Automated Feedback and Recognition through Data Mining in Code Hunt

Daniel Perelman
University of Washington
perelman@cs.washington.edu

Judith Bishop
Microsoft Research
jbishop@microsoft.com

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

Learning to code has become so popular that it is now almost the default that beginners will encounter coding on a website, along with thousands of others at the same time. Providing feedback and recognition in the face of such increasing numbers is a challenge that can be met by the automated test generation. Through automation and access to massive amounts of data, we show that the frequency, coverage, accuracy and personalization of feedback can be improved over earlier systems. Recognition can also be made automatic by using a gaming model. Based on the Code Hunt programming Game, we have developed and tested a system of test-driven synthesis (TDS) and produced results that show that we can accurately produce sensible feedback. Moreover the feedback increases engagement in continuing with the difficult task of learning to code. We also report on the effect of recognition of progress during the game and during contests.

1. Introduction

Everyone thrives on feedback and craves recognition. They are the cornerstone of academic life and, one can say, of scientific progress. Researchers are very familiar with peer review as a means to measuring the validity of their work and improving it. Sometimes, peer review is taken to an additional level and papers or the researchers themselves are judged to be above others and given awards. Feedback and recognition are arguably even more important for those who are learning a skill. Without feedback, they can remain stuck at a certain place, become discouraged and give up. Without recognition, they do not have a yardstick by which to measure their progress and do not know where they stand in relation to others. However, feedback is only meaningful if it is accurate, and recognition is most valuable if it is in relation to a significant number of peers. Traditional subjective feedback and recognition, based on the vagaries of humans, have always had the potential to lead to confusion and unhappiness. Is it possibly to become more objective, and therefore more accurate? Our research shows that within a certain domain, and by employing data at scale, that it is.

Our domain is that of learning to code. In the past few years, coding has moved front and center in the public consciousness [ref]. Introductory programming courses at universities are now but a small part of what has become a national movement in countries such as the USA and the UK {ref to csedweek and Make it Digital} . With so many students from a young age being encouraged to code, there has been an explosion of websites and apps for learning programming. For example, {Code.org} lists over 20 of these. Some of online programming sites are stand-alone and intended for drill and practice; some are accompanied by content, constituting tutoring systems; and some are embedded in massive open online course (MOOCs), or can be. All of them share a common factor - huge numbers of adopters. Over two years, most of the sites showcased on code.org have attracted over a million learners, and the top one over 27 million. The challenge is how to provide feedback while the student is developing a program,

and how to recognize achievement afterwards. Even if grading as such is not required or appropriate, commercial sites are interested in providing recognition in terms of stars and certificates in order to entice users to complete the tutorial or course and to come back for a follow-up if there is one. And everyone is interested in ensuring that the whole experiment in online programming does not backfire because students—and teachers—find it just too hard.

Our goal is to provide both feedback and recognition, defined as:

- feedback: information about a next step that can be taken to make progress towards a solution, known as a hint;
- recognition: information about the quality of the solution, known as a score.

To achieve accuracy, we work at scale, harnessing the hundreds of thousands of users of our online programming game, Code Hunt. Thus the hints and scores we return are not based on a match with a single human's idea of what is correct, but on that of many.

1.1. Automated Feedback

The critical problem is that the scale of online programming systems and MOOCs can run into thousands of simultaneous users making human assistance infeasible. Automation is the key. However, the field of automated feedback generation brings its own challenges [6]. For decades the traditional automated technique for program feedback has been to run the submitted program against a fixed test suite, producing results that indicate for which tests the program passed or failed. There is a survey of approaches in [5] and a compendium of recent tools in [7]. 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 contrast, 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.

Recently, there have been some promising results produced using symbolic execution and modeling [16] which produce specific instructions as to changes that a student should make to bring the program in line semantically with a reference implementation. Though ground breaking in many respects, a limitation of the approach is that it relies on a single reference implementation, and works with the small, neatly specified programs found in introductory programming courses. The feedback is also proscriptive, leaving little room for the student to learn as to why the hint instructions given should be applied. set our horizons wider.

