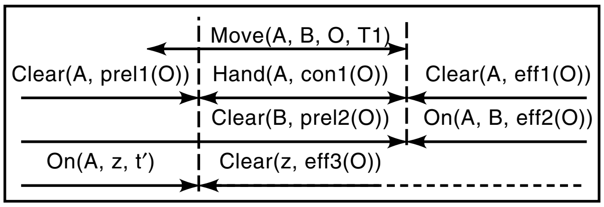
A variety of approaches exists for reasoning about this type of temporal information; many of them are variants or specializations of modal temporal logic. Sandewall (18) gives a comprehensive overview. Only few of them have been used in planning until now. The pragmatic reason is that the reasoning about time that is possible with these powerful formalisms is likely to take too much time itself. Moreover, it requires considerable effort to design the appropriate descriptions for domains of more than toy size.

An example for an interval-based planner is Allen's (19) ILP, which is formally based on his work on reasoning about time intervals using a relational algebra (20). Figure 6 gives an example of how to represent the **MOVE** operator. The predicates Finishes, Meets, Overlaps, and SameEnd are generic predicates for time intervals the reasoning with which is described in Ref. 20; the Clear, Hand, and On predicates are parts of the blocks world domain language as before, whose last arguments specify time intervals over which the respective facts are valid; Move(x, y, o, t) represents the fact that a **MOVE** operator instance o with arguments x, y is executed over the time interval t. The syntactic function move(x, y) is the representation of the actual operator as it appears in an ILP plan, and the Try predicate represents its application over the respective time interval.

More axioms are necessary for formalizing the domain, the temporal predicates, and their interplay. Details are out of the scope of this text; to give an idea of the information required, here is an example axiomatizing the structure of intervals that meet each other:

$$\begin{aligned}\forall r, s, t, u.[&\mathrm{Meets}(r, s) \wedge \mathrm{Meets}(s, t) \wedge \mathrm{Meets}(t, u)\\ &\rightarrow \exists t'.[\mathrm{Meets}(r, t') \wedge \mathrm{Meets}(t', u)]]\end{aligned}$$

Planning now means to find a consistent structure of operator and predicate intervals such that the conjunction of goal conditions is entailed for some interval "at the end." Much of Allen's ILP algorithm can be described in analogy to CPP-like classical planning: The analog of dependency is the matching of two postcondition and precondition intervals of different operators, where the intervals are labeled with identical—or rather, unifiable—propositions; the analog of conflict is the overlap of two intervals that are labeled with propositions that are inconsistent under the domain theory; the analog of



Interval structure:
∀ o ∃x, y, t.
[Move(x, y, o, t) Æ
Overlaps(pre1(o), t) ∧ Finishes(con1(o), t) ∧ Meets(pre1(o), con1(o))^
Meets(t, eff1(o)) ∧ SameEnd(t, pre2(o)) ∧ Meets(t, eff2(o))]

Necessary facts:
∀ x, y, t.
[Move(x, y, o, t) Æ
Clear(pre1(o))   Clear(y, pre2(o)) ∧ Hand(x, con1(o)) ∧
Clear(x, eff1(o)) ∧ On(x, y,, eff2(o))]

Effects on previous x locations:
∀ x, y, z, o, t, t′.
[Move(x, y, o, t) ∧ On(x, z, t′) ∧ Overlaps(t′, t) Æ
Clear(z,eff3(o)) ∧ Meets(t′, eff3(o)) ∧ Meets(t′, con1(o))]

Sufficient execution conditions:
∀ x, y, z, o, t∃t′, t″.
[Try(move(x, y), o, t) ∧ Clear(y, t′) ∧ Overlaps(t′, t) ∧
Clear(y, t″) ∧ SameEnd(t, t″) Æ
Move(x, y, o, t)]

Figure 6. Axiomatization of the **MOVE** operator as for the ILP planner. A graphical representation of the interval structure for the operator O = **MOVE**(A, B) as executed in time interval T1 is given on top. (Adapted from Ref. 19, p. 25, Fig. 13.)

operator insertion is the addition of a set of axiom instances describing an operator (like the ones in Fig. 6).

Coming back to the motivation of interval-based temporal planning, it is possible to make operator effects conditional on facts that hold or cease to hold during its execution. For example, it can easily be expressed that pressing the ⟨A⟩ key on a computer keyboard yields a capital A if it is done During the execution of the operator of pressing ⟨SHIFT⟩, and yields a lowercase A else. However, the technical apparatus needed to achieve this expressivity is considerable.

