Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence

Automatically Generating Algebra Problems

Rohit Singh*
MIT CSAIL
Cambridge, MA, USA
rohit_s@mit.edu

Sumit Gulwani
Microsoft Research
Redmond, WA, USA
sumitg@microsoft.com

Sriram Rajamani
Microsoft Research
Bangalore, India
sriram@microsoft.com

Abstract

We propose computer-assisted techniques for helping with pedagogy in Algebra. In particular, given a proof problem p (of the form Left-hand-side-term = Right-hand-side-term), we show how to automatically generate problems that are similar to p. We believe that such a tool can be used by teachers in making examinations where they need to test students on problems similar to what they taught in class, and by students in generating practice problems tailored to their specific needs. Our first insight is that we can generalize p syntactically to a query Q that implicitly represents a set of problems $[[Q]]$ (which includes p). Our second insight is that we can explore the space of problems $[[Q]]$ automatically, use classical results from polynomial identity testing to generate only those problems in $[[Q]]$ that are correct, and then use pruning techniques to generate only unique and interesting problems. Our third insight is that with a small amount of manual tuning on the query Q, the user can interactively guide the computer to generate problems of interest to her. We present the technical details of the above mentioned steps, and also describe a tool where these steps have been implemented. We also present an empirical evaluation on a wide variety of problems from various sub-fields of algebra including polynomials, trigonometry, calculus, determinants etc. Our tool is able to generate a rich corpus of similar problems from each given problem; while some of these similar problems were already present in the textbook, several were new!

1 Introduction

Algebra is a subject where students learn by solving problems. Teachers give homeworks and exams, and ensure that students get enough practice. However, generating fresh problems that involve using the same set of concepts and have the same difficulty level as the problems discussed in the class, is a tedious task for the teacher. Even motivated students want to have access to such fresh *similar* problems, when they fail to solve a given problem and had to look at the solution. Online learning sites such as Khan Academy (Khan) have started providing practice exercises online. However, providing a fixed set of exercises does not provide sufficient personalization for a student who is trying to learn a particular concept. We desire to automatically generate fresh problems that are "similar" to a given problem, where the user interactively works with the system to fine-tune the notion of "similarity".

There have been two approaches to generating similar problems. In one approach, flexibility is provided for instantiating parameters of a problem with random constants (Jurkovic 2001). However, this flexibility is given only for constants. In another approach, certain features of the problem domain are provided as hard-coded options and users are able to choose among these options and generate problems. For instance, in the domain of quadratic equations, some interesting features could be whether the equation is "simple factorable", "difficult factorable, where the leading coefficient is not 1", or "requires use of general quadratic formula". Another interesting feature can be whether or not it has imaginary solutions. Several math worksheet generator websites are based on this approach. The Microsoft Math-Worksheet Generator goes a step ahead and automatically infers such features from a problem instance (Microsoft b). This approach is limited to simpler algebraic domains such as counting, or linear and quadratic equation solving. Also, each domain has its own set of features that needs to be programmed separately.

In this paper, we present a methodology that works for a general class of *proof problems* (hereby, simply referred to as "problems" for brevity) that involve establishing the validity of a given algebraic identity. Our methodology offers two key benefits over above-mentioned existing approaches. First, our methodology is fairly general and is applicable to several sub-fields of Algebra such as Multivariate Polynomials, Trigonometry, Summations over Series, applications of Binomial theorem, Calculus (Limits, Integration and Differentiation), Matrices and Determinants, etc. Second, we are able to involve the user and interactively fine-tune the notion of "similarity" according to the tastes and needs of the user.

Our methodology works in 3 steps:

1. *Query Generation*: Given a problem p, abstract p to a query Q. The query Q implicitly specifies a set of problems denoted by $[[Q]]$ where $p \in [[Q]]$ by default.
2. *Query Execution*: Automatically executing a query Q to

*Work performed during an internship at Microsoft Research.

BinaryOp	::=	$* \mid + \mid - \mid / \mid$ `exp`
UnaryOp	::=	$- \mid$ `square` $\mid$ `ln` $\mid$ `sqrt` $\mid$ Trig $\mid$ InvTrig
Trig	::=	`sin` $\mid$ `cos` $\mid$ `tan` $\mid$ `cot` $\mid$ `sec` $\mid$ `cosec`
InvTrig	::=	$\texttt{sin}^{-1} \mid \texttt{cos}^{-1} \mid \ldots \mid \texttt{cosec}^{-1}$
Constant	::=	$\infty \mid \pi \mid e \mid 0 \mid 1 \mid 2 \mid \ldots$
Variable	::=	*string*
Term	::=	`const`(Constant) $\mid$ `var`(Variable)
		$\mid$ `uop`(UnaryOp, Term)
		$\mid$ `bop`(BinaryOp, Term)
		$\mid$ `diff`(Variable, Term) // Differentiation
		$\mid$ `indefint`(Variable, Term) // Integration
		$\mid$ `defint`(Variable, Term, Term, Term)
		$\mid$ `summation`(Variable, Term, Term, Term)
		$\mid$ `limit`(Variable, Term, Term) // Limit
		$\mid$ `ncr`(Term, Term) // "n" choose "r"
		$\mid$ `matrix`((Term, *int*, *int*) `list`) // Sparse Matrix
		$\mid$ `det`(Term) // Determinant of a Matrix
VarDomain	::=	`TINT`(Variable, *int*, *int*)
		$\mid$ `TREAL`(Variable, Constant, Constant)
Problem	::=	(Term, Term, VarDomain `list`)