TODO: I would like to show three examples in parallel of the hint levels

The system we shall describe identifies small changes to a submitted program which will transform it into a solution. We move a level above specific changes in the code and talk about features, indicating what kind of change would move the program towards a correct solution. These hints are phrased as "You might find feature x useful here" or "Feature y is rarely used in this solution". We can make such seemingly sweeping statements because we are basing our feedback not on a one-to-one comparison between the submitted code and a single reference solution, but on potentially 100,000s of solutions. The large scale of online programming brings with it an opportunity to match its own challenge: with access to an increasingly large number of students and their attempts on each program, there is a massive amount of data to be mined. The data tells us what the student solutions look like, how they arrive at them, and what mistakes they make. With 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 a specification for each problem. Due to the data mining, our tool can correct students working on any solution strategy, including ones the teacher might not have chosen as the reference, or may not even have been aware of.

1.2. Automated Recognition

It has become increasingly uncommon for students to be told how they are performing in relation to their peers within a formal class environment. However, within the gaming community, competition drives success. It is possible to learn from this model and to inject into a online programming tool elements of gaming such as scores, a leader board and contests so that students have the opportunity to gauge their progress. For top performers, the recognition that they seek is there.

Even if showing relative performance is deemed unwise, immediate recognition that a level has been obtained is a strong motivator for a learner and can lead to continuing perseverance at the task of learning to program.

1.3. Paper Outline

The base system on which we study automated feedback and recognition is the programming game Code Hunt, from Microsoft Research. Code Hunt is a white box testing system based on Pex [19], with a gaming front end, as described in Section 2. To it, we added a fully automated tool, TDS, which uses data mining along with recent developments in program synthesis to provide feedback for player attempts (Sections 3, 4, and 5). The tool provides personalized hints in real-time, which direct the attention of players while they

Figure 1. The Code Hunt test results screen.

code. In Section 6, we evaluate the impact of the feedback we provide. Turning to recognition, we discuss the facilities that are available by default within a system like Code Hunt, and then also how they can be taken to the next level with contests. Section 7 provides a full related work survey.

1.4. Contributions of this paper

TODO:

2. Domain Definition

There are many online programming tools [7]. Code Hunt is distinguished from the others in the following ways: it uses symbolic execution to test programs; and it is progressively open sourcing its data for public use. As such it is ideal for the study we are conducting.

2.1. The Code Hunt Game

Code Hunt [21] is an educational programming game where the player writes a program in a standard program language (Java or C#) in a simple in-browser programming environment, but the hook is [Singh] that the player is not told the specification of the program they are writing. Discovering the specification is part of the puzzle. This is quite unlike a homework assignment where a student might be told to "implement a sorting function". Instead the player is presented with a set of input-output test cases and has to start guessing what the required program might be.

At 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, as shown here.

Figure 2. Code Hunt results screen.

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.

In its basic form, Code Hunt does not have feedback in the form of hints that assist a player when blocked. However, it does have recognition of how succint the code was for the solution. This is shown in

Skill rating for the solution is one, two or three bars, and for this puzzle, the player achieved only one. The score is the rating multiplied by the difficulty of the puzzle. As the game goes on, the multiplier gets greater. The player can replay the level to achieve a better score. It is very important to provide qualitative feedback like a rating in order to encourage learning players to write better code. In Section XXX we report on how many players did so. At any time, the player can consult where he or she stands in a leader board of all players within a particular zone, thus addressing the need for recognition. There is more about recognition in Section XXXX.

2.2. The Code Hunt Data

Code Hunt has a default level progression targeting the AP computer science curriculum (approximately equivalent to a first semester college computer science course), with 130 graded problems. It has also been used for 18 worldwide contests targeted at programmers with high skill levels, or for in house contests for schools or

conferences. Over the course of a year, from March 2014 to March 2015, it has been played by over 130,000 users. They have produced 640,000 final correct solutions to the problems. For each solution, there can be anything from 2 to 50 attempts. All of these programs are logged and will be made available to the community.

The first public data set contains the programs written by students worldwide during a contest over 48 hours. There are approximately 250 users, 24 puzzles and about 13,000 programs.

The data mined for the study presented here is much larger and as it is based on the default zone.

TODO: describe data as of August - Judith can help if needed.

