

Acquisition of Abstract Plan Descriptions for Plan Recognition

Mathias Bauer

German Research Center for Artificial Intelligence (DFKI)

Stuhlsatzenhausweg 3

66123 Saarbrücken, Germany

bauer@dfki.de

Abstract

While most plan recognition systems make use of a plan library that contains the set of available plan hypotheses, little effort has been devoted to the question of how to create such a library. This problem is particularly difficult to deal with when only little domain knowledge is available—a common situation when e.g. developing a help system for an already existing software system. This paper describes how operational decompositions of plans can be extracted from a set of sample action sequences, thus providing the basis for automating the acquisition of plan libraries. Efficient algorithms for the approximation of optimal decompositions and experimental results supporting their feasibility are presented.

Introduction

Most plan recognition systems use a predefined set of plans—described on some level of abstraction—to delimit the search space of potential plan hypotheses. In these cases plan recognition mainly amounts to determining a plan entry that covers the currently observed action sequence as one of its concrete instances. So far, relatively little work has been devoted to the question of how to actually *construct* such a plan library (Mooney 1988; Lesh & Etzioni 1996).

Creating an adequate representation of the plans in a given domain is a nontrivial task. Even when leaving aside completeness and optimality considerations, the knowledge engineer still has to find a concise plan representation supporting the recognition process. This is particularly difficult when sufficient domain knowledge about the actions and objects to be manipulated is not readily available, e.g. when a help system is to be developed for an already existing software system and the semantics of the various commands has to be determined a posteriori. This paper presents an approach to the automatic acquisition of abstract plan descriptions from logged action traces. The algorithms presented even work in the complete absence of domain knowledge, but can also make use of such information if provided. Based on the assumption that an element

of a plan—i.e. an action or a constraint describing its internal structure—is *necessary* for its correct execution just in case it occurs in *all* possible action sequences forming an instance of this plan, the system tries to detect the common aspects of a number of sample logs at some level of abstraction, thereby incrementally establishing an intensional representation of a class of action sequences containing all the samples provided.

The rest of this paper is organized as follows. The next section characterizes a simple representation formalism for plans. Then the basic algorithms are introduced and empirical results supporting their feasibility are presented. Finally, related work is reviewed before the results are summarized and future work is discussed.

Plan Representation

The most basic step in plan recognition is the mapping of observed actions onto the available plan hypotheses. The important information on which actions form the elementary steps of a plan—that is, the “recipe” of how to achieve the associated goal—is contained in the so-called *plan decompositions*.¹ Obviously, a *good* plan representation covers all the essential conditions that have to be met by an observed action sequence in order to be recognized as belonging to a particular class of activities leading to some specific domain goal. Examples of such conditions include the actions involved, their temporal ordering, and structural relationships among the objects being manipulated. However, the description of a plan should not be excessively restrictive by containing conditions that are not crucial to the successful achievement of its associated goal. To summarize, an ideal plan decomposition is restrictive enough to discriminate competing hypotheses without limiting the whole bandwidth of possible behaviors subsumed.

Plans or *plan decompositions* are represented as tuples $\langle p, A_p, C_p \rangle$, where p is the name of a plan, A_p is a set of actions—either primitive or abstract—and C_p is a set of

¹In the following, the notions of “plan” and “plan decomposition” will often be used as synonyms.

constraints regarding these actions such as conditions about temporal relationships among elements of A_p or restrictions regarding the objects manipulated by these actions. So, the complete information on how a given plan can be executed is represented without restricting it to one possible action sequence: p might, for example, describe a *non-linear plan* where the temporal ordering of the actions is only partial and thus allows for a number of concrete execution instances. Further types of abstraction contained in a plan decomposition include abstract actions and variable action arguments instead of domain constants.

Remark: This notion of a plan does not include preconditions or effects. One the one hand, these cannot be inferred without detailed knowledge about the semantics of actions. On the other hand, they are not absolutely required for plan recognition purposes—as is the case here—where the focus is on representing features that support the identification of user goals on the basis of observed user actions only.

