Plan Recognition for Exploratory Domains Using Interleaved Temporal Search

Oriel Uzan and **Reuth Dekel** and **Ya'akov (Kobi) Gal**
Dept. of Information Systems Engineering,
Ben-Gurion University,
Beer-Sheva, Israel

Abstract

In exploratory domains, agents' actions map onto logs of behavior that include switching between activities, extraneous actions, and mistakes. These aspects create a challenging plan recognition problem. This paper presents a new algorithm for inferring students' activities in exploratory domains that is evaluated empirically using a new type of flexible and open-ended educational software for science education. Such software has been shown to provide a rich educational environment for students, but challenge teachers to keep track of students' progress and to assess their performance. The algorithm decomposes students complete interaction histories to create hierarchies of interdependent tasks that describe their activities using the software. It matches students' actions to a predefined grammar in a way that reflects that students solve problems in a modular fashion but may still interleave between their activities. The algorithm was empirically evaluated on peoples interaction with two separate software systems for simulating a chemistry laboratory and for statistics education. It was separately compared to the state-of-the-art recognition algorithms for each of the software. The results show that the algorithm was able to correctly infer students' activities significantly more often than the state-of-the-art, and was able to generalize to both of the software systems with no intervention.

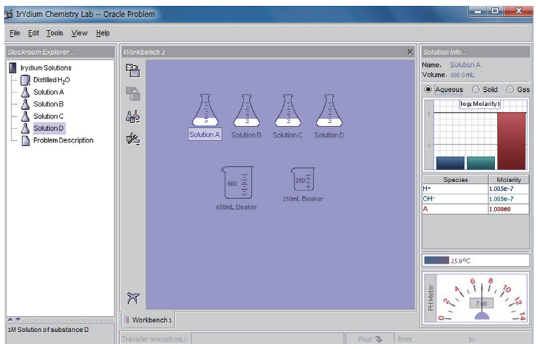
Introduction

This paper describes a challenging plan recognition problem that arises in environments in which agents engage widely in exploratory behavior, and presents a new algorithm for effective plan recognition in such settings that outperforms the state-of-the-art. In exploratory domains, agents' actions map onto logs of behavior that include switching between activities, extraneous actions, and mistakes. Flexible educational software, such as the application considered in this paper for chemistry and statistics education, is a paradigmatic example of such domains, but many other settings exhibit similar characteristics such as interactive drawing tools (Ryall, Marks, and Shieber 1997), Integrated Development Environments (IDEs) and collaborative writing assistants (Babaian, Grosz, and Shieber 2002).

Our empirical analysis uses a new type of educational software systems called Exploratory Learning Environments (ELE) in which students build scientific models and examine properties of the models by running them and analyzing the results(Amershi and Conati 2006; Cocea, Gutierrez-Santos, and Magoulas 2008). Such software is open-ended and flexible and is generally used in classes too large for teachers to monitor all students and provide assistance when needed, and are becoming increasingly prevalent in developing countries where access to teachers and other educational resources is limited (Pawar, Pal, and Toyama 2007). Thus, there is a need to develop tools of support for teachers' understanding of students' activities. Such tools can provide support for teachers and education researchers in analyzing and assessing students' use of the software. However, there are several aspects to students' interactions with ELEs that make it challenging to recognize their activities. Students can engage in exploratory activities involving trial-and-error, they can repeat activities indefinitely, and they can interleave between activities.

This paper presents a plan recognition algorithm that meets these challenges. It works offline, and decomposes students' complete interaction history with the software into hierarchies of interdependent tasks that best describe their work with the software. It matches students' actions to a grammar in a way that reflects the aspects of students' work in ELEs described above. The algorithm was evaluated in an extensive empirical study that involved seven different types of problems and 68 instances of students' interactions in two different ELEs. It was compared to two state-of-the-art algorithms for recognizing students' plans in the ELEs. It was able to correctly recognize significantly more plans than did both of the state-of-the-art algorithms. It executed in reasonable time on real-world logs of students' sessions, despite the exponential worst-case complexity of the algorithm.