Uncertainty

Most real-world application domains involve some degree and some form of uncertainty: Knowledge about the initial conditions may be incomplete and possibly inaccurate; actions may be known to fail sometimes; actions may work differently under different conditions. Pragmatically, there are three ways to approach this. If the uncertainty is too large, then there is no point in planning; more information is needed first, or, if tolerable, one may act according to some given scheme. If the uncertainty is sufficiently small or irrelevant, it is acceptable to ignore it and use the planning techniques described previously. In all other cases, the uncertainty needs to be represented and addressed in planning. As there are many different aspects of uncertainty for planning and different ways to represent and process it, there is a large variety of approaches to planning under uncertainty. Reference 4, Part V gives a comprehensive introduction.

Compared to the classical planning framework, planning under uncertainty typically uses the following additional ingredients:

- A probability distribution over situations, representing uncertainty about the initial state.
- A generalized operator schema that allows one to specify different effects for different execution contexts and different possible effects within one execution context; so the operator format [*Pre*|*Post*] of classical planning stating preconditions *Pre* and postconditions *Post* (e.g., in terms of added and deleted facts) changes to

$$\begin{array}{lll} [Pre_1 & | & Post_{1,1}, \ldots, Post_{1,l(1)} \\ \vdots & & \\ Pre_m & | & Post_{m,1}, \ldots, Post_{m,l(m)}] \end{array}$$

  where the Pre_i denote different execution contexts, and the $Post_{i,j}$ are different, exclusive sets of effect descriptions, typically labeled with probability information stating how likely the respective outcome is. In consequence, an operator maps a probability distribution over situations into another such probability distribution.
- Information about the utility of states, features of states, and/or action applications. As usual, negative utility can be interpreted as cost.

It is natural, then, to think of planning as a Markovian decision process (MDP) or a partially observable MDP (21) as originally introduced in operations research. A plan in this view is a structure that maps a probability distribution over situations to an action, where it is desirable that this action maximize the expected utility; plans of this sort are commonly called *policies.* Different maximization strategies are possible, depending on whether immediate or long-term expected benefit is to be favored. Long-term expected benefit is a natural quality criterion for planning, but within tolerable computation times and for realistic state spaces, it can at best be approximated.

As the information about the respective recent world state is incomplete, it makes sense to consider *sensor actions* in plans/policies or in their execution for disambiguating situations. They are not intended to bring about changes in the world, but to change the plan-executing agent's knowledge about the current state by reducing entropy in the recent probability distribution. Sensor actions play an important practical role in control of autonomous robots; they cannot be modeled adequately in classical planning (see, for example, Ref. 42).

Reactive Planning and Situatedness

Designing methods for helping autonomous agents—such as mobile robots or software agents—act purposefully has always been one of the goals of AI planning research. A line of work in this very area has led to fundamental criticism of the use of representations in mainstream AI in general and towards "deliberative" planning in particular: *behavior-based* agent architectures (22) and *situated action* (23).

The heart of the criticism is this: In designing autonomous agents a number of serious technical and fundamental problems arise if action is based on generating and executing plans in one of the senses described above; moreover, and luckily, it is not necessary to do so, but there is an alternative: situated action. One of the technical problems is that planning with either of the methods described takes time, which is often nonnegligible, but an agent in a dynamic environment must be prepared to act—or react, for that matter—purposefully at any time without calling its planner module and waiting for the output first. One of the fundamental problems is that representation-based planning presumes it is possible and practical to "ground" the symbols in perceptions in the sense that an effective translation exists between the sensor input stream of the agent and a symbolic (e.g., first-order logic) domain representation.

To understand this criticism well, let us briefly state some cases of deliberative planning to which it obviously does not apply. First, not every car manufacturer wishes to generate or change its job shop schedules in milliseconds, so there are planning applications without close reactivity deadlines. Second, the symbol-grounding problem as such need not be solved to design autonomous mobile robots for particular applications in which it is possible to monitor directly the truth or falsity of the crucial state features. Third, in all cases where plans are generated for humans to interpret and execute, we can rely upon their symbol-handling capabilities.