Acquisition of Plan Decompositions

For the following discussion the existence of three kinds of domain knowledge is assumed. D is a logical theory describing general aspects of the given domain like information about structure and types of domain objects. Additionally D_t and D_a represent an *object type* and *action type hierarchy*, resp., containing information about abstraction relations between the types of domain objects and actions.²

The operation msa yielding the most specific *non-trivial* common abstraction of a set of concepts (either object types or action types) is defined on D_t and D_a . Let t_1, \dots, t_n be object types. Then $msa(\{t_1, \dots, t_n\}) =$

$$\begin{cases} t', & \text{if } mss(\{t_1, \dots, t_n\}) = \{t'\} \text{ and } t' \neq \text{root}(D_t) \\ msa(\{t'_1, \dots, t'_m\}), & \text{if } mss(\{t_1, \dots, t_n\}) = \{t'_1, \dots, t'_m\} \\ & \text{and } m > 1 \\ \text{undefined,} & \text{otherwise} \end{cases}$$

where mss is the *most specific subsumer* operation (Woods & Schmolze 1991) computing the set of least general concepts subsuming the disjunction of concepts contained in its argument (the same applies to D_a). If no hierarchical information is available (i.e. if D_t or $D_a = \emptyset$), then $msa(\{x_1, \dots, x_n\}) = x_1$ if $x_i = x_1$ for all $1 \leq i \leq n$. Otherwise its value is undefined.

As will become clear from the following discussion, any component of the potentially available domain knowledge may be empty, that is, the mechanism to be presented will also work without additional information.

²It is important to distinguish between the notions of an *action* or *action instance* and an *action type*. In the UNIX domain, for example, “cp paper.tex /tmp” is a concrete, observable instance of the action type cp. In this case the file paper.tex is copied to another directory.

Input Data

Assume a number of test subjects (e.g. software testers, students) are given a set of goals g_1, \dots, g_m to be achieved. Their attempts in doing so are recorded and grouped according to common goals. Let AS_1, \dots, AS_N be the set of action sequences belonging to some goal g . Each of these sequences consists of a set of temporally ordered actions $\hat{a}_1, \dots, \hat{a}_l$ where $\hat{a}_i = a_i(o_{i1}, \dots, o_{in_i})$. Here a_i is the *action type* this particular observed instance belongs to (e.g. ls or cd in a UNIX context), and the various o_{ij} are constants representing the domain objects being manipulated (the action parameters). In the current version, the actions are assumed to be represented in a canonical way, i.e. with a fixed number of parameters for each action type.

In a first step, these action sequences are transformed into labeled graphs G_{AS_i} making the interrelationships among the constituents of AS_i explicit. While the action instances contained in AS_i form the nodes, there are two types of edges. An edge $\hat{a}_i \xrightarrow{<} \hat{a}_j$ represents the temporal order between both actions (in this case \hat{a}_i occurs before \hat{a}_j), while an edge $a_i(o_{i1}, \dots, o_{in_i}) \xrightarrow{\langle p, k, l \rangle} a_j(o_{j1}, \dots, o_{jn_j})$ represents the fact that relation p holds between the action arguments o_{ik} and o_{jl} . The resulting *action graph* then has the form $G_{AS_i} = \langle AS_i, T_i \cup S_i \rangle$ with a set of *temporal edges* $T_i = \{\hat{a}_i \xrightarrow{<} \hat{a}_j \mid i < j\}$ and a set of *structural edges* $S_i = \{\hat{a}_i \xrightarrow{\langle p, k, l \rangle} \hat{a}_j \mid D \models p(o_{ik}, o_{jl})\}$. If $D = \emptyset$, i.e. if no domain knowledge is provided, only the equality of two objects can be recognized and made explicit within S_i .

Example: If $\hat{a}_1 = ls(/opt)$ ³ and $\hat{a}_2 = cd(/opt/bin)$ then—assuming availability of the corresponding domain knowledge—an edge between \hat{a}_2 and \hat{a}_1 labeled $\langle \text{subdir_of}, 1, 1 \rangle$ indicating that the first argument of \hat{a}_2 is a subdirectory of the first argument of \hat{a}_1 can be established.

The Abstraction Mechanism

As already mentioned, a good plan representation is limited to those elements that are absolutely necessary in order to reach some specific goal and thus have to be contained in *each* concrete execution of this plan. Given only a finite set of sample action sequences, the necessity of an element can only be approximately determined; if it occurs in all sequences *available*, it *might* be necessary. As a consequence, an approximation of the “ideal” plan decomposition can be created by determining those aspects—actions, temporal and structural relations—that are common to all samples and representing them at some level of abstraction. To this end, the components of two action graphs are compared pairwise and—provided they satisfy some compatibility condition—are included in the result in an abstract

³This is the internal representation of the UNIX shell command “ls /opt” displaying the contents of directory /opt.

form. The *join* operator defined in the following performs the required tests.

