Misconception-Driven Feedback: Results from an Experimental Study

Luke Gusukuma
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
lukesg08@vt.edu

Austin Cory Bart
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
acbart@vt.edu

Dennis Kafura
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
kafura@cs.vt.edu

Jeremy Ernst
School of Education
Virginia Tech
Blacksburg, Virginia
jvernst@vt.edu

ABSTRACT

The feedback given to novice programmers can be substantially improved by delivering advice focused on learners' cognitive misconceptions contextualized to the instruction. Building on this idea, we present Misconception-Driven Feedback (MDF); MDF uses a cognitive student model and program analysis to detect mistakes and uncover underlying misconceptions. To evaluate the impact of MDF on student learning, we performed a quasi-experimental study of novice programmers that compares conventional run-time and output check feedback against MDF over three semesters. Inferential statistics indicates MDF supports significantly accelerated acquisition of conceptual knowledge and practical programming skills. Additionally, we present descriptive analysis from the study indicating the MDF student model allows for complex analysis of student mistakes and misconceptions that can suggest improvements to the feedback, the instruction, and to specific students.

CCS CONCEPTS

• **Applied computing** → **Education**; Learning management systems; • **Social and professional topics** → **Computational thinking**; **CS1**; **Student assessment**;

KEYWORDS

CS Education; Immediate Feedback; Student Model; Misconception

ACM Reference Format:
Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-Driven Feedback: Results from an Experimental Study. In *Proceedings of 2018 International Computing Education Research Conference (ICER '18),* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3230977.3231002

ICER '18, August 13–15, 2018, Espoo, Finland

ACM ISBN 978-1-4503-5628-2/18/08…$15.00
https://doi.org/10.1145/3230977.3231002

1 INTRODUCTION

Non-computing majors often struggle with introductory programming, because of their limited prior knowledge and low self-efficacy. These students require quality feedback to guide students to correct answers [21, 32], maintain student motivation [28], and become self-regulated learners [22]. While direct feedback from experts is the "gold standard" [5], it has two pragmatic drawbacks. First, the availability of experts is limited, especially in larger classes where the learner-expert ratio is high or in distance-based learning where an expert is remote. Second, expert feedback may be delayed, requiring an arranged time for the expert and learner to interact.

In some cases, feedback opportunities can be determined automatically and presented to the learner immediately, such as the results of unit tests showing how close a learner is to a correct solution. However, this kind of feedback is centered around a model of the problem solution rather than a model of the learner, expecting that simply pointing out a novice's mistakes will help them infer the correct knowledge. Although suitable for advanced learners, such high level of critical thinking may not be available during low level skill acquisition that occurs in the introductory level.

To improve the impact of immediate feedback on learners we present Misconception-Driven Feedback (MDF) based on the idea of a "knowledge component" [1] from cognitive learning theory. In this model, mistakes detected through program analysis provide evidence for a set of misconceptions defined by the instructor. With the misconceptions in mind, feedback messages can be authored by an instructor to target learners' misunderstandings.

We conducted a quasi-experiment comparing students' learning with the help of conventional feedback vs. MDF feedback; the impact was measured by their summative performance on multiple choice quizzes and programming problems. This study controls for the instructors, problems, grading, and learning materials. This paper reports on the experiment and makes these contributions:

(1) A novel feedback approach based on a learner model connecting a learner's mistakes to underlying misconceptions.
(2) Statistical evidence of the positive impact of misconception-driven feedback on student learning and performance.
(3) Descriptive analysis identifying the influence of specific misconceptions on programming mistakes.

2 BACKGROUND

MDF draws on cognitive learning theories, prior work with misconceptions, and general education theories related to feedback. The technology we have created is related to approaches such as Intelligent Tutoring Systems and Hint Generation Systems.

2.1 Cognitive Theory

A fundamental aspect of MDF is the idea of modeling a student's understanding through knowledge components. A knowledge component can be defined as "an acquired unit of cognitive function or structure that can be inferred from performance on a set of related tasks" [19]. Guided by this perspective, we define the related ideas of a (programming) misconception and a (programming) mistake:

- A programming misconception is a unit of cognitive function or structure that can be inferred from a mistake on a programming task.
- A programming mistake is an incorrect configuration of code elements.

We will often elide the word "programming" and write misconception or mistake for simplicity.

2.2 Misconceptions in Programming

There is a body of work on misconceptions that novice programmers encounter. A considerable subset of this work is on discovering existing misconceptions and developing Concept Inventories[7, 15, 20, 27, 29, 31]. The misconception discovery techniques in these works range from interviews [7, 15, 20] and quizzes [7, 27] to analyzing about mistakes in student code [7, 20, 29, 31]. Our work specifically uses misconception discovery methods described in [12]. While there are many techniques for misconception discovery, there is little work on how to use misconceptions in programming assignments, especially with regards to detecting them in student code and delivering appropriate feedback. Some work discusses systems that can detect misconceptions and deliver appropriate feedback such as [30], but do not present a formal model.

