# Chapter 24
# Other Approaches to Planning

## 24.1 Case-Based Planning

In *case-based planning*, the planning system has a *case library* of solutions to previous problems and uses these cases to solve new planning problems. Here are some of the reasons why one might want to do this.

- One of the most difficult parts of developing robust planning systems for practical applications is compiling the domain-specific knowledge for each new planning domain. Generative planners traditionally require a complete domain description that provides a clear semantics for the planners' inferencing mechanisms. However, in many planning domains, it is not feasible to develop anything more than a partial description of the domain. One way to reason about parts of the domain for which no complete domain description is available is to reuse plans that worked in previous planning episodes.
- As we have seen, planning is a computationally difficult activity even in very restricted planning domains. If similar kinds of planning problems occur often enough in some planning domain, retrieving and adapting old planning information may yield a significant speedup in the planning process.

**Example 24.1** In manufacturing industries, *process planning* is the task of planning which manufacturing operations to use to create a product. The most widely used technique for computer-aided process planning is *variant process planning* [118, 119], a case-based planning technique in which the case adaptation is done manually rather than by the computer. Given some information about a new product, a computer system will retrieve a process plan for a similar previous product, and a skilled human will modify this plan by hand to create a plan for the new product. One reason for the success of this technique is that when a company decides to produce a new product, the new product will usually be similar (in terms of its manufacturing requirements) to previous products that the company has produced. In such cases, adapting an old plan can be done much quicker than creating a new plan from scratch. ■

The typical problem-solving cycle of a case-based planner on a planning problem $\mathcal{P}$ consists of retrieval of one or more cases of solutions to planning problems similar to $\mathcal{P}$ according to some similarity metric, adaptation of the case or cases to solve $\mathcal{P}$, and possibly some revisions to the case base, retrieval mechanism, or adaptation mechanism based on feedback from the current planning episode.

**Similarity Metrics.** The purpose of a similarity metric is to enable the case-retrieval program to retrieve cases that can easily be adapted to solve the current problem. The basic assumption underlying most similarity metrics is that the more similar two planning problems are, the more likely it is that a solution to one of them can easily be adapted to produce a solution to the other one. Thus, the goal of a good similarity metric is to provide a good measure of how similar two problems are.

Several algorithms have been proposed for computing the similarity of two problems. A *static similarity measure* is one that will always return the same similarity value for a given pair of problems, regardless of when that value is computed. Examples are the similarity measures used in PRIAR [301], PRODIGY/Analogy [525], SPA [257], DerSNLP [275], and PARIS [61]. A *dynamic similarity measure* is one whose value may change after each problem-solving episode, in order to incorporate new information learned during that problem-solving episode. Examples include the similarity measures used in CaPER [322] and CAPlan/CbC [407].

**Case Adaptation.** Case-adaptation techniques can be classified into two primary approaches: transformational analogy [103] and derivational analogy [104]. In *transformational analogy*, the case or cases retrieved from the library will be modified to solve the new problem. The modifications may include reordering some of the actions in the plan, removing some of them, or changing some of the parameter bindings. A single case may be retrieved and transformed into a new solution, as in MOLGEN [203], PRIAR [301], SPA [257], and DIAL [356]. Alternatively, multiple cases, each corresponding to a small subproblem, may be combined and extended to solve a single larger problem [452, 453].

In *derivational analogy*, the case library contains *derivational traces* of the planning decisions, rather than the plans themselves, and the case adaptation consists of replaying the old decisions in the context of the new problem. Some examples of derivational case-based planners include PRODIGY/Analogy [525], PARIS [61], DerSNLP [275], CAPlan/CbC [407], and derUCP [25].

In many cases, the solution to $\mathcal{P}$ is obtained partly through case adaptation and partly through generative planning. Some case-based planners (e.g., CHEF [256] and DIAL [356]) do not have a generative component, and thus need a large case base to perform well across a wide variety of problems. PRODIGY/Analogy [523], DerSNLP [275], PARIS [61], and derUCP [25] integrate generative and case-based planning but require a complete domain description.