Let $G_1 = \langle A_1, T_1 \cup S_1 \rangle$ and $G_2 = \langle A_2, T_2 \cup S_2 \rangle$ be two action graphs. Let $\hat{a}_i = a_i(o_{i1}, \dots, o_{in}) \in A_i, i = 1, 2$. Then $join(\hat{a}_1, \hat{a}_2) = a(o_1, \dots, o_n)$ where $a = msa(\{a_1, a_2\})$ and for all $1 \leq i \leq n$:

- $o_{1i} = o_{2i} = o_i$ is a constant or
- o_i is a variable of type $msa(\{type(o_{1i}), type(o_{2i})\})$.

That is, the abstract representation of the common information contained in two action instances \hat{a}_1 and \hat{a}_2 is a new action instance the type of which is the *msa* of both action types—if defined—and the arguments of which are either a concrete domain object represented by a constant o_i (if both actions had the same object as a parameter in the same place) or a newly introduced variable the type of which is determined from the types of the originally occurring domain action arguments.

A temporal or structural edge is considered to be common to both G_1 and G_2 iff the corresponding action nodes to which the edge is incident could be joined using the above criterion. That is,

$$join(\hat{a}_{11} \xrightarrow{\leq} \hat{a}_{12}, \hat{a}_{21} \xrightarrow{\leq} \hat{a}_{22}) = \hat{a}_1 \xrightarrow{\leq} \hat{a}_2$$

if $join(\hat{a}_{11}, \hat{a}_{21}) = \hat{a}_1$ and $join(\hat{a}_{12}, \hat{a}_{22}) = \hat{a}_2$ and undefined otherwise. The join of two structural edges with identical labels is defined analogously.

Summarizing, the abstraction of two action sequences represented as action graphs G_1 and G_2 is $join(G_1, G_2) = \langle A, T \cup S \rangle$ where

$$\begin{aligned} A &= \{join(\hat{a}_{1i}, \hat{a}_{2j}) \mid \hat{a}_{1i} \in A_1, \hat{a}_{2j} \in A_2\} \\ T &= \{join(e_{t1}, e_{t2}) \mid e_{t1} \in T_1, e_{t2} \in T_2\} \\ S &= \{join(e_{s1}, e_{s2}) \mid e_{s1} \in S_1, e_{s2} \in S_2\} \end{aligned}$$

Remarks: 1. The *join* operator is associative and commutative. This is important for the definition of an incremental procedure to the acquisition of plan decompositions. Its complexity is $O(|A_1| \cdot |A_2| + |T_1| \cdot |T_2| + |S_1| \cdot |S_2|)$.

2. It is straightforward to transform $join(G_1, G_2)$ into a plan decomposition by simply reversing the preprocessing step described above.

Example: Assume the action sequences AS_1 and AS_2 as depicted in Figure 1 are given. For sake of simplicity, only structural and direct temporal edges connecting subsequent actions are shown. That is, temporal edges like the one between a1 and a3 are left out. Action arguments like “spag-1” refer to particular domain objects. The domain knowledge available is limited to an action hierarchy containing the relation tuples $\langle make_spaghetti, make_pasta \rangle$, $\langle make_fettucini, make_pasta \rangle$, $\langle make_pesto, make_sauce \rangle$, and $\langle make_marinara, make_sauce \rangle$ which introduce the abstract actions *make_pasta* and *make_sauce* each of which

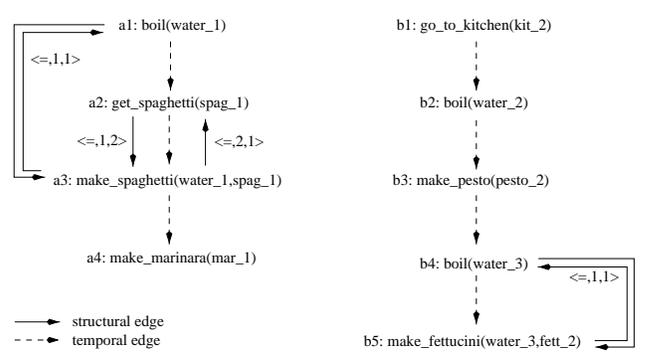


Figure 1: Two action sequences from the “cooking world”. subsumes two of the actions contained in the sample sequences. The following action joins can be computed using the above operator.⁴

$$\begin{aligned} \alpha_1: & \text{join}(a1, b2) = \text{boil}(x_1) \\ \alpha_2: & \text{join}(a1, b4) = \text{boil}(x_2) \\ \alpha_3: & \text{join}(a3, b5) = \text{make_pasta}(x_3, x_4) \\ \alpha_4: & \text{join}(a4, b3) = \text{make_sauce}(x_5) \end{aligned}$$