2.3 Formative Immediate Feedback

Feedback in various forms and styles is a critical element of virtually all learning theories [32]. Our work focuses on formative feedback: "information communicated to the learner that is intended to modify the learner's thinking or behavior for the purpose of improving learning" [28]. Different approaches to formative feedback include verification, explanation, hints, and worked examples[21]. Feedback presentation can be immediate, meaning without explicit request and/or on-demand (e.g., when a program is executed). Our vision for creating effective formative feedback, which we term misconception-driven feedback, is based on two ideas:

- Feedback should be grounded in an understanding of student misconceptions.
- Feedback should be coupled with instruction.

In this way we can assess and improve feedback in the broader context of teaching and learning.

2.4 Intelligent Tutoring and Hint Generation Systems

Intelligent Tutoring Systems for Programming (ITP) are systems used for learning programming, have pre-scripted programming problems, and adapt based on multiple metrics and an algorithm [10]. These systems have a scaffolded programming learning experience, and are useful for online settings with unsupervised instruction. However, ITPs take much effort to assemble [18] and have different types of constraints from typical Intelligent Tutoring Systems (ITS) [10]. ITPs are typically heavily scripted, difficult to adapt to new contexts, and are meant to be self-contained and divorced from other instruction. In contrast, our intent is to augment conventional classroom instruction with more light-weight feedback.

Hint Generation systems give on-demand, logical next steps for a student to take based on prior students' programs, thus enabling student progress while avoiding instructor involvement (achieving scalability). There are many examples of such systems, typically able to be integrated into existing programming environments [23], [24], [25], and [26]. Although useful tools for helping students, these systems do not aim to help students understand *why* their advice and hints should be followed.

The difficulty and effort of developing intelligent tutoring systems [18], and the weakness of hint generation systems not contextualizing hints with instruction [25], suggests a need for a middle ground for writing and delivering feedback for students. This middle ground is still being explored and is relatively underrepresented. CSF^2 cross references results of unit-tests to identify misconceptions within student mistakes based on prior semester data [13]. Mistake Browser and Mistake Propagator [14] incorporate instructor expertise by having the instructors annotate hints found via typical hint generation techniques like in [26] and [24]. Instructor-supervised feedback systems can provide higher quality feedback than hint generation systems, at a cost lower than intelligent tutoring systems. However, the techniques in [13] and [14] suffer from the typical slow start problem of data-driven approaches.

3 APPROACH

In this section, we elaborate on the role of misconceptions in our learner model, the method for detecting mistakes, and feedback delivery. Key ideas are exemplified in Figure 1 and explained below.

3.1 Learner Model

MDF is centered on a model of breakdowns in the learners' understanding as a set of misconceptions, which can be determined by the instructors through analysis of the curriculum and prior student work. In our case, our curriculum was developed in semesters prior to our experiment. The curriculum was analyzed using selected elements of a formal Instructional Design process; a description of the process used to analyze the instruction can be found in [12].

From the Instructional Design process, anticipated misconceptions and associated mistakes were gathered.A general model of a student was then built from these mistakes and misconceptions. At the core of the model, a mistake is associated with a vector of misconceptions. Mistakes are the "observed performance" of the student. By cross-referencing multiple mistakes, a misconception can be isolated. An example misconception is shown in Figure 1a.

Learner does not know the difference between a count and a sum.

(a) Misconception

```
#presence of:
for x in ___:
    count = count + x
```

(b) Mistake 1 Specification

This problem asks for the number of items in the list not the total of all the values in the list.

(c) MDF Feedback

Figure 1: Example of Feedback Specification

3.2 Detecting Mistakes

To provide feedback without human intervention, mistakes can be detected automatically using program analysis, unit tests, runtime violations, and similar tools. In our environment, we use four kinds of mistake detection. First, conventional runtime and syntax errors were detected using the existing execution infrastructure. Second, output checking determined if students met functional correctness. Third, an abstract interpreter checked certain generic mistakes (e.g., def/use errors such as "variables must be defined before they are read"). Fourth, a large set of patterns were prepared which could be matched against students' code. The code pattern matching problem is a variation on the Ordered Tree Inclusion Problem (adapted to ASTs) [17]. Where P is an instructor AST and T is a student AST, we define our Ordered Tree Inclusion variant as follows:

> Given labeled partially ordered trees P and T, can P be obtained by deleting nodes from T. Deleting a node u entails removing all edges incident to u and its descendants.

where a partially ordered tree in our context is defined as:

> A rooted tree in which the order of the subtrees is significant with the exception of nodes that have commutative properties (e.g. multiplication and addition) whose subtrees do not have side effects.

Experts can write patterns declaratively with corresponding feedback. A total of 88 mistakes were codified into patterns by the instructors. An example mistake pattern is shown in Figure 1b.

3.3 Feedback Delivery

While students worked on programming exercises in BlockPy [3], their code was analyzed to detect mistake patterns. When detected, a relevant feedback message was delivered (either on-demand when the code was executed or while the student was editing code) using BlockPy's built-in feedback mechanisms that are normally used to deliver runtime and output errors. We chose immediate feedback delivery because of its acquisitional efficiency for verbal knowledge and procedural skills [2, 6, 9, 11]. In developing the feedback message, the instructors aimed to explain the misconception found and the corresponding mistake's location, rather than how the student should fix the mistake. Multiple patterns might be found in the student code; per common practice, we deliver only one feedback message[4]. Figure 1c shows an example of delivered Misconception-Driven Feedback. Feedback delivery did not require modifications to the BlockPy interface.

