Planning and Scheduling

Thomas Dean, Brown University
Subbarao Kambhampati, Arizona State University

1 Introduction

In this chapter, we use the generic term planning to encompass both planning and scheduling problems, and the terms planner or planning system to refer to software for planning or scheduling. Planning is concerned with reasoning about the consequences of acting in order to choose from among a set of possible courses of action. In the simplest case, a planner might enumerate a set of possible courses of action, consider their consequences in turn, and choose one particular course of action that satisfies a given set of requirements.

Algorithmically, a planning problem has as input a set of possible courses of actions, a predictive model for the underlying dynamics, and a performance measure for evaluating courses of action. The output or solution to a planning problem is one or more courses of action that satisfy the specified requirements for performance. Most planning problems are combinatorial in the sense that the number of possible courses of actions or the time required to evaluate a given course of action is exponential in the description of the problem.

Just because there is an exponential number of possible courses of action does not imply that a planner has to enumerate them all in order to find a solution. However, many planning problems can be shown to be NP-hard, and, for these problems, all known exact algorithms take exponential time in the worst case. The computational complexity of planning problems often leads practitioners to consider approximations, computation time versus solution quality tradeoffs, and heuristic methods.

1.1 Planning and Scheduling Problems

We use the travel planning problem as our canonical example of planning (distinct from scheduling). A *travel planning problem* consists of a set of travel options (airline flights, cabs, subways, rental cars, and shuttle services), travel dynamics (information concerning travel times and costs and how time and cost are affected by weather or other factors), and a set of requirements. The requirements for a travel planning problem include an itinerary (be in Providence on Monday and Tuesday, and in Phoenix from Wednesday morning until noon on Friday) and constraints on solutions (leave home no earlier than the Sunday before, arrive back no later than the Saturday after, and spend no more than one thousand dollars in travel-related costs). Planning can be cast either in terms of *satisficing* (find some solution satisfying the constraints) or *optimizing* (find the least cost solution satisfying the constraints).

We use the job-shop scheduling problem as our canonical example of scheduling (distinct from planning). The specification of a *job-shop scheduling problem* includes a set of jobs, where each job is a partially-ordered set of tasks of specified duration, and a set of machines, where each machine is capable of carrying out a subset of the set of all tasks. A *feasible solution* to a job-shop scheduling problem is a mapping from tasks to machines over specific intervals of time, so that no machine has assigned to it more than one task at a time and each task is completed before starting any other task that follows it in the specified partial order. Scheduling can also be cast in terms of either satisficing (find a feasible solution) or optimizing (find a solution that minimizes the total time required to complete all jobs).

1.2 Distinctions and Disciplines

To distinguish between planning and scheduling, we note that scheduling is primarily concerned with figuring out <u>when</u> to carry out actions while planning is concerned with <u>what</u> actions need to be carried out. In practice, this distinction often blurs and many real-world problems involve figuring out both what and when.

In real job-shops, each task need not specify a rigid sequence of steps (the <u>what</u>). For example, drilling a hole in a casting may be accomplished more quickly if a positioning device, called a fixture, is installed on the machine used for drilling. However, the fixture takes time to install and may interfere with subsequent machining operations for other tasks. Installing a fixture for one task may either expedite (the next job needs the same fixture) or retard (the fixture is in the way in the next job) subsequent jobs. In this version of our canonical scheduling problem, planning can take on considerable importance.

We can, however, design a problem so as to emphasize either planning or scheduling. For example, it may be reasonable to let a human decide the <u>what</u> (*e.g.,* the type of machine and specific sequence of machining steps for each task) and a computer program decide the <u>when</u>

(*e.g.,* the time and machine for each task). This division of labor has allowed the field of operations research to focus effort on solving problems that stress scheduling and finesse planning, as we distinguished the two above. Restricting attention to scheduling has the effect of limiting the options available to the planner, thereby limiting the possible interactions among actions and simplifying the combinatorics. In addition, some scheduling problems do not allow for the possibility of events outside the direct control of the planner, so-called *exogenous* events, Planning researchers in artificial intelligence generally allow a wide range of options (specifying both <u>what</u> and <u>when</u>) resulting in a very rich set of interactions among the individual actions in a given course of action and between actions and exogenous events.

The travel planning problem includes as a special case the classic traveling salesperson problem, a problem of considerable interest in operations research. In the traveling salesperson problem, there is a completely connected graph with L vertices corresponding to L distinct cities, an $L \times L$ matrix whose entries encode the distance between each pair of cities, and the objective is to find a minimal-length tour of a specified subset of the cities. The classic traveling salesperson problem involves a very limited set of possible interactions (*e.g.,* you must finish one leg of a tour before beginning the next, and the next leg of a tour must begin at the city in which the previous leg ended). In contrast, variants of the travel planning problem studied in artificial intelligence generally consider a much richer set of possible interactions (*e.g.,* if you start on a multi-leg air trip it is generally more cost effective to continue with the same airline, travel that extends over a Saturday is less expensive than travel that does not).

Planning of the sort studied in artificial intelligence is similar in some respects to problems studied in a variety of other disciplines. We have already mentioned operations research; planning is also similar to the problem of synthesizing controllers in control theory or the problem of constructing decision procedures in various decision sciences. Planning problems of the sort considered in this chapter differ from those studied in other disciplines mainly in the details of their formulation. Planning problems studied in artificial intelligence typically involve very complex dynamics, requiring expressive languages for their representation, and encoding a wide range of knowledge, often symbolic, but invariably rich and multi-faceted.

2 Classifying Planning Problems

In this section, we categorize different planning problems according to their inputs: the set of basic courses of action, the underlying dynamics, and the performance measure. We begin by considering models used to predict the consequences of action.

2.1 Representing Dynamical Systems

We refer to the environment in which actions are carried out as a *dynamical system.* A description of the environment at an instant of time is called the *state* of the system. We assume that there is a finite, but large, set of states, $\mathcal{S}$, and a finite set of actions, $\mathcal{A}$, that can be executed.

States are described by a vector of *state variables*, where each state variable represents some aspect of the environment that can change over time (*e.g.,* the location or color of an object). The resulting dynamical system can be described as a deterministic, nondeterministic, or stochastic finite state machine, and time is isomorphic to the integers. In the case of a deterministic finite state machine, the dynamical system is defined by a *state-transition function* f that takes a state $s_t \in \mathcal{S}$ and an action $a_t \in \mathcal{A}$ and returns the next state $f(s_t, a_t) = s_{t+1} \in \mathcal{S}$.

If there are N state variables each of which can take on two or more possible values, then there are as many as 2^N states and the state-transition function is N-dimensional. We generally assume each state variable at t depends on only a small number (at most M) of state variables at $t-1$. This assumption enables us to *factor* the state-transition function f into N functions, each of dimension at most M, so that $f(s,a) = \langle g_1(s,a), \ldots, g_N(s,a)\rangle$ where $g_i(s,a)$ represents the ith state variable.

In most planning problems, a plan is constructed at one time and executed at a later time. The state-transition function models the evolution of the state of the dynamical system as a consequence of actions carried out by a *plan executor.* We also want to model the information available to the plan executor. The plan executor may be able to observe the state of the dynamical system, partial state information corrupted by noise, or only the current time. We assume that there is a set of possible observations $\mathcal{O}$ and the information available to the plan executor at time t is determined by the current state and the *output function* $h : \mathcal{S} \rightarrow \mathcal{O}$, so that $h(s_t) = o_t$. We also assume that the plan executor has a clock and can determine the

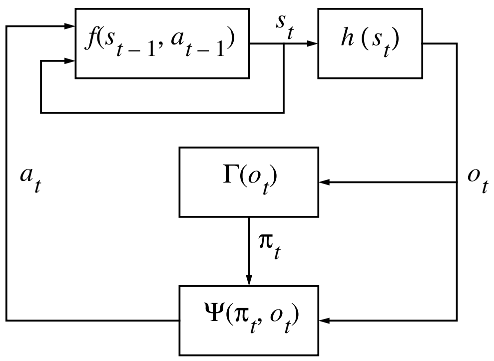

Figure 1: A block diagram for the general planning problem with state-transition function f, output function h, planner Γ, and plan executor Ψ

current time t.

Figure 1 depicts the general planning problem. The planner is notated as Γ; it takes as input the current observation o_t and has as output the current plan, π_t. The planner need not issue a new plan on every state transition and can keep a history of past observations if required. The plan executor is notated Ψ; it takes as input the current observation o_t and the current plan π_t and has as output the current action a_t.

If the state transitions are governed by a stochastic process, then the state-transition and output functions are random functions and we define the state-transition and output conditional probability distributions as follows.

$$\Pr(f(s_t, a_t)|s_t, a_t)$$

$$\Pr(h(s_t)|s_t)$$

In the general case, it requires $O(2^N)$ storage to encode these distributions for boolean state variables. However, in many practical cases, these probability distributions can be factored by taking advantage of independence among state variables.

As mentioned earlier, we assume that the ith state variable at time t depends on a small subset (of size at most M) of the state variables at time $t-1$. Let $\text{Parents}(i, s)$ denote the subset of state variables that the ith state variable depends on in s. We can represent the conditional probability distribution governing state transitions as the following product.

$$\Pr(\langle g_1(s_t, a_t), \ldots, g_N(s_t, a_t)\rangle | s_t, a_t) = \prod_{i=1}^{N} \Pr(g_i(s_t, a_t)|\text{Parents}(i, s_t), a_t)$$

This factored representation requires only $O(N2^M)$ storage for boolean state variables, which is reasonable assuming that M is relatively small.

