Teaching Theoretical Computer Science and Mathematical Techniques to Diverse Undergraduate Student Populations

Dr. Predrag T. Tosic, University of Idaho

Predrag Tosic is an early mid-career researcher with a unique mix of academic research, industrial and DOE lab R&D experiences. His research interests include AI, data science, machine learning, intelligent agents and multi-agent systems, cyber-physical/cyber-secure systems, distributed coordination and control, large-scale complex networks, internet-of-things/agents, and mathematical and computational models and algorithms for ”smart” transportation, energy and other grids. He is interested in applying data analytics, machine learning, intelligent agent and AI techniques to emerging problems related to large-scale decentralized cyber-physical systems, critical infrastructures and ”smart grids”, autonomous vehicles, as well as energy, health care and other domains of major economic and societal impact.

Dr. Tosic holds a doctorate in Computer Science from the University of Illinois at Urbana-Champaign. His doctoral dissertation (2006) was on Distributed AI and large-scale Multi-Agent Systems. Most recently, at Washington State University (2015 - 2017), Dr. Tosic worked on dynamics of large-scale networks, graph pattern mining, Boolean Network models of cyber-physical systems, Internet-of-Agents, as well as AI, data analytics and knowledge engineering applied to problems in health care. While at the University of Houston (2009 – 2012), he did research in machine learning, multi-agent distributed computing and control, data mining and distributed database systems, emerging behavior in complex networks, ”smart energy” and computational game theory. During his graduate studies and combined five years of non-tenure-track academic research, he has authored over 70 peer-reviewed publications. He has a versatile R&D experience spanning three different high-tech industries, with both big companies (Cisco Systems and Microsoft) and high-tech startups, as well as with a leading government research lab (Los Alamos National Laboratory). He holds three USPTO patents (IP of Cisco Systems).

In addition to a doctorate in Computer Science, Predrag Tosic holds three master’s degrees, two in mathematical sciences and one in CS. Tosic has a considerable teaching and student research mentoring experience. He has enjoyed working with students of a broad variety of ethnic, cultural and socio-economic backgrounds and at different types of academic institutions. He has been actively involved with IEEE – the Palouse Section and is currently President of the Section’s Computer Society. He is also an active member of ACM, ASEE and AMS.

Dr. Julie Beeston, University of Idaho

Dr. Julie Beeston has both a Master’s degree (from Carleton University) and a PhD (from the University of Victoria) in Computer Science, and she has developed and taught over a dozen courses at the university level. Beyond her teaching experience, she also has over a decade of industry experience as a software developer.

In industry, she has a history of solving ’unsolvable’ problems. She enjoys a great deal of personal satisfaction when her analytical and problem solving skills can be applied to solve complex technical problems and when she can find creative new ways to pass the things she has learned on to the next generation.

Her first teaching experience was at Ozanam Sheltered Workshop teaching adults with mental and physical disabilities. The experience gave her the opportunity to try unique teaching methods and taught her how to tailor her teaching style to a specific person’s needs. That experience taught her that given enough time any student can master any concept. There is no limiting factor on an enthusiastic student’s ability to learn.

Her primary mission in teaching is to get the students enthusiastic about the subject. She does this by giving real-world examples of how the subject matter she is currently teaching has helped her resolve complex problems in industry.

Teaching theoretical computer science and mathematical techniques to diverse student populations

Predrag Tošić and Julie Beeston
University of Idaho - Coeur d'Alene

Abstract

We share our experience and insights gained from teaching foundations of computer science courses to diverse groups of undergraduate (and at times, also graduate) students coming from a broad variety of educational and social backgrounds. Among all required courses of contemporary Computer Science (CS) curricula across the US and Canadian universities, at most institutions (except for the very top research universities), theoretical computer science courses tend to be among the least popular. The reasons behind these prevalent attitudes among CS college students (esp. the undergraduates) tend to fall into two categories: i) theoretical CS courses are "all math" requiring proofs and rigorous formal reasoning that many CS and other engineering students aren't (yet) comfortable with; and ii) "why do we need all this theory anyway" if our career goal is to become software engineers, develop the next cool mobile app, and similar. Yet, most research universities, as well as quite a few non-research colleges, require at least 1-2 semesters of core undergraduate coursework in theoretical or foundational CS.

We summarize some interesting lessons learned from teaching theoretical CS to mostly undergraduate upperclassmen (as well as a few non-traditional students) at two prominent public research universities in the US Pacific Northwest, as well as at comparable institutions in Canada (specifically, British Columbia). While the academic and professional backgrounds of the two authors of this paper are considerably different, we also share quite a bit in common: in particular, both of us have spent several years in high-tech industry prior to returning to the academic world as Computer Science faculty. In particular, when teaching various CS courses, we try to relate concepts and techniques covered to the "real-world" applications, and in particular the recent and current technology challenges and R&D done in industry. We have applied this general philosophy to virtually all courses we have taught, including the very theoretical ones -- such as those on Automata and Formal Languages. The goal is to get CS and other Engineering students intrigued by how, for example, finite automata or context-free grammars are used in compilers and interpreters (parsing, lexical analysis), or the formal specification of programming languages -- as well as to the more recently emerged technologies, such as computational/applied Natural Language Processing (NLP). Without sacrificing rigor, we try to present highly mathematical content in a manner that relates theoretical models and proofs of their properties to practical challenges in computer science and engineering.

