Discrete Math with Programming: A Principled Approach

Yanhong A. Liu
liu@cs.stonybrook.edu
Stony Brook University, Stony Brook, New York

Matthew Castellana
matcastellan@cs.stonybrook.edu
Stony Brook University, Stony Brook, New York

ABSTRACT

Discrete mathematics is the foundation of computer science. It focuses on concepts and reasoning methods that are studied using math notations. It has long been argued that discrete math is better taught with programming, which takes concepts and computing methods and turns them into executable programs. What has been lacking is a principled approach that supports all central concepts of discrete math—especially predicate logic—and that directly and precisely connects math notations with executable programs.

This paper introduces such an approach. It is based on the use of a powerful language that extends the Python programming language with proper logic quantification ("for all" and "exists some"), as well as declarative set comprehension (also known as set builder) and aggregation (e.g., sum and product). Math and logical statements can be expressed precisely at a high level and be executed directly on a computer, encouraging declarative programming together with algorithmic programming. We describe the approach, detailed examples, experience in using it, and the lessons learned.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Mathematics of computing** → **Discrete mathematics**.

KEYWORDS

discrete math, language, logic, programming, sets, teaching

1 INTRODUCTION

Discrete mathematics is the foundation of computer science. The central concepts in it—from logic and reasoning, to sets and functions, to sequences and recursion, to relations and graphs—are essential mental tools for modeling real-world objects and developing programming solutions, whether for basic problem solving or for advanced software development.

At the same time, the core discipline of computer science is program development. It is taught in introductory programming

sequences, branching at upper levels to projects in courses such as databases, networking, security, and especially compilers that have to deal with sophisticated discrete structures for representing computer programs themselves.

As a result, the two primary courses in computer science are typically discrete math and program development, the two largest-by-hours areas in [33, p.37], but with very different course activities:

- The former teaches concepts and reasoning methods, with the help of paper and pencil or text formatting tools for writing math notations and natural language.
- The latter emphasizes transforming concepts and computing methods into executable programs, overcoming idiosyncratic issues of programming languages and systems.

Students can see some common concepts underlying both, e.g., sets and sequences in the former are realized as some collection types in the latter. However, there are no direct, precise connections between the two kinds of activities—the same concepts are studied in completely different contexts, and computations involving these concepts are expressed completely differently in math notations than in most programming languages, especially Java, the current dominant language used in teaching program development.

Clearly, the instruction of discrete math and programming should be integrated to connect theory with practice, to let them reinforce each other, and to help students better understand and master both. This in fact has been pursued, in a great deal of prior work and effort, as discussed in Section 2. What is lacking is a principled approach for doing this, an approach that covers all central concepts in discrete math and promotes disciplined uses of these concepts in problem specifications and programming.

This paper presents such a principled approach. The approach has four main features:

(1) It covers all central concepts in discrete math, especially including the fundamental, challenging topic of predicate logic.
(2) It is based on a powerful language with precise syntax and semantics, directly connecting math notations with programming language notations.
(3) It supports clear and precise specifications of problem statements using any combination of the language elements for all concepts.
(4) It promotes declarative expression of complex computation problems but also supports easy expression of algorithmic steps.

The language we use, referred to as DA in this paper, extends the Python programming language. It supports proper logic quantification, as well as declarative set comprehension (also known as set former and set builder) and aggregation (e.g., sum and product) over sets and sequences. Declarative specifications of problem statements are directly executable in Python with the module for DA extensions, just as algorithmic steps are.

This approach was used in teaching Foundations of Computer Science at Stony Brook University in Spring 2020, entirely as

extra-credit programming problems added to regular homework assignments. Our results and analysis of using this approach support broader deployment, and we give suggestions for future adoption.

2 RELATED WORK

Many methods for teaching discrete math have been studied since the early ages of computer science, e.g., [6, 25], especially those involving programming, e.g. [23].

Naturally, algorithms introduced in discrete math are frequently selected for implementation in a programming language. For example, students could be required to implement an algorithm of their choosing in any programming language [22]. Even an entire follow-up lab course could be developed, e.g., for students who are already familiar with languages such as Java [31]. However, these programming components do not require writing declarative, logical specifications that are central to discrete math.

