A Constraint-Based System for Automatically Generating Math and Programming Problems

Amy Chou and Justin Kaashoek
Mentor: Rohit Singh

January 3, 2016

Abstract

The ability to automate problem generation to cater to the practice needs of different students presents a problem in large scale courses. In this work we present a constraint-based system to automatically generate programming problems that engage the user in practicing specific concepts. Using constraints, this system can capture the notion of problem difficulty and targeted concepts.. We apply our general technique to generate fill-in-the-blank Python programming problems and math problems in two domains: analytic geometry and calculus.

1 Introduction

The large scale of students in popular courses have forced researches to develop new automated technologies to solve problems such as automated feedback generation [5] and solution generation [2]. Another important problem resulting from this scale is that of automated problem generation to cater to the practice needs of different students as well as for providing different exams to students of a given difficulty level. For example, massive online open courses (MOOCs) with thousands of students may constantly require new problems that are parameterized by complexity and by a set of concepts the students wish to practice.

In this work we present a constraint-based system to automatically generate problems that engage the user in practicing specific concepts. Furthermore, we demonstrate the practicality of this system by generating Python programming problems and math problems in two domains: analytic geometry and calculus derivatives.

There has been some previous work on generating new problems in various domains, namely algebra and programming. The work on generating new algebra problems has mostly been looked upon using two main approaches. In the first approach, a teacher is provided a certain set of parameter values that are fixed for a given domain [3]. For example, for generating a quadratic equation, the parameters can be the number of roots, difficulty of factorization, whether there is an imaginary root, the range of coefficient values etc. Given a set of feature valuations, the tool generates the corresponding quadratic equations. The second approach takes a particular proof problem, and tries to learn a problem template from the problem which is then instantiated with different concrete values [4]. The system first tries to learn a general query from a given proof problem, which is then executed to generate a set of proof problems. Since the query is only a syntactic generalization of the original problem, only a subset of them are valid problems, which are identified using polynomial identity testing.

More recently, a technique was proposed to generate fill-in-the-blank Java problems where certain keywords, variables and control symbols are removed randomly from a correct solution [1]. The technique blanks variables using the condition that at least one occurrence of each variable remains in the scope and blanks control symbols such that at least one occurrence of a paired symbol (such as brackets) remains.

Our techniques for generating math and programming problems, however, are more general since they are constraint-based and can check for more interesting constraints such as unique solutions. Our system can also capture the notion of problem difficulty using complexity functions.

The programming problems we generate take the form of partially complete programs with input/output examples. To solve the problem, the user must correctly fill in the blanks in the partial program based on the input/output constraints. Using this technique, we are able to generate simple Python problems that focus on a set of constructs, including arithmetic, lists and list operations, function calls, and control statements. We are working to extend our ADT grammar to support generation of more complex Python programming problems. We also aim to build a feasible user interface for presentation of this system.

The math problems we generate are specified by a specification class that defines the problem template, compute/correct methods, constraints, and bounds. An instance of this specification class is then passed into a solver. The solver then generates concrete parameters for the problem as constrained by the specification instance. Note that this technique separates the solver from parser, which allows us to try multiple different algorithms at each step. We are able to specify constraints that can generate problems of varying difficulty. Using this technique, we are able to generate standard textbook problems in analytic geometry and calculus derivatives. We are working to make the constraints be specified automatically rather than manually, which will allow the system to be more user-friendly for the general public.

In Section 2, we present our technique for generating Python programming problems. In Section 3, we present our technique for generating math problems. In Section 4, we summarize our work and present directions for further research.

2 Programming Problems

In our system for generating programming problems, users first choose a set of programming constructs that they wish to practice (e.g. control statements, recursion). These constructs are then modeled in Sketch as a subset of Python Abstract Syntax Trees (ASTs) using Abstract Data Types (ADTs). The system then generates a program from the Sketch ADT along with random input/output examples. This complete program can be converted into a partial program by removing pieces and inserting blanks. This partial program with its input/output examples are then presented to the user to solve.

2.1 Overview of Tools and Methods

Our development of an automatic problem generation system is based on preliminary understanding of three methods: parsing of programs using the Python Abstract Syntax Tree (AST) library, use of Abstract Data Types (ADTs), and the Sketch synthesis tool.

2.1.1 Sketch Program Synthesis System

Sketching is a new program synthesis technology that helps programmers write complicated code by self-deriving many details based on constraints. It allows users to leave holes (expressed as double question marks `??` in place of complicated code fragments and is able to synthesize full programs to fill in these holes. Sketch syntax is mostly based on C++, with mainly the addition of the expression `??` to represent holes.