Some additional challenges we have encountered while teaching Theory of Computing courses stem from very diverse educational backgrounds of the students we have encountered: from graduate students overcoming true or perceived deficiencies in the CS foundations, to undergraduates transferring from local community colleges, many of whom still struggle with formulating a mathematical proof of any kind. This mix of students provides unique challenges, but also opportunities -- for example, to revisit teaching methodologies for the "mathematical side of computer science" curricula, how to best relate theory to practice (esp. in terms of technologies and applications these diverse students can most readily relate to), and how to best get students of broadly varying backgrounds actively engage in class discussions.

Keywords:

Teaching Undergraduate Computer Science, Theoretical Computer Science, Computer Science Curricula, Teaching Techniques & Methodologies, Diverse Classroom

1 Introduction

In this paper, we summarize some interesting lessons learned from, and challenges encountered while teaching theoretical Computer Science (CS) to mostly undergraduate upperclassmen at two prominent public, land-granted research universities in the US Pacific Northwest, as well as at comparable institutions in Canada (specifically, British Columbia, Alberta and Ontario). While the academic and professional backgrounds of the two authors of this paper are considerably different from each other, we also share quite a bit in common: in particular, both of us have spent several years in high-tech industry prior to returning to the academic world as CS faculty (at the time of writing this paper, one of us is a Clinical Faculty, and the other, Tenure-Track; however, we both teach across the undergraduate CS curricula; furthermore, we both have taught or are currently teaching theoretical and foundational computer science courses for undergraduate upperclassmen).

In particular, the core teaching philosophy the two of us share is that, when teaching various CS courses, both of us try to relate concepts and techniques covered in the classroom to the "real-world" applications and problems, and in particular the recent and current technology challenges and R&D done in industry. We have applied this general philosophy to virtually all courses we have taught, including the very theoretical ones -- such as the courses on (undergraduate introduction to) Automata and Formal Languages. The goal is to get CS and other Engineering students intrigued by how, for example, finite automata or context-free grammars are used in compilers and interpreters (parsing, lexical analysis), or the formal specification of programming languages -- as well as to the more recently emerged technologies, such as computational/applied Natural Language Processing (NLP). Without sacrificing rigor, we try to present highly mathematical content in a manner that relates theoretical models and proofs of their properties to practical challenges in computer science and engineering. This shared mindset, as well as informal discussions and sharing our experiences of teaching theoretical and other “math heavy” computer science courses, esp. to undergraduate upperclassmen (juniors and seniors), are the main motivations behind this paper and our desire to share some of our key insights with the broader community of the CS and other Engineering university-level educators.

The structure of the rest of this paper is as follows. After this introduction, each of us will separately summarize her or his experience with teaching (CS) courses to undergraduate upperclassmen in general, and then discuss in more detail the insights and challenges involved in teaching theoretical or foundational CS courses, in particular. These two testimonials are necessarily rather individual and in many ways personal, yet we feel there are some general lessons to learn from them. In the last part of the paper, we will discuss what is common across our experiences, what are some bigger lessons learned, and based on those insights, what we propose to incorporate into teaching theoretical and foundational computer science courses in the future, to make that learning experience more fulfilling for our students.

2 An Experience: From Working in Industry to Completing PhD to Teaching Computer Science (Testimony from Dr. Beeston)

As a student of Computer Science, I gravitated towards classes taught by professors with industry experience because, like many of my peers, I did not really understand what a person with a degree in Computer Science did for a living and I needed some idea of where I was heading in order to remain motivated to continue the learning process. That is why when I graduated with my Master's degree, I got a decade of industry experience before I pursued my Ph.D. and then began teaching.

Given my naivety of the industry as a student, I am not discouraged by students who echo the myth dispelled by Elliott[4], "Industry wants training and not education". Elliott[4] confirms that even if industry cannot articulate the difference between training and education, they at least acknowledge that "universities are primarily in the education business".

There are many definitions of the distinction between training and education, but for the purposes of this paper, I refer to training as learning how to *do* something, whereas education is learning *about* something (specifically I am referring here to the foundations and theory). As someone who has been on the hiring side of the job placement process, I say that both education and training are important. Training is important initially for getting a new employee into a position of being productive; however, in the long run, quality education will prove to be far more valuable.

Computer Science is a rapidly changing discipline, so "the long run" in Computer Science could be as little as just a few years. The "training" a freshman receives on the latest and greatest products in the industry might easily be obsolete by the time the student graduates with a four year degree. The theory of computing changes at a much slower rate, and much of it will not change at all over the course of the student's career. Therefore, the student is far better equipped if he is well educated and able to leverage that education to be quickly and easily trained in the latest technologies.

