Towards Collaborative Intelligent Tutors: Automated Recognition of Users' Strategies

Ya'akov Gal[1,2], Elif Yamangil[2], Stuart M. Shieber[2], Andee Rubin[3], and Barbara J. Grosz[2]

[1] MIT Computer Science and Artificial Intelligence Laboratory
[2] Harvard School of Engineering and Applied Sciences
[3] TERC

Abstract. This paper addresses the problem of inferring students' strategies when they interact with data-modeling software that is used for pedagogical purposes. This software enables students to learn about statistical data by building and analyzing their own models. Automatic recognition of students' activities when interacting with pedagogical software is challenging. Students can pursue several plans in parallel and interleave the execution of these plans. The algorithm presented in this paper decomposes students' complete interaction histories with the software into hierarchies of interdependent tasks that may be subsequently compared with ideal solutions. This algorithm is evaluated empirically using commercial software that is used in many schools. Results indicate that the algorithm is able to (1) identify the plans students use when solving problems using the software; (2) distinguish between those actions in students' plans that play a salient part in their problem-solving and those representing exploratory actions and mistakes; and (3) capture students' interleaving and free-order action sequences.

1 Introduction

We report on the development of algorithms for recognizing students' plans when interacting with pedagogical systems for data-generation and analysis. This work is a first step towards building a collaborative pedagogic agent that will support students in their problem-solving and teachers in their analysis of students' modeling and understanding of statistical data. TinkerPlots, the system we use in this paper, gives students great flexibility in representing and analyzing statistical data. It is in essence a data-analysis "construction kit" that allows students to create and analyze a large number of statistical models [8]. While this makes for a rich educational environment, it does pose significant problems for teachers. When an entire classroom of students is using TinkerPlots at the same time, there is no way for a teacher to keep track of what each child is doing, especially since they may be following divergent paths in solving the problem. Without some sort of support, teachers are left with the end-result of students' work on the computer screen, or looking over the shoulder of each student for at most a minute or two.

Automatic plan recognition is an open problem in AI, and the task of recognizing users' plans when interacting with software systems is particularly challenging. Ideally designed systems are flexible, allowing users the convenience of choosing among multiple action sequences for performing the same function, and the ability to perform these action sequences in relatively free order. Traditional algorithms for plan recognition assume a goal-oriented agent whose activities are consistent with its knowledge base and who forms a single encompassing plan [9]. In contrast, one of the objectives of flexible pedagogical software is to allow students to explore and experiment during their interaction process. In these settings, students may interchangeably pursue multiple, interleaving plans; they may be confused about which appropriate plan to take, and they may make mistakes. Recognizing students' actions by exhaustively considering every possible way in which a student can use these systems is infeasible.

This paper describes a computationally tractable algorithm for intelligently recognizing students' problem-solving strategies based on their complete interaction history with the system. The algorithm composes the action sequences from a user's interaction into a series of interdependent plans. It infers the plan that the user was using to complete each activity, and compares this plan with an ideal solution that was designed by domain experts. At the end of this process, the algorithm outputs a hierarchy of the plans that students used during the session and the extent to which they differed from the ideal solutions.

The algorithm was tested using the commercial system TinkerPlots, used world-wide to teach students in grades 4-8 about statistics and mathematics [5]. In TinkerPlots, students actively model stochastic events and construct models that generate data. TinkerPlots is highly 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.

In AI, plan recognition has been used in a range of applications, such as modeling discourse structure from speech and inferring transportation routines from GPS data [11, 10]. Past work in the intelligent tutoring domain has focused on inferring students' activities for the purpose of providing feedback by the tutor. These models have been used for modeling how students solve math [6, 3] and physics problems [4, 14], their help requests from pedagogical software or their misuse of it [13, 2]. Many of these works construct a probabilistic model of students' problem-solving strategies that is subsequently used to update the tutor's beliefs about students' likely future actions given their behavior. In these cases, 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 [1].

In contrast, the work reported in this paper addresses the problem of recognizing students' actions given their complete interaction histories. The system does not intervene with the student's activities during the course of interaction. Straightforward adaptation of probabilistic techniques for this purpose is difficult, because the size of probabilistic models is typically exponential in the length of the history they consider, and students' complete interaction histories often span hundreds of actions. In addition, the model parameters must be trained from data or stipulated by a domain expert. Both of these techniques require considerable effort in the domain we consider.

