Program Representation for Automatic Hint Generation for a Data-Driven Novice Programming Tutor

Wei Jin[1], Tiffany Barnes[2], John Stamper[3], Michael Eagle[2], Matthew W.Johnson[2], and Lorrie Lehmann[2]

[1]Shaw University, Raleigh, NC, USA
weijin.sz@gmail.com
[2]University of North Carolina at Charlotte, NC, USA
tiffany.barnes@uncc.edu
[3]Carnegie Mellon University, Pittsburgh, PA, USA
john@stamper.org

Abstract. We describe a new technique to represent, classify, and use programs written by novices as a base for automatic hint generation for programming tutors. The proposed linkage graph representation is used to record and reuse student work as a domain model, and we use an overlay comparison to compare in-progress work with complete solutions in a twist on the classic approach to hint generation. Hint annotation is a time consuming component of developing intelligent tutoring systems. Our approach uses educational data mining and machine learning techniques to automate the creation of a domain model and hints from student problem-solving data. We evaluate the approach with a sample of partial and complete, novice programs and show that our algorithms can be used to generate hints over 80 percent of the time. This promising rate shows that the approach has potential to be a source for automatically generated hints for novice programmers.

Keywords: Intelligent tutoring systems, automatic hint generation, programming tutors, educational data mining and data clustering.

1 Introduction

Our goal is to create a data-driven intelligent tutor for computer programming using Markov decision processes (MDPs), created from past student data, to generate contextualized hints for students solving a specific problem. This approach has been applied in the logic domain, providing hints for 70-90% of problem-solving steps [Barnes2010a, Barnes2010b, Stamper2011].

To use the MDP approach, we must describe the student's current solution attempt "target" state that can be compared to existing prior attempts, which are potential hint "sources". Jin, et al. proposed linkage graphs to represent novice program states [Jin2011]. In this paper, we present detailed algorithms for automatic linkage graph extraction from programs and automatic hint generation, and our feasibility study to evaluate the approach.

2 Linkage Graphs to Represent Data Flow and Dependencies

A linkage graph for a program is a directed acyclic graph, as shown in Figure 1, where nodes represent program statements and directed edges indicate order dependencies. If statements *I* and *J* access the same variable *x*, and *J* is the first statement after *I* that accesses variable *x*, then *J* directly depends on *I* and we add an edge from node *I* to node *J* with label *x*. We call a single trace through the graph a *linkage*, which connects statements that modify the same variable. A *linkage graph* is the combined set of linkages. Representation for control statements are discussed in [Jin2011] and we do not implement this aspect of linkage graphs here. In this section we describe our representation and extraction for linkage graphs. We use a 2-dimensional matrix to represent a linkage graph; Table 1 (left) shows the matrix for the program in Figure 1. Variable v_0 shows up in statements 0, 9 and 10, represented as 1's in the corresponding rows, indicating that variable v_0's linkage starts with statement 0 and consists of edges (0,9) and (9,10)..

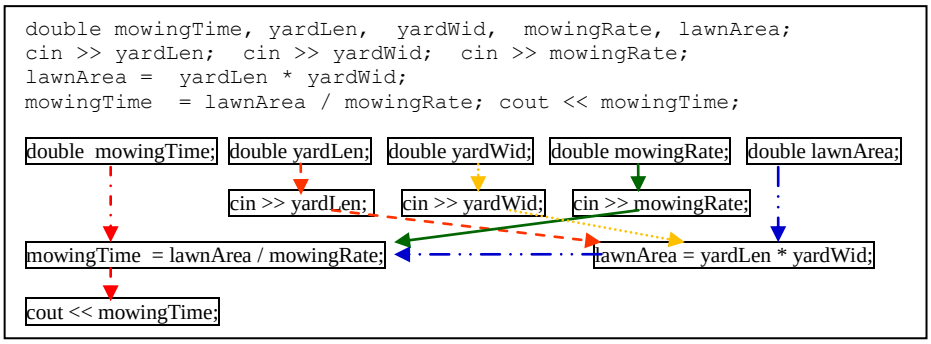

Fig. 1. The linkage graph for a program to calculate money earned for mowing grass. The colored directed edges identify variable dependency between nodes.