One of my primary goals when teaching undergraduates is to get them to see the connection between education and long term success. As a child, I had spent many hours tutoring mentally handicapped adults at a local sheltered workshop, and I am firmly convinced that even a person with a below-average IQ can learn the fundamentals of computer theory as long as they are well motivated, have a good teacher and have unlimited time. Unlimited time is something I cannot give to my students, however. The students have commitments to other classes and often to family and work outside the university; that being said, a well-motivated student will often "find" more time to study the topics that are harder and less interesting if he sees the pay-off at the end. As a good teacher, I try to make the connection between the theory and the practical aspects as soon as possible and guide the students through the process of being first novices, then independent experts, and eventually interdependent team members.

2.1 *Make the Connection Early*

At North Idaho College where I teach undergraduate Computer Science classes, the University administrators are strongly motivated to increase the graduation rate in Computer Science, and they have found that they are losing a large number of students who fail the Discrete Math class and therefore do not have the prerequisites to pursue more advanced undergraduate courses necessary to obtain their Computer Science degree. Often these students tell me that they do not enjoy the rigor of doing formal proofs and do not see the point in learning how to write proofs.

At that level of educational experience, the students often have a view of a computer being essentially self-correcting, so all they have to do is be "close enough". They have experience with spell checkers and search engines that can take an almost random string of characters and often guess what you *meant* to type. At that level, computers are very forgiving and do not require a great deal of precision. If the student's goal is simply to make the next great Android application (app) and get rich quickly, then perhaps those students may indeed have a valid point. However, if the student's goal is simply to make the next great app perhaps they should rethink spending years getting a degree. There are plenty of examples of popular apps that were developed by people who did not have degrees in Computer Science. If the student already has a viable idea, then he can use one of the countless free, on-line tutorials to learn all the technical aspects of creating an app.

Making the next great app, however, is in my opinion a lot like pursuing a career in professional sports. Yes, indeed there are some people who make enormous amounts of money (for example) playing baseball professionally, but a vast majority of people who play that sport never make any money at it -- they play simply for the joy of the game itself.

If the student does want to do more than create an app, one simple exercise that often enlightens a student about the precision required by computers is to have the student check the system control panel (of a Windows OS) for the processor speed. For example on mine it is 3.30 GHz. Explain that that means the computer can do 3.3 billion instructions per second. Ask the student, "If the underlying code is right 99.999% of the time, how many times is it wrong per second?" I realize that this is a gross over-simplification, but it does get the point across in a way that most students can relate to.

Even if the students are well motivated, it also helps if the class itself is entertaining. Karabulut-Ilgu's suggestion of a "Flipped Classroom"[5] where the passive learning is done outside of class using videos and the in-class time is spent on active learning is a very intriguing idea, but it may not always work for highly theoretical classes. On the one hand, it gives the student the chance to review the videotaped material as many times as needed, until he or she is comfortable with the material; however, it does not give the student the opportunity to ask questions until the activity where this material is assumed to have been already learned.

We nowadays have increasingly diverse student populations where people join our classroom from many different cultures, age groups and other diverse backgrounds. To illustrate, I know one professor who was caught off guard by students who could not answer a simple question like, “If you are playing a game of poker and have a full house (of Kings and Aces), what is the probability that your opponent has a better hand than you do if you are using a single deck of cards?” Since it was in a classroom setting, the teacher quickly ascertained the issue was some of the students were from a culture where they had never seen a deck of playing cards, so his brief introduction on the value of each hand was not adequate. The person creating a video for that lecture might anticipate such issues but, in reality, there is really no way to anticipate *all* possible issues. Moreover, even if there was a way to anticipate and cover all possible questions, it would either be very tedious for the students who are familiar with the paradigms to weave through them, or it would be frustrating for the already struggling students to determine which paths to extra information they need to follow to fill in the gaps, or both. That is, even the best educational video cannot always replace a mentor.

In my teaching practice, I have found that a combination of the Flipped Class and the Traditional Class works well. The first portion of the class is spent in traditional lecture style using slides and notes that the students can review in advance if needed. This allows the students to ask questions and interact with the material before breaking off into Active Learning and Problem Solving activities. This technique can be tweaked to fit the needs of a particular class as long as the professor uses a little creativity. Dr. Miller, (a professor I had as an undergrad at the University of Victoria) started all of his Friday lectures with a bet. If the students knew in advance what he was going to teach that day they would know if it was a good bet or a bad bet. (For example he bet that in his class of forty-two students, at least two of them would share a birthday.) The students could either take the bet or try to explain why it was a bad bet. The remainder of the lecture would be spent explaining the mathematics behind why the bet was either good or bad. This approach had two benefits: first, it got the students attention; and second, it firmly established what the lecture that followed was designed to prove.

