Towards Automatic Hint Generation for a Data-Driven Novice Programming Tutor

I. WEI JIN
Shaw University, USA
AND
I.I. LORRIE LEHMANN
University of North Carolina at Charlotte, USA
AND
I.I.I. MATTHEW JOHNSON
University of North Carolina at Charlotte, USA
AND
I.V. MICHAEL EAGLE
University of North Carolina at Charlotte, USA
AND
V. BEHROOZ MOSTAFAVI
University of North Carolina at Charlotte, USA
AND
V.I. TIFFANY BARNES
University of North Carolina at Charlotte, USA
AND
V.I.I. JOHN STAMPER
Carnegie Mellon University, USA

Hint annotation is one of the most time consuming components of developing intelligent tutoring systems. One approach is to use educational data mining and machine learning techniques to automate the creation of hints from student problem-solving data. This paper describes a new technique to represent, classify, and use programs written by novices as a base for automatic hint generation for programming tutors. Our preliminary evaluation shows that this approach can effectively cluster programs and therefore has potential to be a source for automatically generated hints for novice programmers.

Key Words and Phrases: Intelligent tutoring systems, automatic hint generation, programming tutors, educational data mining and data clustering

1. INTRODUCTION

Hint annotation is one of the most time consuming components of developing intelligent tutoring systems. Barnes and Stamper developed a data-driven educational data mining

Authors' addresses: Wei Jin, Department of Computer Information Sciences, Shaw University, Raleigh, NC, USA. E-mail: wjin@shawu.edu; Lorrie Lehmann, Matthew Johnson, Michael Eagle, Behrooz Mostafavi, and Tiffany Barnes, Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC, USA. E-mails: ljlehman@uncc.edu; matjohns@uncc.edu; mjeagle@uncc.edu; bzmostaf@uncc.edu; tiffany.barnes@gmail.com; John Stamper, Human-Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA, USA. E-mail: jstamper@cs.cmu.edu.

technique that uses a Markov decision process (MDP), created from past student data, to generate contextualized hints for students solving a specific problem. They have applied this domain-independent approach called the Hint Factory in a logic tutor called Deep Thought, and shown that it can provide hints for 70-90% of problem-solving steps, and hints prevent students from abandoning the logic tutor [Barnes2010a, Barnes2010b, Stamper2010a, and Stamper2010b].

We plan to extend this MDP-based approach to create a data-driven programming tutor. To do so, we must find a representation for novice programs that shows the incremental steps taken towards complete, correct programs, and build a way to compare the current "target" program to another student's completed, more correct "source program" for hint generation. In this paper, we propose a new technique to represent and classify student programs for hint generation. This process requires us to create a state representation for programs that result in similar states for programs with the same behavior. For statements that do not have variable dependencies among each other, as in Figure 1, any ordering of these statements should not affect the program representation. We should also be able to classify seemingly different but equivalent programs. For example, the three program segments in Figure 2 assign the same value to variable *a*.

a = b * c; **d = e * f;**	**d = e * f;** **a = b * c;**

Figure 1. Different orders but equivalent

We propose to represent a program as a Linkage Graph. A linkage graph can represent a program in a concise manner but is also general enough to include all programs with the same fundamental structure. We also propose Condensed Linkage Graphs to classify equivalent programs like those in Figure 2.

cin >> b; **cin >> c;** **cin >> d;** **a = b * c / d;**	**cin >> b;** **cin >> c;** **cin >> d;** **t = b * c;** **a = t / d:**	**cin >> b;** **cin >> c;** **cin >> d;** **t = b / d;** **a = t * c:**

Figure 2. Equivalent programs

In the remainder of the paper, we present linkage graphs, how we might use them in hint generation, and a preliminary study to investigate the feasibility of the approach. All data are anonymized student lab tests from the Spring 2011 introductory programming course at UNC Charlotte. There were about 300 students, but for most of this work we analyzed only those receiving 100% credit. We conclude by discussing how the approach might be combined with other features of novice programs to support the creation of data-driven tutors for novice programmers.

2. RELATED WORK

There have been various approaches to programming tutors. The earliest programming tutor, a LISP tutor, is a cognitive tutor [Anderson1985, Anderson1989]. Cognitive tutors encode domain knowledge and problem-solving strategies in a "cognitive model", an expert system which can solve the problems in the many ways that students solve them. Cognitive tutors have been shown to speed up learning by as much as a factor of three [Anderson1995]. Jin's approach uses cognitive apprenticeship learning to scaffold students in expert problem solving strategies to construct programs [Jin2011]. Lane uses natural language dialog to help students develop pseudo code [Lane2003]. Boyer's approach uses machine learning to develop effective dialog strategies for a Java tutor [Boyer2011a, Boyer2011b]. Online peer review and tutoring is investigated in [Hsiao2011]. Debugging tutors are used to help master programming concepts [Kumar2001, Fernandes2004].