There have been many uses of various declarative languages, not only functional and logic languages, but also the SQL database language. Examples include using FP, ML, and Prolog in a complementary fashion [11], using SML extensively to write recursive functions over lists for many set and logic operations [36], and using SQL to program with sets and relations and especially its EXISTS operator [26]. All of these efforts had to get around the lack of real universal and existential quantifiers.

Other approaches used dedicated logic and modeling systems. For example, an automated system could take a list of facts, and generate a list of support facts to give students insight into how first-order logic works [24]. The Alloy modeling language [37] is excellent for writing specifications using sets, relations, and predicate logic and then finding models that satisfy a specification [34], although it suffers from issues with recursion. The powerful SMT solver Z3 was also used, for solving puzzles [12]. These are farther away from introductory programming.

There are also many studies of using supporting tools, especially visual tools and more powerful proof tools. For example, specialized programs were used for visualizing graphs and algorithms that do depth-first search, etc. [2], for learning rules in solving visual logic puzzles [5], for proof editing with helpful checks [3], and for giving meta-level support [21]. There were also efforts that encourage students to make their own tools, e.g., a proof checker of a natural deduction system, a database management system, a propositional logic proof system, and a symbolic execution engine [14].

Additionally, dedicated books have been written on the topic of teaching discrete math with a programming language, e.g., C [1] and Python [27, 32]. However, these do not cover writing logical and declarative specifications for the central topic of predicate logic, instead opting for writing iterations or recursions for traditional written exercises on the subject, or avoiding the subject altogether.

3 APPROACH

We first discuss all central topics that a principled approach must cover. We then describe the use of a powerful and precise language, the support for clear and precise problem specifications, the fostering of declarative as well as algorithmic programming, and the use of programming as an enhancement.

Central topics in discrete math. Discrete math is typically one of the first two courses in computer science, the other being introductory programming. Despite many textbooks written and used for the subject, the central topics are well-known, as captured in commonly used textbooks, e.g., [4, 7, 8, 28], as well as [33].

- Logic. This includes propositional logic, the more general predicate logic that includes quantifiers, and proof methods.
- Sets, functions, and relations. This includes definitions, operations, and properties over sets, functions, and relations.
- Sequences and recursion. This includes definitions of sequences, summation and product forms, recursive formulas, and proofs by induction.

Additional topics are often included, but they are generally special, expanded cases of the core topics. Typical such topics are:

- Graphs and trees. A graph is essentially a set of vertices plus a binary relation on the vertices. A tree is simply a graph where each vertex has one incoming edge and multiple outgoing edges forming no cycles.
- Counting and probability. Counting corresponds to the cardinality of sets of interest. Probability is essentially the count of elements of interest divided by the count of all elements.

All topics above include aspects of reasoning and proofs, as well as computations and algorithms. These aspects are often included in the expositions of the topics above, e.g., proofs with logic in [28] and both proofs and algorithms with many topics in [7, 8]. These aspects are sometimes also covered as separate topics, e.g., proofs in [4, 33] and algorithms in [28]. In the expositions of all these topics, examples and applications from number theory are often used [7, 28], and corresponding concepts such as Boolean algebra are often introduced [4, 7, 28].

Among all topics in discrete math, logic is typically viewed as the most fundamental topic—it is usually the first topic to study [4, 7, 28] and is also emphasized particularly as driving the entire subject [10]. Therefore, a principled approach for discrete math with programming must cover logic as well as all other topics.

Powerful and precise language. To cover all central topics, one can see the variety of discrete values, operations on them, and properties about them that must be expressed. To express all of them precisely and unambiguously, and to connect them with programming, a powerful language with precise syntax (i.e., forms) and semantics (i.e., meanings) is needed.

We describe the use of such a language that combines the advantages of the two kinds of languages used in introductory courses:

- Traditional mathematical notations used in discrete math. These notations are high-level and concise. However, they generally do not have formal semantics, and allow loose usage with no automated checking for syntax or semantics.
- Programming languages, such as the dominant language Java, used in introductory programming. These languages have precise syntax and semantics and are automatically checked for the syntax and executed following the semantics. However, they are lower-level, tedious, and verbose.

The language we use, DA, extends Python. Python is well-known for being significantly more concise and higher-level than languages like Java and C/C++, and is already widely used by scientists and

high-schoolers alike and taught to non-CS and novice students, demonstrating its power and ease of use.

