6-1998

The Eight-Minute Halting Problem

J. Paul Myers Jr.
Trinity University, pmyers@trinity.edu

The Eight-Minute Halting Problem

J. Paul Myers, Jr.
Department of Computer Science
Trinity University
San Antonio, Texas USA
pmyers@cs.trinity.edu

Abstract

After years of presumed emphasis of CS theory in the curriculum, it is currently in vogue to downplay, if not disparage, a significant role for theoretical issues. This is being done with such vigor, however, that some are advocating an abandonment of even such topics as unsolvability as being no longer fundamental to a well-educated computing professional. An appeal is made, using the universal acknowledgment of the importance of the liberal arts to any well-educated person, to assert that certain theoretical topics are part of our liberal arts heritage. Moreover, students find even a brief presentation of these topics to be very illuminating.

Introduction

We live in peculiar times; or perhaps it would be better to say that one of the mythical pendulums has begun its swing the other way. In informal discussions with colleagues, over the years one hears the complaint that computer science has been too theory-oriented. At the risk of overstatement (yet trying to find explanations for current developments), at earlier times when mere programming was denigrated, theory was a large portion of what was left. For example, many of the classic texts of our field were written by theoreticians (e.g., Hopcroft & Ullman among others).

Many of these conversations over the years have been hostile to the elevated role of theory in computer science. But most recently, theory-bashing has found a more formal and public forum. For example, it's now guaranteed to elicit some cheap laughter and sneering to point out that theoretical computer science is all but useless in the curriculum as Tanenbaum did in his keynote speech after accepting the SIGCSE Award for Outstanding Contributions to Computer Science Education at the 1997 Technical Symposium. Evidently, a group of people challenged him after the speech, but the damage was done; it is now fashionable to deride theory in our curriculum to an extent heretofore unseen. Theory does noteven appear on the Paper Submission Sheet (Course Related) for the 1998 SIGCSE Symposium!

Presumably, our major focus has become training professionals for careers in industry. The critiques against theory are usually coupled with this priority. Of course, we all have gained as more and more of our topics have become relevant to industry and have come to possess intellectual significance as well. It has become enjoyable straddling the distinctions between the theoretical and the applied. And maybe there was once a conspirational ignoring of industry's needs (as in the old debate between "pure" and "applied" mathematics); but the recent backlash has been intense.

Indeed, I participated in a recent working group to define broad curricular parameters for informatics education (seen as somewhat broader than computer science education and considerably oriented toward the job market). In our deliberations I made the comment that anyone in a computing profession should know about the Halting Problem. While ultimately such topics did make it into the recommendations (perhaps to still my weeping and hysteria!), the immediate response of a few was a joint guffaw. Why is this?

Of course, one could go into a litany of all the reasons why theory should be an important component of any computing professional's background; just read the preface to any theory textbook! These reasons have to do with future applicability, sharpening of reasoning skills, and the like. But here we take another stance; namely that certain theoretical topics may not be useful at all but are part of the intellectual heritage of our field. We will cite only one preface, but it is a notable example as it is an indication of how times have changed. In the late 1970's IBM sponsored a distinguished series of books called the "Systems Programming Series" published by Addison-Wesley. One of these was a delightful book by Frank Beckman, *Mathematical Foundations of Programming*, that is more an expository than a rigorous treatment of computer science theory [1]. He writes of the computing practitioner:

> Yet it seems that those who will spend their working lives in computing should have some curiosity about, and acquire some understanding of, what mathematics has to offer in providing a greater insight into the phenomena surrounding the computer – even when this insight has no apparent immediate utility. [1]

Reviewed Papers

Eight-Minute *(continued from page 53)*

This points to a joy in appreciating the philosophical depths of one's endeavors whatever they may be and in whatever field. But it's exactly this curiosity and intellectual joy that seem to be absent as the field moves ever closer to dedicating itself to industry's needs.

This turn of events seems particularly odd when one considers that computing educators, as indeed educators in virtually all technical fields, make continuing pleas for a solid grounding in the liberal arts for its practitioners [2]. The oddness is that certain parts of computing theory are precisely those aspects of the field in which the liberal arts are interested! Collcagues in philosophy, history, English, and art are always fascinated about topics such as the Halting Problem or intractability – far more than a discussion of spreadsheets or databases. Such people, when told of these topics, find them to be part of their Western cultural heritage! Indeed, there are a number of overview books on computing for the "intelligent layperson" that highlight just such topics [4,6,11 and cf. 7]. Schaffer, author of one breadth-first elementary book has written

> ... the first goal of most texts is to convey practical information, much of which is rather less than earthshaking. ... The topics treated here are of practical value, but they have been chosen primarily on grounds of intellectual significance. I have asked myself what ideas we computer scientists have reason to be proud of and then attempted to present these at an introductory level. [8]

And, of course, his is certainly not primarily a theory text, for as computer scientists we have much to be proud of that is applied: efficient algorithms, incredible architectures and related technologies, the world-wide-web, multimedia, databases, AI & expert systems, the amazing variety of computing models and languages, virtual reality, etc., etc. Unquestionably, applications are important and interesting, increasingly so all the time.

But if programs and applications are at the heart of computing, then we might say that certain theoretical issues are at the soul of the field and are where the real philosophical depth and permanence of the field is to be found (Harel's book, *Algorithmics*, is even subtitled the *Spirit of Computing* [4]). Those in liberal arts sense this; but we call for increased attention to the liberal arts while ignoring the liberal arts aspects of our own field!

Indeed, it is not only philosophical significance that is being ignored, but even the historical roots of the field. Reason is held to be the towering human achievement, at least in the West; and the early thinking about computation was an attempt to mechanize reasoning [5]. This, of course, introduces artificial intelligence, another area in which it seems every computing professional should have *some* knowledge. [As an aside, if AI eventually succeeds, it would be hard to say what other human achievement will ever have been more significant.] Turing, Gödel, and others were attempting to understand the "effective" (or algorithmic) aspects of reason when they developed their computational models and made their extraordinary discoveries. These were discoveries that shook the foundation of our concepts of reason and mind that were dearly held for millennia (at least since the ancient Greeks). And for this work (not for his breaking the German code), Turing is deemed to be the "father of computing" and the namesake for the ACM's most prestigious award.

Our history and philosophical issues are at least as rich and ancient as those in any field or discipline. Yet we call for learning the histories, arts, and philosophies of other fields while ignoring our own.

My students in Computers and Society (a course for humanities majors) are always astonished to learn of unsolvability and intractability. There are treatments that are quite elementary and accessible if one takes the small amount of time needed to present and motivate them.

The Halting Problem in Eight Minutes

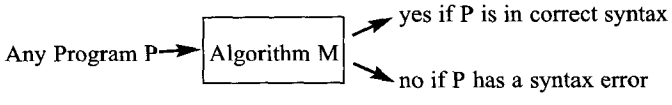
So now we come to the Eight-Minute Halting Problem. After the guffaws of some in the informatics working group, in a curious mixture of tongue-in-cheek and bluster, I asked if we could spare just eight minutes out of our busy curriculum to present something of such philosophical and historical importance. So, of course the topic was given the eight minutes, though there was skepticism as to whether it could be done. Hence this presentation is a response to the skeptics!

This will follow the ordering with which I present certain topics to the aforementioned humanities students. As a background they have already seen and worked with the linguistic/computational concepts of syntax, logical (semantic), and execution (pragmatic) errors at a very elementary level – e.g., examples in QBASIC or pseudocode. Certainly *all* computing folks should know about these! But they do not need to know about models of computation or Church's Thesis; as current texts are pointing out, students today find "self-evident" that their favorite models of computation are fully robust [9].

Later in the course, when it's time to discuss the Halting Problem, I ...

Minute 1: Remind them about syntax errors.

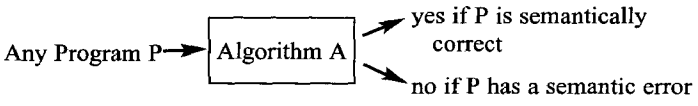
Minute 2: Tell them that we know how algorithmically to solve the "Syntax Problem" – i.e., is *any* given program *syntactically* correct (we don't care about if it does what it's supposed to). The students themselves point out that the algorithm M below is a compiler!