3. LINKAGE GRAPH AS A COMPACT REPRESENTATION OF A PROGRAM

A linkage graph for a program is a hierarchical directed graph, as shown in Figure 3, where nodes represent program statements that modify variables, and directed arcs indicate order dependencies. Control statements such as *for* and *if* are represented by nodes that encompass the entire statement, and can be expanded into their own linkage

graphs, such as those shown in Tables 1 and 2. We call a single trace through the graph a linkage, which connects statements that modify the same variable, and arcs represent variable dependencies.

A linkage graph is a set of linkages intersecting one another. The number of linkages intersecting a node is the number of distinct variables in the node.

Now we introduce the node structure for *if* statements and *for* loops. Similar nodes can be similarly constructed for *while* loops and other control structures. The linkages that go through a node for a programming construct include the linkages for all the variables in the construct, excluding all local variables belonging to the construct. Local variables have their linkages in the sub-linkage graph inside the construct node.

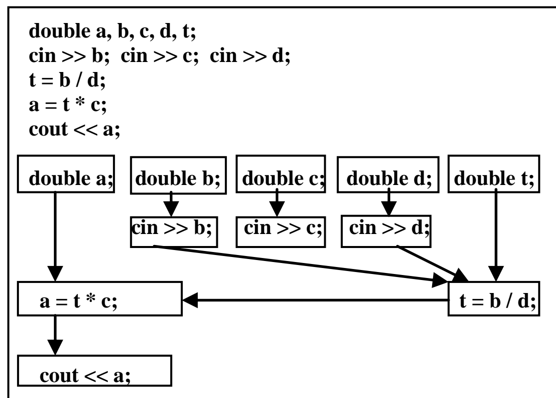
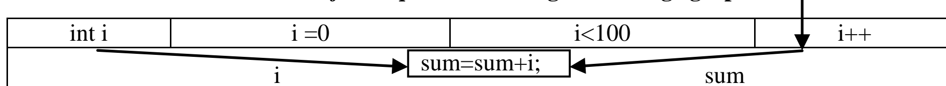


Figure 3. The linkage graph for a program

Table 1. An *if* node

score<0&&score>100	score>=90&&score<=100	score>=80&&score<90	score>=0&&score<80
linkage graph for {grade = 'I';}	linkage graph for {grade = 'A'; }	linkage graph for {grade = 'B';}	linkage graph for {grade = 'F';}

Table 2. A *for-loop* node showing sub-linkage graph



The node for an *if* statement contains specifications for all branches. For each branch, the specification contains the logic expression for that branch and the linkage graph for statements under that branch. The logic expression for each branch should be a complete logic expression with no simplification. Table 1 shows an example *if* node.

A *for-loop* node contains initialization/test/update specifications and the linkage graph for the loop body. An example *for* loop node is given in Table 2.

4. CONDENSED LINKAGE GRAPHS

Even though a linkage graph is a compact and concise representation of a program's data flow and control flow, it is sometimes difficult to determine whether two programs are equivalent directly from linkage graphs. For example, the linkage graph for the program in Figure 4 is different from the one in Figure 3, even though these two programs perform exactly the same set of operations.

To identify equivalent computations, we propose condensed linkage graphs (CLG), which will be used to categorize equivalent linkage graphs. CLG is a linkage graph with all intermediate variables removed. A variable is considered to be an intermediate variable

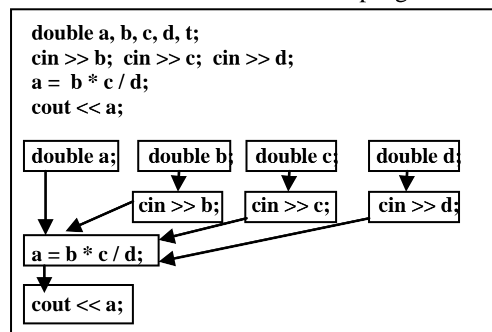


Figure 4. Program with the same operations but different linkage graphs

if it does not store an input value from the user or an output value of the program. To create a CLG, the nodes that reference an intermediate variable are merged. For example, we can obtain the CLG graph in Figure 4 by merging the 3 t-variable nodes in Figure 3.

For programming constructs, such as if statements and loops, the condensation is performed locally. That is, even if a variable is not an input or output variable, if its value is used inside the structure or if it is a loop control variable, we keep this variable in the condensed linkage graph. We may also need to perform between-constructs analysis to determine whether nodes for programming constructs can be merged.

5. LINKAGE GRAPH MATCHING

A linkage graph is a sequence of statement nodes where arcs specify variable dependencies among nodes. For example, Table 3 shows the internal representation of the linkage graphs in Figures 4 and 5. This representation is missing information about parent nodes (i.e. the Preceded By column) to simplify discussion.

