Recognition of Users' Activities using Constraint Satisfaction

Swapna Reddy, Ya'akov Gal, Stuart Shieber

TR-05-09

Computer Science Group
Harvard University
Cambridge, Massachusetts

Recognition of Users' Activities using Constraint Satisfaction

Swapna Reddy, Ya'akov Gal, and Stuart Shieber

School of Engineering and Applied Sciences
Harvard University
Cambridge MA 02138

Abstract. Ideally designed software allow users to explore and experiment during their interaction, and to pursue multiple, interleaving plans. This makes it challenging to automatically recognize users' interactions with such software. This paper shows that this recognition problem can be formalized and solved using constraint satisfaction techniques. It constructs algorithms that use these techniques to recognize users' activities by comparing their interaction histories with a set of ideal solutions. These plan-recognition algorithms are complete, in the sense that they are guaranteed to recognize users' plans if they exist. We evaluate these algorithms empirically on data obtained from different people solving mathematical problems using commercially available pedagogical software. In all cases, these algorithms were able to identify users' activities with the software and distinguish between those actions that played a salient part in the solution and redundant, exploratory actions.

1 Introduction

Developments in technology and connectivity have increased the prevalence and reach of computer systems in our daily lives. A prime example of this effect is the use of computer systems in settings that until recently did not directly involve computers, like pedagogy, and support systems for the cognitively-impaired [1]. In many of these environments, computer systems serve a role that aids and complements the work of human professionals and care-givers. Consider for example, a system for teaching mathematics to students, which includes teachers, a computer tutor, and students. Teachers wish to know the individual strengths and weaknesses of each student and to assess their performance as a group. Thus, in addition to interacting with the students, the computer tutor needs to assess and understand their actions.

Therefore, a key requirement for computer systems in these settings is to recognize the activities of their users. This is important for (1) informing care-givers about the performance and difficulties encountered by individuals users, (2) facilitating machine-generated support for the purpose of guiding the user in its interactions, (3) providing software developers with information relating to how the software is used and to alert about bugs, system loads, and so forth.

Traditional approaches to plan recognition assume goal-oriented agents whose actions are consistent with their knowledge base and who form a single encompassing plan towards achieving their goal. In contrast, an objective of flexible software is to allow users to explore and experiment during their interaction process. Users may interchangeably pursue multiple, interleaving plans; they may be confused about which appropriate plan to take, and they may make mistakes. Clearly, reasoning about every possible way in which a user can interact in this systems is infeasible.

This paper reports on the development of plan-recognition algorithms for meeting these challenges. These algorithms work work directly on users' interaction histories and compare them to ideal solutions designed by domain experts. They are able to infer users' plans for solving problems as well as the extent to which these plans differed from ideal solutions. All of the algorithms make use of constraint satisfaction techniques to formalize a match between users' interaction histories and potential solutions. Constraint satisfaction problems (CSP) have been used to express and solve a wide spectrum of problems in computer science. We show the utility of these approaches towards the construction of plan recognition algorithms that are complete and correct, in the sense that the plans they derive necessarily match users' activities, and that these plans are guaranteed to be found. We show empirically that these computational guarantees do not impede their performance.

We provide an empirical evaluation of our algorithms using a commercially-available pedagogical software for mathematics education, used in schools worldwide. This software provides users with a "construction kit" that allows students to create and analyze a large number of statistical models [2]. We illustrate how to formalize plan-recognition in this software, and evaluate our algorithms on interactions obtained from twelve subjects using this software to solve problems. We show that our methods outperform a recently proposed approach for inferring users' activities which was not able to recognize some of the solutions. Lastly, we discuss the trade-offs between these algorithms in terms of their computational complexity.

1.1 Prior Work

Past approaches for inferring users' activities with software have relied on querying the user for clarification about his or her actions in order to reduce the search space of possible plans [3, 4]. However, interrupting users impedes their satisfaction and performance, and the information they provide to the system cannot be assumed to be correct or helpful.