Understanding what fact(s) about the world a proof is trying to establish is a crucial first step, especially for a novice. It helps the student organize his thoughts and evaluate different possible paths if the end goal is clearly defined in his mind. In the same vein, understanding the purpose and value of education is a crucial first step towards long term success in Computer Science. If the student understands the long-term benefits, he or she is much more likely to devote the time and energy needed to master complex technical skills.

2.2 *Strategies in Upper Level Undergraduate Classes*

The issues in the lower level classes all still apply but there is another issue that is unique to the upper level classes: the need to build on an existing foundation that is assumed but may actually

not already exist. Efimba[3] points out "Some instructors approach prerequisite knowledge as an assumed asset of their students, only to find as they immediately delve into the concepts for their courses that prerequisite knowledge is more of a deficiency than an asset." The paper points out, "At first, it is easy to want to ascribe the problem to lack of coverage of pertinent material in the prerequisite course(s). However, the more likely cause is an amazing amnesia that leaves the students with very few of the important concepts learned in the prerequisite courses that are needed."[3]

Regardless of why the students do not have the assumed prerequisite skill set, most upper-level professors will eventually have a student that does not have the assumed prerequisite knowledge. Elliott[4] suggests a review of the prerequisite material before delving into new material, however that is not always possible for upper-level classes with a lot of prerequisites because that would not leave enough time to teach the new material.

In Computer Science, upper level classes often have group projects following the pattern laid out in the classic book "The 7 Habits of Highly Effective People"[2]. In that book, Covey[2] describes a progression from dependence to independence, followed by interdependence. This closely mirrors the ideal path of a Computer Science student: they start in lower level classes being dependent on the instructor and the textbook in order to acquire the basic proficiency in using computers and computing technology. As their knowledge base increases, they eventually achieve independence and can write solid code independently. Once they have achieved independence they are ready to be an interdependent contributor to a group. Covey[2] warns about the danger of jumping strait to interdependence since it has a superficial resemblance to dependence and dependence can often be mistaken for interdependence.

In Computer Science, one easy solution that I have found for determining the *independent* prerequisite deficiency is to start with a small, independent coding activity with a short deadline. The students with the required skills finish the project easily. The others are usually quick to identify themselves to the professor either directly or indirectly by not completing the assignment. On one particular recent occasion, this method revealed that almost 14% of the students in my upper-level Computer Science class did not have the skills to write a relatively simple piece of code. Most of the students had a good reason for not having developed coding skills, and once identified, I could give those students the extra attention they needed since they also agreed that they had a deficiency. At this level, the three important steps are to 1) identify deficiencies, 2) have the student agree to devote the time and effort required to overcome the deficiencies and 3) lay out a plan for the student's success.

3 Another Experience: From Mathematics to Computer Science to Work in Industry Back to Academia & Teaching Theoretical Computer Science (Testimony from Dr. Tosic)

In contrast to my colleague, my main interests from the childhood years were mathematics, chess and astronomy; of the three, I only systematically pursued the first interest (and, from early boyhood through my undergraduate university education but not since, to a lesser extent also the second). This lead to first completing a B.S. in Mathematics, and subsequently two different Masters degrees in Mathematical Sciences, prior to "converting" to Computer Science only in graduate school, after already having completed two M.S. degrees.

Once I found myself in a very demanding Computer Science graduate program, while I naturally gravitated towards "theoretical" courses, problems and research topics, my Ph.D. program requirements (at a leading research University in the US Midwest) also entailed a very thorough graduate-level training in all core areas of modern computing, including computer architecture, operating systems, and programming languages. This exposure to "the engineering side" of computer science (as opposed to its "mathematical side") turned out to be useful later on in my career – not the least reason being that, upon completion of my Ph.D. in CS from a Top-10 computer science research program in the United States (and in spite of having a double-digit number of research papers by that time, most of which as the first author!), I failed to secure an academic research position. That failure (or, what I certainly perceived as a failure at the time!) lead me to pursue a career in high-tech industry. Shortly after the PhD completion, I moved to Silicon Valley to join Cisco Systems as a software engineer; after about two and a half years (and three filed patents) with Cisco, I subsequently worked for several other companies, including a couple of high-tech startups. I returned to the academic world only fairly recently (2015), and have been both teaching and doing research in academia ever since.

Due to a combination of factors, ranging from the respect that the colleagues tend to pay to my mathematical background (or, perhaps, to the reputation of my PhD *alma mater*!) to the difficulties that the CS Department Chairs often have when it comes to finding instructors for theoretical computer science courses, in the three years since having returned to academia, I have mostly taught the foundational theoretical computer science courses, as well as some other, invariably "very mathematical" undergraduate (and, to a lesser extent, graduate) CS courses. Below, I briefly summarize some of the observations, challenges and key takeaways.