4 EXPERIMENTAL DESIGN

We hypothesized that Misconception-Driven Feedback would improve student performance on near-transfer tasks, even if MDF was not provided during programming assessments. We also expected that students who received MDF while learning would see gains in their conceptual understanding of the topics. To test these hypotheses, we conducted a quasi-experimental study in an introductory, undergraduate Computational Thinking course for non-majors.

4.1 Class Description

The study was conducted at Virginia Tech, a large public university located in a rural area of the eastern United States. At Virginia Tech, students completing an undergraduate degree in any major must satisfy a set of "general education" requirements by completing designated courses in several broad areas of study. For example, in the area of "quantitative reasoning" students must complete three courses from an approved list of courses in mathematics, computer science, logic, or similar subjects. The computational thinking course in this study is typically used by students in non-STEM majors to satisfy the quantitative reasoning requirement.

The study collected data over three consecutive semesters. The baseline data for the control group was collected in the spring (January-May) term of 2017. Comparative data for the treatment groups was collected in the fall (August-December) term of 2017 and the spring term of 2018. Two sections of the course were taught each semester, each meeting twice a week for 75 minutes. The classroom environment and the sections' weekly schedule remained the same, though the time of day varied. The data was collected under an IRB-approved protocol.

4.2 Demographics

Tables 1 through 3 show demographic information about the students whose data is reported in the study. The enrollment in each semester for each instructor is shown in Table 1. Enrollment in the two treatment semesters was limited by the classroom size.

Semester	S2017	F2017	S2018	Combined
Instructor 1	47 (13%)	61(17%)	66 (19%)	174 (49%)
Instructor 2	47 (13%)	64 (18%)	67 (19%)	180 (51%)
Total	94 (27%)	125 (36%)	133 (38%)	352 (100%)

Table 1: Enrollment Statistics

Table 2 shows the gender and class of students in the study. Students in the study were approximately gender balanced. Also, there were significant numbers of student from each of the four years of study (Freshman through Senior). It is common in general education classes to see this diversity among years, because the class does not serve as a pre-requisite to other courses.

Gender	Class
Female: 172 (49%) Male: 180 (51%)	Freshman: 103 (29%)
	Sophomore: 118 (34%)
	Junior: 74 (21%)
	Senior: 57 (16%)

Table 2: Gender and Class Demographics

Each section in each semester included students from a variety of majors as summarized in Table 3. University Studies and General Engineering are students who have not yet selected a specific major. Building Construction was particularly prevalent because this course is a major requirement. There are 47 other majors that account for the remaining 138 (39%) students. Students self-select to enroll in the class; the instructors have no influence over the students who enroll or in which section.

Major	Population
Building Construction	52 (15%)
Criminology	32 (9%)
Psychology	31 (9%)
University Studies	26 (7%)
Fashion Merchandise and Design	21 (6%)
International Studies	14 (4%)
Statistics	14 (4%)
Political Science	14 (4%)
General Engineering	10 (3%)

Table 3: Major Demographics

4.3 Curriculum and Learning Environment

The curriculum, pedagogy, and technology for the computational thinking course evolved over a period of five (5) semesters and was stable by the start of the study. Details of the curriculum can be found at https://think.cs.vt.edu/ct and in [16]. The major technical topics in the curriculum were data abstraction and algorithms. Throughout the study, the curriculum's resources (readings, assignments, projects, presentation materials, grading scale, etc.) remained fixed with changes limited to correcting typographical mistakes or minor ambiguities. The pedagogy for the course included both active learning and peer learning. Students were organized by the instructors into four person teams that persisted throughout the semester. Groups were formed to maximize diversity of majors and balance gender within each group. Each class day students were engaged in solving problems individually, but encouraged to seek and provide help to others in their group. This group model was used in all semesters of the study. The technology included a learning management system (Canvas), an environment for block-based programming (BlockPy), and a standard Python environment (Spyder). These systems were used throughout the study and no significant changes in functionality were introduced.

The course staff consisted of two instructors, one graduate assistant, and ten undergraduate assistants (UTAs). The UTAs had completed the course in previous semesters and attended each class. The instructors and GTA were the same throughout the study. The UTAs varied during the study. A staff meeting was held each week to provide guidance and coordination for the UTAs. For the control group the course staff were proactive in providing in-class assistance. In the treatment classes during the period when the feedback intervention was applied the course staff was reactive, only providing assistance when explicitly called upon. Anecdotally, there was a noticeable decline in the support provided by the course staff in the two treatment semesters. The instruction using the block-based language covered a two week period (classes 7-10) with another class (class 12) devoted to a programming project. The topic of list-based iteration was the focus of classes 8-10. One class (class 13) was a transition from blocks to text.

The feedback intervention was focused on the instruction related to iteration (classes 8-10) and a project (class 12). This topic was chosen because it was the first difficult programming concept in the course, was the focus of a lengthy period of instruction, and in prior semesters was known by the instructors to be a time where students struggled, requiring much assistance from the course staff.

4.4 Assessment

The impact of the feedback intervention was measured through one pre-test and two post-tests (administered electronically). The pre-test was administered at the end of the class preceding the start of the iteration unit. The first post-test was administered at the beginning of class 10. This "embedded" post-test came after two classes of instruction and was approximately the mid-point of the instructional unit. The second post-test was administered at the end of the students' work in BlockPy (class 12).