Gal et al. [5] proposed a non-intrusive approach for recognizing users' activities based on a greedy algorithms that uses various heuristics to match between users' interactions and solutions. This algorithm is not guaranteed to recognize a solution in the interaction history, and in case that the user solved the problem several times, it will only identify one of those solutions. Other non-intrusive approaches employed machine learning to predict users' activities given their past patterns of behavior [6, 7]. These methods assume that people are likely

to repeat particular behavioral patterns because they usually perform the same activities (e.g., deleting files, writing reports). These assumptions do not hold for pedagogical software, where students build on their past interactions with the software to solve new problems, and their goals may change dynamically as they interact with the software. In addition, the model parameters must be trained from data or stipulated by domain experts. Both of these techniques require considerable effort in pedagogical domains.

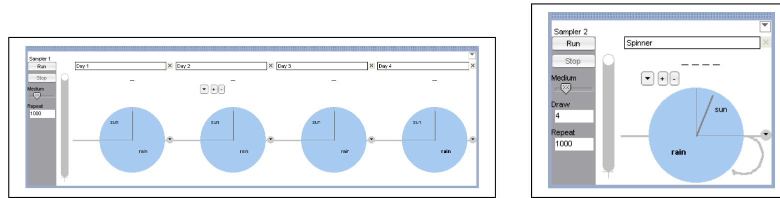
Lastly, our work is distinguished from probabilistic approaches that attempt to predict users' actions given their past interaction[8, 9]. Our work addresses a different problem, that of recognizing complete plans given entire interaction histories. Straightforward adaptation of probabilistic techniques for this purpose is not possible because the size of probabilistic models is typically exponential in the length of the history they consider, and users' complete interaction histories may span hundreds of actions.

2 The TinkerPlots Domain

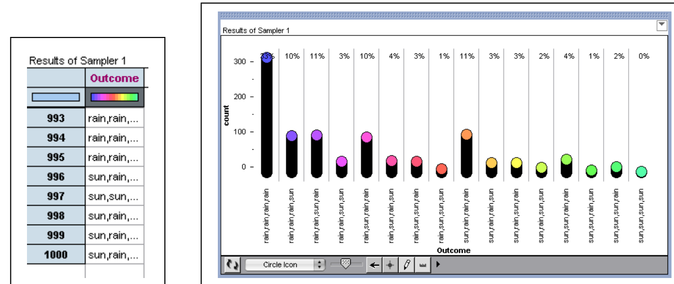
Our study involves the use of a commercial system called TinkerPlots, used world-wide to teach students in grades 4-8 about statistics and mathematics [10]. In TinkerPlots, students actively model stochastic events and construct models that generate data. TinkerPlots is flexible, allowing for data to be modeled, generated, and analyzed in many different ways using an open-ended interface. Our empirical studies focused on two different problems in which students used TinkerPlots to model and analyze stochastic data.

We will use the following example, called RAIN, drawn from a set of problems posed to students using TinkerPlots: "The probability it will rain on any given day is 75%. Use TinkerPlots to compute the probability that it will rain on the next four consecutive days." A few approaches towards modeling this problem using TinkerPlots is shown in Figure 1. The model shown in Figure 1a includes a sampler containing a "spinner" device, which contains two possible events, "rain" and "sun". The distribution mass of "rain" is three times as likely as "sun" as can be shown by the surface area of this event 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 in four consecutive days. Another possible approach towards solving this problem is presented in Figure 1b, showing a sampler with four spinner devices. Each of these devices is drawn once, and is a stochastic model of the weather on a given day. In both of these approaches, the sampler is effectively a model of the joint probability distribution over the weather for four consecutive days. There are other approaches towards modeling this scenario which we do not show here. Figure 1b shows part of the data generated by the sampler and the end-result of a process in which this data is projected onto a histogram, for the purpose of inferring the likelihood of rain.

Students interact with TinkerPlots through a series of rudimentary operations that create, modify or delete objects such as spinners, devices and plots.



