Automatically Generating Questions in Multiple Variables for Intelligent Tutoring

Tao Li and Sam Sambasivam
Azusa Pacific University, Azusa, CA, USA

tli@apu.edu ssambasivam@apu.edu

Abstract

In our previous research, we investigated the automatic generation of questions with single variable and the application to computer architecture teaching. In the current research, we extend the previous approach to generating questions with multiple variables on Directed Acyclic Graph (DAG) knowledge structures. Questions generated with the new algorithm are more complex and require more mathematical skills to solve. The algorithms can be applied to any discipline for which the conceptual and analytical knowledge can be represented by a DAG.

Keywords: Intelligent tutor, automatic question generation, difficulty assessment, multiple variables, guided problem solving.

Introduction

Research on intelligent tutoring has produced many interesting results, for example Frasson, Gauthier, and Lesgold (1996), Larkin, Chabay, and Sheftic (1990), and Wenger (1987). Web-based online courses are also being developed at many educational institution and much research effort is focused on Web-based courses, for example, Boysen and Van Gorp (1997), Culwin (1998), and Wolz (1993). However, research on automatic question generation and difficulty analysis based on conceptual structures (Sowa, 1984) and ontological engineer (Heflin; 2001; "The Simple HTML", 2000) is still a weak link. Some researchers produced results (Soldatova & Mizoguchi, 2003; Kunichika, Katayama, Hirashima, & Takeuchi, 2003) using approaches not based on conceptual structures.

In a previous paper (Li & Sambasivam, 2003), we presented research results on automatic question generation and difficulty assessment for intelligent tutoring. That approach was successfully applied to the computer architecture course using a quantitative hierarchy. A knowledge structure, the concept graph, based on semantic networks, was used to automatically generate verbal, descriptive questions about computer architecture. That algorithm can only generate a question in a single variable. In addition, we also presented a quantitative method for assessing the difficulty of questions.

In this paper, the focus is on relaxing the restrictions of the previous algorithm. A new algorithm will allow the generation of a question in multiple variables. The knowledge structure for question generation is a Directed Acyclic Graph (DAG) concept graph.

This paper is organized as follows:

- This section presents an introduction to the subject of the paper. It also outlines the contributions of the paper.
- Section 2 is a review of our approach to automatic question generation. Concepts pertinent to the new algorithm are also defined in this section.
- Section 3 presents a new algorithm for automatic question generation in multiple variables.
- Applications and examples using the new algorithm will be presented in section 4.
- A summary with concluding remarks will be given in the last section.

Our approach needs well-formulated concept hierarchies. The development of a meaning hierarchy in a course may take years of experience and much time. Simply following a book or two will not be sufficient. We allow a combination of fixed object composition and dynamic multiple instantiations of a class of objects. Our approach is applicable to a wide range of subjects that do ***not*** require development of new data structures and algorithms. In other words, our approach, in its current stage, is not readily applicable to such courses as data structures and algorithm design.

The contribution of this paper is in its development of a new algorithm for automatic question generation on DAG based knowledge structure with multiple variables. This new algorithm is also guaranteed to generate solvable questions when the system of linear or non-linear equations are solvable.

A Review of Automatic Question Generation

A hierarchical functional concept graph is used in our teaching system. Our functional concept graph is an augmented version of conceptual graphs. A functional concept graph **G** consists of a set of nodes **V** and a set of directed edges **E**. The edges connect the nodes of **V**. In this paper, **G** is always a directed Acyclic Graph (DAG).