1.1 Example Scenario: The Two-Dice Problem

To illustrate the algorithm we will use the following example, drawn from a set of problems posed to seventh grade students using TinkerPlots. "We rolled two dice over and over a huge number of times and kept track of their sums. For example, when the first die came up 5 and the second die came up 6, we recorded their sum of 11. Using TinkerPlots, build a model that you can use to roll two dice 1,000 times and see whether 11 came up more often than 12."

The purposes behind this exercise are for children to learn about the joint distribution of non-ordered random events and to explore how sample distributions vary, even if they are drawn from the same population. Each roll of two dice generates a pair of values, one for each die. There are two events that make up the sum 11, namely (5,6) and (6,5), while there is only one event that makes up the sum 12, namely (6,6). Since each of these events is equally likely, in theory the sum 11 will occur more often than the sum 12. (Of course, as students run their models, they will discover that, while this is generally true, there will be samples in which there are more 12s than 11s, especially if the sample size is small.)

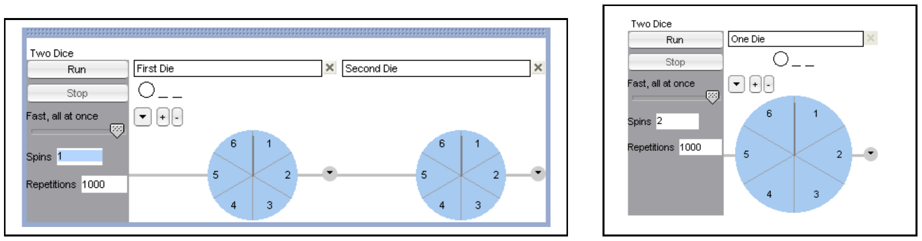
One of the possible approaches towards modeling this situation using TinkerPlots is shown in Figure 1. The model includes a sampler device comprising two spinners, shown in Figure 1(a), each of which is a model of one die. The sampler will randomly select a value for each of its spinners every time it is run. Each spinner has six possible values. The surface area specified for each value determines its weight in the sample. Effectively, this sampler models a joint probability distribution over two independent random variables with six values distributed uniformly. The value of "Repetitions", set to 1,000 in this example, determines the number of times the sampler is run. The value of "Spins", set to "1" in this example, determines the number of rolls of the two dice at each run.

Figure 1(b) shows some of the data generated by the sampler once it has run. Each pair in the table represents a roll of two dice. This pair has been separated, by instigating a "Separate Individual Draws" function in the sampler. To the right is a graphical representation of all of the sums in the form of a histogram. Figure 1(a) shows an additional way to model this problem. Here, a single die is used that is thrown twice at each repetition, hence the value of "spins" is set to "2". There are many other ways to use TinkerPlots to solve this problem.

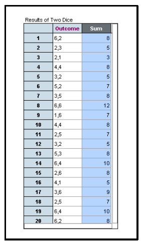
2 Recipes, Planning and Plan Recognition

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

Fig. 1: TinkerPlots Sessions Snapshot

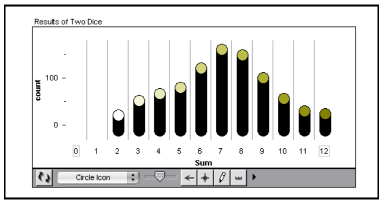


(a) Two Possible Sampler Models



Results of Two Dice

	Outcome	Sum
1	6,2	8
2	2,3	5
3	2,1	3
4	4,4	8
5	3,2	5
6	5,2	7
7	3,5	8
8	6,6	12
9	1,6	7
10	4,4	8
11	2,5	7
12	3,2	5
13	5,3	8
14	6,4	10
15	2,6	8
16	4,1	5
17	3,6	9
18	2,5	7
19	6,4	10
20	6,2	8



(b) Displaying Sampler Data as a Histogram

will use the term *basic actions* to refer to these operations, which can often be carried out by a single keystroke or mouse action. TinkerPlots interactions are recorded as a linear sequence of basic actions in order of their occurrence. Each basic action uses a unique tag to refer to an object, which is transparent to the user. A subset of such an interaction sequence is shown in the leaves of the trees in Figure 3. For example, the basic action $\mathsf{New}(\mathsf{Spinner}(S_1))$ adds a new spinner with ID S_1. These actions are serially labeled in order of occurrence. (Due to layout constraints, the leaves in this figure are not aligned on the same plane.)