One way to do case-based planning without a complete domain description is through interactions with an expert human user. HICAP (see Chapter 22) gathers information it needs to determine the relevance of its cases through interaction with the user. CAPlan/CbC [407] and Mitchell's system [397] use interactions for plan adaptation.

**Revisions.** Several ways have been proposed for revising the planner's knowledge to incorporate information about the latest planning episode. Prominent among these are machine learning approaches, particularly explanation-based learning (EBL). PRODIGY [525] uses EBL to learn control rules to guide its search, PARIS [61] uses EBL to develop abstract cases that can later be refined after they have been retrieved, and DerSNLP+EBL [277] uses EBL to analyze cases where a retrieved plan was unhelpful, in order to improve the selection of retrieved cases in the future.

**Performance.** Nebel and Koehler's formal analysis of the complexity of plan adaptation [421] says that if certain conditions on the adaptation strategy are satisfied, plan adaptation can be exponentially harder than planning from scratch. At first glance, this result would seem to conflict with empirical studies, in which case-based planners based on derivational analogy have consistently outperformed the base-level generative planners on which the case-based planners were constructed [276, 405, 525].

However, a closer examination of the limitations of Nebel and Koehler's result resolves the conflict. In general, there may be several alternative ways to adapt a case to a new problem, and Nebel and Koehler's analysis requires the plan-adaptation strategy to find the adaptation that reuses as much of the case as possible. Au *et al.* [25] have shown (1) that derivational analogy does not satisfy this assumption, and hence the complexity result does not apply, and (2) that derivational analogy will never increase the size of the planner's search space and may potentially decrease it by an exponential amount.

## 24.2 Linear and Integer Programming

*Linear programs* (LPs) and *integer programs* (IPs) have a long history in the field of operations research and have been used to model a large number of problems in resource allocation, facility location, distribution, production, reliability, and design [422]. One potential advantage of using LP and IP techniques in planning is that they quite naturally allow the incorporation of numeric constraints and objectives into planning domains (e.g., [317]).

The general format for an LP or IP is

$$\text{minimize } CX \text{ subject to } AX \geq B,$$

where $A$ is an $(m \times n)$ matrix, $B$ is an $m$-dimensional column vector, $C$ is an $n$-dimensional row vector, and $X$ is an $n$-dimensional column vector. In an LP, $X$'s elements $x_1, \ldots, x_n$ may be any real numbers; in an IP, they must be integers; and in a mixed IP, some but not all of them must be integers. If the integer variables of an IP (or mixed IP) are required to be in $\{0, 1\}$, then it is an 0-1 IP (or mixed 0-1 IP). A vector $X$ which satisfies the constraints is called a *feasible solution.* If $X$ also minimizes the objective function $CX$, then it is called an *optimal solution* and $CX$ is called the *optimal value.*

Linear programming relies on the well-known Simplex technique and several recent algorithmic improvements that make it a polynomial and very efficiently solved problem, even for very large LP instances. LP is often used as a relaxation for an IP.

The standard technique for solving an IP involves a branch-and-bound search (see Appendix A) in which the bounding function is the LP relaxation of the IP. There are several solvers available for doing this; probably the best known of these is Cplex. The size of the problems that can be solved has increased by several orders of magnitude over the last few years. Current research on IPs generally involves exploring various ways to formulate other problem domains as IPs, to find formulations that increase the efficiency of problem solving.

Several recent works have explored ways to use LP and IP techniques for planning. Bylander [102] uses an LP formulation as a heuristic for classical plan-space planning, but it does not seem to perform well compared to planning-graph and satisfiability-based planners. Bockmayr and Dimopoulos [76, 77] describe domain-dependent IP models for specific classical planning domains, and Vossen *et al.* [539] compare two domain-independent ways to translate classical planning problems into IPs. Wolfman and Weld [556] use LP formulations for reasoning about resources in a satisfiability-based planner, and Kautz and Walser [317] use IP formulations for planning problems with resources, action costs, and complex objective functions.