Additionally the following temporal and structural edges are contained in the join of G_{AS_1} and G_{AS_2} .

$$\begin{aligned} \alpha_1 &\xrightarrow{\leq} \alpha_4 & \alpha_1 &\xrightarrow{\leq} \alpha_3 & \alpha_2 &\xrightarrow{\leq} \alpha_3 \\ \alpha_2 &\xrightarrow{\langle =, 1, 1 \rangle} \alpha_3 & \alpha_3 &\xrightarrow{\langle =, 1, 1 \rangle} \alpha_2 \end{aligned}$$

Let’s have a closer look at the last temporal edge. Action a1 occurred before a3 in AS_1 and b4 was before b5 in AS_2 . Joining a1 with b4 and a3 with b5 yielded α_2 and α_3 , resp. As a consequence, the temporal relationship between the boiling of water and the respective making of pasta—the water must be boiled *before* the noodles can be cooked—could be recognized as common to both action sequences.

As obviously at most one boil action is necessary—otherwise AS_1 would have contained more than one occurrence—it is not desirable to include both α_1 and α_2 in the plan decomposition. Generally speaking, each action may be joined with at most one action from the other sequence.

Definition 1: A *valid join* of two action graphs $G_1 = \langle A_1, T_1 \cup S_1 \rangle$ and $G_2 = \langle A_2, T_2 \cup S_2 \rangle$ is a maximum subgraph $\langle A_v, T_v \cup S_v \rangle$ of $join(G_1, G_2)$ such that $\forall a_1 \in A_1, a_2 \in A_2 : \text{if } join(a_1, a_2) \in A_v \text{ then } \forall a'_1 \in A_1, a'_2 \in A_2 : join(a'_1, a_2) \notin A_v \text{ and } join(a_1, a'_2) \notin A_v$. \square

Example (contd.): Two valid joins exist in the above example:

$$\begin{aligned} G_{v1} &= \langle \{ \alpha_1, \alpha_3, \alpha_4 \}, \{ \alpha_1 \xrightarrow{\leq} \alpha_4, \alpha_1 \xrightarrow{\leq} \alpha_3 \} \rangle \\ G_{v2} &= \langle \{ \alpha_2, \alpha_3, \alpha_4 \}, \\ &\quad \{ \alpha_2 \xrightarrow{\leq} \alpha_3, \alpha_2 \xrightarrow{\langle =, 1, 1 \rangle} \alpha_3, \alpha_3 \xrightarrow{\langle =, 1, 1 \rangle} \alpha_2 \} \rangle \end{aligned}$$

⁴The various x_i denote variables that are untyped due to missing type information.

While G_{v1} represents more temporal information than G_{v2} , the latter additionally reflects the structural interdependencies between the boil and the make_pasta actions by making explicit the fact that the previously boiled water is exactly the one used for cooking the noodles.

- Remarks:** 1. Each valid join of a given pair of action sequences has the same number of actions.
2. Plan libraries of existing plan recognition systems often contain alternative decompositions for single plans when the associated goals can be achieved in a number of ways differing significantly from each other. As a consequence action sequences observed during the execution of these alternatives will have little in common, that is, applying the join operator is likely to produce trivial results containing only a small number of common concepts. To deal with this problem, a similarity test between the action graphs currently under consideration is performed. The join will be computed just in case the percentage of actions that can be joined exceeds a certain threshold for both sequences.
3. As each valid join is an action graph itself, the join operator can also be used to form a common abstraction of the results of two or more previous steps. This way, to use the well-known “cooking world” example from (Kautz 1991), joining the abstract plan descriptions for making pasta and meat dishes, resp., the even more abstract plan of preparing a meal can be derived without having to restart the whole process beginning with the sample action sequences.