We model students' reasoning about problems using abstract entities, called *complex actions*, which capture higher-level, more abstract TinkerPlots activities, such as adding two dice to a sampler, computing the sum of a roll of two dice, or fitting sampler data to plot. Complex actions can be decomposed into sub-actions [7]. A sub-action can be a basic TinkerPlots action or it can be a complex action itself. A useful distinction between complex and basic actions is that students can "see" both basic and complex actions, while the TinkerPlots system can only "see" and register basic actions.

A *recipe* for a complex action is an ideal sequence of operations for fulfilling the complex action. Formally, a recipe is a set of sub-actions and constraints such that performing those sub-actions under those constraints constitutes completing the action [12]. These sub-actions are referred to as the recipe's constituents. Figure 2 presents recipes for solving the two-dice problem and its constituent sub-actions. Each recipe for a complex action is represented as a tree of depth two, in which the leaves correspond to the recipe's constituent actions (whether basic or complex), and the root corresponds to the complex action. Basic actions

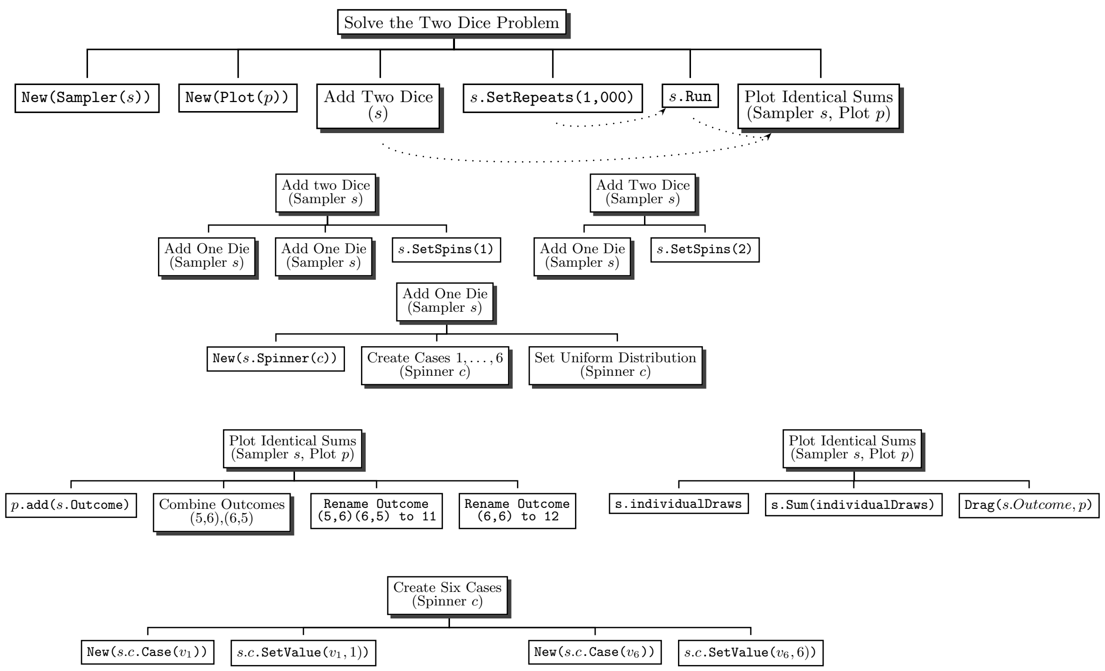

Solve the Two Dice Problem
New(Sampler(s))
New(Plot(p))
Add Two Dice (s)
s.SetRepeats(1,000)
s.Run
Plot Identical Sums (Sampler s, Plot p)
Add two Dice (Sampler s)
Add One Die (Sampler s)
Add One Die (Sampler s)
s.SetSpins(1)
Add Two Dice (Sampler s)
Add One Die (Sampler s)
s.SetSpins(2)
Add One Die (Sampler s)
New(s.Spinner(c))
Create Cases $1, \ldots, 6$ (Spinner c)
Set Uniform Distribution (Spinner c)
Plot Identical Sums (Sampler s, Plot p)
p.add(s.Outcome)
Combine Outcomes (5,6),(6,5)
Rename Outcome (5,6)(6,5) to 11
Rename Outcome (6,6) to 12
Plot Identical Sums (Sampler s, Plot p)
s.individualDraws
s.Sum(individualDraws)
Drag($s.Outcome, p$)
Create Six Cases (Spinner c)
New($s.c$.Case(v_1))
$s.c$.SetValue(v_1, 1))
New($s.c$.Case(v_6))
$s.c$.SetValue(v_6, 6))

Fig. 2: Recipes for Solving the Two-Dice Problem. Dashed edges represent temporal constraints between actions.