The argument applies in one part to *cognitive* AI research, that is, to that line of AI work which is concerned with modeling and understanding intelligent behavior in general, or with "achieving artificial intelligence through building robots," in Brooks's terms. Among the various AI researchers, it is far from generally accepted, by the way; see, for example, the debate in Ref. 24.

For some application fields, such as control of autonomous mobile robots, reactivity and sensor interpretation are obvious issues, and work along the lines of behavior-based control and situated action has helped shape the understanding of planning and of the uses of plans. Summing up constraints for a general robot control architecture, McDermott (15, p. 76) states two points as mandatory, among others:

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." 3T (25) is an example for such a robot control architecture integrating deliberative planning, plan execution, and a reactive layer; it has been demonstrated to work for a number of different application areas.

Plan at Any Time. "Make use of fast interruptible algorithms. . . . When the planner finds a better plan, swap it in." A class of algorithms allowing for this type of behavior are *anytime algorithms,* or, more generally, algorithms that allow for *deliberation scheduling* (26), that is, for explicit control of their computation under time constraints.

A different line of work argues for generating (or even hand-coding) plans for the most likely problems off line before and applying them in reaction to the situational patterns that the agent encounters. The Procedural Reasoning System (PRS) (27) has been influential in this direction; policies in planning under uncertainty as sketched above can also be understood in that way.

APPLICATIONS

Applications of AI planning are as diverse as suggested by its definition "finding in advance some course of action" in the introduction of this article. A recent collection (28) features five application systems that are in use or on the way toward commercial products and employ planning techniques in the sense described above for the following problems: declarer play in contract bridge; reaction to marine oil spills; project management in spacecraft assembly, integration, and verification; operating communication antennas; and military air campaign planning. In addition to such systems that explicitly build upon the generic planning methods as described here, there have always been systems specially designed for special applications; an early example for such a system, which has influenced the development of generic planning methods, was Stefik's MOLGEN (29), a knowledge-based system for designing experiments in molecular genetics.

A push for transferring planning methods and software prototype systems into real-world applications has resulted from the DARPA Planning and Scheduling Research Initiative, which has been in effect since 1991; Ref. 30 is a collection of papers from this context. Economically, the initiative was a definite success, judging from a report by the US Department of Commerce, saying that (quoted from Ref. 30, p. vii)

> the deployment of a single logistics support aid called DART during the Desert Shield/Desert Storm Campaign paid back all US government investments on AI/KBS research over a 30 year period.

Much of the application success of the ARPI initiative is owed to two powerful generic planning systems that are based upon the methods described earlier: SIPE-2, on which the DART system was built, and O-Plan (see Refs. 31 and 32, respectively, for comprehensive descriptions of their basics).

Among current industrial applications of AI planning technology, logistics planning and integrated planning and scheduling stand out as application classes; see Ref. 33 for a collection of papers. If recent market estimations turn out to be correct and service robots have the market potential that is currently suspected, then another broad field for industrial application of planning technology lies ahead, as planning is unavoidable for high-level task and mission control of autonomous mobile robots; McDermott (15) gives an overview of the planning issues that are involved. Latombe (43) reviews comprehensively the fields of path planning and motion planning, which are essential ingredients for mobile robot control, but are normally based on special-purpose algorithms. Finally, as software agents ("softbots") in the World Wide Web become practical, so does the planning capability that they require; see, for example, Ref. 44.

HISTORICAL AND BIBLIOGRAPHICAL REMARKS

Much of the research in AI planning was originally motivated by models of problem solving from cognitive psychology; most influential was the work on the *General Problem Solver* (GPS) by Newell, Simon, and co-workers (34). Another root of AI planning—in particular for deductive planning—is work on automatic program generation from input/output specifications; Green's (7) work on deductive planning is an example. Both these lines of work have motivated and influenced the design of STRIPS (1), which shaped work on classical planning for a long time. In today's terms, the original STRIPS was to some extent a nonclassical planner. It was part of the control system of the mobile robot SHAKEY (35), which motivated exploring in the context of planning approaches to domain representation, learning, execution control, and embedded planning; other research tackled these problems only considerably later.

Soon after the publication of STRIPS the field of classical planning unfolded, with nonlinear plans (36,37) and HTN planning (36) as prominent topics. A paper by Chapman (6) was influential in stressing the need for and presenting first results in formal descriptions and theoretical investigations of classical planning. For a number of years, the planner UCPOP (11) has been a reference system for classical plan-space planning, with systems of the GRAPHPLAN (9) and SATPLAN (12) families yielding performance benchmarks at the time of writing this article.