Although plan recognition is a cornerstone problem in AI, traditional algorithms do not capture the flexible and open-ended nature of ELEs (Gal et al. 2012). Recent works have developed algorithms for inferring students' plans tailored to specific ELEs. Gal et al. (2008; 2012) proposed two algorithms for an ELE for statistics education, one which used heuristics to match students' actions to the grammar, and another which modeled the plan recognition task as a CSP. This approach assumes a non-recursive grammar and cannot recognize students' plans in cases where students engage in indefinite repetition, as in performing repeated pours of mea-



Name: Solution A
Volume: 100.0 mL

Species	Molarity
H+	1.005e-7
OH-	1.005e-7
A	1.000e0

1M Solution of substance D

Figure 1: Snapshot of VirtualLabs

surements in chemistry or physics. Amir and Gal (2011) allow for recursive grammars in their plan recognition algorithm for an ELE for chemistry education, but make greedy choices about how to match students' actions to the grammar. In contrast, our algorithm makes choices that are inferred by students' sequential and interleaving interaction styles. The results of this paper show that our algorithm was able to outperform this approach. Also, none of these past approaches have been shown to work for more than a single ELE, whereas our algorithm is shown to generalize to both the statistics and chemistry ELEs. Lastly, we mention works that use recognition techniques to model students' activities in Intelligent Tutoring Systems (VanLehn et al. 2005; Conati, Gertner, and VanLehn 2002; Vee, Meyer, and Mannock 2006). These are not applicable to ELEs because students' exploration is significantly more constrained.

Representing Students' Activities

Throughout the paper we will use an ELE called VirtualLabs to demonstrate our approach. VirtualLabs allows students to design and carry out their own experiments for investigating chemical processes (Yaron et al. 2010) by simulating the conditions and effects that characterize scientific inquiry in the physical laboratory. We use the following problem called "Oracle" as a running example:

> Given four substances A, B, C, and D that react in a way that is unknown, design and perform virtual lab experiments to determine the correct reaction between these substances.

The flexibility of VirtualLabs affords two classes of solution strategies to this problem (and many variations within each). The first strategy mixes all four solutions together, and infer the reactants by inspecting the resulting solution. The second strategy mixes pairs of solutions until a reaction is obtained. A snapshot of a student's interaction with VirtualLabs when solving the Oracle problem is shown in Figure 1.

We use the term *basic actions* to define rudimentary operations that cannot be decomposed. These serve as the input to our plan recognition algorithm. For example, the basic "Mix Solution" action ($\mathsf{MS}_1[s=1, d=3]$) describes a pour from flask ID 1 to flask ID 3. The output of a student's interaction with an ELE (and the input to the plan recognition algorithm described in the next section) is a sequence of basic level actions representing students' activities, also referred to as a "log". *Complex actions* describe higher-level, more abstract activities that can be decomposed into sub-actions, which can be basic actions or complex actions themselves. For example, the complex action $\underline{\mathsf{MSD}}[s = 6+8, d = 2]$ represents separate pours from flask ID 6 and 8 to flask ID 2.

$$
\begin{aligned}
&(a)\ \underline{\mathsf{MSD}}[s_1+s_2, d] \rightarrow \underline{\mathsf{MSD}}[s_1, d], \underline{\mathsf{MSD}}[s_2, d]\\
&(b)\ \underline{\mathsf{MIF}}[s_1, d_2] \rightarrow \underline{\mathsf{MSD}}[s_1, d_1], \underline{\mathsf{MSD}}[d_1, d_2]\\
&(c)\ \underline{\mathsf{MSD}}[s, d] \rightarrow \underline{\mathsf{MIF}}[s, d]\\
&(d)\ \underline{\mathsf{MSD}}[s, d] \rightarrow \mathsf{MS}[s, d]
\end{aligned}
$$