The above descriptions of dynamical systems provide the semantics for a planning system embedded in a dynamic environment. There remains the question of syntax: specifically, how do you represent the dynamical system? In artificial intelligence, the answer varies widely. Researchers have used first-order logic [1], dynamic logic [34], state-space operators [13], and factored probabilistic state-transition functions [9]. In the later sections, we examine some of these representations in more detail.

In some variants of job-shop scheduling the dynamics are relatively simple. We might assume, for example, that if a job is started on a given machine, it will successfully complete in a fixed, predetermined amount of time known to the planner. Everything is under the control of the planner and evaluating the consequences of a given plan (schedule) is almost trivial from a computational standpoint.

We can easily imagine variants of the travel planning problems in which the dynamics are quite complicated. For example, we might wish to model flight cancellations and delays due to weather and mechanical failure in terms of a stochastic process. The planner cannot control the weather but it can plan to avoid the deleterious effects of the weather (*e.g.,* take a Southern route if a chance of snow threatens to close Northern airports). In this case, there are factors not under control of the planner and evaluating a given travel plan may require significant computational overhead.

2.2 Representing Plans of Action

We have already introduced a set of actions $\mathcal{A}$. We assume that these actions are *primitive* in that they can be carried out by the hardware responsible for executing plans. Semantically, a plan π is a mapping from what is known at the time of execution to the set of actions. The set of all plans for a given planning problem is notated Π.

For example, a plan might map the current observation o_t to the action a_t to take in the current state s_t. Such a plan would be independent of time. Alternatively, a plan might ignore observations altogether and map the current time t to the action to take in state s_t. Such a plan is independent of the current state, or at least the observable aspects of the current state.

If the action specified by a plan is dependent on observations of the current state, then we say that the plan is *conditional.* If the action specified by a plan is dependent on the current time, then we say that the plan is *time variant*, otherwise we say it is *stationary.* If the mapping is one-to-one, then we say that the plan is deterministic, otherwise it is nondeterministic and possibly stochastic if the mapping specifies a distribution over possible actions.

Conditional plans are said to run in a *closed loop*, since they enable the executor to react to the consequences of prior actions. Unconditional plans are said to run in an *open loop*, since they take no account of exogenous events or the consequences of prior actions that were not predicted using the dynamical model.

Now that we have the semantics for plans, we can think about how to represent them. If the mapping is a function, we can use any convenient representation for functions, including decision trees, tabular formats, hash tables, or artificial neural networks. In some problems, an unconditional, time-variant, deterministic plan is represented as a simple sequence of actions. Alternatively, we might use a set of possible sequences of actions perhaps specified by a partially ordered set of actions to represent a non-deterministic plan (*i.e.,* the plan allows any total order (sequence) consistent with the given partial order).

2.3 Measuring Performance

For a deterministic dynamical system in initial state s_0, a plan π determines a (possibly infinite) sequence of states $h_\pi = \langle s_0, s_1, \ldots \rangle$, called a *history* or *state-space trajectory.* More generally, a dynamical system together with a plan induces a probability distribution over histories, and h_π is a random variable governed by this distribution. A value function V assigns to each history a real value. In the deterministic case, the performance J of a plan π is the value of the resulting history, $\mathrm{J}(\pi) = \mathrm{V}(h_\pi)$. In the general case, the performance J of a plan is the expected value according to V over all possible histories, $\mathrm{J}(\pi) = \mathrm{E}[\mathrm{V}(h_\pi)]$, where E denotes taking an expectation.

In artificial intelligence planning (distinct from scheduling), much of the research has focused on goal-based performance measures. A *goal* $\mathcal{G}$ is a subset of the set of states $\mathcal{S}$.

$$\mathrm{V}(\langle s_0, s_1, \ldots \rangle) = \begin{cases} 1 & \text{if } \exists i, s_i \in \mathcal{G} \\ 0 & \text{otherwise} \end{cases}$$

Alternatively, we can consider the number of transitions until we reach a goal state as a measure of performance.

$$\mathrm{V}(\langle s_0, s_1, \ldots \rangle) = \begin{cases} -\min_i s_i \in \mathcal{G} & \text{if } \exists i, s_i \in \mathcal{G} \\ -\infty & \text{otherwise} \end{cases}$$

In the stochastic case, the corresponding measure of performance is called *expected time to target*, and the objective in planning is to minimize this measure.

Generalizing on the expected-time-to-target performance measure, we can assign to each state a cost using the cost function C. This cost function yields the following value function on histories.

$$\mathrm{V}(\langle s_0, s_1, \ldots \rangle) = -\sum_{i=0}^{\infty} \mathrm{C}(s_i)$$

In some problems, we may wish to discount future costs using a discounting factor $0 \geq \gamma < 1$.

$$\mathrm{V}(\langle s_0, s_1, \ldots \rangle) = -\sum_{i=0}^{\infty} \gamma^i \mathrm{C}(s_i)$$

This performance measure is called *discounted cumulative cost.* The above value functions are said to be *separable* since the total value of a history is a simple sum or weighted sum (in the discounted case) of the costs of each state in the history.

It should be noted that we can use any of the above methods for measuring the performance of a plan to define either a satisficing criterion (*e.g.,* find a plan whose performance is above some fixed threshold) or an optimizing criterion (*e.g.,* find a plan maximizing a given measure of performance).

2.4 Categories of Planning Problems

Now we are in a position to describe some basic classes of planning problems. A planning problem can be described in terms of its dynamics, either deterministic or stochastic. We might also consider whether the actions of the planner completely or only partially determine the state of the environment.

A planning problem can be described in terms of the knowledge available to the planner or executor. In the problems considered in this chapter, we assume that the planner has an accurate model of the underlying dynamics, but this need not be the case in general. Even if the planner has an accurate predictive model, the executor may not have the necessary knowledge to make use of that model. In particular, the executor may have only partial knowledge of the system state and that knowledge may be subject to errors in observation (*e.g.,* noisy, error-prone sensors).

We can assume that all computations performed by the planner are carried out prior to any execution, in which case the planning problem is said to be *off-line*. Alternatively, the planner may periodically compute a new plan and hand it off to the executor; this sort of planning problem is said to be *on-line*. Given space limitations, we are concerned primarily with off-line planning problems in this chapter.

Now that we have some familiarity with the various classes of planning problems, we consider some specific techniques for solving them. Our emphasis is on the design, analysis and application of planning algorithms.

3 Algorithms, Complexity and Search

Once we are given a set of possible plans and a *performance function* implementing a given performance measure, we can cast any planning or scheduling problem as a search problem. If we assume that evaluating a plan (applying the performance function) is computationally simple, then the most important issue concerns how we search the space of possible plans. Specifically, given one or more plans currently under consideration, how do we extend the search to consider other, hopefully better performing plans? We focus on two methods for extending search: *refinement* methods and *repair* methods.

A refinement method takes an existing partially specified plan (schedule) and refines it by adding detail. In job-shop scheduling, for example, we might take a (partial) plan that assigns machines and times to k of the jobs, and extend it so that it accounts for $k + 1$ jobs. Alternatively, we might build a plan in chronological order by assigning the earliest interval with a free machine on each iteration.

A repair method takes a completely specified plan and attempts to transform it into another completely specified plan with better performance. In travel planning, we might take a plan that makes use of one airline's flights and modify it to use the flights of another, possibly less expensive or more reliable airline. Repair methods often work by first analyzing a plan to identify unwanted interactions or bottlenecks and then attempting to eliminate the identified problems.

The rest of this section is organized as follows. In Section 3.1, we briefly survey what is known about the complexity of planning and scheduling problems, irrespective of what methods are used to solve them. In Section 3.2, we focus on traditional search methods for generating plans of actions given deterministic dynamics. We begin with open-loop planning problems with complete knowledge of the initial state, progressing to closed-loop planning problems with incomplete knowledge of the initial state. In Section 3.3, we focus on methods for generating schedules given deterministic dynamics. In both Sections 3.2 and 3.3, we discuss refinement- and repair-based methods. In Section 3.4, we mention related work in machine learning concerned with learning search rules and adapting previously generated solutions to planning and scheduling problems. In Section 3.5, we consider a class of planning problems involving stochastic dynamics and address some issues that arise in trying to approximate the value of conditional plans in stochastic domains. Our discussion begins with a quick survey of what is known about the complexity of planning and scheduling problems.

3.1 Complexity Results

Garey and Johnson [14] provide an extensive listing of NP-hard problems, including a great many scheduling problems. They also provide numerous examples of how a hard problem can be rendered easy by relaxing certain assumptions. For example, most variants of job-shop scheduling are NP-hard. Suppose, however, that you can suspend work on one job in order to carry out a rush job, resuming the suspended job on completion of the rush job so that there is no time lost in suspending and resuming. With this assumption, some hard problems become easy. Unfortunately, most real scheduling problems are NP-hard. Graham *et al.* [17] provide a somewhat more comprehensive survey of scheduling problems with a similarly dismal conclusion. Lawler *et al.* [26] survey results for the traveling salesperson problem, a special case of our travel planning problem. Here again the prospects for optimal, exact algorithms are not good, but there is some hope for approximate algorithms.

With regard to open-loop, deterministic planning, Chapman [5], Bylander [4], and Gupta and Nau [18] have shown that most problems in this general class are hard. Dean and Boddy [8] show that the problem of evaluating plans represented as sets of partially ordered actions is NP-hard in all but the simplest cases. Bäckström and Klein [3] provide some examples of easy (polynomial time) planning problems, but these problems are of marginal practical interest.

Regarding closed-loop, deterministic planning, Papadimitriou and Tsitsiklis [29] discuss polynomial-time algorithms for finding an optimal conditional plan for a variety of performance functions. Unfortunately, the polynomial is in the size of the state space. As mentioned earlier, we assume that the size of the state space is exponential in the number of state variables. Papadimitriou and Tsitsiklis also list algorithms for the case of stochastic dynamics that are polynomial in the size of the state space.