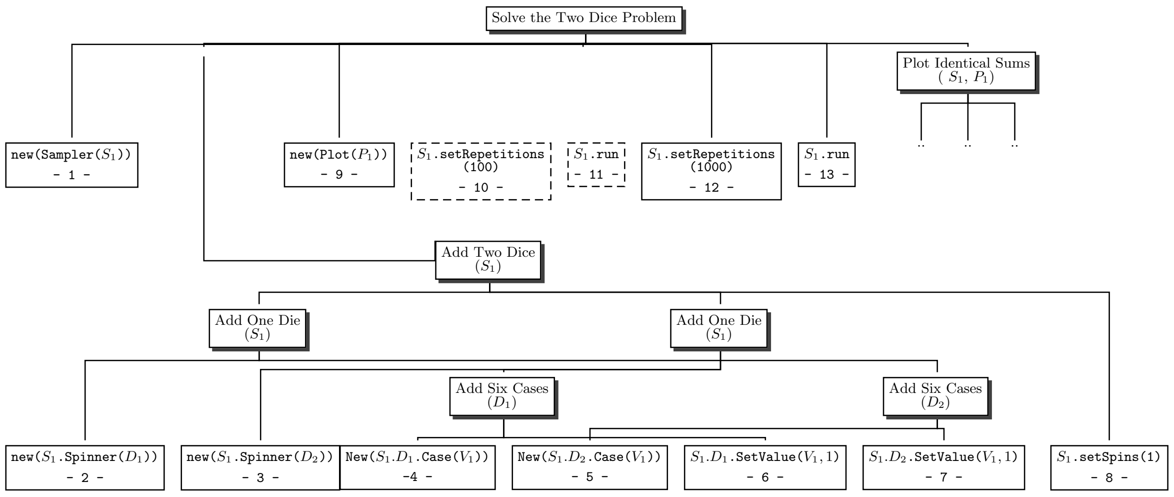
are outlined in plain boxes, while complex actions are outlined in shadowed boxes. TinkerPlots objects are identified by a unique tag, and recipe actions use parameters to refer to the TinkerPlots objects they modify. For example the recipe for the complex action AddTwoDice (s) modifies the sampler object that is bound to the parameter s.

The order in which actions are performed in a recipe can be constrained by including temporal constraints between actions, represented as a dotted edge. Actions within the same recipe can occur in any order as long as they meet the specified temporal constraints. For example, in the recipe for the action SolveTheTwoDiceProblem, both actions AddTwoDice (s) and s.SetRepeats(1,000) can come in any order as long as they both occur before the basic action s.Run.

In addition to the constraints embedded in the recipes, some action combinations are disallowed by the TinkerPlots system itself. For example, it is impossible to add a spinner to a sampler until the sampler has been created. For expository convenience, we do not show these constraints in the recipes.

Recipes may be ambiguous, in the sense that there may be several recipes for completing the same complex action. For example, Figure 2 shows two possible recipes for completing the complex action AddTwoDice (s). One possible recipe uses a single die that is rolled twice. It includes the sub-actions AddOneDie (s) and s.SetSpins(1). The other recipe uses two dice that are rolled once. It includes

Fig. 3: A Sample Plan


Solve the Two Dice Problem
Plot Identical Sums (S_1, P_1)
new(Sampler(S_1)) - 1 -
new(Plot(P_1)) - 9 -
S_1.setRepetitions (100) - 10 -
S_1.run - 11 -
S_1.setRepetitions (1000) - 12 -
S_1.run - 13 -
Add Two Dice (S_1)
Add One Die (S_1)
Add One Die (S_1)
Add Six Cases (D_1)
Add Six Cases (D_2)
new(S_1.Spinner(D_1)) - 2 -
new(S_1.Spinner(D_2)) - 3 -
New(S_1.D_1.Case(V_1)) -4 -
New(S_1.D_2.Case(V_1)) - 5 -
S_1.D_1.SetValue(V_1, 1) - 6 -
S_1.D_2.SetValue(V_1, 1) - 7 -
S_1.setSpins(1) - 8 -

two sub-actions AddOneDie (s) and the sub-action s.SetSpins(1). In addition, the same action may be a constituent of several different recipes. For example, the complex action AddOneDie (s) appears in both recipes for the complex action AddTwoDice (s).