Table 1. Linkage Graph Matrices. The left is for the program in Fig. 1; the right is equivalent. (v_0=mowingTime, v_1=yardLen, v_2=yardWidth, v_3=mowingRate, and v_4=lawnArea)

	v_0	v_1	v_2	v_3	v_4
0. double v_0;	1				
1. double v_1;		1			
2. double v_2;			1		
3. double v_3;				1	
4. double v_4;					1
5. cin >> v_1;		1			
6. cin >> v_2;			1		
7. cin >> v_3;				1	
8. v_4 = v_1 * v_2;		1	1		1
9. v_0 = v_4 / v_3;	1			1	1
10. cout << v_0;	1				

	v_0	v_1	v_2	v_3	v_4
0. double v_4;					1
1. double v_3;				1	
2. double v_2;			1		
3. double v_1;		1			
4. double v_0;	1				
5. cin >> v_3;				1	
6. cin >> v_2;			1		
7. cin >> v_1;		1			
8. v_4 = v_1 * v_2;		1	1		1
9. v_0 = v_4 / v_3;	1			1	1
10. cout << v_0;	1				

Table 1 shows matrices for two programs that differ only in order. Since there are no variable dependencies among statements 0-4 and among 5-7, they are equivalent. We note that programs with the same output are not necessarily equivalent. For ex-

ample, $a = b * c / d$ is not equivalent to $t = b * c; a = t / d$. Our goal is that equivalent programs should have the same linkage matrix representation. To accomplish this, we must determine the order of the variables (corresponding to columns of the matrix) and the order of the statements (corresponding to rows of the matrix). The statement order will be determined based on the variable order and the variable dependencies.

Instructor-Provided Specification File: An initial list of variables is taken from an instructor-provided variable specification file for the given programming problem, as shown in Table 2, or could alternatively be generated from the problem description using a bag-of-words approach. Each item specifies a program variable, and consists of three parts: (1) correct data types, (2) phrases that describe the item and may compose the variable name for that item, and (3) how the variable is assigned a value, with the keyword 'input' indicating user-entered values. In Table 2, a slash means "or" – either one of them may be present in the name.

Table 2. A Possible Variable Specification File for the Programming Problem in Figure 1.

Name	Types	Variable Name Terms	Assignment
v_0	float, double	mowing time/hours	$v_1 * v_2 / v_3$
v_1	float, double	yard/lawn length	*Input*
v_2	float, double	yard/lawn width	*Input*
v_3	float, double	mowing rate/speed	*Input*
v_4	float, double	yard/lawn area	$v_1 * v_2$

Assigning variables and extending the variable specification: Meaningful variable names, such as *yardLength or yardLen*, are a common requirement in introductory programming courses. A preliminary analysis of novice programs shows that students choose variable names in this fashion. Second choice names are also common. This suggests that a simple list of all the variable names could be aggregated from all programs and compared to the instructor specification file. For those matching the specification, they are assigned the given variable names. If any remain, we can compute simple similarity and thesaurus lookups to determine if any match to the existing variables or one another. We can cluster the remaining variables and add representative variables to the variable specification.

Variable Normalization: In order to avoid the problem of having a program categorized as different simply because of varying names for variables, we normalize variable names. The variable specification file determines the variable order and normalized names. If a variable name is ambiguous, for example, *length* may refer to yard length or house length, we can use how the variable is used to determine its purpose. If programs are collected in an interactive environment, we could also ask the student which data item the variable refers to.

Statement Sorting: After variables are normalized, the statements will be sorted. Statement sorting consists of three steps. *Step 1 – Preprocessing.* We break a declaration statement for multiple variables into multiple declaration statements, with each declaring only one variable; we do the same for input and output statements. We also break a declaration with initialization into a declaration and an assignment.

Step 2 – Create statements sets according to variable dependencies. The first set consists of statements that do not depend on any other statement. The second set consists of the statements that depend only on those from the first set. The third set con-

sists of the statements that depend only on those from the first and second sets, and so on. For example, for the programs in Table 1, the first set consists of statements *0 – 4*, the second set *5 – 7*, the third set *8*, the fourth set *9*, and the last set *10*.