Figure 1 shows an example Sketch program that will synthesize a value of `??` such that `x + x = x * ??`, where `x` is a parameter passed to the function. The output of the program will be the the same statement with `??` replaced by the number `2`.

```
harness void doublesketch(int x){
  int t = x*??;
  assert t == x + x;
}
```

Figure 1: Sample Sketch program

2.1.2 The Python Abstract Syntax Tree (AST) Library

Pythons AST modules processes Python programs as trees in Python abstract syntax grammar. In an abstract syntax tree, each part of the program is broken down into nodes that represent different values such as expressions, variables, integers, booleans, etc. We represent our Python programs as Python ASTs to map and control the grammar of the program. Figure 2 shows an AST of a simple program with three statements.

2.1.3 Sketch Abstract Data Types (ADTs)

Python ASTs are expressed in Sketch as Abstract Data Types (ADTs), which are data structures defined by the user to model behavior of certain operations. ADTs are defined in Sketch with as the type **adt**. Figure 3 gives an example of how Python ASTs are modeled as ADTs in Sketch. Each possible AST node is imperatively implemented as a sketch ADT **expr** object.

2.2 System Overview

Our system can be broken into four concrete steps from user input of constructs to user submission of solution. The steps are as follows:

1. **Translation** of users specification of constructs to Sketch ADT specification that incorporates grammar to test those constructs.
2. **Sketch solving** of the ADT to generate a full Python program (abstractly modelled as a Python AST) along with input/output examples that describe the program.
3. **Abstraction** of the full Python AST into a partially complete AST with blanks to make into an interesting programming problem for the user.
4. **Presentation** of the program to the user formatted as a regular program with blanks.

Figure 4 shows these steps in a flow chart.

2.3 Complete Example

Suppose the user specifies the constructs he or she wishes to practice to be constants, variables, and arithmetic. Figure 5 shows the Sketch ADT that encompasses these constructs and its accompanying interpret statement.

Sketch is able to synthesize a full Python AST from these abstract data type objects using a **harness void** function.

This synthesis is able to generate the short program containing a single expression. Figure 7.1 depicts an abstract syntax tree modeling a generated program, and Figure 7.2 shows the incomplete expression that would then be presented with input/output examples to the user.

3 Math Problems

For generating math problems, we present a technique broken down into four main steps:

1. Translate the user's specification of constructs to a specification class that incorporates a question template and constraints.
2. Generate input/output parameters adhering to problem constraints by passing an instance of the problem specification to a problem solver.
3. Present the problem to user by formatted as the problem template with the generated input parameter values.
4. Check the user's solution to the problem with the generated output values.

With manual specification of each type of problem, we are able to generate simple analytic geometry problems and calculus problems involving derivatives. Examples of problems in these categories include:

- `Find the equation of the line with slope -12 and passing through point (-68, 5).` (Analytic Geometry)
- `Find the equation of the line passing through point (14, 5) and point (-6, -5).` (Analytic Geometry)
- `Find the slope of the line perpendicular to the line 3x + 4y = 5.` (Analytic Geometry)
- `Find the equation of the line perpendicular to the line 3x + 4y - 11 = 0 at point (1, 2).` (Analytic Geometry)

3.1 Problem Specification

The problem specification includes three components of the problem: (i) question template i.e. a string with holes where input values are not yet determined, (ii) constraints, bounds, and correctness methods which check that the input and output parameters adhere to the constraints that the user wishes to incorporate, (iii) input and output parameters.

Currently the user must manually specify the concrete methods in a specification class. We are working to make the process more automated and user-friendly so that users do not need to write

```
public abstract class Specification {
        public int[] input; // paramaters used in the problem question
        public int[] output; // parameters used in the problem answer

        /* Returns a general template for this type of problem.
         * e.g. Template(Ax \in R, Ay \in R,Bx \in R,By in R):
         * "Find the distance between point (" + str(Ax) + ", " + str(Ay) + ")" +
         * " and point (" + str(Bx) + ", " + str(By) + "): ")) */
        public abstract String template();

        /* Allows solver to set output parameters */
        public void setOutput(int[] output){
                this.output = output;
        }

        /* Allows solver to set input parameters */
        public void setInput(int[] input){
                this.input = input;
        }

        /* Returns true if input parameters satisfy user-specified constraints */
        public abstract boolean constraints();

        /* Returns true if output parameters satisfy problem correctness */
        public abstract boolean correct();

        /* Returns bounds for input or output parameters */
        public abstract int[][] bounds();
}
```

Figure 2: Specification Abstract Class

code in order to generate problems of the constraints they wish to apply. Figure 2 shows a general abstract template for all specification classes. Figure 3 shows a concrete specification class for the analytic geometry problem, “Find the equation of the line with slope m and passing through point (a, b).”