From the perspective of worst-case, asymptotic time and space complexity, most practical planning and scheduling problems are computationally very difficult. The literature on planning and scheduling in artificial intelligence generally takes it on faith that any interesting problem is at least NP-hard. The research emphasis is on finding powerful heuristics and clever search algorithms. In the remainder of this section, we explore some of the highlights of this literature.

3.2 Planning with Deterministic Dynamics

In the following section, we consider a special case of planning in which each action deterministically transforms one state into another. Nothing changes without the executor performing some action. We assume that the planner has an accurate model of the dynamics. If we also assume that we are given complete information about the initial state, it will be sufficient to produce unconditional plans that are produced off-line and run in an open loop.

Recall that a state is described in terms of a set of state variables. Each state assigns to each state variable a value. To simplify the notation, we restrict our attention to boolean variables. In the case of boolean variables, each state variable is assigned either true or false. Suppose that we have three boolean state variables: P, Q, and R. We represent the particular state s in which P and Q are true and R is false by the *state-variable assignment*, $s = \{P = \text{true}, Q = \text{true}, R = \text{false}\}$, or, somewhat more compactly, by $s = \{P, Q, \neg R\}$, where $X \in s$ indicates that X is assigned true in s and $\neg X \in s$ indicates that X is assigned false in s.

An action is represented as a *state-space operator* α defined in terms of *preconditions* ($\text{Pre}(\alpha)$) and *postconditions* (also called effects) ($\text{Post}(\alpha)$). Preconditions and postconditions are represented as state-variable assignments that assign values to subsets of the set of all state variables. Here is an example operator α_{eg}:

Operator α_{eg}

Preconditions: $P, \neg R$

Postconditions: $\neg P, \neg Q$

If an operator (action) is applied (executed) in a state in which the preconditions are satisfied, then the variables mentioned in the postconditions are assigned their respective values in the resulting state. If the preconditions are not satisfied, then there is no change in state.

In order to describe the state-transition function, we introduce a notion of consistency and define two operators $\oplus$ and $\ominus$ on state-variable assignments. Let φ and ϑ denote state-variable assignments. We say that φ and ϑ are *inconsistent* if there is a variable X such that φ and ϑ assign X different values; otherwise, we say that φ and ϑ are *consistent*. The operator $\ominus$ behaves like set difference with respect to the variables in assignments. The expression $\varphi \ominus \vartheta$ denotes a new assignment consisting of the assignments to variables in φ that have no assignment in ϑ (*e.g.*, $\{P,Q\} \ominus \{P\} = \{P,Q\} \ominus \{\neg P\} = \{Q\} \ominus \{\} = \{Q\}$). The operator $\oplus$ takes two consistent assignments and returns their union (*e.g.*, $\{Q\} \oplus \{P\} = \{P,Q\}$, but $\{P\} \oplus \{\neg P\}$ is undefined).

The state-transition function is defined as follows.

$$f(s,\alpha) = \begin{cases} s & \text{if } s \text{ and } \mathrm{Pre}(\alpha) \text{ are inconsistent} \\ \mathrm{Post}(\alpha) \oplus (s \ominus \mathrm{Post}(\alpha)) & \text{otherwise} \end{cases}$$

If we apply the operator α_{eg} to a state where the variables P and Q are true, and R is false, we have

$$f(\{P, Q, \neg R\}, \alpha_{eg}) = \{\neg P, \neg Q, \neg R\}$$

We extend the state-transition function to handle sequences of operators in the obvious way.

$$f(s, \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle) = f(f(s, \alpha_1), \langle \alpha_2, \ldots, \alpha_n \rangle)$$

$$f(s, \langle\rangle) = s$$

Our performance measure for this problem is goal based. Goals are represented as state-variable assignments that assign values to subsets of the set of all state variables. By assigning values to one or more state variables, we designate a set of states as the goal. We say that a state s <u>satisfies</u> a goal ϕ, notated $s \models \phi$, just in case the assignment ϕ is a subset of the assignment s. Given an initial state s_0, a goal ϕ, and a library of operators, the objective of the planning problem is to find a sequence of state-space operators $\langle \alpha_1, \ldots, \alpha_n \rangle$ such that $f(s_0, \langle \alpha_1, \ldots, \alpha_n \rangle) \models \phi$.

Using a state-space operator to transform one state into the next state is called *progression*. We can also use an operator to transform one goal into another, namely, the goal that the planner would have prior to carrying out the action corresponding to the operator. This use of an operator to transform goals is called *regression*. In defining regression, we introduce the notion of an impossible assignment, denoted $\perp$. We assume that if you regress a goal using an operator with postconditions that are inconsistent with the goal, then the resulting regressed goal is impossible to achieve. Here is the definition of regression:

$$b(\phi, \alpha) = \begin{cases} \perp & \text{if } \phi \text{ and Post}(\alpha) \text{ are inconsistent} \\ \text{Pre}(\alpha) \oplus (\phi \ominus \text{Post}(\alpha)) & \text{otherwise} \end{cases}$$

Conditional Postconditions and Quantification

Within the general operator-based state-transition framework described above, a variety of syntactic abbreviations can be used to facilitate compact action representation. For example, the postconditions of an action may be <u>conditional</u>. A conditional postcondition of the from $P \Rightarrow Q$ means that the action changes the value of the variable Q to true only if the value of P is true in the state where the operator is applied. It is easy to see that an action with such a conditional effect corresponds to two simpler actions, one which has a precondition P and the postcondition Q, and the other which has a precondition $\neg P$ and does not mention Q in its postconditions.

Similarly, when state variables can be typed in terms of objects in the domain to which they are related, it is possible to express preconditions and postconditions of an operator as quantified formulas. As an example, suppose in the travel domain, we have one state variable $loc(c)$ which is true if the agent is in city c and false otherwise. The action of flying from city c to city c' has the effect that the agent is now at city c', and the agent is not in any other city. If there are n cities, $c_1, \ldots, c_n$, the latter effect can be expressed either as a set of propositional postconditions $\neg loc(c_1), \ldots, \neg loc(c_{j-1}), \neg loc(c_{j+1}), \ldots, \neg loc(c_n)$ where $c' = c_j$, or, more compactly, as the quantified effect $\forall_{z:city(z)}\ z \neq c' \Rightarrow \neg loc(z)$. Since operators with conditional

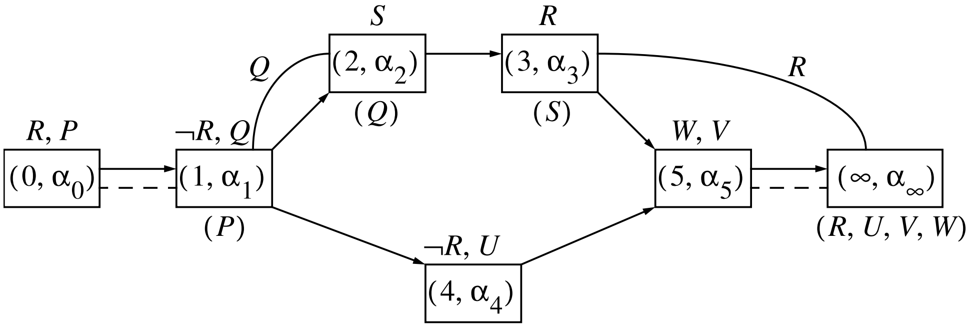

Figure 2: This figure depicts the partial plan π_{eg}. The postconditions (effects) of the steps are shown above the steps, while the preconditions are shown below the steps in parentheses. The ordering constraints between steps are shown by arrows. The interval preservation constraints are shown by arcs, while the contiguity constraints are shown by dotted lines.

postconditions and quantified preconditions and postconditions are just shorthand notations for finitely many propositional operators, the transition function, as well as the progression and regression operations can be modified in straightforward ways to accommodate them. For example, if a goal formula $\{W, S\}$ is regressed through an operator having preconditions $\{P, Q\}$ and postconditions $\{R \Rightarrow \neg W\}$, we get $\{\neg R, S, P, Q\}$. Note that by making $\neg R$ a part of the regressed formula, we ensure that $\neg W$ will not be a postcondition of the operator, thereby averting the inconsistency with the goals.

Representing Partial Plans

Although solutions to the planning problems can be represented by operator sequences, to facilitate efficient methods of plan synthesis, it is useful to have a more flexible representation for partial plans. A *partial plan* consists of a set of *steps*, a set of *ordering constraints* that restrict the order in which steps are to be executed, and a set of *auxiliary constraints* that restrict the value of state variables over particular intervals of time. Each step is associated with a state-space operator. To distinguish between multiple instances of the same operator appearing in a plan, we assign each step a unique integer i and represent the ith step as the pair (i, α_i) where α_i is the operator associated with the ith step.

Figure 2 shows a partial plan π_{eg} consisting of seven steps. The plan π_{eg} is represented as follows.

$$
\begin{aligned}
\langle \quad & \{(0,\alpha_0),(1,\alpha_1),(2,\alpha_2),(3,\alpha_3),(4,\alpha_4),(5,\alpha_5),(\infty,\alpha_\infty\}, \\
& \{(0 \mathrel{\bar{\prec}} 1),(1 \prec 2),(1 \prec 4),(2 \prec 3),(3 \prec 5),(4 \prec 5),(5 \mathrel{\bar{\prec}} \infty)\}, \\
& \{(1 \stackrel{Q}{-} 2),(3 \stackrel{R}{-} \infty)\} \quad \rangle
\end{aligned}
$$

An ordering constraint of the form $(i \prec j)$ indicates that Step i precedes Step j. An ordering constraint of the form $(i \mathrel{\bar{\prec}} j)$ indicates that Step i is contiguous with Step j, that is Step i precedes Step j and no other steps intervene. The steps are partially ordered in that Step 2 can occur either before or after Step 4. An auxiliary constraint of the form $(i \stackrel{P}{-} j)$ is called an *interval preservation constraint* and indicates that P is to be preserved in the range between Steps i and j (and therefore no operator with postcondition $\neg P$ should occur between Steps i and j). In particular, according to the constraint $(3 \stackrel{R}{-} \infty)$, Step 4 should not occur between Steps 3 and ∞.

