Artificial Intelligence 176 (2012) 2270–2290

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Plan recognition in exploratory domains

Ya'akov Gal[a,b,*], Swapna Reddy[b], Stuart M. Shieber[b], Andee Rubin[c], Barbara J. Grosz[b]

[a] *Department of Information Systems Engineering, Ben-Gurion University of the Negev, Israel*
[b] *School of Engineering and Applied Sciences, Harvard University, USA*
[c] *TERC, USA*

A R T I C L E   I N F O

Article history:
Received 1 March 2010
Received in revised form 8 August 2011
Accepted 11 September 2011
Available online 3 October 2011

Keywords:
Plan recognition
User modeling

A B S T R A C T

This paper describes a challenging plan recognition problem that arises in environments in which agents engage widely in exploratory behavior, and presents new algorithms for effective plan recognition in such settings. In exploratory domains, agents' actions map onto logs of behavior that include switching between activities, extraneous actions, and mistakes. Flexible pedagogical software, such as the application considered in this paper for statistics education, is a paradigmatic example of such domains, but many other settings exhibit similar characteristics. The paper establishes the task of plan recognition in exploratory domains to be NP-hard and compares several approaches for recognizing plans in these domains, including new heuristic methods that vary the extent to which they employ backtracking, as well as a reduction to constraint-satisfaction problems. The algorithms were empirically evaluated on people's interaction with flexible, open-ended statistics education software used in schools. Data was collected from adults using the software in a lab setting as well as middle school students using the software in the classroom. The constraint satisfaction approaches were complete, but were an order of magnitude slower than the heuristic approaches. In addition, the heuristic approaches were able to perform within 4% of the constraint satisfaction approaches on student data from the classroom, which reflects the intended user population of the software. These results demonstrate that the heuristic approaches offer a good balance between performance and computation time when recognizing people's activities in the pedagogical domain of interest.

1. Introduction

In this paper we report on the development and evaluation of algorithms for recognizing users' plans in domains in which users engage in exploratory and error-prone behaviors. The challenges presented by these domains were made evident by our work with students using open-ended computer software for learning statistics, but they arise in human–computer interaction more broadly.

Indeed, developing technology is changing rote and monolithic interaction styles between computers and their users to more flexible types of interactions that allow users to explore and interleave between different activities. Examples of these flexible systems include interactive drawing tools [44], Integrated Development Environments (IDEs), collaborative writing assistants [4], computer games, and educational software [51].

To be effective partners, these systems need to recognize the activities their users are carrying out and to use that information to provide support in a way that guides users' interactions effectively. For example, an intelligent drawing tool

* Corresponding author at: School of Engineering and Applied Sciences, Harvard University, USA.
E-mail address: gal@eecs.harvard.edu (Y. Gal).

0004-3702/$ – see front matter © 2011 Published by Elsevier B.V.
doi:10.1016/j.artint.2011.09.002

may infer that several objects on the canvas are all representatives of the same class. When the user modifies an attribute in one of the forms, the system will identify and duplicate this change in the other objects in the class. Another benefit of recognizing users' activities in software is to provide assessments of user performance. Such capabilities in educational and pedagogical systems could increase teachers' abilities to identify those students who are having difficulty.

Classical approaches to plan recognition have assumed a goal-oriented agent whose activities are consistent with the recognizers' knowledge base and who forms a single encompassing plan. In contrast, flexible systems allow users to follow multiple plans, interleave actions from different plans, and perform redundant actions; they also tolerate user mistakes. Thus, inferring users' plans in these systems gives rise to a more complex sort of plan recognition problem.

This paper presents several new algorithms for keyhole plan recognition in exploratory domains.[1] The algorithms are post-hoc, in that they infer plans from complete interaction sequences, rather than after each observed action, as in on-line recognition [13]. The algorithms we present vary in completeness (that is, whether plans are guaranteed to be found) and computational complexity. We investigate the trade-off between completeness and complexity empirically, by comparing the performance of different plan recognition algorithms on real-world data.

Our empirical analysis uses an educational software system for statistics education. Educational software is increasingly designed to be open-ended and flexible in order to support the types of exploratory activities that facilitate students' learning experience. This gives students the resources to explore concepts in new ways, but their interactions may be erratic or unfocused, making it challenging to recognize plans. During the chaos of a lab session, it is impossible for teachers to track each student's progress. As a result it is difficult to adapt their teaching to their students' work. Educational software thus provides an important domain for plan recognition. A well structured post-hoc representation of the plans behind students' activities would enable teachers to make better pedagogical decisions in the classroom.

The research we report used a commercial system called TinkerPlots, used world-wide to teach students in grades 4 through 8 about statistics and mathematics [34]. Using TinkerPlots, students build stochastic models and generate pseudo-random samples to analyze the underlying probability distributions. Our study used four different problems for which students interacted with TinkerPlots to model hypothetical situations and to determine the probability of events.

Students' interactions with TinkerPlots are complex. They may pursue multiple plans and interleave actions from different plans. They may be confused about the appropriate plan to take, and they may make mistakes. These behaviors create a challenging domain for plan recognition algorithms. Any number of extraneous actions may be interleaved among those that are a part of a successful plan. In addition, actions that are crucial to successful plans may occur in almost any order.

All of the algorithms presented in the paper compose (possibly non-contiguous) interaction sequences from users' interactions into a series of interdependent tasks and sub-tasks. They infer students' plans by comparing their interaction sequence to ideal solutions, or recipes, that were specified by domain experts. At the end of this process, the algorithms output a hierarchical plan that explains the student's strategy during the session. The algorithms separate those actions that contribute to solving the problem from extraneous actions and mistakes.

This paper integrates and extends initial reports of past studies [23,43] and makes several contributions. First, it formally defines the task of plan recognition in exploratory domains and provides a proof of its NP-completeness. Second, it presents new greedy and complete algorithms for solving the plan recognition problem in these domains, providing a formal complexity analysis of these algorithms and comparing them to existing methods. Third, it is the first work to evaluate plan recognition algorithms on real-world data in the domain of flexible pedagogical software.

We compared two algorithmic approaches for recognizing users' interactions. One of the approaches employed incomplete greedy algorithms to attempt to build plans from the bottom-up. The complexity of one of these algorithms is polynomial in the size of the interaction sequence, while the complexity of the other algorithm is exponential (in the worst case) in the size of this sequence. The second approach converts the recognition process to a Constraint Satisfaction Problem (CSP) using one of two methods. One of these methods builds a complete plan to recognize the entire interaction sequence. The other method works piecemeal in a way that uses subsets of the activity sequence to eliminate infeasible plans before attempting to recognize the entire sequence. This second method was suggested by Quilici et al. [42] but first tested empirically here. In contrast to the greedy approach, the constraint satisfaction approach is complete, in the sense that if all of the recipes for solving a given TinkerPlots problem exist, and the student solved the problem, the algorithm is guaranteed to find the plan that explains the student's interaction. The complexity of both of the complete methods is exponential in the size of both the interaction sequence and the data set containing ideal solutions.

We conducted a number of empirical studies to evaluate the ability of these algorithms to recognize the plans used to solve TinkerPlots problems. The studies involved two types of settings: adults using TinkerPlots in a lab setting, and middle school students using TinkerPlots in a classroom setting. The results confirmed that the complete algorithms were able to recognize all plans when the relevant recipes for the TinkerPlots problems existed, and students were able to solve the problems. However, there was a systematic difference between these two empirical settings and their effect on the plan recognition algorithms. For adult data, the complete methods outperformed the heuristic approaches by 25%. For student data, which reflects the intended user population of TinkerPlots, this difference was just 4%. In addition, the heuristic approaches were (on average) an order of magnitude faster than the complete approaches for both data sets. We show

[1] We use the term "keyhole plan recognition," coined by Cohen et al. [17], to refer to the fact that the acting agent is not signalling its plan to the observer.

that the determinant for run-time is the size of the interaction sequence for the heuristic approaches, and the size of the plan database for the complete approaches. Lastly, the interaction sequences obtained from middle school students were significantly longer than those of adults, and in general, students' interactions corresponded to complete solutions less often than adults.

These results show that the heuristic algorithms we devised provide a good balance between performance and time in the pedagogical software domain we considered. More generally, they demonstrate the feasibility of using Artificial Intelligence techniques to support the analysis of users' interaction with flexible, open-ended software. Although our study uses one type of software, the techniques presented here are general and can be used to support the analysis of users' interactions for different types of exploratory systems. Our techniques are of value to software designers and researchers who wish to understand the way people learn and use computer software, as well as to teachers.

After describing related work (Section 1.1), we introduce the TinkerPlots software (Section 2), highlighting the properties that characterize an exploratory domain. In Section 3, we describe the formal tools for representing plans in exploratory domains and the abstract problem of recognizing plans relative to idealized recipes for achieving domain goals. We then (Section 4) draw an analogy between this plan recognition problem and grammar recognition, showing that the problem is equivalent to context-free recognition under a variant interpretation of such grammars. The analogy allows a simple proof of the NP-completeness of plan recognition in exploratory domains. We present a variety of plan recognition algorithms for this problem in Section 5, and evaluate their performance empirically on data obtained from users' interactions with TinkerPlots in Section 6, demonstrating the practicality of the best of our algorithms in both coverage and speed.

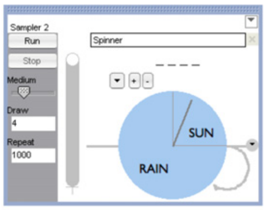
1.1. Related work

Plan recognition is a cornerstone problem of AI and a necessary component of many applications such as software help systems [7,37], story understanding [16,50], and natural language dialogue [14,29]. Early approaches have assumed a goal-oriented agent whose activities were consistent with its knowledge base, and which formed a single encompassing plan [33, 36]. A notable exception is Pollack [39] that allowed for agents to have ill-formed plans about achieving certain goals, and Brown and Burton [12] that allowed for agents' knowledge to be possibly incorrect. We refer the reader to Carberry [15] for a detailed account of these approaches and focus this section on more recent works which capture some of the endemic qualities of exploratory domains, namely extraneous actions or mistakes, interleaving of activities, and free order among plan constituents.

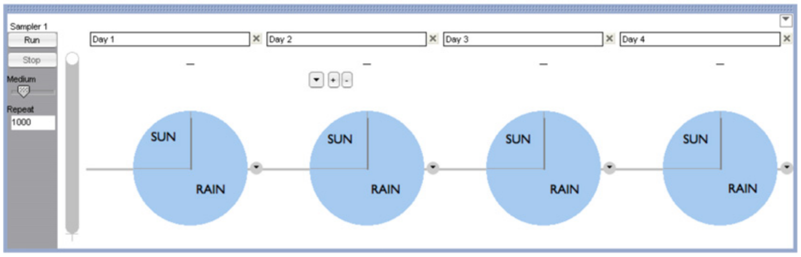
We first detail approaches that considered temporal relationships among actions that make up agents' plans. Weida and Litman [49] proposed a method for recognizing plans that explicitly included ordering constraints in the plan library and suggested various criteria for matching plans to action sequences, assuming that each action is directed at completing one of the plans in the library. Avrahami-Zilberbrand and Kaminka [3] encoded relationships between action parameters in plans using tree structures and provided methods for plan recognition that traverse the tree in a manner that is temporally consistent with the observations. Another approach to handling temporal relationships in plans derives from the analogy between plan recognition and grammar recognition [46,25]. Immediate-Dominance/Linear-Precedence (ID/LP) grammars [24] describe languages that allowed for linear precedence and free word ordering over rule constituents. Algorithms for parsing ID/LP grammars, which are analogous to recognizing plans, can at times provide exponential savings as compared to considering every possible order configuration of the rule constituents [45,5]. Pynadath and Wellman [41] developed a probabilistic grammar for modeling agents' plans that also included their beliefs about the environment. These techniques did not allow for interleaving plans. All reordering among plan constituents in these above works was restricted to local permutation among the constituent actions of sub-plans.

Goldman et al. [28] proposed a probabilistic model of plan recognition that recognized interleaving actions and output a disjunction of plans—rather than a single hierarchy—to explain an action sequence. It also accounted for missing observations (e.g., not seeing an expected action in a candidate plan makes another candidate plan more likely). The algorithm was generative, that is, with each observation, a pending set of possible hypotheses were generated that were subsequently matched against future observations. Geib and Goldman [26] have augmented this work to allow to recognize multiple instances of the same plan, in addition to interleaving actions. This work provides a bottom-up algorithm that maintains a distribution over the set of possible explanations matching users' observations, while not assuming that agent's top-level goal is known. Our work is distinct from this approach in several ways. First, the settings studied by Geib and Goldman do not account for agents' extraneous actions, an endemic property of the exploratory domain we consider in this paper. Second, the probabilistic approach used by Geib and Goldman is complete when considering full observation sequences, with a worst-case complexity that is exponential in the size of the grammar. We provide heuristic algorithms that may be exponentially more efficient than complete approaches. Our algorithms are parameter-free, designed for ecologically realistic settings (such as classrooms) in which tuning or learning parameters is difficult because of the effort involved in obtaining large amounts of training data. Third, we show the efficacy of our approach on real-world data obtained from students and adults using pedagogical software, whereas Geib and Goldman use synthetic data.

