Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments

Daniel Perelman
University of Washington
Seattle, WA
{last}@cs.washington.edu

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

Dan Grossman
University of Washington
Seattle, WA
djg@cs.washington.edu

ABSTRACT

With the recent popularity of computer science massive open online courses (MOOCs) and websites for learning programming, there is a need for high-quality automated feedback on introductory computer science assignments. Current courses usually use test cases, which is effective for determining whether is submission is incorrect but not why. Particularly for students new to programming, a counterexample from a failing test suite may be insufficient to guide a student to understanding their error. In a traditional classroom, a teacher may be able to identify where the errors are in a student's code if the student is close to a correct solution. We present a fully automated tool for producing such corrections given only a single reference solution written by the teacher. Additionally, our tool mines correct solutions submitted by students to better handle multiple approaches to a problem. We use this tool to produce hints for the educational programming game Code Hunt.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education—*Computer-assisted instruction (CAI)*

General Terms

Algorithms, Experimentation

Keywords

automated grading, educational games, hint generation, computer science education

1. INTRODUCTION

The recent popularity of computer science massive open online courses (MOOCs) and websites for learning programming presents a major challenge in the area of automated feedback generation on introductory computer science assignments [3]. Due to the scale of the courses, human grading is infeasible while traditional automated techniques (running the submitted program on a fixed test suite) provide very limited feedback. Particularly for students new to programming, a counterexample from a failing test suite may be insufficient to guide a student to understanding their error.

Even when no test cases pass, a human grader is often able to identify the mistakes the student made and therefore provide precise personalized feedback. Similarly, in this work, our system identifies small changes to a student's program which transform it into a correct solution. Given this information, different forms of feedback are possible, but in our present system, the line numbers of the changes are given to the student in order to point them toward the solution without giving it away.

For this purpose, the large scale of MOOCs brings with it an opportunity to match its challenge: due having access to a large number of students and attempts on each assignment, there is a massive amount of data to be mined about what student solutions look like, how they get there, and what mistakes they make. By building an automated technique, the scale helps instead of hurts the quality of feedback: using this data, the only explicit input the teacher needs to give our system is a single reference solution defining the specification for each assignment. Due to the data mining, our tool can correct students working on any solution strategy, including ones the teacher may not have been aware of.

We present a fully automated tool which uses recent developments in program synthesis to provide feedback for student attempts. We use this tool to provide personalized hints in the educational programming game Code Hunt [14] in real-time, which direct players to which lines of their program to focus on.

Outline. Section 2 gives a precise description of the Code Hunt game and our goals for our hint mechanism. Section 3 and Section 4 describe the program synthesis algorithm powering our hint generator while Section 5 describes the hint generator itself. Section 6 includes some initial results. Section 7 discusses related work. Section 8 describes some future work we will do on hint generation. Section 9 concludes.

2. PROBLEM DEFINITION

First we describe the context we are working in, the Code Hunt game, in Section 2.1. Then more precisely articulate what we want out of a hint mechanism in this context in Section 2.2.

2.1 Code Hunt game

Code Hunt [14] is an educational programming game where the player is writing a program in a standard program language (Java or C#) in a simple in-browser programming environment, but the hook is that the player is not told the specification of the program they are writing and discovering it is a puzzle, unlike a homework assignment where a student might be told "implement a sorting function".

In each level of the game, the player starts with an empty

method named "`Puzzle`" with the proper signature (for example, `int Puzzle(int x, int y)` for a secret function that takes two integers and returns an integer). At any time, the player can click the "Capture Code" button. The result of a capture may be

1. a "You win!" message indicating that their program satisfies the secret specification,

2. a compiler error, as would be given in a normal programming environment,

3. or a set of test cases showing inputs on which the player's program agrees and disagrees with the secret specification along with the corresponding outputs of both the player's program and the secret specification.

The last case is the core of the game. Given those test cases, the player then attempts to intuit the pattern to determine the specification and modifies their program according to their theory. This process repeats until the player has both correctly guessed the specification and properly implemented a program for that specification, at which point the game will tell them they have won the level and encourage them to move on to the next level.

Code Hunt has a default level progression targeting the AP computer science curriculum (approximately equivalent to a first semester college computer science course), but also supports allowing teachers to add their own levels.