(a) Two Possible Sampler Models



(b) Generating Sampler Data

Fig. 1: Solving the RAIN Problem with TinkerPlots

We will use the term *basic actions* to refer to these operations, which can often be carried out by a single menu or mouse operation. TinkerPlots interactions are recorded as a linear sequence of basic actions in order of their occurrence. Each basic action uses a unique identification tag and parameters to refer to objects that are transparent to the user. For example, the action $\mathsf{ADS}[\mathsf{s} = 11, \mathsf{d} = 2, \mathsf{dt} = \text{"spinner"}]$ (Add Device to Sampler) refers to the action of adding a device of type "spinner" and ID 3 to sampler ID 11. The following is a partial sequence of the actions used to make up the creation of the spinner device of Figure 1a. Actions are presented from top to bottom and left to right in order of occurrence, with the following abbreviated action names: AED (Add Event to Device), AS (Add Sampler), CPD (Change Probability in Device)

$\ldots, \mathsf{ADS}[\mathsf{s} = 11, \mathsf{d} = 2, \mathsf{dt} = \text{"spinner"}], \mathsf{AED}[\mathsf{s} = 11, \mathsf{d} = 2, \mathsf{ei} = 1, \mathsf{el} = \text{"rain"}], \mathsf{AS}[\mathsf{s} = 9],$
$\quad \mathsf{AED}[\mathsf{s} = 11, \mathsf{d} = 2, \mathsf{ei} = 2, \mathsf{el} = \text{"sun"}], \mathsf{CPD}[\mathsf{s} = 11, \mathsf{d} = 2, \mathsf{ss} = 1 : 3],$
$\quad \mathsf{ADS}[\mathsf{s} = 11, \mathsf{d} = 3, \mathsf{dt} = \text{"spinner"}], \ldots,$

Fig. 2: Partial Action Sequence of an Interaction.

3 Actions, Recipes and Restrictions

In this section we make a distinction between basic TinkerPlots actions and more abstract activities that characterize a user's interaction with software. We call these activities complex actions. Examples of such actions in the TinkerPlots domain include creating four spinners, solving the RAIN problem, or fitting sampler data to a plot. Complex actions can be decomposed into sub-actions [11]. A sub-action is a basic TinkerPlots action or it can be a complex action itself. A recipe for a complex action is defined as an ideal sequence of operations for fulfilling the complex action [12].

Formally, an *action* is a pair (a, P_a) where a is an identifier and P_a is a set of parameters. (we also refer to an action simply by its identifier). A *recipe* for an action a includes (1) a set of sub-actions $s_1, \ldots s_n$ and (2) a set of restrictions R_a that are Boolean functions over the sub-actions. There are two types of restrictions. An *ordering restriction* defines a precedence relation between actions. A *parametric restriction* returns value `true` if the value of an expression is satisfied. For example, the parameters of two separate actions could be required to agree (or disagree) on the value of a constant.

Figure 3 shows a possible recipe for completing the complex action $\boxed{\mathsf{CCD}}$ (Create Correct Device) that includes the basic actions ADS (Add Device to Sampler), CPD (Change Probabilities in Device) and the complex action $\boxed{\mathsf{AED}}$ (Add Event to Device). When denoting constraints for a recipe, we specify the actions and parameters that involve each restriction in the recipe above, using a wild-card symbol "*" to express "don't-care" values for the parameters that are not mentioned explicitly. The first restriction in the recipe of Figure 3 specifies

$$\boxed{\mathsf{CCD}[\mathsf{s},\mathsf{d},\mathsf{i}_1,\mathsf{i}_2]} \longrightarrow \mathsf{ADS}[\mathsf{s},\mathsf{d},\mathsf{dt}], \boxed{\mathsf{AEDs},\mathsf{d},\mathsf{i}_1,\mathsf{e}_1]}, \boxed{\mathsf{AEDs},\mathsf{d},\mathsf{i}_2,\mathsf{e}_2]}, \mathsf{CPD}[\mathsf{s},\mathsf{d},\mathsf{ss}]$$

with constraints

$$\begin{aligned} \mathsf{ADS}[*] &\prec \mathsf{AED}[*] \prec \mathsf{CPD}[*] \\ \mathsf{CCD}[\mathsf{s},\mathsf{d},*] &= \mathsf{ADS}[\mathsf{s},\mathsf{d},*] = \mathsf{AEDs},\mathsf{d},*] \\ \mathsf{CCD}[\mathsf{i}_1,*] &= \mathsf{AEDi}_1,*] \\ \mathsf{CPD}[\mathsf{ss},*] &= (3:1) \end{aligned}$$

Fig. 3: A Recipe for Creating a Device for the RAIN problem

that the addition of a device must precede the addition of events in the device as well as setting the likelihood of each event. The second and third restriction specify that actions in the recipe agree on the parameter identification tags that are bound to the TinkerPlots objects they use. The last restriction specifies that one of the events in the device should be weighted three times more likely than the other event.