Abstraction Heuristics

In realistic applications like command traces from the UNIX domain with 20 or more steps, the number of valid joins quickly becomes intractable. So a criterion is needed to characterize the various alternatives and efficiently compute the desired variant without enumerating all possibilities. Depending on the application context in which the plan recognition system will work, at least two interesting alternatives exist. If, for example, the system is to monitor an agent's activity in order to recognize suboptimal behavior, it is important to detect deviations from the intended way at an early stage. To this end, it is important for the plan decompositions to be as restrictive as possible. In a help system context, however, it may be desirable to be more lenient towards spurious actions of the user as long as his/her acting is somehow related to the hypothesized plan. In this case the plan decompositions should contain as few constraints as possible in order to cover a wide spectrum of possible user behaviors.

The *degree of restrictiveness* of a candidate plan description can be quantified using the function

$$deg_r(\langle A, T \cup S \rangle) = w_a \cdot |A| + w_p \cdot |PA| + w_t \cdot |T| + w_s \cdot |S|.$$

$PA \subseteq A$ is the set of *primitive actions* contained in A , i.e. those that do not abstract another domain action in the abstraction hierarchy D_a . If $D_a = \emptyset$, PA and A are obviously

identical. The *non-negative* numerical weights w_a , w_p , w_t , and w_s assess the relative influence of the various components of a plan description. The degree of restrictiveness deg_r monotonically decreases with each join operation.

Example (contd.): Assuming the values $w_a = 1$, $w_p = 1$, $w_t = 1$, and $w_s = 2$, the resulting values for the valid joins G_{v1} and G_{v2} as computed above are

$$deg_r(G_{v1}) = 6, \quad deg_r(G_{v2}) = 9.$$

Here structural constraints are considered more important than temporal ones when comparing the restrictiveness of various decompositions. So the most restrictive plan decomposition is G_{v2} which represents the partially ordered abstract plan with the actions $boil(x)$, $make_pasta(x,y)$, and $make_sauce(z)$ and the additional conditions that the first occurs before the second and both share their first action parameter.

The optimal choice according to maximum/minimum value of deg_r can be reliably approximated using two heuristics. After computing the join of the action graphs G_1 and G_2 , the resulting nodes are numerically assessed using the same parameters as in deg_r :

$$deg_n(a) = w_a + w_p \cdot \delta_a + w_t \cdot |T_a| + w_s \cdot |S_a|. \quad (1)$$

Here δ_a is 1 if a contains a primitive action and 0 otherwise. T_a (S_a) is the subset of temporal (structural) edges of $join(G_1, G_2)$ incident to a . For the first heuristic, APL^+ , approximating the selection of the most restrictive valid join, all action nodes of $join(G_1, G_2)$ are sorted according to *decreasing* value of (1). Then action nodes are taken from this ordered list starting with the highest deg_n values and added to the result without violating the condition of unique action joins from Definition 1. Eventually, the corresponding temporal and structural edges are added.

For the second heuristic, APL^- , approximating the selection of the least restrictive valid join the action nodes are sorted according to *increasing* values of (1), that is, those with the smallest deg_n values are added first.

Experimental Results

The performance of both heuristics can be read from Figures 2 and 3. In the first case the test data were UNIX command sequences mainly collected at the University of Washington containing up to 24 steps. For these tests, 8 collections each containing 11 sequences with identical associated goals were joined. This was repeated 10 times with randomly permuted input sets. The average degree of restrictiveness after each join is depicted relative to the maximum value that could be reached after 10 joins (this value is depicted as 100%). The “max” and “min” curves represent the respective deg_r values of the actually most and least restrictive valid joins (that is, the final value of “max” corresponds to 100%). Obviously both APL^+ and APL^-

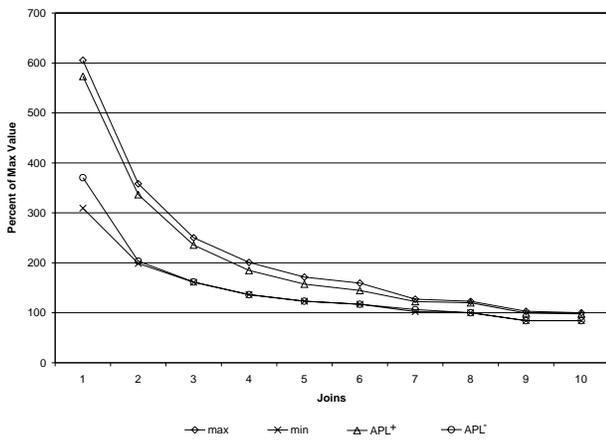


Figure 2: Performance of APL^+ and APL^- in the UNIX domain.