The set of steps $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ with contiguity constraints

$$\{(\sigma_0 \mathrel{\bar{\prec}} \sigma_1),(\sigma_1 \mathrel{\bar{\prec}} \sigma_2),\ldots,(\sigma_{n-1} \mathrel{\bar{\prec}} \sigma_n)\}$$

is called the *header* of the plan π. The last element of the header σ_n is called the *head step*. The state defined by $f(s_0, \langle \alpha_{\sigma_1}, \ldots, \alpha_{\sigma_n} \rangle)$, where α_{σ_i} is the operator associated with σ_i is called the *head state*. In a similar manner, we can define the *tail*, *tail step*, and *tail state*.

As an example, the partial plan π_{eg} shown in Figure 2 has the Steps 0 and 1 in its header, with Step 1 being the head step. The head state (which is the state resulting from applying α_1 to the initial state) is $\{P, Q\}$. Similarly, the tail consists of Steps 5 and ∞, with Step 5 being the tail step. The tail state (which is the result of regressing the goal conditions through the operator α_5) is $\{R, U\}$.

Refinement Search

A large part of the work on plan synthesis in artificial intelligence falls under the rubric of refinement search. Refinement search can be seen as search in the space of partial plans. The search starts with the empty partial plan, and adds details to that plan until a complete plan results. Semantically, a partial plan can be seen as a shorthand notation for the set of complete plans (action sequences) that are consistent with the constraints. A refinement strategy converts a partial plan π into a set of new plans $\{\pi_1, \ldots, \pi_n\}$ such that all the potential solutions represented by π are represented by at least one of $\pi_1, \ldots, \pi_n$. Syntactically, this is accomplished by generating each of the children plans (refinements) by adding additional constraints to π.

The following is a general template for refinement search. The search starts with the null plan $\langle \{(0, \alpha_0), (\infty, \alpha_\infty)\}, \{(0 \prec \infty)\}, \{\} \rangle$, where α_0 is a dummy operator with no preconditions and postconditions corresponding to the initial state, and α_∞ is a dummy operator with no postconditions and preconditions corresponding the goal. For example, if we were trying to find a sequence of actions to transform the initial state $\{P, Q, \neg R\}$ into a state satisfying the goal $\{R\}$, then we would have $\mathrm{Pre}(\alpha_0) = \{\}$, $\mathrm{Post}(\alpha_0) = \{P, Q, \neg R\}$, $\mathrm{Pre}(\alpha_\infty) = \{R\}$, and $\mathrm{Post}(\alpha_\infty) = \{\}$.

We define a generic refinement procedure, Refine(π), as follows [23].

1. If an action sequence $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ corresponds to a total order consistent with both ordering constraints and the auxiliary constraints of π, and is a solution to the planning problem, then terminate and return $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$.

2. If the constraints in π are inconsistent, then eliminate π from future consideration.

3. Select a refinement strategy, and apply the strategy to π and add the resulting refinements to the set of plans under consideration.

4. Select a plan π' from those under consideration and call Refine(π').

In Step 3, the search selects a refinement strategy to be applied to the partial plan. There are several possible choices here, corresponding intuitively to different ways of splitting the set of potential solutions represented by the plan. In the following sections, we outline four popular refinement strategies employed in the planning literature.

State-Space Refinements

The most straightforward way of refining partial plans involves using progression to convert the initial state into a state satisfying the goal conditions, or using regression to convert a set of goal conditions into a set of conditions that are satisfied in the initial state. From the point of view of partial plans, this corresponds to growing the plan from either the beginning or the end.

Progression (or forward state-space) refinement involves advancing the head state by adding a step σ, such that the preconditions of α_σ are satisfied in the current head state, to the header of the plan. The step σ may be newly added to the plan or currently present in the plan. In either case, it is made contiguous to the current head step and becomes the new head step.

As an example, one way of refining the plan π_{eg} in Figure 2 using progression refinement would be to apply an instance of the operator α_2 (either the instance that is currently in the plan $(2, \alpha_2)$ or a new instance) to the head state (recall that it is $\{P, Q\}$). This is accomplished by putting a contiguity constraint between $(2, \alpha_2)$ and the current head step $(1, \alpha_1)$ (thereby making the former the new head step).

We can also define a refinement strategy based on regression, which involves regressing the tail state of a plan through an operator. For example, the operator α_3 is applicable (in the backward direction) through this tail state (which is $\{R, U\}$), while the operator α_4 is not (since its postconditions are inconsistent with the tail state). Thus, one way of refining π_{eg} using regression refinement would be to apply an instance of the operator α_3 (either the existing instance in Step 3 or a new one) to the tail state in the backward direction. This is accomplished by putting a contiguity constraint between $(3, \alpha_3)$ and the current tail step.

From a search control point of view, one of the important questions is deciding which of the many refinements generated by progression and regression refinements are most likely to lead to a solution. It is possible to gain some focus by using *state difference heuristics*, which prefer the refinements where the set difference between the tail state and the head state is the smallest.

While the state difference heuristic works well enough for regression refinements, it does not provide sufficient focus to progression refinements. The problem is that in a realistic planning problem, there potentially may be many operators that are applicable in the current head state, and only a few of them may be relevant to the goals of the problem. Thus, the strategy of generating all the refinements and ranking them with respect to the state difference heuristic can be prohibitively expensive. We need a method of automatically zeroing on those operators which are possibly relevant to the goals.

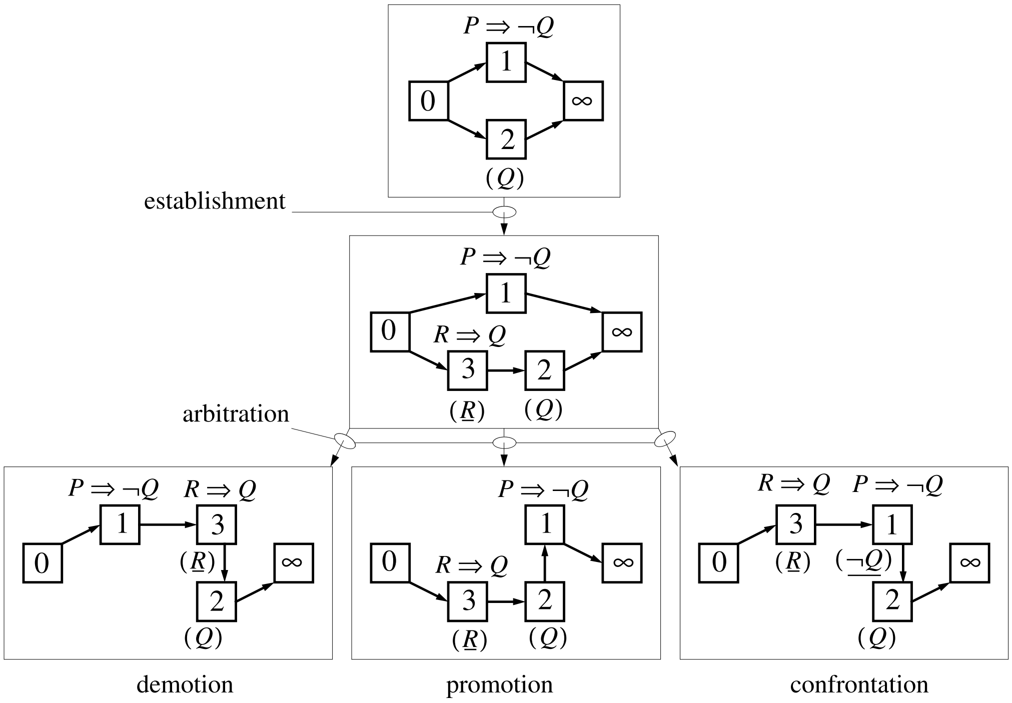

Figure 3: An example of precondition establishment. This diagram illustrates an attempt to establish Q for Step 2. Establishing a postcondition can result in a potential conflict, which requires arbitration to avert the conflict. Underlined preconditions correspond to secondary preconditions.

One popular way of generating the list of relevant operators is to use *means-ends analysis*. The general idea is the following. Suppose we have an operator α whose postconditions match a goal of the problem. Clearly, α is a relevant operator. If the preconditions of α are satisfied in the head state of the current partial plan, we can apply it directly. Suppose they are not all satisfied. In such a case, we can consider the preconditions of α as subgoals, look for an operator α' whose postconditions match one of these subgoals, and check if it is applicable to the head state. This type of recursive analysis can be continued to find the set of relevant operators, and focus progression refinement.

Plan-Space Refinements

As we saw above, in state-space refinements, partial plans are extended by adding new steps and new contiguity constraints. The contiguity constraints are required since without them the head state and tail state are not well defined. State-space refinements have the disadvantage that they completely determine the order and position of every step introduced into the plan.

While it is easy to see whether or not a given step is relevant to a plan, often the precise position at which a step must occur in the final plan is not apparent until all of the steps have been added. In such situations, state-space refinement can lead to premature commitment to the order of steps, causing extensive backtracking.