Figure 2: Recipes for VirtualLabs

A *recipe* for a complex action specifies the sequence of actions required for fulfilling the complex action. Figure 2 presents a set of basic recipes for VirtualLabs. In our notation, complex actions are underlined, while basic actions are not. Actions are associated with parameters that bind to recipe parameters. Recipe (a) in the figure, called Mix to Same Destination ($\underline{\mathsf{MSD}}$), represents the activity of pouring from two source flasks (s_1 and s_2) to the same destination flask (d). Recipe (b), called Mix via Intermediate Flask ($\underline{\mathsf{MIF}}$), represents the activity of pouring from one source flask (s_1) to a destination flask (d_2) via an intermediate flask (d_1). Recipes can be recursive, capturing activities that students can repeat indefinitely. For example, the constituent actions of the complex action $\underline{\mathsf{MSD}}$ in recipe (a) decompose into two separate $\underline{\mathsf{MSD}}$ actions. In turn each of these actions can itself represent a Mix to Same-Destination action, an intermediate-flask pour (by applying recipe (c)) or a basic action mix which is the base-case recipe for the recursion (recipe (d)).[1] More generally, the four basic recipes in the figure can be permuted to create new recipes, by replacing $\underline{\mathsf{MSD}}$ on the right side of the first two recipes with $\underline{\mathsf{MIF}}$ or MS. An example of a derivation is the following recipe for creating an intermediate flask out of a complex Mix to Same Destination action and basic Mix Solution action.

$$\underline{\mathsf{MIF}}[s_1, d_2] \rightarrow \underline{\mathsf{MSD}}[s_1, d_1], \mathsf{MS}[d_1, d_2] \tag{1}$$

A *plan* is a set of trees of basic and complex actions that describe a student's interaction with an ELE. Figure 3 shows a plan describing part of a student's interaction when solving the Oracle problem. The leaves of the trees are the actions from the student's log, and are labeled by their order of appearance in the log. A node representing a complex action is labeled by a pair of indices indicating its earliest and latest constituent actions in the plan. For example, the node labeled with the complex action $\underline{\mathsf{MSD}}[s = 1+5, d = 3]$ includes the activities for pouring two solutions from flask ID 1 and ID 5 to flask ID 3. The pour from flask ID 5 to 3 is

[1]Recipe parameters also specify the type and volume of the chemicals in the mix, as well as temporal constraints between constituents, which we omit for brevity.

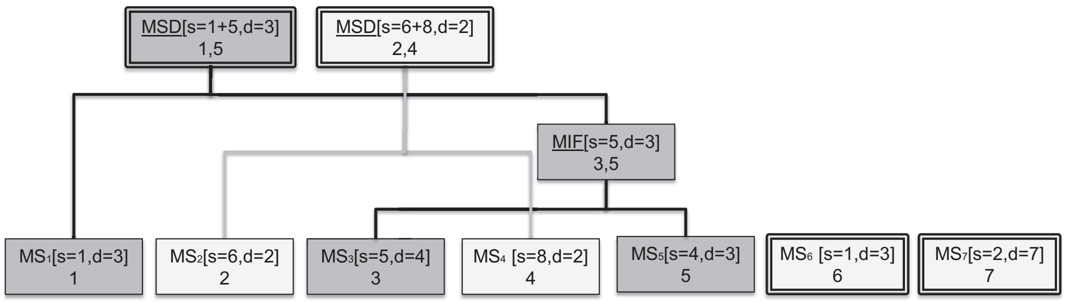


Figure 3: A partial plan for a student's log

an intermediate flask pour ($\underline{\mathsf{MIF}}[s = 5, d = 3]$) from flask ID 5 to ID 3 via flask ID 4.

A set of nodes N in a plan *fulfils* a recipe R_C if there exists a one-to-one matching between the constituent actions in R_C and their parameters to nodes in N. The matching can occur between non-contiguous nodes in the plan, capturing interleaving activities in a student's interaction. For example, the nodes $\underline{\mathsf{MSD}}[s = 6 + 8, d = 2]$ and $\mathsf{MS}_7[s = 2, d = 7]$ fulfill the Mixing via an Intermediate Flask recipe shown in Equation 1.

The PRISM Algorithm

Our algorithm, called Plan Recognition via Interleaved Sequential Matching (PRISM), provides a tradeoff between the following two complementary aspects of students' interactions with ELEs. First, students generally solve problems in a sequential fashion, by which we mean that actions that are (temporally) closer to each other are more likely to relate to the same sub-goal. For example, in Figure 3 the basic actions $\{\mathsf{MS}_1, \mathsf{MS}_5\}$ are more likely to fulfill the recipe $\underline{\mathsf{MSD}}[s_1 + s_2, d] \longrightarrow \mathsf{MS}[s_1, d], \mathsf{MS}[s_2, d]$ than the basic actions $\{\mathsf{MS}_1, \mathsf{MS}_6\}$ because they are closer together in proximity. Second, students may interleave between activities relating to different sub-goals. For example, in Figure 3, the node $\mathsf{MS}_4[s = 8, d = 2]$ (which is a constituent action of the complex action $\underline{\mathsf{MSD}}[s = 6 + 8, d = 2]$) occurs in between the nodes representing the constituent actions of the complex action $\underline{\mathsf{MIF}}[s = 5, d = 3]$.