One difficulty in developing IP formulations of planning problems is that the performance depends critically on *how* the planning problems are formulated as IPs. The easiest approach is to encode either the planning problem or a planning graph as a satisfiability problem (see Chapter 7) and then to express the satisfiability problem directly as an 0-1 IP. However, this approach does not compare well in efficiency with planning-graph or satisfiability-based planners. Better performance can be obtained by "compiling away" the original fluent variables and replacing them with "state-change" variables (not to be confused with state variables). In this way, Vossen *et al.* [539] obtained performance approaching that of Blackbox [316], a well-known classical planner that uses planning-graph and satisfiability techniques.

## 24.3 Multiagent Planning

Distributed and cooperative planning generalizes the problem of planning in domains where several agents plan and act together and have to share resources, activities, and goals. The problem, often called *multiagent planning* (MAP), is a major issue for the distributed AI and multiagents community. It arises in application areas such as multirobot environments, cooperating software (or softbots) distributed over the Internet, logistics, manufacturing, evacuation operations, and games.

Approaches to MAP vary depending on the type of planning problems and distribution processes. For example, one may consider a single planner that creates plans for several agents, and this planner may operate in either a single-agent [432] or multiagent [157] environment. Alternatively, there may be several coordinated planning processes local to each agent [167]. One may consider loosely coupled agents, each pursuing its own goal, while either competing for resources [495] or sharing them with others, or a team of agents pursuing a shared goal. The cooperation may rely on different assumptions about the agents' communication, perception, action, and computational capabilities and on their a priori knowledge. These capabilities can be either identical or distinct for the agents.

The plan-space paradigm offers a flexible framework for MAP. In this approach, planning evolves by modifying a current plan (considered to be a partial plan), aiming at completing it consistently by removing all its flaws. An agent may start planning from the plans of other agents, or it may modify its own plan with respect to the plans of others (e.g., repairing flaws due to conflicts). This scheme can be directly of use to MAP in the case where several agents share resources and a priori knowledge. A possible approach can use plan merging (see Section 24.4) and a coordination scheme where an agent generates a provisional plan and modifies it with respect to the set of plans whose coordination has already been achieved.

This coordination scheme has been instantiated into a fairly sophisticated multirobot system solving the DWR problem in a harbor [10]. It relies on a sound coordination protocol. It identifies coordination failures and the subset of robots concerned by failures by relying on global planning for these subsets, and it is complete as long as the robots' goals are not constrained in time. Other extensions to this system involve coordination with IxTeT, a planner handling time and resources [204], and coordination involving altruistic robots that have their own goals but may help each others [8, 86].

## 24.4 Plan Merging and Plan Rewriting

There are several cases in which it may be useful to decompose a planning problem into separate subproblems, plan for those subproblems separately, and merge the resulting plans. Here are some examples:

- In a multiagent planning situation, the agents may need to develop their plans separately.
- If a planning problem is decomposable into subproblems that are largely independent, then solving them separately may be more efficient than solving them together.

- If the subproblems are sufficiently different from each other, it may be best to solve them using separate domain-specific planning systems, such as a planner and a scheduler (see Chapter 15), or a manufacturing process planning system and a production planning system [117].

Plan merging often involves *operator merging*, i.e., merging some set of operators within a plan or set of plans into a single operator that achieves the same goals. The motivation for doing this is that if the operators can be merged, this may reduce the plan's cost, time, or resource requirements.

Domain-specific approaches for plan merging have been developed in some application domains, such as Hayes's Machinist system [263] for manufacturing process planning.

For domain-independent plan merging, the simplest situation in which plans for separate subproblems can be merged is if the subproblems are completely independent of each other. In this situation, Korf [336] has shown that solving each of the subgoals separately (and essentially concatenating the results) will divide both the base and the exponent of the complexity function by the number of subgoals.