Plan-space refinement attempts to avoid this premature commitment. The main idea in plan-space refinement is to shift the attention from advancing the world state to establishing goals. A precondition P of a step (i, α_i) in a plan is said to be *established* if there is some step (j, α_j) in the plan that precedes i and causes P to be true, and no step that can possibly intervene between j and i has postconditions that are inconsistent with P. It is easy to see that if every precondition of every step in the plan is established, then that plan will be a solution plan. Plan-space refinement involves picking a precondition P of a step (i, α_i) in the partial plan, and adding enough additional step, ordering, and auxiliary constraints to ensure the establishment of P.

We illustrate the main ideas in precondition establishment through an example. Consider the partial plan at the top in Figure 3. Step 2 in this plan requires a precondition Q. To establish this precondition, we need a step which has Q as its postcondition. None of the existing steps have such a postcondition. Suppose an operator α_3 in the library has a postcondition $R \Rightarrow Q$. We introduce an instance of α_3 as Step 3 into the plan. Step 3 is ordered to come before Step 2 (and after Step 0). Since α_3 makes Q true only when R is true before it, to make sure that Q will be true following Step 3, we need to ensure that R is true before it. This can be done by posting R as a precondition of Step 3. Since R is not a normal precondition of α_3, and is being posted only to guarantee one of its conditional effects, it is called a *secondary precondition* [30].

Now that we have introduced Step 3 and ensured that it produces Q as a postcondition, we need to make sure that Q is not violated by any steps possibly intervening between Steps 3 and 2. This phase of plan-space refinement is called *arbitration*. In our example, Step 1, which can possibly intervene between Steps 3 and 2, has a postcondition $P \Rightarrow \neg Q$, that is potentially inconsistent with Q. To avert this inconsistency, we can either order Step 1 to come before Step 3 (demotion), or order Step 1 to come after Step 2 (promotion), or ensure that the offending conditional effect will not occur. This last option, called confrontation, can be carried out by posting $\neg P$ as a (secondary) precondition of Step 1. All these partial plans, corresponding to different ways of establishing the precondition Q at Step 2 are returned as the refinements of the original plan.

One problem with this precondition-by-precondition establishment approach is that the steps added in establishing a precondition might unwittingly violate a previously established precondition. Although this does not affect the completeness of the refinement search, it can lead to wasted planning effort, and necessitate repeated establishments of the same precondition within the same search branch. Many variants of plan-space refinements avoid this inefficiency by <u>protecting</u> their establishments. Whenever a condition P of a step σ is established with the help of the effects of a step σ', an interval preservation constraint $(\sigma' \overset{P}{-} \sigma)$ is added to remember this establishment. If the steps introduced by later refinements violate this preservation constraint, those conflicts are handled much the same way as in the arbitration phase discussed above. In the example shown in Figure 3, we can protect the establishment of precondition Q by adding the constraint $3 \overset{Q}{-} 2$.

Although the order in which preconditions are selected for establishment does not have any effect on the completeness of a planner using plan-space refinement, it can have a significant impact on the size of the search space explored by the planner (and thereby its efficiency). Thus, any available domain specific information regarding the relative importance of the various types of preconditions can be gainfully exploited. As an example, in the travel domain, the action of taking a flight to go from one place to another may have as its preconditions having a reservation, and being at the airport. To the extent that having a reservation is considered more critical than being at the airport, we would want to work on establishing the former first.

Task-Reduction Refinements

In both the state-space and plan-space refinements, the only knowledge that is available about the planning task is in terms of primitive actions (that can be executed by the underlying hardware), and their preconditions and postconditions. Often, one has more structured planning knowledge available in a domain. For example, in a travel planning domain, we might have the knowledge that one can reach a destination by either "taking a flight" or by "taking a train". We may also know that "taking a flight" in turn involves making a reservation, buying a ticket, taking a cab to the airport, getting on the plane etc. In such a situation, we can consider "tak-

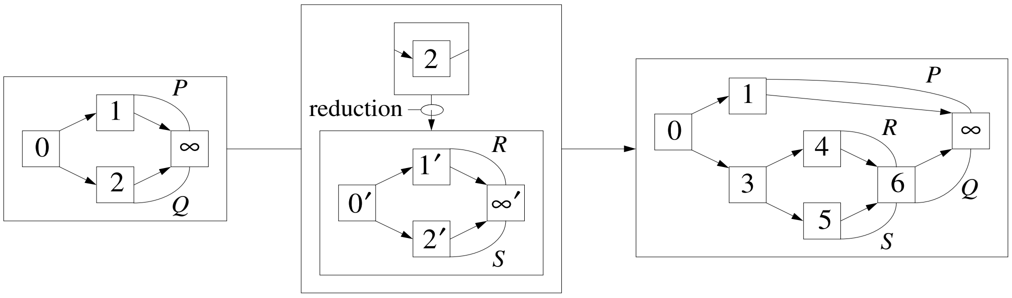

Figure 4: Step 2 in the partial plan shown on the left is reduced to obtain a new partial plan shown on the right. In the new plan, Step 2 is replaced with the (renamed) steps and constraints specified in the reduction shown in the center box.

ing a flight" as an abstract task (which cannot be directly executed by the hardware). This abstract task can then be reduced to a plan fragment consisting of other abstract or primitive tasks (in this case "making a reservation", "buying a ticket", "going to the airport", "getting on the plane"). This way, if there are some high-level problems with the "taking flight" action and other goals, (e.g. there is not going to be enough money to take a flight as well paying the rent), we can resolve them *before* we work on low level details such as getting to the airport.

This idea forms the basis for task reduction refinement. Specifically, we assume that in addition to the knowledge about primitive actions, we also have some abstract actions, and a set of schemas (plan fragments) that can replace any given abstract action. Task reduction refinement takes a partial plan π containing abstract and primitive tasks, picks an abstract task σ, and for each reduction schema (plan fragment) that can be used to reduce σ, a refinement of π is generated with σ replaced by the reduction schema (plan fragment). As an example, consider the partial plan on the left in Figure 4. Suppose the operator α_2 is an abstract operator. The central box in Figure 4 shows a reduction schema for Step 2, and the partial plan shown on the right of the figure shows the result of refining the original plan with this reduction schema. At this point any interactions between the newly introduced plan fragment and the previously existing plan steps can be resolved using techniques such as promotion, demotion and confrontation discussed in the context of plan-space refinement. This type of reduction is carried out until all the tasks are primitive.

In some ways, task reduction refinements can be seen as "macro refinements" that package together a series of state-space and plan-space refinements, thereby reducing a considerable amount of search. This, and the fact that in most planning domains, canned reduction schemas are readily available, have made task reduction refinement a very popular refinement choice for many applications.

Hybrid Refinements

Although early refinement planning systems tended to subscribe exclusively to a single refinement strategy, it is possible and often effective to use multiple refinement strategies. As an example, the partial plan π_{eg} shown in Figure 2 can be refined with progression refinement (*e.g.,* by putting a contiguity constraint between Step 1 and Step 2), with regression refinement (*e.g.,* by putting a contiguity constraint between Step 3 and Step 5), or plan-space refinement (*e.g.,* by establishing the precondition S of Step 3 with the help of the effect Step 2). Finally, if the operator α_4 is a non-primitive operator, we can also use task reduction refinement to replace α_4 with its reduction schema. There is some evidence that planners using multiple refinement strategies intelligently can outperform those using single refinement strategies [21]. However, the question as to which refinement strategy should be preferred when is still largely open.

Handling Incomplete Information

Although the refinement methods described above were developed in the context of planning problems where the initial state is completely specified, they can be extended to handle incompletely specified initial states. Incomplete specification of the initial state means that the values of some of the state variables in the initial state are not specified. Such incomplete specification can be handled as long as the state variables are observable (*i.e.,* the correct value of the variable can be obtained at execution time).

Suppose the initial state is incomplete with respect to the value of the state variable ϕ. If ϕ has only a small number of values, K, then we can consider this planning problem to be a collection of K problems, each with the same goal and a complete initial state in which ϕ takes on a specific value. Once the K problems are solved, we can make a K-way conditional plan that gives the correct plan conditional given the observed value of the state variable ϕ. There exist methods for extending refinement strategies so that instead of working on K unconditional plans with significant overlap, a single, multi-threaded conditional plan is generated [32].

Conditional planning can be very expensive in situations in which the unspecified variable ϕ has a large set of possible values or there are several unspecified variables. If there are U unspecified variables each with K possible values, then a conditional plan that covers all possible contingencies has to account for U^K possible initial states. In some cases, we can avoid a combinatorial explosion by performing some amount of on-line planning; first plan to obtain the necessary information, then, after obtaining this information, plan what to do next. Unfortunately, this on-line approach has potential problems.

In travel planning, for example, you could wait until you arrive in Boston's Logan Airport to check on the weather in Chicago in order to plan whether to take a Southern or Northern route to San Francisco. But, if you do wait and it is snowing in Chicago, you may find out that all of the flights taking Southern routes are already sold out. In this case, it would have been better to anticipate the possibility of snow in Chicago and reserve a flight to San Francisco taking a Southern route. Additional complications arise concerning the time when you observe the value of a given variable, the time when you need to know the value of a variable, and whether or not the value of a variable changes between when you observe it and when you need to know a value.

Uncertainty arises not only with respect to initial conditions, but also as a consequence of the actions of the planner (*e.g.,* you get stuck in traffic and miss your flight) or the actions of others (*e.g.,* the airline cancels your flight). In general, uncertainty is handled by introducing *sensing* or *information gathering* actions (operators). These operators have preconditions and postconditions similar to other operators, but some of the postconditions, those corresponding to the consequences of information gathering, are nondeterministic; we will not know the actual value of these postconditions until after we have executed the action [11].