Table 3. Implementation of the linkage graphs for programs in Figures 4 & 5.

	Figure 4 Source Linkage		Figure 5 Target Linkage	
Node #	Node	Followed By	Node	Followed By
1	double a;	8	double t;	8
2	double b;	5	double b;	5
3	double c;	6	double c;	6
4	double d;	7	double d;	7
5	cin >> b;	8	cin >> b;	8
6	cin >> c;	8	cin >> c;	Null (Need Hint)
7	cin >> d;	8	cin >> d;	8
8	a = b * c / d;	9	t = b / d;	Null (Need Hint)
9	cout << a;	Null	double a;	Null

Graph matching is based on the following observation: correct program solutions for a problem have the same number and type input variables. So, although students may use different variable names for particular quantities, their linkages will reveal the way each is used. Therefore if we normalize the variable names in programs, we can match one program to another. That is, we have to make sure that the same data item will be labeled with the same variable name across different linkage graphs. Starting with matching input variables and then matching computations, as we describe below, we can compare a work-in-progress program with an existing correct solution and provide hints accordingly based on the differences.

```
double a, b, c, d, t;
cin >> b;  cin >> c;  cin >> d;
t =  b / d;
```

Figure 5. A work-in-progress version of the program in Figure 3

A linkage graph is constructed for the work-in-progress program. It will be compared with a collection of linkage graphs to determine which linkage graph is the best match. If no student linkage graph matches, we can use a condensed linkage graph generated from an expert solution. The condensed graph can then be expanded to match the work-in-progress program.

Figure 5 shows a work-in-progress version of the complete program in Figure 3. Its linkage graph is shown on the right in Table 3. Assume the only linkage graph available for matching, the hint "source", is the condensed linkage graph on the left in Table 3. By comparing these linkages, we can determine what is missing from the target work-in-progress graph to obtain a complete program. The differences can be used to generate hints for students. We can compare each equation in the graph for matching values when we apply associative and commutative rules to get that a=t*c and generate the hint that

the student should “multiply t by c to get a”. Table 4 shows how we expand the source linkage graph (Table 3 (left)) to match the work-in-progress graph (Table 3 (right)).

Table 4. Entry 8 in Table 3 (left) is expanded to match the work-in-progress graph in Table 3 (right).

ID	Node			Followed By
8.	a = b * c / d;	8a	double t;	8b
	➔	8b	**t = b/d;**	8c (Use for hint (“multiply t with c.”))
	a = **b/d** * c;	8c	a = t * c;	9

When generating hints, we will need to use variable names that are meaningful to students, not normalized ones. A preliminary analysis of novice programs shows that students choose similar variable names. We have written an algorithm to match the current work-in-progress program’s variable names to those most commonly used in a data set. In fact, of 300 instances of the first program in our introductory programming class at UNC Charlotte, 97 of them use the variable name houseArea. A simple k-means type algorithm suffices to match a student’s program variables to those most commonly used across a set of solutions to a given programming assignment.

6. EFFECTIVENESS OF CLGS FOR CATEGORIZING PROGRAMS

To evaluate the effectiveness of condensed linkage graphs at categorizing students’ programs, we analyzed student submissions for two homework assignments, one requiring students to write program with basic statements and one with for loops.

For the first assignment, students need to write a program to calculate how much a person earns mowing a lawn. We analyzed 37 programs that were graded as being correct. If we don’t apply the distributive rule to arithmetic expressions (i.e. *a*b+a*c* is considered different from *a*(b+c)*), 25 solutions (68%) have the same condensed linkage graph CLG-A while 12 others (32%) share the same condensed linkage graph CLG-B. If we apply the distributive rules (i.e. *a*b+a*c* is considered the same as *a*(b+c)*), CGL-A and CGL-B are equivalent; so 37 solutions have the same condensed graph.

For the second assignment, students need to write a program to determine the number of wrong answers a person gives in a driver’s license test. We randomly chose 10 correct solutions and found that three of them (30%) belong to one category, three (30%) belong to the second category, while two (20%) are very close to the second category.

7. ANOTHER APPROACH IN REPRENSENTING PROGRAMS

To capture both the control and data flow of the student’s submitted work we can embed the program’s branching into each variable. We can perform a static program analysis and track the declaration, use, and assignment of each variable in the program. The result is a set of regular expressions that describe all the possible values for each of the variables used in the program. We can then normalize the values in this representation to the base inputs. As shown in Table 5, students used *if* statements in different ways for a lab assignment. Student A’s solution represented by Table 5 row 1 is shown in Figure 6.