The functional concept graph is a hierarchical data structure: each node n_i of the DAG is associated with a level l_i. Node n_i may have a set of incoming edges $\{e_{i,1}, e_{i,2}, \ldots e_{i,k}\}$, connecting lower level nodes. A node without any incoming edge is a *source* node. A node is also associated with a value. The value v_i of a node n_i is computed by a function $v_i = \psi_i(v_{i,1}, v_{i,2}, \ldots, v_{i,k})$, where $v_{i,j}$ is the value of the input node n_j connected by edge $e_{i,j}$. The set of input nodes form the set of partners for node n_i. Associated with each node n_i is also a value interval (l_i,u_i), which is empirically determined by the author of the tutoring system. In automatic question generation, this interval is used to generate meaningful random value for a node. The description of the concept graph here is brief. A more detailed description can be found in Li (1997). There, an augmented concept graph is also defined. The augmented concept graph is what we use in several designs.

The normal direction of computation flow is to compute an output value v_i for node n_i from its input node values using the function $\psi_i(v_{i,1}, v_{i,2}, \ldots, v_{i,k})$. However, we do allow other directions of computation flow. An inverse function ϕ_{ij} is defined, when appropriate, for an input edge $e_{i,j}$ such that $v_{i,j} = \phi_{ij}(e_{i,1}, \ldots, e_{i,j-1}, n_i, e_{i,j+1}, \ldots, e_{i,n})$. This value can be assigned to the corresponding partner node n_j. The existence of inverse functions may bring an extra degree of flexibility and allow the system to generate more sophisticated questions.

It is important to note that our definition of concept graphs allows each function to generate a single output value from a set of inputs regardless of the direction of computation flow.

Question Difficulty Assessment

We define the degree of difficulty as $D = w_1N + w_2P + w_3M$, where N is the number of conditions given in the questions (that is, the number of terminal nodes), P is the number of downward edges in the paths traversed during question generation, and M is the number of upward edges traversed during question generation. We use three weight factors w_1, w_2 and w_3 to balance between the path length and the number of conditions. Typically, $w_1<w_2<w_3$ because upward edges traversed represent more difficult concept association and downward edges, and the total number of edges, which corresponds to the number of problem solving steps involved, carries more information about the effort needed in problem solving.

This definition of degree of difficulty is based on a concrete knowledge structure. Hence, it is more subjective and is a more accurate measure of the effort needed to solve a problem. To our knowledge, this is the first difficulty measurement based on a computation structure and algorithm.

Question Generation Algorithm

For easy understanding and completeness of presentation, we first review the original algorithm for automatic generation of single variable questions. Due to space limitation, we only present the top level of the question generation algorithm. The Generate procedure in the algorithm is not included here. Interested readers are referred to Li and Sambasivam (2003).

Algorithm Question Generation:

1. Randomly choose a question type (either 1 or 2);
2. Randomly select a node nd from the functional concept graph as the destination node at which a result is to be computed;
   Initialize all the node flags to "off";
3. If the question type is 1, call procedure Generate(nd);
   collect the generated conditions and paraphrase the question;
4. If the question type is 2, call procedure Generate(nd);
   collect the generated conditions in set S1;
   call procedure Generate(nd) again;
   collect the generated conditions in set S2;
   paraphrase the question from S1, S2 and the comparison operator;

We are able to show that given the DAG, the node functions and the inverse functions, the above algorithm produces only solvable questions. This is a fundamentally important result, for the users will not waste time to tackle an unsolvable problem and will not suffer from the frustration of not being able to solve a problem after devoting a significant amount of time. A guided learning approach using DAG concept graphs is also found in Li (1997).

Definitions for Multiple Variable Questions

A direct graph is a graph in which the edges have directions. Each edge leads from one node to another. A direct acyclic graph (DAG) is a directed graph that contains no directed cycles.

A node with multiple edges ***coming** from* other nodes is said to have *multiple fan-ins*. If some of the multiple fan-in paths converge in the same node again, these are *paths are reconvergent*. Our algorithm uses multiple fan-in paths to generate questions with multiple variables.

If two paths from node ***z*** lead to node ***x*** and node ***y***, then ***z*** is the common ancestor of ***x*** and ***y***. This definition also includes the case when x is an ancestor of y or y is an ancestor of x. When z is different from x and y, it is called a ***proper*** ancestor of x and y. The concept of proper ancestry is very important in question generation with multiple variables.