We are now ready to begin to describe the processes that form the building blocks of algorithms for recognizing users' activities in TinkerPlots. An *expansion* of a complex action a is the set of all actions and restrictions such that performing those actions under the restrictions constitute completing the action. We state this formally below.

Definition 1. *Let a be a complex action with restrictions R. An* expansion *is a set of actions $s_1, \ldots, s_n$ and restrictions R^* that meet the following criteria:*

1. *The actions $s_1, \ldots, s_n$ are the constituent actions of a recipe for a.*
2. *Any restriction in R_a also exists in R^*.*
3. *For any restriction $r \in R$ involving a, and for each action s_i, there exists a restriction in R^* in which action a is substituted by s_i. (The substitution operation is formally defined as $r\{a/s_i\}$ in the normal term writing sense).*

For example, consider the complex action $\boxed{\mathsf{CCD}}$ and the restriction $\mathsf{AS}[*] \prec \boxed{\mathsf{CCD}[*]}$, implying that the process by which a device is created can only occur after a sampler was added using the AS action. An expansion of the $\boxed{\mathsf{CCD}}$ action with the recipe of Figure 3 will yield the constituent actions and restrictions of the recipe, as well as the following restrictions (we use set notation for imposing an ordering between AS and each action in the set).

$$\mathsf{AS}[*] \prec \{\mathsf{ADS}[*], \boxed{\mathsf{AED}[*]}, \mathsf{CPD}[*]\}$$

We now define a structured tree representation that contains all possible recipes for completing an action a. A *recipe tree* has two types of nodes: "AND" nodes, whose children represent actions that must be carried out in order to complete a recipe; and "OR" nodes, whose children represent a choice of action for completing recipe.

Definition 2. *A recipe tree for action a given a (possibly empty) set of restrictions R is a tree whose root is action a and labeled as an "OR" node. For each recipe of a, with sub-actions $s_1, \ldots s_n$ and restrictions R_a, there is an "AND" child node labeled with $s_1, \ldots s_n$. The children of this "AND" node are the recipe trees of each sub-action s_j given restrictions R_a.*

A partial recipe tree for the the $\boxed{\mathsf{CCD}}$ (Create Correct Device) action is presented below. We only expand the descendants associated with the recipes that appear in the first and second child nodes of the $\boxed{\mathsf{CCD}}$ action (from left to right). Figures 1a and 1b show the end-result of choosing these recipes to complete the $\boxed{\mathsf{CCD}}$ action in TinkerPlots.

An *exhaustive expansion* for a given a set of restrictions R is the result of iteratively expanding a to get a set of basic level actions and restrictions on these actions. These represent the set of necessary actions in TinkerPlots towards completing a task. Figure 5 shows the actions and restrictions comprising a possible expansion for completing the action $\boxed{\mathsf{CCD}}$