Before presenting the algorithm we first make the following definitions. Let P denote the partial plan at an intermediate step of the algorithm. The *frontier* of P is all nodes in the plan that do not have parents. For example, the frontier of the plan shown in Figure 3 is the set of all the nodes that are outlined with a double line. Let R_C denote a recipe for a complex action C. We say that a set of nodes N is *frontier compatible* with a recipe R_C if N is a subset of the frontier of P and N fulfills R_C in P. Intuitively, the nodes in a frontier compatible set of R_C may be used to fulfill the recipe and add C to the plan. For example, In Figure 3, the set of nodes $F = \{\underline{\mathsf{MSD}}[s = 6+8, d = 2], \mathsf{MS}_7[s = 2, d = 7]\}$ is frontier compatible with the Mix Intermediate Flask recipe of Equation 1.

A *match* for recipe R_C in P is a triple $M_C = (C, N, \langle i, j\rangle)$ such that R_C is a recipe for completing action C, N is a set of nodes which is frontier compatible with R_C in P and $\langle i, j\rangle$ are delimiters specifying the indices corresponding to the earliest and latest actions in N. For example, the triple $(\underline{\mathsf{MIF}}[s = 6 + 8, d = 7], F, \langle 2, 7\rangle)$ is a match for the recipe of Equation 1. For brevity, we omit the frontier compatible set when referring to matches and write $M_C = (\underline{\mathsf{MIF}}[s = 6 + 8, d = 2], \langle 2, 7\rangle)$.

We define a function $\textsc{FindMatch}(R_C, P, D)$ that returns the set of all matches for R_C in P, where D is in the frontier compatible set of R_C. For example, consider the call $\textsc{FindMatch}(R_C, P, \underline{\mathsf{MSD}}[s = 6 + 8, d = 2])$, where R_C is the intermediate flask recipe of Equation 1, and P is the plan from Figure 3. This call will result in a set containing the single match that was presented earlier: $(\underline{\mathsf{MIF}}[s = 6 + 8, d = 7], F, \langle 2, 7\rangle)$.

The main functions comprising the algorithm are shown in Figure 4. The algorithm uses a global queue for storing potential matches for updating the student's plan. The queue is sorted lexicographically by the first and second indices in the delimiters of the matches. This reflects the sequential aspect of students' activities in ELEs.

The core of the algorithm is the $\textsc{ConsiderMatch}(M_C)$ function. This function begins by popping from the queue the first match with indices between $\langle i, j\rangle$ (line 3). The function then recursively updates the plan with all of the matches in P whose delimiters lie between i and j (line 5).

After all inner matches are exhausted, the algorithm checks whether M_C itself can be added to the plan (line 8), which is possible if the set of nodes N is still frontier compatible with R_C. If M_C can be added to the plan, the function $\textsc{AddMatchToPlan}(M_C)$ is called, which adds (1) a node to the plan with label C and delimiters i and j, and (2) edges from C to all of the nodes in N as they appear in the plan. At this point, all of the matches in the queue involving nodes represented in N are obsolete because C was added to the plan, so they are no longer in the frontier. Therefore we remove them from the queue (line 18). The final step is a call to the function $\textsc{ExtendMatch}(M_C)$ to update the queue with new matches in which C is a constituent.