2.1 Planning

Planning is the process by which students use recipes to compose basic and complex actions towards completing tasks using TinkerPlots. We say that a recipe for a complex action is *fulfilled* by a set of temporally-ordered sub-actions if (1) there is a one-to-one correspondence from each of the sub-actions to one of the recipe's constituents; (2) all of the sub-actions agree on the identification tags for the TinkerPlots objects that are modified by the recipe; and, (3) the order between sub-actions is consistent with the temporal constraints that are defined between recipe constituents. Formally, a plan is an ordered set of basic and complex actions, such that each complex action is decomposed into sub-actions that fulfill a recipe for some task. Each time a recipe for a complex action is fulfilled in a plan, there is an edge from the complex action to its sub-actions, representing the recipe constituents. For example, in Figure 3, the recipe for the complex action AddTwoDice(S_1) is fulfilled by the two AddOneDie(S_1) actions and the action S_1.SetSpins(1).

Each tree in Figure 3 represents a plan that was carried out by the student. The leaves of the trees represent the basic actions corresponding to the user's interaction history. (For expository convenience, we have only included a subset of this interaction history.) The plan that emanates from the complex action

SolveTheTwoDiceProblem shows that the student was able to complete the two-dice problem.

In a plan, the constituent sub-actions of complex actions may interleave with other actions. In this way, the plan combines the free-order nature of TinkerPlots recipes with the exploratory nature of students' learning strategies. Formally, we say that two ordered complex actions *interleave* if at least one of the sub-actions of the first action occurs after some sub-action of the second action.

An example of interleaving actions in this plan are the two complex actions AddOneDie (S_1) We can see this because a constituent of the recipe for the first AddOneDie (S_1) (the action AddSixCases (D_2)) occurs after a constituent of the recipe for the second AddOneDie (S_1) (the action AddSixCases(D_1)). In Figure 3, there are crossing edges between the constituent sub-actions of any two interleaving actions.

Also shown in Figure 3 are two basic actions outlined in dashed boxes (S_1.SetRepeats(100) and S_1.run) that were not necessary for solving the two-dice problem. This happened because the user first ran the sampler for 100 repetitions, before running the sampler for 1,000 repetitions, as required by the problem formulation. These could represent a student's exploration or a mistake.

3 Plan Recognition

The task of *plan recognition* in the TinkerPlots domain is to infer students' plans based on their interaction history and a set of recipes. A naive approach would search through the space of all possible plans that are consistent with a user's interaction, the recipes, and their constraints. This approach is not feasible. For each possible action in the plan, we would need to consider all possible expansions of basic and complex sub-actions as long as their order is permitted by the recipe constraints. In the worst case, the number of possible plans to consider will be factorial in the number of basic actions in an interaction sequence.

However, certain qualities of the TinkerPlots domain serve to constrain the search process. First, it is not possible to generate an infinite plan using TinkerPlots recipes. Therefore, we can choose the recipes to be fulfilled in an incremental fashion, ordered by depth. We define the "depth" of a recipe for a complex action as the maximum depth of the tree for any plan to complete the complex action[4]. Second, the sub-actions of a complex action will always agree on the identification tags of those TinkerPlots objects that are modified by the complex action. Therefore, we do not need to consider any action combination that disagree on the ID tags. We can also ignore those action combinations disallowed by the TinkerPlots system.

As a result, we can construct the following algorithm that incrementally builds a sequence of plans to explain a user's interaction history from the leaves upwards. Each step t of the algorithm maintains an ordered set of actions, denoted P_t. Each of these actions is a root of a tree that is a partial plan that

[4] For example, the depth of the recipe for solving the two-dice problem is three, because there is no possible plan for this task whose depth is greater than three.

explains some subset of the user's interaction history. P_0 is initialized to include all of the basic actions in the interaction history. Let G be the set of recipes, and let the recipe in G for a complex action C be denoted as R_C. The algorithm proceeds as follows:

```
For each $R_C$ in $G$, sorted by depth
  Initialize $P_{t+1}$ with $P_t$
  For any sequence $S_{t+1}$ of actions in $P_{t+1}$ that fulfill $R_C$:
    Add a new action $C$ in $P_{t+1}$, positioned after the first action in $S_{t+1}$
    Let $S_t$ be the set of actions in $P_t$ corresponding to $S_{t+1}$
    Add edges from $C$ in $P_{t+1}$ to all actions in $S_t$
    Remove all actions in $S_{t+1}$ from $P_{t+1}$
```