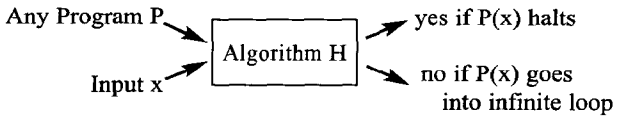
Reviewed Papers

The students even realize that we have solved this problem so thoroughly that the compiler does a number of other things at the same time, and it even returns useful messages.

Minute 3: Talk about the "Semantic Problem" – does a given program do what it's supposed to do. Might there be an algorithm to solve this one, especially since syntax was so easy?



Minutes 4-6: Let them know that this one is harder and that we might look at a small piece of the overall problem. Is an infinite loop an error? Generally yes, though an alert student might point out that some programs (OS, real-time monitors, ...) are in error if they halt. In either case, one wants to know whether a given program P will go into an infinite loop or not. Such a loop or its absence will be an error we want to know about, a small aspect of an overall semantic analysis. We'd like to have as good a solution as a compiler provides, namely not just that there is an error, but precisely the nature of the error. That is, we'd like to know for what inputs does P go into an infinite loop. But instead we're going to look at even a simpler aspect of the problem: Given a particular input x, will P halt on that x or go into an infinite loop?



A couple trivial examples are created, e.g., in pseudocode, to show that we can sometimes answer this question (and even the more general version) for particular programs.

Minute 7: Point out that it's not really bizarre to think of a program as being input to itself. For example, we could compile a compiler using its own code. Or we could run a program that counts lines-of-code (or some other metric) through itself to count its own lines of code. The program has no "awareness" that it is its own input. No problem, and the students agree.

Minute 8: Now for the hand-waving; after all, I'm almost out of time! One may simply point out that by using a very odd form of reasoning, one can set things up so that a modified version of our last program H above (Minute 5) is run into itself leading to a contradiction from which we conclude that H cannot exist. One can give references to *elementary* treatments [3,6,11] and alert students that the mode of reasoning is unusual, odd. But it doesn't require solving nonlinear differential equations or triple integrals over infinite domains or even developing a parser to be unambiguous! It's just weird; but an intelligent high school student can "get it" after some consideration. But it's weird; our minds are not accustomed to reasoning in that fashion.

My intent in the past few sentences was not to be silly. Somehow, the repeated emphasis on the unusual (but elementary) nature of the proof is quite satisfying to these mathematically unsophisticated students. And if one did want to take the additional fifteen to twenty minutes or so to present the proof, it's quite easy to set up [again, 3,6,11].

But for many students, just knowing that they *could* read and eventually understand the proof is sufficient. After all, they do accept that the Syntax Problem is solvable by compilers long before they have any idea how a compiler works. Probably if we look at any elementary curriculum in any field, we would notice far more hand-waving than appears at first glance.

And now our students have seen something really interesting. It can be emphasized here that now that we know there is an unsolvable problem, Pandora's Box is forever open; there are many such problems. This is very bad news for the software industry; and, by the way, now gives a meaningful answer to the bright student who just might ask (or be led to ask by the teacher!): "Why, since programs are mathematical objects of a sort, can't we *prove* them to be correct rather than *test* them for errors?" Sometimes we can, but the unsolvability of the Halting Problem is a theoretical limitation on that possibility for all cases.

Moreover, this simple result (and related ones) in this century completely overthrew our view of and blind confidence in mathematics and the rational mind that has been part of our Western heritage since ancient times. Previously it was accepted that any well-posed problem had a solution if we were just clever enough. And, by using the technique of self-reference (a program running on itself), computer science and mathematics have joined in a cultural phenomenon of primarily the twentieth century as mind looks at mind (psychology), art is about art, music is written about music, and literature becomes reflexive. Students find these discussions fascinating, as indeed they should; these ideas are fascinating. And, as a discipline we can take great pride that some of our results have had such profound implications in the world of the intellect and the liberal arts.