Yang *et al.* [560] defined a set of allowable interactions among subproblems that are less severe than complete independence yet allow efficient plan merging to occur and developed a branch-and-bound algorithm for performing the merging. In addition, in Sacerdoti's NOAH system [461], some of the "critics" for improving plans can be viewed as operator-merging operations, as can phantomization of a goal in Wilkins's SIPE and Tate's NONLIN systems and in Kambhampati's plan-resuse framework.

Yang [560] describes a comprehensive theory for plan merging in classical planning and a dynamic-programming algorithm for plan merging based on their theory. This work was further elaborated and refined by Yang. Tsamardinos *et al.* [514] have extended this theory to use temporal networks similar to the ones described in Chapter 14.

Closely related to plan merging is *plan rewriting*, a hill-climbing approach in which a planner starts with a single solution to the planning problem and then repeatedly makes modifications to the solution in order to get better and better solutions. This idea was developed by Ambite and Knoblock [19]. In their experiments with this approach on a number of classical planning problems, it compared favorably with IPP [332], a well-known planning system based on planning graphs (Chapter 6).

## 24.5 Abstraction Hierarchies

In the AI planning literature, the phrase "planning with abstraction" has taken on a more specific meaning than a literal interpretation of the words might suggest. It refers to the following technique. Given a planning problem $\mathcal{P}_0$, first formulate a sequence of increasingly relaxed versions $\langle \mathcal{P}_1, \ldots, \mathcal{P}_k \rangle$ of $\mathcal{P}_0$. Next, find a solution $\pi_k$ for $\mathcal{P}_k$, then modify $\pi_k$ so that it is a solution $\pi_{k-1}$ for $\mathcal{P}_{k-1}$, and so forth until a solution has been found for $\mathcal{P}_0$. The motivation for doing this is that if each solution can be used to help find the next one, then solving the sequence of relaxations can sometimes be much less work than solving $\mathcal{P}_0$ from scratch.

Because there are many different ways to generate relaxations of a planning problem, "planning with abstraction" has been used variously to refer to the skeletal plans used in some case-based planning systems [61], the task networks used in HTN planning [560], and the notion of partitioning a planning problem recursively into smaller and smaller planning problems [352]. However, the most commonly used notion of abstraction in planning, and the subject of this section, is *precondition-elimination abstraction.*

In precondition-elimination abstraction, the relaxations of a classical planning problem $\mathcal{P}_0$ are produced by omitting preconditions from the operators in $\mathcal{P}_0$. The following technique is generally used to decide which preconditions to omit. To each literal $l$ that is a precondition of one or more planning operators in $\mathcal{P}_0$, assign a *criticality level* $c(l) \in \{1, \ldots, k\}$, where $k$ is a fixed positive integer. Then, let $\mathcal{P}_{k-i}$ be the planning problem produced from $\mathcal{P}_0$ by omitting every precondition $l$ such that $c(l) < k - i$.

Precondition-elimination abstraction can be implemented as a modification to any classical planning algorithm *Alg*, as follows. First, generate a sequence of relaxations $P_k = (O_k, s_0, g)$, $P_{k-1} = (O_{k-1}, s_0, g), \ldots, P_0 = (O_0, s_0, g)$ as described above. Next, try to solve $\mathcal{P}_k$ using *Alg*. If *Alg* finds a solution plan $\pi_k = \langle a_1, \ldots, a_n \rangle$, then let $\langle s_0, s_1, \ldots, s_n \rangle$ be the sequence of states produced by executing $\pi_k$ in $\mathcal{P}_k$. To modify $\pi_k$ to get a solution for $\mathcal{P}_{k-1} = (O_{k-1}, s_0, g)$, use *Alg* to find solutions for each of the planning problems $(O_2, s_0, s_1)$, $(O_2, s_1, s_2)$, $\ldots$, $(O_2, s_{n-1}, s_n)$, and insert these solutions in between the actions of $\pi_k$ to create a plan $\pi_{k-1}$ that solves $\mathcal{P}_{k-1}$. In the same manner, modify $\pi_{k-1}$ to get a solution $\pi_{k-2}$ for $\mathcal{P}_{k-2}$, $\pi_{k-2}$ to get a solution $\pi_{k-3}$ for $\mathcal{P}_{k-3}$, and so forth.