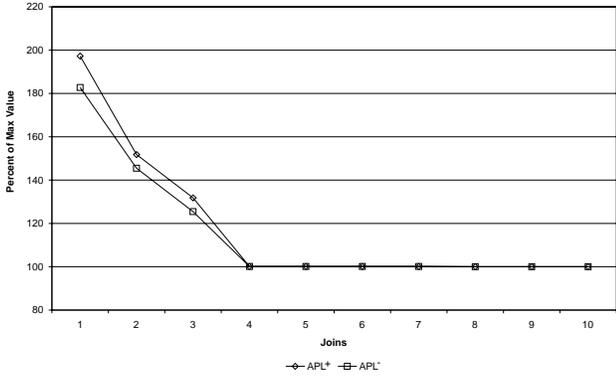


Figure 3: Performance of APL^+ and APL^- in the cooking world.

yield almost optimal results. Additionally, this figure shows that the observed action sequences contained a huge number of spurious commands (in this case mostly “sensory” commands like $1s$) that could be “abstracted away”.⁵

In the second case, identical tests as above were performed using artificially generated action sequences for the 5 basic and 4 abstract goals contained in the “cooking-world” library (Kautz 1991, Section 2.2.2) including spurious actions like leaving the kitchen to answer the phone. These sequences contained about 7 actions on average. The number of valid joins turned out to be much smaller than in the UNIX domain.⁶ As a consequence, both APL^+ and APL^- produce optimal results, i.e. their curves in Figure 3 are identical to the max and min curves, resp. Additionally, the optimum value (corresponding to the 100% line) is reached after only 4 joins.

⁵The average maximum degree of restrictiveness after 10 joins is only about one sixth of the value after the first join.

⁶This is mostly due to the fact that activities in the kitchen domain were more goal-directed as they did not involve a lengthy “navigation phase” as was the case in the first test.

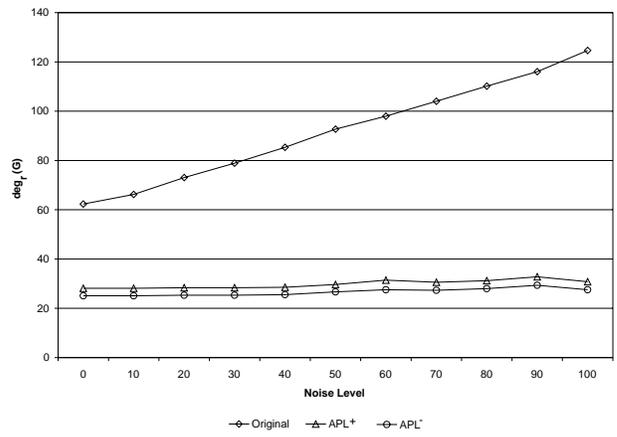


Figure 4: Impact of noise on APL^+ and APL^- .

Figure 4 depicts the impact of noise on the performance of both heuristics in the UNIX case. The same data as above were enhanced by a certain percentage—ranging from 0 to 100—of spurious actions and corresponding temporal and structural constraints.⁷ This is reflected in the upper-most curve representing the average deg_r values of the modified sample sequences. The other curves depict the average deg_r values after 10 joins as in the experiment described above. As could be expected, both heuristics are only minimally affected as elements not occurring in *all* input sequences are filtered out.

Besides these quantitative aspects, the *quality* of plan decompositions produced may not be neglected. Applying APL^+ to the cooking world sequences yielded results that in most cases were equivalent to the decompositions presented in (Kautz 1991). The number of samples required to reach this level of precision was between 4 and 10. First experiments in both domains suggest that the acquired plans can indeed be successfully used by a plan recognition system (Bauer 1996).

Remark: An additional speedup can be reached by sorting the input sequences according to their lengths—the number of actions occurring—and first joining the shortest sequences.

Related Work

There has been a number of approaches devoted to generalizing problem solutions to create plans to be reused for efficient planning from second principles (e.g. (Minton 1985)). As a consequence an exact characterization of their preconditions and effects is required that has to be extracted from the semantics of the domain actions. In the context of the approach discussed in this paper—acquisition of discriminating schemata for plan recognition in scarcely modeled

⁷That is, in the highest noise level, each action sequence contained the double number of actions and constraints compared to the original.

domains—this knowledge is usually not available.