Figure 1: Syntax of Algebra Proof Problems.

generate the subset of valid problems from $[[Q]]$. Since Q is a syntactic generalization of the original problem p, only a subset of problems in $[[Q]]$ are valid problems. Query execution rules out generation of invalid problems, trivial problems, and problems that are equivalent to each other (maintaining one representative), and makes use of elegant results from (generalized) polynomial identity testing (Schwartz 1980), combined with several algorithmic techniques to improve efficiency.

3. *Query Tuning*: Allow the user to change query Q to Q' if the set of generated problems in Step 2 is not satisfactory followed by re-execution.

We completely automate the query generation and execution steps, while query tuning is done interactively and manually. We evaluate our technique on several problems from algebra textbooks and measure its effectiveness.

This paper makes the following contributions.

- We present a query language for representing a set of problems that are similar to a given problem (§2).
- We describe an efficient query execution engine that can generate valid problems from the set of problems represented by a given query (§3).
- We propose an interactive methodology wherein queries can be automatically generated from a given problem, and depending on the returned results, be possibly refined by the teacher (§5).
- We present experimental results that illustrate the efficacy of our methodology for problem generation (§6).

2 Query Language

The design of the query language is central to our technique, and embodies several trade-offs we have made. Before we describe the query language, we first describe our term language used to specify problems. A problem (see Figure 1) is

QUnaryOp	::=	UnaryOp $\mid \mathbf{ChoiceU}$(Id, UnaryOp `set`)
QBinaryOp	::=	BinaryOp $\mid \mathbf{ChoiceB}$(Id, BinaryOp `set`)
QTerm	::=	Rules similar to Term
		$\mid \mathbf{ChoiceT}$(Id, Term `set`)
QProblem	::=	(QTerm, QTerm, VarDomain `list`)
QConstraint	::=	ϕ(Id `list`) $\mid$ Id $:= \psi$(Id `list`)
Query	::=	(QProblem, QConstraint `list`)

Figure 2: Syntax of Query Language.

a triple (t_1, t_2, l) comprising of two terms and a list of domain constraints on the free variables appearing in t_1 and t_2. Here, t_1 is the "left-hand-side" (LHS) term and t_2 is the "right-hand-side" (RHS) term. Each domain constraint of the form `TINT` $\mid$ `TREAL`(x, a, b) specifies the range $[a, b]$ and type of values (int or $real$) for the variable x. The goal of the problem is to prove equality between the two sides for all values of the free variables under the domain constraints. (We consider only equalities in the paper, and leave generalizations to deal with inequalities for future work). Terms are recursively defined trees with unary, binary and other operators (e.g., differential, definite and indefinite integrals, sums, limits, matrices, determinants) in the internal nodes of the tree, and with constants and variables as the leaves of the tree. Note that we distinguish between BinaryOp and other operators like `diff` because the two have different evaluation strategies.

A query generalizes a problem to a set of problems. Syntactically, a query (see Figure 2) comprises of a QProblem (which is a pair of QTerms (for "query terms") and a list of variable domain constraints) and a list of QConstraints (for "query constraints"). The tree structure of QTerms is exactly the same as that for Terms except in two ways: (i) Every UnaryOp, BinaryOp and Term nodes are replaced by QUnaryOp, QBinaryOp and QTerm nodes respectively. (ii) QUnaryOp, QBinaryOp and QTerm have special internal nodes (called "choice nodes") identified by an Id and represent a set of unary operators, binary operators and Terms provided as their second argument respectively .

Replacing each choice node in a query Q with an arbitrary element from the set represented by the choice node gives a concrete problem p. The set of all such problems that can be generated from a query Q is denoted by $[[Q]]$. Our aim is to obtain a subset of $[[Q]]$ comprising only of problems that are correct for all values of free variables in the range provided by variable domain constraints and that satisfy the query constraints.

Query constraints are used to specify dependencies between the choices made at various choice nodes in a Query Q. We identify two classes of query constraints:

1. *Relational Constraint:* Comprises of a Boolean function ϕ : (Id `list`) $\mapsto$ *bool* over choice nodes' Ids, ensuring a relationship among the choices made at those choice nodes.
2. *Functional Constraint:* Comprises of an Id (say c_i) and a function ψ : (Id `list`) $\mapsto$ `TypeOf(`c_i`)` over choice nodes' Ids that are different from c_i. This constraint en-

sures that the choice node c_i will always choose the value obtained from the function ψ.

Note that relational constraints are general enough to capture functional constraints but the latter can be used to improve efficiency. Instead of enumerating instantiations and then pruning the instantiations that violate relational constraint, it is more efficient to specify the desired constraint as a functional constraint, and only generate the instantiations desired. Also note that we do not give any syntax for specifying the functional or relational constraints. Any constraint that we can execute (or check) during query instantiation suffices as a function constraint (or relational constraint). The constraints are in terms of the ld's as well as free variables of the problem we are generalizing from. All free variables of the problem that occur in the constraint are assumed to be universally quantified.

We now give various examples to illustrate how we use the query language. Each example first gives the problem we start with, and then shows the query we use to generalize the problem (including the constraints that form part of the query) and then presents the problems we were able to obtain by running our engine on the query. We note that our execution engine uses some results from polynomial identity testing (Schwartz 1980) to generate valid problems, and other techniques (also described later) to generate only distinct problems. The examples motivate not only the design of our query language, but also the optimization techniques presented in the rest of the paper.

Example 1 (Limits/Series)

PROBLEM:

$$\lim_{n\to\infty}\sum_{i=0}^{n}\frac{2i^2+i+1}{5^i}=\frac{5}{2}$$

Note that there are no variable domain constraints as there are no free variables. This problem is a generalization of the famous arithmetico-geometric series usually found in high school textbooks.

QUERY: A natural generalization for this problem is obtained by replacing some constant integer coefficients by a choice node that can take values from a bounded set of integers.

Query Problem:

$$\lim_{n\to\infty}\sum_{i=0}^{n}\frac{C_0i^2+C_1i+C_2}{(C_3)^i}=\frac{C_4}{C_5}$$

where $C_j \equiv \mathbf{ChoiceT}(c_j, \{0,1,2,\ldots,k\})$ for each j and a bounded k.

Query Constraints:

- The constraint $(c_5 \neq 0 \wedge c_4 \neq 0 \wedge \gcd(c_4, c_5) = 1)$ ensures that c_4 and c_5 do not share a common non-trivial factor.
- The constraint $(\gcd(c_0, c_1, c_2) = 1)$ ensures that the coefficients of the LHS do not share a common factor.
- The constraint $(c_0 \neq 0)$ ensures that the difficulty level of the problem is similar to that of the original one (because highest degree monomial term in i is preserved).
- The functional constraint $(c_4 := c_3)$ fixes the same term for c_4 as chosen for c_3 (as was the case in the original problem). Note that we specify this as a functional constraint to achieve efficient instantiation.

RESULTS: Our engine generated 21 similar problems with $k = 7$, some of which are mentioned below:

$$\lim_{n\to\infty}\sum_{i=0}^{n}\frac{3i^2+2i+1}{7^i}=\frac{7}{3}$$

$$\lim_{n\to\infty}\sum_{i=0}^{n}\frac{3i^2+3i+1}{4^i}=4$$

$$\lim_{n\to\infty}\sum_{i=0}^{n}\frac{i^2}{3^i}=\frac{3}{2}$$

$$\lim_{n\to\infty}\sum_{i=0}^{n}\frac{5i^2+3i+3}{6^i}=6$$

Example 2 (Trigonometry)

PROBLEM:

$$\frac{\sin A}{1+\cos A}+\frac{1+\cos A}{\sin A}=2\csc A$$

From (Loney).

QUERY:

Query Problem:

$$\frac{T_0(A)}{C_1 \pm_9 T_2(A)} \pm_8 \frac{C_3 \pm_{10} T_4(A)}{T_5(A)} = C_6T_7(A)$$

where $\pm_u \equiv \mathbf{ChoiceB}(c_u, \{+,-\})$, $T_i(A) \equiv \mathbf{ChoiceT}(t_i, \{\sin A, \cos A, \ldots\})$ comprises of all 6 trigonometric function applications on A, and $C_j \equiv \mathbf{ChoiceT}(c_j, \{0,1,\ldots,k\})$ for a bounded integer k (here $u \in \{8,9,10\}$, $i \in \{0,2,4,5\}$ and $j \in \{1,3,6\}$).

Query Constraints: We use the functional constraints $c_3 := c_1$, $t_4 := t_2$ and $t_5 := t_0$.

RESULTS: Our engine generated 8 similar problems, of which 2 are present in the same textbook, and the remaining 6 are new problems, and it turns out that they all have similar proof strategies. We list some of the them below:

$$\frac{\cos A}{1-\sin A}+\frac{1-\sin A}{\cos A}=2\tan A$$

$$\frac{\cos A}{1+\sin A}+\frac{1+\sin A}{\cos A}=2\sec A$$

$$\frac{\cot A}{1+\csc A}+\frac{1+\csc A}{\cot A}=2\sec A$$

$$\frac{\tan A}{1+\sec A}+\frac{1+\sec A}{\tan A}=2\csc A$$

Example 3 (Determinants)

PROBLEM:

$$\begin{vmatrix} (x+y)^2 & zx & zy \\ zx & (y+z)^2 & xy \\ yz & xy & (z+x)^2 \end{vmatrix} = 2xyz(x+y+z)^3$$

From (Khanna).

QUERY:

Query Problem:

$$\begin{vmatrix} F_0(x,y,z) & F_1(x,y,z) & F_2(x,y,z) \\ F_3(x,y,z) & F_4(x,y,z) & F_5(x,y,z) \\ F_6(x,y,z) & F_7(x,y,z) & F_8(x,y,z) \end{vmatrix} = C_{10}F_9(x,y,z)$$

where F_i $(0 \leq i \leq 8)$ is a $\mathbf{ChoiceT}$ term with $\mathsf{Id} = c_i$ and represents a set of homogeneous second degree polynomials $\{(x+y)^2, (y+z)^2, \pm xy, (s-x)^2, \pm(s-x)(s-y), (x^2 \pm yz), (x^2 \pm xy), \ldots\}$ where $s = \frac{x+y+z}{2}$. We considered all such polynomials and their cyclic rearrangements that were already being used in some problems in the textbooks. C_{10} is a constant choice block (as in the previous examples). The choice node F_9 represents a degree six homogeneous and symmetric polynomial comprising of 3 second degree factors ($\{x^2+y^2+z^2, xy+yz+zx\}$) or 2 third degree factors ($\{xyz, (x+y+z)^3, x^3+y^3+z^3, x^2y+y^2z+z^2x, y^2x+z^2y+x^2z\}$).

Query Constraints: Let c_i be the Id associated with choice term F_i. We add functional constraints to establish cyclicity of the expressions inside the determinant along all the diagonals $c_i := c_j[x \to y, y \to z, z \to x]$ for the pairs $(i,j) \in \{(4,0),(8,4),(5,1),(6,5),(3,2),(7,3)\}$. We also add a relational constraint to avoid generating a new problem if the set of expressions used in the determinant and the RHS are exactly the same as those for some other previously generated problem.

RESULTS: Our engine generated 6 similar problems, of which 3 are already present in the same textbook and the remaining 3 are new. Some of the generated problems are given below:

$$\begin{vmatrix} x^2 & (s-x)^2 & (s-x)^2 \\ (s-y)^2 & y^2 & (s-y)^2 \\ (s-z)^2 & (s-z)^2 & z^2 \end{vmatrix} = 2s^3(s-x)(s-y)(s-z)$$

$$\begin{vmatrix} y^2 & x^2 & (y+x)^2 \\ (z+y)^2 & z^2 & y^2 \\ z^2 & (x+z)^2 & x^2 \end{vmatrix} = 2(xy+yz+zx)^3$$

$$\begin{vmatrix} -xy & yz+y^2 & yz+y^2 \\ zx+z^2 & -yz & zx+z^2 \\ xy+x^2 & xy+x^2 & -zx \end{vmatrix} = xyz(x+y+z)^3$$

$$\begin{vmatrix} yz+y^2 & xy & xy \\ yz & zx+z^2 & yz \\ zx & zx & xy+x^2 \end{vmatrix} = 4x^2y^2z^2$$

Example 4 (Integration)

PROBLEM:

$$\int (\csc x)\,(\csc x - \cot x)\,dx = \csc x - \cot x$$

From (Wiki c).

QUERY:

Query Problem:

$$\int T_0(x)\,(T_1(x) \pm_3 T_2(x))\,dx = T_4(x) \pm_6 T_5(x)$$

where $T_i(x) \equiv \mathbf{ChoiceT}(c_i, \{\sin x, \cos x, \ldots\})$ comprises of all 6 trigonometric function applications on x.

Query Constraints: We add the relational constraints $(c_6 = \text{“}-\text{”}) \Rightarrow (c_4 \neq c_5)$ and $(c_3 = \text{“}-\text{”}) \Rightarrow (c_1 \neq c_2)$ to ensure non-zero LHS and RHS.

RESULTS: Our engine generated 8 problems, of which 4 are present in the tutorial and the remaining 4 are new problems. We list some of the new problems below:

$$\int (\tan x)\,(\cos x + \sec x)\,dx = \sec x - \cos x$$

$$\int (\sec x)\,(\tan x + \sec x)\,dx = \sec x + \cos x$$

$$\int (\cot x)\,(\sin x + \csc x)\,dx = \sin x - \csc x$$

3 Query Execution

Given a query Q, the set $[[Q]]$ can be obtained by systematically enumerating and instantiating all possible resolutions of choice nodes in Q. If there are n types of choice nodes in Q with different identifiers (say c_1 to c_n), and the choice node with identifier c_i can be resolved in m_i ways, then the cardinality of the set $[[Q]]$ is given by $m_1 \times m_2 \times \cdots \times m_n$, assuming there are no constraints specified as part of the query. If there are relational constraints, then instantiations that violate relational constraints are discarded. Functional constraints are used to constrain the instantiations that the enumeration engine generates in the first place, and are hence more efficient than relational constraints.

Several problems in $[[Q]]$ may be incorrect (that is, LHS may not equal RHS on all values of the free variables). We check for this using generalized polynomial identity testing (Schwartz 1980). Let t be a term with free variables X. Let $t[X \leftarrow v]$ denote the term obtained by substituting for the variables in X with the value v. Let $t \Downarrow$ denote the evaluation of a term t with no free variables using standard semantics for all nodes in t.