Lastly, we show how to make the first call to the algo-

```
1:  function ConsiderMatch($M_C$)        ▷ $M_C$: a match
2:      $\langle i, j\rangle \leftarrow$ Limits of the match $M_C$
3:      $M'_C \leftarrow$PopFromGlobalQueue(i,j)
4:      while $M'_C$ not null do
5:          ConsiderMatch($M'_C$)
6:          $M'_C \leftarrow$ PopFromGlobalQueue(i,j)
7:      end while
8:      if $M_C$ can be added to $P$ then
9:          AddMatchToPlan($M_C$)
10:         ExtendMatch($M_C$)
11:     end if
12: end function

13: function AddMatchToPlan($M_C$)
14:     add node $(C, i)$ to $P$   ▷ $i$ is the index of the earliest
        frontier compatible constituent of the match $M_C$
15:     for all C $\in$ childs($M_C$) do
16:         for all $M'_C \in$ GlobalQueue do
17:             if C $\in$ childrenOf($M'_C$) then in $P$
18:                 remove $M'_C$ from GlobalQueue
19:             end if
20:         end for
21:     end for
22: end function

23: function ExtendMatch($M_C$)
24:     for all $R_Q \in Recipes$ do
25:         M $\leftarrow$ FindMatch($R_Q, P, C$)
26:         for all $m \in M$ do
27:             add m to GlobalQueue
28:         end for
29:     end for
30: end function
```

Figure 4: Main functions of the PRISM algorithm

rithm. We initialize the plan by creating a node for each of the basic actions in the log; the global queue is initialized by pushing all of the matches for each recipe in the initial plan; an initial call CONSIDERMATCH($*, \langle 1, N\rangle$) is made that includes a fictitious match *, where N is the length of the log.

To demonstrate the algorithm, suppose that the plan is initialized to include only the leaves shown in Figure 3, corresponding to the student's log. The initial queue will contain the following matches, sorted by their delimiters from left-to-right as described above:

$(\underline{\mathsf{MSD}}[s = 1+4, d = 3], \langle 1, 5\rangle), (\underline{\mathsf{MSD}}[s = 1+1, d = 3], \langle 1, 6\rangle), (\underline{\mathsf{MSD}}[s = 6+8, d = 2], \langle 2, 4\rangle), (\underline{\mathsf{MIF}}[s = 6, d = 7], \langle 2, 7\rangle), (\underline{\mathsf{MIF}}[s = 5, d = 3], \langle 3, 5\rangle), (\underline{\mathsf{MIF}}[s = 8, d = 7], \langle 4, 7\rangle), (\underline{\mathsf{MSD}}[s = 1+4, d = 3], \langle 5, 6\rangle)$

The first match to be popped out from the queue in CONSIDERMATCH($*, \langle 1, 7\rangle$) is $(\underline{\mathsf{MSD}}[s = 1+4, d = 3], \langle 1, 5\rangle)$. The function performs a recursive call (line 5) to update the plan with matches involving actions occurring within the delimiters of 1 and 5. The next match to be popped off the queue is $(\underline{\mathsf{MSD}}[s = 6+8, d = 2], \langle 2, 4\rangle)$. The frontier compatible set of this match is $\{\mathsf{MS}_2[s = 6, d = 2], \mathsf{MS}_4[s = 8, d = 2]\}$. At this point, the queue subset is empty because there are no matches between the delimiters $\langle 2, 4\rangle$, so the function skips to line 8. Because the nodes $(\mathsf{MS}_2, \mathsf{MS}_4)$ in the frontier compatible set of the match have no parents, we can add the $\underline{\mathsf{MSD}}[s = 6+8, d = 2]$ complex action to the plan, by calling ADDMATCHTOPLAN($\underline{\mathsf{MSD}}[s = 6+8, d = 2], \langle 2, 4\rangle$)). This function also removes the following matches which involve MS_2 or MS_4 in their frontier compatible set from the queue (line 18):

$(\underline{\mathsf{MIF}}[s = 6, d = 7], \langle 2, 7\rangle), (\underline{\mathsf{MIF}}[s = 8, d = 7], \langle 4, 7\rangle), (\underline{\mathsf{MSD}}[s = 6+8, d = 2], \langle 2, 4\rangle)$

Finally, in line 10, the function EXTENDMATCH($\underline{\mathsf{MSD}}[s = 6+8, d = 2], \langle 2, 4\rangle$) is called to find all matches that involve the action $\underline{\mathsf{MSD}}[s = 6+8, d = 2]$ in their frontier compatible set and add them to the queue (line 27). There is only one such match: $(\underline{\mathsf{MSD}}[s = 6+8, d = 7], \langle 2, 7\rangle)$.