Precondition-elimination abstraction was first developed and used by Sacerdoti [459] to modify STRIPS (see Section 4.4) in order to produce his ABSTRIPS planner. The same technique was subsequently used by Yang *et al.* [561] to modify Chapman's TWEAK plan-space planner [120], producing a planning procedure called ABTWEAK.

Precondition-elimination abstraction is guaranteed to be sound because it terminates only after $\mathcal{P}$ has been solved. However, it is not necessarily complete. Knoblock *et al.* [326] defined a property called *downward monotonicity* which, if satisfied, ensures that precondition-elimination abstraction is complete. Subsequently, Knoblock [325] used this property as the basis of an algorithm for generating abstraction hierarchies. Successively better algorithms for this task were devised by Bacchus and Yang [37] and by Bundy *et al.* [98], and Garcia and Laborie [205] generalized the approach for use in temporal planning using a state-variable representation.

There has been some debate about how much benefit precondition-elimination abstraction provides, and both positive and negative examples can be found in the literature [40, 98, 485, 561]. For a summary and discussion of these, see Giunchiglia [233].

## 24.6 Domain Analysis

*Domain analysis* is the technique of analyzing a planning domain to gather information that may help a planner find solutions more quickly for problems in that domain. We describe several in this section.

**Typed Variables and State Invariants.** Most of the research on domain analysis has focused on automatic discovery of typed variables and state invariants in classical planning domains. A *typed variable* (see Section 2.4) is one that can have only a limited range of possible values, namely, the ones in its type. A *state invariant* is a property that is true of every state. For example, in the DWR domain, one state invariant is that for each robot $r$, there is exactly one location $l$ such that at($r, l$) holds.

If it is known that some variable has only a limited range of values or that certain properties must be true in every state, this information can improve the efficiency of some planning systems. For example, in a planning-graph planner (Chapter 6) it can reduce the size of the planning graph, and in a satisfiability-based planner (Chapter 7) it can reduce the size of the search space.

Even if data types or state invariants are not explicitly specified for a planning domain, they sometimes can be discovered through domain analysis. For example, in the classical representation of the DWR domain, consider all atoms of the form at($r, l$). In the initial state, suppose the only atoms of the form at($r, l$) are at(r1,l1) and at(r2,l2). The only planning operator that ever asserts at($r, \ldots$) is the move operator, which requires at($r, l$) as a precondition. Thus, in every state reachable from the initial state, at($r, l$) will be true only for $r \in$ {r1,r2}. Thus, $r \in$ {r1,r2} constitutes a data type for the first argument $r$ of at($r, l$).

Several procedures have been developed for discovering types and/or state invariants, particularly TIM [192] and DISCOPLAN [218, 219]. TIM is used as the domain-analysis module in STAN, a planning-graph planner [367]. DISCOPLAN has been used to provide input to both SATPLAN [314], a satisfiablity-based planner, and MEDIC [172]. In all three cases, the domain analysis led to significant speedups.

TALplanner, a control-rule planner (Chapter 10) uses domain-analysis techniques to discover state invariants. If a state invariant is discovered that guarantees some of its control rules will always be satisfied, then TALplanner can avoid the overhead of using those control rules [341].

**Other Kinds of Domain Analysis.** Smith and Peot [486, 487] developed algorithms for constructing *operator graphs* for classical planning domains. In plan-space planners, these graphs can be used to decide which threats can be ignored during the planning process and to detect some cases in which a branch in the search space will be infinite—both of which can significantly improve the efficiency of plan-space planning.