Theorem 1 ((Gulwani, Korthikanti, and Tiwari 2011)) *Let $p = (t_1, t_2)$ be a problem with free variables X. Let V_X denote the space of values for the free variables. Then, suppose we pick a value v at random from V_X and suppose that $(t_1[X \leftarrow v] \Downarrow) = (t_2[X \leftarrow v] \Downarrow)$. Then, we have that the problem p is correct with high probability over the choice of the random value v.*

The theorem follows from a corresponding theorem in (Gulwani, Korthikanti, and Tiwari 2011) for analytic functions. Our problem grammar (excepting the case of binomial coefficients) maintains differentiability of any order at a point as long as the terms can be evaluated at that point. This makes these functions real analytic. We extend the technique to Binomial coefficients as well by picking random integer

values for free integer variables. For all our experiments we use 4-6 "random evaluations" (evaluations on randomly generated values for each free variable) before announcing a problem as being correct.

For several queries Q, we find that the size of the generated problem set $[[Q]]$ is quite large. Thus, we perform the following optimizations to improve the scalability of our evaluation procedure.

Backtracking. The algorithm constructs a set of choice node Ids for each sub-term location in LHS and RHS of the problem p that are responsible for the evaluation of the sub-term. As long as the selection for those choice nodes remains the same, we do not need to evaluate this sub-term more than once. These values are stored in a map, and at any later stage the algorithm can backtrack the value of this sub-term to avoid re-evaluation. Also, the values of simple function applications (like $\sin, \det$ etc) are stored so as to avoid calling a library function for the same computation with the identical numerical arguments. In many cases where the search space is quite large, the backtracking based algorithm outperforms the naive algorithm and reduces the running time to around a minute from 10-20 minutes.

Approximate computation with thresholds. We desire to efficiently evaluate a term t with free variables X for a randomly generated value v assigned to the free variables. Unlike polynomial identity testing (Schwartz 1980), where evaluation can be done by performing modular arithmetic (modulo a randomly chosen prime), we have general algebraic terms with trigonometric functions, square roots, integrals, derivatives etc. Thus, we use floating point arithmetic with a threshold ϵ_1 for comparing two floating point numbers for equality. We use approximate numerical methods for each specific computation. For evaluating definite integrals we use adaptive quadrature based algorithm with absolute error threshold ϵ_2 (Wiki a). For approximating derivatives, we use 8-point finite difference method (Wiki b) in small intervals (length of each interval being equal to a parameter ϵ_3). To check correctness of indefinite integral problems like $\int f(x)dx = g(x)$, we use two random values v_1 and v_2 for the bounded variable x and check equality (within threshold) of $\int_{v_1}^{v_2} f(x)dx = g(v_1) - g(v_2)$ under a random assignment to other free variables. For evaluating limits tending to finite values, we use sequences approaching the value from both sides and stop when the difference of two values on both sides becomes smaller than a threshold value ϵ_4. To evaluate infinite limits, we use a sequence of values tending to ∞ and stop when two consecutive evaluations are close enough within threshold ϵ_5. Determinants are evaluated using Dense LU decomposition of the matrix (Wiki d).

4 Query Generation

In this section, we describe how to automatically abstract a given problem into a query that is likely to yield similar problems desired by the user. This involves generalization of the problem to get a query problem and also generation of query constraints.

Problem Generalization A problem p can be generalized to a QProblem Qp by systematically replacing patterns of sub-terms with choice operators in both the LHS and RHS. The algorithm generalizes the expression tree at the leaves (constants) and the intermediate nodes representing most unary and binary operations using the following rules:

1. Every occurrence of a unary trigonometric (inverse-trigonometric) operator in p is replaced by a choice node which generalizes it so as to allow any trigonometric (inverse-trigonometric) unary operator.
2. Every occurrence of a constant is replaced with a choice node allowing a list of constants between a lower and an upper bound (parameters controlled by the user).
3. Every add or subtract (binary) operator is replaced by a choice node allowing both operations (i.e. $\pm$).
4. Every occurrence of a homogeneous polynomial in 2 or 3 variables (say inside a determinant) is replaced by a choice node representing a set of homogeneous polynomials in the same variables (For our experiments, we fixed this set by looking at some problems from textbooks).
5. Every occurrence of a particular free variable is replaced by a choice node representing a set of its powers (both negative and positive ones, bounded in the value of the exponent).
6. For summations, multiply by a choice node in front with values 1 or $(-1)^i$ where i is the variable on which the summation is being computed.
7. For summations, replace every occurrence of the iterated variable i by a choice node representing values $i, 2i, 2i+1$ (along with reducing the upper bound by making it half for the two new cases using functional constraints). This allows considering alternate terms of a summation.
8. In presence of radicals, replace every constant by choice node representing bounded integers and all constant radicals (such as $\sqrt{2}$) appearing in the original problem.

All problems in Examples $1, 2, 3$ and 4 in section §2 can be generalized by using rules (2),$(1, 2, 3)$,(4) and $(1, 3)$ respectively. The query execution engine provides the user options to control how much generalization should be done. The size of the search space and the number of problems generated is proportional to the degree of generalization selected by the user.

Query Constraint Generation We generate the following constraints automatically during query generation. The engine provides the user options to remove any of these constraints or to add new constraints.