The complexity of the algorithm is $O(R \cdot \frac{L \cdot N!}{(N-L)!})$, where R be the number of recipes, N be the size of the log, and L be the maximum number of constituent actions in any recipe. In practice, we can assume that L is significantly smaller than N (because the number of constituent actions in a recipe will generally be well smaller than the log size). We get that the algorithm complexity is polynomial in the size of the log and the recipe database.

Let R be the size of the recipe database and N be the size of the log. Let L be the maximum number of constituent actions in any recipe in R. The bottleneck of the PRET algorithm is the FINDMATCH(R_P, P, C) function, so we analyze it first. The number of actions in the frontier is bounded by the log size, N. The number of constituent actions of any recipe is bounded by L. The number of frontier compatable set of R_P in which C is a constituent, is $\binom{N-1}{L-1}$. This is because we already have 1 certain node, C, so we need to choose $L-1$ constituents for R_P out of the rest $N-1$ nodes in the frontier. In the worst case, every permutation of every frontier compatible set fulfills the recipe. Thus the number of matches considered is bounded by $\frac{(N-1)!}{(N-L)! \cdot (L-1)!} \cdot L! = \frac{L \cdot (N-1)!}{(N-L)!}$.

Next, for each calll for UPDATEMATCHINQUEUE, FINDMATCH is called R times, so in total, there will be $R \cdot \frac{L \cdot (N-1)!}{(N-L)!}$ pushes of matches to the queue for each call for UPDATEMATCHINQUEUE. How many times is UPDATEMATCHINQUEUE called? In the worst case, each of the actions in the log will be matched by a (binary) recipe for a complex action, and added to the plan. In turn, each of these complex actions will also be matched and added to the plan and so on. In this case, the resulting plan will be a full tree and there will be $N-1$ calls to the ADDMATCHTOPLAN function. As shown in lines 9 and 10 of the UPDATEPLANBETWEEN, the function UPDATEMATCHINQUEUE is called the same number of times as ADDMATCHTOPLAN. Therefore UPDATEMATCHINQUEUE is called exactly $N-1$ times. Assuming that the complexity of ADDMATCHTOPLAN is O(1), we get that the resulting complexity of the algorithm is $O((N-1)\cdot R\cdot \frac{L\cdot(N-1)!}{(N-L)!}) \leq O(R\cdot \frac{L\cdot N!}{(N-L)!})$. In practice, the number of constituent actions in a recipe will be considerably smaller than the log. Therefore we can assume that L is significantly smaller than N. If we treat L as a constant, we get that the algorithm complexity is polynomial in the size of the log and the recipe database. Note that this is more expensive than the algorithm by Amir and Gal (2011) which was only polynomial in the recipes, and is similar to the complexity for the algorithm by Gal et al. (2012).

Empirical Methodology

We evaluated the algorithm on real data consisting of students' interactions. To demonstrate the scalability of the PRISM algorithm we evaluated it on two different ELEs: the VirtualLabs system as well as an ELE for teaching statistics and probability called TinkerPlots (Konold and Miller 2004) used worldwide in elementary school and colleges. In TinkerPlots, students build models of stochastic events, run the models to generate data, and analyze the results. It is an extremely flexible application, allowing for data to be modeled, generated, and analyzed in many ways using an open-ended interface.

For VirtualLabs, we used four problems intended to teach different types of experimental and analytical techniques in chemistry, taken from the curriculum of introductory chemistry courses using VirtualLabs in the U.S. One of these was the oracle problem that was described earlier. Another, called "Coffee", required students to add the right amount of milk to cool a cup of coffee down to a desired temperature. The third problem, called "Unknown Acid" required students to determine the concentration level and Ka level of an unknown solution. The fourth problem, called "Dilution", required students to create a solution of a base compound with a specific desired volume and concentration. For TinkerPlots, we used two problems for teaching probability to students in grades 8 through 12. The first problem, called "ROSA", required students to build a model that samples the letters A, O, R, and S and to compute the probability of generating the name ROSA using the model. The second problem, called "RAIN", required students to build a model for the weather on a given day, and compute the probability that it will rain on each of the next four consecutive days.

We compared PRISM to the best algorithms from the literature for each ELE: the algorithm of Gal et al. (2012) for TinkerPlots, and the algorithm of Gal and Amir (2011) for VirtualLabs. For each problem instance, a domain expert was given the plans outputted by PRISM and the other algorithm, as well as the student's log. We consider the inferred plan to be "correct" if the domain expert agrees with the complex and basic actions at each level of the plan hierarchy that is outputted by the algorithm. The outputted plan represents the student's solution process using the software.