Python already supports sets and sequences, comprehensions over them, and generator expressions with operators `all`, `any`, `sum`, `max`, etc. on top of commonly used loops and recursive functions for programming at a high level like pseudocode. The DA extensions we use support the following main language constructs that are not in Python, but are essential for expressing all central concepts clearly and directly.

- Proper universal and existential quantifications. These capture the exact meaning of quantified statements (that use universal quantifier ∀ and existential quantifier ∃) in predicate logic.
- Comprehensions over sets and sequences with logic/pattern variables. These correspond to set builder notations for forming expressions over sets, relations, and sequences.
- Aggregations over sets and sequences with logic/pattern variables. These are similar to comprehensions but support summation, product, counting, maximum, and minimum.

The quantification forms are built on the best previous languages that support quantified expressions, SETL [13, 30], designed exactly as a set-theoretic programming language, and ABC [9], designed exactly to teach introductory programming.[1] These are discussed in detail in Section 4.

Use of what we call logic variables, or pattern variables, in comprehension and aggregation, as well as quantification, was motivated by a history of informal use in writing declarative set expressions, e.g., [15, 19, 20], which led to its precise formalization with patterns by Liu et al [17, 18]. These are discussed in Section 5.

Together, the extended language DA supports simpler and clearer problem specification as well as expression of computations and algorithms. In particular, logical statements about mathematical concepts can be directly executed for computation and checking, unlike math notations on paper that are disjoint from low-level programs executed on computers.

The only aspect not supported in DA is formal development of complete proofs, but such proof development is well-known to be challenging even for the best experts. Support for easier writing of proofs remains a direction for future study [29].

Clear and precise problem specification. With a powerful language for expressing logical statements, problem specifications can be written more easily and clearly. Whether for computations or for proofs, precise problem specification is the most critical task. The language we use supports such specification for any aspect that needs it.

- Input specification. This specifies all sets, functions, etc. given, plus logical statements specifying additional relationships among the given structures.
- Output specification. This similarly specifies the sets, numbers, etc. to be produced, plus logical statements specifying how the output is related to the input.
- Auxiliary value specification. This specifies auxiliary sets, functions, etc. to use, plus logical statements relating them to input, output, and to each other.

The language constructs we use for quantification, comprehension, and aggregation can be arbitrarily combined in writing logical statements. Section 4 gives example specifications.

Declarative as well as algorithmic programming. Dominant programming languages such as Java and C/C++ do not support quantification, comprehension, or aggregation, so they must be programmed using iteration or recursion. For commonplace programming taught in courses and used in practice, the most important language constructs for expressing computations are iterations carried out in loops, where assignments and conditions are used to set and test values for starting and ending the loop and to update variables in between.

Languages like Python with DA extensions support quantification, comprehension, and aggregation as built-ins, which are compiled into loops automatically for execution. When a specification specifies an output using these constructs, which can be executed automatically, the specification serves as a good example of higher-level, declarative programming. Section 4 gives examples.

Programming as enhancement. When teaching discrete math with programming in the DA language, all discrete structures, operations on them, and properties about them can be expressed directly and precisely, and then executed directly. The programming part is exactly to write these precisely and directly.

To ensure that students still learn and build their full skills at least as well as they learn discrete math without programming, we used three general methods: (1) assign the programming part several days later than the non-programming part, (2) give in-class exercises for doing the most important of these problems before the programming part was assigned, and (3) assign more or larger problems for the programming part so as to benefit more from automatic execution.

Method (1) allows students to do homework problems first in their head. Method (2) forces students to do, or at least try, homework problems earlier. Method (3) helps show that programming is an enhancement.

4 PREDICATE LOGIC WITH PROGRAMMING

The topic on logic starts with propositional logic. It is about using propositions and logical operators. Programming with these is straightforward.

- Propositional logical statements can be directly and precisely written in commonly-used programming languages: use program variables for propositions, and use Boolean operators for conjunction, etc.
- A small issue is that implication (e.g., `p implies q`) is not in common programming languages, but it can be easily defined or directly written using its equivalent form with negation and disjunction (i.e., `not p or q`).

Predicate logic extends propositional logic to include the use of predicates with arguments and quantifiers quantifying over values of these arguments. It is significantly more sophisticated and cannot be expressed directly in most programming languages, not even Prolog. Our approach for teaching predicate logic with programming is as follows: (1) introduce the language, emphasizing

---

[1]In fact, SETL is one of the earliest and most powerful programming languages. ABC is a descendant of SETL, and Python is a descendant of ABC and C [35].