1. For every occurrence of the same constant or unary function with the same parameter, we add a functional constraint assigning the multiple occurrences to the same constant or parameter to the first one.
2. In case of division of two choice nodes representing integer constants, we add a relational constraint using $\gcd$ (as in Ex. 1) to ensure that they are relatively prime.
3. If the input problem has a matrix with cyclicity property, we add a functional constraint to ensure cyclicity during query instantiation as well.
4. We add two relational constraints saying that the RHS

and LHS are not identically equal to 0.

5. Discard "trivial" or "simpler" problems (obtained by assigning the choice nodes with respective choices) that can be syntactically simplified by canceling/accumulating terms in the case when there are multiple sub-terms being added or multiplied.
6. Ensure that LHS of current problem is not a multiple or inverse of LHS of some previously processed problem.

Note that the first three constraints above are easily checked during query instantiation. The fourth constraint is universally quantified over the free variables of the problem, and is enforced by checking for equality on randomly generated values of the free variables. The fifth and sixth constraints are implemented using some simple symbolic reasoning, and our implementation of the the sixth constraint searches a database of previously generated problems. However, since our query language allows us flexibility to use any function as long as it can be executed, such sophisticated constraints are easily incorporated.

5 Query Tuning

Even though we have provided some strategies for automatic query generation in §4, reaching a satisfactory set of generated problems is quite subjective and automatic generation may not capture insights of the user. The user can reduce the number of problems being generated by adding more constraints to the query. For instance, in Example 1 (§2) the constraint $c_0 \neq 0$ is added to ensure that the degree of the polynomial in the numerator remains 2. If the user wants to see some more problems than the ones generated by the engine, the user can remove some automatically generated constraints or improve the generalization by adding more choices in existing choice nodes or new choice nodes to the query. For instance, in Example 2 (§2), the user may want to generate problems that do not satisfy the automatically generated functional constraint for equal trigonometric operators. In our experience this interactive process of query tuning and execution proceeds for at most a couple of iterations before the engine generates a set of "interesting" problems for the user.

6 Empirical Results

We have implemented our algorithm using the $F\#$ language. The implementation uses XML representations for Terms, Queries and Constraints to interact with the user. The user can provide a problem in an XML file, and our engine generates an intermediate Query as an XML file output. The user can either choose this automatically generated query or modify it before passing it on for execution. Query execution generates new problems based on the query. The user can view the generated problems, and fine-tune the query if the new problems are not interesting enough. The user can also add or remove default constraints using command-line flags and set various parameters like number of random tests to be done, threshold values etc. These I/O XML representations can also be converted to $MathML$ syntax which can potentially enable a web-based GUI for Query tuning. For computation of determinants, derivatives & integrals, we used $F\#$ PowerPack (Microsoft a) and MathNet.Numerics (MathNet) libraries.

We chose our benchmarks from 5 algebra domains (8-13 from each domain). We present our experimental evaluation based on these benchmarks in Table 1. The first column describes type of the query (T-Trigonometry, L-Limits, I-Integral Calculus, B-Binomial Theorem, and D-Determinants). The table contains size of the search space($|[Q]|$), number of Tuning iterations done (n_T), Number of non-trivial correct problems evaluated(n_C), Number of non-trivial and non-equivalent problems generated (n_{Gen}) and Time taken (t in seconds) for Query evaluation. Note that for each row n_{Gen} is the number of problems that were left after the set of correct problems n_C is pruned by the engine using query constraints from Section 4 (so that the user is presented with distinct problems, without repetitions). Our collection of benchmark examples comprises of problems from S.L. Loney's Trigonometry Textbook (Loney), NCERT Math textbook (NCERT), M.L. Khanna's Algebra textbook (Khanna) and internet based tutorials like (Wiki c). We note that, for each of the benchmark examples, we were able to generate similar problems using our approach. In some cases, we had to fine-tune the query, and in all such cases, we did at most one iteration of fine-tuning (see column n_T) to generate interesting problems. We have manually validated every generated problem to ensure that they are indeed correct and interesting problems. All benchmark examples are described in a full version of this paper (Singh, Gulwani, and Rajamani 2012).

7 Related Work

The ever increasing class sizes, thanks to recent online educational initiatives (Khan ; mit), has reinforced the need for use of technology in education. Technology has been used to help both students and teachers with various structured and repetitive tasks in education including solution generation (Gulwani, Korthikanti, and Tiwari 2011) and grading (Singh, Gulwani, and Solar-Lezama 2012). In this paper, we show how yet another aspect of education, namely problem generation, can be automated.

There has been substantial work on query languages for a variety of domains including documents with numerical data (Agrawal and Srikant 2003), structured web data (Spertus and Stein 2000; Mendelzon, Mihaila, and Milo 1996), web-based question answering (Cafarella and Etzioni 2005), or for defining text regions in a document (Miller and Myers 1999). We present a query language for a new domain, namely algebra problem generation. Past work on query execution has been mainly built around appropriate indexing schemes, while our work relies on exhaustive search made efficient by using generalized polynomial identity testing, and various optimizations.