Automatic Question Generation in Multiple Variables

In science and engineering application, a question of two or three variables is considered difficult to solve although solving a system of equations with two or three variables in a math course is common. The difficulty arises from understanding the problem concept and setting up the equations. Solving the equations is just one step in the overall exercise. In this section, we consider the automatic generation of questions with two variables.

Generally speaking, two variable questions can be generated using the common ancestor relationship. Specifically, we consider two classes of two variable questions as described below.

- The two nodes n1 and n2 representing the two variables have two or more proper common ancestors ai and aj, and further more neither is ai an ancestor of aj nor aj is an ancestor of ai.
- One node is the ancestor of another.

The two classes of questions need different algorithms to generate and the latter class typically is easier to solve than the former.

The difficulty of a question is given in a variable ***credit***. The amount of credit is distributed among the nodes descending from the common ancestor, including the terminal nodes of the question. The algorithm below generates questions for the case when one node is the ancestor of another.

Algorithm Multi-variable Question Generation1:

1. Select a node x in the DAG graph such that x must have more than one edge leading to other nodes. (That is, node x has multiple fan-outs.) Use the name of node x as one variable.
2. Following one path leading from x to a descendant y that also has descendants. Use the name of node y as another variable.
3. Distribute the total credit to node x and node y proportionally as credit_x and credit_y.
4. Call GenerateOne(y, credit_y). This procedure recursively descends the DAG until the amount of credit is exhausted. It also generates random values within the ranges specified in the terminal nodes. This process is similar to that in Li and Sambasivam (2003).
5. For each descendant w of x
    a. Allocate credit_w from credit_x;
    b. Call GenerateOne(w, credit_w);
6. Append the verbal question strings generated from all the GenerateOne() calls.

The GenerateOne() procedure marks the nodes it visited during its recursive calls. It must also avoid nodes that have been marked. The pseudo code of the GenerateOne procedure is shown below.

Procedure GenerateOne(n, credit):

1. remaining_credit = credit – n.credit;
2. Mark node n;
3. If remaining_cedit <= 0, return;
4. Find the set d = {n1, n2, …, nk} of unmarked descendants of node n;
5. Randomly allocate the remaining_credit to the set of unmarked descendants as {c1, c2, …, ck};
6. For each ni in d, GenerateOne(ni, ci);

For guided problem solving,

- one starts from the terminal descendants of node y and performs computation in a node when all its inputs are available.
- This process proceeds recursively until a value for y is computed.
- Recursively compute values for ancestors of y until a node which is a direct descendant of x.
- Recursively compute values for other descendants of x.
- Compute a value for x from the values of all its descendants.

Proposition1: The algorithm **Multi-variable Question Generation1** always generates two variable questions that are solvable.

The following algorithm generates questions for the case when two nodes have two or more proper common ancestors. The algorithm requires a preprocessing step that finds all the least common ancestors of the nodes in the DAG and tabulates the least common ancestors in a table *Tab*.

Algorithm Multi-variable Question Generation2:

1. Mark all the nodes in the DAG that have multiple fan-ins as “I”.
2. Mark all the nodes in the DAG that have multiple fan-outs as “O”.
3. Select, from the table *Tab*, two nodes x and y that have two proper common ancestors ***a*** and ***b*** where ***a*** and ***b*** are two *least common ancestors* (LCAs) of x and y.
4. Set the name of node x as one variable and the name of node y as another variable.
5. Allocate credits to the nodes x, y, a, b as credit_x, credit_y, credit_a and credit_b.
6. Call GenerateOne(a, credit_a). This call must terminate if x or y or a marked node is encountered.
7. Call GenerateOne(b, credit_b). This call must terminate if x or y or a marked node is encountered.

8. Append the verbal strings from the above four GenerateOne() calls to form the question.

There are many known algorithm for finding all least common ancestors in a DAG, for example Eppstein (1995), Harel and Tarjan (1984) and Schieber and Vishkin (1988). For this second algorithm, the problem generated can be solved with a numeric iterative algorithm. If the algebraic operations involved are complex, it becomes difficult to use guided problem solving. We do not yet have a way to map it to an intuitive GUI design for guided learning.

Guided problem solving is well-developed for the single variable algorithm, as discussed in Li (1997). For the second multiple variable algorithm, however, guided problem solving is not yet clearly defined.

Proposition2. When two least common ancestors are available for two nodes in the DAG, the algorithm **Multi-variable Question Generation2** always generates two variable questions that are solvable.

The above algorithms can be extended to generate questions with three or more variables. However, the extension is a non-trivial task.

Examples will be presented to illustrate the operations of the above algorithms in the next section. These examples will help to clarify some key points of the algorithms.

Applications

Our algorithms can be applied to many subjects in science and engineering, for example, physics, electronics, computer architecture, computer networking, etc.

This approach does not lend itself readily to programming and algorithm courses. Algorithm animation is effective in programming and algorithm design courses.

A combination of algorithm animation and our approach will be effective to a wide range of courses. For example, Operating Systems and Local Area Network (LAN) are two subjects that would benefit significantly from a combination of the two approaches. We are currently developing ontologies and algorithm simulators for the following subjects.

- Local Area Networks (LAN). An online teaching web site is under development for our LAN course. Many algorithm simulators have been developed to assist the learning of protocols and algorithms.
- Operating Systems (OS). We are designing algorithm simulators for OS course. We are also developing an upper ontology for OS.
- Database Design. An upper ontology has been extracted for database principles from Rob and Coronel (2003). We are also developing ontology for query design.
- Computer Architecture. A comprehensive upper ontology has been developed for quantitative computer architecture based on the work of Hennessey and Patterson (1996). Algorithmic simulators are being considered.

A simplified concept graph for computer architecture is shown in Figure 1. We illustrate the working of our algorithms with a few examples.

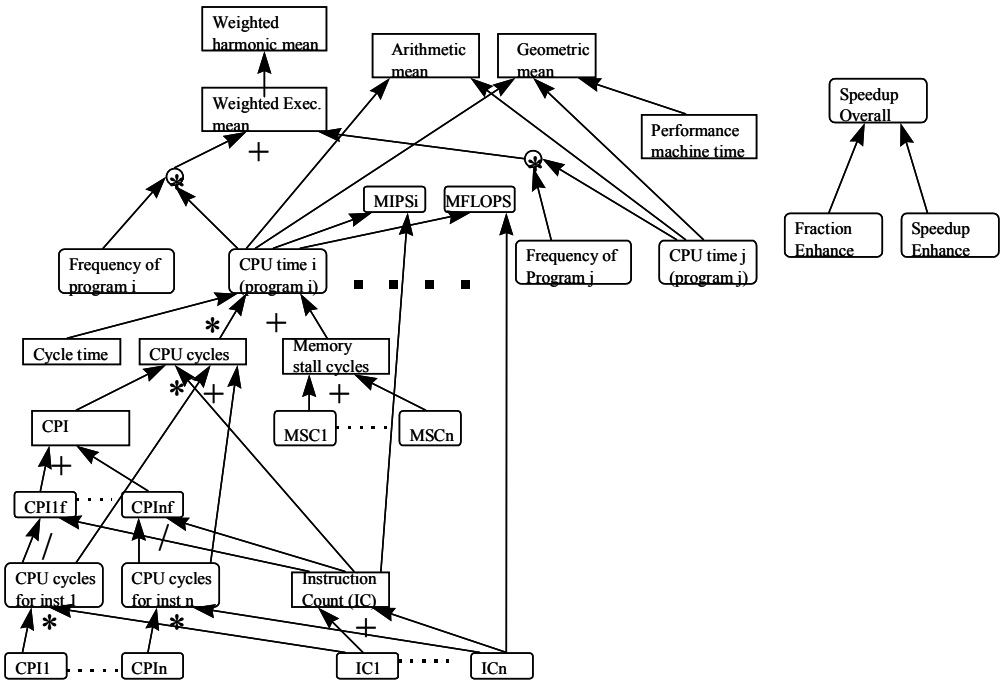

Figure 1. A Simplified Ontology for Computer Architecture.

