DOI:10.1145/3382036

John MacCormick

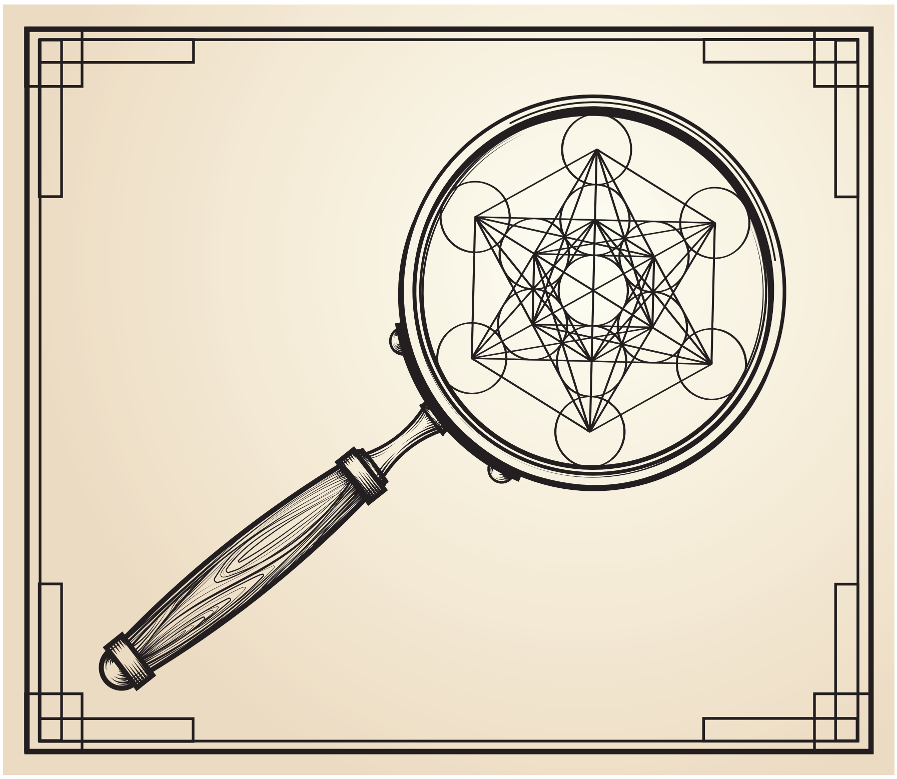
Viewpoint

Using Computer Programs and Search Problems for Teaching Theory of Computation

Recognizing the significance of a cornerstone of computer science.

The theory of computation is one of the crown jewels of the computer science curriculum. It stretches from the discovery of mathematical problems, such as the halting problem, that cannot be solved by computers, to the most celebrated open problem in computer science today: the P vs. NP question. Since the founding of our discipline by Church and Turing in the 1930s, the theory of computation has addressed some of the most fundamental questions about computers: What does it mean to compute the solution to a problem? Which problems can be solved by computers? Which problems can be solved efficiently, in theory and in practice?

Yet computational theory occupies an ambiguous role in the undergraduate curriculum. It is a required core course for the computer science major at many institutions, whereas at many others it is an upper-level elective. And whether required or not, the theory course can have a reputation as an austere and perhaps even irrelevant niche, disconnected from the skills and ideas that comprise computer science. This is not a new phenomenon, and in recent decades the CS community has worked diligently to improve the accessibility and perceived relevance of the theory course. Notable contributions include the JFLAP software for experimentation with automata,[8] and various efforts to promote "NP-completeness for all" via visualizations and practical laboratory exercises.[1]



This Viewpoint discusses two specific suggestions for continuing to improve the accessibility of the theory course while maintaining its rigor: first, emphasizing *search problems* rather than decision problems in certain parts of the course; and second, employing *computer programs* written in a real programming language as one of the standard computational models, complementing the use of automata such as Turing machines. The suggestions here apply specifically to an undergraduate course in which students encounter theory of computation for the first time. The content of such courses varies widely, and the following suggestions are most applicable to introductory courses that incorporate both computability and complexity theory.

Emphasizing Search Problems

The theory of computation is usually phrased in terms of *decision* problems: questions with a single-bit yes/no response. In other areas of computer science, however, we are usually interested in *search* problems, whose solutions consist of more than a single bit. As an example, consider the problem of finding a Hamilton cycle in a graph—that is, a route visiting every vertex exactly once. Theory courses usually discuss the decision problem, "Does the graph G contain a Hamilton cycle?" But if the answer is yes, we still do not know the route of a Hamilton cycle in G. It is more natural and useful to consider the related search problem, "Find and output a Hamilton cycle of G, if one exists."