Programming by example (Lieberman 2001; Cypher 1993; Gulwani 2012) is a very popular paradigm for automating end-user programming tasks and has been used in a wide variety of domains including text-editing programs (Lau, Domingos, and Weld 2000), spreadsheet data manipulation (Gulwani, Harris, and Singh 2012; Gulwani

Q	n_T	$\|[Q]\|$	n_C	n_{Gen}	t (seconds)
$T1$	0	21600	6	6	4
$T2$	0	1866240	12	2	167
$T3$	1	7776000	76	4	16
$T4$	0	129600	8	3	50
$T5$	0	259200	6	2	9
$T6$	1	972000	4	2	2
$T7$	0	2592	8	3	1
$T8$	1	373248	96	9	16
$T9$	0	12960	8	3	1
$T10$	0	25920	8	2	2
$T11$	0	10368	6	4	3
$T12$	0	5184	6	3	1
$T13$	0	233280	8	2	95
$L1$	1	179159040	709	139	422
$L2$	0	18750	34	34	1
$L3$	1	6048	6	6	1
$L4$	0	531441	29	29	8
$L5$	0	13176900	971	38	30
$L6$	1	497664	38	38	3
$L7$	1	9216	67	67	2
$L8$	0	6000	21	21	19
$D1$	0	46656	12	12	1
$D2$	1	373248	686	21	12
$D3$	1	46656	18	18	1
$D4$	0	1296	8	8	1
$D5$	0	16384	22	22	1
$D6$	1	216	2	2	1
$D7$	1	675000	40	39	2
$D8$	0	291600	126	3	8
$D9$	1	9920232	16254	4	261
$D10$	1	6718464	540	3	291
$D11$	1	40310784	1701	5	1137
$I1$	0	629856	86	8	17
$I2$	1	3024	10	10	3
$I3$	0	432	10	10	1
$I4$	0	864	6	6	1
$I5$	0	62208	474	15	5
$I6$	1	186624	6	6	4
$I7$	1	5971968	1160	10	244
$I8$	0	62500	52	29	2
$B1$	0	78125	90	32	3
$B2$	0	3125	5	2	1
$B3$	1	3750	4	4	1
$B4$	0	104976	2029	8	5
$B5$	0	729	17	17	1
$B6$	0	36	5	5	1
$B7$	1	390625	29	29	10
$B8$	1	9765625	5	2	1
$B9$	0	180000	489	9	2
$B10$	0	194481	927	20	4

Table 1: Empirical Results for Benchmark Problems

2011; Singh and Gulwani 2012a; 2012b; Harris and Gulwani 2011), repetitive robot programs (Pardowitz, Glaser, and Dillmann 2007), shell scripts (Lau et al. 2004), and imperative Python programs (Lau, Domingos, and Weld 2003). We have applied this by-example paradigm to a novel domain, that of algebra problem generation where the teacher provides an example problem as input and our tool outputs similar problems. Unlike previous work, which is based on version-space algebras (Mitchell 1982; Lau, Domingos, and Weld 2000), our techniques perform brute-force search based on non-trivial principles and optimizations.

Template-based verification (Gulwani, Srivastava, and Venkatesan 2008; Gulwani and Tiwari 2008) and synthesis (Srivastava, Gulwani, and Foster 2010; Taly, Gulwani, and Tiwari 2009; Solar-Lezama 2008; Solar-Lezama et al. 2005) is an upcoming line of work in the areas of Programming Languages and Hybrid Systems. The programmer writes down a template (formulas with holes whose values range over finite or infinite domains) for invariants/program structures and the verifier/synthesizer finds instantiations for these holes such that the overall specification is met. Our work also falls in this category where our queries are like templates and their free variables range over a finite set of choices. Unlike previous work, which is based on use of SAT/SMT solvers to navigate the state space, we use brute-force search based on non-trivial principles and optimizations. Furthermore, we also address the challenging issue of automatically generating an initial query or template.

8 Conclusion and Future Work

There is a resurgence of interest in online learning recently, fueled by popularity of websites such as Khan Academy (Khan). In this paper, we show that a very important aspect of learning, namely the ability to provide fresh practice problems related to the concept which a student is trying to learn, can be automated for the domain of algebraic proof problems. Our automated algorithm has the advantage that it can generate problems that are "similar" in terms of the structure and difficulty, to problems worked out by the teacher in class, or to an exercise the student has tried out, and hence is more relevant to what the student is trying to learn. We have manually verified that for all our benchmark examples, the problems generated by our system are indeed similar to the original problem in the difficulty level and the set of concepts required to solve them. We are now planning to put our system online to also obtain statistical validation of the similarity level of the problems generated by our tool with the original problem. We also want to conduct user studies to measure the usability and utility of our system for both teachers and students.

Acknowledgements

We thank Umair Z. Ahmed for helping with the implementation of the tool reported in the paper, and for providing useful suggestions.

References

Agrawal, R., and Srikant, R. 2003. Searching with numbers. *Knowledge and Data Engineering, IEEE* 15(4):855–870.

Cafarella, M. J., and Etzioni, O. 2005. A search engine for natural language applications. In *World Wide Web (WWW)*, 442–452.