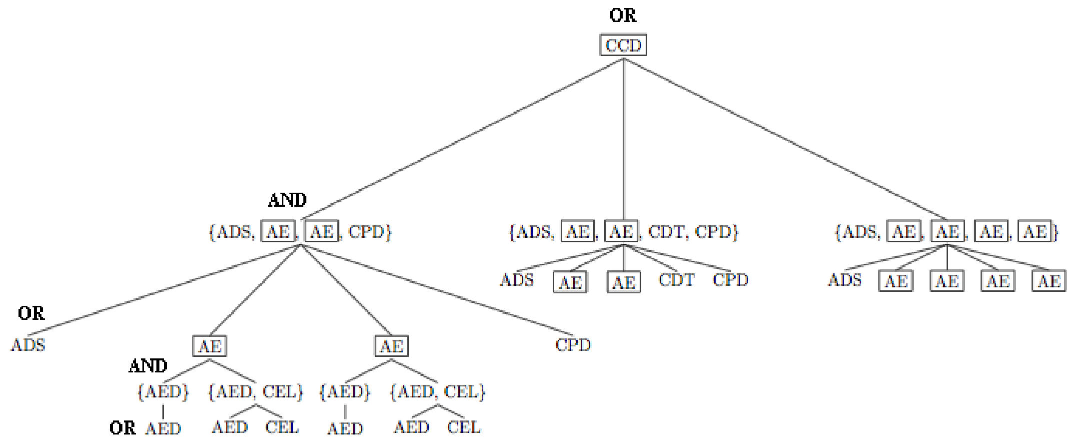


Fig. 4: A Partial Recipe Tree for the Create Correct Device (CCD) Action

$$\{\mathsf{ADS}[\mathsf{s},\mathsf{d},\mathsf{dt}],\mathsf{AED}[\mathsf{s},\mathsf{d},\mathsf{ei},\mathsf{el}],\mathsf{AED}[\mathsf{s},\mathsf{d},\mathsf{ei},\mathsf{el}],\mathsf{CPD}[\mathsf{s},\mathsf{d},\mathsf{ss}]\}$$

$$\mathsf{ADS}[*] \prec \mathsf{CPD}[*]$$
$$\mathsf{ADS}[\mathsf{dt},*] = \text{"spinner"}$$
$$\mathsf{ADS}[\mathsf{s},\mathsf{d},*] = \mathsf{AED}[\mathsf{s},\mathsf{d},*]$$

Fig. 5: An Exhaustive Expansion of the $\boxed{\mathsf{CCD}}$ Action

We say a *match* exists between a user interaction and an exhaustive expansion if each action in the expansion can be mapped to a distinct user action such that the user actions together satisfy all restrictions. This mapping should allow for a users' interaction to contain redundant actions representing users' exploration and mistakes.

Definition 3. *A match between an expansion (S, R) with actions S and restrictions R and a user interaction X is an injective mapping, ϕ:$A \to X$, such that the following holds: for all $r \in R$ and for any subset $S' = (s_1, \ldots, s_m)$ of S, if r involves S' then the restriction $r\{S'/\phi(s_1), \ldots, \phi(s_m))\}$ will satisfy r. The notation $r\{S'/\phi(s_1), \ldots, \phi(s_m))\}$ refers to the substitution of S' with $\phi(s_1), \ldots, \phi(s_m)$ in restriction r.*

For example, the following actions in the interaction history of Figure 2 match their respective action in the exhaustive expansion of Figure 5.

$$\mathsf{ADS}[\mathsf{s}=11,\mathsf{d}=2,\mathsf{dt}=\text{"spinner"}],\mathsf{AED}[\mathsf{s}=11,\mathsf{d}=2,\mathsf{ei}=1,\mathsf{el}-\text{"rain"}],$$
$$\mathsf{AED}[\mathsf{s}=11,\mathsf{d}=2,\mathsf{ei}=2,\mathsf{el}=\text{"sun"}],\mathsf{CPD}[\mathsf{s}=11,\mathsf{d}=2,\mathsf{ss}=1:3]$$

This match represents a complete account of the salient actions used to complete the action $\boxed{\mathsf{CCD}}$. The actions $\mathsf{NS[s = 9]}$ and $\mathsf{ADS[s = 11, d = 3, dt = "spinner"]}$ in the interaction history were deemed redundant by the match.

Lastly, planning is the process by which users use recipes to compose basic and complex actions towards completing tasks in TinkerPlots. A plan for a is a set of basic and complex actions, such that each complex action is expanded into sub-actions that fulfill a recipe for the complex action. The basic actions in a plan are an exhaustive expansion of a. Figure 6 shows a possible plan for solving the $\boxed{\mathsf{CCD}}$ action.

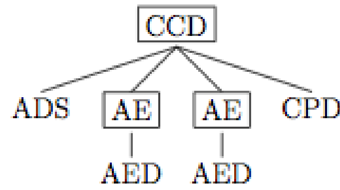


Fig. 6: A Plan for fulfilling the $\boxed{\mathsf{CCD}}$ action

4 From Recipes to Constraint Satisfaction Problems

We now show how to combine a recipe tree and a user's interaction sequence to create a constraint satisfaction problem. The solution to a CSP provides an explanation of a user's activities by providing a match (in the sense of Definition 3) between an exhaustive expansion and an interaction history. A Constraint Satisfaction Problem (CSP) is a triple (X, Dom, C). $X = \{x_1, ..., x_n\}$ is a finite set of variables with respective domains $Dom = \{d_1, ..., d_n\}$, in which each domain contains the possible values for its corresponding variable, $D_i = \{v_1, ..., v_k\}$, and a set of constraints $\mathcal{C} = \{c_1, ..., c_m\}$ that limit the values that can be assigned to any set of variables. Each constraint C_i is a tuple containing a set of variables, and a set of values taken from the domain of each variable. Essentially, C_i is a Boolean function over V.