3.1 *Observations on Students' Preparedness for and Attitudes about Theoretical CS Courses*

In recent years, across two prominent public research universities in the US Pacific Northwest, I have taught four different courses with considerable mathematical components including proofs. Two of those courses were cross-listed, that is, taught jointly to undergraduate seniors as well as graduate students. The other two were core undergraduate courses, usually taken by the students during their junior year. The discussion that follows is going to mostly be based on those two core

courses: one on *Automata Theory and Formal Languages*, and the other on *Algorithm Design and Analysis*. At both Universities, it is necessary to complete both of those courses in order to receive a B.S. (or B.A.) degree in Computer Science. At Washington State University, a letter grade of C or better is required; whereas at University of Idaho, grade D is "good enough" (or, as both students and instructors like to remark only partially jokingly, "[at University of Idaho] D stands for Diploma"). At both institutions, the junior-level course on automata theory & formal languages was the first exposure to the theory of computing for an overwhelming majority of students. In case of Washington State University, a number of students in my class had taken that course at least once before, but ended up either dropping out, or getting a grade below C, and hence they needed to retake it in order to progress towards completing (in most cases) their B.S. in Computer Science. At University of Idaho, in contrast, as far as I am aware, all students were taking (my "edition" of introduction to the theory of computation) for the first time. I will briefly summarize the particular circumstances of teaching introductory theory of computing at each institution, and then discuss some common traits and insights from both institutions.

My class at Washington State University initially had about 60 students, of whom over 50 completed the course with a passing grade. The textbook I adopted is one of the two most widely used texts on introductory theory of computing, one by MIT professor M. Sipser[8]. While the vast majority of the students in this class were undergraduate CS majors (mostly juniors, and some seniors), I also had a handful of i) other undergraduate majors (ranging from Mathematics to other Engineering majors to Biology) and ii) graduate students whose advisors felt that theory of computing was a "deficiency" that needed to be addressed by (first) completing an undergraduate, introductory level course on the theory of computing, prior to taking the required graduate-level theoretical CS courses.

The levels of preparedness of these diverse students broadly varied, especially with respect to i) familiarity with and understanding of core concepts and principles of (discrete) mathematics and ii) ability to formally reason, and in particular, to understand simple-to-intermediate proofs from the lectures and the textbook, and, closely related to that, iii) be able to write simple mathematical proofs themselves. While it came as no surprise that, for example, the Biology majors in my class would find *any* mathematical proofs difficult, even some of the CS graduate students found proofs rather challenging. Moreover, I noticed this early – while reviewing the basics of discrete math and, in that context, "reminding" the students of, for example, how to prove by mathematical induction what is the (maximum) number of leaf nodes (or all nodes) in a binary tree. Many students claimed, they had done proofs by mathematical induction before; and a majority of the class nodded confidently, when I asked about the meaning and the application of the "Pigeonhole Principle". However, once actual concrete problems involving proofs using these two basic techniques were assigned, the outcomes were rather mixed. It therefore wasn't surprising, later in the semester, that many students struggled with understanding the (in)famous "Pumping Lemma" first in the context of Regular, and then later on, Context-Free (formal) languages. In particular,

while my (closed-book) exams always contained "cheat sheets" with the statements of Pumping Lemma(s), well under a half of the students were able to correctly apply this proof technique in order to establish that a given formal language (usually, of binary or ternary strings) actually is not Regular (or Context-Free).

The class I taught at University of Idaho initially only had 15 students total; however, those students were located at three different campuses of University of Idaho (the course was offered via video-conferencing to branch campuses; I myself commuted and gave live lectures at 2 out of 3 locations, including having office hours and ad hoc face-to-face interaction with students at those too locations – while making myself available via phone, email and Zoom/video-chat to the students at the third branch, which was simply too far for me or indeed all University of Idaho's instructors to drive to). The students at the three branches tended to generally have rather different academic backgrounds, as well. Students at the main campus have completed their first two years there, and were generally the best prepared of the three groups. Students at the remote branch were mostly graduate students, who were taking my class to overcome a deficiency; they all have seen and done mathematical proofs before – however, to somewhat of my surprise, their limited ability to write crisp, correct proofs even on basic discrete math problems (which, technically, were prerequisites for taking the theory of computing, as opposed to the content of my class that would be reasonable to assume that the students haven't seen before) made me wonder, about the quality of their prior training, esp. in undergraduate mathematics, and in particular in basic discrete math (including the elementary proof techniques).

The third group of students, located at a relatively recently opened branch campus, were generally transfers from a local community college, where they completed their Associate degrees. While I was given many assurances that that community college had been ensuring quality and rigor when it comes to Computer Science (freshmen and sophomore) courses in general, and their core Discrete Math (as the most important prerequisite for my intro to theory of computing class) in particular, my experience with these students was that, overall, they were the least prepared of the three groups. Some of them worked hard and did well, or at least performed OK. Some had very strong, intuitive conceptual minds, enabling them to compensate for the lack of thorough prior mathematical training. Some, however, were so underprepared that, in retrospect, I feel they were simply failed by their academic advisors, by being told to take a theory of computing course without having basically any computing background (programming or otherwise) to speak of, and likewise with little or no mathematical background.