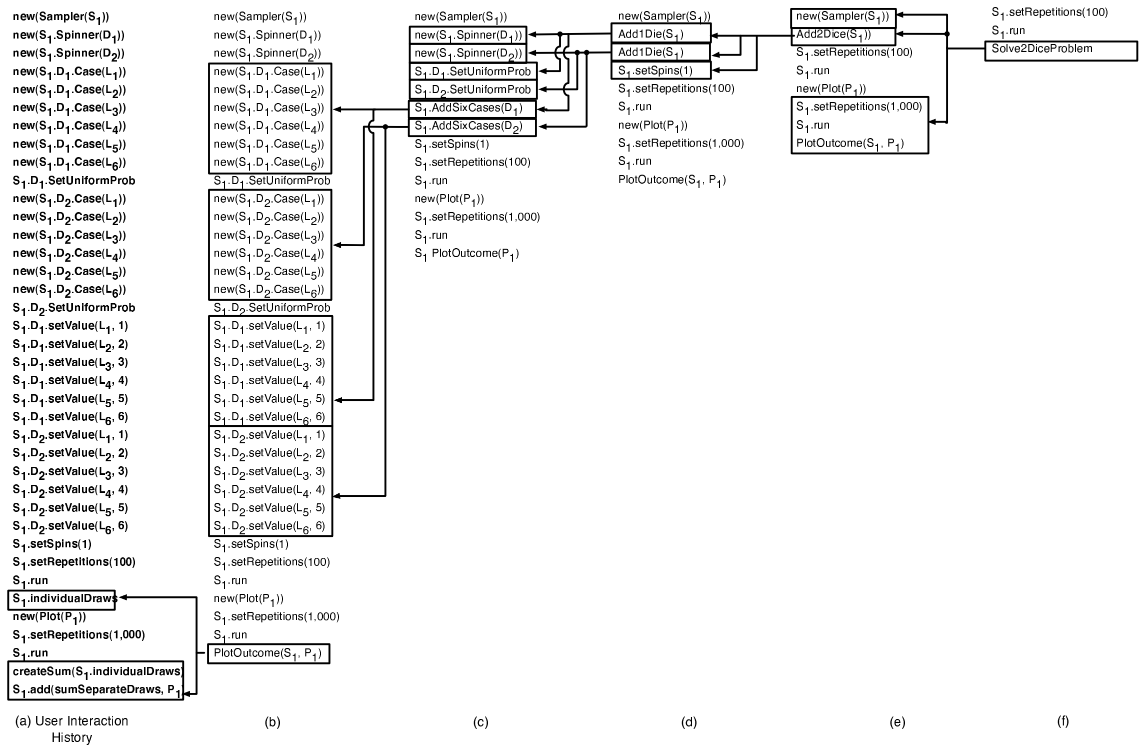
A key step in this algorithm is the selection of actions in P_{t+1} to fulfill the recipe for R_C. We select these actions in any order that is allowed by the temporal constraints of the recipe for R_C. In particular, the actions in P_{t+1} may be non-contiguous; this allows the algorithm to capture interleaving plans. This step is greedy, because once a complex action is chosen to fulfill R_C, it is removed from P_{t+1} and will not be considered again. Therefore, there may be instances where the algorithm fails because it picked the wrong actions to fulfill a recipe in earlier steps. An interesting consequence is that the order in which we traverse the actions in P_{t+1} can affect the way in which the algorithm fulfills recipe, and thus, its output. We currently do so by traversing P_{t+1} sequentially, from the last action to the first. A different order may fulfill different recipes, and produce a different plan.

The complexity of this approach can be computed as follows. Let n be the length of a student's interaction sequence. The number of times a recipe can be fulfilled is bounded by n. In the worst case, it will take a complete pass over the actions in P_t to fulfill the recipe. The number of actions in P_t is bounded by n. Therefore, it will take at most n^2 steps to exhaust all of the possible applications of R_C. In fact, a slightly more sophisticated implementation complete this process in linear time. Given that the size of the recipes is constant, we conclude that the complexity of the algorithm is quadratic in the length of the interaction history.

We demonstrate part of this process in Figure 4. We show the complete interaction history of the user in Figure 4(a), outlined in bold. (This is the same interaction history of the plan in Figure 3). These actions are presented top-to-bottom in order of their occurrence. Each sub-figure in Figure 4 shows the partial plans that the algorithm maintains for recipes of a given depth. When fulfilling a recipe for a complex action, we draw directed edges from the sub-actions to the complex action. For instance, in the step shown in Figure 4 (b), the algorithm fulfills two separate instances of the recipe AddSixCases (d), one for spinner ID D_1 and one for spinner ID D_2. These complex actions interleave, as can be seen from the crossing edges.