3. The Feedback System

3.1. Layers of Feedback

Code Hunt with our hint generation system offers four kinds of feedback listed in Table 1 which are specialized to the level by mining increasing amounts of data:

1. Compiler errors. No specialization to the level, so no data needed. If there is a compiler error, no other feedback is given as there is no program to analyze.
2. Counterexamples. Requires a single solution to the level in order to generate counterexamples with Pex [18]. The counterexamples are the core of the Code Hunt game experience, although they also act as hints and are used as such in other systems.
3. Line hints, telling the user the location(s) of one possible set of fixes to get a correct solutuion, similar to Autograder [16]. Requires multiple solutions (mined from other users) as it is better at directing users toward solutions similar to ones in the data.
4. Recommendation hints, specifying structures or expressions which the user might find useful or which are in the user's program but shouldn't appear in a solution. Requires multiple solutions and attempt sequences leading up to them in order to have enough evidence that a given structure or expression is usually useful or usually not useful.

This paper focuses on the last two kinds of feedback, which together we call Code Hunt's hint generation system.

3.2. Hint generation

TODO reference for flow in games?

In any game, a major design goal is to keep the player engaged and in flow. Staying in flow is dependent on maintaining the proper level of difficulty: too easy and the game feels unfulfilling and boring, too hard and the game feels frustrating, but just right and the game feels challenging but doable and fun. In Code Hunt the puzzle aspect of figuring out what specification the counterexamples are leading toward keeps the player engaged, but if the player is unable to figure out what to do, they will get frustrated and give up. The hints give additional feedback to help prevent the player getting stuck without giving away the answers, so the game remains challenging.

We generate two kinds of hints:

1. "Line hints" tell a player they are close and only need to change a specific line or set of lines.
2. "Recommendation hints" are for players that are further from a solution and warn them away from elements that will be unhelpful or toward ones that will be helpful.

Line hints For line hints, the problem we target 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 not be limited by a hand-coded teacher model like Autograder [16], the hint generator mines other player's solutions to determine what may appear in a solution to that level. The hint generator is powered by a program synthesis algorithm which can use that information to build new solutions as long as it knows the expressions used in it. 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.

Recommendation hints When a player is far away from a solution, it's not helpful to tell them to change every line of their attempt. Instead, we need to devise other forms of hints to give them.

Recommendation hints are built around recommending for or against the use of specific features in a program. Features may be structures like loops or nested loops or expressions like `int.MaxValue` or `3 + <int>`.

If a player is off-track, then we may be able to give a hint to nudge them away from a bad strategy.

Kind of feedback	Example	Level-specific info
Compiler errors	Cannot implicitly convert type “int” to “string”	none
Counterexamples	Correct output for x=1 is 2, your code returns 0	a solution
Line hints	Look at line 4 to capture the code	many solutions
Recommendation hints	You may find a loop useful on this level	many attempt sequences
	The expression `<int> + <int>` is rarely used to solve this level	

Table 1. Different kinds of feedback in Code Hunt

Specifically, if they are using a feature that will not lead to a solution, they can be given a hint suggesting they not use it. We need to be careful to avoid recommending against features that merely lead to an unknown or rarely used solution: we do not want to tell players they are wrong when they are merely being original. To do so, we only recommend against a feature if many other players have used it in their early attempts but not their solutions.

In the other direction, if there's no features to recommend against, then we can instead nudge the player toward a solution by recommending features that are commonly used in solutions, particularly ones that are not used by players that never reach a solution.

Both kinds of recommendation hints require data mining the attempt sequences of many players on the same puzzle. For each of those attempts, the system needs to analyze them to detect which features they use and then collect summary statistics across all of the players for use in deciding which hint to give.

Outline The program synthesis algorithm powering the line hints is described in Section 4 while the hint generation algorithm is described in Section 5.

4. Component-based synthesis

The program synthesis algorithm for the line hints is based on the DSL-Based Synthesis component of Test-Driven Synthesis (TDS) [12].

TDS performs synthesis as an iterative process: each iteration generates a new program which satisfies all of the constraints used so far which, along with additional constraints, is the input to the next iteration. Those iterations consist of running the DSL-Based Synthesis algorithm which takes an input program and modifies to satisfy additional constraints.

Because the DSL-Based Synthesis algorithm is already designed to take a program and modify it, we can easily use it to take a player's attempt and modify it into a solution. The entire TDS algorithm is not used because after multiple iterations it often veers far away from the player's attempt that it started from, which, while great for generating a working program, is less useful for generating an informative hint. So, instead TDS is only used for a single iteration, which has the effect of limiting the size of changes the algorithm is able to find.