Insofar as the choice of textbook and course material for this class, while I was given the usual "instructor's discretion", I was also strongly advised (as a new faculty member of the Computer Science Department at University of Idaho) to stick to the textbook that was used by virtually all their faculty who had recently taught this course. That text is the book by P. Linz[7]. Personally, I find the textbook by Sipser[8] to be better; however, certainly the case can be made that, for students

with less rigorous prior training, Linz[7] presents core ideas and concepts on automata and formal languages in arguably an easier-to-digest manner than Sipser[8]. Overall, while the choice of the primary textbook is certainly important, it is my strong impression that that choice isn't among the main factors determining whether a given student will or will not do well in this course.

The third theoretical advanced undergraduate CS course is the one on design & analysis of algorithms, which I am currently teaching at University of Idaho (spring semester, 2018). This class involves students at two out of three aforementioned University of Idaho's branch campuses, including about 50 students at the main campus and another 10 at the new branch campus. The textbook I am using is the one by A. Levitin[6]. The early impressions are, that most students find this course more directly relevant to the practical, engineering side of computer science, and more intuitive overall (incl. the "mathematical" aspects of esp. analysis of algorithms' time and space efficiency). Also, many of these UG junior and seniors are currently interviewing for internships and full-time positions in industry, and several have shared that some of the interview questions they have encountered are very directly related to the algorithmic design and analysis techniques covered in the class. While it is still too early to draw broader lessons, it certainly appears that the students have more appreciation for the algorithm design and analysis, than for the automata, grammars and formal languages.

What are, then, some common takeaways from teaching introductory theoretical computer science at these two Universities, to very diverse student populations? Most of the insights I share below primarily apply to the "hard-core" theory of computing (as opposed to algorithm design & analysis) classes, based on several semesters of combined teaching experience at Washington State University and University of Idaho.

Students' preparedness:

- One should not assume that, just because formal prerequisites have been met, a student actually has the (primarily, mathematical) skills necessary to do well in a theory of computing course.
- Students "having seen" (or even, presumably, having written) mathematical proofs before by no means implies, that those same students will be able to produce even the simplest of proofs on their own; in particular, a thorough review of core proof techniques (such as mathematical induction, proof by contradiction, and the Pigeonhole Principle) isn't merely a good idea: for a clear majority of the students at both institutions, it was indeed necessary.
- Relating highly abstract concepts from theory of computing to "the real" computer science (for example, parsing and lexical analysis in compilers) is generally both helpful and appreciated by the students – but may not be enough for the students to actually fully

appreciate why they have to "suffer" through (at least one) theory of computing undergraduate course, in order to get their CS degree.

Students' attitudes and mindsets:

- Most CS students do not like proofs, period. Moreover, many remain unconvinced, that formal reasoning and mathematical proofs are much needed, let alone necessary, for them to become good programmers, computer scientists and/or computer engineers.
- Automata theory and formal languages are found to be a very "dry" subject. To the extent that they want to deal with proofs and mathematical techniques at all, most students prefer the math involved in the course on *Algorithm Design & Analysis*, to those they encounter in the *Automata & Formal Languages* course.
- Most if not all CS students appreciate why they are required to complete an *Algorithms* course. However, it is my strong impression that (whether the students are comfortable speaking out their minds or not) a majority of these undergraduate students does not necessarily see why do they have to "suffer through" a hard-core theory of computing class (which is usually based on studying automata, grammars and formal languages).

3.2 *Making It Better: Some Thoughts on Possible Improvements to Teaching Computing Theory*

Undergraduate theoretical computer science courses at solid, second-tier public research universities tend to result in a rather mixed experience from both students' and instructors' standpoints. The key question for computer science educators, then, is how to make that experience more fulfilling and enjoyable for the students, without compromising on the necessary rigor or the core learning objectives associated with this challenging subject.

Thinking of this challenge as of a "constrained optimization problem" is important: by compromising on the rigor and sacrificing the core objectives, which is to train computer science & engineering students to reason formally about programs, computations and systems, one may well improve his or her student evaluation scores, as well as become "better liked" by those students looking for an easy way out. To do that, however, would be a betrayal to the profession, and ultimately also a let-down to those very same students, especially in today's highly competitive world of high-tech and global competition for the top engineering and computing talent. Ensuring that the core learning outcomes and objectives are met, therefore, must remain a "hard constraint" imposed on any curriculum and/or teaching methodology changes to how one delivers the theory of computing content to today's undergraduate students in general, and to aspiring computer scientists and engineers, in particular. (In our opinion, a significant part of the problem here is that many a computing or other engineering educator's career considerably depends on the student evaluations, especially if one happens to be a clinical/teaching faculty or a tenure-track, yet untenured, "research + teaching" faculty. While feedback from students on the quality of instruction and effort of their instructors is of utmost importance, it is my strong conviction that student evaluations decidedly should not be the sole criterion for evaluating an instructor's

teaching effectiveness – moreover, I would dare argue, it should not even be used as the primary criterion. Being liked by one's students is important, but it is not synonymous with providing quality education – especially when it comes to largely unpopular subjects, such as theory and formal methods of computing. Further elaborating on this important subject, however, is beyond the scope of the present paper.)

Based on a combination of my own teaching experience and numerous discussions with colleagues across the North American (and some European) colleges and universities, here is a tentative (and necessarily incomplete) list of some methodologies and alternative approaches that I intend to deploy moving forward, and that will hopefully move the needle of undergraduate theoretical CS education in the right direction.

- Relating computing theory to practice early and often; moreover, that "practice" should, in addition to traditional computing disciplines such as compilers and interpreters, also include the cutting-edge current technology trends, referring to tools, products, apps etc. that today's undergraduate students can most readily relate to. Indeed, unlike the CS undergraduates from 20 or 30 years ago, many contemporary students are largely unaware of how compilers and interpreters actually work.
- This is to a considerable extent due to the fact that, at many institutions (incl. Washington State University), undergraduate CS majors are not required to complete any coursework on compilers -- or else, if they are, they get exposed to compiling only after they have already "suffered through" the theory of computing, including but not limited to their first-ever exposure to concepts and methods that underlie parsing, lexical analysis, etc. in a theory of computing, and not programming languages or compilers, class setting (this latter scenario is the case with the undergraduate CS curriculum at University of Idaho).
- One should not assume that, just because the students have completed the prerequisites and "seen" (or even "done") mathematical proofs in the past, they will be ready to jump straight into the core theoretical CS topics including proofs of key results. Instead, a thorough review of both the key ideas in discrete math and the main proof techniques, even if it comes across as "boring", will very likely be very much appreciated by the majority of the students later in the semester (or quarter).
- In particular, it is perfectly fine to "refresh" student's understanding of e.g. mathematical induction or proof by contradiction on concrete problems that, properly speaking, "belong to" the discrete math prerequisites, as opposed to the content of the theory of computing course itself. Examples include illustrating mathematical induction on simple problems about binary (or other) types of trees, or explaining how proof by contradiction works using simple examples from elementary set theory or integer arithmetic. (One of my favorite examples is to demonstrate how arguing by contradiction works by showing to students that there are infinitely many positive integers that are primes.)
- Rigor of a very mathematical and formal subject should be creatively combined with "fun", including logic puzzles that emphasize key ideas from the theory of computing, various

online resources including but not limited to related courses available online (such as, e.g., Aaronson[1]), YouTube or other videos capturing key concepts or even providing (alternative) proofs of key results, etc.

- Even as formal and "dry" a topic as the Pumping Lemma(s) (for Regular and Context-Free languages) can be visualized, and studying the related material made interactive. There are some good YouTube videos nowadays on even the most formal and technical of the topics, that students tend to appreciate at least insofar as the visualization and conceptual understanding of the difficult, mathematically rigorous ideas and topics. Similarly, an interactive, 2-player-game with alternating moves alternative presentation of specifically the Pumping Lemma(s) appears to be more appreciated by some students than the classical way of introducing these concepts.
- Based on my own experience as well as feedback from the more senior colleagues, I intend to add more of both video/visualization based contents and interactive ("let's play a game") approaches to my future delivery of theory of computing and algorithm courses.
- Lectures on a very formal subject should be interrupted "Chicago style", that is, early and often, with questions to the audience; this ensures both that students "stay awake" (and hopefully also alert), and that they get a real-time feedback on whether and how well are they following the pace of the lectures and the material being covered.
- It appears that a larger number of relatively small assignments works better for most students than a smaller number of large assignments (although, this hypothesis does require additional testing and evaluation in practice).
- Funny stories and logical puzzles, especially when used in-class (if necessary, as brief digressions from covering the "core material") and if judiciously chosen, will make students think and do some introspection on how thorough their understanding of the important concepts really is. One of my favorite examples in this context is the famous "All horses are of the same color" riddle, testing students' understanding of the principle of mathematical induction.
- While the most common practice nowadays appears to be simply posting the solutions to homework assignments online, I have found that discussing at least some of the more challenging problems from assignments during the lectures is helpful in enforcing students' understanding of the core ideas and concepts that those assignments are supposed to test.
- While indeed we all have syllabi to cover, and covering all designated topics is arguably particularly important in case of the "core" undergraduate courses, it is my experience that incorporating some "live" discussion of solutions to assignments is generally appreciated by a majority of the students, and the benefits of such "detours" tend to outweigh the down-side related to slower coverage of new course material. Of course, it is important to not get

too “off-course” with such detours, and also to connect the review of presumably familiar concepts and esp. mathematical techniques with concrete problems as soon as possible.

4 Discussion and Conclusions

From the discussion above, it is clear that both professors (Dr. Beeston and Dr. Tosic) took fairly similar paths to the same destination where they are now working as colleagues, but they had different motivations for choosing that path.

Despite their different motivations, there are some points on which the two authors agree on completely. The most important thing they agree on is that students are often not prepared by the prerequisites to a course, yet the unprepared students believe they are. They will nod confidently when you ask if they have mastered previous concepts, but only when an assignment is given are these deficiencies revealed.