(Yoshida & Motoda 1995) describes how a graph representing a UNIX user's typical behavior can be learned from sample action sequences. *Graph-based induction* identifies common patterns in these traces. Using transition probabilities between the various nodes (actions), the user's next command can be predicted. However, only the use of deep domain knowledge guarantees a high prediction accuracy. While the notion of *plans* plays no role, these might be extracted by following the most probable paths through the graph.

(Bauer 1996) describes how both qualitative and quantitative information about a user's typical reaction to certain situations can be gained from an interaction history. The resulting model of the user's *preferences* is used to improve focusing during the plan recognition process. One drawback is that, as usual, the existence of a fixed plan library is assumed without indicating how it might be constructed.

In (Albrecht *et al.* 1997) dynamic Bayesian networks are used to map the behaviors of players in a multi-user dungeon to the goals (“quests”) currently being pursued. While the set of possible goals can be completely enumerated, the enormous number of actions and places renders any attempt to create a complete domain model futile. As a consequence, there is no such thing as an operational description of the various possible ways to achieve a particular goal. Instead, given a set of sample action sequences, the current quest is predicted on the basis of statistical correlations between the previous quest and the current location and action. This method demonstrates its feasibility in a scenario where there is no standard way to reach a particular goal state. In a help system context, however, it is of little use to merely recognize the intended plan/goal if the system cannot give hints on what has to be done next.

(Mooney 1988) uses explanation-based learning (EBL) to generalize the plans underlying short narratives. As EBL is a knowledge-intensive technique, its feasibility is limited to well-formalized domains.

In (Lesh & Etzioni 1996) the goal and plan libraries are only *implicitly* defined by a set of goal predicates and actions and a corresponding goal and plan *bias* that allow the search space to be enumerated. While this enables the application of efficient search algorithms in the version space so constructed, doing so has at least two drawbacks. First, the system is not able to handle plans for *arbitrary* domain goals. Second, complete planning knowledge is required in order to determine the consistency of an observed action with the hypothesized goals. That is, a perfect description of all the preconditions and effects of each action to be observed has to be provided.

Conclusion and Future Work

This paper presented an approach towards the acquisition of plan decompositions from logged action sequences. The

necessity of an action (or condition) that is manifested by its occurrence in *all* possible action sequences instantiating an abstract plan description is approximated by its presence in all sample data.

The mechanism presented can make use of various kinds of domain knowledge without being dependent on their availability. That is, minimum input data are sufficient to come up with approximated plan decompositions the quality of which can be improved by exploiting information about the elements occurring in a given domain. In case no domain knowledge is provided, the plan decompositions will exclusively contain basic actions. In this case the abstraction is limited to removing unnecessary temporal and structural relations among actions and replacement of domain constants by variables. Empirical tests demonstrated that the optimal choice of a most or least restrictive plan decomposition can be efficiently approximated within given bounds.

Future work will be devoted to handling a richer set of temporal relations and control structures like loops within plan decompositions. Additionally, the acquisition of abstraction relations among plans—an important component of plan *hierarchies*—will be a central topic.

Acknowledgments

I would like to thank Hiroshi Motoda and Neal Lesh for providing rich test data from the UNIX domain and Dietmar Dengler and the anonymous reviewers for their valuable comments.

References

- Albrecht, D.; Zukerman, I.; Nicholson, A.; and Bud, A. 1997. Towards a Bayesian Model for Keyhole Plan Recognition in Large Domains. *User Modeling* 97, 365–376.
- Bauer, M. 1996. Acquisition of User Preferences for Plan Recognition. *User Modeling* 96, 105–112.
- Kautz, H. 1991. A Formal Theory of Plan Recognition and its Implementation. In *Reasoning About Plans*. Morgan Kaufmann. chapter 2.
- Lesh, N., and Etzioni, O. 1996. Scaling up goal recognition. *KR96*, 178–189.
- Minton, S. 1985. Selectively Generalizing Plans for Problem-Solving. *IJCAI* 85, 596–599.
- Mooney, R. 1988. *A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding*. Ph.D. Diss., Univ. of Illinois, Urbana.
- Woods, W., and Schmolze, J. 1991. The KL-ONE family. *Computers and Mathematics with Applications* 23(2-5):133–177.
- Yoshida, K., and Motoda, H. 1995. CLIP: concept learning from inference patterns. *Artificial Intelligence* 75:63–92.