Algorithm 1: `CBS`(P, S, e)

input : program P, set of test cases S, set of expressions e to build new expressions from

output: a program P' that satisfies S or `TIMEOUT`

```
/* Try generates one or more programs and
   if one satisfies S, CBS returns it.  */
allExprs ← e;
C ← contexts of P that fail some test case in S;
m ← number of branches in P;
while not timed out do
    foreach c ← C do
        foreach expr ← allExprs do
            Try c(expr);
    Try conditional solutions up to m branches;
    allExprs ← generate new expressions;
return TIMEOUT;
```

4.1. Algorithm

The component-based synthesis (`CBS`) algorithm takes an input program P, a set of expressions e that may be used to build the output program, and a set of test cases S the output program must satisfy. The algorithm modifies the input program by finding some subexpression and replaces it with a new expression, generating a new program P'.

The decision of **where** to modify the program and **what** to put there are the interesting aspects of the algorithm. The choices of **where** are decided when the algorithm starts, while the choices of **what** are continually generated as the algorithm runs until it either finds a solution or times out. Each new expression generated is tried in every type-correct location the algorithm identified as worth modifying.

4.2. Where to modify

Where the input program may be modified is defined in terms of contexts where a context is a program with a

typed hole that can be filled in by an expression of the proper type to produce a program.

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. The test cases let us know which parts of the program are correct and therefore do not need to be considered for replacement. Specifically, only code executed by some failing test case may be replaced.[1]

Each context represents a hypothesis about which part of the program is correct and correspondingly that the expression removed is overspecialized. Due to the way the expressions to fill in the contexts are generated, the removed expression appears in the set of expressions, so if a small change is sufficient, the expression will not be changed much.

In the full TDS, one such hypothesis is always that the entire program is wrong and should be replaced entirely. This is not a useful hypothesis for the setting of hint generation: the use of the hint is to tell the player to not get distracted by modifying the parts of their program that is already correct. If there are no such parts, then there's no hint to give. Furthermore, since we can't be sure the synthesis algorithm will always find a solution if there is one (for instance, the player's strategy may work but only if they use an expression no other player has used on the level so it is not available to the synthesis algorithm), we can never be sure the player is actually off-track. If we could, then we could tell the player to scrap their attempt and try something new.

Contexts are made out of each branch as well as the entire program in order to better support building new conditional structures (Section 4.4) 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 to produce many hints.

4.3. Choosing new expressions

The initial set of expressions comes from the input along with every subexpression of the input program. The latter is due to both the fact that a subexpression might be modified slightly before being put back into the context it was removed from and due to the general hypothesis in program repair work that repairs to a program tend to be similar to code already existing in that program.

New expressions to use in the contexts are generated by component-based synthesis [8]. 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 test cases. As opposed to previous component-based synthesis work, the generated expressions are used to fill in contexts producing larger programs which are then tested.

In our system, all components are expressions. 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 type. 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 components (i.e., functions and constants in the input expression set) in the worst case, a set that is too large will cause `CBS` to run out of time or memory before finding a solution. In practice, around 40–50 components seems to be the limit for `CBS` running on the available hardware with a 30 second timeout.

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 a hard-coded set of algebraic rewrite 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.

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 $x = 2$

[1] Angelic debugging [2] could be used to further narrow down which subexpressions are worth modifying, but it's unclear it would be worth the setup time (thereby increasing the minimum time for generating a hint) to do so in such a time-constrained application working on such small programs.

and $x = -1$. This is similar to the redundant expression elimination in version space algebras [9]. The exceptions are if the expression is part of a recursive program or lambda expression, in which case this optimization is not used.

4.4. 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 `CBS` 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 S 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.

4.5. Loops

TODO Actually discuss loop templates? It's not obvious they actually help hint generation. At best they may be a significant performance optimization, but I'm pretty sure they don't add expressivity.