Both professors also agree that the optimal learning path includes a large number of small assignments rather than fewer large ones. For project based courses Dr. Beeston includes milestones that are no more than a week apart, with the possibility of non-fatal failure of the earlier milestones, on which the later ones are built. One challenge here is, that usually the theoretical and foundational CS courses are not project-based; hence, homework assignments, quizzes and mini-tests are the most common ways of identifying early which students and in what specific areas are most likely to have difficulties with the course material.

Both professors also agree that rigor should be combined with creativity and fun. Often this means introducing a topic with a strategic *mistake*. Dr. Tosic uses the “all horses are of the same color” to test students’ understanding of mathematical induction. Dr. Beeston starts with having the students write a program that results in data corruption because two threads are updating the same unprotected data. In both cases, these *mistakes* help the students better understand the underlying issues in a way that is entertaining and retainable.

A more abstract agreement comes from the connection between the use of cheat sheets on exams and the difference between training and education. Dr. Tosic found that a cheat sheet with statements of *Pumping Lemma(s)* was not sufficiently useful to students who did not have the education to leverage that knowledge to formulate a proof that a given formal language was not *Regular*. This mirrors Dr. Beeston’s assertion that a student is far more equipped to tackle real world problems if they have the education. Cheat sheets covering syntax or protocols are readily available, but the education needed to apply those “training documents” is a valuable (and apparently fairly rare!) talent.

Both professors have taught (and are currently teaching) Computer Science at the University of Idaho, their classes usually straddle three separate branch campuses; and both faculty have willingly chosen to commute between the two close(st) campuses of the University of Idaho in order to enhance the students' learning experience. As Dr. Beeston stated, even the best video in some situations cannot replace a good mentor. Dr. Tosic accentuated this point by adding the component of live discussions of challenging previous homework assignments. Dr. Beeston also had the issues discussed by Dr. Tosic that the Department had (de facto) chosen the textbook for the class, even though (in our opinion at least) there were better textbooks available. Both of us agree that the choice of textbooks is not an insurmountable issue, but it is a large part of the students' learning experience, and hence textbooks should be chosen carefully.

Both instructors found that a large number of students were underprepared by their prerequisites, but Dr. Beeston did not see a clear division between the students at the different branch campuses of the University of Idaho. Instead, Dr. Beeston saw the same distribution of students on both campuses. Dr. Beeston asked one student who was excelling in the course how he seemed to retain the information from his prerequisite courses so much better than his peers. He replied, "I am a good student, and a good student does not need a good teacher. Just tell me what material you want me to learn and I will learn it no matter how you teach it." Dr. Beeston found the student's comment demotivating since Dr. Beeston's primary goal is to be a good teacher and this student did not think that was valuable or even necessary. Dr. Beeston had to agree that for an exceptionally bright student a good teacher may not be as important as it is for an average student; however, for the vast majority of students, a good teacher makes a big difference.

Both professors agree that the qualities of a good teacher include motivating the students by helping them understand the connection between complex theoretical concepts and real world applications. Professors are indeed trying to solve a "constrained optimization problem". The rigor of the course cannot be lost without betraying the trust that the students and the industry have placed in the university system. On the other hand, human nature is to follow the path of least resistance unless there is a perceived better destination at the end of another path.

It would be rare indeed to find an accomplished musician whose first exposure to music was the (often) tedious first lessons on how to play scales. In the same way it would be rare to find a student who masters computer theory if their only exposure to it is boring lectures with no context. In order for students to navigate the rigors required to gain a quality education, they need a vision of how this knowledge will help them in the future.

References

1 Aaronson, S. *6.080 Great Ideas in Theoretical Computer Science,* Spring 2008. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu License: Creative Commons BY-NC-SA

2 Covey, S.R. *The 7 Habits of Highly Effective People*, Simon & Schuster, New York, 1989.
3 Efimba, R. E., & Rhoulac Smith, T. (2012, June), "Prerequisite Courses and Retentivity as a Challenge", Paper presented at 2012 ASEE Annual Conference & Exposition, San Antonio, Texas. https://peer.asee.org/21821
4 Elliott, C. S., & Winn, A. (1998, June), "Building An Industry Academic Engineering Education Consortia: Some Myths and Realities", Paper presented at the ASEE 1998 Annual Conference Seattle, Washington. https://peer.asee.org/6947
5 Karabulut-Ilgu, A., & Yao, S., & Savolainen, P. T., & Jahren, C. T. (2016, June), "A Flipped Classroom Approach to Teaching Transportation Engineering", Paper presented at the 2016 ASEE Annual Conference & Exposition, New Orleans, Louisiana. 10.18260/p.26317
6 Levitin, A. *Introduction to The Design and Analysis of Algorithms*, 3rd edition, Pearson, 2012
7 Linz, P. *An Introduction to Formal Languages and Automata,* 6th edition, Jones & Bartlett Learning, 2016
8 Sipser, M. *Introduction to the Theory of Computation*, 3rd edition, Cengage Learning, 2013