To illustrate how plans were presented to domain experts, Figure 5 shows the visualization of the final plan for the log actions in the leaves of Figure 3.[2] The visualization groups all trees in the student's plans as children to a single root node "Solve Oracle problem". Complex nodes are labeled with information about the chemical reactions that occurred during the activities described by the nodes. For example, each of nodes labeled $A+B+C+D \longrightarrow A+B+D$ represents an activity of mixing four solutions together which resulted in a chemical reaction that consumed substance C. (In effect, the reaction also increased the amount of substances A and D. This amount is displayed in the data panel below the plan in Figure 5). The coloring of the labels indicate the type of chemical reaction that has occurred.

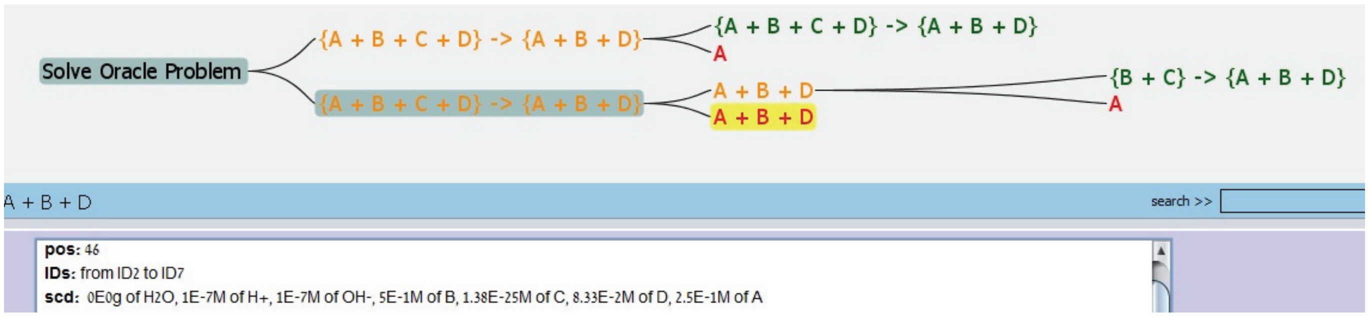
Table 1 summarizes the performance of the PRISM algorithm according to accuracy and run time of the algorithm (in seconds on a commodity core i-7 computer). The column "SoA" (State-of-the-art) refers to the appropriate algorithm from the literature for each problem. All of the reported results were averaged over the different instances in each problem. As shown in the table, the PRISM algorithm was able to recognize significantly more plans than did the state-of-the-art ($p < 0.001$ using a proportion based Z test). Specifically, the PRISM algorithm was able to recognize all of the plans in both ELEs that were correctly recognized by the state-of-the-art. In VirtualLabs, the state-of-the-art algorithms were not able to recognize ten plan instances (3 for Oracle; 3 for Unknown Acid; 3 for Coffee and 1 for Dilution). In TinkerPlots, the state-of-the-art algorithm failed to recognize 10 plan instances (5 for Rain; 5 for ROSA). The instances that the algorithms failed to recognize are false negatives that represent bad matches in the plan recognition process.

We conclude this section with discussing the limitations of PRISM. First, PRISM is not a complete plan recognition algorithm, as can be attested by the fact that it failed to recognize 6 out of 40 instances in TinkerPlots. Second, it was significantly slower than the state-of-the-art approaches. This is not surprising given its worst case complexity. Although PRISM is, in practice, polynomial in the size of the log and recipes, the algorithm by Gal and Amir is only polynomial in the size of recipes (which is significantly smaller than the log size). However, PRISM is designed to run offline after the completion of the student's interaction. Therefore an average run time of 25 seconds is a "low price to pay" given the significant increase in performance and its ability to generalize across different ELEs.