5. Hint generation algorithm

TODO describe algorithms, emphasizing that both use large-scale data mining

5.1. Line hints

The line hint generation algorithm is based off the component-based synthesis algorithm described in Section 4. While it is based on the TDS algorithm [12] developed for end-user programming-by-example (PBE [4, 10]); 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 [17, 24]. 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 The synthesis algorithm 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 the synthesis algorithm (in the worst case, the time to test a possible solution is proportional to the number of test cases to run it on), 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 Component-based synthesis requires a set of expressions as an input which guides the search by limiting the set of programs to search through to those using those expressions. 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 level, we mine player solutions for what expressions they used. As long as there are at least 50 solutions, expressions that appear in at least 5 other players' solutions for a given level 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.

Given that each level will have multiple solution strategies involving different expressions, it would be

great if we could mine expressions only from solutions that used the same strategy that the player is currently attempting. While there's no obvious way to automatically partition the solution strategies, one easy approximation is to consider the set of temporary variable types in the program. Different strategies might involve different variable types and working with different variable types will definitely involve different expressions, so it's a reasonable way to partition the expression sets. Specifically, we collect an expression set for each level and each set (ignoring repeats) of temporary variable types that appeared in an attempt leading to a solution where that expression was used.

This is a relatively shallow mining of the data collected by Code Hunt, but it is sufficient to produce the expression set needed to run component-based synthesis 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 player. As our goal was to present the smallest change to the player, we rank the hints by textual edit distance: changing a single character on one line is better than rewriting an expression on another, even though the distance measured in edits to the AST may be the same or even larger.

5.2. Recommendation hints

TODO Rewrite this section. Actually include the greek.

Recommendation hints involving suggesting a player use or not use a given feature in their program. By "feature", we mean structures like "nested loops" as well as expressions like "`<char>.toString()`".

We want to advise against features that won't lead to a solution without discouraging different solution and recommend features that do lead to a solution without suggesting features that appear in both good and bad attempts. We discover these by data mining attempt sequences.

We place attempts in three categories:

1. Solutions.
2. Incorrect attempts by players who eventually reached a solution.
3. Incorrect attempts by players who have not yet reached a solution.

Due to the structuring of the game, there's no way to be sure players in (3) won't soon find a solution. We can't be sure they're off-track, they might just be in the middle of playing. Additional assumptions about time since the last attempt might help there (e.g. if the last attempt was at least a day ago, then assume the player has given up on the level), but the current system does not use the timing of the attempts.

Clearly features in (1) are likely to be useful for a solution, but, more importantly, features in (1) but not in (2) or (3) are especially interesting. Similarly, features in (2) and (3) but not (1) are worth warning against.

In a large corpus, we want to be able to ignore small counts as noise: an expression used nowhere but a single solution might be great but is more likely just a fluke. Similarly, even a bad feature will end up in some solutions if only due to its presence in dead code in those solutions.

All of this together leads to a statistical model where we want to, for each feature, estimate the probability that a program containing that feature is a solution (for positive recommendations) or a dead-end (for negative recommendations).

The naive computation would be to count all of the users who used the feature ever and find the proportion that submitted a solution containing it. That gives a simple estimate of the probability that a player using that feature will find a solution using it. We only generate a hint if the probability estimate exceeds 75%.

To clean up the issues remarked upon above relating to features with too little data to draw conclusions, we add noise to the counts. Specifically, we pretend that there are an additional 20 players who used the feature that aren't in our data and that half of them used the feature in a solution. The selection of 20 is empirically based on seeming to give good results for our data.

Additionally, we do not use a flat 75% threshold. Instead we adjust the probability estimate with its standard deviation to acknowledge that we have lower confidence in an estimate based on fewer players.

5.3. Combining the hint mechanisms

While both hint mechanisms may generate a hint, at most one hint is shown to the user. The following kinds of hints are attempted in order, going to the next in the list only if there is no hint of the previous kinds:

1. A line hint that does not recommend changing all of the code.
2. A line hint on the only line of the program, specifying that the expression to be changed is a "constant", "number", "string", "method call", or "variable".
3. A recommendation hint for an "unhelpful" feature that the player has used.
4. A recommendation hint for a "helpful" feature that the player has not used.

If there are no hints of those four kinds, then no hint is shown.

6. Evaluation

Section 6.1 presents an A/B test of disabling hints for some players in order to collect data on the effect our hint mechanism has on player behavior.

Section 6.2 presents an experiment which lends some credence to our intuition that line hints get generated when "close" to a solution and the system successfully falls back on recommendation hints when "far" from a solution.