In the step shown in Figure 4(d), the algorithm chooses to fulfill one of two possible recipes for completing the complex action AddTwoDice (s). This choice is possible because of the basic action S_1.SetSpins(1), which is a unique

Fig. 4: The execution of the algorithm on a user's interaction history. Crossing edges represent interleaving actions.



new(Sampler(S_1))
new(S_1.Spinner(D_1))
new(S_1.Spinner(D_2))
new(S_1.D_1.Case(L_1))
new(S_1.D_1.Case(L_2))
new(S_1.D_1.Case(L_3))
new(S_1.D_1.Case(L_4))
new(S_1.D_1.Case(L_5))
new(S_1.D_1.Case(L_6))
S_1.D_1.SetUniformProb
new(S_1.D_2.Case(L_1))
new(S_1.D_2.Case(L_2))
new(S_1.D_2.Case(L_3))
new(S_1.D_2.Case(L_4))
new(S_1.D_2.Case(L_5))
new(S_1.D_2.Case(L_6))
S_1.D_2.SetUniformProb
S_1.D_1.setValue(L_1, 1)
S_1.D_1.setValue(L_2, 2)
S_1.D_1.setValue(L_3, 3)
S_1.D_1.setValue(L_4, 4)
S_1.D_1.setValue(L_5, 5)
S_1.D_1.setValue(L_6, 6)
S_1.D_2.setValue(L_1, 1)
S_1.D_2.setValue(L_2, 2)
S_1.D_2.setValue(L_3, 3)
S_1.D_2.setValue(L_4, 4)
S_1.D_2.setValue(L_5, 5)
S_1.D_2.setValue(L_6, 6)
S_1.setSpins(1)
S_1.setRepetitions(100)
S_1.run
S_1.individualDraws
new(Plot(P_1))
S_1.setRepetitions(1,000)
S_1.run
createSum(S_1.individualDraws)
S_1.add(sumSeparateDraws, P_1)
PlotOutcome(S_1, P_1)
new(S_1.Spinner(D_1))
new(S_1.Spinner(D_2))
S_1.D_1.SetUniformProb
S_1.D_2.SetUniformProb
S_1.AddSixCases(D_1)
S_1.AddSixCases(D_2)
S_1 PlotOutcome(P_1)
Add1Die(S_1)
Add1Die(S_1)
new(Sampler(S_1))
Add2Dice(S_1))
Solve2DiceProblem
(a) User Interaction History
(b)
(c)
(d)
(e)
(f)

constituent action for one of the recipes for CreateTwoDice (s) but not for the other. In the step shown in Figure 4(e), the algorithm succeeds in collapsing the complex action Solve Two-Dice Problem, and terminates, because it cannot fulfill any more recipes. As shown in the final step in Figure 4(f), the algorithm has determined that two actions (S_1.SetRepeats (100) and S_1.Run) were redundant.

3.1 Evaluation

We collected eight TinkerPlots interaction histories of six people using TinkerPlots to solve the two-dice problem, and two people using TinkerPlots to solve another problem involving the modeling of ordered stochastic events. Two of these people were middle school students in an after-school TinkerPlots club. Three were adults who had experience with using technology in education, but not with TinkerPlots. One was an adult who was very familiar with TinkerPlots. The middle school students had been using TinkerPlots for several months and did the two-dice problem as part of their regular after-school work. The three adults who were not familiar with TinkerPlots watched a 5-minute introductory video, saw a brief demonstration of the Sampler functions, then did the problem.

In all cases, an experimenter tracked the activities of each participant (e.g., what samplers were created, when actions were interleaved, etc.).

We considered the plan constructed by an algorithm to be "correct", if the actions in the plan corresponded to the students' actual activities using the software, including the interleaving of action sequences. The algorithm was able to recognize the strategies for all of these interaction histories but one. In this instance, a student solved the two dice problem twice, using the same sampler in both solutions. The algorithm recognized one solution, but not the other. This is because the plan recognition algorithm grows a sequence of trees, so the same action cannot simultaneously fulfill several recipes.