We can combine a given interaction history and an exhaustive expansion to a CSP as follows: Let $S = (s_1, \ldots, s_n)$ and R be a set of actions and restrictions in the expansion. Each basic action in the expansion becomes a unique variable in the CSP. If the same action appears more than once in the expansion, it is subscripted accordingly. For example, the expansion of Figure 5 includes two AED actions, which will yield two variables $\mathsf{AED_1}$ and $\mathsf{AED_2}$ in the CSP. The domain of each variable is derived from the user's interaction and is denoted $\{(s, i_1), \ldots, (s, i_n)\}$, where s is some action in the expansion S, and $i_1, \ldots, i_n$ is

a set of indices, representing the temporal order of the various occurrences of s in the interaction history. For each occurrence of action s at index i in the interaction history, there is an element (s, i) in the domain of s in the CSP.

For example, let us take the expansion of Figure 5 for the CCD action. The domains of the actions in the exhaustive expansion for CCD action are as follows:

$$\begin{aligned} Dom(\mathsf{ADS}) &= \{(\mathsf{ADS}, 0), (\mathsf{ADS}, 5)\} \\ Dom(\mathsf{AED_1}) &= \{(\mathsf{AED}, 1), (\mathsf{AED}, 3)\} \\ Dom(\mathsf{AED_2}) &= \{(\mathsf{AED}, 1), (\mathsf{AED}, 3)\} \\ Dom(\mathsf{NS}) &= \{(\mathsf{NS}, 2)\} \\ Dom(\mathsf{CPD}) &= \{(\mathsf{CPD}, 4)\} \end{aligned}$$

4.1 Adding Constraints

Having defined the variables and their associated domains in the CSP, we now show how to map restrictions R in the expansion S to constraints in the CSP. For each restriction $r(s_1, \ldots, s_m)$ in R involving actions $(s_1 \ldots, s_m) \in S$, we add a constraint over the corresponding variables in the CSP in the form of a tuple of variables and values. For any subset of actions in an expansion S we add constraint $\{(s_1, \ldots, s_m), ((s_1, i_1), \ldots, (s_m, i_m))\}$ when there is a restriction $r(s_1, \ldots, s_m)$ in R, so that the indices $i_1, \ldots, i_m$ are decreasing. For example, given the constraint $\mathsf{ADS} \prec \mathsf{CPD}$ in the exhaustive expansion of Figure 5, we will add the following constraint to the CSP:

$$\{(\mathsf{ADS}, \mathsf{CPD}), ((\mathsf{ADS}, 5), (\mathsf{CPD}, 4))\}$$

Intuitively, this constraint will represent a prohibitive event in which action ADS precedes CPD, which violates the restriction. Parametric restrictions are added in the same way. For example, we get that

$$\{(\mathsf{ADS}, \mathsf{AED}), ((\mathsf{ADS}[\mathsf{d} = 3], 5), (\mathsf{AED_1}[\mathsf{d} = 2], 1))\}$$

Intuitively, this constraint will not allow a match to occur in which actions ADS and $\mathsf{AED_1}$ disagree on the value of the parameter representing the device ID. Similar constraints are added for the other parametric restrictions in the example. Lastly, for variables corresponding to the same action ID, such as AED_1 and AED_2, we add a redundancy constraint ensuring that these variables are assigned distinct values, because they represent separate occurrences of the same action.

5 Algorithms for Recognizing Interaction Histories

A solution for a CSP provides a *match* between an exhaustive expansion and an interaction history. The path traversed on the recipe tree to generate the exhaustive expansion is effectively the user's plan towards completing a task

in TinkerPlots. We provide two algorithms that use CSPs that output a plan for action a in an interaction history H given a data base of recipes RP. Both algorithms rely on the recipe tree of a to inform their recognition process. The first algorithm, shown in Figure 7, is a brute force approach, which traverses the recipe tree and seeks to solve a CSP for every possible exhaustive expansion of a. The second algorithm, shown in Figure 8 is more sophisticated. It identifies the

Use the recipes in RP to construct the recipe tree for action a.
For each exhaustive expansion E in the tree,
  Convert E and the interaction history H to a CSP C.
  If there is a solution to C, output the plan for E
If no solution has been found, then fail.