The approach to conditional planning sketched above theoretically extends to arbitrary sources of uncertainty, but in practice search has to be limited to consider only outcomes that are likely to have a significant impact on performance. In Section 3.5 we briefly consider planning using stochastic models that quantify uncertainty involving outcomes.

Repair Methods in Planning

The refinement methods for plan synthesis described in Section 3.2 assume access to the complete dynamics of the system. Sometimes, the system dynamics are complex enough that using the full model during plan synthesis can be inefficient. In many such domains, it is often possible to come up with a simplified model of the dynamics that is approximately correct. As an example, in the travel domain, the action of taking a flight from one city to another has potentially many preconditions, including ones such as "have enough money to buy tickets" and "have enough clean clothes to take on the travel." Often, most of these preconditions are trivially satisfied, and we are justified in approximating the set of preconditions to simply ensure that we have a reservation and are at the airport on time. In such problems, a simplified model can be used to drive plan generation using refinement methods, and the resulting plan can then be tested with respect to the complete dynamical model of the system. If the testing shows the plan to be correct, we are done. If not, the plan needs to be repaired or debugged. This repair process involves both adding and deleting constraints from the plan.

If the complete dynamical model is declarative (instead of being a black box), it is possible to extract from the testing phase an explanation of why the plan is incorrect (for example, in terms of some of the preconditions that are not satisfied, or are violated by some of the indirect effects of actions). This explanation can then be used to focus the repair activity [35, 19]. Similar repair methods can also be useful in situations where we have probably approximately correct canned plans for generic types of goals, and we would like to solve planning problems involving collections of these goals by putting the relevant canned plans together and modifying them.

3.3 Scheduling with Deterministic Dynamics

As we mentioned earlier, scheduling is typically concerned with deciding when to carry out a given set of actions so as to satisfy various types of constraints on the order in which the actions need to be performed, and the ways in which different resources are consumed. Artificial intelligence approaches to scheduling typically declaratively represent and reason with the constraints.

Constraint-based schedulers used in a real applications generally employ sophisticated programming languages to represent a range of constraints. For example, many schedulers require temporal constraints that specify precedence, contiguity, duration, earliest and latest start and completion times for tasks.

In some schedulers, temporal constraints are enforced rigidly, so they never need to be checked during search. Many scheduling problems also manage a variety of resources. In the job-shop scheduling problem, machines are resources; only one task can be performed on a machine at a time. Other resources encountered in scheduling problems include fuel, storage space, human operators and crew, vehicles, and assorted other equipment. Tasks have constraints that specify their resource requirements and resources have capacity constraints that ensure that a schedule does not over allocate resources.

In addition to constraints on the time of occurrence and resources used by tasks, there are also constraints on the state of the world that are imposed by physics: the dynamics governing the environment. For example, a switch can be in the on or off position but not both at the same time. In some scheduling problems, the dynamical system is represented as a large set of state constraints.

Scheduling and Constraint Satisfaction

Scheduling problems are typically represented in terms of a set of variables and constraints on their values. A schedule is then represented as an assignment of values to all of the variables that satisfies all the constraints. The resulting formulation of scheduling problems is called a *constraint satisfaction problem* [36].

Formally, a constraint satisfaction problem is specified by a set of n variables, $\{x_1, \ldots, x_n\}$, their respective value domains, $\Omega_1, \ldots, \Omega_n$, and a set of m constraints, $\{C_1, \ldots, C_m\}$. A constraint C_i involves a subset $\{x_{i_1}, \ldots, x_{i_k}\}$ of the set of all variables $\{x_1, \ldots, x_n\}$ and is defined by a subset of the Cartesian product $\Omega_{i_1} \times \cdots \times \Omega_{i_k}$. A constraint C_i is *satisfied* by a particular assignment in which $x_{i_1} \leftarrow v_{i_1}, \ldots, x_{i_k} \leftarrow v_{i_k}$ just in case $\langle v_{i_1}, \ldots, v_{i_k} \rangle$ is in the subset of $\Omega_{i_1} \times \cdots \times \Omega_{i_k}$ that defines C_i. A solution is an assignment of values to all the variables such that all of the constraints are satisfied. There is a performance function that maps every complete assignment to a numerical value representing the cost of the assignment. An optimal solution is a solution that has the lowest cost.

As an example, consider the following formulation of a simplified version of the job-shop scheduling problem as a constraint satisfaction problem. Suppose we have N jobs, $1, 2, \ldots, N$, each consisting of a single task, and M machines, $1, 2, \ldots, M$. Since there is exactly one task for each job, we just refer to jobs. Assume that each job takes one unit of time and there are T time units, $1, 2, \ldots, T$. Let $z_{ij} = 1$ if the jth machine can handle the ith job and $z_{ij} = 0$ otherwise. The z_{ij} are specified in the description of the problem. Let x_i for $1 \geq i \geq N$ take on values from $\{j | z_{ij} = 1\} \times \{1, 2, \ldots, T\}$, where $x_i = (j, k)$ indicates that the ith job is assigned to the jth machine during the kth time unit. The x_i are assigned values in the process of planning. There are $N(N-1)$ constraints of the form $x_i \neq x_j$, where $1 \leq i, j \leq N$ and $i \neq j$. We are searching for an assignment to the x_i that satisfies these constraints.

Refinement-Based Methods

A refinement-based method for solving a constraint satisfaction problem progresses by incrementally assigning values to each of the variables. A partial plan (schedule) π is represented as partial assignment of values to variables $\{x_{\pi_1} \leftarrow v_{\pi_1}, \ldots, x_{\pi_k} \leftarrow v_{\pi_k}\}$, where $\{x_{\pi_1}, \ldots, x_{\pi_k}\}$ is a subset of the set of all variables, $\{x_1, \ldots, x_n\}$. The partial assignment π can be seen as a shorthand notation for all of the complete assignments that agree on the assignment of values to the variables in $\{x_{\pi_1}, \ldots, x_{\pi_k}\}$. A partial assignment π is said to be inconsistent if the assignment of values to variables in π already violates one or more constraints. If the partial assignment π is consistent, it can be refined by selecting a variable x_j that is not yet assigned a value in π and extending π to produce a set of refinements each of which assigns x_j one of the possible values from its domain Ω_j. Thus, the set of refinements of π is $\{\pi \cup \{x_j \leftarrow v\} | v \in \Omega_j\}$. In the case of satisficing scheduling, search terminates when a complete and consistent assignment is produced. In the case of optimizing scheduling, search is continued with <u>branch-and-bound</u> techniques until an optimal solution is found.

From the point of view of efficiency, it is known that the order in which the variables are considered and the order in which the values of the variables are considered during refinement have a significant impact on the efficiency of search. Considering variables with least number of possible values first is known to provide good performance in many domains. Other ways of improving search efficiency include using <u>look ahead</u> techniques to prune inconsistent partial assignments ahead of time, to process the domains of the remaining variables so that any infeasible values are removed, or using dependency directed backtracking techniques to recover from inconsistent partial assignments intelligently. See [36] for a description of these techniques and their tradeoffs.

Repair-Based Methods

A repair-based method for solving constraint satisfaction problems is to start with an assignment to all of the variables in which not all of the constraints are satisfied and reassign a subset of the variables so that more of the constraints are satisfied. Reassigning a subset of the variables is referred to as as repairing an assignment. Consider the following repair method for solving the simplified job-shop scheduling problem.

We say that two variables x_i and x_j $(i \neq j)$ *conflict* if their values violate a constraint; in the simplified job-shop scheduling problem considered here, a constraint is violated if $x_i = x_j$. *Min-conflicts* is a heuristic for repairing an existing assignment that violates some of the constraints to obtain a new assignment that violates fewer constraints. The hope is that by performing a short sequence of repairs as determined by the min-conflicts heuristic we obtain an assignment that satisfies all of the constraints. The min-conflicts heuristic counsels us to select a variable that is in conflict and assign it a new value that minimizes the number of conflicts. See the Johnston and Minton article in [37] for more on the min-conflicts heuristic.

Min-conflicts is a special case of a more general strategy that proceeds by making local repairs. In the job-shop scheduling problem, a local repair corresponds to a change in the assignment of a single variable.

For the traveling salesperson problem, there is a very effective local repair method that works quite well in practice. Suppose that there are five cities A, B, C, D, E and an existing tour (a path consisting of a sequence of edges beginning and ending in the same city) $(A, B), (B, C), (C, D), (D, E), (E, A)$. Take two edges in the tour, say (A, B) and (C, D), and consider the length of the tour $(A, C), (C, B), (B, D), (D, E), (E, A)$ that results from replacing (A, B) and (C, D) with (A, C) and (B, D). Try all possible pairs of edges (there are $O(L^2)$ such edges where L is the number of cities), and make the replacement (repair) that results in the shortest tour. Continue to make repairs in this manner until no improvement (reduction in the length of the resulting tour) is possible. Lin and Kernighan's algorithm [27] which is based on this local repair method generates solutions that are within 10% of the length of the optimal tour on a large class of practical problems.

Rescheduling and Iterative Repair Methods

Repair methods are typically implemented with iterative search methods; at any point during the scheduling process, there is a complete schedule available for use. This ready-when-you-are property of repair methods is important in applications that require frequent rescheduling, such as job-shops in which change orders and new rush jobs are a common occurrence.

Most repair methods employ greedy strategies that attempt to improve the current schedule on every iteration by making local repairs. Such greedy strategies often have a problem familiar to researchers in combinatorial optimization. The problem is that many repair methods, especially those that perform only local repairs, are liable to converge to local extrema of the performance function and thereby miss an optimal solution. In many cases, these local extrema correspond to very poor solutions.