Once they have finished their first theory course, computer science undergraduates will recognize the connections between search and decision problems. But search problems are more familiar and more immediately applicable, so there are good reasons to teach the more elementary parts of the theory course with an emphasis on search problems. It is worth noting that Knuth prize winner Oded Goldreich is an advocate of this approach[2]; his books are among the few modern textbooks[3,6] that adopt search problems as a primary paradigm. But search problems can easily be incorporated into more traditional approaches that retain decision problems as the standard model, and I do recommend this as a means of connecting the theory course more closely to other parts of the undergraduate CS curriculum.

The advantages of, and techniques for, teaching CS theory via search problems have been discussed in detail elsewhere.[5] One interesting result, based on a survey of CS undergraduates, is that search problems are perceived as significantly more useful than decision problems. Because perceived relevance is known to be a factor in achieving good learning outcomes, this provides indirect evidence the approach is beneficial.

Python program example.

```
# yesOnStr(P,I) returns 'yes' if the Python function with
# source code P returns 'yes' after receiving input I.
# We assume yesOnStr(P,I) exists and works correctly
# on all inputs.
from yesOnStr import yesOnStr

# Below is a diagonalized and inverted version of
# yesOnStr(P,I). What happens when the
# parameter P is a string consisting of the
# source code of diagYesOnStr?
def diagYesOnStr(P):
    if yesOnStr(P, P)=='yes':
        return 'no'
    else:
        return 'yes'
```

The source code of Python function `diagYesOnStr()`. This code provides the core of a proof by contradiction. When given its own source code as input, the function `diagYesOnStr()` outputs 'yes' if and only if it outputs 'no'. This contradiction means our assumption that the function `yesOnStr(P,I)` can exist is not valid. Therefore, the problem YesOnStr is undecidable: no Python function can correctly answer the question "does Python function P output 'yes' on input I?" for all inputs.

Using Real Computer Programs to Complement Automata

Another technique for increasing student engagement and connections with other parts of the CS curriculum is to employ code in a real programming language. This can provide a beneficial supplement to the automata and grammars that typically dominate a course in theory of computation. Formal models such as Turing machines are of course essential, especially for providing a rigorous definition of computation itself. However, it is possible to teach a mathematically rigorous theory course using a programming language as the primary model of computation. In this approach, the program model is layered over Turing machines as an underlying model, and Turing machines are still employed when required in certain proofs and definitions. A strong majority of CS theory textbooks do not employ a programming language as the primary computational model, but several authors have done so, for example using Python,[6] Ruby,[9] and a variant of LISP.[4,7] As an example of the approach, consider the Python program shown in the figure here.

This code provides the basis for a proof by contradiction, demonstrating that a certain computational problem is undecidable. Specifically, it proves the undecidability of the following question: "Given a Python function `P()` and input string `I`, does `P` return the value `'yes'` when invoked with input `I`?" A detailed explanation of the proof is outside the scope of this Viewpoint; here, I focus on the potential advantages for undergraduate students who are encountering this type of material for the first time. Note that, in practice, the code shown in the figure would be presented in class only after exposure to and experimentation with prerequisite concepts, such as Python functions that take the source code of other Python functions as input and analyze them or transform them in some way. Nevertheless, for concreteness and compactness in this Viewpoint, I describe the potential benefits to students directly as they appear in this proof.

First, note the undecidability result itself can be described in terms of Python programs: "It is impossible to write a Python program that determines whether other Python programs will output `'yes'` on a given input." From one point of view, this is a purely cosmetic change from the equivalent statement in terms of Turing machines: "there does not exist a Turing machine that determines whether other Turing machines will accept a given input." After all, students in any theory course must come to understand the equivalence between Turing machines and computer programs. Nevertheless, the practice of discussing results in terms of computer programs that have clear connections to other areas of computer science provides the instructor with opportunities for increased engagement; this has certainly been my own experience.

