# Generating Hints From Multiple Program Embeddings

**Julian Park**
University of California, Berkeley
julianpark@berkeley.edu

## ABSTRACT

For students learning how to code programs, intelligent tutoring systems have successfully provided hints and feedback tailored to the observed program state and the inferred level of conceptual mastery. Past work has primarily focused on deriving syntactic representations of student code, like abstract syntax trees alone, to suggest changes to student code. Given recent demonstrations of success with semantic program embeddings, this paper applies multiple types of program embeddings for a pedagogical purpose: automatic hint generation. We present a new approach of utilizing both the syntactic (static) and semantic (dynamic) embeddings of programs to find a helpful code snippet for students. We evaluate our novel algorithm on programs attempted and solved by students from the CodeHunt dataset. Detailed analysis shows the viability of such algorithms for educational benefits.

### ACM Classification Keywords

K.3.1 Computer Uses in Education

### Author Keywords

Automatic hint generation; Program embeddings; Intelligent tutoring systems; Code search.

## INTRODUCTION

In 1985, Reiser et al showed that providing immediate feedback on a student's generation of a LISP program can be significantly beneficial to achieving pedagogical objectives [9]. Since then, such intelligent tutoring systems (ITS) have been researched and developed in contexts of school classrooms and Massive Open Online Courses.

One of the challenges of operating a large class is the lack of personalized feedback that a student receives, as most classrooms have the model of one teacher lecturing to an audience of students. Software technology has in some ways amplified a teacher's ability to provide guided hints and feedback upon student performance, but manually writing each hint for each student can impair feedback quality and be an ineffective usage of the teacher's time. To give more adaptive, personalized hints for students learning to program, yet maintain the large scale of computer science education, we need to build an ITS that understands and tailors to partial solutions attempted by students throughout their learning process.


ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Recent models for code assessment either provide hints independent of the student's current program or hints that are solely based on the syntactic representation of their intermediate program. Program syntax refers to the literal arrangement of variables and controls of a program, which can be analyzed purely in a static setting. Many approaches in the past have used token sequences and abstract syntax trees (ASTs) to syntactically embed programs in a space when generating hints for students [7] [10].

However, to truly enable students to learn programming at scale, an ITS must also be capable of understanding the student program's functionality, which can only be learned from dynamically executing the program (if feasible for the student's current program state). This refers to semantic representations of programs, the meaningful functionality of code.

This paper focuses on optimally searching and returning a potentially helpful code snippet for a student working on a programming task. The code snippet can inspire the student to consider a digestible modification to their code, making progress in solving their assigned problem. Our work uses a novel algorithm to search for and yield a relevant program with similar intent for the student to deliberate on.

## RELATED WORK

There has been substantial work on automated hint generation for programming tasks, like the development of a data-driven Python programming tutor ITAP by Rivers et al, which uses state abstractions to construct path-based hints [10]. They also frame the syntax-based hint in natural language for the student to consider. Piech et al has also used ASTs to infer problem solving policies with probabilistic paths [6].

Measuring the similarity between source code of student-written programs have been a major concern in related papers. Huang et al illustrates the complexity of using ASTs to model student code from MOOCs [1], and Jin et al uses linkage graphs as representations of programs [2].

Some papers extended their representation methods, like Peddycord et al using code-states, snapshots of student source code, as well as world-states representing the program outputs [3]. Recognizing this importance of program outputs, Piech et al explored a novel approach of using input-output pairs of student programs to encode programs as a mapping from an