To improve performance and reduce the risk of becoming stuck in local extrema corresponding to badly suboptimal solutions, some schedulers employ stochastic techniques that occasionally choose to make repairs other than those suggested by their heuristics. Simulated annealing [24] is one example of a stochastic search method used to escape local extrema in scheduling. In simulated annealing, there is a certain probability that the scheduler will choose a repair other than the one suggested by the scheduler's heuristics. These random repairs force the scheduler to consider repairs that at first may not look promising but in the long term lead to better solutions. Over the course of scheduling this probability is gradually reduced to zero. See the article by Zweben *et al.* in [37] for more on iterative repair methods using simulated annealing.

Another way of reducing the risk of getting stuck in local extrema involves making the underlying search systematic (so that it eventually visits all potential solutions). However, traditional systematic search methods tend to be too rigid to exploit local repair methods such as the min-conflicts heuristic. In general, local repair methods attempt to direct the search by exploiting the local gradients in the search space. This guidance can sometimes be at odds with the commitments that have already been made in the current search branch. Iterative methods do not have this problem since they do not do any bookkeeping about the current state of the search. Recent work on partial order dynamic backtracking algorithms [16] provides an elegant way of keeping both systematicity and freedom of movement.

3.4 Improving Efficiency

While the previous sections surveyed the methods used to organize the search for plans and discussed their relative advantages, as observed in Section 3.1, most planning problems are computationally hard. The only way we can expect efficient performance is to exploit the structure and idiosyncrasies of the specific applications. One attractive possibility involves dynamically customizing the performance of a general-purpose search algorithm to the structure and distribution of the application problems. A variety of machine learning methods have been developed and used for this purpose. We briefly survey some of these methods below.

One of the simplest ways of improving performance over time involves “caching” plans for frequently occurring problems and subproblems, and reusing them in subsequent planning scenarios. This approach is called *case-based planning (scheduling)* [19, 22] and is motivated by similar considerations to those motivating task-reduction refinements. In storing a previous planning experience, we have two choices: store the final plan, or store the plan along with the search decisions that lead to the plan. In the latter case, we exploit the previous experience by replaying the previous decisions in the new situation.

Caching typically involves only storing the information about the successful plan and the decisions leading to it. Often, there is valuable information in the search failures encountered in coming up with the successful plan. By analyzing the search failures and using *explanation-based learning* techniques, it is possible to learn *search control rules* that, for example, can be used to advise a planner as to which refinement or repair to pursue under what circumstances. For more about the connections between planning and learning see [28].

3.5 Approximation in Stochastic Domains

In this section, we consider a planning problem involving stochastic dynamics. We are interested in generating conditional plans for the case in which the state is completely observable (the output function is the identity $h(x_t) = x_t$) and the performance measure is expected discounted cumulative cost with discount γ. This constitutes an extreme case of closed-loop planning in which the executor is able to observe the current state at any time without error and without cost.

In this case, a plan is just a mapping from (observable) states to actions $\pi : \mathcal{S} \rightarrow \mathcal{A}$. To simplify the presentation, we notate states with the integers $0, 1, \ldots, |\mathcal{S}|$, where $s_0 = 0$ is the initial state. We refer to the performance of a plan π starting in state i as $\mathrm{J}(\pi|i)$. We can compute the performance of a plan by solving the following set of $|\mathcal{S}| + 1$ equations in $|\mathcal{S}| + 1$ unknowns.

$$\mathrm{J}(\pi|i) = \mathrm{C}(i) + \gamma \sum_{j=0}^{|\mathcal{S}|} \Pr(f(i, \pi(i)) = j | i, \pi(i)) \mathrm{J}(\pi|j)$$

The objective in planning is to find a plan π from the set of all possible plans Π such that for all $\pi' \in \Pi$, $\mathrm{J}(\pi|i) \geq \mathrm{J}(\pi'|i)$ for $0 \leq i \leq |\mathcal{S}|$.

As an aside, we note that the conditional probability distribution governing state transitions, $\Pr(f(i, \pi(i)) = j | i, \pi(i))$, can be specified in terms of *probabilistic state space operators*, allowing us to apply the techniques of Section 3.2. A probabilistic state space operator α is a set of triples of the form $\langle \phi, \rho, \omega \rangle$ where ϕ is a set of preconditions, ρ is a probability, and ω is a set of postconditions. Semantically, if ϕ is satisfied just prior to α, then with probability ρ the postconditions in ω are satisfied immediately following α. If a proposition is not included in ϕ, then it is assumed not to affect the outcome of α; if a proposition is not included in ω, then it is assumed to be unchanged by α. For example, given the following representation for α

$$\alpha = \{\langle \{P\}, 1, \emptyset \rangle, \langle \{\neg P\}, 0.2, \{P\} \rangle, \langle \{\neg P\}, 0.8, \{\neg P\} \rangle\}$$

if P is true prior to α, nothing is changed following α; but, if P is false, then 20% of the time P becomes true and 80% of the time P remains false. For more on planning in stochastic domains using probabilistic state space operators, see [25].

There are well known methods for computing an optimal plan for the problem described above [33]. Most of these methods proceed using iterative repair-based methods that work by improving an existing plan π using the computed function $\mathrm{J}(\pi|i)$. On each iteration, we end up with a new plan π' and must calculate $\mathrm{J}(\pi'|i)$ for all i. If, as we assumed earlier, $|\mathcal{S}|$ is exponential in the number of state variables, then we are going to have some trouble solving a system of $|\mathcal{S}| + 1$ equations. In the rest of this section, we consider one possible way to avoid incurring an exponential amount of work in evaluating the performance of a given plan.

Suppose that we know the initial state, s_0 and a bound $\mathrm{C}_{\max}$ ($\mathrm{C}_{\max} \geq \max_i \mathrm{C}(i)$) on the maximum cost incurred in any state. Let π be any plan, $\mathrm{J}_\infty(\pi) = \mathrm{J}(\pi|0)$ be the performance of π accounting for an infinite sequence of state transitions, and $\mathrm{J}_K(\pi)$ the performance of π accounting for only K state transitions. We can bound the difference between these two measures of performance as follows (see [12] for a proof).

$$|\mathrm{J}_\infty(\pi) - \mathrm{J}_K(\pi)| \leq \gamma^K \mathrm{C}_{\max}/(1-\gamma)$$

The above result implies that if we are willing to sacrifice a (maximum) error of $\gamma^K \mathrm{C}_{\max}/(1-\gamma)$ in measuring the performance of plans, we need only concern ourselves with histories of length K. So how do we calculate $\mathrm{J}_K(\pi)$? The answer is a familiar one in statistics, namely, we estimate $\mathrm{J}_K(\pi)$ by sampling the space of K-length histories.

Using a factored representation of the conditional probability distribution governing state transitions, we can compute a random K-length history in time polynomial in K and N (the number of state variables), assuming that M (the maximum dimensionality of a state-variable function) is constant. The algorithm is simply, given s_0, for $t = 0$ to $K-1$, determine s_{t+1} according to the distribution, $\Pr(s_{t+1}|s_t, \pi(s_t))$. For each history $\langle s_0, \ldots, s_K \rangle$ so determined we compute the quantity $\mathrm{V}(\langle s_0, \ldots, s_K \rangle) = \sum_{j=0}^{K} \gamma^j \mathrm{C}(s_j)$ and refer to this as one *sample*.

If we compute enough samples and take their average, we will have an accurate estimate of $\mathrm{J}_K(\pi)$. The following algorithm takes two parameters, ϵ and δ, and computes an estimate $\hat{\mathrm{J}}_K(\pi)$ of $\mathrm{J}_K(\pi)$ such that

$$\Pr[\mathrm{J}_K(\pi)(1-\epsilon) \leq \hat{\mathrm{J}}_K(\pi) \leq \mathrm{J}_K(\pi)(1+\epsilon)] > 1-\delta$$

1. $T \leftarrow 0$; $Y \leftarrow 0$
2. $S \leftarrow 4\log(2/\delta)(1+\epsilon)/\epsilon^2$
3. While $Y < S$ do
   (a) $T \leftarrow T + 1$
   (b) Generate a random history $\langle s_0, \ldots, s_K \rangle$

(c) $Y \leftarrow Y + \mathrm{V}(\langle s_0, \ldots, s_K \rangle)$

4. Return $\mathrm{J}_K(\pi) = S/T$

The above algorithm terminates after generating $\mathrm{E}[T]$ samples, where

$$\mathrm{E}[T] \leq 4 \log(2/\delta)(1+\epsilon) \left(\mathrm{J}_K(\pi)\epsilon^2\right)^{-1}$$

so that the entire algorithm for approximating $\mathrm{J}_\infty(\pi)$ runs in expected time polynomial in $1/\delta$, $1/\epsilon$, $1/(1-\gamma)$ (see [6] for a detailed analysis).

Approximating $\mathrm{J}_\infty(\pi)$ is only one possible step in an algorithm for computing an optimal or near-optimal plan. In most iterative repair-based algorithms, the algorithm evaluates the current policy and then tries to improve it on each iteration. In order to have a polynomial time algorithm, we not only have to establish a polynomial bound on the time required for evaluation but also a polynomial bound on the total number of iterations, The point of the above exercise is that when faced with combinatorial complexity, we need not give up but we may have to compromise. In practice, making reasonable tradeoffs is critical in solving planning and scheduling problems The simple analysis above demonstrates that we can trade time (the expected number of samples required) against the accuracy (determined by the ϵ factor) and reliability (determined by the δ factor) of our answers.