Second, there are some steps in theory proofs that are surprisingly subtle when expressed in terms of Turing machines, but become obvious and familiar in a programming language. One example in the figure is the trick in which a single parameter `P` is duplicated and passed on in two separate roles to the two-parameter function `yesOnStr(P,P)`. In class, this can be further explicated by stepping through the program in a debugger and showing the dual roles of `P`: as source code in the first parameter and as a text string in the second parameter. (Even Alan Turing recognized the challenges inherent in proofs based on automata. In the seminal 1936 paper that introduced Turing machines, he sympathized with readers who might feel "there must be something wrong" in his first such proof.[10])

Third, students can build an intuitive understanding of code-based proofs by active experimentation with the code. In the example in the figure, one can provide an approximate version of `yesOnStr()` that works correctly on a limited class of inputs. Students can then predict the output of `diagYesOnStr()` on various inputs, and check their answers by running the code. They can construct variants of the code, discussing which variants produce the desired contradiction and which do not. By implementing `yesOnStr()` via simulation, students can discover an important extension to this result: we can in fact write a Python program that always terminates correctly on positive instances of this problem, so the problem is *recognizable* but not decidable.

Many theory courses could benefit from making more explicit connections to other parts of the computer science curriculum.

Fourth, some students may find the programming approach transfers more easily to novel problems. In recent years I have taught three different approaches for undecidability proofs to all students: traditional reductions employing prose descriptions of Turing machines; explicit Python programs (similar to the example in the figure here) supplemented by a prose explanation of the desired contradiction; and the application of Rice's theorem. In tests and exams, students may choose which proof method to use, and there is an approximately even split among these three proof techniques. In particular, a significant fraction of students choose to write out a Python program as part of their exam answer. This provides empirical evidence that the programming approach is beneficial for some students, and it is plausible all students gain improved understanding from seeing multiple approaches.

Conclusion

Over a period of eight years, I have experimented with techniques for making the undergraduate theory course more accessible and engaging. This Viewpoint suggests two possibilities: emphasizing search problems and employing real computer programs. I do not advocate the universal or complete adoption of these suggestions. I have backed away from some aspects of the approach myself. For example, after experimenting with teaching NP-completeness based on search problems, I concluded this part of the course works better when taught with the traditional focus on decision problems. Similarly, I found there are some technical results about polynomial-time verifiers that can be proved more instructively—for the target audience of novice undergraduates—using Turing machines rather than computer programs.

Every instructor and every group of students is different; instructors must adopt a style of teaching that is authentic to themselves, achieves the goals of the students, and is based realistically on the students' level of preparedness. I do believe that many theory courses could benefit from making more explicit connections to other parts of the computer science curriculum, and it is possible to do this incrementally. If decision problems and Turing machines are retained as the central paradigms, search problems can be still be mentioned when relevant, and snippets of code can be used to illustrate subtleties.

Whether or not the specific ideas suggested here are adopted, it seems important that we continue to strive for accessibility and engagement in the undergraduate theory course. The theory of computation is a profound and important cornerstone of computer science; I hope that in the years ahead, an ever-growing number of students will appreciate both its beauty and its significant connections to the rest of the computer science curriculum. C

References

1. Crescenzi, P., Enstrom, E. and Kann, V. From theory to practice: NP-completeness for every CS student. In *Proceedings of ITiCSE*, 2013.
2. Goldreich, O. On teaching the basics of complexity theory. *Theoretical Computer Science: Essays in Memory of Shimon Even.* (2006), 348–374.
3. Goldreich, O. *P, NP, and NP-Completeness: The Basics of Computational Complexity.* Cambridge University Press, 2010.
4. Jones, N.D. *Computability and Complexity: From a Programming Perspective.* MIT Press, 1997.
5. MacCormick, J. Strategies for basing the CS theory course on non-decision problems. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (*SIGCSE '18*), 2018.
6. MacCormick, J. *What Can Be Computed?: A Practical Guide to the Theory of Computation.* Princeton University Press, 2018.
7. Reus, B. *Limits of Computation: From a Programming Perspective.* Springer, 2016.
8. Rodger, S.H. and Finley, T.W. *JFLAP: An Interactive Formal Languages and Automata Package.* Jones & Bartlett, 2006.
9. Stuart, T. *Understanding Computation: From Simple Machines to Impossible Programs.* O'Reilly, 2013.
10. Turing, A.M. On Computable Numbers, With An Application To The Entscheidungsproblem. In *Proc. London Math Soc.*, Vol. 2–42, 1, (1937), 230–265.

John MacCormick (jmac@dickinson.edu) is Associate Professor of Computer Science at Dickinson College, Carlisle, PA, USA. He is the author of *Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers.*