In HTN planning (Chapter 11), an *external condition* is a condition that is needed in order to achieve a task but is not achieved in any possible decomposition of the task. Tsuneto *et al.* [516] wrote a domain-analysis algorithm to find external conditions and modified the UMCP planner to plan how to achieve each task's external conditions before planning how to achieve the task. This made UMCP significantly faster.

Finally, the automatic generation of abstraction hierarchies, discussed in Section 24.5, is a kind of domain analysis.

## 24.7 Planning and Learning

Two of the earliest systems to integrate machine learning and planning were SOAR [348], a general cognitive architecture for developing systems that exhibit intelligent behavior, and PRODIGY [396], an architecture that integrates planning and learning in its several modules [524].

Much of the work done on the integration of learning and planning is focused on classical planning. Usually, this work, as formulated by Minton [391], learns search-control rules to speed up the plan generation process or to increase quality of the generated plans. These rules give the planner knowledge to help it decide at choice points and include *selection rules* (i.e., rules that recommend using an operator in a specific situation), *rejection rules* (i.e., rules that recommend not using an operator in a specific situation or avoiding a world state), and *preference rules* (i.e., rules that indicate some operators are preferable in specific situations). As mentioned by Langley [351], the input for this kind of learning generally consists of partial given knowledge of a problem-solving domain and a set of experiences gained through search of the problem's search space.

Mitchell *et al.* [398] first suggested the use of *learning apprentices*, which acquire their knowledge by observing a domain expert solving a problem, as control rule learning algorithms. EBL has been used to induce control rules [390]. STATIC [182] uses a graph representation of problem spaces to elicit control rules based on analysis of the domain. Katukam and Kambhampati [309] discuss the induction of explanation-based control rules in partial-order planning. Leckie and Zukerman [358] use inductive methods to learn search-control rules. SCOPE [179] learns domain-specific control rules for a partial-order planner that improve both planning efficiency and plan quality [181] and uses both EBL and Inductive Logic Programming techniques.

There has been some recent work on applying various learning algorithms in order to induce task hierarchies. Garland *et al.* [207] use a technique called *programming by demonstration* to build a system in which a domain expert performs a task by executing actions and then reviews and annotates a log of the actions. This information is then used to learn hierarchical task models. KnoMic [522] is a learning-by-observation system that extracts knowledge from observations of an expert performing a task and generalizes this knowledge to a hierarchy of rules.

These rules are then used by an agent to perform the same task. CaMeL [278] learns preconditions for HTN methods using a modified version of candidate elimination.

Another aspect concerning the integration of planning and learning is *automatic domain knowledge acquisition*. In this framework, the planner does not have the full definition of the planning domain and tries to learn this definition by experimentation. Gil [227, 229] introduces a dynamic environment in which the preconditions or effects of operators change over time and discusses methods to derive these preconditions and effects dynamically. In another work by Gil [228], instead of revising existing operators, new operators are acquired by direct analogy with existing operators, decomposition of monolithic operators into meaningful suboperators, and experimentation with partially specified operators.

## 24.8 Planning and Acting, Situated Planning, and Dynamic Planning

In this book we focused mainly on the problem of plan generation. However, most real-world applications require systems with *situated planning* capabilities, i.e., systems that interleave planning with acting, execution monitoring, failure recovery, plan supervision, plan revision, and replanning mechanisms. Several situated planners have been proposed so far [4, 59, 190, 215, 411, 478] and have been successfully applied in particular application domains (like mobile robots and fault diagnosis for real-time systems). All these systems address the issue of integrating reactivity and reasoning capabilities, even if each of them uses quite different techniques.

Firby [191] introduced *reactive action packages* (RAP), which are programs that run until either the goal is reached or a failure occurs. The RAP interpreter refines tasks into more primitive commands and controls activation and conflict resolution. RAPs describe how to achieve a given task. Their semantics are strictly oriented toward task achievement and sequencing subtasks. The related system [190] implements a "partitioned architecture" where a strategic planner interacts with a completely reactive system (RAP Executor) through a shared world model and plan representation.