A common element in the pre-test and post-tests was a nine question multiple-choice quiz. The quiz was administered through the learning management system (Canvas). Each question was presented one at a time without the option to return to a previous question. The questions were developed through an instructional design process. A pair of learning objectives were defined for the iteration unit. One learning objective was the ability to write a program using iteration to compute a quantitative measure from a list of data. The other learning objective was to write a program using iteration to produce a visualization from a list of data. Instructional analysis was used to identify the sub-tasks needed to achieve each objective. Multiple performance objectives were created for each task from which the nine multiple choice questions were created. The distractors on each question came from mistakes identified by an analysis of student solutions to iteration problems in the prior semester. Figure 2 shows an example of one of the questions on the pre-test. In this question students were shown a fragment of a block-based program containing a list-based iteration and asked to identify which one of seven alternatives should be placed in the body of the iteration. The figure also shows the most frequently selected distractor (item e) and the correct choice (item g).

The same concepts were tested on the pre-test and post-test quizzes. While the question ordering remained the same, the question wording varied. For example, pre-test question 4 shown in Figure 2 appeared on the first post-test using *distance_sum*, *distance*, and *distance_list* instead of the text shown (the problem changes

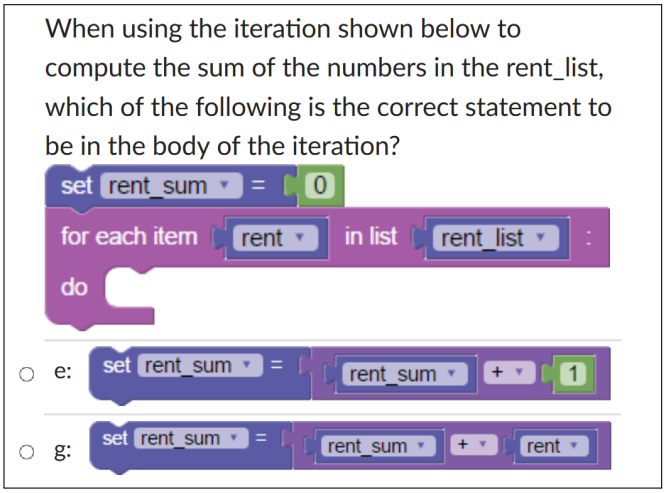

Figure 2: Example Pre-test Question

from one about rents to a problem about distances). However, the essential nature of the question remained the same.

In addition to the multiple-choice test, each post-test contained one or two open-ended programming problems. The first post-test contained one programming problem related to the first learning objective (computing a quantitative measure). The material related to the second learning objective (producing a visualization) was not covered until after the first post-test. The second post-test contained two programming problems - each related to one of the learning objectives. The statement of the three programming problems are:

(1) The data block in the BlockPy canvas below provides a list of the number of students taking the 2015 SAT test in each state. Write an algorithm to compute and print the total number of students taking the SAT test in 2015.
(2) The data block in the BlockPy canvas below provides a list of the per capita income of each state. Write an algorithm that computes and prints the number of states with a per capita income greater than 28000 dollars.
(3) The data block in the BlockPy canvas below provides a list of the sale price in US dollars of books sold by Amazon. Write an algorithm that produces a histogram of sale prices in Euros. A dollar amount is converted to a Euro amount by multiplying the dollar amount by 0.94.

The first problem requires an iteration that counts the number of elements in the list. The second problem requires an iteration that counts only some of the elements in the list. The third problem requires an iteration that produces a new list with transformed values; this list is then visualized as a histogram.

It is important to note that during the programming of the open-ended questions, the students were provided limited feedback: only run-time errors were reported (e.g., adding a number to a list or using an uninitialized variable). The feedback was limited to assess the extent that students had internalized the knowledge and ability to write the program without relying on the feedback for guidance. Each question clearly explained that there would be limited feedback and no indication of correct output would be given.

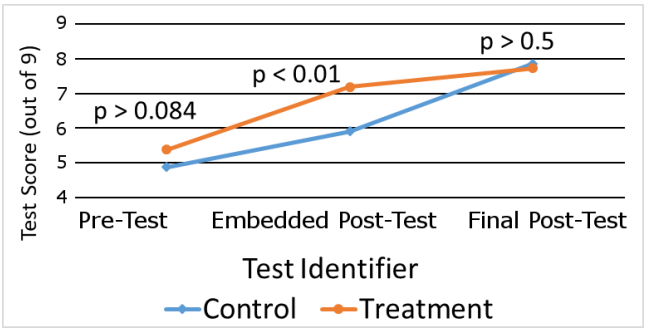

Figure 3: Comparison of Score Over Time Between Control and Treatment

5 RESULTS AND ANALYSIS

In this section, the data from the multiple choice tests and free response (programming) problems are analyzed. Each of the nine questions on the multiple choice test is scored as correct or incorrect. A student's score is the total number of correct answers given. Each free response question is also scored as correct or incorrect. Correct means that the student's program produced the expected output exactly; all other programs are incorrect. A Mann-Whitney U test is used to measure the statistical significance of differences between the performance of students in the treatment group in comparison to the performance of students in the control group. The Mann-Whitney U test is appropriate for this data because the responses are ordinal and we cannot satisfy an assumption of normality.