Fig. 7: Brute Force Recognition Algorithm

descendants of a whose actions cannot be explained by the interaction history and refrains from using these actions in potential solutions.

Use the recipes in RP to construct the recipe tree for action a.
Traverse the tree bottom-up. For each "OR" node corresponding to a complex action s:
  If s has not been visited before, then
    Use the brute-force algorithm to recognize action s given recipes RP.
    If the brute-force algorithm finds a match, cache a as successful.
    If a match is not found we prune the "AND" node parent of s from the tree.
  If s has been visited before, and is cached as failed, then prune the "AND" node parent of s from the tree.
Call the brute-force algorithm to recognize action a given recipes RP.

Fig. 8: Bottom-Up Recognition Algorithm

Both approaches are complete, in the sense that if the user was able to solve a TinkerPlots problem using actions in the recipe data-base, the algorithms are able to recognize the solution. In particular, if the user solves a TinkerPlots problem several times within the same session, the algorithms would recognize this. However, we hypothesized that the Bottom-Up algorithm would be more efficient than the Brute Force algorithm, because it removes all nodes for which there is no solution from the recipe tree prior to calling the brute-force algorithm to recognize the root.

6 Empirical Methodology

We evaluated our algorithms on interaction histories obtained from twelve adult subjects with a broad array of educational backgrounds, varying from some high

school to some post graduate education. Each subject was given an identical 30-minute tutorial to TinkerPlots and was then asked to complete two problems in succession. One of these was the RAIN problem; the other, called ROSA, asked the user to build a distribution over the values of four distinct objects and to find the likelihood of a distinct combination. We consider a plan that is found by a recognition algorithm to be "correct" if it agreed with the opinion of a domain expert that had access to the same inputs as did the algorithms. For this to occur, the algorithm and domain expert had to agree not only on whether the problem was solved by the user, but also on those actions in the user's interaction history that played a salient part in the solution.

We compared between the performance of three recognition algorithms: the Brute Force and Bottom-Up techniques described above, as well as the recognition algorithm proposed by Gal et al. [5], denoted as the "Greedy" algorithm. Four of the 24 instances that were collected were discarded because of a bug in the data-registration facility, which failed to register some of the users' actions. Therefore we present results on the remaining 20 instances.

Both the Brute Force and Bottom-Up algorithms generated the same output, so we present evaluation for both algorithms at once. We used an open-source test-bed by Gustavo Niemeyer for solving the CSPs, available at the following url: http://labix.org/python-constraint.

Overall, both CSP algorithms inferred correct solutions for all 20 interaction histories, whereas the greedy algorithm inferred correct solutions for only 13 out of the 20. The following table shows the number of times that each algorithm identified a plan, or failed to find a plan, for the interaction histories relating to each problem. We also show the opinions of the domain expert, which we will consider to be the "gold standard".

	Problem	**Recognition Stats.**	
		Yes	No
Domain Expert	ROSA	7	3
	RAIN	7	3
Greedy algorithm	ROSA	3	7
	RAIN	4	6
CSP algorithms	ROSA	7	3
	RAIN	7	3

As the table shows the greedy algorithm failed to find a solution in 3 instances of RAIN and four instances of the ROSA problem. These are considered "false-negatives", because the domain expert determined that solutions existed for those problems. In contrast, the CSP algorithms agreed with the opinions of the domain expert on all instances.

User logs range in size from 14 to 80 actions, and plans range in size from 16 to 34 actions. The average user logs for the ROSA and RAIN were 33 and 23.25 actions and the average plans found were 18.14 and 19.57 actions, respectively.

To compare the Brute Force and Bottom-Up algorithms, we collect 3 statistics: the number of CSPs modeled for each log, the average number of variables

in the CSPs built for each log, and the run-time expended for each log. The Brute Force method creates a greater number of CSPs and requires a longer run-time than does the Bottom-Up method in 64% and 69% of cases, respectively. In all cases, Bottom-Up creates fewer variables per constraint network. The summary statistics are found below.