Logic and practice of programming

- CS: <u>Logic/math</u> versus <u>practice</u> of programming

- Practice: Python has become the most widely used language, by both the least experienced and the most experienced. C...

- Python's inspiration and predecessor was ABC --- for beginners; Python was also influenced by C --- system programming.

- ABC was inspired by SETL--- based on mathematical theory of sets.

- Logic and practice should be together --- much easier and simpler, providing much more assurance, and fun!   ...DistAlgo, Alda

Precise quantifiers each and some

- Universal quantification   • Existential quantification

```
∀ x ∈ S  ,   P(x)        ∃ x ∈ S s.t. P(x)     textbook
∀ x ∈ S  |   P(x)        ∃ x ∈ S  ,   P(x)     others
∀ x ∈ S  :   P(x)            ...      |
    ...                          ...
each x in S  has  P(x)   some x in S  has  P(x)   ABC,ideal
each(x in S, has= P(x))  some(x in S, has= P(x))  da in py

all( P(x)  for x in S )  any( P(x)  for x in S )  py
forall x in S  |  P(x)   exists x in S  |  P(x)   setl
```

Figure 1: Slides introducing precise quantifications and relationships.

quantifications, (2) write specifications, especially through examples, and (3) execute the specifications, directly in Python with DA extensions.

Language. Logical operators in Python are used, because they are simple and easy to read: and for conjunction ($\land$ in math notation), or for disjunction ($\lor$), and not for negation ($\sim$ or $\neg$).

Quantifications, for writing statements with universal and existential quantifiers ($\forall$ and $\exists$, respectively), are first discussed in class as usual. Then, only two additional slides, shown in Figure 1, are discussed.

The first slide gives an overview relating logic and practice of programming, including programming languages related to Python. In particular, Python and C are widely-used languages in practice, and Python's roots ABC and SETL were meant for beginners and actually based on set theory, respectively. It ends with the name of the language to be used, DistAlgo [17, 18], plus a tentative new name, Alda, for it, abbreviated for both as DA in this paper.

The second slide shows the precise language constructs for quantifications: the universal quantification means that each element x in set S has property P(x), and the existential quantification means that some element x in S has property P(x).

- The first line with $\forall$ and $\exists$ is from the textbook [7, 8] used for the course. The second and third lines show a few other math notations.
- The first line with each and some is from ABC and ideal for reading. The next line is the form in DA as implemented in Python.
- The last two lines show the forms in Python and SETL.

All ABC, DA, and SETL forms match the math notations better than the Python form. More critically, the constructs in ABC, DA, SETL, and informally in math notations—but not in Python—also give a witness:

> When the existential quantification is true, variable x is bound to a value in set S that makes P(x) true.

This powerful feature is important for expressing search using math and logic at a high level [13, 30].

We see that the second slide directly and precisely connects the many different math notations with the best, easy-to-read programming language constructs.

No more study of Python or DA was done in class, for three reasons. (1) Time diverted from teaching all regularly taught materials should be minimized. (2) The homework gave program files that contained examples. (3) We were confident in the power and ease of Python and DA extensions from past teaching experience.

Specification. We show the use of DA quantifications in specifying examples with different combinations. Two main examples are used in the textbook: a college cafeteria with students choosing items at different stations, and Tarski's world as a grid of blocks of various colors and shapes. We name them cafe and tarski, respectively, and use parts of cafe as examples.

For cafe, the textbook provides a figure with example students, food stations, items in those stations, and the items each student chose. It then lists four statements in math notations and discusses their truth values in English. The first example has:

> $\exists$ an item I such that $\forall$ students S, S chose I.
> "There is an item that was chosen by every student."

We write the corresponding precise statement as an example in the program file given to students:

```
some(I in items, has= each(S in students, has= chose(S,I)))
```

The homework then asks that several statements written in math notations or English be written in DA. For example, an exercise problem in the textbook was used in the homework, asking for the truth values of a list of statements. The first statement is:

> $\forall$ students S, $\exists$ a dessert D such that S chose D

The expected answer is:

```
each(S in students, has= some(D in desserts, has= chose(S,D)))
```

In total, in the programming part on predicate logic, 5 statements for cafe and 2 for tarski were asked, all involving nested alternating quantifiers as the example above, and some also involving and, or, and not.

Execution. The most practically attractive aspect in teaching discrete math with programming is that programs are executable to give the specified meaning. We show execution of the statements written, using the cafe example.

To execute a statement for cafe, the sets of stations, items, and students with their choices for predicate chose must be given. The given program file for cafe contains the following definitions, followed by example quantifications as discussed earlier, plus printing of the values of the quantifications.

```
1  # page 87 of textbook: Example 3.3.3.
2  # given knowledge from the first paragraph and shown in Figure 3.3.2:
3  salads = {'green salad', 'fruit salad'}
```

```
 4   main_courses = {'spaghetti', 'fish'}
 5   desserts = {'pie', 'cake'}
 6   beverages = {'milk', 'soda', 'coffee'}
 7   stations = [salads, main_courses, desserts, beverages]
 8   choices = {
 9     'Uta': {'green salad', 'spaghetti', 'pie', 'milk'},
10     'Tim': {'fruit salad', 'fish', 'pie', 'cake', 'milk', 'coffee'},
11     'Yuen': {'spaghetti', 'fish', 'pie', 'soda'} }
12   students = choices.keys()

13   # helper set and function, to capture English more easily:
14   items = setof(item, sta in stations, item in sta)
15   def chose(student, item): return item in choices[student]
```

Later extra-credit programming assignments gradually include writing more or all of such sets and definitions by students. Later assignments also tell students to use witnesses, e.g., print the value of I in the first precise statement earlier in this section.

The resulting programs as well as the given programs can be executed directly in Python with the module for DA extensions. Students are given two files, `cafe.da` and `tarski.da`, and two commands to run, `python -m pip install pyDistAlgo` for installing DA, and `python -m da cafe.da` for running the cafe example.

Programming after thinking and preparing. Programming should be an enhancement to traditional discrete math coursework, while minimizing the chores of working with a computer system. The homework instructions try to accomplish this. For the predicate logic part, there are two prerequisites to doing extra-credit programming.

First, for the `cafe` problem, students were given in-class exercises to do the exercise problem in the textbook that will be used in the programming part, before the programming part was posted. This requires them to think about the problem more before programming. Only 3 of the 5 statements for `cafe` were given for the written part of the homework.

Second, students should install Python and be able to run it on a command line. This was not needed if students used machines in the computer lab.

5 OTHER TOPICS

The same approach was also used in teaching all other topics. We highlight the power of DA extensions to facilitate this.

Language. Topics other than predicate logic require simpler language constructs, and no dedicated slides are used—only one or two slides for each topic with some notes on the side. Figure 2 shows two examples. Most such slides only have one or two lines of notes on them; these slides are the exceptions.

The first slide shows set operations and properties, all of which can be directly written in DA. The notes added on the bottom-right use set intersection as an example, showing it is in given file sets.da, and giving the exact comprehension constructs for expressing it in DA, in Python (thus in DA too, as DA extends Python), and in an ideal notation.

- The comprehension in DA means the set of x satisfying membership clauses x in A and x in B. In general, any membership and Boolean conditions can be written; the semantics of DA automatically avoids the well-known Russell's paradox.
- In particular, x is a logic variable, meaning that different occurrences of it in a comprehension automatically have the same

Properties of sets

- Inclusion of intersection: $A \cap B \subseteq A$ and $A \cap B \subseteq B$
- Inclusion in union: $A \subseteq A \cup B$ and $B \subseteq A \cup B$
- Transitivity of subset: $A \subseteq B$ and $B \subseteq C \rightarrow A \subseteq C$

- Set operations: logical definitions (textbook calls them procedural)
  - $x \in A \cup B \Leftrightarrow x \in A$ or $x \in B$
  - $x \in A \cap B \Leftrightarrow x \in A$ and $x \in B$ in given file sets.da
  - $x \in B - A \Leftrightarrow x \in B$ and $x \notin A$
  - $x \in A^c$ $\Leftrightarrow x \notin A$
  - $(x, y) \in A \times B \Leftrightarrow x \in A$ and $y \in B$

```
setof(x, x in A, x in B)  da
{x for x in A if x in B}  da/py
{x: x in A, x in B}       da ideal
```