4 Research Issues and Summary

In this article, we provide a framework for characterizing planning and scheduling problems that focuses on properties of the underlying dynamical system and the capabilities of the planning system to observe its surroundings. The presentation of specific techniques distinguishes between refinement-based methods that construct plans and schedules piece by piece, and repair-based methods that modify complete plans and schedules. Both refinement- and repair-based methods are generally applied in the context of heuristic search.

Most planning and scheduling problems are computationally complex. As a consequence of this complexity, most practical approaches rely on heuristics that exploit knowledge of the planning domain. Current research focuses on improving the efficiency of algorithms based on existing representations and on developing new representations for the underlying dynamics that account for important features of the domain (*e.g.,* uncertainty) and allow for the encoding of appropriate heuristic knowledge. Given the complexity of most planning and scheduling problems, an important area for future research concerns identifying and quantifying tradeoffs, such as those involving solution quality and algorithmic complexity.

Planning and scheduling in artificial intelligence cover a wide range of techniques and issues. We have not attempted to be comprehensive in this relatively short article. Citations in the main text provide attribution for specifically mentioned techniques. These citations are not meant to be exhaustive by any means. General references are provided in the 'Further Information' section at the end of this article.

5 Defining Terms

closed-loop planner: a planning system that periodically makes observations of the current state of its environment and adjusts its plan in accord with these observations.

dynamical system: a description of the environment in which plans are to be executed that accounts for the consequences of actions and the evolution of the state over time.

goal: a subset of the set of all states such that a plan is judged successful if it results in the system ending up in one of these states.

off-line planning algorithm: a planning algorithm that performs all of its computations prior to executing any actions.

on-line planning algorithm: a planning algorithm in which planning computations and the execution of actions are carried out concurrently.

open-loop planner: a planning system that executes its plans with no feedback from the environment, relying exclusively on its ability to accurately predict the evolution of the underlying dynamical system.

optimizing: a performance criterion that requires maximizing or minimizing a specified measure of performance.

plan: a specification for acting that maps from what is known at the time of execution to the set of actions.

planning: a process that involves reasoning about the consequences of acting in order to choose from among a set of possible courses of action.

progression: the operation of determining the resulting state of a dynamical system given some initial state and specified action.

regression: the operation of transforming a given (target) goal into a prior (regressed) goal so that if a specified action is carried out in a state in which the regressed goal is satisfied, then the target goal will be satisfied in the resulting state.

satisficing: a performance criterion in which some level of satisfactory performance is specified in terms of a goal or fixed performance threshold.

state-space operator: a representation for an individual action that maps each state into the state resulting from executing the action in the (initial) state.

state-transition function: a function that maps each state and action deterministically to a resulting state. In the stochastic case, this function is replaced by a conditional probability distribution.

References

[1] Allen, J. F., Kautz, H. A., Pelavin, R. N., and Tenenberg, J. D., *Reasoning about Plans*, (Morgan-Kaufmann, San Francisco, California, 1991).

[2] Allen, James F., Hendler, James, and Tate, Austin, (Eds.), *Readings in Planning*, (Morgan Kaufmann, San Francisco, California, 1990).

[3] Bäckström, C. and Klein, I., Parallel Non-Binary Planning in Polynomial Time, *Proceedings IJCAI 12, Sydney, Australia*, IJCAII, 1991, 268–273.

[4] Bylander, Tom, Complexity Results for Planning, *Proceedings IJCAI 12, Sydney, Australia*, IJCAII, 1991, 274–279.

[5] Chapman, David, Planning for Conjunctive Goals, *Artificial Intelligence*, **32** (1987) 333–377.

[6] Dagum, P., Karp, R., Luby, M., and Ross, Sheldon M., An Optimal Stopping Rule for Monte Carlo Estimation, *Proceedings of the 1995 Symposium on the Foundations of Computer Science*, 1995.

[7] Dean, Thomas, Allen, James, and Aloimonos, Yiannis, *Artificial Intelligence: Theory and Practice*, (Benjamin/Cummings Publishing Company, Redwood City, California, 1995).

[8] Dean, Thomas and Boddy, Mark, Reasoning About Partially Ordered Events, *Artificial Intelligence*, **36**(3) (1988) 375–399.

[9] Dean, Thomas and Kanazawa, Keiji, A Model for Reasoning About Persistence and Causation, *Computational Intelligence*, **5**(3) (1989) 142–150.

[10] Dean, Thomas and Wellman, Michael, *Planning and Control*, (Morgan Kaufmann, San Francisco, California, 1991).

[11] Etzioni, Oren, Hanks, Steve, Weld, Daniel, Draper, Denise, Lesh, Neal, and Williamson, Mike, An Approach to Planning with Incomplete Information, *Proceedings of the 1992 International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[12] Fiechter, Claude-Nicolas, Efficient Reinforcement Learning, *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, 1994, 88–97.

[13] Fikes, Richard and Nilsson, Nils J., STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, **2** (1971) 189–208.

[14] Garey, Michael R. and Johnson, David S., *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, (W. H. Freeman and Company, New York, 1979).

[15] Georgeff, Michael P., Planning, Traub, J. F., (Ed.), *Annual Review of Computer Science, Volume 2*, (Annual Review Incorporated, 1987).

[16] Ginsberg, Matthew L. and McAllester, David, GSAT and Dynamic Backtracking, *Proceedings of the 1994 International Conference on Principles of Knowledge Representation and Reasoning, Bonn, Germany*, 1994.

[17] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G., Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Proceedings Discrete Optimization, Vancouver, British Columbia*, 1977.

[18] Gupta, Naresh and Nau, Dana S., Complexity Results for Blocks-World Planning, *Proceedings AAAI-91, Anaheim, California*, AAAI, 1991, 629–633.

[19] Hammond, Kristian J., *Case-Based Planning*, (Academic Press, New York, 1989).

[20] Hendler, James, Tate, Austin, and Drummond, Mark, AI Planning: Systems and techniques, *AI Magazine*, **11**(2) (1990) 61–77.

[21] Kambhampati, Subbarao, A comparative analysis of partial-order planning and task-reduction planning, *ACM SIGART Bulletin*, **6**(1) (1995).

[22] Kambhampati, Subbarao and Hendler, James, A Validation Structure Based Theory of Plan Modification and Reuse, *Artificial Intelligence*, **55**(2-3) (1992) 193–258.

[23] Kambhampati, Subbarao, Knoblock, Craig, and Yang, Qiang, Refinement Search as a unifying framework for evaluating design tradeoffs in partial order planning, *Artificial Intelligence*, (1995).

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

[25] Kushmerick, Nicholas, Hanks, Steve, and Weld, Daniel, An Algorithm for Probabilistic Planning, *Proceedings AAAI-94, Seattle, Washington*, AAAI, 1994.

[26] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., *The Travelling Salesman Problem*, (Wiley, New York, 1985).

[27] Lin, S. and Kernighan, B. W., An Effective Heuristic for the Travelling Salesman Problem, *Operations Research*, **21** (1973) 498–516.

[28] Minton, Steve, (Ed.), *Machine Learning Methods for Planning and Scheduling*, (Morgan Kaufmann, San Francisco, California, 1992).

[29] Papadimitriou, Christos H. and Tsitsiklis, John N., The Complexity of Markov Chain Decision Processes, *Mathematics of Operations Research*, **12**(3) (1987) 441–450.

[30] Pednault, Edwin P. D., Synthesizing Plans that Contain Actions with Context-Dependent Effects, *Computational Intelligence*, **4**(4) (1988) 356–372.

[31] Penberthy, J. S. and Weld, Daniel S., UCPOP: A Sound, Complete, Partial Order Planner for ADL, *Proceedings of the 1992 International Conference on Principles of Knowledge Representation and Reasoning*, 1992, 103–114.

[32] Peot, Mark and Shachter, Ross, Fusion and Propagation with Multiple Observations in Belief Networks, *Artificial Intelligence*, **48**(3) (1991) 299–318.

[33] Puterman, Martin L., *Markov Decision Processes*, (John Wiley & Sons, New York, 1994).

[34] Rosenschein, Stan, Plan Synthesis: A Logical Perspective, *Proceedings IJCAI 7*, IJCAII, 1981, 331–337.

[35] Simmons, Reid and Davis, Randall, Generate, Test and Debug: Combining Associational Rules and Causal Models, *Proceedings IJCAI 10, Milan, Italy*, IJCAII, 1987, 1071–1078.

[36] Tsang, Edward, *Foundations of Constraint Satisfaction*, (Academic Press, San Diego, California, 1993).

[37] Zweben, Monte and Fox, Mark S., (Eds.), *Intelligent Scheduling*, (Morgan Kaufmann, San Francisco, California, 1994).

Further Information

Research on planning and scheduling in artificial intelligence is published in the journal "Artificial Intelligence," "Computational Intelligence," and the "Journal of Artificial Intelligence Research." Planning and scheduling work is also published in the proceedings of the "International Joint Conference on Artificial Intelligence" and the "National Conference on Artificial

Intelligence." Specialty conferences such as the "International Conference on Artificial Intelligence Planning Systems" and the "European Workshop on Planning" cover planning and scheduling exclusively.

Georgeff [15] and Hendler *et al.* [20] provide useful summaries of the state of the art. Allen, Hendler, and Tate [2] is a collection of readings that covers many important innovations in automated planning. Dean *et al.* [7] and Penberthy and Weld [31] provide somewhat more detailed accounts of the basic algorithms covered in this chapter. Zweben and Fox [37] is a collection of readings that summarizes many of the basic techniques in knowledge-based scheduling. Allen *et al.* [1] describe an approach to planning based on first-order logic. Dean and Wellman [10] tie together techniques from planning in artificial intelligence, operations research, control theory, and the decision sciences.