The tables presented in this section contain four items:

$\bar{x}$: the mean response for the given group
s: the standard deviation from the mean
n: the size of the group
p: the significance
r: the effect size expressed in variance

A p value less than .05 is taken as significant. Modified r for non-parametric effect size is taken as per normal for variance effect size (see [8]). Each table also shows four groups: the control group, the first treatment group (Fall 2017), the second treatment group (Spring 2018), and the two treatment groups combined (All).

5.1 Multiple Choice Tests

First, we compare the pre-test performance of the treatment groups to that of the control group. If the performance of the treatment group is significantly different from that of the control group then the groups are not directly comparable (e.g., students in one group might have a higher level of prior programming experience than students in the other group).

The Pre-test row in Table 4 shows the comparison of the pre-test performance between the groups. There was no significant difference between the control group and the combined treatment groups (All) and no significant difference between the control group and the first treatment group. However, the null hypothesis is not rejected for the second treatment group. Therefore, comparisons to assess the impact of the feedback intervention use the combined (All) treatment group in comparison to the control group.

The impact of the feedback intervention is shown in the two Post-test rows of Table 4. The Embedded Post-test data shows that there is a significant difference between the control group's mean performance ($\overline{x}$ = 5.9067) and the treatment group's mean performance ($\overline{x}$ = 7.1878). On average, the combined treatment group performed better by the equivalent of one full question on the test. Statistically, this difference is a large effect size. Recall that the embedded post-test occurs midway through the instruction. However, the Final Post-test row in Table 4 shows that there is no significant difference between the treatment and control groups. The mean response data ($\overline{x}$) is also shown in Figure 3.

Finally, considering each column in Table 4 separately, the gain in student learning within each group can be seen. The mean response in all treatment groups shows similar improvement.

Assessment	Control ($\overline{x}$,n)	Treatment ($\overline{x}$,n,p,r)		
	S2017	F2017	S2018	All
Pre-test	$\overline{x}$ = 4.8806 s = 2.1499 n = 67	$\overline{x}$ = 5.1982 s 1.9485 n = 111 p = 0.2970 r = 0.0783	$\overline{x}$ = 5.5536 s = 2.0961 n = 112 p = 0.0396 r = 0.1539	$\overline{x}$ = 5.3767 s = 2.0272 n = 223 p = 0.0845 r = 0.1013
Embedded Post-test	$\overline{x}$ = 5.9067 s = 1.2646 n = 75	$\overline{x}$ = 7.1500 s = 1.6291 n = 100 p <0.0001 r = 0.4148	$\overline{x}$ = 7.2212 s = 1.5853 n = 113 p <0.0001 r = 0.4171	$\overline{x}$ = 7.1878 s = 1.6026 n = 213 p <0.0001 r = 0.3717
Final Post-Test	$\overline{x}$ = 7.8611 s = 1.3972 n = 72	$\overline{x}$ = 7.7300 s = 1.5561 n = 100 p = 0.6443 r = 0.0353	$\overline{x}$ = 7.7232 s = 1.4900 n = 112 p = 0.4852 r = 0.0516	$\overline{x}$ = 7.7264 s = 1.5179 n = 212 p = 0.5121 r = 0.0390

Table 4: Student performance on multiple choice assessment

5.2 Free Response Tests

The free response tests consisted of individual programming problems as described in Section 4. The Embedded Post-test had one programming problem and the Final Post-test had two programming problems. As described in Section 4.4 the students were told that they would receive limited feedback on these problems. The results of the free response tests are shown in Table 5 and Table 6.

Table 5 shows the analysis of all three programming problems grouped together (row 1) and the two Final Post-test programming problems grouped together (row 2). Overall, there was a significant difference between the control and the combined treatment groups (All), with a small effect size. This suggests that MDF supports their acquisition of programming skills. On average, 32% of the control group completed all three problems correctly versus 41% of the treatment group. A similar result occurs when considering only the two Final Post-test questions. In this case 38% of the control group completed the two programming problems correctly compared to 48% in the combined treatment groups.

Table 6 shows the analysis for each of the three problems separately. Although similar differences and effect sizes between the treatment and control groups are reported in Table 6 as in Table 5, the sample size is not large enough to claim statistical significance. It can be noted that the effect size is consistent despite the increasing complexity of the problems. The last Post-test problem requires list construction, list appending, and plotting that are not part of the other problems. However, the same difference between control and treatment is seen for this problem as for the others.

5.3 Discussion

The data in Table 4 and summarized in Figure 3 indicates that there is an accelerated level of learning in the treatment over the control group at the point of the embedded post-test, but that by the end of the instruction the control group has closed the gap with the treatment groups. Two interpretations of this data are possible. One interpretation is that additional practice (classwork and homework problems) and additional interaction with the course staff (in class and during office hours) can compensate over time for the lack of more effective feedback. Even with this interpretation the potential demotivating impact on students of more failed attempts and the need for more staff interaction should be kept in mind. Anecdotally, there was a highly noticeable decline in the need for interaction with the course staff during the two treatment semesters. A second interpretation is that the quality of the feedback in the first part of the instruction is better than that during the later part of the instruction. Our analysis of student solutions for the later part of the instruction should be revisited in this light. If the feedback can be improved, it is possible that the performance gap might persist.

On the programming tasks, the feedback intervention helped to improve the level of success by about 10%. We believe that with refinement, this degree of improvement can be increased.

Another impact of the feedback intervention can only be reported anecdotally. In the case of the control group the course staff played a proactive role in probing students' progress and offering help. This included undergraduate teaching assistants who were paired with fixed groups of students in a ratio of approximately 16:1. In contrast, during both interventions the course staff played a reactive role, only providing assistance when explicitly asked for by a student. The course instructors noted a dramatic drop in the level of help required of the course staff. This is especially significant in the light of the students' increased performance on the post-tests.

Another observation is that while the treatment group performed equivalently on conceptual questions (multiple choice test) compared to the control group by the end of the intervention, the treatment group performed better on the free response questions. It is positive that the feedback intervention did support a higher level of students' programming ability. However, this improvement in programming ability did not seem to be associated with an improvement on the multiple-choice test. One interpretation is that the feedback intervention occurred in the context of free-response questions similar to the questions on the post-tests. Thus, the effect of the improved feedback transferred better to a similar, constructive task, but not to the recall and analysis tasks of the multiple-choice test. However, this explanation is not completely satisfactory because the embedded post-test seemed to indicate that there was a positive impact on the multiple-choice tests earlier. Another interpretation is that after being given the same multiple

choice test three different times (albeit, contextualized differently) the students may have "learned" how to answer the questions. Additional analysis will be needed to resolve this question.

Problems	Control ($\overline{x}$,n)	Treatment ($\overline{x}$,n,p,r)		
	S2017	F2017	S2018	All
All Post-Test Free Response	$\overline{x}$ = 0.3227 s = 0.2781 n = 94	$\overline{x}$ = 0.3920 s = 0.3115 n = 125 p = 0.0919 r = 0.1140	$\overline{x}$ = 0.4987 s = 0.3585 n = 133 p = 0.0001 r = 0.2549	$\overline{x}$ = 0.4137 s = 0.3078 n = 258 p = 0.0127 r = 0.1330
Final Post-Test Free Response	$\overline{x}$ = 0.3855 s = 0.3648 n = 83	$\overline{x}$ = 0.4858 s = 0.3544 n = 106 p = 0.0498 r = 0.1428	$\overline{x}$ = 0.4835 s = 0.3707 n = 121 p = 0.0622 r = 0.1307	$\overline{x}$ = 0.4845 s = 0.3636 n = 227 p = 0.0316 r = 0.1224

Table 5: Cumulative Student Performance on Post-Test Programming problems

Problems	Control ($\overline{x}$,n)	Treatment ($\overline{x}$,n,p,r)		
	S2017	F2017	S2018	All
Embedded Post-Test Free Response	$\overline{x}$ = 0.3253 s = 0.4948 n = 83	$\overline{x}$ = 0.4259 s = 0.3976 n = 108 p = 0.1576 r = 0.1024	$\overline{x}$ = 0.4463 s = 0.3873 n = 121 p = 0.0838 r = 0.1212	$\overline{x}$ = 0.4367 s = 0.3914 n = 229 p = 0.0774 r = 0.1000
Final Post-Test Free Response 1	$\overline{x}$ = 0.3780 s = 0.4934 n = 82	$\overline{x}$ = 0.4571 s = 0.5013 n = 105 p = 0.2792 r = 0.0792	$\overline{x}$ = 0.4793 s = 0.5019 n = 121 p = 0.1550 r = 0.0999	$\overline{x}$ = 0.4709 s = 0.5010 n = 226 p = 0.1565 r = 0.0808
Final Post-Test Free Response 2	$\overline{x}$ = 0.4125 s = 0.5006 n = 80	$\overline{x}$ = 0.5294 s = 0.5016 n = 102 p = 0.1185 r = 0.1158	$\overline{x}$ = 0.5268 s = 0.5010 n = 112 p = 0.1194 r = 0.1125	$\overline{x}$ = 0.5280 s = 0.5001 n = 214 p = 0.0785 r = 0.1027

Table 6: Student Performance on Individual Post-Test Programming Problems

5.4 Mistake Driven Analysis

The previous sections have established that Misconception-Driven Feedback supports acquisition of student programming skills and conceptual knowledge. This section focuses on the particular role that misconceptions play in deeply analyzing student programming problems. Recall from Section 3.1, that a mistake is a vector of misconceptions. By cross-referencing mistake vectors we mean identifying the misconceptions common to the cross-referenced mistakes. Through cross-referencing we can isolate misconceptions. To highlight the power of this approach, we present some examples below. Specifically, we divide the discussion into three parts. The first part demonstrates how MDF can be used to more deeply analyze mistakes. The second part demonstrates how new misconceptions can be discovered by using MDF. The third part demonstrates how MDF can be used to understand and reason about the impact of the feedback on students.

To illustrate this approach, we analyze mistakes made by students that were identified in the second free response problem:

> The data block in the BlockPy canvas below provides a list of the per capita income of each state. Write an algorithm that computes and prints the number of states with a per capita income greater than 28000 dollars.

This problem asks a student to count (the number of states) and filter (include only a portion of the data) using iteration.