Example 1. In this example, we will focus on the equation generation part of the algorithm Multi-variable Question Generation**1** and leave out the credit assignment details. Consider the path from (CPU cycles) → (CPU Time i) → (Weighted Exec Mean).

The algorithm may select (*Weighted Exec Mean*) as the first variable and (*CPU Time 1*) as the second variable. A call GenerateOne(CPU_Time_1) assigns (Cycle time) to 10 (ns), CPU cycles to 12,000,000, (Memory stall cyclces) to 75,000 cycles. A second call GenerateOne(Weighted_Exec_Mean) assigns (Frequency of program 1) to 0.8. The algorithm terminates after generating these values.

The algorithm effectively forms the following equations,

- CPU_Time_1 = Cycle_time * (CPU_cycles + Memory_stall_cycles) = 10ns * (12,000,000 + 75,000), and
- Weighted_Exec_Mean = 0.8 * CPU_Time_1.

The first variable (CPU Time i) can be computed from the terminal node values. (Weighted Exec Time) can then be computed from (CPU Time i) and (Frequency of program i = 0.8).

Example 2. Again, we focus on the question generation part and leave out the credit assignment details. Assume that we have two programs Prog1 and Prog2 in a test-bench to estimate the performance of a machine. The two variables nodes are (*CPU cycles 1*) and (*CPU cycles 2*) for Prog1 and Prog2, respectively. The two least common ancestor nodes are the (*Weighted Exec Mean*) and the (*Arithmetic Mean*).

The call GenerateOne(*Arithmetic_Mean*) yields the assignment of *Cycle_Time* = 10ns, *Memory_stall_cycles1* = 4000, *Memory_stall_cycles2* = 7,500, and Arithmetic_Mean = 232,500.

The call GenerateOne(*Weighted_Exec_Mean*) yields the assignment of *Frequency_Prog1*=0.6, *Frequency_Prog2*=0.4,and *Weighted_Exec_Mean* = 218,000.

The algorithm effectively forms the following equations in two variables,

- CPU_cycles1 * Frequency_Prog1 + CPU_cycles2 * Frerquency_Prog2 = Weighted_Exec_Mean, and
- (CPU_cycles1 + CPU_cycles2) / 2 = Arithmetic_Mean.

Let CPU_cycles1 be x_1 and CPU_cycles2 be x_2. Substitute known values into the above, we obtain the following simultaneous equations,

- 0.6x1 + 0.4x2 = 218,000, and
- (x1 + x2) / 2 = 232,500

The variables can be computed by solving the above equations using either substitution or any numerical method.

Summary

In this paper, we extend previous research to automatic question generation with multiple variables. The algorithms are guaranteed to generate questions that are solvable. A credit assignment method is applied to control the complexity of the generated questions.

This approach is being applied to several subjects. In the future, we will develop Web-based courses using a combination of this approach and algorithm simulation. We hope our methods will produce fruitful results for science and engineering education.

For future research, we will focus on provided guided learning and related GUI for questions with multiple variables.

References

Boysen, P. &Van Gorp, M.J. (1997). CLASSNET: Automated support of web classes. *ACM SIGUCCS XXV*.

Culwin, F. (1998) Web hosted assessment – Possibilities and policy. *ITiCSE 98 Conference*, Dublin, Ireland.

Eppstein, D. (1995). Finding common ancestors and disjoint paths in DAGs. *Technical Report 95-52*. Department of Information and Computer Science, University of California at Irvine.

Frasson, C., Gauthier, G. & Lesgold, A. (Eds.). (1996). Intelligent tutoring systems, LNCS-1086. *3rd International Conference on Intelligent Tutoring Systems (ITS'96),* Montreal, Canada, June 1996.

Harel, D. & Tarjan, R.E. (1984). Fast algorithms for finding nearest common ancestors, SIAM J. *Computing, 13* (2), 338-355.

Heflin, J. (2001). Towards the semantic web: Knowledge representation in a dynamic distributed environment. Ph.D. Thesis, University of Maryland, College Park, 2001. Retrieved from http://www.cs.umd.edu/projects/plus/SHOE/pubs/#heflin-thesis

Hennessy, J. L, & Patterson, D.A. (1996). *Computer architecture: A quantitative approach* (2nd ed.). Morgan Kaufmann.

Kunichika, H, Katayama, T, Hirashima, T & Takeuchi, A. (2003). Automated question generation methods for intelligent English learning systems and its evaluation. *Proceedings of ICCE2004,* Dec 2-5, Hong Kong.

Larkin, J., Chabay, R. & Sheftic, C. (Eds.). (1990). *Computer assisted instruction and intelligent tutoring systems: Establishing communication and collaboration.* Erlbaum.

Li, T., (1997). *Question Generation and Guided Problem Solving for Intelligent Tutoring*, Technical Report CUCS97-IT1, Department of Computer Science, Concordia University, Montreal, QC, Canada, 1997.

Li, T. & Sambasivam, S. (2003). Question difficulty assessment in intelligent tutor system for computer architecture. In *Proceedings of ISECON 2003*, San Diego.

Rob, P. & Coronel, C. (2003). *Database systems: Design, implementation and management.* Boston, MA: Course Technology.

Shieber, B. & Vishkin, U. (1988). On finding lowest common ancestors: Simplification and parallelization, SIAM J. *Computing, 17*, 1253-1262,

The Simple HTML Ontology Extensions (SHOE language specification). (2000 April). Available at http://www.cs.umd.edu/projects/plus/SHOE/spec.html

Soldatova, L & Mizoguchi, R. (2003). Test Generation Systems. Proceedings of *SIG-IES-A203-09*, pp51-56, Japan.

Sowa, J. (1984). *Conceptual structures: Information processing in mind and machines*. Reading, MA: Addison-Wesley.

Wenger, E. (1987). *Artificial intelligence and tutoring systems*. Morgan Kaufmann.

Wolz, U. (1993). Providing opportunistic enrichment in customized on-line assistance. *Intelligent User Interface, 93*.

Biographies

Dr. Tao Li is at the Department of Computer Science, Azusa Pacific University, Azusa, CA 91702. Dr. Li graduated from the University of Utah in 1985 with a Ph.D in computer science. He taught at Adelaide University and Monash University in Australia and Concordia University in Canada. He has offered a wide range of computer science courses. Dr. Li has done research in parallel computing, VLSI design, neural networks and data networking. He served on the editorial board of International Journal of Computer-Aided VLSI Design and on organizing committees of international conferences as well as session chairs of international conferences. He was also invited speaker at conferences and various institutions. His research focus is currently on intelligent systems for computer science education and on hardware based systems for networking.

Dr. Samuel Sambasivam is the chairman of the Department of Computer Science of Azusa Pacific University. Professor Sambasivam has done extensive research, publications, and presentations in both computer science and mathematics. His research interests include optimization methods, expert systems, Fuzzy Logic, client/server, Databases, and genetic algorithms. He has taught computer science and mathematics courses for over 24 years. Professor Sambasivam has run the regional Association for Computing Machinery (ACM) Programming Contest for six years. He has developed and introduced several new courses for computer science majors. Professor Sambasivam teaches Database Management Systems, Information Structures and Algorithm Design, Microcomputer Programming with C++, Discrete Structures, Client/Server Applications, Advanced Database Applications, Applied Artificial Intelligence, JAVA and others courses. Professor Sambasivam coordinates the Client/Server Technology emphasis for the Department of Computer Science at Azusa Pacific University.