One approach to be able to consider all ways of fulfilling a recipe, is to build possible partial plans in parallel, rather than greedily. Once the current partial plan reaches a dead-end, the algorithm backtracks. Because partial plans in TinkerPlots may interleave, there is no straightforward way to accomplish this without having to construct a separate partial plan for each possible way a recipe can be fulfilled. This naive approach is exponential in the length of interaction, hence computationally intractable. We intend to see whether dynamic programming can be used to make this process more efficient.

Lastly, it is important to note that even a partial account of students' interactions can still convey information about the techniques they used and their approach that is useful to teachers. For example, the greedy algorithm was still able to recognize all of the constituent actions for the second application of the recipe for the two-dice problem.

4 Conclusion and Future Work

This work presented a simple and computationally efficient algorithm for recognizing students' interactions with data-modeling software. The algorithm is able to capture the nature of interaction of users with flexible computer software, which allow users to interleave their activities in relatively free order. We showed that the algorithm was successfully able to recognize students' plans when solving two separate problems using a commercially available application.

This work is a first step towards a pedagogical agent that is truly collaborative, in the sense that it provides the right machine-generated support for its users. For teachers, this support consists of notification of students' performance both after and during class. For students, this support will guide their problem-solving in a way that maximizes their learning experience while minimizing interruption.

To this end, we are currently pursuing work in several directions. First, we are constructing vivid, coherent representations of students' plans to show teachers. These presentations need to support a "birds' eye view" of class performance during a session, as well as the ability to focus on the behavior of individual students. We will develop algorithms that enable teachers to access the state of the system at critical points in students' work. The system state conveys different information from a plan, in that it provides a snapshot of the TinkerPlots objects a user is using at a given point in time, rather than a post-session analysis of students' interaction. Our future research will include developing algorithms for keeping track of the state of the system and experimenting with presenting teachers with some combination of plan information and state information. Lastly, we are pursuing a machine-learning approach towards learning recipes from data by observing students' interaction.

5 Acknowledgements

This work is supported by NSF grant number REC-0632544. Thanks very much to Heather Pon-Barry and Swapna Reddy for useful comments and discussion.

References

1. J. R. Anderson, A. T. Corbett, K. Koedinger, and R. Pelletier. Cognitive tutors: Lessons learned. *The Journal of Learning Sciences*, 4(2):167–207, 1995.
2. R. S. Baker, A. T Corbett, K. R. Koedinger, and I. Roll. Generalizing detection of gaming the system across a tutoring curriculum. In *Proc. of 8th Internatioanl Conference on Intelligent Tutoring Systems*, 2006.
3. J.E. Beck and B.P. Wolf. Using a learning agent with a student model. In *Proc. of 4th international conference on Intelligent Tutors*, 1998.
4. Conati C., Gertner A., and VanLehn K. Using bayesian networks to manage uncertainty in student modeling. *Journal of User Modeling and User-Adapted Interaction,*, 12(4):371–417, 2002.
5. C. Miller C. Konold. *TinkerPlots Dynamic Data Exploration 1.0.* Key Curriculum Press, 2004.
6. A. Corebette, M. McLaughlin, and K.C. Scarpinatto. Modeling student knowledge: Cognitive tutors in high school and college. *User Modeling and User-Adapted Interaction*, 10:81—108, 2000.
7. B.J. Grosz and S. Kraus. The evolution of sharedplans. *Foundations and Theories of Rational Agency*, pages 227–262, 1999.
8. J. K. Hammerman and A. Rubin. Strategies for managing statistical complexity with new software tools. *Statistics Education Research Journal*, 3(2):17–41, 2004.
9. H. Kautz. *A formal theory of plan recognition.* PhD thesis, University of NY, Rochester, 1987.
10. L. Liao, D.J. Patterson, D. Fox, and H. Kautz. Learning and inferring transportation routines. *Journal of Artificial Intelligence Research*, 171:311–331, 2007.
11. K. Lochbaum. A collaborative planning model of intentional structure. *Computational Linguistics*, 4(525–572), 1998.
12. M. Pollack. Plans as complex mental attitudes. MIT Press, 1990.
13. I. Roll, V. Aleven, B. M. McLaren, and K. R Koedinger. Can help seeking be tutored? In *International Conference on Artificial Intelligence in Education 2007*, 2007.
14. K. Vanlehn, C. Lynch, K. Schulze, J. A. Shapiro, R. H. Shelby, L. Taylor, D. J. Treacy, A. Weinstein, and M. C. Wintersgill. The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence and Education*, 15(3), 2005.