5.4.1 Deeper Analysis. One mistake we observe in this problem is the absence of the pattern seen in Figure 4. This pattern indicates

```
#absence of:
for _item_ in _list_:
    count = count + 1
```

Figure 4: Mistake Example 2

that a student is missing the statement to count the items in the list. In the treatment group, 43.81% exhibited this mistake, while 57.32% of the control group exhibited this mistake. While this is an improvement over the control group, MDF can facilitate a deeper analysis. Mistake 2 is indicative of multiple possible misconceptions, including but not limited to:

- The student does not understand the difference between summing and counting.
- The student does not understand the difference between an accumulator (count) vs. the iteration variable (item).
- The student does not understand that the iteration variable (item) takes on each value of the list (list).

In the treatment group, 99 of 226 treatment solutions made mistake 2. To more deeply analyze this mistake, we also detect the co-occurence of another mistake: the presence of the code pattern shown in Figure 1b. By cross-referencing these two mistakes, we increase the evidence that the student has the misconception of not understanding the difference between summing and counting. However, this pairing of mistakes accounts for only 20 of the 99 occurrences of mistake 2, suggesting that the remaining 79 occurrences have to be one of the other misconceptions associated with mistake 2. By cross-referencing with other mistakes we might be able to isolate the frequency of the other two misconceptions.

5.4.2 Discovering New Misconceptions. Cross-referencing mistakes can also reveal new misconceptions. For example, in the above problem, 61 students had mistake 2 plus another anticipated mistake: a missing print statement. An inspection of the programs of these 61 students showed that 51 of them incorrectly used an append statement. This pairing of mistakes, made by a substantial number of students, indicates the existence of an unexpected misconception: confusing creating a list of items with counting the

number of these items. Although recognizable in retrospect, this misconception only emerged through the MDF analysis.

5.4.3 Understanding Anomalies. Finally, MDF can be used to diagnose anomalous results in student data. Consider mistake 3 shown in Figure 5, the absence of a necessary conditional check. The control group exhibited this mistake in 1.22%(1) of its population whereas the treatment condition exhibited mistake 3 in 12.83%(29) of its population. While overall performance of the treatment group was better than the control group with respect to the free response in general, this particular mistake contradicts the general result. Cross-referencing mistakes allows us a deeper understanding and suggests why the feedback had a negative impact on this mistake.

```
#presence of equivalent expression
if x > 28000:
    pass
```

Figure 5: Mistake Example 3

These three cases cross-reference mistake 3 with other mistakes:

Case 1 : income >= 28000 or income <= 28000 or income < 28000
- condition wrong and no other feedback

Case 2 : income > "28000" or income >= "28000"
- condition wrong, output wrong, and incompatible types

Case 3 : income > 2800 or income >= 2800
- condition wrong, and output wrong

The feedback associated with this mistake is "In this problem you should be finding XXX above/below XXX units", where XXX, above/below, and units are contextualized with specific problems).

Case 2 is interesting because students in the treatment group did not receive feedback about incompatible types (the runtime feedback the control received by default), because this message was superseded by the feedback associated with mistake 3. This case accounts for 38%(11) of the occurrences of mistake 3 in the treatment group. This particular error indicates an issue with students' awareness of operations on data types. This particular misconception coincides with results from our post-tests, as discussed in [12]. This suggests a failure of our feedback, contradicting our goal of grounding feedback in misconceptions. Our failure lay in choosing to provide feedback about the mistake (pointing to the incorrect comparison) rather than the underlying misconception (confusing the types of numbers vs. strings). Case 3, rather than being a misconception, is likely a typo or careless reading by the students. Case 3 captures 41%(12) of the occurrences of mistake 3. Of the 12 in Case 3, 11 were "income > 2800." This means of the 29 occurrences of mistake 3, only 7 of these were issues with conditionals. Correcting for these, misconceptions with conditionals parallels performance in the control group (1.22% vs 3.1%). These nuances demonstrate how MDF allows more critical analysis of free response data.

6 THREATS TO VALIDITY AND LIMITATIONS

While the results presented in this work are promising, there are several threats to validity. First, in the pretest for the control group, one of the class sections for the control had technical difficulties and took the quiz outside class. However, a Mann-Whitney U test between the two sections in the control showed no significant differences in pretest results. Second, feedback creation, misconception identification, and analysis were all done by the same researchers. Despite our efforts to be guided by the analysis of data and the principles of instructional design, it is possible that the perspectives of the team were too similar, allowing alternate explanations to be overlooked. Third, UTAs were instructed to be more passive in the treatment group as opposed to the active assistance they provided in the control group. This change may have required students to be more self-reliant in their learning even in the absence of the MDF feedback. However, one of our aims is to reduce UTA workload.

There are limitations in this work. First, our population was a specific demographic of non-technical majors. It is unknown how MDF would affect CS majors. Second, we only targeted a specific unit of instruction (collection-based iteration). Future work includes targeting different units of instruction. For both limitations, the specific misconceptions and feedback may change with different populations or topics. However, we do believe that the method itself would prove equally useful in other contexts.