Step 3 – Within each set, statements are sorted in the decreasing order of their variable signatures. Assume that there are *n* data items in the variable specification file. A statement's variable signature is $s_0 s_1 \ldots s_{n-1}$, where s_i is *1* if the normalized variable name v_i is in the statement and *0* otherwise. For example, the sorted version for the matrix in Table 1 (right) is the matrix shown in Table 1 (left).

Linkage Matrix Representation Uniqueness: The matrix columns are labeled and ordered by normalized variable names. The rows are labeled and ordered by sorted statements. Step 2 guarantees the equivalency of the new program to the original; sorting ensures that equivalent programs have the same matrix representation.

3 Hint Generation for Work-in-Progress Programs

The first step in hint generation for a programming problem is to collect a set of correct solutions from previous students. Then we build linkage graphs for these model solutions. They serve as the sources for hint generation. New solutions may be added to the set. We also build linkage graphs for intermediate states (e.g. program snapshots saved when the compile button is pressed), which are linked by directed arcs that indicate the order the program was written. Each complete program results in a sequence of states illustrating each step in development. These sequences are composed into a single large graph, with equivalent states (linkage graphs) mapped to one another. We then assign a reward value to each state (say 1 point for each linkage) and the correct solutions (say 100), and apply value iteration to create a Markov Decision Process [Barnes2010b]. The linkages act as state features for the states and the reward function computes the state value based its closeness to being complete.

When a student requests a hint, the tutor will build a linkage graph for the partial program. The tutor will find a linkage graph in the MDP that is closest in structure, or a 'match'. When a student's state is matched in the MDP, the MDP allows us to select the next best state by choosing the one with the highest value. We may generate a hint based on the next best state in the MDP or on the final complete solution if the student were to follow the path with the highest values at each step. Suppose that for the partial programs in Table 3, the complete linkage graph as the source for hint generation is Table 1 (left). A partial linkage graph matrix has the same underlying structure as the complete linkage graph: The statements and variables are in the same order as those in the complete graph. The numbers in the matrices (Table 3) represent the order of the statements in the partial programs. We can use missing items or items with wrong orders from the complete graph to generate hints. For example, $v_4 = v_1 * v_2$ is missing from Table 3 (left), so the hint might be "*Calculate* $v_1 * v_2$ *instead of* v_1 / v_2". In Table 3 (right), *cin*>> v_1 and *cin*>> v_2 are after $v_4 = v_1 * v_2$, so the hint might be "*cin*>> v_1 *and cin* >> v_2 *should be before* $v_4 = v_1 * v_2$". Note that when generating hints, we use student variable names (e.g. *yardLen*) instead of normalized names (e.g. v_0 and v_1).

No Matching State Found: Linkage Graph Transformation. If the work-in-progress solution takes a different approach from all existing correct solutions, we

have to determine whether any of the existing complete solutions can be modified to fit the current work-in-progress program. Table 4 shows how we expand the source linkage graph to match the target partial program by adding new rows (and columns) and splitting existing ones as needed. Once this transformation occurs, the new linkage graph can be compared to the partial program to generate hints. This allows us to provide hints right away with a provided expert solution.

Table 3. The Linkage Matrices for the Partial/Incorrect Programs.

… cin >> v_1; cin >> v_2; cin >> v_3; v_4 = v_1 / v_2; // wrong expression	v_0	v_1	v_2	v_3	v_4
0. double v_0;					
1. double v_1;		1			
2. double v_2;			1		
3. double v_3;				1	
4. double v_4;					1
5. cin >> v_1;		2			
6. cin >> v_2;			2		
7. cin >> v_3;				2	
8. v_4 = v_1 * v_2;					
9. v_0 = v_4 / v_3;					
10. cout << v_0;					

… v_4 = v_1 * v_2; // wrong order
cin >> v_1; cin >> v_2; cin >> v_3;
	v_0	v_1	v_2	v_3	v_4

0. double v_0;					
1. double v_1;		1			
2. double v_2;			1		
3. double v_3;				1	
4. double v_4;					1
5. cin >> v_1;		**3**			
6. cin >> v_2;			**3**		
7. cin >> v_3;				2	
8. v_4 = v_1 * v_2;		2	2		
9. v_0 = v_4 / v_3;					
10. cout << v_0;					

Table 4. Transformed Linkage Graph Matrix to Match a Partial Program.