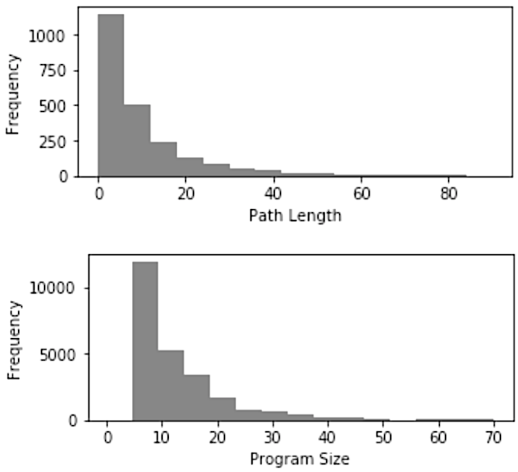


**Figure 1. Histograms of dataset statistics.**

embedded precondition space to an embedded postcondition space [5]. With Hoare triples, these conditioned input-output pairs have been successful in generating feedback.

More recently, work has been shown to manifest the power of semantic program embeddings over the aforementioned syntactic program embeddings. They have shown successful results: Want et al uses dynamic state trace embeddings to track the evolution of variable values in programs throughout runtime [11], and Reed et al uses recurrent neural networks to execute programs while maintaining memory [8]. Therefore, this paper utilizes both syntactic and semantic embeddings to develop a policy of finding the optimal code-based hints for students.

## DATASET

The dataset used in this paper is a subset of student programs from the Microsoft CodeHunt platform, downloaded at `https://github.com/Microsoft/Code-Hunt`. CodeHunt is an educational coding game which has been played by over 140,000 students and enthusiasts over recent years. This particular subset contains 259 students with 25,261 total attempts.

A student program is denoted as "winning" if it correctly solves the given problem, i.e. passes all of CodeHunt's test cases. In this dataset, 8.7% of the total attempts were winning. Exact statistics are shown in Table 1. Figure 1 shows the distributions of student path lengths (the number of attempts taken to reach a winning program) and of the program size (the number of lines written in the program). We can see that the average student takes about 10 attempts to correctly solve a problem, by writing about 15 lines of code for their program.

The dataset has been considerably preprocessed prior to analysis; for example, student programs written in C# have been transpiled to use Java for execution on macOS, and the names of each program `class` have been renamed to reflect the attempt's filename for systematic compilation.

| *Statistic* | |
|---|---|
| Students | 259 |
| Total Attempts | 25,261 |
| Winning Attempts | 2,202 |
| Average Path Length | 9.9 |
| Average Program Lines | 14.6 |

**Table 1. Summary of the dataset analyzed in this paper.**

Although we only build on an available subset of CodeHunt's data, the smaller size used here can better resemble a real-world computer science class in which the teaching staff can run the presented model in this paper to construct an embedding space of student programs for their own code-writing problems.

## PROGRAM EMBEDDING

We explore multiple ways to embed programs onto some dimensional space, mainly separated into techniques using syntactic features and those using semantic methods:

### Syntactic Features

Syntactic features can be completely determined with a static analysis of the student program, and we pick token sequencing and ASTs as two valid syntactic metrics.

#### *Token Sequences*

A straightforward way to model the syntax of a short program is to extract and sequence each token used in the program. This has been done by running a basic lexer on each program. To compare syntactic difference between two programs, we use each of their token sequences as inputs to the Levenshtein distance function, which yields a positive number of modifications necessary to convert between the string sequences. With this metric, the lower the number, the more similar the programs would be.

#### *Abstract Syntax Trees*

ASTs have been a standard method of representing the syntactical structure of a program. The idea is to build a tree in which each node is a syntactic construct of the source code and each branch connects to a construct contained inside its parent's. For instace, a `while` statement generally has two children nodes, one for the condition and the other for the body. If the boolean condition node denoted a binary comparison operation, it would then have the two variables as its children nodes.

### Semantic Features

An equally, if not more, important aspect of an intermediate student program is its functionality. Two programs that have similar AST representations may have significantly different dynamic execution at runtime. Figure 2 shows an example of such difference: the top program is incorrect, but with a small syntactic change (only the two lines underlined), the semantic

```
public class Program {
  public static int Puzzle(int a, int b) {
    int i;
    if(a==b)
       return a;
    else if(a>b) {
       for(i=2;i<=b;i++) {
          if(a%i==0 && b%i==0) b=b/i;
       }
    } else {
       for(i=2;i<=a;i++) {
          if(a%i==0 && b%i==0) a=a/i;
       }
    }
    return a*b;
  }
}
```

↓

```
public class Program {
  public static int Puzzle(int a, int b) {
    int i;
    if(a==b)
       return a;
    else if(a>b) {
       for(i=b;i>=2;i--) {
          if(a%i==0 && b%i==0) b=b/i;
       }
    } else {
       for(i=a;i>=2;i--) {
          if(a%i==0 && b%i==0) a=a/i;
       }
    }
    return a*b;
  }
}
```

**Figure 2. Two attempts sampled from User044.**

behavior improves dramatically and wins. Therefore, a sophisticated ITS should not only evaluate the syntactic construction of the program code but also the program's runtime semantics.

There were however limitations to utilizing a universal execution trace for the programs in the dataset because most programs only involved several lines within one function, not invoking any other method for an observable Java stack trace. In addition, universally injecting print statements inside for loops has been challenging because only a certain fraction of programs use loops in predictable ways. Still, upon sufficient preprocessing, we present two methods that have extracted semantic representations of programs:

*Test Injections*

Simpler programs that only necessitate a few numerical operations do not have a deep enough execution trace to meaningfully derive semantics from the stack. Thus for these programs, we systematically inject a list of print statements to test the semantic results of running the student program. Each print statement is essentially a test case running the written function with arbitrary arguments of acceptable types.

We then use the list of printed outputs by vectorizing all their numerical values into a unique semantic program embedding. Each embedding here represents the relative correctness of the student's attempt; this is a simple yet crucial measure of its functionality.

*Neural Embeddings*

For problems that fortunately involve loops or the manipulation of string literals, there was enough depth to the control flow to extract a more detailed semantic representation of the program. Inspired by the variable trace embedding model presented in a recent paper by Wang et al [11], we track how certain variable values are updated during execution by injecting print statements immediately after the target variable is updated.

Recently, deep contextualized word representations in the form of ELMo embeddings have been shown to be successful in improving performance on complex natural language processing tasks [4]. The vector embeddings are learned functions of states inside bidirectional language models. In order to transform each sequence of printed strings into viable semantic embeddings, we use their ELMo representations and subsequently compare the embeddings via cosine similarity. Although some printed strings may not be common English words, this approach is more expressive than a simple string similarity measure because the ELMo embeddings have layers of context-dependency between tokens, incorporating information about how the strings are changed over time. Each layer of the ELMo embedding has been averaged with equal weights.

## RANKING ALGORITHM

In order to find the next program hint to display for the student's current state, we must have a policy algorithm that ranks neighboring programs using the aforementioned program embeddings. We want to find a neighboring program that performs concretely better than the current student attempt, but still preserves the conceptual intent in the student code; that is, the displayed hint program should not use a drastically different way of approaching the problem. Thus, the core optimization problem here is to maximize the program's semantic improvement, i.e. how much closer the program can function like a nearby solution, yet minimize the syntactic difference, i.e. how much code must be edited by the student to reach the improved program.

We present a novel algorithm for this task, functionally named Maximized Semantics Minimized Syntax (MS2), that appropriately ranks and outputs the optimal next step in the solution path. For some intermediate student program $p$ and each neighbor $n$ in its neighboring programs $N(p)$:

$$MS2(p) = \arg\max_{n \in N(p)} [\alpha * Sem(p,w) - \beta * Syn(p,n)]$$

where $w$ is a winning solution program, $Sem(p,w)$ is the cosine similarity between the embeddings of $p$ and $w$, and $Syn(p,n)$ is the structure distance between the embeddings of $p$ and $n$. The Levenshtein distance is used to calculate the syntactic distance between program token sequences, and the AST edit distance is used between program trees. The positive coefficient hyperparameters $\alpha, \beta$ in the linear model have been empirically tuned by inspection upon grid search.