Sequences in computer programming

- Array: a[1], a[2], …, A[50]   a = [7,4,25,9] list in py/da
- **for** $i := 1$ **to** n for i in range(1,n+1):   ints(1,n) da
       print $a[i]$ print (a[i])
  next i

- Summation
  $s := a[1]$ s = a[1]
  for $k := 2$ **to** n for k in …
      $s := s + a[k]$
  next k

           s = sum(a[k] for k in range(1,n+1))
           s = sumof(a[k], k in ints(1,n)) da

Figure 2: Slides on set properties and operations, and on programming with sequences.

value. So either order of the two membership clauses has the same meaning, and the DA compiler can decide how to implement them efficiently.

It is critical to note that the comprehension in Python, with for followed by if, is different (and so are comprehensions in SETL): it means to iterate over elements in A and, for each element x, if it is also in B, put it in the resulting set.

Aggregations such as sum and product over sets and sequences can also be expressed easily in DA. For example, given a set of sequence S, sumof(x, x in S) means the sum of all elements of S.

The second slide shows pseudocode for programming with sequences. The notes added on the right show corresponding constructs in Python, in green, and alternatives in DA, in blue. In particular, for computing summation, a for-loop block can be programmed as a 1-line aggregation for sum, in both Python and DA.

In general, Python is both powerful and easy to use and is well-known to be close to pseudocode when used to program algorithms. Comprehensions and aggregations in DA improve over those in Python by being completely declarative, exactly as in math.

Specification. Using more powerful constructs in DA, specifications of problem statements and computations for other topics can also be written easily and precisely as for predicate logic, and be made executable in Python.

Dozens of extra-credit programming problems were given on expressing operations on and properties of sets, sequences, functions, and relations, e.g., writing operations on sequences using
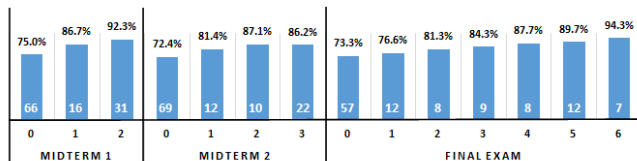
Figure 3: Exam performance vs. number of relevant programming assignment submissions. The number of students in each category is shown on the bottom of the bar.

both aggregations and recursive definitions; writing definitions of 1-1 and onto functions and using them to check given functions; writing definitions of reflexivity, symmetry, and transitivity and using them to check given relations; writing Euclid algorithm using both iteration and recursion; writing recursion for Hanoi Tower; and expressing transitive closure with an existential quantification with witness in a `while` loop.

Execution. All specifications in Python and DA can be executed directly as discussed in Section 4. The main difference is that, for later topics, students are asked to write more parts or even all parts of the solution to a problem on their own.

Programming as enhancement. Besides similar use of programming as for predicate logic, more computation problems are given, such as Hanoi Tower. There was even an extra extra-credit programming part on solving the online exam scheduling problem that the course itself had, by writing quantifications, comprehensions, and aggregations in DA and feeding them to a solver.

6 RESULTS, ANALYSIS, AND ADOPTION

To give insight into whether extra-credit programming was truly beneficial, we mainly considered two metrics: student test performance, a quantitative metric; and student surveys, a qualitative metric. Additional details can be found in [16].

Student test performance. To see if extra-credit programming helped students learn better, we examine whether students who did more programming on certain topics performed better on exams covering those topics. We consider three groups of programming assignments: the first two assignments before Midterm 1, the next three before Midterm 2, and all six before the Final, grouped by their relevancy to the exam. In each group, students are categorized by how many they submitted for that group, and the test average from each category was taken. Figure 3 shows the results.

With a very slight exception in group 2, there is a clear positive correlation between the number of programming assignments submitted and exam performance. Unfortunately, this is not enough to prove that the assignments by themselves improved student performance, due to self-selection bias—it is probable that students who would do better on exams would do more extra-credit work. In the future, it may be worthwhile to introduce these assignments in a way that eliminates self-selection bias.

Student surveys. For each homework, an online survey was created, containing a section asking for qualitative feedback on extra-credit programming.

For the first programming assignment, on predicate logic, of 115 students enrolled, 109 submitted the homework; 53 (48.7%) did

not try the extra credit, 20 (18.3%) tried but had issues installing or running Python, and 36 (33.0%) completed all or part of the programming. For the 53 who did not try, and the 20 who tried but failed, the optional nature of the extra credit, being <1% of the course grade, mostly likely did not motivate them enough. Of the 36 who did programming successfully, 31 (86.1%) completed both `cafe` and `tarski`, and 5 (13.9%) completed one of them. Overall, 21 of 36 (58.3%) indicated they enjoyed the programming, and 24 (66.7%) indicated they wanted more programming like this.

For comparison, the last programming assignment had 96 students respond; only 9 (9.4%) failed to install or run Python—much less than before—but it also reveals that these students did not try sooner. 43 (45.3%) did not try the extra credit, and 44 (46.3%) successfully did one or more programming problems—again an improvement. Of the 44 who succeeded, 32 (72.7%) indicated they enjoyed the programming—another positive result.

Suggestions for adoption. The feedback we have received, from both the surveys and in-person interactions with students, supports that students enjoy discrete math more with programming. For students who succeeded installing and running Python, extra-credit programming was well received.

Clearly not all students come to a discrete math course equally capable of configuring the minutia of their programming environment. As programming was optional, system configuration was only briefly covered in lectures. For the future, we suggest making programming required to a degree, and providing in-class configuration sessions.

Once required, the programming part could be given more weight in the course grade, and could be tested in the exams as well. Because it was extra-credit programming worth <1% of the course grade for each assignment, and because it came with entirely different and much longer problem descriptions, it was harder to initially motivate the average student. However, this can be easily overcome by increasing its weight in the course grade.

Indeed, most students who did not do extra-credit programming indicated they would have liked to, and some asked about exercises after the course ended. Already, we have had positive results with 56 students who successfully did some or all programming problems. Some went beyond—running distributed algorithms using DA, asking deep questions about Python, and asking to do research projects.

We will make problem descriptions and program files for the programming assignments publicly available. Programming solutions will be made available to instructors by requests.

7 CONCLUSION

We have presented a principled approach for teaching discrete math with programming by using a powerful language that extends Python. The approach and language cover all central topics, and allows novice users to understand the concepts precisely, write them rigorously in specifications, and use them directly in executions.

Our results and analysis of using the approach support broader deployment. Exploiting Python also allows the approach to be built on to teach more advanced subjects later on, especially with the increasing growth of Python libraries.

REFERENCES

[1] Alfred V. Aho and Jeffrey D. Ullman. 1994. *Foundations of Computer Science: C Edition.* W. H. Freeman.

[2] Jonathan Berry. 1997. Improving discrete mathematics and algorithms curricula with LINK. *ACM SIGCSE Bulletin* 29, 3 (1997), 14–20.

[3] Elin Björnsson, Fredrik Johansson, Jan Liu, Jesper Olsson, Henry Ly, and Andreas Widbom. 2017. *Proof Editor for Natural Deduction in First-order Logic: The Evaluation of an Educational Aiding Tool for Students Learning Logic.* Bachelor Thesis. Chalmers University of Technology, University of Gothenburg.

[4] Gary Chartrand and Ping Zhang. 2011. *Discrete Mathematics.* Waveland Press.

[5] John Cigas and Wen-Jung Hsin. 2005. Teaching proofs and algorithms in discrete mathematics with online visual logic puzzles. *Journal on Educational Resources in Computing (JERIC)* 5, 2 (2005), 2–es.

[6] GL Engel and ND Jones. 1973. Discrete structures in the undergraduate computer science curriculum. *ACM SIGCSE Bulletin* 5, 1 (1973), 56–59.

[7] Susanna S Epp. 2011. *Discrete Mathematics: Introduction to Mathematical Reasoning.* Nelson Education.

[8] Susanna S Epp. 2011. *Discrete Mathematics with Applications* (4th ed.). Nelson Education.

[9] Leo Geurts, Lambert Meertens, and Steven Pemberton. 1990. ABC Programmer's Handbook. https://homepages.cwi.nl/~steven/abc/programmers/handbook.html

[10] David Gries and Fred B Schneider. 2013. *A Logical Approach to Discrete Math.* Springer Science & Business Media.

[11] James L Hein. 1993. A declarative laboratory approach for discrete structures, logic, and computability. *ACM SIGCSE Bulletin* 25, 3 (1993), 19–25.

[12] Shin Hong. 2020. Using SMT Solver and Logic Puzzles for Teaching Computational Logics in Discrete Mathematics Class. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* 1381–1381.

[13] K Kennedy and J Schwartz. 1975. An introduction to the set theoretical language SETL. *Computers & Mathematics with Applications* 1, 1 (Jan. 1975), 97–119.

[14] Tun Li, Wanwei Liu, Juan Chen, Xiaoguang Mao, and Xinjun Mao. 2019. Towards connecting discrete mathematics and software engineering. *Tsinghua Science and Technology* 25, 3 (2019), 325–335.

[15] Yanhong Annie Liu. 2013. *Systematic Program Design: From Clarity To Efficiency.* Cambridge University Press.

[16] Yanhong A. Liu and Matthew Castellana. 2020. Discrete Math with Programming: A Principled Approach. *Computing Research Repository* arXiv:2011.14059 [cs.CY] (Nov. 2020), 15 pages.

[17] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms. *ACM Transactions on Programming Languages and Systems* 39, 3 (May 2017), 12:1–12:41.

[18] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications.* 395–410.

[19] Yanhong A. Liu and Tim Teitelbaum. 1995. Systematic Derivation of Incremental Programs. *Science of Computer Programming* 24, 1 (1995), 1–39.

[20] Yanhong A. Liu, Chen Wang, Michael Gorbovitski, Tom Rothamel, Yongxi Cheng, Yingchao Zhao, and Jing Zhang. 2006. Core Role-Based Access Control: Efficient Implementations by Transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation.* 112–120.

[21] Mehak Maniktala and Tiffany Barnes. 2020. Deep Thought: An Intelligent Logic Tutor for Discrete Math. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* 1418–1418.

[22] Kenneth E Martin. 1984. The role of discrete structures & operations research in a computer science curriculum. *ACM SIGCSE Bulletin* 16, 4 (1984), 4–6.

[23] Kirby McMaster, Nicole Anderson, and Brian Rague. 2007. Discrete math with programming: better together. *ACM SIGCSE Bulletin* 39, 1 (2007), 100–104.

[24] Daniel Nohl. 2007. Using an automated reasoning program as a CS application in discrete mathematics. *Journal of Computing Sciences in Colleges* 22, 4 (2007), 7–13.

[25] James F Power, Thomas Whelan, and Susan Bergin. 2011. Teaching discrete structures: a systematic review of the literature. In *Proceedings of the 42nd ACM technical symposium on Computer science education.* 275–280.

[26] Anja Remshagen. 2010. Making discrete mathematics relevant. In *Proceedings of the 48th Annual Southeast Regional Conference.* 1–6.

[27] Isabella Romeo. 2018. *Hacking Discrete Math with Python 3.* CreateSpace Independent Publishing Platform.

[28] Kenneth H Rosen. 2012. *Discrete Mathematics and Its Applications* (7th ed.). McGraw-Hill.

[29] Jacob Schwartz, Domenico Cantone, and Eugenio G Omodeo. 2011. *Computational Logic and Set Theory: Applying Formalized Logic to Analysis.* Springer.

[30] Jacob T. Schwartz, Robert B. K. Dewar, Ed Dubinsky, and Edmond Schonberg. 1986. *Programming with Sets: An Introduction to SETL.* Springer.

[31] Ben Setzer. 2009. A lab course for discrete mathematics. In *Proceedings of the 47th Annual Southeast Regional Conference.* 1–2.

[32] Allan M. Stavely. 2014. *A Gentle Introduction to Discrete Math Featuring Python.* The New Mexico Tech Press.

[33] The Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013.* Technical Report. ACM and IEEE Computer Society. https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

[34] Leo C Ureel and Charles Wallace. 2016. Discrete mathematics for computing students: A programming oriented approach with Alloy. In *2016 IEEE Frontiers in Education Conference (FIE).* IEEE, 1–5.

[35] Guido Van Rossum. 1993. An Introduction to Python for UNIX/C Programmers. *Proceedings of the NLUUG najaarsconferentie. Dutch UNIX users group* (1993), 8 pages.

[36] Thomas VanDrunen. 2017. Functional programming as a discrete mathematics topic. *ACM Inroads* 8, 2 (2017), 51–58.

[37] Hillel Wayne. 2020. Alloy Documentation.