2.2 Hint generation

The design of our hint mechanism for Code Hunt should be driven by the educational goals of Code Hunt. The goal of any educational game is for the player to learn. If the player stops playing the game, then they are no longer learning from that game. This may seem like an oversimplification, but lacking other metrics like the ability to give players pre- and post-tests, number of levels completed is a good proxy for how much the players learn and is used in [10] for the same reason. One reason a player may stop playing is due to frustration if the game is too difficult. By providing hints before a player gets frustrated and gives up, we hope to increase player engagement. This must be balanced against giving away too much information in hints making the game too easy, in which case the player may stop playing due to boredom.

The problem we target in this paper is to guide players who are near a solution to focus on the parts of their program that need to be changed instead of the majority of their program which is already correct. Specifically, the feedback given will be one or more lines of their program which can be modified to produce a correct solution. This both lets the player know they are on the right track and prevents them from getting distracted away from the correct solution.

The non-trivial part of this problem comes from the fact that there are many solutions to each level. While many solutions may be nearly identical—for instance, differing only by variable names—other solutions may actually be fundamentally different approaches to the problem that the creator of the level never thought of. We want to allow players creativity and not push them toward a hard-coded approved solution. Additionally, a student one typo away from a correct answer may be nowhere near a solution the creator of the level had ever thought of.

In order to accomplish this goal, the hint generator is powered by a program synthesis algorithm which has access to information mined from all players' solutions. At a high level, the algorithm starts from the player's attempt and attempts to solve the level. If successful, it returns which lines of code it changed.

The program synthesis algorithm, Test-Driven Synthesis or TDS, is described in Section 3 and Section 4 while the hint generation algorithm is described in Section 5.

3. TEST-DRIVEN SYNTHESIS

This and the following section are adapted from the original presentation of Test-Driven Synthesis (TDS) by Perelman et al. in [11].

TDS is inspired by test-driven development (TDD) [4, 8] in which a programmer switches between authoring test cases and modifying their program code to satisfy those new test cases. TDS takes as input a sequence of test cases and iteratively modifies the program code to satisfy those test cases. TDS is structured such that TDS is the outer loop which goes over the test cases in order and it uses a subroutine called DBS (for DSL-Based Synthesis) to generate the modification to satisfy that new test case.

The Test-Driven Synthesis methodology synthesizes a program by considering a sequence of examples in order and building up iteratively more complicated programs. Section 3.1 describes the algorithm in detail. Section 3.2 details how the iterative nature of the algorithm works.

3.1 Algorithm

The Test-Driven Synthesis algorithm (`TDS` in Algorithm 1) synthesizes a program P given a sequence of examples S and a set of base components. By "program" we mean a single function with a specified set of input parameters and return type. By "example" we mean a set of input values for those parameters and the correct output value. By "component" we mean any of the set of expressions known to the synthesizer which are used as the building blocks for the synthesized program; the base components are the functions referenced by the DSL but components may also be partially filled-in function calls or larger DSL expressions.

In the spirit of TDD, we build P up, a little at a time, to allow synthesis of larger programs. P_i satisfies the first i examples; its successor program P_{i+1} is built by the DSL-based synthesis (`DBS`) algorithm using the first $i+1$ examples along with information from P_i. The previous program P_i is used in three ways:

1. Its subexpressions are added to the component set.

2. *Contexts* to synthesize in are formed by removing each subexpression of P_i one at a time.

3. The number of branches may only exceed the number of branches in P_i if `failuresInARow` > 0. New conditionals are allowed only after failures in order to avoid overfitting to the examples by creating a separate branch for each one.