Several works have used probabilistic reasoning to recognize students' goals when interacting with pedagogical software. Conati et al. [18] used Bayesian networks to model students' interactions with an intelligent tutor, using probabilistic inference to recognize interleaving actions. Albrecht et al. [1] suggested a probabilistic approach to infer players' goals as well as their future actions from observation sequences. They used Dynamic Bayesian Networks to compute a posterior distribution



(a) Using one spinner



(b) Using four spinners

Fig. 1. Two sampler models for RAIN.

over possible goals given players' actions in the game. They are able to capture agents' mistakes, but infer the likelihood of a single goal or action, rather than recognizing a hierarchical plan representing the entire action sequence. Quilici et al. [42] proposed an algorithm for implementing plan recognition as a constraint satisfaction problem but do not evaluate it on real data. We augment this work in several ways. First, by implementing this algorithm on ecologically realistic data, that of adults and middle school students using pedagogical software. Second, by describing alternative, heuristic approaches to complete algorithms in exploratory domains that provide a balance between completeness and time, and comparing the efficacy of these algorithms to the complete, CSP approach.

Lastly, we will mention work in the intelligent tutoring systems community that has been applied to models of students' learning of mathematics and physics [19,8,2,47]. In these domains the tutor is an active participant in the student's learning process and ambiguities or uncertainties about the students' plan of action are resolved by querying the student. By contrast, the TinkerPlots style of educational software allows students to "learn by doing" in an exploratory open-ended manner without explicit guidance by a software tutor. Our approach addresses a different problem, that of non-intrusive recognition of students' activities given their complete interaction histories with the software. Past work on recognition of users' goals with computer systems has focused on fixed, strongly constrained settings such as UNIX command line syntax [9], or applications such as medical diagnosis and email notifications in which users tend to adopt the same goals many times [6, 31,35]. In educational domains, goals are constantly evolving to reflect new concepts, and it may be difficult to collect student-specific training data for each type of goal.

2. The TinkerPlots domain

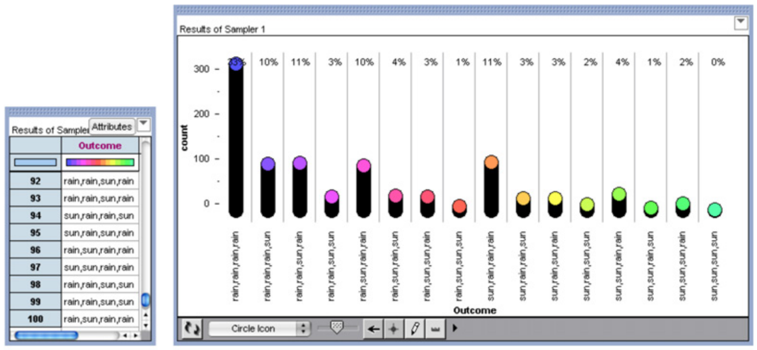
TinkerPlots is an educational software system used world-wide to teach students in grades 4 through 8 about statistics and mathematics [34]. It provides students with a toolkit to actively model stochastic events, and to create and investigate a large number of statistical models [30]. As such, it is an extremely flexible application, allowing for data to be modeled, generated, and analyzed in many ways using an open-ended interface.

To demonstrate our approach towards recognizing activities in TinkerPlots we will use the following running example, called RAIN.

RAIN: *The probability of rain on any given day is 75%. Use TinkerPlots to compute the probability that it will rain on each of the next four consecutive days.*

This problem is a simple example drawn from a set of problems posed to students using TinkerPlots in schools and to subjects during our data collection procedure.

Two of the several possible approaches towards modeling this problem in TinkerPlots are shown in Fig. 1. One uses the same stochastic device multiple times, while the other uses multiple stochastic devices. Fig. 1(a) shows a sampler object containing a single "spinner" device. Devices are added to sampler objects to model distributions. There are several types of devices; spinner devices recall the distribution formed by spinning a dial. The spinner device in the left-hand model contains two possible events, "rain" and "sun". The likelihood of "rain" is three times that of "sun", as determined by the surface area of these events within the spinner. Each draw of this sampler will sample the weather for a given day. The number of draws is set to four, making the sampler a stochastic model of the weather on four consecutive days.



Results of Sampler

	Outcome
92	rain,rain,sun,rain
93	rain,rain,sun,rain
94	sun,rain,rain,sun
95	sun,rain,sun,rain
96	rain,sun,rain,rain
97	sun,sun,rain,rain
98	rain,rain,sun,sun
99	rain,rain,sun,sun
100	rain,sun,rain,rain

(a) Table of Sample Results

(b) Plotted Results

Fig. 2. Generating and analyzing data for the RAIN problem.

Another possible approach to modeling the RAIN distribution is presented in Fig. 1(b), which shows a sampler with four spinner devices. Each of these devices is a stochastic model of the weather on a given day, and the sampler, set to a single draw, draws once from each device. In both of these approaches, the sampler, along with the contained devices, is a model of the joint probability distribution over the weather for four consecutive days.

When a sampler is run, it generates data that is sampled according to the distribution defined by the parameters of its model. Fig. 2(a) shows a table object holding a portion of the sample generated by either of the sampler models in Fig. 1. Each line in the table represents a single repetition of the sampler, consisting of a "sun" or "rain" value for each of four days. Fig. 2(b) shows the end-result of a process in which this data is plotted onto a histogram for the purpose of inferring the likelihood of four consecutive days of rain. There are many other approaches for modeling the RAIN distribution and organizing the resulting data, which we do not show here.

As students interact with TinkerPlots through its engaging direct manipulation interface, they create and modify devices, sample stochastic events, graph the results, modify and retry aspects, in a fluid manner in which all kinds of objects can be manipulated in different orders, and with false starts and retries adding complexity to the exhibited behaviors. The TinkerPlots system is metered to log all the primitive direct manipulation actions of the user.

These logs constitute the trace of the observable behavior of the user. Our goal is to explain the log in terms of the problem-solving goal that the user was engaged in, such as solving the RAIN problem.

3. Methods and representation

In this section we introduce representations and algorithms towards describing TinkerPlots activities in a formal way.

3.1. Actions, recipes and plans

The nomenclature in this paper follows the foundational planning terminology grounded in philosophy [11,10,20]. The most fundamental components we define are called *basic actions*, which are atomic, and cannot be decomposed. *Complex actions* describe higher-level, more abstract activities that can be decomposed into sub-actions, which can be basic actions or complex actions themselves. To emphasize the distinction between basic and complex actions, we notate complex actions using an underline notation.

A *recipe* [40] for a complex action characterizes the sequences of actions that result in successful completion of the action. The recipe for a complex action $\underline{C}$ is a set of sub-actions S and constraints R such that performing the sub-actions under the constraints constitutes completing the complex action. We do not allow recursively defined recipes; i.e., a recipe for the complex action $\underline{C}$ may not hereditarily include $\underline{C}$ in sub-action list S.

The set of *restrictions* R constrains how sub-actions may be completed by expressing relationships over the parameters of the sub-actions that must hold. Restrictions may take the form of any Boolean relation over sub-actions' parameters, which includes mathematical equations and inequalities. A common type of restriction uses inequalities and the pos (position) parameter of various actions to limit the order in which these actions must occur. In the absence of ordering restrictions, a recipe is completely free-ordered. Other restrictions may enforce a relationship between object identifiers in several TinkerPlots actions, requiring, for example, that two actions share the same i_s parameter to represent constraints that are imposed on the same sampler object s. To complete complex action $\underline{C}$ according to a recipe, all sub-actions in S must be completed without violating any restrictions in R.

We notate a recipe for complex action $\underline{C}$ with sub-actions $\{s_1, \ldots, s_n\}$ and restrictions R as

$$\underline{C} \rightarrow s'_1, \ldots, s'_n \quad \text{where } R$$

```
...
AS[i_s = 11, pos = 5]
ADS[i_s = 11, i_d = 2, t_d = spinner, pos = 6]
ALE[i_s = 11, i_d = 2, i_e = 1, l_e = a, pos = 7]
ALE[i_s = 11, i_d = 2, i_e = 2, l_e = b, pos = 8]
AS[i_s = 9, pos = 9]
CEL[i_s = 11, i_d = 2, i_e = 1, l_e = rain, pos = 10]
CEL[i_s = 11, i_d = 2, i_e = 2, l_e = sun, pos = 11]
CPD[i_s = 11, i_d = 2, ss = 1 : 3, pos = 12]
ADS[i_s = 11, i_d = 3, t_d = mixer, pos = 13]
...
```

Fig. 3. A snippet of an action sequence taken from a user's interaction with TinkerPlots.

Basic actions	**Parameters**
ADS = Add Device to Sampler	i_d = Device ID
ALE = Add Labelled Event	i_e = Event ID
AS = Add Sampler	i_s = Sampler ID
CEL = Change Event Label	l_e = Event Label
CPD = Change Probability in Device	t_d = Device Type
CSA = Create Sampler with Event A	ss = Subsection Size
	pos = Temporal Position
Complex actions	
$\underline{\text{AED}}$ = Add Event to Device	
$\underline{\text{CCD}}$ = Create Correct Device	

Fig. 4. Action and parameter abbreviation key.

where s_i' is the name of the sub-action s_i (with optional subscripts to uniquely identify sub-actions with the same name). The restrictions R use the notation $A[p]$ to refer to the value of a parameter p of some sub-action with name A. (In case multiple sub-actions have the same name, the subscripts are used to disambiguate.) Standard conventions are used to notate multiple inequalities (for example, $a \prec b \prec c$).[2]

A *recipe library* [11] contains the complete set of recipes for all of the complex actions of the domain. Each complex action type may have multiple recipes in the library providing alternatives for its completion.[3]

A *plan* is a hierarchical construction of basic and complex actions used to complete a complex action called the *root action*. The plan for completing a root action $\underline{C}$ is a tree of parametrized actions rooted at $\underline{C}$ such that each complex action is decomposed into sub-actions according to some recipe in the database.

3.2. Representation of TinkerPlots activities

The nature of the questions (such as RAIN) for teaching statistical skills in TinkerPlots typically will require students to plan a series of activities to derive answers. Students interact with TinkerPlots through a series of operations that create, modify, or delete objects such as samplers, plots, and tables. Basic actions in TinkerPlots refer to rudimentary operations that can be carried out by a single keystroke or mouse action. It is these instances of basic actions that are logged as the system is used. Examples of basic actions in TinkerPlots include creating a new sampler, generating a random sample, or deleting a plot. Complex actions in TinkerPlots are activities such as adding a spinner with six equally weighted events to a sampler, fitting sampler data to a plot, or solving the RAIN problem. We impute complex actions to users of the software in our analysis of users' actions as pursuing plans.

Users' interactions with TinkerPlots are recorded as finite, chronological sequences of basic actions that are performed by the users. It is these action sequences that constitute the input to the plan recognition algorithm. Fig. 3 shows a portion of an action sequence for creating the stochastic component (called a "device") in the sampler of Fig. 1(a).[4] For example, the action $\text{ADS}[i_s = 11, i_d = 2, t_d = \text{spinner}, \text{pos} = 6]$ (Add Device to Sampler) refers to the action of adding a device with an identifier $i_d = 2$ and type $t_d = \text{spinner}$ to a sampler with an identifier $i_s = 11$. The pos parameter specifies the temporal position of the action within an action sequence; in this case ADS is the sixth action performed by the user. Fig. 4 provides a key to abbreviations for all actions (in upper-case script) and parameters (in lower-case script) used throughout the paper.

In the TinkerPlots domain, a recipe captures an ideal sequence of actions for performing a particular activity. We represent each basic or complex action type in TinkerPlots as a unique name (such as ADS for the basic action Add Device to Sampler, or $\underline{\text{CCD}}$ for the complex action Create Correct Device); they are parametrized to describe features of the objects to which an instance of the action refers. We notate an action with its parameters by placing the parameter values, keyed

[2] Our representations of recipes and restrictions are similar to classical planning formalisms such as Hierarchical Task Networks [27], but do not allow for recursion.

[3] Other works have used the term "plan library" to refer to complete plan hierarchies from an agent's root goal down to the basic level actions; see for example Nau et al. [38]. We use the term "recipe library" to refer to a set of recipes.