Problem	**Brute Force**			**Bottom-Up**		
	CSPs	Variables	Run-time (s)	CSPs	Variables	Run-time (s)
ROSA	19	18	0.12	14	4	0.13
RAIN	9300	29	131.54	141	9	21.88

The bottleneck of both algorithms is the use of the recipe tree. The complexity of the recipe tree's construction is exponential in both the number of recipes and the maximum number of constituents for a recipe in the recipe database. On average, Bottom-Up creates fewer CSPs and uses fewer variables per constraint network. Though Brute Force has a slightly shorter run-time for the ROSA problem, Bottom-Up has a considerably shorter run-time for the RAIN problem. Scalability issues appear to be linked to the complexity of recipes rather than the length of user logs. We measure recipe complexity by the number of distinct plans for a problem, and we find that increased recipe complexity for a problem corresponds to increased average run-time. In contrast, the longest user log for each problem experienced among the shortest run-time for that problem, and the average log size for a problem did not correspond to increased average run-time.

7 Conclusion and Future Work

This work provided a comprehensive study of the use of constraint satisfaction techniques towards automatic recognition of users' activities with computer software. We showed that under certain conditions, namely the existence of a set of ideal solutions (recipes), this approach provides a robust, tractable and complete solution to this problem. We evaluated our techniques in "real-world" conditions, showing their ability to recognize users' plans, when interacting with commercial software for mathematics education. These algorithms were able to outperform a related approach from the literature for inferring user-software interaction. In future work, we wish to explore the use of these techniques in applications of rich interaction structure between user and system, such as programming languages. We also wish to use the solutions obtained by our algorithms to construct a collaborative pedagogical agent that generates support for guiding the student in the learning process. Lastly, we are exploring different representation and visualization approaches for presenting information relating to students' plans to teachers.

Acknowledgements

Thanks to Andee Rubin for her great help in running the user study, and for providing comments and suggestions to sections in the paper. Thanks to Elif Yamangil for assisting with the development of the greedy algorithm. Thanks to Cliff Konold for helpful discussions. Thanks to Craig Miller for developing the logging capability for TinkerPlots.

References

1. Pollack, M.: Intelligent technology for an aging population: The use of AI to assist elders with cognitive impairment. AI Magazine **26**(9) (2006)
2. Hammerman, J.K., Rubin, A.: Strategies for managing statistical complexity with new software tools. Statistics Education Research Journal **3**(2) (2004) 17–41
3. Lesh, N., Rich, C., Sidner, C.: Using Plan Recognition in Human-Computer Collaboration. (1999) 23–32
4. Anderson, J.R., Corbett, A.T., Koedinger, K., Pelletier, R.: Cognitive tutors: Lessons learned. The Journal of Learning Sciences **4(2)** (1995) 167–207
5. Gal, Y., Yamangil, E., Rubin, A., Shieber, S., Grosz, B.J.: Towards collaborative intelligent tutors: Automated recognition of users' strategies. In: Proceedings of Ninth International Conference on Intelligent Tutoring Systems (ITS), Montreal, Quebec
6. Bauer, M.: Acquisition of user preferences for plan recognition. In: Proceedings of the Fifth International Conference on User Modeling. (1996) 105–112
7. Lesh, N.: Adaptive Goal Recognition. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence, Morgan Kaufmann (1997) 1208–1214
8. Conati, C., Gertner, A., VanLehn, K.: Using bayesian networks to manage uncertainty in student modeling. Journal of User Modeling and User-Adapted Interaction **12**(4) (2002) 371–417
9. Corebette, A., McLaughlin, M., Scarpinatto, K.: Modeling student knowledge: Cognitive tutors in high school and college. User Modeling and User-Adapted Interaction **10** (2000) 81—108
10. C. Konold, C.M.: TinkerPlots Dynamic Data Exploration 1.0. Key Curriculum Press. (2004)
11. Grosz, B., Kraus, S.: The evolution of sharedplans. Foundations and Theories of Rational Agency (1999) 227–262
12. Pollack, M.: Plans as complex mental attitudes. MIT Press (1990)