Work on nonclassical planning variants has developed in parallel, with many of the issues already addressed in early papers. Vere (16) and Allen (19) have largely influenced temporal planning. Feldman and Sproull (38) have given an early formulation of planning under uncertainty based on techniques from decision theory. Based on a decision-theoretic planning framework, Dean and Boddy (26) have developed the notion of *anytime* planning, that is, planning that is able to respect deadlines for plan delivery. Agre and Chapman (23) have made an influential argument for situated action rather than deliberative planning.

Early application systems in AI planning are better described in terms of knowledge-based systems than in terms of the notions and algorithms presented in this article; an example is Stefik's MOLGEN (29). Reference 28 features current application systems based on the generic planning methods described here. The first and most prominent generic planners that have allowed application systems to be built were SIPE (31) and O-Plan (32); both systems still exist in enhanced versions.

Reference 39 is a collection of classical papers. Weld (2) and Yang (3) give comprehensive introductions into planning, both with an emphasis on classical planning. Introductions are also contained in typical AI textbooks: Russell and Norvig (4) present planning comprehensively. Reference 30 is a collection of recent application-oriented papers.

Recent planning research is regularly presented in two biannual conferences, namely the International Conference on AI Planning Systems (AIPS) and the European Conference on Planning (ECP). The most recent proceedings volumes at the time of writing are Refs. 40 and 41.

BIBLIOGRAPHY

1. R. E. Fikes and N. J. Nilsson, STRIPS: A new approach to theorem proving in problem solving, *Artif. Intell.,* **2** (3–4): 189–208, 1971.
2. D. S. Weld, An introduction to least commitment planning, *AI Mag.,* **15** (4): 27–61, 1994.
3. Q. Yang, *Intelligent Planning. A Decomposition and Abstraction Based Approach,* Berlin: Springer, 1997.
4. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NJ: Prentice-Hall, 1995.
5. T. Bylander, The computational complexity of propositional STRIPS planning, *Artif. Intell.,* **69** (1–2): 165–204, 1994.
6. D. Chapman, Planning for conjunctive goals, *Artif. Intell.,* **32** (3): 333–377, 1987.
7. C. Green, Application of theorem proving to problem solving, *Proc. IJCAI-69,* San Mateo, CA: Morgan Kaufmann, 1969, pp. 219–239.
8. W. Bibel, Let's plan it deductively! *Proc. IJCAI-97,* 1997, pp. 1549–1562.
9. A. L. Blum and M. L. Furst, Fast planning through plan graph analysis, *Artif. Intell.,* **90**: 281–300, 1997.
10. D. McAllester and D. Rosenblitt, Systematic nonlinear planning, *Proc. AAAI-91,* 1991, pp. 634–639.
11. J. S. Penberthy and D. Weld, UCPOP: A sound, complete, partial order planner for ADL, *Proc. 3rd Int. Conf. Princ. Know. Represent. (KR-92),* San Mateo, CA: Morgan Kaufmann, 1992, pp. 103–114.
12. H. Kautz and B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, *Proc. AAAI-96,* 1996, pp. 1194–1201.
13. E. P. D. Pednault, ADL: Exploring the middle ground between STRIPS and the situation calculus, *Proc. Int. Conf. Prin. Knowl. Represent. (KR-89),* 1989, pp. 324–332.
14. T. L. Dean and M. P. Wellman, *Planning and Control,* San Mateo, CA: Morgan Kaufmann, 1991.
15. D. McDermott, Robot planning, *AI Mag.,* **13** (2): 55–79, 1992.
16. S. A. Vere, Planning in time: Windows and durations for activities and goals, *IEEE Trans. Pattern Anal. Mach. Intell.,* **PAMI-5**: 246–267, 1983.
17. M. Ghallab and H. Laruelle, Representation and control in IXTET, a temporal planner, *2nd Int. Conf. Artif. Intell. Planning Syst. (AIPS-94),* 1994, pp. 61–67.
18. E. Sandewall, *Features and Fluents. A Systematic Approach to the Representation of Knowledge about Dynamical Systems,* London: Oxford Univ. Press, 1994.
19. J. Allen, Temporal reasoning and planning, in J. Allen et al. (eds.), *Reasoning about Plans,* San Mateo, CA: Morgan Kaufmann, 1991, Chap. 1, pp. 1–68.