[4] This sampler was used to generate the data in Fig. 2.

$$\begin{gathered}\underline{\mathsf{CCD}} \to \mathsf{ADS}, \underline{\mathsf{AED}_1}, \underline{\mathsf{AED}_2}, \mathsf{CPD}\\ \text{where}\end{gathered}$$
$$\begin{aligned}&\mathsf{ADS[pos]} \prec \underline{\mathsf{AED_1[pos]}} \prec \underline{\mathsf{AED_2[pos]}} \prec \mathsf{CPD[pos]}\\ &\mathsf{ADS[i_s]} = \underline{\mathsf{AED_1[i_s]}} = \underline{\mathsf{AED_2[i_s]}} = \mathsf{CPD[i_s]}\\ &\mathsf{ADS[i_d]} = \underline{\mathsf{AED_1[i_d]}} = \underline{\mathsf{AED_2[i_d]}} = \mathsf{CPD[i_d]}\\ &\mathsf{CPD[ss]} = (3:1)\end{aligned}$$

Fig. 5. A recipe for the $\underline{\mathsf{CCD}}$ (Create Correct Device) complex action.

$$\begin{gathered}\underline{\mathsf{AED}} \to \mathsf{ALE}, \mathsf{CEL}\\ \text{where}\end{gathered}$$
$$\begin{aligned}&\mathsf{ALE[pos]} \prec \mathsf{CEL[pos]}\\ &\mathsf{ALE[i_s]} = \mathsf{CEL[i_s]}\\ &\mathsf{ALE[i_d]} = \mathsf{CEL[i_d]}\\ &\mathsf{ALE[i_e]} = \mathsf{CEL[i_e]}\end{aligned}$$

Fig. 6. A recipe for the $\underline{\mathsf{AED}}$ (Add Element to Device) complex action.

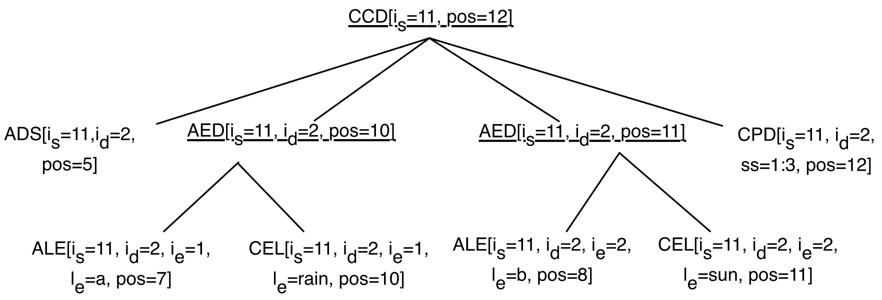

CCD[is=11, pos=12]
ADS[is=11,id=2, pos=5]
AED[is=11, id=2, pos=10]
AED[is=11, id=2, pos=11]
CPD[is=11, id=2, ss=1:3, pos=12]
ALE[is=11, id=2, ie=1, le=a, pos=7]
CEL[is=11, id=2, ie=1, le=rain, pos=10]
ALE[is=11, id=2, ie=2, le=b, pos=8]
CEL[is=11, id=2, ie=2, le=sun, pos=11]

Fig. 7. A possible plan for the $\underline{\mathsf{CCD}}$ complex action.

to the parameter names, in brackets after the name. For example, the set of sub-actions for one of the possible recipes for solving RAIN includes creating a sampler that models the weather on four consecutive days, running the sampler, and plotting the results on a graph. Examples of such samplers are shown in Figs. 1(a) and 1(b). An example of a graph is shown in Fig. 2(b).

Fig. 5 shows a recipe for the complex action $\underline{\mathsf{CCD}}$, which creates the sampler shown in Fig. 1(a). It includes two basic sub-actions, ADS and CPD, as well as two complex $\underline{\mathsf{AED}}$ (Add Event to Device) actions. This recipe also contains several restrictions. The first, an ordering restriction, mandates that a device must be added to a sampler (action ADS) before any events are added to that device (action $\underline{\mathsf{AED}}$), which in turn must occur before the probability of those events is changed (action CPD). The second and third restrictions require that sampler and device identifiers are consistent across these actions, and the fourth restriction requires that the surface area of events be resized to a 3 : 1 ratio.

The purpose of giving TinkerPlots problems to students is to test their ability to construct appropriate, applicable models for solving the problem. We treat recipes as idealized descriptions of the use of TinkerPlots to solve problems by constructing a plan for achieving the complex root action for solving the problem. For example, a plan for the complex action $\underline{\mathsf{CCD}}$ is shown in Fig. 7. Here, each complex $\underline{\mathsf{AED}}$ action is decomposed into sub-actions ALE and CEL using the recipe shown in Fig. 6. The plan was inferred from the student's actions in the sequence shown in Fig. 3, and results in the creation of a spinner device, such as the one shown in Fig. 1(a) for solving the RAIN problem.

The flexible nature of TinkerPlots supports exploratory and open-ended use of the software in several ways. First, students may perform extraneous activities that do not play a salient part in the solution to the problem. For example, the action $\mathsf{AS[i_s = 9, pos = 9]}$ in the student's action sequence of Fig. 3 plays no role in the student's plan in Fig. 7. Second, students may interleave the sub-actions of different complex actions. For example, in the plan of Fig. 7, the $\mathsf{ALE[i_s = 11, i_d = 2, i_e = 2, l_e = b, pos = 8]}$ action in position 8 (a sub-action of the complex action $\underline{\mathsf{AED}[i_s = 11, i_d = 2, i_e = 2, \mathsf{l_e = sun, pos = 11}]}$, temporally occurs among the actions $\mathsf{ALE[i_s = 11, i_d = 2, i_e = 1, l_e = a, pos = 7]}$ and $\mathsf{CEL[i_s = 11, i_d = 2], i_e = 1, l_e = rain, pos = 10}$ in positions 7 and 10 (sub-actions of the complex action $\underline{\mathsf{AED[i_s = 11, i_d = 2, pos = 10]}}$). Lastly, students may make mistakes when solving problems or only succeed in solving part of the problem. The combination of these different properties make it challenging to recognize the plans underlying students' interactions with TinkerPlots, as we argue formally in the next section.

4. Grammars and complexity

The general problem of recognizing whether a sequence of basic actions embeds a plan that accords with the recipes in the database includes satisfying an arbitrary set of constraints. This problem is at least as complex as constraint satisfaction for the constraint language, which can be NP-hard or worse depending on the particular of the constraint language. But even without the solving of restrictions, the plan recognition problem is NP-complete. Geib and Goldman [26] show that a related plan recognition problem—involving interleaving and ordering restrictions but not extraneous actions—is NP-hard via a simple reduction from three-dimensional matching to a grammar formalism they call plan tree grammars. The extension to our context, in which extraneous actions in the log are allowed, is straightforward.

In this section, we review and extend the Geib and Goldman proof to derive a complexity result for plan recognition in exploratory domains. For simplicity, we use a simpler formal characterization than the plan tree grammars used by Geib and Goldman. We define a grammar formalism that, like plan tree grammars, allows interleaving, but unlike plan tree grammars, has no temporal ordering restrictions. This allows us to greatly simplify the description of the formalism. For concreteness, we call the formalism *simple plan grammars*.

A simple plan grammar is structured exactly like a context-free grammar, with a set of terminal and nonterminal symbols including a specified start symbol, plus a set of productions rewriting a nonterminal symbol to a sequence of terminals and nonterminals. As with other grammatical characterizations of planning, the nonterminal symbols correspond to complex actions and the terminal symbols to basic actions. Under this analogy, a recipe corresponds to a grammatical production, a plan to a parse tree, and an action sequence to a string to be parsed. Reconstructing a plan from an action sequence relative to a recipe library would then correspond to parsing a string relative to a grammar [48].

Although simple plan grammars are structured identically to context-free grammars, the language of a simple plan grammar is defined differently from the corresponding context-free grammar, so as to manifest interleaving and extraneous actions. (Indeed, one can think of a simple plan grammar as an alternate interpretation of the context-free grammar notation.)

The language of a simple plan grammar is defined in two steps. First, we define the *base language* of a simple plan grammar to be the language of the corresponding context-free grammar. The *language* of a simple plan grammar is then the set of all strings containing a subsequence that is a permutation of a string in the base language.

This simple definition captures exactly the reordering, interleaving, and extraneous action aspects of the plan recognition problem, while abstracting away from temporal ordering and other constraints. The reordering and interleaving is captured by the fact that all permutations of the base language strings are in the language of the grammar. The extraneous actions are captured by including supersequences in the language as well, the extra symbols constituting the extraneous actions.

What is not captured by simple plan grammars is the ordering restrictions. It is the ordering restrictions that greatly complicates the definition of plan tree grammars. As we will show, the ordering restrictions are not needed to carry through the NP-hardness proof, and therefore the simpler formalism is sufficient for showing that the plan recognition setting we are considering is NP-hard.

By way of example, consider the following productions:

$$\begin{aligned} S &\rightarrow M\ M\ M \\ M &\rightarrow a\ b\ c \\ M &\rightarrow d\ e\ f \\ M &\rightarrow g\ h\ i \end{aligned} \tag{1}$$

If we take these to be the productions of a context-free grammar, the grammar recognizes several strings, including the strings *abcdefghi* and *abcabcabc*. However, when viewed as a simple plan grammar, it recognizes all supersequences of permutations of these strings, including the strings themselves, but also strings like *adgbehcfi* or *ihgfedcba* or *aaaabcdefghiaa*.

It is easy to show that the problem of string recognition for simple plan grammars is NP-complete. We extend the proof of Geib and Goldman [26], which uses a reduction from the NP-complete problem 3-DIMENSIONAL MATCHING:

3-DIMENSIONAL MATCHING (3DM): Given three identically sized disjoint sets $W = \{w_1, \ldots, w_q\}$, $X = \{x_1, \ldots, x_q\}$ and $Y = \{y_1, \ldots, y_q\}$, and a set $M \subseteq W \times X \times Y$, does there exist a matching consisting of a subset $M' \subseteq M$ of size q such that no two elements of M' agree on any coordinate (that is, all elements of W, X, and Y appear exactly once in M').

This problem was shown to be NP-complete by Karp [32].

We reduce an instance of 3DM to a simple plan grammar as follows. Given an instance of 3DM, we construct a simple plan grammar with two nonterminals S (the start symbol) and M, and terminal symbols $W \cup X \cup Y$. The productions of the grammar include one for each element $\langle w, x, y\rangle$ of M:

$$M \rightarrow w\ x\ y \tag{2}$$

and a production generating q instances of nonterminal M:

$$S \rightarrow \overbrace{M \cdots M}^{q \text{ times}} \tag{3}$$

Note that the base language of the grammar, that is, the language of the productions when viewed as a context-free grammar, comprises strings all of length $3q$. Thus, if a string in the language includes all of the elements of W, X, and Y, each must occur exactly once.

In addition, we construct the string $s = w_1 \cdots w_q x_1 \cdots x_q y_1 \cdots y_q$ containing each of the elements of W, X, and Y exactly once. We ask whether the string s is admitted by the constructed grammar.

By way of example, the simple plan grammar (1) is exactly the one generated by this construction for the 3DM problem in which $W = \{a, d, g\}$, $X = \{b, e, h\}$, and $Y = \{c, f, i\}$ and where $M = \{\langle a, b, c\rangle, \langle d, e, f\rangle, \langle g, h, i\rangle\}$. The constructed string to recognize would be *adgbehcfi*.

The construction has the property that s is admitted by the constructed simple plan grammar if and only if the corresponding 3DM instance has a solution. (In the example, the constructed string *adgbehcfi* is in the language of the simple plan grammar because its permutation *abcdefghi* is in the base language.) The argument is straightforward, and essentially that of Geib and Goldman [26], with variation only for the lack of ordering restrictions and the issue of extraneous items.

If there is a solution to the 3DM problem, then there is a subset M' of M that covers all and only the $3q$ elements of $W \cup X \cup Y$. By construction, then, there is a string in the base language that includes all of these elements as well. The constructed string s is a permutation of that string, hence is in the language of the simple plan grammar.

If the string s is in the language of the simple plan grammar, then there is a context-free derivation for some permutation of some subset of the elements of s. Because all strings in the base language are of length $3q$, which is the length of s itself, the base language string must be an improper subset, that is, have exactly the elements of s. But in that case, a solution of the 3DM problem can be read off of the context-free derivation of the base language string. The particular M-productions used in that derivation correspond to the M' subset.

This proof differs from that of Geib and Goldman [26] in a few ways. First, we do not incorporate ordering constraints in the rule $M \to w\,x\,y$ to require $w \prec x$ and $x \prec y$ as they do. These constraints are not necessary, because by construction the string to be recognized obeys such constraints directly. The same is true of the proof by Geib and Goldman [26]; the ordering constraints are superfluous there too. By observing the superfluity of ordering restrictions for the proof, we allow a simpler grammar setup.

Second, our definition of the language of a simple plan grammar incorporates all supersequences of base language strings, corresponding to allowing extraneous actions in logs in the plan recognition problem. The original proof was modified to hold even in this context by forcing all base language strings to include exactly $3q$ elements, the same as the constructed string to be recognized, so the issue of supersequences becomes irrelevant. Although the simple plan grammar constructed does admit strings longer than $3q$, they are irrelevant to the argument, as the string to be parsed is of length $3q$. Forcing the string to be of length at least $3q$ is the role of the S production, which has no analog in the Geib and Goldman proof.

Along similar lines, nothing in the constructed grammar enforces the condition that the elements of M chosen are distinct (that is, that no M-production is reused), and no such constraint on the grammar is necessary. If such duplication were to occur, the string generated would have repeated elements as well, but in that case, the derivation will never admit the string to be recognized, which by construction has no repeated elements.

We can conclude, then, that simple plan grammar recognition is NP-hard. The problem is also clearly in NP, as the constructed grammar is polynomial in the size of the 3DM instance, and the context-free derivation for the base language permutation of s serves as a polynomially-sized witness for the recognition problem. Checking that the witness is for a permutation of s is trivially done in polynomial time.

Thus, plan recognition in our model, in which recipes can be interleaved and extraneous actions can be observed, is NP-hard as well. Indeed, this holds whether or not extraneous actions can be observed; they were not made use of in the proof. Similarly, no use was made of recursion in the grammar, so a restriction to non-recursive recipes does not reduce the complexity. Finally, no use of ordering restrictions was made in the proof; satisfying such restrictions makes the recognition problem only more complex.

Given the computational complexity of this plan recognition problem, the question arises as to whether it can be solved in practice for problems of the scope that confront us in real-world cases. We turn to heuristic plan recognition algorithms and their performance in the next section.

5. Plan recognition algorithms

In this section we present several plan recognition algorithms that are able to handle the interleaving and extraneous actions that are endemic to exploratory domains such as TinkerPlots. All of these approaches make use of a structure called a *plan tree* for representing and reasoning about recipes in the database, essentially a search tree for capturing the set of possible plans consistent with the recipe database. A plan tree has two types of nodes: AND nodes, whose children represent actions that must be carried out to complete a recipe, and OR nodes, whose children represent a choice of recipes for completing an action. The root, action C, is an OR node. For each recipe for C, a child AND node is added to the root and labeled with the sub-actions of that recipe. The children of this AND node are the plan trees of each sub-action. A branch terminates when a basic action is reached, as a basic action has no recipe by definition.

A partial plan tree for the $\underline{\text{CCD}}$ action is shown in Fig. 8. The AND nodes contain set brackets, while OR nodes do not. Triangles denote unfinished subtrees which were omitted for expository convenience. The plan for creating the spinner

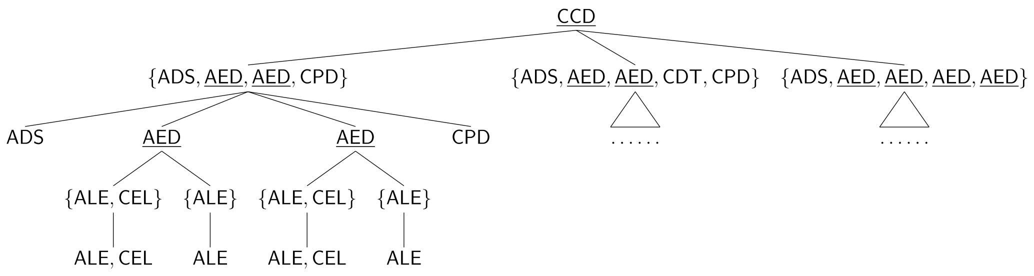


Fig. 8. A partial plan tree for the CCD complex action.

```
1: procedure BuildPlan(R, X)                                ▷ R: a recipe list, X: an action sequence
2:     $t \leftarrow 0$
3:     $P_0 \leftarrow X$                                     ▷ $P_t$: list of actions at stage $t$
4:     for $R_C \in$ SortRecipes(R) do                        ▷ $R_C$: a recipe for action $C$
5:         $P_{t+1}, OL \leftarrow P_t$                       ▷ $OL$: an open list
6:         $(M_C, OL) \leftarrow$ FindMatch($R_C, OL$)         ▷ $M_C$: a match
7:         while $M_C$ is not null do
8:             Add $C$ to $P_{t+1}$ positioned after last $a \in M_C$     ▷ $a$: an action
9:             for all $a \in M_C$ do
10:                Create a branch from $C$ in $P_{t+1}$ to $a$ in $P_t$
11:                Remove $a$ from $P_{t+1}$
12:            $(M_C, OL) =$ FindMatch($R_C, OL$, null)
13:        $t \leftarrow t + 1$
```

Fig. 9. Bottom-up plan recognition method.

object shown in Fig. 1(a) can be found by selecting the leftmost child at each OR node. This resulting plan mirrors the plan shown in Fig. 7.

5.1. Greedy algorithms

We present two greedy algorithms for inferring users' plans. Informally speaking, the algorithms work bottom-up, starting with the user log, and iteratively replacing a set of actions that match the sub-actions of a given recipe by the complex action the recipe implements so as to form a new action list.

A brute-force approach would involve non-deterministically finding all ways in which a complex action might be implemented in the action list. For example, the recipe library for the RAIN problem includes ten recipes and six complex actions. The different recipes for the RAIN problem can form 167,076 possible plans without considering different orderings between actions.[5] If we consider all possible orderings within recipes, we get that there are 2,109,182,681,760 possible plans for the RAIN problem. Naively considering each of these possibilities is infeasible.[6] The heuristic approaches presented in this section make various assumptions about exploratory domains that serve to significantly reduce their complexity as compared to the brute-force method. However, they are incomplete, in the sense that users may construct valid plans that the algorithms fail to infer.

At each step t, the algorithms incrementally build a plan by maintaining an ordered sequence of actions, denoted P_t. The action sequence P_0, representing the "ground level" of the user's plan, is denoted as **X**. During each step, the algorithms attempt to replace subsets of actions from P_t with the complex actions they represent. Each of the complex actions in P_t is a partial plan that explains some activity in the user's interaction.

Because our recipe formalization does not allow for recursion, we can define an ordering over all complex actions in the plan library that reflects their depth. Specifically, if B is a constituent sub-action for complex action A, then all recipes for B must appear before the recipes for A in the ordering. The heuristic algorithms consider recipes according to this order.

Both greedy algorithms are based on a function BuildPlan shown in Fig. 9 for constructing users' plans bottom-up. BuildPlan takes two inputs: An action sequence **X**, and a recipe library, **R**. The method calls SortRecipes to topologically sort **R** by depth from lowest to highest.

For each recipe R_C for complex action C, the action list P_{t+1} and an open list OL are initialized with the actions in P_t. The algorithm repeatedly tries to find a match for R_C in the open list by calling the function FindMatch(R_C, OL), which returns a tuple (M_C, OL), representing the actions M_C in the match and a modified open list. (The two methods we shall

[5] For example, there are 16 possible ways to complete the CCD action, because there are two possible recipes in our recipe library for each of the four MS actions.

[6] While it is theoretically possible to use string matching to align recipes to the action sequences, a naive approach would need to consider a prohibitive number of possible orderings. The complete approaches (the CSP algorithms) that we describe in the next session essentially perform this matching more efficiently.

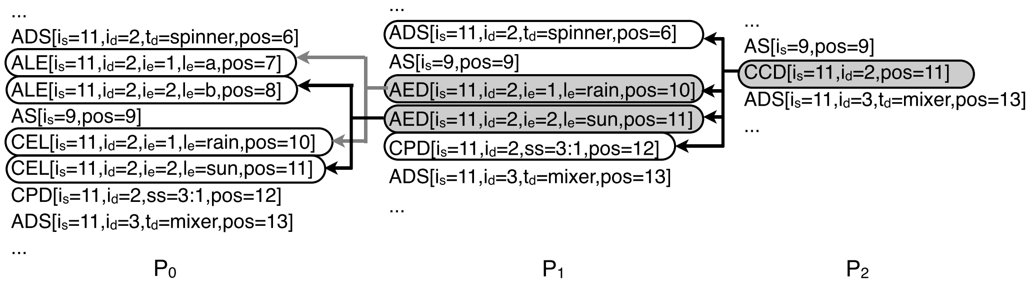

...
ADS[is=11,id=2,td=spinner,pos=6]
ALE[is=11,id=2,ie=1,le=a,pos=7]
ALE[is=11,id=2,ie=2,le=b,pos=8]
AS[is=9,pos=9]
CEL[is=11,id=2,ie=1,le=rain,pos=10]
CEL[is=11,id=2,ie=2,le=sun,pos=11]
CPD[is=11,id=2,ss=3:1,pos=12]
ADS[is=11,id=3,td=mixer,pos=13]
...
P0
...
ADS[is=11,id=2,td=spinner,pos=6]
AS[is=9,pos=9]
AED[is=11,id=2,ie=1,le=rain,pos=10]
AED[is=11,id=2,ie=2,le=sun,pos=11]
CPD[is=11,id=2,ss=3:1,pos=12]
ADS[is=11,id=3,td=mixer,pos=13]
...
P1
...
AS[is=9,pos=9]
CCD[is=11,id=2,pos=11]
ADS[is=11,id=3,td=mixer,pos=13]
...
P2

Fig. 10. Progression of BuildPlan over three steps.

soon present for FindMatch modify the open list in different ways.) If a match exists, BuildPlan removes the actions in M_C from P_{t+1} and replaces them with the complex action C, placed where the latest action in the match occurred. (In addition, it marks the actions in M_C in P_t to be the children of action C in P_{t+1}.) Actions are removed from P_t only when a match is found, according to the criteria defined in the method FindMatch. (We will provide two possible types of criteria in the algorithms below.) Once no more matches for R_C can be found, the algorithm moves on to the next recipe, until all the recipes have been considered.

Fig. 10 shows several stages of the BuildPlan procedure. Edges are shown between complex actions in each step and their constituent actions in the previous step. The first stage, titled P_0, contains only basic user actions. During the second stage, titled P_1, two pairs of non-contiguous basic ALE and CEL actions are found to be matches for two complex $\underline{\text{AED}}$ actions. This is an example of interleaving actions, because in P_0, the ALE constituent of the second $\underline{\text{AED}}$ action is positioned between the ALE and CEL constituents of the first $\underline{\text{AED}}$ action. In the third stage, a match for the $\underline{\text{CCD}}$ action is found, whose sub-actions consist of the first ADS action, both complex $\underline{\text{AED}}$ actions, and the CPD action. Fig. 10 defines a structure that is similar to the ideal plan for the $\underline{\text{CCD}}$ action shown in Fig. 7, except that it explicitly indicates the interleaving action sequences for the two $\underline{\text{AED}}$ actions, as well as includes redundant actions that were not part of the plan (e.g., $\text{AS}[\text{i}_\text{s} = 9, \text{pos} = 8]$).

BuildPlan is a greedy approach because it does not backtrack. After it chooses a match C for a given recipe R_C, it replaces the actions in P_{t+1} with complex action C without looking ahead to future stages. As a result, the algorithm may fail to find a match for a recipe because a necessary sub-action was committed to another match at an earlier stage. The complexity of the BuildPlan algorithm is dominated by the complexity of the FindMatch algorithm, call it C_{FM}, discussed in the next section. Let $|\mathbf{R}|$ and $|\mathbf{X}|$ be the number of recipes in $\mathbf{R}$ and the number of actions in the action sequence $\mathbf{X}$, respectively. Then, BuildPlan calls FindMatch at most $|\mathbf{X}|$ times per recipe, yielding an overall complexity of $O(|\mathbf{R}| \cdot |\mathbf{X}| \cdot C_{FM})$ for BuildPlan.

5.1.1. Matching algorithms

We present two possible matching algorithms for implementing the FindMatch(R_C, OL) process. Both of these make use of the Extends function, a Boolean function that takes as input an action a_P, a partial match M_C, and recipe R_C. It returns true if a_P can be added to M_C, such that (1) a_P corresponds to one of the constituent sub-actions of R_C and is not already in M_C and (2) the addition of a_P to M_C will not violate any of the recipe constraints in R_C. For example, the basic action $\text{ADS}[\text{i}_\text{s} = 11, \text{i}_\text{d} = 2, \text{t}_\text{d} = \text{spinner}, \text{pos} = 6]$ in the action sequence of Fig. 3 extends the recipe for $\underline{\text{CCD}}$ shown in Fig. 5 given that $M_C = \emptyset$.

The Boolean function Fulfills(M_C, R_C) returns true if M_C is a complete match for the recipe R_C. We then say that M_C *fulfills* R_C. Note that M_C can include both basic and complex actions. For example, the actions

$$\text{ADS}[\text{i}_\text{s} = 11, \text{i}_\text{d} = 2, \text{t}_\text{d} = \text{spinner}, \text{pos} = 6]$$

$$\underline{\text{AED}[\text{i}_\text{s} = 11, \text{i}_\text{d} = 2, \text{t}_\text{d} = \text{spinner}, \text{pos} = 10]}$$

$$\underline{\text{AED}[\text{i}_\text{s} = 11, \text{i}_\text{d} = 2, \text{t}_\text{d} = \text{spinner}, \text{pos} = 11]}$$

$$\text{CPD}[\text{i}_\text{s} = 11, \text{i}_\text{d} = 2, \text{pos} = 12]$$

fulfill the recipe for $\underline{\text{CCD}}$ shown in Fig. 5.

Both of the matching algorithms choose actions that extend M_C in any order that is allowed by the restrictions of recipe R_C. In particular, the actions in OL may be non-contiguous; this allows the algorithms to capture interleaving plans. However, the two methods differ in the way they update the action list OL as they build a match.

The first algorithm, NoBktrk(R_C, OL), shown in Fig. 11, is an extension of an earlier algorithm proposed by Gal et al. [23]. It receives as input R_C, a recipe for some complex action C and an open list OL, which is initially equivalent to the action

```
 1: procedure NoBktrk($R_C$, $OL$)                ▷ $R_C$: a recipe, $OL$: an open list
 2:   $M_C \leftarrow$ null
 3:   for $a_P \in OL$ do                                   ▷ $a_P$: an action
 4:     if Extends($a_P$, $M_C$, $R_C$) then                ▷ $M_C$: a partial match
 5:       Add $a_P$ to $M_C$
 6:       Remove $a_P$ from $OL$
 7:   if Fulfills($M_C$, $R_C$) then
 8:     return ($M_C$, $OL$)
 9:   else if $M_C$ is null then
10:     return (null, $OL$)
11:   else
12:     clear $M_C$ and goto line 2
```

Fig. 11. Algorithm for finding a match without backtracking.

```
 1: procedure SomeBktrk($R_C$, $OL$)              ▷ $R_C$: a recipe, $OL$: open list
 2:   return SomeBktrkRec($R_C$, $OL$, null)
 3: procedure SomeBktrkRec($R_C$, $OL$, $M_C$)               ▷ $M_C$: a partial match
 4:   if Fulfills($M_C$, $R_C$) then
 5:     return ($M_C$, $OL$)
 6:   $OL' \leftarrow OL$
 7:   for $a_P \in OL$ do                                   ▷ $a_P$: an action
 8:     remove $a_P$ from $OL'$
 9:     if Extends($a_P$, $M_C$, $R_C$) then
10:       Add $a_P$ to $M_C$
11:       ($M_C$, $OL$) = FindMatch($R_C$, $OL'$, $M_C$)
12:       if Fulfills($M_C$, $R_C$) then
13:         return ($M_C$, $OL$)
14:       remove $a_P$ from $M_C$
15:   return (null, $OL$)
```

Fig. 12. Algorithm for finding a match with depth-first search.

$$\underline{\text{CSA}} \rightarrow \text{AS, ADS, ALE}$$

where

$$\text{AS[pos]} \prec \text{ADS[pos]} \prec \text{ALE[pos]}$$
$$\text{AS}[\text{i}_\text{s}] = \text{ADS}[\text{i}_\text{s}] = \text{ALE}[\text{i}_\text{s}]$$
$$\text{ADS}[\text{i}_\text{d}] = \text{ALE}[\text{i}_\text{d}]$$
$$\text{ALE}[\text{I}_\text{e}] = \text{'A'}$$

Fig. 13. A recipe for the CSA (Create Sampler with event "A") complex action.

set in P_{t+1}. NoBktrk removes actions, one by one, from the open list and places them into a partial match, M_C. Once removed from the open list, these actions will not be reconsidered until a new recipe is provided at step $t+2$.[7]

The algorithm is quadratic in the size of the action sequence $|\mathbf{X}|$. To see this, consider that in the worst case, it takes a complete pass over the action list, which is bounded by the size of the action sequence, to fulfill a recipe. Because recipes in TinkerPlots are non-recursive, the number of times a recipe can be fulfilled is also bounded by the size of the action sequence. Therefore, the complexity of NoBktrk is $O(|\mathbf{X}|^2)$.

The second algorithm for finding a match, called SomeBktrk (Fig. 12), performs a complete depth-first search given a recipe R_C and an open list OL. It defines a sub-function that extends a partial match M_C with a single action, and makes a recursive call to the sub-function. In contrast to the NoBktrk algorithm, it is complete given a recipe R_C and an open list OL; that is, it is guaranteed to find a match for R_C if one exists in OL. However, SomeBktrk cannot guarantee that a plan is found, because BuildPlan itself is greedy. Due to BuildPlan's lack of forward-checking or backtracking across time steps, SomeBktrk may assign an action to a match during an early step and permanently remove that sub-action from the open list. SomeBktrk may later be unable to fulfill a crucial recipe requiring the same sub-action because a match no longer exists in the open list.

As an example of the way these two algorithms differ, consider a recipe for a complex action CSA (Create Sampler with Event A) for creating a sampler with one device and a single event labeled "A", shown in Fig. 13. Recall that both NoBktrk and SomeBktrk algorithms extend the current partial match by choosing actions in any order from the interaction sequence that meets the recipe constraints. Given the action sequence shown in Fig. 3, the SomeBktrk algorithm, which is complete given the recipe for CSA and the action sequence, will find the following match, which includes a sampler with identifier $\text{i}_\text{s} = 11$ and device with identifier $\text{i}_\text{d} = 2$.

[7] We hypothesized that actions that occur late in the interaction process are more salient than actions that occur earlier. However, in practice, traversing the open list in reverse order and increasing order of the pos parameter yielded the same results.

$$\mathrm{AS}[\mathrm{i_s} = 11, \mathrm{pos} = 5]$$

$$\mathrm{ADS}[\mathrm{i_s} = 11, \mathrm{i_d} = 2, \mathrm{t_d} = \mathrm{spinner}, \mathrm{pos} = 6]$$

$$\mathrm{ALE}[\mathrm{i_s} = 11, \mathrm{i_d} = 2, \mathrm{i_e} = 1, \mathrm{l_e} = \text{‘A’}, \mathrm{pos} = 7]$$

However, the NoBktrk may decide to add the following actions to the partial match M_C:

$$\mathrm{ADS}[\mathrm{i_s} = 11, \mathrm{i_d} = 3, \mathrm{t_d} = \mathrm{mixer}, \mathrm{pos} = 13]$$

$$\mathrm{AS}[\mathrm{i_s} = 11, \mathrm{pos} = 5]$$

The NoBktrk algorithm will now try to find an $\mathrm{ALE}[\mathrm{i_s} = 11, \mathrm{i_d} = 3, \mathrm{l_e} = \text{‘A’}]$ action which relates to a device with identifier $\mathrm{i_d} = 3$, which does not exist in the interaction sequence. Therefore, it will remove the actions in the partial match from consideration. As a result, it will fail to find the $\mathrm{AS}[\mathrm{i_s} = 11, \mathrm{pos} = 5]$ action in future calls, and will not be able to fulfill the recipe.

To compute the complexity of SomeBktrk, let S be the maximum number of sub-actions in any recipe. A first-depth recursive call can be made at most $|\mathbf{X}|$ times. Within each of these recursive calls, at most $|\mathbf{X}| - 1$ actions can remain in OL'. So, at most $|\mathbf{X}| - 1$ second-depth recursive calls can be made for each first-depth recursive call, yielding an overall maximum of $|\mathbf{X}|(|\mathbf{X}| - 1)$ second-depth recursive calls. After $S - 1$ recursive calls of increasing depth have been made, a match must be completed or backtracking must occur. Within each lowest-depth recursive call, there can be at most $|\mathbf{X}| - (S - 1)$ actions left to consider. So, a worst-case complexity of SomeBktrk is $O(\frac{|\mathbf{X}|!}{S!})$.

5.2. Complete algorithms

In this section we present two plan recognition algorithms that are complete. Both algorithms work by converting the plan recognition problem into one or more constraint satisfaction problems and using standard techniques for their solution. The conversion makes use of the Expand function, shown in Fig. 15, to convert plans to flat representations containing solely basic actions, called *expanded recipes*. Note that like their conventional counterparts, expanded recipes also include constraints defined over their set of actions. The first complete algorithm performs the conversion naively, while the second use a cascade of conversions to significantly reduce the size of expanded plans.

Expand(T_A) takes as input a plan tree T_A for complex action A and returns a set of expanded recipes for A. Each and node represents a possible recipe for its parent node, a complex action. For each and node, Expand recursively generates all expanded recipes for each sub-action of the recipe. This algorithms alternates between two sub-procedures, DirectSum and Union. Given a recipe, the DirectSum procedure computes all possible replacements of complex sub-actions with basic actions. Each time a complex action is replaced, DirectSum ensures that all restrictions involving the complex action are propagated to its sub-actions. For example, consider a single recipe R_A with sub-actions $\underline{\mathrm{B}}$, $\underline{\mathrm{C}}$, and D:

$$\underline{\mathrm{A}} \rightarrow \underline{\mathrm{B}}, \underline{\mathrm{C}}, \mathrm{D}$$

with restrictions:

$$\underline{\mathrm{B}} \prec \mathrm{D}$$

$$\underline{\mathrm{C}} \prec \mathrm{D}$$

Suppose also that recursive calls of Expand have found the expanded recipes for $\underline{\mathrm{B}}$ to be $\{\mathrm{E}, \mathrm{F}\}$ and the expanded recipes for $\underline{\mathrm{C}}$ to be $\{\mathrm{G}, \mathrm{H}\}$ and $\{\mathrm{I}, \mathrm{J}\}$. In this case, DirectSum will return the following expanded recipes for $\underline{\mathrm{A}}$:

$$\{\mathrm{E}, \mathrm{G}, \mathrm{H}, \mathrm{D}\}, \{\mathrm{E}, \mathrm{I}, \mathrm{J}, \mathrm{D}\}, \{\mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{D}\}, \{\mathrm{F}, \mathrm{I}, \mathrm{J}, \mathrm{D}\}$$

with each recipe including either the restriction $\mathrm{E} \prec \mathrm{D}$ or $\mathrm{F} \prec \mathrm{D}$ in place of the $\underline{\mathrm{B}} \prec \mathrm{D}$ restriction, and including either the restriction $G \prec D$ and $H \prec D$, or $I \prec D$ and $J \prec D$ in place of the $\underline{\mathrm{C}} \prec \mathrm{D}$ restriction. Lastly, the Union sub-procedure takes the union over the expanded recipes generated for each recipe of $\underline{\mathrm{A}}$.

An expanded recipe is a series of basic actions (with associated restrictions) that the user may perform to realize a potential plan. To create an expanded recipe, a path is traversed through the plan tree, beginning at the root and ending with basic actions at the leaves. This path provides a trace of the plan corresponding to the expanded recipe. For example, one expanded recipe can be achieved by traversing the plan tree in Fig. 8 and choosing the leftmost recipe at each or node. Notice that the path taken matches the plan in Fig. 7. In this expanded recipe, each complex AED action and its restrictions are replaced with two basic actions, ALE and CEL, and corresponding restrictions, as shown in Fig. 14.

The complexity of Expand is costly in the worst case. Let S be the maximum number of *complex* sub-actions for each recipe, N be the maximum number of recipes for a *single* complex action, and C be the number of distinct complex actions. A plan tree has depth of at most $C + 1$, as we do not allow for recursive recipes. At the lowest depth of the plan tree, all actions are basic and do not have recipes. At the second lowest depth, complex actions have at most N expanded recipes, as none of the N recipes contain any complex sub-actions. At the third lowest depth, each recipe for a complex action may

$$\underline{\mathsf{CCD}} \rightarrow \mathsf{ADS}, \mathsf{ALE}_1, \mathsf{CEL}_1, \mathsf{ALE}_2, \mathsf{CEL}_2, \mathsf{CPD}$$

where

$$\mathsf{ADS} \prec \mathsf{ALE}_1, \mathsf{ALE}_2 \prec \mathsf{CPD}$$
$$\mathsf{CEL}_1, \mathsf{CEL}_2 \prec \mathsf{CPD}$$
$$\mathsf{ADS}[\mathsf{i_s}] = \mathsf{ALE}_1[\mathsf{i_s}] = \mathsf{CEL}_1[\mathsf{i_s}] = \mathsf{ALE}_2[\mathsf{i_s}] = \mathsf{CEL}_2[\mathsf{i_s}] = \mathsf{CPD}[\mathsf{i_s}]$$
$$\mathsf{ADS}[\mathsf{i_d}] = \mathsf{ALE}_1[\mathsf{i_d}] = \mathsf{CEL}_1[\mathsf{i_d}] = \mathsf{ALE}_2[\mathsf{i_d}] = \mathsf{CEL}_2[\mathsf{i_d}] = \mathsf{CPD}[\mathsf{i_d}]$$

Fig. 14. Expanded version of CCD recipe.

```
1: procedure EXPAND($T_C$)                                ▷ $T_C$: the plan tree for action $C$
2:     ERs[$C$] ← ∅                                       ▷ ERs[$C$]: the expanded recipes for $C$
3:     for all $r_j$, a child of $C$ do                   ▷ $r_j$: a recipe
4:         ERs[$r_j$] ← ∅
5:         for all $a_i$, a child of $r_j$ do             ▷ $a_i$: an action
6:             ERs[$r_j$] ← DIRECTSUM(EXPAND($T_{a_i}$), ERs[$r_j$])
7:         ERs[$a$] ← UNION(ERs[$a$], ERs[$r_j$])
8:     if ERs[$a$] = ∅ then
9:         ERs[$a$] ← {$a$}
10:    return ERs[$a$]
```

Fig. 15. Algorithm for generating expanded recipes.

```
1: procedure CONVERTTOCSP($\mathbf{E_A} = (S, R)$, $\mathbf{X}$)   ▷ $\mathbf{E}_A$: an expanded recipe $S$ and restrictions $R$ for
   complex action $A$, $\mathbf{X}$: an action sequence
2:     for all $s \in S$ do                                       ▷ $S$: a set of sub-actions
3:         ADDVARIABLEANDDOMAIN($s$, $\mathbf{X}$)
4:     for all $r \in R$ do                                       ▷ $R$: a set of restrictions
5:         ADDRESTRICTIONCONSTRAINT($r$)
6:     for all $s \in S$ do
7:         ADDREDUNDANCYCONSTRAINT($s$)
```

Fig. 16. Converting an expanded recipe and action sequence to a CSP.

contain at most S complex sub-actions, and each sub-action may have at most N recipes. The DIRECTSUM procedure then creates at most N^S expanded recipes per recipe. The UNION procedure collects the expanded recipes resulting from each recipe for that action, resulting in a maximum of $N(N)^S$, or N^{S+1}, recipes. At the fourth lowest depth, each complex action can again have at most N recipes with at most S complex sub-actions in each. Each of these S sub-actions can contain at most $N(N)^S$ expanded recipes. So, the DIRECTSUM and UNION procedures create at most $N(N(N)^S)^S$, or N^{S^2+S+1}, expanded recipes per recipe. Continuing this reasoning, the top-level action can have at most $N^{\sum_{i=0}^{C-1} S^i}$ recipes, yielding an overall complexity of $N^{O(S^C)}$.

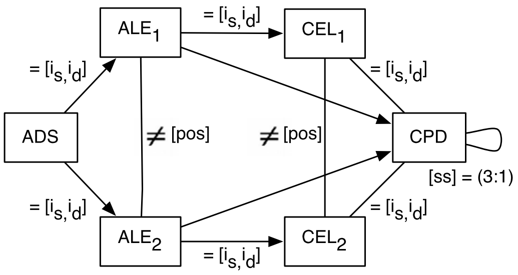
5.2.1. Constraint satisfaction algorithms

In this section we explain how to combine an expanded recipe and action sequence to create a constraint satisfaction problem (CSP). A solution to the resulting CSP is the plan representing the users' activities. Formally, a CSP is a triple (X, Dom, C), where $X = \{x_1, \ldots, x_n\}$ is a finite set of variables with respective domains $Dom = \{D_1, \ldots, D_n\}$, each a set of possible values for the corresponding variable, $D_i = \{v_1^i, \ldots, v_k^i\}$, and a set of constraints $C = \{c_1, \ldots, c_m\}$ that limit the values that can be assigned to any set of variables.

The algorithm CONVERTTOCSP, shown in Fig. 16, receives as input an expanded recipe E_A and an action sequence $\mathbf{X}$ and returns a CSP. If a solution exists for this CSP, a subset of the actions in $\mathbf{X}$ realize the expanded recipe E_A. We first show how to create variables in the CSP, and we use as a reference Fig. 17, which provides a graphical representation of the CSP resulting from the action sequence of Fig. 3 and the expanded recipe of Fig. 14. We used a graphical layout suggested by Dechter [21]. Note that parameters belonging to actions are not pictured unless they participate in some constraint.

Let $S = \{s_1, \ldots, s_n\}$ and R be the set of sub-actions and restrictions in the expanded recipe, respectively. Each action in S becomes a unique variable in the CSP by calling the subroutine ADDVARIABLEANDDOMAIN(s, $\mathbf{X}$). Based on the expanded recipe, six variables are added at this time: ADS, ALE_1, CEL_1, ALE_2, CEL_2, and CPD. These variables appear, outlined, in the graph of Fig. 17. Each variable's domain is then derived from the actions in the action sequence. For each occurrence of action s in the action sequence, a value is added to the domain of s in the CSP. The right-hand box of Fig. 17 gives the resulting domain for each variable based on the action sequence.

Lastly, we add restrictions to our CSP. For each restriction r in R over actions $(s_1, \ldots, s_m)$ in S, a constraint over the corresponding CSP variables is added to the CSP using the ADDRESTRICTIONCONSTRAINT(r) subroutine. At this point, all restrictions listed in Fig. 14 are added, including $\mathsf{CPD}[\mathsf{ss}] = (3:1)$. Directed edges in the figure represent temporal constraints between two variables. Undirected edges represent other parametric constraints. The edge from ADS to ALE_1 expresses the constraint $\mathsf{ADS} \prec \mathsf{ALE}_1$ as well as the constraint $\mathsf{ADS}[\mathsf{i_s}, \mathsf{i_d}] = \mathsf{ALE}_1[\mathsf{i_s}, \mathsf{i_d}]$.



Variable:	**Domain**

ADS:	ADS[i_s = 11, i_d = 2, t_d = spinner, pos = 6],
	ADS[i_s = 11, i_d = 3, t_d = mixer, pos = 13]

ALE$_1$,ALE$_2$:	ALE[i_s = 11, i_d = 2, i_e = 1, l_e = a, pos = 7],
	ALE[i_s = 11, i_d = 2, i_e = 2, l_e = b, pos = 8]

CEL$_1$,CEL$_2$:	CEL[i_s = 11, i_d = 2, i_e = 1, l_e = rain, pos = 10],
	CEL[i_s = 11, i_d = 2, i_e = 2, l_e = sun, pos = 11]

CPD:	CPD[i_s = 11, i_d = 2, ss = 1 : 3, pos = 12]

Fig. 17. CSP resulting from an action sequence and an expanded recipe.

```
procedure CSPBRUTE($T_C$, $\mathbf{X}$)          ▷ $T_C$: the plan tree for action $C$, $\mathbf{X}$: an action sequence
    $E \leftarrow$ EXPAND($T_C$)                  ▷ $E$: a set of expanded recipes
    for all $e \in E$ do
        $\mathbf{C} \leftarrow$ CONVERTTOCSP($e$, $\mathbf{X}$)          ▷ $\mathbf{C}$: a CSP
        $solution \leftarrow$ SOLVE($\mathbf{C}$)
        if $solution \neq \emptyset$ then
            return $solution$
    return $\emptyset$
```

Fig. 18. Brute-force algorithm.

```
procedure CSPPRUNE($C$, $\mathbf{R}$, $\mathbf{X}$)          ▷ $\mathbf{R}$: a recipe library, $\mathbf{X}$: an action sequence
    $T_C \leftarrow$ CREATERECIPETREE($C$, $\mathbf{R}$)          ▷ $T_C$: the plan tree for action $C$
    Perform a bottom-up traversal of $T_C$.
    for each OR node representing action $A$ do
        if $A$ has not been cached then
            Cache $solution[A] \leftarrow$ CSPBRUTE($T_A$, $\mathbf{X}$)
        if $solution[A] = \emptyset$ then
            Prune parent of $A$ from $T_A$
    return $solution[C]$
```

Fig. 19. Bottom-up algorithm.

For variables corresponding to the same action, additional redundancy constraints are added using the ADDREDUNDANCYCONSTRAINT subroutine. These constraints ensure that such variables are assigned distinct values, as these variables share the same domain. An example is the constraint connecting the ALE$_1$ and ALE$_2$ variables, which requires that these variable assignments have distinct pos parameters.

5.2.2. Brute-force algorithm

A solution for a CSP provides a match between an expanded recipe and an action sequence. In this section we present two algorithms that use CSPs to output a plan from an action sequined $\mathbf{X}$ for a desired complex action C given a set of recipes $\mathbf{R}$.

The first algorithm, shown in Fig. 18, takes a brute-force approach, calling EXPAND to generate each expanded recipe for C, converting it to a CSP and solving the CSP. This algorithm returns the first solution found to the CSP or $\emptyset$ if no solution is found.

The complexity of CSPBRUTE can be analyzed in terms of the FINDMATCH2 and EXPAND procedures. Recall that calling EXPAND results in at most $N^{O(S^C)}$ expanded recipes, where N is the maximum number of recipes for a single complex action. In the worst case, all expanded recipes are considered, and for each expanded recipe a CSP solver must be run. The complexity of this CSP solver can be bounded by the complexity of a complete backtracking search, which we have seen to be $\frac{|\mathbf{X}|!}{S!}$. So, an overall worst-case complexity of CSPBRUTE is $N^{O(S^C)}O(\frac{|\mathbf{X}|!}{S!})$.

5.2.3. Pruning algorithm

The second algorithm, shown in Fig. 19, takes a more sophisticated approach and traverses the plan tree from the bottom-up. At each OR node, the algorithm determines whether the user completed the corresponding sub-action by either calling CSPBRUTE or referring to the cached result of an earlier CSPBRUTE call. If the user failed to complete that sub-action, the algorithm prunes the relevant recipe, the parent of the current node, from the tree, as the user cannot complete a recipe without completing each sub-action in that recipe. By eliminating branches from the plan tree for the desired complex action C, this pruning process narrows the search space of possible expanded recipes for root action C. This algorithm was suggested by Quilici et al. [42].

The CSPPRUNE method calls the CSPBRUTE algorithm once per distinct complex sub-action. Let C again represent the number of distinct complex sub-actions in our recipe list. Then, the worst-case complexity of the pruning method is $N^{O(S^C)}O(\frac{C|\mathbf{X}|!}{S!})$. The worst case occurs when the user has completed each of the C complex actions, causing no poten-

Table 1
Number of possible plans per problem, maximal number of complex actions per recipe (S), maximal number of recipes for a single complex action (N), maximal number of distinct complex actions (C).

	# possible plans	S	N	C
ROSA	$O(2^6)$	4	2	13
RAIN	$O(2^{17})$	4	2	6
EARRINGS	$O(2^{11})$	4	2	6
SEATBELTS	$O(2^{12})$	4	4	6

tial expanded recipes to be eliminated. However, we hypothesized that users would be likely to solve TinkerPlots problems just once, and therefore some complex actions within the plan trees would not be matched in the action sequence.

6. Evaluation

The plan recognition problem inherent in domains such as TinkerPlots, where agents are engaged in exploratory behavior with false starts in addition to successful plan construction, leads, as we have shown, to an NP-complete computation. The algorithms presented in the previous section, both incomplete and complete, were intended to allow solution of real-world problems in practice, despite the complexity issue inherent in the problem. To determine their real-world performance, we collected actual logs of TinkerPlots usage on standard pedagogical problems, and compared the algorithms' coverage and performance. The results show that the best of the algorithms has excellent coverage and practical performance.

6.1. Experimental design

We collected interaction sequences of people's interaction with TinkerPlots in two different settings. The first setting included 12 adults with a wide variety of educational backgrounds, ranging from some high school to some post-graduate education. The second setting included 12 eighth grade students in a middle school in Cambridge MA.[8] Each adult subject received an identical 30-minute tutorial about TinkerPlots and was then asked to complete four problems in succession; these problems are detailed in Appendix A. Students were given a slightly longer 45 minute demonstration of the software and were asked to solve two of the four problems. User logs and videos of the users' screens were recorded for all user sessions. TinkerPlots is equipped with a logging facility that records the basic actions that make up users' action sequences. To evaluate the various plan recognition algorithms, we manually traced the videos of their interaction with TinkerPlots. We noted whether each problem was solved, and we constructed the (possibly multiple) plans used to solve the problem. We define a recognition algorithm to be "correct" if the plans that it outputs exactly corresponds to the plans constructed from the videos, or if it fails to output a plan when the student did not successfully complete the problem as determined by an expert.[9] If a user has solved a problem in several different ways, a recognition algorithm is deemed correct if it recognizes any of these solutions.