EXAMPLE 1 (WALKTHROUGH OF `TDS`). *We will use the DSL* $\boldsymbol{C}$ ::= *`CharAt`*$(\boldsymbol{S}, \boldsymbol{N})|$*`ToUpper`*$(\boldsymbol{C})$, $\boldsymbol{S}$ ::= *`Word`*$(\boldsymbol{S}, \boldsymbol{N})$ $|$*`_PARAM`*, $\boldsymbol{N}$ ::= $0|1$ *where `Word(s, n)` selects the* $\boldsymbol{n}^{th}$ *word from the string `s` and `_PARAM` is any function parameter and e is the set of all grammar rules in that DSL to demonstrate*

Algorithm 1: TDS(S, $\mathcal{L}$, $P_0 \leftarrow \bot$)

input : sequence of examples S, DSL specification $\mathcal{L}$, initial program P_0 (defaults to $\bot$)
output: a program P that satisfies S or FAILURE

```
1  e ← all grammar rules in L;
2  failuresInARow ← 0;
3  foreach i ← 0, ..., |S| − 1 do
4      if P_i(input(S_i))=output(S_i) then
5          P_{i+1} ← P_i;
6          failuresInARow ← 0;
7      else
8          contexts ← ∅, exprs ← e ∪ parameters of P_i;
9          foreach subexpression s of P_i do
10             Add λexpr.P_i[s/expr] to contexts;
11             Add s to exprs;
12         foreach branch B of P_i do
13             foreach subexpression s of B do
14                 Add λexpr.B[s/expr] to contexts;
15         P_{i+1} ← DBS(contexts, (S_0, ..., S_i), exprs, L,
16                       num_branch(P_i)+failuresInARow);
17         if P_{i+1} is TIMEOUT then
18             P_{i+1} ← P_i;
19             failuresInARow ← failuresInARow + 1;
20         else
21             failuresInARow ← 0;
22 if failuresInARow = 0 then
23     return P_{|S|};
24 else
25     return FAILURE;
```

synthesizing the function `f(a)⇒ToUpper(CharAt(Word(a, 1), 0))`*:*

i=0: $S_0 = (a =$ `"Sam Smith"`$, RET =$ `'S'`$)$. `exprs` $= e \cup \{$`a`$\}$ *(the whole language plus the parameter* `a`*) and* `contexts` $= \{\circ\}$ *(the set containing just the trivial context) because the previous program* $P_0 = \bot$*, so there are no subexpressions to remove to build contexts out of. The smallest program to compute* `'S'` *is to select the first character of* `a`*, and therefore* $P_1 =$ `f(a)` $\Rightarrow$ `CharAt`$(a, 0)$.

i=1: $S_1 = (a =$ `"Amy Smith"`$, RET =$ `'S'`$)$. `contexts` $= \{\circ, $`CharAt`$(\circ, 0), $`CharAt`$(a, \circ)\}$*, replacing the two subexpressions of* P_1 *can be removed with holes.* `exprs` *now also contains the expression* `CharAt`$(a, 0)$*. The simplest program consistent with both examples selects the second word of* `a` *instead of* `a` *itself, so* `DBS` *will generate* `Word`$(a, 1)$ *to select the second word and plug it into the second context to generate* $P_2 =$ `f(a)` $\Rightarrow$ `CharAt`$($`Word`$(a, 1), 0)$.

i=2: $S_2 = (a =$ `"jane doe"`$, RET =$ `'D'`$)$. `contexts` $= \{\circ, $`CharAt`$(\circ, 0), $`CharAt`$($`Word`$(\circ, 1), 0), $`CharAt`$($`Word`$(a, \circ), 0)\}$. `exprs` $= e \cup \{a, $`Word`$(a, 1), $`CharAt`$($`Word`$(a, 1))\}$. *Notably,* `CharAt`$(a, 0)$ *does not appear in* `exprs` *despite it appearing in* `exprs` *for the* `i` $= 1$ *step because it is not a subexpression of* P_2*. It is important that such temporary diversions are forgotten so time is not wasted on them in later steps.* `DBS` *will output* $P_3 =$ `f(a)⇒ToUpper(CharAt(Word(a, 1), 0))` *which takes only a single step because it is the application of* `ToUpper` *to* P_2 *which appears in* `exprs`*.*

We now discuss a few details of the algorithm to clarify the description and justify some design choices.

Relation between TDS *and* DBS. DBS is described later in Section 4. We separate TDS from DBS both to show explicitly how the previous program P_i is used when constructing the next program P_{i+1} and to highlight the two key novel ideas in our approach:

1. TDS encapsulates the new idea of treating the examples as a sequence and using that fact to iteratively build up P by way of a series of programs which are correct for a subset of the input space defined by a prefix of the examples.