Table 5: Regular expressions for all possible variable values

SID	Variable Name	Resulting Value
Student A	discountPrice	((cin * .01) || (cin * .03) || (cin *.05)) || 0
Student B	discPrice	(.1 || .03 || .05 || 0) * cin

Evaluation of these variables allows us to check for functionally equal variables and ignore differences in the algorithms students choose. While the algorithms students use are important, this process will aid us by determining what the intended use of variables are in a student's program. It can also help in identifying potential logical errors in a program. If in the above example the user needed to divide *cin* by the decimals rather than multiply Student B’s variable would contain a divide by zero error.

This method of representing variables by the set of all possible values will help us normalize student programs; when combined with other control flow details, it can help build a student program state-space we can use to automatically generate hints. By using the normalized variable values and mapping them to the variable names students are using, perhaps in real-time, we can offer hints that reference elements of an individual student's program.

```
cin >> price;
if (membership == "bronze")
    discountPrice = price * 0.1;
else if (membership == "silver")
    discountPrice = price * 0.3;
else if (membership == "gold")
    discountPrice = price * 0.5;
else
    discountPrice = 0;
```

Figure 6. One possible Student A solution for Table 5 row 1

8. CONCLUSION

Linkage graphs concisely capture a program's data flow and make it possible to discover the fundamental approach a student takes in solving a problem, which provides a good foundation for hint generation using our MDP approach. The clustering of student solutions means that a student's solution is very likely to be the same as that of another student. Therefore, with a collection of previous student solutions to a problem, it is very likely that we can find one that is on the same "route" as the student who is working on the problem and we can use it as a basis for hints generation.

ACKNOWLEDGEMENTS

This work was partially supported by NSF grant IIS-0845997.

REFERENCES

ANDERSON, J.R. AND CONRAD, F. G., AND CORBETT, A. T. 1989. Skill Acquisition and the LISP Tutor. *Cognitive Science* 13(4), 467-505.

ANDERSON, J. R., CORBETT, A. T., KOEDINGER, K. R., AND PELLETIER, R. 1995. Cognitive Tutors: Lessons Learned. *J. of the Learning Sciences*, 4(2), 167-207.

ANDERSON, J.R. AND REISER, B. 1985. The Lisp tutor. *Byte* 10(4), 159-175.

BARNES, T. AND STAMPER, J. 2010. Automatic hint generation for logic proof tutoring using historical data. *Journal of Educational Technology & Society*, 13 (1) - *Special issue on Intelligent Tutoring Systems*, 3-12

BARNES, T. AND STAMPER, J. 2010. Using Markov decision processes for student problem-solving visualization and automatic hint generation. *Handbook on Educational Data Mining*. CRC Press.

BOYER, K. E., HA, E. Y., PHILLIPS, R. AND LESTER, J. C. 2011. The Impact of Task-Oriented Feature Sets on HMMs for Dialogue Modeling. *Proceedings of the 12th Annual SIGdial meeting on Discourse and Dialogue,* Portland, Oregon.

BOYER, K. E., PHILLIPS, R., INGRAM, A., HA, E. Y., WALLIS, M. D., VOUK, M. A. AND LESTER, J. C. 2011. Investigating the Relationship Between Dialogue Structure and Tutoring Effectiveness: A Hidden Markov Modeling Approach. *The International Journal of Artificial Intelligence in Education (IJAIED).*

FERNANDES, E. AND KUMAR, A. N. 2004. *A Tutor on Scope for the Programming Languages Course*. ACM SIGCSE Bulletin, volume 36, issue 1 (March 2004), 90 -93.

HSIAO, I. AND BRUSILVSKY, P. 2011. The Role of Community Feedback in the Student Example Authoring Process: an Evaluation of AnnotEx. *British Journal of Educational Technology*, 42(3), 482 – 499.

JIN, W. AND CORBETT, A. T. 2011. Effectiveness of Cognitive Apprenticeship Learning (CAL) and Cognitive Tutors (CT) for Problem Solving Using Fundamental Programming Concepts. SIGCSE'11: *Proceedings of the 42nd SIGCSE technical symposium on Computer Science Education*, 305-310.

KUMAR, A. N. 2001. Learning the Interaction between Pointers and Scope in C++, *Proceedings of The Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001)*, Canterbury, UK.

LANE, H. AND VANLEHN, K. 2003. Coached Program Planning: Dialogue-Based Support for Novice Program Design. *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, 148-152

STAMPER, J., BARNES T. AND CROY M. 2010. Using a Bayesian knowledge base for hint selection on domain specific problems. *Proceedings of the 2010 Intl. Conf. on Educational Data Mining (EDM 2010)*. Pittsburgh, PA, USA, June 11-13, 2010.

STAMPER, J., BARNES, T. AND CROY, M. 2010. Enhancing the automatic generation of hints with expert seeding. Proceedings of the 2010 Intl. Conf. on Intelligent Tutoring Systems (ITS 2010). Pittsburgh, PA, USA, June 14-18, 2010.