We created a set of recipes to the TinkerPlots problems in our study to serve as input to the plan recognition algorithms. Our purpose in this study was to evaluate algorithms for matching these ideal solutions with the appropriate basic level actions in users' interaction sequences. Therefore, the recipes we manually constructed were created prior to the collection of and were not informed by the data of people's interactions with TinkerPlots. Rather, they represented what we perceived a priori to be a broad range of possible solutions for TinkerPlots problems. Ultimately, our database contained recipes sufficient to explain all but three user interactions. The lack of inclusion of these recipes is discussed in Section 6.2.2. The recipe library used to run the recognition algorithms on each problem consisted of those recipes that were constructed for the problem. This corresponds to knowing which TinkerPlots problem students are trying to solve. This assumption is logical in the context of pedagogical software.[10]

6.2. Results and discussion

We compared the performance of the four recognition algorithms presented, called the NoBktrk, SomeBktrk, CSPbrute, and CSPprune techniques.[11] We analyzed the user logs corresponding to the problems outlined in Appendix A. Table 1 lists features for each problem that affect the complexity analysis of Section 5.1.1.

The analyzed user logs ranged in length from 14 to 80 actions. The average length of an interaction sequence for problems collected from adult subjects was 35 actions. Adults solved the assigned problems 70% of the time. In contrast, the

[8] Appropriate IRB approval was obtained for both settings, and parental consent was obtained for the data collected from the eighth graders.

[9] The domain expert was a researcher of educational technology who has worked with TinkerPlots for several years. For each action sequence, the expert was shown a movie of the desktop of the user, as well as the plan outputted by the algorithm.

[10] Dropping this assumption corresponds to running our algorithms using a recipe data base that contains the complete set of recipes for all problems. This may affect correctness for the greedy algorithms, but not for the complete algorithms.

[11] We used the python-constraint package created by Gustavo Niemeyer and available at http://labix.org/python-constraint to implement our constraint satisfaction algorithms.

Table 2
Accuracy of recognition algorithms by percentage. Parenthesized numbers are the number of logs.

Data	Problem		Algorithm							
			NoBktrk		SomeBktrk		CSPbrute		CSPprune	
Adults + Students	rosa	(23)	78%	(18)	87%	(20)	100%	(23)	100%	(23)
	rain	(16)	88%	(14)	88%	(14)	100%	(16)	100%	(16)
Adults	seat belts	(12)	42%	(5)	67%	(8)	75%	(9)	75%	(9)
	earrings	(11)	91%	(10)	100%	(11)	100%	(11)	100%	(11)
	Overall	(62)	76%	(47)	85%	(53)	95%	(59)	95%	(59)

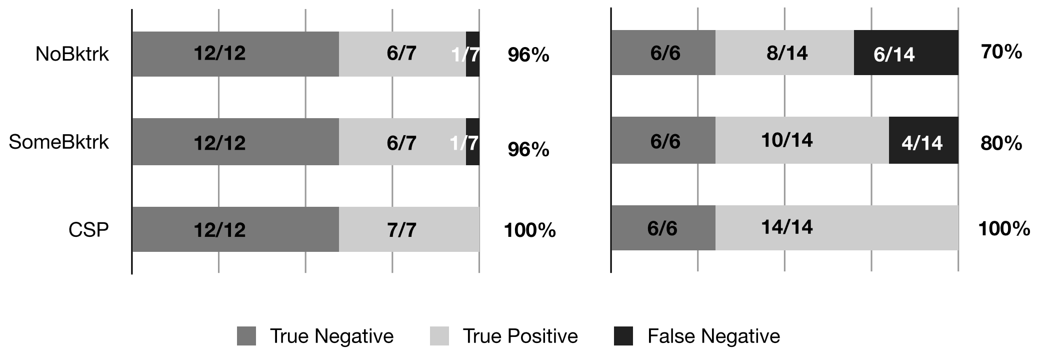


Fig. 20. Performance for data obtained from students (left) and adults (right).

average length of an interaction sequence for problems collected from students was 68 actions. Students solved the assigned problems 60% of the time. Also, people engaged in exploratory behavior using the software. For example, there were on average 15 exogenous actions in each problem that was obtained from adults.

Table 2 shows the accuracy of the recognition algorithms on data collected for students and adults. As shown by the table, the heuristic algorithms NoBktrk and SomeBktrk were correct for 47 of 62 (76%) and 53 of 62 (85%) interactions, respectively. The heuristic algorithms were outperformed by both CSPbruteForce and CSPprune algorithms, which performed correctly for 59 of 62 (95%) interactions. The incorrect inferences in all algorithms were false negatives, that is, the algorithms were unable to find solutions existing within the interaction sequence. All of the solutions outputted by all of the algorithms matched the expert's opinion regarding the activities used by the students.

The constraint satisfaction algorithms are guaranteed to find users' plans if they exist, provided that the relevant recipes are contained in the recipe database. Each of the three incorrect inferences of these algorithms can be traced to recipes missing from the database. In addition to incorrect inferences due to missing recipes, the incomplete approaches suffered from prematurely committing to a match without being able to backtrack. An example of the latter case occurred frequently within the seat belts problem (see Appendix A). This problem required the user to construct a conditional distribution representing the fact that people wearing seat belts are less likely to be hurt in an accident. Some users created a sampler with the wrong distribution parameters and proceeded to correct these parameter settings. The incomplete approaches tended to match sub-recipes for creating a sampler with actions corresponding to the wrong parameters. Consequently, they failed to find matches for the sampler construction recipe. In contrast, the CSP algorithms were able to backtrack and pick the right match for the expanded sampler construction recipe.

6.2.1. Performance of algorithms on ecologically realistic data

There was a significant difference in the performance of the heuristic algorithm on data that was obtained from adults and the data that was obtained from middle school students. Fig. 20 shows the average accuracy of the plan recognition algorithms on student data (left, 19 instances) and adult data (right, 20 instances) from two problems, rosa and rain.[12] The figure details the performance of these algorithms when correctly identifying successful solutions (true positives), correctly identifying failed solutions (true negatives), and incorrectly identifying failed solutions (false negatives).

As shown in Fig. 20, both of the heuristic algorithms were always able to recognize unsolved problems (true negatives). There were 6 such cases for adult data and 12 cases for student data. More generally, for student data the accuracy of both heuristic algorithms averaged 96%, just 4% lower than the accuracy of the complete CSP algorithms for this data. However, for adult data, the average accuracy of the heuristic algorithms was 75%, significantly lower than that the average accuracy of the CSP algorithms. This is also apparent from Table 3, which compares the precision and recall measures of the different approaches on student and adult data. All of the algorithms achieved perfect precision, because there were no

[12] We performed analysis on these two problems for which we have both student and adult data.

Table 3
Precision (left) and recall (right) measures for student and adult data.

Data	Algorithm		
	NoBktrk	SomeBktrk	CSP
Student	(1, 0.86)	(1, 0.86)	(1, 1)
Adult	(1, 0.57)	(1, 0.71)	(1, 1)

false positive classifications, as we described above. The complete CSP algorithms both achieved perfect recall because they did not incur false negatives. For the heuristic approaches, both NoBktrk and SomeBktrk algorithms achieved lower recall measures than the CSP approaches for both student and adult data. However, the average recall measure achieved by the heuristic approaches for student data was significantly higher than the average recall they achieved for adult data.

To explain the effect of the type of the setting on the various plan recognition algorithms, we compared the number of times in which students and adults were able to solve the problems that were assigned to them. As shown in Fig. 20 students were able to solve TinkerPlots problems less often than adults (for students, 12 data instances represented failures out of a total of 19 instances; for adults, 6 data instances represented failures out of a total of 20 instances). We attribute this to the fact that students were more likely to engage in exploratory behavior or to make mistakes while using TinkerPlots, as is attested by their longer interaction sequences. Thus, one explanation for the success of the heuristic algorithms when analyzing student data is that there were more true negative classifications for student data, and these were always recognizable by our recognition algorithms.

However, the performance of the heuristic algorithms on student data cannot be attributed solely to the fact that there were more true negative classifications for student data. Both of the heuristic algorithms also achieved higher accuracy rates for students than for adults on data instances representing true positives. As shown by Fig. 20, for student data, the heuristic algorithms were able to recognize 6 out of 7 true positive instances (average accuracy 85%), but were only able to recognize 8 and 10 out of 14 true positive instances (average accuracy 65%) for adult data. Thus, both NoBktrk and SomeBktrk algorithms were better at recognizing successful and unsuccessful plans for students than for adults. These results provide additional support for the applicability of these algorithms, showing that they are particularly suitable for data that was obtained in an ecologically realistic fashion, rather than a lab setting.

6.2.2. *Limitations of approaches*

A significant hurdle to accurate recognition is the difficulty of expressing certain types of user strategies as recipes. As previously mentioned, both the greedy and constraint satisfaction algorithms were susceptible to the fact that the recipe database failed to capture all of the possible ways in which people solved problems. Because we do not allow recipes to be defined recursively, our recipes can only specify that a fixed, rather than variable, number of actions occur. In the RAIN problem, for instance, there are several possible samplers that model the probability of rain as 75%. A natural solution is one (as in Fig. 1(a)) in which there are unique events for "rain" and "sun", causing "rain" to be weighted as three times more likely than "sun". In actuality, any proportional number of "rain" and "sun" events would suffice, as long as "rain" is three times more likely than "sun". This limitation prevented our library from succinctly expressing the infinite number of permissible strategies—three of which were used by students—for the SEAT BELTS problem.

Although there were no false-positive classifications in our experimental sessions, they can happen in theory. One reason for this is that the recipe language cannot express actions that must not occur. For example, the ROSA problem requires the user to create a device with four events labeled *R*, *O*, *S*, and *A*. If a fifth event were added, the sampler would no longer be correct. However, all of our recognition algorithms would incorrectly match the actions corresponding to the four previous values with the recipe for creating a sampler.

It is possible to construct plans using recipes that cannot happen in practice because they are disallowed by the software. For example, two interleaved actions may add and delete each other's preconditions.

Finally, we note that our recipes are not designed to describe approximately correct user approaches. For example, one problem required users to add two identically labeled events to a device, without specifying what that label must be. One user failed to complete this task according to the recipe database because he or she used event labels "pierced" and (sic) "piecred". Despite this oversight, a teacher would likely consider this strategy a successful one. However, our algorithms were unable to distinguish this mistake from other, more conceptual mistakes. One way to overcome such difficulties is to search for the "closest" plan for explaining users' TinkerPlots activities in terms of recipes, rather than searching for only complete and correct plans. This can be done by using flexible CSP solvers, which search for solutions that minimize the number of violated constraints.

6.2.3. *Performance considerations*

In this section we compare the performance incurred in practice by the four recognition algorithms we have presented (measured as run-time on a commodity computer). By working to recognize interleaving plans in user logs containing incorrect strategies and up to 80 actions, we test whether the worst-case exponential complexity presents a significant barrier to real-world plan recognition. In Table 4 we present run-times for each of our four algorithms organized by increasing theoretical computational complexity. These results are averaged over all action sequences for each problem.

Table 4
Average runtime (in seconds) of recognition algorithms,

Problem	Algorithm			
	NoBktrk	SomeBktrk	CSPbrute	CSPprune
rosa	0.15	3.34	0.11	0.09
rain	0.35	0.54	131.54	21.88
seat belts	0.02	0.34	6.07	8.42
earrings	0.01	1.82	6.92	3.99
Overall	0.13	1.54	36.75	8.02

Table 5
Average number of CSPs and variables for constraint satisfaction algorithms.

Problem	CSPbrute		CSPprune	
	CSPs	Variables	CSPs	Variables
rosa	19	18	14	4
rain	9300	29	141	9
seat belts	347	23	298	11
earrings	292	21	176	14
Overall	2490	23	157	10

As shown in the table, the average run-time of the heuristic approaches, which employed limited or no backtracking, was significantly faster than that of the complete approaches. Within the complete approaches, the pruning algorithm was significantly faster than the brute-force algorithm. This is because there were "dead-ends" in the recipe tree that the pruning algorithm was able to exploit. As can be seen in the table, CSPprune outperformed the CSPbruteForce algorithm, and in reasonable time, despite having greater worst-case complexity. However, in cases where the pruning approach fails to eliminate branches from the plan-trees, it may turn out to be slower than the brute-force approach. For example, as shown in the table, for the seat belts problem, the CSPbruteForce algorithm was faster than the CSPprune algorithms.

To examine the relationship between runtime and student interaction length, we measured the correlation between these variables for each algorithm. The heuristic algorithms NoBktrk and SomeBktrk showed positive correlations between runtime and interaction length of .752 and .508, respectively. The CSPbruteForce algorithm showed a negative correlation of −.333, while CSPprune showed a very weak, positive correlation of .050. Also, as shown in Table 4, the complete approaches were significantly slower when running on rain than when running on the other problem. The number of possible plans for rain, as shown in Table 1, was significantly higher than the number of possible plans for the other problems.