Cypher, A., ed. 1993. *Watch What I Do – Programming by Demonstration*. MIT Press.

Gulwani, S., and Tiwari, A. 2008. Constraint-based approach for analysis of hybrid systems. In *Computer Aided Verification (CAV)*, 190–203.

Gulwani, S.; Harris, W.; and Singh, R. 2012. Spreadsheet data manipulation using examples. *Communications of ACM (CACM)*.

Gulwani, S.; Korthikanti, V. A.; and Tiwari, A. 2011. Synthesizing geometry constructions. In *Programming Language Design and Implementation (PLDI)*, 50–61.

Gulwani, S.; Srivastava, S.; and Venkatesan, R. 2008. Program analysis as constraint solving. In *Programming Language Design and Implementation (PLDI)*, 281–292.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *Principles of Programming Languages (POPL)*.

Gulwani, S. 2012. Synthesis from examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings* 10(2).

Harris, W. R., and Gulwani, S. 2011. Spreadsheet table transformations from examples. In *Programming Language Design and Implementation (PLDI)*.

Jurkovic, N. 2001. Diagnosing and correcting student's misconceptions in an educational computer algebra system. In *Symbolic and Algebraic Computation (ISSAC)*, 195–200.

Khan. Khan Academy. http://www.khanacademy.org/.

Khanna, M. L. *IIT Mathematics*. Jai Prakash Nath Publications.

Lau, T.; Bergman, L.; Castelli, V.; and Oblinger, D. 2004. Programming shell scripts by demonstration. In *Supervisory Control of Learning and Adaptive Systems, AAAI*.

Lau, T. A.; Domingos, P.; and Weld, D. S. 2000. Version space algebra and its application to programming by demonstration. In *Machine Learning (ICML)*, 527–534.

Lau, T. A.; Domingos, P.; and Weld, D. S. 2003. Learning programs from traces using version space algebra. In *Knowledge Capture (K-CAP)*, 36–43.

Lieberman, H. 2001. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann.

Loney, S. L. *Plane Trigonometry*. Cambridge University Press.

MathNet. $F\#$ Math.NET Numerics Library. http://numerics.mathdotnet.com/.

Mendelzon, A. O.; Mihaila, G. A.; and Milo, T. 1996. Querying the world wide web. In *Parallel and Distributed Information Systems*, 80–91.

Microsoft. $F\#$ PowerPack Library. http://fsharppowerpack.codeplex.com/.

Microsoft. Math Worksheet Generator. http://www.educationlabs.com/projects/MathWorksheetGenerator/Pages/default.aspx.

Miller, R. C., and Myers, B. A. 1999. Lightweight structured text processing. In *USENIX Annual Technical Conference*, 131–144.

MITx: MIT's New Online Learning Initiative. http://mitx.mit.edu/.

Mitchell, T. M. 1982. Generalization as search. *Artificial Intelligence* 18(2):203–226.

NCERT. *Mathematics - Grade 11 and Grade 12*. http://ncertbooks.prashanthellina.com/.

Pardowitz, M.; Glaser, B.; and Dillmann, R. 2007. Learning repetitive robot programs from demonstrations using version space algebra. In *International Conference on Robotics and Applications*, 394–399. ACTA Press.

Schwartz, J. T. 1980. Fast probabilistic algorithms for verification of polynomial identities. *Journal of ACM (JACM)* 27(4):701–717.

Singh, R., and Gulwani, S. 2012a. Learning semantic string transformations from examples. *Very Large Databases (VLDB)* 5.

Singh, R., and Gulwani, S. 2012b. Synthesizing number transformations from input-output examples. In *Computer Aided Verification (CAV)*.

Singh, R.; Gulwani, S.; and Rajamani, S. 2012. Automatically generating algebra problems. Technical Report MSR-TR-2012-43.

Singh, R.; Gulwani, S.; and Solar-Lezama, A. 2012. Automated semantic grading of programs. Technical Report arXiv:1204.1751.

Solar-Lezama, A.; Rabbah, R.; Bodík, R.; and Ebcioglu, K. 2005. Programming by sketching for bit-streaming programs. In *Programming Language Design and Implementation (PLDI)*.

Solar-Lezama, A. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation, University of California, Berkeley.

Spertus, E., and Stein, L. A. 2000. Squeal: a structured query language for the web. In *World Wide Web conference on Computer Networks*, 95–103.

Srivastava, S.; Gulwani, S.; and Foster, J. S. 2010. From program verification to program synthesis. In *Principles of Programming Languages (POPL)*, 313–326.

Taly, A.; Gulwani, S.; and Tiwari, A. 2009. Synthesizing switching logic using constraint solving. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 305–319.

Wiki. Adaptive Quadrature. http://en.wikipedia.org/wiki/Adaptive_quadrature.

Wiki. Finite Differencing. http://en.wikipedia.org/wiki/Finite_difference_coefficients.

Wiki. Integration Problems. http://www.math10.com/en/university-math/integrals/2en.html.

Wiki. LU Decomposition. http://en.wikipedia.org/wiki/LU_decomposition#Determinant.