Conclusion and Future Work

This paper proposed a new algorithm for recognizing students' plans in exploratory domains. The algorithm is empirically evaluated using a new type of open-ended educational software called Exploratory Learning Environments (ELE). The algorithm is shown to significantly outperform the state-of-the-art plan recognition algorithms for two different ELEs for teaching chemistry and statistics. It is also the first recognition algorithm that is shown to generalize successfully to several ELEs. We are currently applying these results in several directions. First, we are developing a recognition algorithm that will be used for on-line recognition of students' plans and based on probabilistic parsing

[2]The visualization is meant to facilitate the analysis of the expert by including additional information from the log. It does not perform any inference over the output of PRISM.


Solve Oracle Problem
{A + B + C + D} -> {A + B + D}
{A + B + C + D} -> {A + B + D}
A
{A + B + C + D} -> {A + B + D}
A + B + D
A + B + D
{B + C} -> {A + B + D}
A
A + B + D
search >>

pos: 46
IDs: from ID2 to ID7
scd: 0E0g of H_2O, 1E-7M of H+, 1E-7M of OH-, 5E-1M of B, 1.38E-25M of C, 8.33E-2M of D, 2.5E-1M of A

Figure 5: Visualization of Oracle Plan

		num of instances	PRISM accuracy	PRISM run-time	SoA accuracy	SoA run-time
Virtual Labs	Oracle	6	100%	8.015	50%	1.06
	Unknown Acid	7	100%	31.589	57%	0.8
	Camping	2	100%	0.929	100%	0.4
	Coffee	9	100%	17.567	67%	0.4
	Dilution	4	100%	1.529	75%	0.54
TinkerPlots	Rosa	23	87%	55.22	78%	3.34
	Rain	17	82%	3.049	71%	0.54
Average			91%	25.841	71%	1.537

Table 1: Results of PRISM algorithm

methods. Second, we are investigating methods for aggregated analysis of students' interactions by adapting clustering methods to work on the space of students' plans.

References

Amershi, S., and Conati, C. 2006. Automatic recognition of learner groups in exploratory learning environments. In *Intelligent Tutoring Systems (ITS)*.

Amir, O., and Gal, Y. 2011. Plan recognition in virtual laboratories. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*.

Babaian, T.; Grosz, B. J.; and Shieber, S. M. 2002. A writer's collaborative assistant. In *Intelligent User Interfaces Conference*, 7–14.

Cocea, M.; Gutierrez-Santos, S.; and Magoulas, G. 2008. S.: The challenge of intelligent support in exploratory learning environments: A study of the scenarios. In *Proceedings of the 1st International Workshop in Intelligent Support for Exploratory Environments on European Conference on Technology Enhanced Learning*.

Conati, C.; Gertner, A.; and VanLehn, K. 2002. Using Bayesian networks to manage uncertainty in student modeling. *User Modeling and User-Adapted Interaction* 12(4):371–417.

Gal, Y.; Yamangil, E.; Rubin, A.; Shieber, S. M.; and Grosz, B. J. 2008. Towards collaborative intelligent tutors: Automated recognition of users' strategies. In *Intelligent Tutoring Systems (ITS)*.

Gal, Y.; Reddy, S.; Shieber, S.; Rubin, A.; and Grosz, B. 2012. Plan recognition in exploratory domains. *Artificial Intelligence* 176(1):2270 – 2290.

Konold, C., and Miller, C. 2004. *TinkerPlots Dynamic Data Exploration 1.0*. Key Curriculum Press.

Pawar, U.; Pal, J.; and Toyama, K. 2007. Multiple Mice for Computers in Education in Developing Countries. In *Conference on Information and Communication Technologies and Development*, 64–71.

Ryall, K.; Marks, J.; and Shieber, S. M. 1997. An interactive constraint-based system for drawing graphs. In *Proceedings of the 10th Annual Symposium on User Interface Software and Technology (UIST)*.

VanLehn, K.; Lynch, C.; Schulze, K.; Shapiro, J. A.; Shelby, R. H.; Taylor, L.; Treacy, D. J.; Weinstein, A.; and Wintersgill, M. C. 2005. The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence and Education* 15(3).

Vee, M.; Meyer, B.; and Mannock, K. 2006. Understanding novice errors and error paths in object-oriented programming through log analysis. In *Proceedings of Workshop on Educational Data Mining at ITS*, 13–20.

Yaron, D.; Karabinos, M.; Lange, D.; Greeno, J.; and Leinhardt, G. 2010. The ChemCollective–Virtual Labs for Introductory Chemistry Courses. *Science* 328(5978):584.