6.1. A/B test

We wanted to know how hints affect players interacting with the game. The best possible data would be to be able to watch many people where each one played the game with and without hints. This has two problems

1. Bringing people in and watching them is expensive and time-consuming. We can instead take advantage of the fact that user interactions with the game are logged, so our experiment can be done on normal users. Furthermore, because the hints are not a promised feature, the users do not even need to know they are part of an experiment, both making recruiting easier and not perturbing the data by letting users know they are being studied.
2. It's not meaningful to have the same people play the game twice (with and without hints) because they will already know the answers when playing the second time.

6.1.1. Design

To that end, we ran an A/B test on https://www.codehunt.com where for a two week period all new users were sorted into one of three conditions:

1. Hints always (whenever a hint is generated)
2. Hints never
3. Hints on some levels

We only collected data on new users in order to avoid users having already seen hints and therefore expecting them.

6.1.2. Results

Data was collected on 407 users who began playing between the start of the experiment and two days before the data collection period ended.

Those users submitted a total of 34886 attempts. 7734 (22%) of those attempts were correct solutions. Of the remaining 27152 incorrect attempts, 5143 (19%) of them did not compile, so semantic hints could not be generated, instead compiler errors were shown.

Of the 21868 incorrect attempts that did compile, the program synthesis algorithm was able to solve 9604 (44%) of them. As not all solutions lead to good line hints, only 6284 (65%) of those were used to produce line hints. Similarly, 2762 (13%) had "helpful" recommendation hints available and 13580 (62%) had "unhelpful" recommendation hints available for a total of 16342 (75%) of attempts having a recommendation hint available, while only 12394 (75%) of those were selected as the hint generated.

The breakdown of hint kinds generated[2] for the 21868 incorrect attempts that did compile were 1498 (7%) normal line hints, 4786 (22%) line hints referencing the expression kind to change, 2273 (10%) "helpful" recommendation hints, and 10121 (46%) "unhelpful" recommendation hints. No hint was generated for the remaining 3190 (15%) attempts.

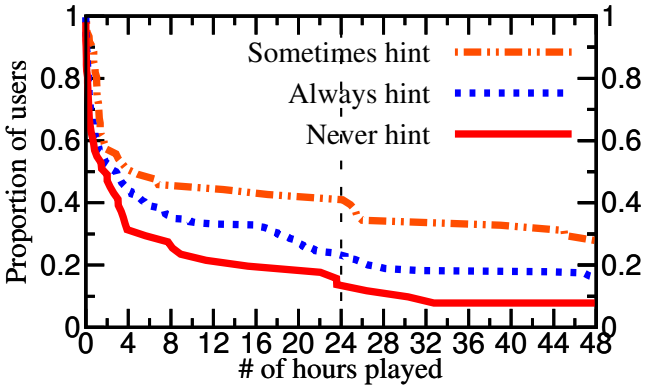

Figure 3. CDF of time between first and last play by A/B test condition

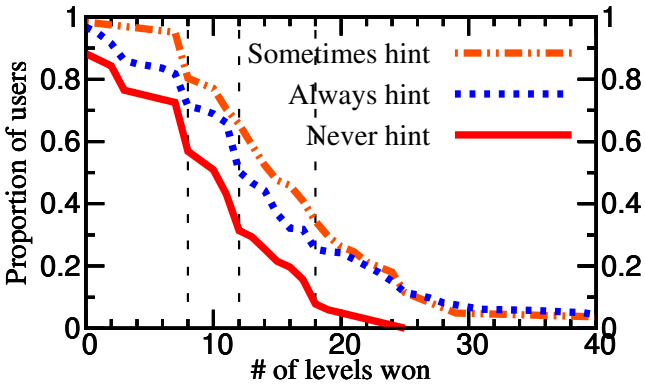

Figure 4. CDF of # of levels won by A/B test condition

6.1.3. Stickiness

What we actually want to know is whether hints affect player behavior, and, perhaps more importantly, is that effect positive. In order to answer that in the positive,

[2] These numbers are about hints generated, not hints shown to the player. Hints are still generated for players with hints disabled, they just aren't shown to the player.

we looked at the effect of hints on the stickiness of the game; that is, how long the players continue playing. In line with our hypothesis that hints reduce frustration, we expect some users who do not see hints may stop playing out of frustration while users who do see hints are less likely to do so.