Going further toward the integration of planning and reacting, Lyons and Hendriks [372] developed a system called RS, where planning is seen as a permanent adaptation of a reactive process. The latter is a set of rules which is modified according to the context and goals. This has been tested for the design of an assembly robot [371]. Similarly, Bresina [93] proposes in the ERE system a planner that is able to synthesize new situated control rules when a failure situation is met.

3T [79] is a three-layer architecture. It comprises a set of skills and uses the RAP system in one of its layers to sequence them. One layer is a planning system that reasons on goal achievement, including timing constraints. XFRM [59], depending on the time constraints, executes either default plans or new plans obtained by transforming the default plans by means of heuristic rules.

The Task Control Architecture (TCA), developed by Simmons [478, 479], organizes processing modules around a central controller that coordinates their interactions. A goal is decomposed into a task tree with subgoals that are achieved by the decentralized modules. All communication is supported by the centralized control. TCA provides control mechanisms for task decomposition and takes into account temporal constraints in task scheduling. TCA is based on plans (called *task trees*) that provide a language for representing the various planning activities: plan formation, plan execution, information acquisition, and monitoring. The system can flexibly perform interleaving planning and execution, run-time changing of a plan, and coordinating multiple tasks. The TCA exception-handling facilities support context-dependent error recovery because different error handlers can be associated with different nodes in the task tree.

In Georgeff and Lansky's Procedural Reasoning System (PRS) [215], plans (called *KAs*) describe how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations. Metalevel KAs encode various methods for choosing among multiple applicable KAs. They provide a high amount of flexibility in forming plans. Despouys and Ingrand [154] integrate a planner on top of a PRS-like system.

Finally, several methods interleave planning and execution in order to deal with large state spaces, among which most notably is the method described by Koenig and Simmons [333], an approach based on Min-Max LRTA*. These methods open up the possibility of dealing with large state spaces. However, these methods cannot guarantee to find a solution, unless assumptions are made about the domain. For instance, Koenig and Simmons [333] assume "safely explorable domains" without cycles.

## 24.9 Plan Recognition

Plan recognition is the inverse problem of plan synthesis. The problem involves two characters: an *actor* and an *observer*. Given as input a sequence of actions executed by the actor, the observer has to map this observed sequence to the plan currently followed by the actor and ultimately to its goal. The problem arises in many application areas, such as natural-language processing and story understanding [548], psychological modeling and cognitive science [467], intelligent computer interfaces [140, 439], surveillance and supervision systems [91, 92, 266], plan execution monitoring [154], and multiagent cooperative planning.

Plan recognition arises in different settings. For example, the actor can cooperate with the observer and try to communicate his or her intentions clearly.

This problem is called the *intentional plan recognition problem.* It is of interest in particular for multiagent cooperative planning. In another setting, the actor is in an adverserial framework, trying to hide his or her intentions, e.g., in surveillance and military applications. Another setting has a neutral actor who ignores the observer. This *keyhole plan recognition problem* is the most widely studied case of plan recognition.

Approaches to plan recognition depend also on various assumptions, such as the following.

- *The a priori knowledge available to the observer.* Some possibilities include these.
  - The observer has complete knowledge of the actor's set of plans. It needs only to map the observed sequences to these plans.
  - The observer has complete knowledge of the actor's models of actions (its planning domain), and it may need to synthesize new plans that account for the observations.
  - The available knowledge is partial and/or uncertain.
- *The observations given as input.* These can be either the actor's actions or the effects of these actions on the world. In the latter case, these effects can be mixed with other unrelated events that are also observed. Moreover, observations may be reliable and complete or partial and erroneous.
- *The temporal assumptions about the actions models.* Examples of these assumptions include that time is explicit and the observations are time-stamped and/or the temporal assumptions about the plan recognition process itself (e.g., the observer is online with the actor and has real-time constraints).