Complete/Correct Program: `... v0 = (v1 * v2) / v3; cout << v0;` **Partial Program that Needs Hints:** `... v4 = v1 * v2;`		v_0	v_1	v_2	v_3	v_4	//column *v_4* added
	0. double v_0;	1					
	1. double v_1;		1				
	2. double v_2;			1			
	3. double v_3;				1		
	3b. double v_4;					*1*	// a row is added
	4. cin >> v_1;		2				
	5. cin >> v_2;			2			
	6. cin >> v_3;				2		
7. v_0=v_1*v_2/v_3;	*7a. v_4 = v_1 * v_2 ;*		*3*	*3*		*2*	// a row broken into 2 rows
	7b. v_0 = v_4 / v_3;	*2*			*3*	*3*	
	8. cout << v_0;	3					

4 Effectiveness of Linkage Graph for Hint Generation

We have implemented the algorithms described herein in the context of jFlex/CUP. To evaluate the effectiveness of using linkage graphs to generate hints, we analyzed student submissions for a lab from the Spring 2011 introductory programming course

at UNC-Charlotte. The program is to calculate the pay for mowing the lawn around a house. There are 200 total submissions with 37 correct solutions.

We performed vertical and horizontal evaluations. The 'vertical' evaluation was applied to the set of correct submissions to generate hints for the first intermediate version that can compile from its later complete counterpart. Since the same student wrote the partial and complete programs, we expected the hints to make sense. This baseline was to confirm that our 'overlay' hint generation approach would work. Among 16 randomly selected correct submissions, good hints were generated for 14 (87.5%) of them, with six correcting a mistake, four finishing one more step, four no hints due to program already complete. The only two cases, where the hints were not appropriate, occurred when variable names were reused for different purposes.

We applied a 'horizontal' evaluation to a sample of 15 randomly selected incorrect submissions. For each of incorrect solutions, we manually selected a similar correct solution, which was not necessarily the best match. We then ran our program to generate linkage graphs and generate hints based on their differences. We found that we could provide meaningful hints for 10 (66.6%) of the incorrect submissions. We believe this rate is promising, since we did not perform a best-match search. With a best-match search and full MDPs, we could leverage partial solutions on paths to correct solutions to provide intermediate states for hints. The remaining 5 programs fall into the following two categories: (1) Variable name reuse. (2) The current algorithm looks at each linkage separately. A hint is provided for the first missing statement along each individual linkage. For example, if a program didn't convert the lawn area from square feet to square yards, the hints will most likely include "*double lawnSqYds*" A better hint should be the next statement along that linkage "*lawnSqYds = lawnSqFt / 9*". This can be addressed by considering the relevant linkages together.

In both cases, implementing our proposed algorithms for detecting variable name reuse would bring the successful hint rates to over 86%. We believe this success rate indicates that our approach is likely to work.

Future Work will include implementing variable reuse detection, linkage graph transformation when a match cannot be found, and further automating the variable normalization process. Finally, we will also determine strategies for hint presentation, since a full list of 'missing' items may be intimidating for novices.

Acknowledgement. This work was partially supported by NSF grants IIS-0845997 and CCLI-0837505.

References.

BARNES, T. AND STAMPER, J. 2010. Automatic hint generation for logic proof tutoring using historical data. *Journal 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.

JIN, W., LEHMANN, L., JOHNSON, M., EAGLE, M., MOSTAFAVI, B., BARNES, T. STAMPER, J. 2011. Towards Automatic Hint Generation for a Data-Driven Novice Programming Tutor. *Workshop on Knowledge Discovery in Educational Data, 17th ACM Conference on Knowledge Discovery and Data Mining*.

STAMPER, J., BARNES, T. CROY, M. Enhancing the automatic generation of hints with expert seeding. To appear in *Intl Journal of AI in Education, Special Issue "Best of ITS"*, 2011. IOS Press.