20. J. F. Allen, Maintaining knowledge about temporal intervals, *Commun. ACM,* **26**: 832–843, 1983.
21. A. Cassandra, L. Pack Kaelbling, and M. Littman, Acting optimally in partially observable stochastic domains, *Proc. AAAI-94,* Menlo Park, CA: AAAI Press, 1994, pp. 1023–1028.
22. R. Brooks, Intelligence without representation, *Artif. Intell.,* **47** (1–3): 139–159, 1991.
23. P. Agre and D. Chapman, Pengi: An implementation of a theory of action. *Proc. AAAI-87,* San Mateo, CA: Morgan Kaufmann, 1987, pp. 268–272.
24. Special issue: Situated action, *Cogn. Sci.,* **17** (1): 1993.
25. P. Bonasso et al., Experiences with an architecture for intelligent, reactive agents, *J. Exp. Theor. Artif. Intell.,* **9**: 237–256, 1997.
26. M. Boddy and T. Dean, Deliberation scheduling for problem solving in time-constrained environments, *Artif. Intell.,* **67** (2): 245–285, 1994.
27. M. P. Georgeff and A. L. Lansky, Reactive reasoning and planning, *Proc. AAAI-87,* San Mateo, CA: Morgan Kaufmann, 1987, pp. 677–682.
28. C. A. Knoblock (ed.), AI planning systems in the real world, *IEEE Expert,* **11** (6): 4–12, 1996.
29. M. Stefik, Planning with constraints (MOLGEN: Part 1); Planning and meta planning (MOLGEN: Part 2), *Artif. Intell.,* **16** (2): 111–170, 1981.
30. A. Tate (ed.), *Advanced Planning Technology. Technological Achievements of the ARPA / Rome Laboratory Planning Initiative,* New York: AAAI Press, 1996.
31. D. Wilkins, *Practical Planning. Extending the Classical AI Planning Paradigm,* San Mateo, CA: Morgan Kaufmann, 1988.
32. K. Currie and A. Tate, O-plan: The open planning architecture, *Artif. Intell.,* **52** (1): 49–86, 1991.
33. M. Zweben and M. S. Fox (eds.), *Intelligent Scheduling,* San Francisco: Morgan Kaufmann, 1994.
34. G. W. Ernst and A. Newell, *GPS: A Case Study in Generality and Problem Solving,* New York: Academic Press, 1969.
35. N. J. Nilsson, *Shakey the Robot,* Tech. Rep. TN 323, Stanford, CA: SRI International, 1984.
36. E. D. Sacerdoti, *A Structure for Plans and Behavior,* Amsterdam: Elsevier/North-Holland, 1977.
37. A. Tate, Generating project networks, *Proc. IJCAI-77,* San Mateo, CA: Morgan Kaufmann, 1977, pp. 888–893.
38. J. A. Feldman and R. F. Sproull, Decision theory and artificial intelligence II: The hungry monkey, *Cogn. Sci.,* **1**: 158, 1977.
39. J. Allen, J. Hendler, and A. Tate (eds.), *Readings in Planning,* San Mateo, CA: Morgan Kaufmann, 1990.
40. R. Simmons, M. Veloso, and S. Smith (eds.), *AIPS-98, Proc. 4th Int. Conf. Artif. Intell. Planning Syst.,* Menlo Park: AAAI Press, 1998.
41. S. Steel and R. Alami (eds.), *Recent Advances in AI Planning, 4th Eur. Conf. Plann., ECP-97.* (LNAI, Vol. 1348), New York: Springer, 1997.
42. K. Golden and D. Weld, Representing sensing actions: The middle ground revisited, *Proc. 5th Int. Conf. Princ. Know. Represent. Reasoning (KR-96),* San Mateo, CA: Morgan Kaufmann, 1996, pp. 174–185.
43. J.-C. Latombe, *Robot Motion Planning,* Dordrecht: Kluwer, 1991.
44. O. Etzioni, Intelligence without robots: A reply to Brooks, *AI Magazine,* **14** (4): 7–13, 1993.

JOACHIM HERTZBERG
GMD—German National Research
Center for Information
Technology