7 CONCLUSIONS

In this paper we presented and assessed misconception-driven feedback (MDF). Informed by cognitive learning theory, the learner model underlying MDF focuses on the mental misconceptions that learners reveal by their observed mistakes. The philosophy of MDF is that immediate formative feedback about an observed mistake should address the underlying misconception. We evaluated MDF using both quantitative and descriptive analysis of a three semester quasi-experimental study. Quantitatively, there was a statistically significant, 10% increase in performance on open-ended programming problems and an acceleration in performance on a multiple-choice quiz. This improvement occurred despite the removal of MDF during the assessment and the deliberate reduction of proactive assistance from the course staff during the instruction. The cross-referencing of mistakes allowed us to reason about the underlying misconceptions in more precise way than would otherwise be possible. This analysis also revealed new misconceptions, allowing additional feedback to be created. The analysis enables instructors to have deeper insight into their learners and identify instructional improvement that will address persistent misconceptions.

In a broader context, MDF provides a novel and practical connection between learning and instruction about programming. Misconceptions result from cognitive breakdowns in learning. Mistakes, the manifestation of the breakdown in practice, point to areas of insufficient instruction or where the instruction should be reconsidered. This connection between the theory of learning and the pragmatics of instructional design offers not only an opportunity for more effective feedback but also the possibility of more coherent and integrated learning environments.

ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation grants DUE 1624320, DUE 1444094, and DGE 0822220.

REFERENCES

[1] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. 2004. An integrated theory of the mind. *Psychological review* 111, 4 (2004), 1036.

[2] John R Anderson, Frederick G Conrad, and Albert T Corbett. 1989. Skill acquisition and the LISP tutor. *Cognitive Science* 13, 4 (1989), 467–505.

[3] Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A Shaffer, and Dennis Kafura. 2017. BlockPy: An Open Access Data-Science Environment for Introductory Programmers. *Computer* 50, 5 (2017), 18–26.

[4] Brett A Becker. 2015. *An exploration of the effects of enhanced compiler error messages for computer programming novices.* Master's thesis. Dublin Institute of Technology.

[5] Benjamin S Bloom. 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher* 13, 6 (1984), 4–16.

[6] Gary M Brosvic and Beth D Cohen. 1988. The horizontal-vertical illusion and knowledge of results. *Perceptual and motor skills* 67, 2 (1988), 463–469.

[7] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. 2016. Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th ACM Technical Symposium on Computer Science Education*. ACM, 364–369.

[8] Jacob Cohen. 1988. Statistical power analysis for the behavioral sciences. 2nd.

[9] Albert T Corbett and John R Anderson. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 245–252.

[10] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*. ACM, 53–62.

[11] Roberta E Dihoff, Gary M Brosvic, and Michael L Epstein. 2003. The role of feedback during academic testing: The delay retention effect revisited. *The Psychological Record* 53, 4 (2003), 533–548.

[12] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, Jeremy Ernst, and Katherine Cennamo. 2018. Instructional Design+ Knowledge Components: A Systematic Method for Refining Instruction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 338–343.

[13] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D Nguyen. 2018. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 278–283.

[14] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. ACM, 89–98.

[15] Lisa Kaczmarczyk, Elizabeth Petrick, J Philip East, and Geoffrey L Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 107–111.

[16] Dennis Kafura, Austin Cory Bart, and Bushra Chowdhury. 2015. Design and Preliminary Results From a Computational Thinking Course. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. ACM, 63–68.

[17] Donald E Knuth. 1969. The art of computer programming. Vol. 1: Fundamental algorithms. Second printing.

[18] Kenneth R Koedinger, Vincent Aleven, Neil Heffernan, Bruce McLaren, and Matthew Hockenberry. 2004. Opening the door to non-programmers: Authoring intelligent tutor behavior by demonstration. In *International Conference on Intelligent Tutoring Systems*. Springer, 162–174.

[19] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. 2012. The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science* 36, 5 (2012), 757–798.

[20] Einari Kurvinen, Niko Hellgren, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. 2016. Programming misconceptions in an introductory level programming course exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 308–313.

[21] Nguyen-Thinh Le. 2016. A classification of adaptive feedback in educational systems for programming. *Systems* 4, 2 (2016), 22.

[22] David J Nicol and Debra Macfarlane-Dick. 2006. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education* 31, 2 (2006), 199–218.

[23] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969* (2015).

[24] Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating Data-driven Hints for Open-ended Programming.. In *EDM*. 191–198.

[25] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 483–488.

[26] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.

[27] Takayuki Sekiya and Kazunori Yamaguchi. 2013. Tracing quiz set to identify novices' programming misconceptions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. ACM, 87–95.

[28] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.

[29] Teemu Sirkiä and Juha Sorva. 2012. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. ACM, 19–28.

[30] Juha Sorva and Teemu Sirkiä. 2011. Context-sensitive guidance in the UUhistle program visualization system. In *Proceedings of the 6th Program Visualization Workshop (PVWâĂŹ11)*. 77–85.

[31] JC Spohrer and Elliot Soloway. 1986. Alternatives to construct-based programming misconceptions. In *Acm sigchi bulletin*, Vol. 17. ACM, 183–191.

[32] Marieke Thurlings, Marjan Vermeulen, Theo Bastiaens, and Sjef Stijnen. 2013. Understanding feedback: A learning theory perspective. *Educational Research Review* 9 (2013), 1–15.