2. DBS encapsulates the parameterization by a DSL which allows for the flexibility of the algorithm.

TDS runs DBS repeatedly, each time giving it the next example from S along with expressions and contexts from the program synthesized in the previous iteration. In other words, it iteratively synthesizes P_k for each $k \leq |S|$ where P_k is synthesized using $S_0, S_1, \ldots, S_{k-1}$ and the previous program P_{k-1}. In this formulation, P_0 is the empty program $\bot$, or `throw new NotImplementedException();` in C#.

State. As described, the only state kept between iterations is the program P_i and the failure count. DBS does not maintain state, and `contexts` and `exprs` depend only on P_i. Additionally, DBS is passed all of the examples up to S_i, not just S_i. One could imagine a more general problem definition where arbitrary (or at least more) state could be kept between invocations of DBS, but in our experience this tended to be more harmful than helpful: **preserving state essentially corresponds to not forgetting about failed attempts.**

3.2 Contexts and subexpressions

The intuition for the strategy of replacing subexpressions is that the program generated so far is doing the correct computation for some subset of the input space and is overspecialized to that subset. In the example above, after the $i = 0$ step, we had the program that returns the first character of the string instead of the first character of the second word of the string. That program was overspecialized to inputs where the first and second word start with same letter. Selecting the first letter was the right computation but on the wrong input, so filling in the context `CharAt`$(\circ, 0)$ with the right input gave the desired program.

Each context represents a hypothesis about which part of the program is correct and correspondingly that the expression removed is overspecialized. Note that the expression appears in the set of components, so if a small change is sufficient, the effort to build it in previous iterations will not be wasted. Also, one such hypothesis is always that the entire program is wrong and should be replaced entirely.

Contexts are made out of each branch as well as the entire program in order to better support building new conditional

structures (Section 4.2) using parts of one or more of the existing branches.

This theory does not limit contexts to a single hole, but, empirically, doing so keeps the number of contexts manageable and seems to be sufficient in practice. Also, it allows the algorithm to prune away locations based on whether they are reached when executing a failing example: modifications elsewhere could not possibly affect whether such examples are handled correctly. If we allowed multiple modifications, the choice of modification points would have to be changed after any modification affecting control flow.

4. DSL-BASED SYNTHESIS

The DSL-Based Synthesis algorithm (`DBS` in Algorithm 2) is the part of `TDS` that actually generates new programs. `DBS` takes as input a set of examples S, a set of contexts C which generated DSL expressions are plugged into to form synthesized programs matching $\mathcal{L}$, a set of expressions e, a DSL definition $\mathcal{L}$, and a maximum number of branches m. It outputs a new program P' that satisfies all examples in S or `TIMEOUT` if it is unable to do so.

The algorithm is built on five key concepts:

1. New programs are not generated directly; instead expressions are generated and plugged into contexts provided as hints to narrow the search space. This is used by `TDS` to indirectly provide the previous program as a hint (Section 3.2).

2. New expressions are formed from all compositions of expressions according to the DSL $\mathcal{L}$. To produce all smaller expressions before generating larger ones, `DBS` runs as a series of iterations, where, in each iteration, only expressions from previous iterations are composed into new expressions. Section 4.1 discusses generation of new expressions (and important optimizations).

3. A new branching structure will be synthesized if no generated program satisfies all examples in S and $m > 1$. Only programs containing at most m branches will be synthesized in order to avoid over-specializing to the examples. Synthesis of conditionals is discussed in detail in Section 4.2.

4. If the algorithm times out before a solution is found, it will return a special failure value `TIMEOUT`. In `TDS`, this case increments m allowing for more branches in the next run of `DBS`.

5. The search space can be reduced even further using specialized strategies for some functions. We demonstrate this by describing strategies we defined for a couple common loop forms in Section 4.3.

Algorithm 2: `DBS`(C, S, e, $\mathcal{L}$, m)

input : set of contexts C, set of examples S, set of expressions e to build new expressions from, DSL specification $\mathcal{L}$, maximum number of branches m
output: a program P' that satisfies S or `TIMEOUT`

```
/* Try generates one or more programs and if one
   satisfies S, DBS returns it.                  */
Try loop strategies in a separate thread (Section 4.3);
allExprs ← e;
while not timed out do
    foreach c ← C do
        foreach expr ← allExprs do
            Try c(expr);
    Try conditional solutions up to m branches (Section
    4.2);
    allExprs ← generate new expressions (Section 4.1);
return TIMEOUT;
```

4.1 Choosing new expressions

New expressions to use in the contexts are generated by component-based synthesis [5]. In component-based synthesis, a set of components (expressions and methods) are provided as input and iteratively combined to produce expressions in order of increasing size until an expression is generated that matches the specification. In our case, the "specification" is the examples. As opposed to previous component-based synthesis work, the generation of new expressions is guided by a DSL $\mathcal{L}$ and instead of testing the expressions against the specification, they are used to fill in contexts producing larger programs which are then tested.

In our system, all components are expressions marked with which non-terminal in the grammar defined them. Methods are represented as curried functions. The synthesizer generates new expressions by taking one curried function and applying it to an expression marked with the correct non-terminal. Each iteration of the synthesizer does so for every valid combination of previously generated expressions in order to generate programs of increasing size. Representing methods as anonymous functions also simplifies handling methods that themselves take functions as arguments, which are common in higher-order functions like `map` and `fold`.

As the number of components generated after k iterations is exponential with the base being the number of grammar rules (i.e., functions and constants in the DSL) in the worst case, a DSL that is too large will cause `DBS` to run out of time or memory before finding a solution. In practice, around 40–50 grammar rules seems to be the limit for `DBS`, but it depends greatly on the structure of the DSL. An earlier version of `DBS` without the optimizations described below could not handle more than around 20–30 grammar rules. Further optimizations to better prune the search space could possibly allow for even larger DSLs.

Optimizations

Minimizing the number of generated expressions is important for performance. Redundant expressions are eliminated in two ways: the first is syntactic and hence it is fast and always valid, while the second is semantic and valid only when an expression does not take on multiple values in a single execution (e.g., if the program is recursive).

Syntactic. All expressions constructed are rewritten into canonical forms according to the `rewrite` rules in the DSL and duplicates are discarded. For example, `x+y` and `y+x` are written differently but can be rewritten into the same form so one will be discarded. `DBS` will only accept sets of `rewrite` rules which are acyclic (once commutativity and other easily broken cycles are removed) to ensure there is a canonical form. Related to this, constant folding is applied

where possible, so, for example, `2*5` and `5+5` would both be constant folded to `10`, further reducing the search space.

Semantic. The vast majority of the time, an expression takes on only a single value for each example input. In other words, the expression is equivalent to a lookup table from the example being executed to its value on that example. Only expressions with distinct values are interesting, so, for example `x*x` and `2+x` would be considered identical if the only example inputs were $\texttt{x} = 2$ and $\texttt{x} = -1$. This is similar to the redundant expression elimination in version space algebras [6]. The exceptions are if the expression is part of a recursive program or lambda expression, in which case this optimization is not used.

4.2 Conditionals

So far we have not considered synthesizing programs containing conditionals, which are of course necessary for most programs. We consider first synthesizing programs where a single cascading sequence of `if...else if...else` expressions occur at the top-level of the function body, with each branch not containing conditionals. Then the goal is to have as few branches in the one top-level conditional as possible. The problem is to partition the examples into which-branch-handles-them to achieve this goal.

For every program p `DBS` tries, the set of examples it handles correctly is recorded and called $T(p)$. If $T(p) = S$ (all examples handled), p is a correct solution and can be returned. Otherwise, each set of programs Q (where $|Q| \leq m$) whose union of handled examples $\bigcup_{p \in Q} T(p)$ equals S is a candidate for a solution with appropriate conditionals. To be a solution, Q also needs guards that lead examples to a branch that is valid for them; to simplify this, whenever a boolean expression g is generated, the set of examples it returns true for, $B(g)$, is recorded. The sets Q are considered in order of increasing size, so if there are multiple solutions, the one with the fewest branches will be chosen.

If the conditional does not appear at the top-level, then it must appear as the argument to some function. To handle this case, we note that if every branch of the conditional generated as described already happens to contain a call to a function f with different arguments, then it could be rewritten such that the call to f occurs outside of the conditional if that is allowed by the DSL. In that case, we can say that all of the branches match the context $f(\circ)$.

In the algorithm, for each non-terminal the DSL allows for conditionals at, each program p is put into zero or more buckets labeled with the context that non-terminal appears in. For example, if the argument of f may be a conditional and $p = f(f(x))$ then p would be put in the buckets for $f(\circ)$ and $f(f(\circ))$. Then the same algorithm as above is run for each bucket with the conditionals being rewritten to appear inside the context. Inserting multiple conditionals just involves following this logic multiple times.

4.3 Loops

The primary way `DBS` handles loops is to simply not do anything special at all: recursion and calling higher-order functions like `map` and `fold` are handled by the algorithm as described so far. A general `WhileLoop` higher-order function can be used to express arbitrary loops that `DBS` may synthesize like any other DSL-defined function. On the other hand, the use of loops in code often corresponds to patterns in the input/output examples. This section discusses two such common patterns we have written strategies for; experts designing DSLs may additionally define their own strategies for other forms of loops.

Foreach. The "foreach" loop strategy's hypothesis is that there is a 1-to-1 correspondence between an input array and an output array. Assuming that hypothesis, the examples can be split into one example for each element where `i` is the index, `current` is the element at that index, and `acc` is the array of outputs for previous indexes: $(\texttt{in} = \{3,5,4\}, \texttt{RET} = \{9,25,16\})$ would become the examples $(\texttt{in} = \{3,5,4\}, \texttt{i} = 0, \texttt{current} = 3, \texttt{acc} = \{\}, \texttt{RET} = 9)$, $(\texttt{in} = \{3,5,4\}, \texttt{i} = 1, \texttt{current} = 5, \texttt{acc} = \{3\}, \texttt{RET} = 25)$, and $(\texttt{in} = \{3,5,4\}, \texttt{i} = 2, \texttt{current} = 4, \texttt{acc} = \{9,25\}, \texttt{RET} = 16)$. Those examples could be used to synthesize the loop body `current*current` using `TDS`. The strategy includes the boilerplate code to take the loop body `current*current` and output a `foreach` loop over the input array.

While that example captured by a `map`, loop strategies also allow for loops that are not as easily expressed with higher-order functions. For example, the loop bodies examples could also include the values computed so far: $(\texttt{in} = \{5,2,3\}, \texttt{RET} = \{5,7,10\})$ would become $(\texttt{in} = \{5,2,3\}, \texttt{i} = 0, \texttt{current} = 5, \texttt{acc} = \{\}, \texttt{RET} = 5)$, $(\texttt{in} = \{5,2,3\}, \texttt{i} = 1, \texttt{current} = 2, \texttt{acc} = \{5\}, \texttt{RET} = 7)$, and $(\texttt{in} = \{5,2,3\}, \texttt{i} = 2, \texttt{current} = 3, \texttt{acc} = \{5,7\}, \texttt{RET} = 10)$. Then the synthesized loop body would be `acc.Length > 0 ? current + acc.Last() : current`, which could be rewritten into a loop computing the cumulative sum of `in`.

For. Patterns may also show up across examples. For instance, given the examples $(\texttt{in} = 0, \texttt{RET} = 0), (\texttt{in} = 1, \texttt{RET} = 1), (\texttt{in} = 2, \texttt{RET} = 3), (\texttt{in} = 3, \texttt{RET} = 6)$ we can see, looking across examples, that for each input value the result should be the result for the previous input value plus the new input (which is an indirect way of saying "sum the numbers up to `in`"). In terms of loop strategies, the hypothesis is that pairs of examples where the input `in` differ by one correspond to adjacent loop iterations so by combining those pairs we can get examples for the loop body where `i` is current value of the loop iterator and `acc` is the return value of the $\texttt{i} - 1$ iteration: $(\texttt{i} = 1, \texttt{acc} = 0, \texttt{RET} = 1)$, $(\texttt{i} = 2, \texttt{acc} = 1, \texttt{RET} = 3)$, and $(\texttt{i} = 3, \texttt{acc} = 3, \texttt{RET} = 6)$. Then `TDS` will give `i + acc` for the loop body. The loop strategy will identify that $(\texttt{in} = 0, \texttt{RET} = 0)$ indicates that the loop iterator should start at 0 and the accumulator should start at 0 and produce a `for` loop `for(int i = 1; i <= in; i++) acc = i + acc;`.

Other strategies. Different loop strategies can give different information like including the index in a `foreach` or giving `acc` corresponding to going in reverse order. Furthermore, the concept of spliting up arrays by element to find patterns can also be applied to splitting strings (by length or delimiters), XML nodes, or whatever other structured data may be in the target domain.

5. HINT GENERATOR

The hint generation algorithm (Algorithm 3) is based off the Test-Driven Synthesis (TDS) program synthesizer [11] described above. While TDS was developed for end-user

Algorithm 3: `GenerateHint(`P`,` R`)`

input : player incorrect program P, reference solution R
output: a hint (line numbers to change) or `FAILURE`

$\mathcal{L} \leftarrow$ expressions commonly used by players to solve R;
$S \leftarrow$ test cases commonly given to players for R;
$P' \leftarrow \texttt{TDS}(S, \mathcal{L}, P_0 \leftarrow P)$, giving up after 30 seconds;
if *P'=FAILURE* **then**
  return `FAILURE`;
return list of lines changed in `ASTDiff(`P`,` P'`)`;

programming-by-example (PBE [2, 7]); in this work we adapt it for the educational setting.

Two novel features of TDS make it appropriate for this use.

1. TDS takes an iterative approach to synthesizing programs, where at each intermediate step a program is generated that satisfies some subset of the test cases. We take advantage of this design to insert the player's attempt as a program for the algorithm to build upon.

2. Unlike other program synthesis technologies that work across multiple domains, TDS does not rely on an SMT solver or similar technology [13, 16]. This allows for the flexibility to work in any domain without worrying about support for that domain from the underlying solver.

Additionally, the support for synthesizing loops and conditionals makes it flexible in the control flow structures of programs it can support. Just a few loop synthesis strategies can cover many of the loops that appear in simple programming assignments.

Initial program. In TDS as used for end-user programming-by-example, the initial program P_0 is $\perp$, the empty program that fails on all inputs. For hint generation, the initial program is instead the player's attempt.

Test cases. TDS depends on test cases to determine the correct program. It could get them from the test cases shown to the player, but the hint should direct the player toward the actual correct solution, so those test cases may be insufficient. They could be augmented by querying Code Hunt just like what happens when the player clicks the "Capture Code" button, but, in practice, this is much too slow as generating counterexamples takes several seconds, so it cannot appear in the inner loop of the hint generator which has to be able to display a hint to the player within at most several seconds. In practice, all test cases that have been shown to all players as counterexamples are recorded. Using more test cases slows down TDS, so initially only the 10 test cases most commonly shown to players are used, but if more are needed, then it continues down the list.

Datamining solutions. TDS requires a domain-specific language (DSL) as an input. This DSL guides the search by limiting the set of programs to search through to those expressible in the DSL. Limiting the search space is important because the search space of all programs using all constructs appearing in all Code Hunt levels is too large to search through quickly. Therefore, for each levels, we mine player solutions for what expressions they used. Expressions that appear in at least 10 other players' solutions for a given levels are considered by the hint generator. The cut-off is due to the observation that there is a large long-tail of useless expressions in the set of all expressions appearing any player's solution.

This is a relatively shallow mining of the data collected by Code Hunt, but it is sufficient to produce the DSL needed to run TDS efficiently without per-level human effort.

Selecting the best hint. Although the process has been described as generating a single hint, in reality, if there is one small change to correct the program, the synthesizer often finds several more soon after. Therefore, the synthesizer is not stopped immediately after finding the first solution. This gives an opportunity to rank the hints and select the best one to show to the student. As our goal was to present the smallest change to the student, we rank the hints by textual edit distance: changing a single character on one line is better than rewriting an expression on another.

6. EVALUATION

We have recently deployed this system on the educational programming game Code Hunt [14]. The system runs on Microsoft's Windows Azure cloud computing platform in order to be able to scale to generating hundreds of hints simultaneously. Each invocation of the hint generator is given access to 4 cores and 7 GB of memory and allowed to run for 30 seconds before it is considered to have timed out (in which case no hint is generated).

In order to evaluate the system on its ability to generate hints without confusing the data with attempts players made to determine the specification, we ran the system treating the test cases the player had seen as the entire specification. In this setting, the system generated hints for 65% of the attempts. That is, 65% of the time, the system was able to determine which lines the player would have to change to satisfy the test cases they had already seen.

7. RELATED WORK

First, it should be noted that Code Hunt itself (and the older Pex4Fun game [15] it is based on) is a form of automated personalized feedback for computer science education: it generates test cases specialized to the code the player submits in order to always provide a counterexample if the player has not actually solved the level. Other automated feedback tends to use a fixed set of test cases and therefore can be tricked by attempts that happen to satisfy those test cases, which would be a problem for a game like Code Hunt where the player can make an unlimited number of attempts. Test cases alone are not always sufficient feedback, and they are a poor measure of how close to a solution a player actually is.

AutoGrader [12] is the most similar work on automated personalized feedback for computer science education. The primary difference is tha AutoGrader depends on a hand-written error model being provided for each assignment. In our system, the equivalent to AutoGrader's error model is mined from the solutions provided by other players and therefore our system is fully automated, unlike AutoGrader.

The Codewebs project [9] generates feedback by clustering

solutions to a programming problem by edit distance in the space of abstract syntax trees (ASTs). Given their data, they determine which subtrees are highly correlated with incorrect solutions representing common mistakes. They demonstrate this does a better job of detecting common mistakes than test cases directed at those mistakes.

Alur et al. restrict themselves to the easier problem of automated personalized feedback for finite automata [1]. Unlike general programs, many properties of finite automata are computable, but it turns out generating meaningful feedback is difficult. Their work is based on modeling a few varieties of student errors in order to determine the best feedback to give; one of their models is to search for small changes to the student's program to generate a hint, similar to our system.

8. FUTURE WORK

We intend to do A/B testing in order to determine if hints in fact increase time spent playing Code Hunt.

The feedback given to the player is currently just the line numbers where they need to make changes. While the hints are intentionally information-poor to avoid giving away the answer, especially on smaller levels (some of which have single line solutions), this is insufficient to help the player. Ideas for other forms of feedback which may be appropriate may include telling the player what constructs (e.g., "plus" or "`for`") or family of constructs (e.g., arithmetic operators or loops) they should consider adding to their program or semi-automatically categorizing an error as matching a common class of errors like off-by-one errors.

While this paper discusses hint generation for an educational game, the same technology may be applicable to other educational settings. We intend to investigate using the technology for grading in large classes and standardized tests.

Our approach to hint generation focuses on the problem of directing a player toward a solution they are close to. This has a notable limitation of only providing hints when the player is indeed close to a solution. While important, for a more complete hint mechanism, we also intend to address the problem of how to direct a player away from an incorrect path, or, similarly, toward a solution they are far away from. One possible approach would be to identify features that appear in no correct solution that appear in the player's attempt and warn the player away from those features.

9. CONCLUSION

We have presented a completely automated approach to hint generation for introductory computer science education. We have deployed a system using this approach to a publicly available educational game. Further, we believe this technology can be useful in other educational settings including standardized testing and grading for traditional classrooms and MOOCs.

10. REFERENCES

[1] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 1976–1982. AAAI Press, 2013.

[2] A. Cypher, D. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. Myers, and A. Turransky. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.

[3] P. Hyman. In the year of disruptive education. *Communications of the ACM*, 55(12):20–22, 2012.

[4] D. Janzen and H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9), sept. 2005.

[5] S. Katayama. Systematic search for lambda expressions. TFP, 2005.

[6] T. Lau, S. Wolfman, P. Domingos, and D. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1), 2003.

[7] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[8] R. Martin. The transformation priority premise. `http://blog.8thlight.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html`, 2010.

[9] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International World Wide Web Conference (WWW 2014)*, 2014.

[10] E. O'Rourke, C. Ballweber, and Z. Popovií. Hint systems may negatively impact performance in educational games. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 51–60. ACM, 2014.

[11] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2014.

[12] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26. ACM, 2013.

[13] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[14] N. Tillmann, J. Bishop, R. N. Horspool, D. Perelman, and T. Xie. Code hunt: Searching for secret code for fun. In *SBST*, 2014.

[15] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Pex4Fun: Teaching and learning computer science via social gaming. CSEE&T, 2012.

[16] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152. ACM, 2013.