3.2 Problem Solver

The problem solver takes an instance of the problem specification class and generates appropriate input and output parameters adhering to the specified constraints, bounds, and correctness. Figure 4 shows the solver that is used to generate input and output parameters for any problem specification.

4 Conclusions and Directions for Future Research

4.1 Programming Problems

Sketch synthesis and abstraction of Python programming problems has the potential to be a quick and easy way to automatically generate many problems for teachers and students. We have pre-

```
public class Problem1 extends Specification {

        public Problem1(){
                super();
                this.input = new int[3];
        }

        public String template(){
                return "Find␣the␣equation␣of␣the␣line␣with␣slope␣" + input[0] + "␣and␣"
                + "passing␣through␣point␣(" + input[1] + ",␣" + input[2] + ")";
        }

        /* Constrains input parameters to be between the bounds -100 and 100.
         * Constrains input parameters so that the point (0, 0) is not on
         * the line. */
        public boolean constraints(){
                boolean b1 = -100 <= input[0] && input[0] <= 100;
                boolean b2 = -100 <= input[1] && input[1] <= 100;
                boolean b3 = -100 <= input[2] && input[2] <= 100;
                boolean b4 = input[0] * input[1] != input[2];
                return b1 & b2 & b3 & b4;
        }

        /* Dictates that output parameters must satisfy correct equations for
         * those of a line.
         */
        public boolean correct(){
                return output[1]*input[1] + output[2]*input[2] == output[0] &&
                                (double) input[0] == -(double) output[1]/output[2];
        }

        /* Returns bounds of (-100, 100) for input parameters */
        public int[][] bounds(){
                int[][] intervals = new int[input.length][2];
                intervals[0] = new int[]{-100, 100};
                intervals[1] = new int[]{-100, 100};
                intervals[2] = new int[]{-100, 100};
                return intervals;
        }
}
```

Figure 3: Concrete User Specification for an Analytic Geometry Problem

```
public class Solver {
        private Specification spec;

        public Solver(Specification spec) {
                // pass in instance of some specification class
                this.spec = spec;
        }

        /* randomly set input parameters for problem adhering to constraints */
        public void setInputParams(){
                int[][] bounds = spec.bounds();
                int[] params = new int[bounds.length];

                while (!spec.constraints()){
                        for (int i = 0; i < bounds.length; i ++){
                                params[i] = (int)  (Math.random()*
                                (bounds[i][1]-bounds[i][0]))+bounds[i][0];
                        }
                        spec.setInput(params);
                }
        }
}
```

Figure 4: Problem Solver

sented an overview of our pipeline and are working to coordinate these parts together to create a scalable system and a feasible user interface.

We currently have tested the different parts of the system independently of each other. We have also been able to generate ASTs for different programming problems, and translate simple ones into Sketch. We have been able to use Sketch separately from the other parts of the pipeline to generate ASTs, and we are currently working on putting the different parts together.

In the future, we will be implementing the UI section of the system, which involves inserting holes into ASTs generated by Sketch, having a method for getting the users to fill in the blanks, determining if the users solution is correct, and providing feedback for what the user has done incorrectly and what they need to work on. We will also be making differents processes more automated, including abstracting with heuristics, generating the Sketch Specification, and generating optimal problems. The optimal problems that we want to automatically generate are not complex enough that the user cannot find a solution, but not so easy that the problem is no longer interesting to solve.

4.2 Math Problems

5 Acknowledgements

The authors, Amy Chou and Justin Kaashoek, would like to thank Mr. Rohit Singh for his invaluable mentorship throughout the year 2015. The authors also thank Dr. Rishabh Singh for introducing related research and for his guidance in the previous year. They further thank Dr.

Armando Solar-Lezama for suggesting the research topic. Lastly, the authors are eternally grateful to MIT PRIMES Computer Science Section; this research would not have happened without the coordination and support of this program.

References

[1] N. Funabiki, Y. Korenaga, T. Nakanishi, and K. Watanabe. An extension of fill-in-the-blank problem function in java programming learning assistant system. In *Humanitarian Technology Conference (R10-HTC), 2013 IEEE Region 10*, pages 85–90, Aug 2013.

[2] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 50–61, 2011.

[3] N. Jurkovic. Diagnosing and correcting student's misconceptions in an educational computer algebra system. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, ISSAC '01, pages 195–200, New York, NY, USA, 2001. ACM.

[4] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[5] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 15–26, 2013.

6 Appendix

Here is where we put all the code we have for other problems (more analytic geometry problems, derivatives problems) + more examples of Python programming problems.

6.1 Problem 1: