# Inferring Student Coding Goals Using Abstract Syntax Trees

Paul Freeman[1], Ian Watson[1], Paul Denny[1]

University of Auckland

**Abstract.** The rapidly growing demand for programming skills has driven improvements in the technologies delivering programming education to students. Intelligent tutoring systems will potentially contribute to solving this problem, but development of effective systems has been slow to take hold in this area. We present a novel alternative, Abstract Syntax Tree Retrieval, which uses case-based reasoning to infer student goals from previous solutions to coding problems. Without requiring programmed expert knowledge, our system demonstrates that accurate retrieval is possible for basic problems. We expect that additional research will uncover more applications for this technology, including more effective intelligent tutoring systems.

## 1 Introduction

Across nearly all fields of study, students today are increasingly motivated to improve their skills in computer science, software engineering and, more specifically, computer programming [17]. As a result, programming education courses are being added to curricula at many universities [7]. Within the Science, Technology, Engineering, and Mathematics (STEM) fields, most students are expected to have skills in computer programming after completing their university studies [20], while many other faculties are also beginning to encourage education in this area [19].

This increased demand for computer programmers has pressured academic institutions to modify the delivery method of introductory computer science materials. In addition to textbooks and lectures, computer science courses now frequently include laboratory programming exercises. While programming exercises are nothing new, the access methods being used by students today have changed to keep up with improved technology.

Online services ease the delivery of educational materials to the students. Such services increase access to more students, regardless of platform choice, software configuration, or access to university computer labs. Additionally, this cloud-based method of instruction provides a valuable record of all student activities, making it easy for schools, universities, and education researchers to access data on student learning trends [14].

Our research addresses the application of Case-Based Reasoning (CBR) into the area of computer science education. Specifically, our work investigates the potential for a system to intelligently infer the goal of a struggling student through naive analysis of both the student's current progress and existing solutions to the problem. Through inference of the goal, it is then possible to generate hints for the student.

This research targets students learning basic programming skills and completing short programming assignments. While some aspects of our research may be applicable to advanced courses, this is not the direct intention of this research.

## 2 Problem Description

Laboratory exercises are frequently used in programming courses, during which students develop solutions to simple programming problems. It is reasonable to assume that the data collected by these online learning environments is of some utility to future students solving the same problems. The data logs amount to hundreds or even thousands of prior code submissions for each programming problem. Could this data be used to generate hints for future students solving the same problem?

Our research uses a CBR approach, which we call Abstract Syntax Tree Retrieval (ASTR) to data mine prior solutions contained in a large dataset. Through analysis of retrieved solutions for specific code states, we attempt to answer the following questions:

– How can prior solutions be retained such that they are both readily accessible and easily compared to future submissions?
– What method of similarity can accurately approximate the work required by a student to move from the current state of a code string to the state of a solution string?
– Can our system frequently use the similarity method developed to select an appropriate existing solution that is not a drastically different approach to solving the problem?

The accuracy of the system is evaluated in two ways. Expert analysis is used to decide if the solution retrieved by ASTR is a reasonable solution to pursue, given the current code state. The percentage of retrievals determined to be *appropriate* provides the measure of success for this test. In a second test, we measure the system's success at retrieving a student's final solution when all of their prior attempts are submitted for ASTR. If the system selects the student's solution, this indicates that the system was able to *exactly* predict what the student would eventually do.

### 2.1 Human Tutor Emulation

ASTR attempts to emulate a human tutor, who is assumed to be an expert at solving the problem in question. The tutor observes the current failing state of the student's submission. If the student has used the correct approach, but has used incorrect code, the tutor should provide hints that lead to the correction of the code, but without changing the approach. Consider the following code strings:

```
def max_of_two_values(a, b):       def max_of_two_values(a, b):
    def max_of_two_values(a, b):       if (a <= b):
        if (a <= b):                       return b
            return b                   else:
        else:                              return a
            return a
```

The student solved the problem correctly (shown on the left), but they were unaware that the system would add the function definition to the code they submitted. This is an artefact of the online platform and we would expect a tutor to guide the student toward an appropriate solution (shown on the right).

When the wrong approach has been taken by the student, the tutor should suggest modifications that result in a solution that uses as many parts of the current attempt as possible. The following submission and solution is considered a misunderstanding of the problem.

```
def max_of_two_values(a, b):       def max_of_two_values(a, b):
    if (a < b):                        if (a > b):
        return a                           return a
    else:                              else:
        return b                           return b
```

On the left, the student calculated the incorrect return value. The tutor could help the student towards the solution on the right. Such a solution is certainly a logical step from the current state of the student's code, however, the tutor may also guide the student to a solution that swaps the return values, `a` and `b`, into the appropriate position, leaving the `<` operator alone. Both goals are considered equally appropriate.

The tutor should try to find a solution that is similar to the code the student has already created. In the previous example, it would be inappropriate for the tutor to move the student towards a radically different solution, such as the following:

```
def max_of_two_values(a, b):
    return max(a, b)
```

Ideally, an expert human tutor is able to provide appropriate guidance because they are able to consider many different ways of solving the problem and can accurately select a solution that seems most similar to what the student has written. ASTR provides this solution, which could then be used to generate any number of hints and guide the student in the appropriate direction.

### 2.2 A Case for Case-Based Reasoning

Online learning environments provide a tremendous opportunity to make use of CBR. Specifically, computer programming exercises lend themselves to CBR due to the fact that students are typically attempting to generate one of many (possibly infinite) solutions.

During many programming exercises, it is desirable to allow the student to explore the entire solution space rather than restricting them to finding one of only a few solutions. The use of CBR gives us the opportunity to allow this open exploration of solutions by using prior solutions to guide students back into the area of the total search space known to contain solutions.

## 3 Related Work

Our research builds off prior work in a number of areas. The most important of these areas include: Abstract Syntax Trees (ASTs), code clone detection, hint generation, and CBR. Here, we briefly introduce the current state of research in these topics as it relates to our work.

### 3.1 Abstract Syntax Tree

An AST is a hierarchical representation of a program into a branching sequence of operators. Each tree represents the hierarchy of a program. The leaves of the AST indicate which calculations are performed first. The results of these calculations become the next level of leaves. The process moves up the tree until the total program execution is calculated at the root of the trees. An example of a simple Python AST is provided in Fig. 1.

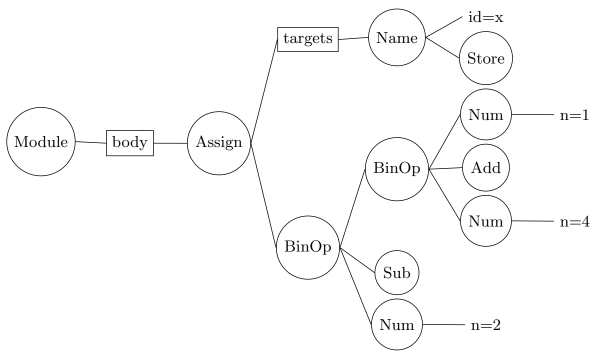

**Fig. 1.** The Python AST for `x = 1 + 4 - 2`.

ASTs provide a method for comparing source code fragments that focuses on the operations occurring within programs rather than the string labels used to identify the parts. They have a wide range of application areas. A tool developed by Falleri et al. [5] provides edit scripts (or *diff* files) for two versions of code by analyzing the AST. Generating edit scripts through a text-based process, while still correctly documenting the intended changes of the programmer, has an algorithmic complexity of $O(n^3)$ at a minimum. Using an AST method, the authors were able to develop an algorithm running, in worst case, in $O(n^2)$ time.

The use of ASTs in the educational domain has also been examined. Rivers and Koedinger [16] investigated the potential of using ASTs as part of an Intelligent Tutoring System (ITS). They highlight many of the advantages to using ASTs to compare student code to existing solutions. Their work attempts to establish a common framework upon which to build simple programming tutors using AST comparisons. They also explore possible methods of quickly generating hints for students from this existing data.

### 3.2 Code Clone Detection

One of the more successful applications of Abstract Syntax Trees has been in the areas of *code clone detection*, which is used to identify code segments that perform the same function, in the same way. Code clones are created through several processes and detecting these clones is useful. Many tools have been made that detect these clones [2] and they are often used to identify areas within a project for refactoring.

AST comparison methods cannot detect code clones in all situations. Inversion of `if-else` statements was identified as a problem area for AST-based code clone detection [4]. This clone generation process involves inverting the condition clause of an `if` statement and swapping the code contained within the `if` and `else` blocks. Other techniques, such as loop unrolling or inserting dummy methods, can also mask code clones from these detectors.

Tao et al. [18] researched methods by which ASTs can be adjusted to catch additional code obfuscation techniques. Their process provided procedural analysis for many common changes in logical structure. However, if the system translates the code into an AST so as to account for this, many common logical syntactic changes used to create code clones will be unable to mask their semantic similarity to the original.

Detecting code clones is useful for identifying potentially unwanted duplication of code. However, code clone detection can also be used in a more positive manner. In introductory programming courses, when students attempt to solve a programming exercise, they are attempting to create a code clone of one of the possible solutions. The measure to which a student's code is a clone of a given solution can be used as a measurement of their distance from that solution.

Leveraging this distance calculation allows the computation of a *nearest* solution when many are available. This observation is an important component of our ASTR system. Once a nearest solution is calculated, hint generation may be performed.

### 3.3 Hint Generation

During a problem solving task, especially one in which a tutor is involved, the ability to provide hints to the user is of interest.

Hints that have been preprogrammed into a system are known as *authored hints*. Authored hints are commonly used to provide feedback [12], but can only ever provide hints for a finite number of cases. It is therefore of interest to develop methods of automatically generating hints.

Arguably, some of the most robust hint generation systems fall into a category of systems known as *goal-directed hint generators*. Such systems possess the domain knowledge necessary to compute a path from any current state to the goal state. Goal-directed hint generation provides the greatest flexibility in generating hints, but at the expense of requiring large amounts of expert domain knowledge [14].

*State-based hint generation* is a broad category of hint generators that provide hints based on a known or computed path to a goal state, but lack the ability to provide hints for *all* possible states. State-based hint generators frequently make use of authored hints [1], [13]. Because the system is waiting for specific state-driven events, the variety of hints required is limited. It is easy to give instructors the opportunity to improve the hint language used for a particular state.

Some tutoring systems have made use of annotations within authored solutions to improve the readability of generated hints [6]. Solutions are processed and the system indicates with a placeholder when annotations are needed. This allows the author to input custom annotations for hints, at the direction of the system.

## 3.4 Case-Based Reasoning

The availability of solutions leads to potential application of CBR. CBR aims to retrieve a prior solution, or case, with similarity to the current problem. The retrieved prior solution and the current problem can be used in conjunction to propose a solution to the current problem.

CBR has a range of applications in education. Kolodner et al. [10] wrote a commentary on the use of many different CBR-inspired approaches to learning. They concluded, "CBR-inspired educational approaches will be making their way more into the e-learning mainstream." Essentially, since CBR takes a similar problem solving approach to that of students, CBR technologies can be used to develop cognitive models of students.

Regan and Slater [15] developed a case-based hint generator for the virtual world DollarBay. The tutoring system was designed to teach users how to play the game effectively. An agent inside the virtual world would visit players who were struggling with certain aspects of the basic strategy. The agent would provide a message to the user, advising them how to improve their gameplay strategy. Inaction by the agent is also considered a valid action, and was the proposed "solution" to a subset of cases, as well. This event-driven approach is not a true CBR system, but does demonstrate the effective use of a case-base in an educational environment.

Although never explicitly referred to as a CBR system, the data mining hint generator, developed by Jin et al. [8], performs as such. Previous solutions and intermediate steps are mined by the system, with each solution being stored as a linkage graph, indicating relationships between variables. When a student requests a hint, the system transforms the code into a linkage graph and calculates the most similar "case". Since intermediate steps are saved by the system, the next step along a path to the goal is used to provide a hint to the student.

## 4 Abstract Syntax Tree Retrieval

Our system uses a process we refer to as Abstract Syntax Tree Retrieval (ASTR). It requires no prior knowledge of the problem being solved. It uses CBR and the grammar of the programming language to retrieve a prior solution with high similarity to a struggling student's failing submission.

Like many programming education systems, ASTR does not classify solutions as *more* or *less* correct. The system defers to the results of acceptance tests to determine correctness of a submission. *Every* student attempt at solving *any* programming problem is processed by the ASTR system. Attempts that pass the acceptance tests of a programming problem, and have not already been seen, are retained by the case-base as new solutions, while attempts that do not are matched to the most similar solutions from the case-base. Ideally, ASTR should perform in the same manner as the human tutor described in Section 2.1. During the retrieval process, the goal of the system is to retrieve a solution from the case-base that matches the intended solution of the student.

The case retrieved from the case-base is referred to as the *goal* of the student. If an expert believes the goal would be appropriate for the student to work towards, the case is referred to as an *appropriate goal.* The challenge of ASTR is to correctly retrieve an appropriate goal from the case-base as frequently as possible.

### 4.1 Student Goal Assumptions

In the field of computer science, *edit distance* is a well-known method for determining the similarity of strings [11], trees [3], [21, 22], or graphs [9], [23]. An underlying assumption is made that the student is attempting to reach the goal that will require the fewest number of edits to the current state of their program. By observing a student's initial failing attempt, and comparing it to their eventual solution, we can show this is frequently true.

In ASTR, edits are defined to be one of three possible modifications to the current state of the student submission:

1. Adding to the code.
2. Removing from the code.
3. Modifying an existing part of the code.

Additionally, edits are assumed to be made directly to the AST, despite the reality of students making edits to the source code text. However, since text modifications resulting in the same syntax tree are of no interest to our research, these types of modifications can be ignored.

It is also assumed that the case-base will always return one of the goals retained by the case-base. Although the goal will not always be in the case-base, assuming this would simply result in failing to return any case.

With these assumptions in place, we have reduced our problem to that for which CBR is most appropriate. Given a student submission and a set of successful cases, return the successful case that is the fewest number of edits from the state of the student submission.

## 4.2 Preprocessing and AST Generation

When a submission is received by the system, the Python string is parsed into an AST using the `ast` module. However, in addition to AST generation, a number of preprocessing steps are used to improve similarity calculations.

**Removing Unreachable Code.** Novice programmers frequently include unreachable code in their programs. Commercial tutoring systems would probably choose to notify the student about the unreachable code, but our research system would benefit from removing unreachable code as a preprocessing step before making any comparisons.

The following Python keywords are identified as being markers for potentially unreachable code: `break`, `continue`, and `return`. Code following any of these keyword arguments, provided it's at the same indentation level, will not execute. Consider the following example, which is a passing submission:

```
def max_of_two_values(a, b):
    if a > b:
        return a
        print a
    if a < b:
        return b
        print b
```

The `print` statements in this code cannot execute. Functionally, this code would be the same as a submission without the print lines, however, the system would identify them each as a different solution, since the ASTs of the submissions would be different. It is for this reason that the preprocessing step removes any code following a `return` statement, or other block-terminating Python keyword.

In addition to these removals, the system removes code following `if/else` statements when one of these block-terminating keywords exists in both the `if` and `else` blocks, as in the following example:

```
def is_odd(num):
    if num % 2 == 0:
        return False
    else:
        return True
    is_odd(52)
```

These rules can be written into a tree pruning algorithm which traverses the AST recursively, removing unreachable code, resulting in fewer unique solutions for a given problem.

**Variable Standardization.** Online programming exercises rarely enforce naming conventions for the variables used in problem submissions. ASTs do not reach a level of abstraction that removes variable names. Therefore, submissions with different variable names will create different syntax trees.

A simple method for standardizing labels is to simply replace each variable name as it occurs with algorithmically generated values. In our system, each variable, as it is encountered, is replaced with an enumerated value. The associated variable names are stored in a table and future values are replaced with the same value as was used prior. This method is effective for problems with shorter length, but would need to be expanded to support problems with a larger number of variables. Additionally, there may be rare instances where the order of variable declaration makes the system unable to assign variables in a syntactically consistent way between submissions; however, such instances were not observed during our experimentation.

### 4.3 Case Retrieval

The preprocessed AST generated from the submission is now ready for ASTR. All submissions, whether passing for failing, are of some interest to the system. If a submission has successfully solved the problem, then solution retrieval is not needed. However, the submission should be added to the case-base if it is a new solution. The system does an equality check against existing solutions in the case-base. If the submission is unique, it is added to the case-base. Duplicate submissions are currently ignored, although retaining the duplication count for each submission could have future application. If the submission has not solved the problem (i.e. it is a failing submission), ASTR is used to calculate and return the most similar existing solution.

**Zhang Shasha tree edit distance.** The similarity between ASTs is calculated with the Zhang Shasha (ZSS) tree edit distance algorithm [22]. The algorithm uses a dynamic programming approach to calculate the exact edit distance between the two input trees. The implementation used by the system is a Python implementation provided in the `zss` module. The implemented algorithm allows for customization of the cost weights for different edit operations. The algorithm returns a total cost value based on the cost summation of all edit operations necessary to transform one tree into the other.

For our experiments, we use two different weightings for the costs of edit operations. The first weighting assigns the cost to each edit operation as 1. We refer to this as the *metric* tree edit distance (TED) calculation in our results. The second weighting places a weight of 3 on *add* node operations, a weight of 2 on *change* node operations, and leaves the *remove* node weight as 1. We refer to this as the *weighted* TED calculation. The weighted calculation slightly favors smaller ASTs and is shown to perform better in many of our tests.

The ZSS algorithm is used to calculate a similarity value for all ASTs stored in the case-base. The AST with the smallest tree edit distance is the AST with the highest similarity and is the result of ASTR.

## 5 Dataset

Our research uses a large dataset to test the effectiveness of ASTR. The data consists of submissions created by students to solve a variety of programming problems. The submissions are written in the Python programming language. There are a total of 57,234 submissions, from 24 programming questions, in the dataset. The level of the exercises is targeted to beginning programmers.

### 5.1 Programming Questions

ASTR was evaluated against three problems from the dataset, which were uniquely identified as problems: 2593, 2594, and 2600.

Problem 2600 required students to determine the maximum value of two inputs. It had the greatest ratio of *total solutions* (154) to *unique solutions* (20), with an average of 7.7 repetitions of each solution. It was also the submission with the largest average number of users submitting the same solution, which was 6.35 users per solution. Almost half of the successful submissions, 74 out of 154, were reduced to the same unique solution. From a complexity standpoint, solutions to this problem were short, relative to the other problems in the dataset, and provided an acceptable baseline test for the system.

Problem 2593 required students to determine if an input was odd. Both problems 2593 and 2600 had a similar number of unique solutions, but this problem had a much greater number of unique failures. There could be many explanations for the large number of failures, but it would seem that a larger number of failures might correlate with a lower understanding of the problem by the students. However, student anomalies in submission patterns can also cause spikes in unique failures. For instance, some students will begin making arbitrary changes to their failing code in an effort to stumble upon a solution. This is especially likely when the student is given unlimited submissions and checking each submission is almost instantaneous. A sample solution to the problem 2593 can be seen here:

```
def is_odd(num):
    if ((num % 2) == 0):
        return False
    else:
        return True
```

Problem 2594 was more complicated than the other problems in the experiment. It required students to calculate the largest divisor of the input value. Below is a sample solution to problem 2594:

```
def largest_divisor(num):
    largest = 0
    for i in range(1, num):
        if ((num % i) == 0):
            largest = i
    return largest
```

Students created 82 unique solutions to this problem and over 300 unique failing submissions. When examining the failing submissions, it was clear that many students either did not understand the definition of *largest divisor* or did not understand the algorithm used to calculate the largest divisor. This contrasted well with the other questions we used, as the definition of *odd* and *maximum* are typically well understood. Programming problem 2594 demonstrates our system's ability to handle submissions that correctly solve *a* problem, but the *wrong* problem.

## 6 Experiments

The intent of the experiments was to determine the performance of the case-base under ideal circumstances. To simulate ideal conditions, we would manually insert every solution into the case-base. With the well developed case-base, the experiment then presented each failing submission to the system. Identical failing submissions were removed. For each failing case, the case-base returned the most similar passing case. A log was created showing the source code of the failing case and the source code of the most similar matching case selected by the case-base. The log was reviewed by a human expert and marked for the appropriateness of the case returned by the case-base.

In assessing appropriateness, the expert is attempting to infer the goal of the student when the failing submission was written. Whether or not the retrieved case indicates the actual goal of the student is highly subjective. Therefore, the expert must decide if the goal proposed by the case-base is a natural progression from the current state of the failing code. Assessing the retrieved solutions was straightforward, but time intensive, requiring a couple hours to review logs containing the results of a couple hundred submissions.

The overall score of the case-base is the overall percentage of appropriate retrievals compared to the total number of failing submissions processed.

By providing the case-base with all solutions, there is an increased chance that the actual intended solution of a failed attempt will be contained. For example, if a student made a submission that failed the acceptance tests but then later made a second submission that passed the acceptance tests, it is likely that the second submission represents the original *goal* (see Section 4) of the student (although this is not always the case). By loading all solutions into the case-base prior to performing case retrieval, it is possible to match a student's failing submission with any future passing submission. If the case-base selects a student's own future submission as the most similar solution, this may be another indication that the ASTR system is performing accurately.

A separate experiment was performed to determine the frequency with which the student's own solution was retrieved as the goal solution by ASTR. For each failing submission, any solutions to the same problem, by the same user, were added to the case-base before any other cases were added. It was necessary to add the user's solutions first because the case-base only retains the first instance of each unique solution. Adding them first ensured that the user's solutions would be available for ASTR. The test performed the experiment and recorded the numbers of both accurate and inaccurate retrievals.

## 7 Results and Discussion

When ASTR was performed using a TED metric, the system was frequently able to retrieve an appropriate goal solution. Performance using a weighted TED calculation improved accuracy on problems 2594 and 2600 significantly while reducing the accuracy of problem 2593 only marginally. The results are shown in Table 1.

**Table 1.** ASTR appropriate retrieval performance

| | Metric | | | Weighted | | |
|---|---|---|---|---|---|---|
| Question | 2593 | 2594 | 2600 | 2593 | 2594 | 2600 |
| Unique Solutions | 25 | 81 | 21 | 25 | 81 | 21 |
| Unique Failures | 157 | 383 | 31 | 157 | 383 | 31 |
| Correct Retrievals | 113 | 247 | 20 | 107 | 309 | 30 |
| Incorrect Retrievals | 44 | 136 | 11 | 50 | 74 | 1 |
| Accuracy | 72% | 64% | 65% | 68% | 81% | 97% |

Question 2593 was the only programming problem that had a lowered accuracy using the weighted distance calculation. However, the number of inaccuracies only increased by 4%. The cause for this change is not readily apparent. In any case, 2593 was the lowest performing test question.

Question 2594 contained the largest number of submissions, but still performed well. The initial performance of 64% was the lowest recorded accuracy for the metric retrieval test, but this question responded very well to the weighted calculation, making 62 more correct retrievals than with metric calculation and putting the accuracy for this test at 81%.

Question 2600 had the fewest examples of correct submissions and the fewest examples of unique failures. ASTR returned reasonable solutions for 30 of 31 submissions, or 97% accuracy, using the weighted distance calculation. The metric calculation performed similarly to the other questions, with an accuracy of 65%.

**Table 2.** User's solution identified as goal

| | Metric | | | Weighted | | |
|---|---|---|---|---|---|---|
| Question | 2593 | 2594 | 2600 | 2593 | 2594 | 2600 |
| Success | 111 | 66 | 30 | 76 | 64 | 34 |
| Failure | 142 | 415 | 22 | 177 | 417 | 18 |
| Accuracy | 44% | 14% | 58% | 30% | 13% | 65% |

ASTR accuracy rates were lower when we examined the rate at which a user's own solution was retrieved as the goal solution. Question 2600 performed the best, with over half the failing submissions being matched up with the correct solution. This could be a by-product of having many students submitting the same solution, however, question 2593 also had a high accuracy rate on this test (although the results dropped 14% for the weighted distance test). Question 2593 underperformed on this test, with similar scores of 13% and 14% for the two similarity variations.

## 8 Conclusion

The results achieved by our system are encouraging. The ASTR system contains no information about the programming problem prior to observing successful submissions. Additionally, the system has no understanding of Python syntax. Despite these limitations, the system is usually able to correctly identify a solution that appears to an expert to be the student's intended goal.

The ASTR system is generally accurate for 2 out of 3 submissions at a minimum, with favorable problems potentially performing much better. Weighted TED calculations seem to result in improved retrieval accuracy over metric TED calculations, as was hypothesized. Retrieval performance was not improved enough to show this definitively, but the results are encouraging.

Student's "own solution retrieval" test had less favorable results. When taken together with the other results, this may indicate a need for more interaction with students during programming exercises. We have shown that ASTR frequently returns an appropriate goal, but this test shows that students may not have discovered this appropriate goal on their own. Although the exact reason for this is not known at this time, there is a potential application of ASTR to this problem in the future.

Prior solutions to introductory programming problems contain valuable knowledge for assisting students who are currently struggling. AST analysis seems to provide a strong naive data structure from which similarity values can be calculated between current state and goal state. Solutions with similar ASTs are often determined to be appropriate goals. Our ASTR system is able to leverage this knowledge with minimal effort, providing potential for the development of ITSs, hint generation systems, and working across a large of domain of problems.

## References

1. Antonucci, P., Estler, C., Nikolic, D., Piccioni, M., Meyer, B.: An incremental hint system for automated programming assignments. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15). pp. 320–325 (2015)
2. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance (ICSM '98). vol. 98, pp. 368–377 (1998)
3. Bille, P.: A survey on tree edit distance and related problems. Theoretical Computer Science 337(1-3), 217–239 (2005)
4. Cui, B., Li, J., Guo, T., Wang, J., Ma, D.: Code comparison system based on abstract syntax tree. In: 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT). pp. 668–673 (2010)
5. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Montperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14). pp. 313–324 (2014)
6. Gerdes, A., Heeren, B., Jeuring, J.: An interactive functional programming tutor. In: Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12). pp. 250–255 (2012)
7. Guzdial, M.: A media computation course for non-majors. SIGCSE Bulletin 35(3), 104 (2003)
8. Jin, W., Barnes, T., Stamper, J., Eagle, M.J., Johnson, M.W., Lehmann, L.: Program representation for automatic hint generation for a data-driven novice programming tutor. In: Proceedings of the 11th International Conference on Intelligent Tutoring Systems (ITS '12). pp. 304–309 (2012)
9. Kammer, M.L.: Plagiarism detection in Haskell programs using call graph matching. Master's thesis, Utrecht University (2011)
10. Kolodner, J.L., Cox, M.T., González-Calero, P.A.: Case-based reasoning-inspired approaches to education. Knowledge Engineering Review 00(1997), 1–4 (2005)
11. Lu, W., Du, X., Hadjieleftheriou, M., Ooi, C.: Efficiently supporting edit distance based string similarity search using B+-trees. IEEE Transactions on Knowledge and Data Engineering 26(12), 2983–2996 (2014)
12. Mckendree, J.: Effective feedback content for tutoring complex skills. Human-Computer Interaction 5(4), 381–413 (1990)
13. Paquette, L., Lebeau, J.f., Beaulieu, G., Mayers, A.: Automating next-step hints generation using ASTUS. In: 11th International Conference Intelligent Tutoring Systems (ITS '12). pp. 201–211 (2012)
14. Piech, C., Sahami, M., Huang, J., Guibas, L.: Autonomously generating hints by inferring problem solving policies. In: Proceedings of the Second (2015) ACM Conference on Learning @ Scale (L@S '15). pp. 195–204 (2015)
15. Regan, P.M., Slator, B.M.: Case-based tutoring in virtual education environments. In: Proceedings of the 4th International Conference on Collaborative Virtual Environments (CVE '02). pp. 2–9 (2002)
16. Rivers, K., Koedinger, K.: A canonicalizing model for building programming tutors. In: Proceedings of the 11th International Conference on Intelligent Tutoring Systems (ITS '12). pp. 591–593 (2012)
17. Snyder, L.: Being fluent with information technology. National Academy of Sciences (1999)

18. Tao, G., Guowei, D., Hu, Q., Baojiang, C.: Improved plagiarism detection algorithm based on abstract syntax tree. In: 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies (EIDWT). pp. 714–719 (2013)
19. Vaidyanathan, S.: Fostering creativity and innovation through technology. Learning & Leading with Technology pp. 24–28 (2012)
20. Wilson, C., Sudol, L.A., Stephenson, C., Stehlik, M.: Running on empty: The failure to teach K-12 computer science in the digital age. Tech. rep., Association for Computing Machinery (ACM) (2010), http://runningonempty.acm.org/fullreport2.pdf
21. Yang, R., Kalnis, P., Tung, A.K.H.: Similarity evaluation on tree-structured data. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05). pp. 754–765 (2005)
22. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal on Computing 18(6), 1245–1262 (1989)
23. Zheng, W., Zou, L., Lian, X., Wang, D., Zhao, D.: Efficient graph similarity search over large graph databases. IEEE Transactions on Knowledge and Data Engineering 27(4), 964–978 (2015)