Conclusion

Much of what is written above is in a rather dramatic style, making this somewhat fun to write! But I really do think the issue is important and I really can't understand emphasizing the liberal arts while withholding from students the profound ways in which their own field has impacted those same liberal arts. If we want liberal arts to produce a well-rounded individual, then it must be important for students to see these aspects of their own field.

The "eight minutes" is, of course, not entirely serious; but it's not far off. And it indicates that some discussion of these basic ideas need not intrude into a computing cur-

riculum. It still strikes me as very odd to have to write a sentence like this last one – that these ideas could ever seem intrusive seems absurd (in the liberal arts absurdist sense!). But as mentioned, we live in peculiar times; probably knowledge of the Halting Problem does not translate well into development of web-editors or telecommunications packages.

As a final note, I might mention that I have covered both the Halting Problem and the presumed intractability of the Travelling Salesman Problem (giving economic motivation for its solution) in a single fifty-minute class. By starting with just three, then five, cities, the students see how the search tree develops and they themselves determine that the growth is factorial. Then the rest is just a matter of having a student pull out a calculator to compute, say, 48! (the capitals of the contiguous US states), assuming a few billion comparisons per second, and counting the eons! This is probably the students' favorite lecture of the semester.

And here no talk of NP-Completeness is necessary; the students are stunned by the news anyway!

Appendix

A wonderful alternative to augment the presentation of the Halting Problem without adding too many extra minutes (!) is derived from Smullyan's very clever machine illustration of Gödel's proof of incompleteness [10]. Gödel showed that there can be no algorithmic all-purpose means of determining all the true statements in a sufficiently robust formal system.

Smullyan's machine prints expressions of only four symbols: P,N,R,*. An expression is printable if the machine can print it. A sentence is in one of four forms, where x is any expression constructed from the four symbols: (1) P*x (true iff x is printable); (2) NP*x (true iff x is not printable); (3) PR*x (true if the repetition xx of x is printable; and (4) NPR*x (true iff xx is not printable). The machine is assumed to be completely accurate: every sentence printed by the machine is true. So now, for example, if PP is printable, we can additionally write P*PP and PR*P. And with these, we can now write PP*PP and PPR*P; etc.

Next consider the sentence NPR*NPR* (true iff the repeat of NPR* is not printable). But the repeat of NPR* is in fact NPR*NPR* which now essentially says "I am not printable." If true, then it can't be printed and so our machine can not print all true sentences. If, on the other hand, NPR*NPR* is false, then it can be printed, defying the presumed accuracy of our machine. So if the machine is accurate, there are true sentences it can not print. This is the odd sort of reasoning that proves that the Halting Problem is unsolvable. In Gödel's context, a true but unprovable sentence might be "Program P will halt on input x."

Bibliography

1. Beckman, Frank S., *Mathematical Foundations of Programming*, IBM Systems Programming Series, Addison-Wesley, Reading, MA1980.
2. Cassel, Lillian N., "Computing and Education at the University Level," Proceedings of the IFIP WG 3.2 Working Conference, 1997, to appear.
3. Goldschlager, L. and Lister, A., *Computer Science — A Modern Introduction*, 2nd ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
4. Harel, David., *Algorithmics: The Spirit of Computing*, Addison-Wesley, Reading, MA, 1987.
5. Haugeland, John, *Artificial Intelligence: The Very Idea*, MIT Press, Cambridge, MA, 1985
6. Hofstadter, Douglas R., *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1979.
7. Myers, J. Paul, Jr., "The New Generation of Computer Literacy," *SIGCSE Bulletin*, Vol. 21, no. 1, ACM Press, February 1989.
8. Schaffer, C., *Principles of Computer Science*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
9. Sipser, Michael, *Introduction to the Theory of Computation*, PWS Publishing Company, Boston, 1997.
10. Smullyan, Raymond, *5000 B.C. and Other Philosophical Fantasies*, St. Martin's Press, New York, 1983.
11. Walker, Henry M., *The Limits of Computing*, Jones and Bartlett Publishers, Boston, 1994.