These results support our complexity findings that the bottleneck of the incomplete algorithms is the size of the user's interaction log, while the bottleneck of the complete algorithms is the complexity of the recipe library. These results also demonstrate the applicability of our algorithms to be used in actual classrooms. As shown in Table 1, the number of possible plans for some TinkerPlots problems is very large. Despite this fact, the almost-perfect record of the heuristic approach, and the short runtime of the complete approaches on these instances speaks well for their overall performance.

Lastly, to further compare the performance of our constraint satisfaction algorithms, Table 5 presents two additional statistics: the average number of CSPs built per log and the average number of variables contained in each CSP. CSPprune outperforms CSPbruteForce in each category, building fewer and smaller CSPs, on average.

7. Future work and conclusion

This paper investigated a class of plan recognition problems for domains in which agents engage widely in exploratory behavior. We showed that constraint satisfaction algorithms are a viable and practical approach for plan recognition in one such domain, that of an educational software application. These algorithms were able to correctly capture users' plans in real-world logs of users' sessions in reasonable time despite the theoretical worst-case behavior and the flexible nature of the software. The algorithms compared favorably to faster but greedier approaches.

This work is a first step towards a pedagogical agent that is truly collaborative, in the sense that it provides useful machine-generated support to teachers and students. For teachers, this support consists of information about students' performance both during and after class. For example, presenting teachers with a visualization of students' plans will convey whether and how students solved a particular problem more quickly than would be possible if they had to analyze snapshots. Teachers can also benefit from the fact that our algorithms capture false starts and incorrect solutions, alerting them to mistakes and misconceptions by the students. Existing systems for assessing students' performance with pedagogical software work in highly constrained settings, and report simple statistics, such as the number of correct answers solved [22]. Our work extends such systems to exploratory domains, in which students' performance can be explained in part by inferring their plans. In a recent user study we conducted with teachers using TinkerPlots in the classroom, teachers favored the plan-based presentation to other types of visualizations, such as seeing selected snapshots of the students' work.

We are currently extending these results in two ways. First, we are developing methods for presenting plan recognition output to teachers in order to provide them with a broad and organized view of students' activities. Second, we are evaluating the ability of the algorithms to generalize to a different pedagogical software system for teaching chemistry to college students [51].

This work raises several new opportunities for involving the use of plan recognition algorithms as a basis for building intelligent tutors that will augment existing software tools for mathematics education. These collaborative tutors will provide machine-generated support that decides when and how to intervene with students based on teachers' feedback to the plans inferred by the system. They will contribute to the thoughtful analysis of probabilistic models by students and increased ability of teachers to identify those students who would benefit from teacher advice. Together these abilities should lead to improvements in both teaching and learning.

Acknowledgements

This work was supported by NSF grant number REC-0632544. We are indebted to Craig Miller for developing the TinkerPlots logging facility. Thanks very much to the anonymous reviewers for their helpful comments. Thanks to Elif Yamangil for assisting with the development of the greedy algorithm, and to Cliff Konold for helpful discussions. A special thanks goes to Ofra Amir for reading and commenting on previous drafts of this work.

Appendix A. Experimental problems

We detail the four TinkerPlots questions posed to subjects and considered in our empirical evaluation of the algorithms.

ROSA: Jessica has 4 letters printed on cards: R, O, S, and A. After mixing them up, she blindly picks the 4 letters one at a time and arranges them in line in the order she chose them. Build a TinkerPlots model and use it to help you estimate the probability of Jessica spelling the word ROSA.

RAIN: There is a 75% chance of rain for each of the next 4 days. Build a TinkerPlots model and use it to help you estimate the probability of getting rain on all 4 days.

SEAT-BELTS: If you get into an accident, you are much less likely to be injured if you are wearing your seat belt. Build a TinkerPlots model for people that
1. are either wearing seat belts or not,
2. then are either injured in accident or not injured in an accident.

Design your factory so that the people wearing seat belts are less likely to get injured than those not wearing them. In your model, what is the probability of people being injured in an accident? What is the probability of being injured in an accident when you are wearing a seat-belt?

EARRINGS: Build a TinkerPlots factory that
1. makes people that are girls or boys,
2. then either pierces their ears or not.

According to your model, what is the probability that
1. a boy has a pierced ear?
2. a girl has a pierced ear?

According to your model, approximately what fraction of people you meet on the street will have pierced ears?

References

[1] D.W. Albrecht, I. Zukerman, A.E. Nicholson, Bayesian models for keyhole plan recognition in an adventure game, User Modeling and User-Adapted Interaction 8 (1) (1998) 5–47.
[2] J.R. Anderson, A.T. Corbett, K.R. Koedinger, R. Pelletier, Cognitive tutors: Lessons learned, The Journal of Learning Sciences 4 (2) (1995) 167–207.
[3] D. Avrahami-Zilberbrand, G.A. Kaminka, Fast and complete symbolic plan recognition, in: Proceedings of the International Joint Conference on Artificial Intelligence, vol. 14, 2005.
[4] T. Babaian, B.J. Grosz, S.M. Shieber, A writer's collaborative assistant, in: Intelligent User Interfaces Conference, 2002, pp. 7–14.
[5] G.E. Barton, On the complexity of ID/LP parsing, Computational Linguistics (ISSN 0891-2017) 11 (4) (1985) 205–218.
[6] M. Bauer, Acquisition of user preferences for plan recognition, in: Proceedings of the Fifth International Conference on User Modeling, 1996, pp. 105–112.
[7] M. Bauer, S. Biundo, D. Dengler, J. Koehler, G. Paul, A logic-based tool for intelligent help systems, in: Proc. 13th International Joint Conference on Artificial Intelligence (IJCAI), 1993.
[8] J.E. Beck, B.P. Woolf, Using a learning agent with a student model, in: Proc. of 4th International Conference on Intelligent Tutors, 1998.
[9] N. Blaylock, J. Allen, Recognizing instantiated goals using statistical methods, in: Workshop on Modeling Others from Observations, 2005, pp. 79–86.
[10] M.E. Bratman, Intention, Plans, and Practical Reason, Harvard University Press, Cambridge, MA, 1987.
[11] M.E. Bratman, D.J. Israel, M.E. Pollack, Plans and resource-bounded practical reasoning, Computational Intelligence 4 (3) (1988) 349–355.
[12] J.S. Brown, R.R. Burton, Diagnostic models for procedural bugs in basic mathematical skills, Cognitive Science 2 (2) (1978) 155–192.
[13] H.H. Bui, A general model for online probabilistic plan recognition, in: Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI), 2003.
[14] S. Carberry, Plan Recognition in Natural Language Dialogue, MIT Press, 1990.
[15] S. Carberry, Techniques for plan recognition, User Modeling and User-Adapted Interaction 11 (1) (2001) 31–48.
[16] E. Charniak, R.P. Goldman, A Bayesian model of plan recognition, Artificial Intelligence 64 (1) (1993) 53–79.

[17] P.R. Cohen, C.R. Perrault, J.F. Allen, Beyond question-answering, in: W. Lehnert, M. Ringle (Eds.), Strategies for Natural Language Processing, 1981, pp. 245–274.
[18] C. Conati, A. Gertner, K. VanLehn, Using Bayesian networks to manage uncertainty in student modeling, Journal of User Modeling and User-Adapted Interaction 12 (4) (2002) 371–417.
[19] A. Corbett, M. McLaughlin, K.C. Scarpinatto, Modeling student knowledge: Cognitive tutors in high school and college, User Modeling and User-Adapted Interaction 10 (2000) 81–108.
[20] D. Davidson, Intending Essays on Actions and Events, Clarendon Press, 1980, pp. 83–102.
[21] R. Dechter, Constraint Processing, Morgan Kaufmann, 2003.
[22] M. Feng, N. Heffernan, K. Koedinger, Addressing the assessment challenge with an online system that tutors as it assesses, User Modeling and User-Adapted Interaction (ISSN 0924-1868) 19 (3) (2009) 243–266.
[23] Y. Gal, E. Yamangil, A. Rubin, S.M. Shieber, B.J. Grosz, Towards collaborative intelligent tutors: Automated recognition of users' strategies, in: Proceedings of Ninth International Conference on Intelligent Tutoring Systems (ITS), Montreal, Quebec, 2008.
[24] G. Gazdar, Generalized Phrase Structure Grammar, Harvard Univ. Press, 1985.
[25] C.W. Geib, M. Steedman, On natural language processing and plan recognition, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007, pp. 1612–1617.
[26] C.W. Geib, R.P. Goldman, A probabilistic plan recognition algorithm based on plan tree grammars, Artificial Intelligence 173 (11) (2009) 1101–1132.
[27] M. Ghallab, D.S. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann Publishers, 2004.
[28] R.P. Goldman, C.W. Geib, C.A. Miller, A new model of plan recognition, in: Proc. 15th Conference on Uncertainty in Artificial Intelligence (UAI), 1999.
[29] B.J. Grosz, C.L. Sidner, Plans for discourse, Intentions in Communication (1990) 417–444.
[30] J.K. Hammerman, A. Rubin, Strategies for managing statistical complexity with new software tools, Statistics Education Research Journal 3 (2) (2004) 17–41.
[31] E. Horvitz, Principles of mixed-initiative user interfaces, in: Proc. of ACM SIGCHI Conference on Human Factors in Computing Systems, 1999.
[32] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Springer, New York, 1972, pp. 85–103.
[33] H.A. Kautz, A formal theory of plan recognition, PhD thesis, University of Rochester, 1987.
[34] C. Konold, C. Miller, TinkerPlots Dynamic Data Exploration 1.0, Key Curriculum Press, 2004, http://www.keypress.com/x5715.xml.
[35] N. Lesh, Adaptive goal recognition, in: Proceedings of the 15th International Joint Conference on Artificial Intelligence, 1997, pp. 1208–1214.
[36] K.E. Lochbaum, A collaborative planning model of intentional structure, Computational Linguistics 4 (1998) 525–572.
[37] J. Mayfield, Controlling inference in plan recognition, User Modeling and User-Adapted Interaction 2 (1) (1992) 55–82.
[38] D.S. Nau, S.J.J. Smith, K. Erol, et al. Control strategies in HTN planning: Theory versus practice, in: Proceedings of the National Conference on Artificial Intelligence, 1998, pp. 1127–1133.
[39] M.E. Pollack, Some requirements for a model of the plan inference process in conversation, in: Communication Failure in Dialogue and Discourse: Detection and Repair Processes, 1987.
[40] M.E. Pollack, Plans as complex mental attitudes, Intentions in communication, 1990.
[41] D.V. Pynadath, M.P. Wellman, Probabilistic state-dependent grammars for plan recognition, in: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, 2000, pp. 507–514.
[42] A. Quilici, Q. Yang, S. Woods, Applying plan recognition algorithms to program understanding, Automated Software Engineering 5 (3) (1998) 347–372.
[43] S. Reddy, Y. Gal, S.M. Shieber, Recognition of users' activities using constraint satisfaction, in: Proceedings of the First and Seventeenth International Conference on User Modeling, Adaptation and Personalization, 2009.
[44] K. Ryall, J. Marks, S.M. Shieber, An interactive constraint-based system for drawing graphs, in: Proceedings of the 10th Annual Symposium on User Interface Software and Technology (UIST), 1997.
[45] S.M. Shieber, Direct parsing of ID/LP grammars, Linguistics and Philosophy 7 (2) (1984) 135–154.
[46] C.L. Sidner, Plan parsing for intended response recognition in discourse, Computational Intelligence 1 (1) (1985) 1–10.
[47] K. VanLehn, C. Lynch, K. Schulze, J.A. Shapiro, R.H. Shelby, L. Taylor, D.J. Treacy, A. Weinstein, M.C. Wintersgill, The Andes physics tutoring system: Lessons learned, International Journal of Artificial Intelligence and Education 15 (3) (2005).
[48] M.B. Vilain, Getting serious about parsing plans: A grammatical analysis of plan recognition, in: AAAI, 1990, pp. 190–197.
[49] R. Weida, D. Litman, Terminological reasoning with constraint networks and an application to plan recognition, in: Proc. of the 3rd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR-92), Citeseer, 1992, pp. 282–293.
[50] R. Wilensky, Why John married Mary: Understanding stories involving recurring goals, Cognitive Science 2 (3) (1978) 235–266.
[51] D.J. Yaron, M. Karabinos, A. Borek, B. McLaren, K.L. Evans, G. Leinhardt, Tracking and supporting learning in open-ended activities involving a virtual lab simulation, in: The 238th American Chemical Society National Meeting, Washington, DC, 2009.