Figure 3 shows how long players kept playing the game. The y-axis is the proportion of players in each condition while the x-axis is the time between the first and last attempt submitted by a player. The dashed vertical line in the middle is the 1-day mark. Looking there, we see that about 40% of the players in the "sometimes hint" condition played for at least a day, as did about 20% of those in the "always hint" condition, but only about 10% of those in the "never hint" condition did. The lines going off the graph to the right indicates players that continued playing after the 48-hour period shown in the graph.

Looking at time played in a different way, Figure 4's x-axis is the number of levels won in the same 48-hour period. The vertical dashed lines mark a few of the harder levels early on in the game where the sharp drop in the "never hint" line is visible but smaller in the other lines.[3] Looking to the right side of the graph, we can clearly see that the players who received hints went on to beat more levels than those that did not receive hints. Of course, this makes sense, since the hints make the game easier.

From these two graphs, we can conclude that hints lead to players playing the game longer, which was the goal. Furthermore, giving hints sometimes instead of always seems to be somewhat better at getting players to continue playing, perhaps due to constant hints making the game too easy.

6.2. Distance to solution

TODO Include charts

7. Related work

[27] xxx [26] yyy [16] zzz [15] aaa [14] bbb [11] ccc [3] ddd [13] eee [25] ffff [1] ggg, [23] hhh [22] iii [19] jjj [20] kkkk

Acknowledgements

Parts of this work were done across multiple interships at Microsoft Research. This work was supported by the Code Hunt team, including Peli de Halleux, Nikolai Tillman, Tao Xie, and Nigel Horspool.

References

[3] Although the game does allow players to skip around to some extent, most play the levels in order and stop playing when they get stuck instead of switching to a different level.

[1] Judith Bishop, R Nigel Horspool, Tao Xie, and Jonathan de Halleux. Code hunt: Experience with coding contests at scale. *Proc. ICSE, JSEET*, 2015.

[2] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. ICSE, 2011.

[3] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41 (2): 121–131, 2003.

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

[5] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5 (3): 4, 2005.

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

[7] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0520-4. doi:10.1145/1930464.1930480. URL http://doi.acm.org/10.1145/1930464.1930480.

[8] Susumu Katayama. Systematic search for lambda expressions. TFP, 2005.

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

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

[11] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*, pages 491–502. International World Wide Web Conferences Steering Committee, 2014.

[12] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2014.

[13] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 195–204. ACM, 2015.

[14] Kelly Rivers and Kenneth R Koedinger. A canonicalizing model for building programming tutors. In *Intelligent Tutoring Systems*, pages 591–593. Springer, 2012.

[15] Kelly Rivers and Kenneth R Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Edu-*

cation for Computer Science (AIEDCS 2013), page 50, 2013.

[16] Rishabh Singh, Sumit Gulwani, and Armando 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.

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

[18] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .NET. TAP, 2008a.

[19] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008b.

[20] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Pex4fun: A web-based environment for educational gaming via automated test generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 730–733. IEEE, 2013.

[21] Nikolai Tillmann, Judith Bishop, R Nigel Horspool, Daniel Perelman, and Tao Xie. Code hunt: Searching for secret code for fun. In *SBST*, 2014a.

[22] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Code hunt: Gamifying teaching and learning of computer science at scale. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 221–222. ACM, 2014b.

[23] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Constructing coding duels in pex4fun and code hunt. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 445–448. ACM, 2014c.

[24] Emina Torlak and Rastislav 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.

[25] Janina Voigt, Tim Bell, and Bengt Aspvall. Competition-style programming problems for computer science unplugged activities. *A New Learning Paradigm: Competition Supported by Technology, CEDETEL, Boecillo, Spain*, pages 207–234, 2009.

[26] Christopher Watson, Frederick WB Li, and Jamie L Godwin. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Advances in Web-Based Learning-ICWL 2012*, pages 228–239. Springer, 2012.

[27] Daniel S Weld, Eytan Adar, Lydia Chilton, Raphael Hoffmann, Eric Horvitz, Mitchell Koch, James Landay, Christopher H Lin, and Mausam Mausam. Personalized online education—a crowdsourcing challenge. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.