In the simple case of keyhole plan recognition with complete knowledge of the actor's plans, and where the observations are directly the actor's actions, the popular approach of Kautz and Allen [311], well formalized in Kautz [310], relies on a hierarchical model of plans. This model is quite similar to the ordered task model of Chapter 11, where a task can be achieved by a disjunction of possible methods, and each method decomposes a task into a sequence of subtasks, down to primitive actions. The set of root tasks (tasks that are not subtasks of any other tasks) corresponds to the possible goals the actor may pursue.

From the observation of a sequence of primitive actions, one can find the set of possible goals of the actor that account for these actions: the decomposition of each goal in this set starts with the observed sequence. This set is called the *covering model* of the observed actions. Initially, when the observed sequence is empty, this set contains all known goals, then it decreases along with newly observed actions, and it may possibly become empty. A problem arises if the actor is pursuing more than one goal at a time. This problem is addressed with *minimal covering models* where each explanation is a minimal set of goals not subsumed by some other set.

Hence a single goal explanation, even if less likely, is preferred to several concurrent goals.[1]

Other approaches to plan recognition are more concerned with handling uncertainty in models and/or observations. They take a pattern recognition point of view, where one has to map some observations to an interpretation, with an explanation of why this interpretation accounts for the observations, and the likelihood of that interpretation. Among the proposed techniques are the Bayesian inference approach of Charniak and Goldman [121] and the abductive probabilist theory of Goldman *et al.* [239], which takes into account more properly negative evidence.

A third class of approaches to the plan recognition problem, called the *chronicle recognition problem*, emphasizes temporal models, low-level observations, and online recognition [161]. The given knowledge is a set of chronicle models. A chronicle, as defined in Chapter 14, is a set of temporal assertions, i.e., events and persistence conditions on values of state variables, together with domain and temporal constraints. The observer's input is a possibly infinite sequence of time-stamped, instantiated events, which can be the effects of the actor's actions or elements of the environment's dynamics. The chronicle models are supposed to describe the actor's plans *and* the environment's dynamics that are relevant to these plans. The observer does not need to account for all observed events but only for those that meet a possible instance of a chronicle model.

In the simplest case, the observations are assumed to be complete and reliable, i.e., any change in a state variable is supposed to be observed and all observations correspond to occurred changes. The chronicle recognition system has to detect online any subset of the sequence of observed events that meets a chronicle instance. The problem can be addressed as a formal language-recognition problem with temporal constraints [222]. An observed event may trigger a hypothesis of a chronicle instance. This hypothesis corresponds to a set of predicted events and their temporal windows of possible occurrence and associated deadline.

The hypothesis may progress in two ways: (1) a new event may be detected that can either be integrated into the instance and make the remaining predictions more precise or may violate a constraint for an assertion and make the corresponding hypothesis invalid; or (2) time passes without anything happening and, perhaps, may make some deadline violated or some assertion constraints obsolete. For each chronicle model, one needs to manage a tree of hypotheses of current instances. When a hypothesis of a chronicle instance is completed or killed (because of a violated constraint), it is removed from this tree. Because of the temporal constraint, the size of the hypothesis tree is bounded.

The chronicle recognition techniques have been extended to take into account uncertainty and partial observation (e.g., [246]). They proved to be quite effective in addressing supervision problems in various application areas, ranging from gas

1. In Kautz's example [310], going hunting *and* cashing a check at a bank can be more likely than robbing the bank, two possible explanations for the same set of observations "about a person in a bank with a gun."

turbines and blast furnaces to power networks and telecommunication networks. They have also been used in surveillance applications [266] and in the execution monitoring of plans, where the observer is also the actor [154].

## 24.10 Suggestions for Future Work

Clearly, more effort is needed on the development of planning from an engineering perspective. More effort is also needed to integrate various approaches to automated planning, especially the integration of planning and scheduling and the integration of planning and acting. These integration efforts should be not merely from an engineering perspective but also at the levels of knowledge representation and domain modeling and should involve consistent integration of several problem-solving techniques. We hope that future planning books will describe the results of such work, and we encourage the reader to contribute to their development.