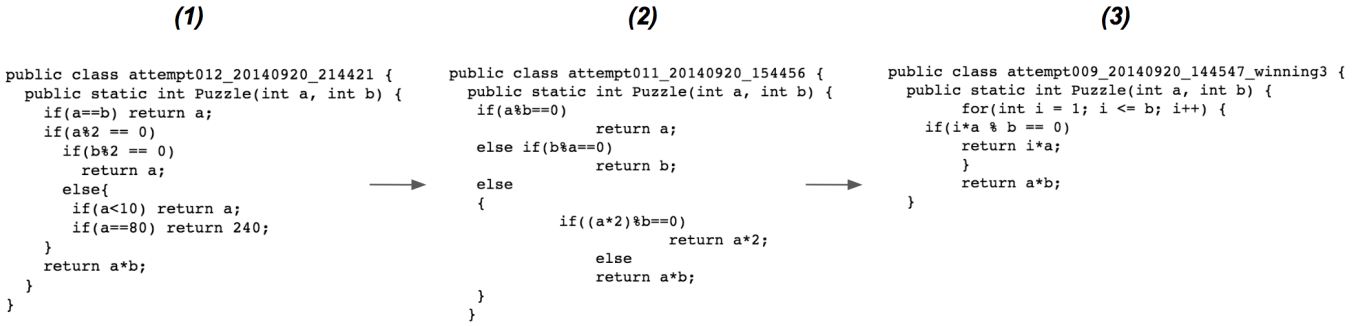


**Figure 3. An example path where *(1)* is User053's last attempt to solve the Least Common Multiple problem since User053 did not reach a winning program; *(2)* is the hint output by running MS2 (Token & Test); *(3)* is the next hint, a winning solution.**

## RESULTS AND DISCUSSION

While developing the model, we mostly relied on visual inspection to validate the effectiveness of a suggested code hint. However, to quantitatively measure the performance of the methods, we operationalize how "optimal" the sequentially predicted path is in order to reach a valid solution program state. Given that the student would adhere to the hinted code snippets, the constructed path would be an indicator of how useful the model is.

The input to the model is the last incorrect program written by a student in the dataset who was not able to reach a winning state after their sequence of attempts. This allows the model to generate a program hint that could have been helpful for the student when solving this problem.

Running our algorithm once outputs the optimal program to show the student. That is, the output program is predicted by selecting the highest ranking program neighbored in the current embedded space. An example pair of a student input program **(1)** and a model output program hint **(2)** is shown in Figure 3. We then repeat this algorithmic procedure until a winning program is reached, and we denote this sequence of intermediate programs as a *path*.

In Figure 3, we observe a 27% change in syntax (Levenshtein distance of 20 out of 73 tokens) and 19% change in semantics (increase of cosine similarity with solution program) from program **(1)** to **(2)**. Likewise, we see a 25% change in syntax (Levenshtein distance of 17 out of 67 tokens) and 4% change in semantics from program **(2)** to **(3)**.

We will analyze such solution paths mainly with three metrics of interest: average number of intermediate program steps taken in the path, the average syntactic change per step (Levenshtein distance or AST edit distance), and the average semantic change per step (percent increase in cosine similarity with solution embedding).

Numerical results upon running this procedure with different methods are displayed in Table 2, where the column metrics respectively denote the aforementioned three measures. The baseline randomly samples an arbitrary program within the attempted program space to get the next step, and repeats until a winning state is discovered. The two injection and

| *Method* | *Path Length* | *%Syntactic* | *%Semantic* |
|---|---|---|---|
| Baseline (Random) | 14.4 | 57.3 | −27.2 |
| Test Injection | 2.4 | 39.5 | 15.6 |
| Neural Embedding | 2.7 | 42.2 | 11.4 |
| MS2 (Token&Test) | 3.8 | 21.5 | 9.1 |
| MS2 (Token&Neural) | 4.2 | 25.3 | 3.7 |
| MS2 (AST&Test) | 3.5 | 20.6 | 8.4 |
| MS2 (AST&Neural) | 3.9 | 23.8 | 5.5 |

**Table 2. Comparing the results from running different iterative search methods.**

embedding methods run the procedure only maximizing semantic improvements, disregarding any syntactic measure in its ranking algorithm. Conversely, methods only using syntactic measures did not yield meaningful results because without some measure of semantic performance, most trials never reached a winning state and looped infinitely to local optima of syntactic similarity. Finally, the last four rows refer to all possible usages of the MS2 algorithm. Numerical values in the table were averaged over 10 trials for each of 5 different problems.

The MS2 model performs better than only measuring the semantic improvement of intermediate programs, although there is no significant difference between performances of the four varied MS2 types. Still, we do see that MS2 (AST & Test) performed the best overall, finding the shortest path that adequately improves the semantic performance of the sequential programs, yet reduces the need to modify program syntax from the student's perspective.

## CONCLUSION

In response to past work on syntactic and, more recently, semantic program embeddings, we present a new way to provide meaningful hints to code-writing students and develop an algorithm that works reasonably well upon inspection. This

paper opens new possibilities of representing and returning intermediate programs as potential hints for students.

As for future work, qualitatively testing this hint generation model with actual students in introductory programming classes would evaluate the effectiveness of our model in a more authentic way. Also, if a dataset permits, exploring how this model works with more complex program structures that students would encounter in intermediate computer science courses would be an interesting extension to the research problem. A program consisting of layered functions will give more room for more involved semantic embeddings, like proper stack tracing. In addition, effectively translating the shown code snippets to natural language would be helpful for students when receiving hints, but will be a substantial challenge. That is because direct mappings from code differences to hint words will not be effective because oftentimes the worded hint will just be a long, incoherent list of sequential syntactic changes.


## ACKNOWLEDGMENTS

I would like to thank Professor Marti Hearst for her guidance and many of her graduate students who gave helpful feedback for this project.


## REFERENCES


1. Jonathan Huang, Chris Piech, Andy I Nguyen, and Leonidas J. Guibas. 2013. Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC. In *AIED Workshops*.
2. Wei Jin, Tiffany Barnes, John C. Stamper, Michael Eagle, Matthew W. Johnson, and Lorrie Lehmann. 2012. Program Representation for Automatic Hint Generation for a Data-Driven Novice Programming Tutor. In *ITS*.
3. Barry W. Peddycord, Andrew Hicks, and Tiffany Barnes. 2014. Generating Hints for Programming Problems Using Intermediate Output. In *EDM*.
4. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matthew Gardner, Christopher Clark, Kenton Lee, and Luke S. Zettlemoyer. 2018. Deep contextualized word representations. *CoRR* abs/1802.05365 (2018).
5. Chris Piech, Jonathan Huang, Andy I Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. 2015a. Learning Program Embeddings to Propagate Feedback on Student Code. In *ICML*.
6. Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas J. Guibas. 2015b. Autonomously Generating Hints by Inferring Problem Solving Policies. In *L@S*.
7. Thomas W. Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating Data-driven Hints for Open-ended Programming. In *EDM*.
8. Scott E. Reed and Nando de Freitas. 2015. Neural Programmer-Interpreters. *CoRR* abs/1511.06279 (2015).
9. Brian J. Reiser, John R. Anderson, and Robert G. Farrell. 1985. Dynamic Student Modelling in an Intelligent Tutor for LISP Programming. In *IJCAI*.
10. Kelly Rivers and Kenneth R. Koedinger. 2015. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27 (2015), 37–64.
11. Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *CoRR* abs/1711.07163 (2017).