Pedagogical Considerations

Introduction

What are some current trends in the teaching and learning of computer science? What are the controversies of the day in terms of the pedagogy of computer science education? In this section, a top few trends, controversies, and challenges have been listed for each knowledge area as well as the curriculum as a whole. These issues are expected to influence the future evolution of computer science curricula.

Curriculum-Wide Considerations

The adoption of CS2023's recommendations presents numerous challenges at both the macro and the micro level. Some of the macro considerations include the following.

- Active learning is an important component of any computer science course – doing helps learn computer science. Courses that use interactive electronic resources (e.g., eBooks, python notebooks) are a significant improvement over the traditional lecture-based courses that do not involve any active learning component – they provide ample opportunities to learn by solving problems. In this regard, it is important to emphasize that ideally, active learning should cover the entire gamut of skill levels – not just *apply*, but also *evaluate* and *develop*.
- The success of generative AI systems is causing faculty to reconsider their approach to assessment. Student grades are increasingly being based on in-class, closed-book assessments. Care should be taken not to lose sight of the objective of assessment when using expedient assessment techniques.
- The landscape of computer science textbooks is in flux. The number of traditional publishing houses is shrinking, the cost of their offerings is increasing, and the vast majority of these resources are primarily only in English – though generative AI systems are making important inroads in providing accurate translations. Digital rentals, while cheaper, usually disappear at term's end and are unavailable as future references. Open Educational Resources (OER) present a cost-effective alternative, but often lack formal review. Compounding this are students who, for financial and generational reasons, forgo acquiring the selected text(s) in favor of self-selected YouTube videos.
- Broadening participation in computing is an ongoing concern in many contexts. Adopters of CS2023 should remain cognizant of how their curricular and pedagogical choices affect this important issue.
- Studying abroad is a quintessential high impact learning experience. One potential impediment to its wide-spread adoption is curriculum design. This can include long prerequisite chains and dense curricula. Care should be taken to leave room for, if not encouraged, penalty-free study abroad.

Considerations by Knowledge Area

Artificial Intelligence (AI)

- A balance must be struck between 1) the need to study fundamental issues of search and other approaches that are still in widespread use and 2) the desire to focus on cutting-edge AI and machine learning methods.
- Given AI’s current and future potential for societal impact, educators should ensure that all students are well-versed in the ethical and societal considerations and implications of applying AI methods.
- Since AI is rapidly evolving as a field, it is challenging to create a curriculum that will remain current for long. Consequently, the onus is on the instructors to keep up to date with current methods and use their judgment in determining what to teach in order to keep their courses current.

Algorithmic Foundations (AL)

- Should computer science graduates be able to explain, at some level, algorithmic approaches making headlines in the popular press, such as Blockchain, SHA-246, and Quantum Computing algorithms? Topics focused on these, and other algorithms, received less than 50% support in the community survey of Algorithmic Foundations area. Consequently, they were not listed as CS Core topics. However, if computer science graduates are expected to explain these algorithms to non-technical members of the society, they should be exposed to these topics somewhere in the curriculum.
- Students are unlikely to implement a lot of historic algorithms in industry such as Bubble sort. A question that merits case-by-case consideration is whether these algorithms should be covered in the curriculum and, if so, the level of skill at which they should be covered.

Architecture and Organization (AR)

- Software tools for open hardware have lowered the cost and complexity of understanding the design of new processors. The community-driven RISC-V open instruction set architecture “democratizes” the design and evaluation of processors and presents a cutting-edge opportunity in this regard.
- Quantum Computing does not yet have a fully standardized interface between software and hardware. What should be the minimum set of Quantum Computing topics a computer science graduate should be able to explain?

Data Management (DM)

- For the most part, students write code that either reads/writes to a file or is interactive. Yet, in industry, most data are obtained programmatically from a database. This is a source of mismatch between academic preparation and industry expectation.
- Even though most databases are relational, NoSQL databases are enjoying a significant degree of popularity. Balancing the coverage of relational vs NoSQL databases is a concern for curricula.

Foundations of Programming Languages (FPL)

- Shell scripting is a skill that students should master to automate laborious tasks. It is also a helpful tool to coordinate the work of multiple applications. An interesting curricular exercise is determining how and when to teach it as a paradigm.
- There is an increasing need to develop large, complex software systems that have the potential for catastrophic failure (e.g., software driving medical devices such as robotic surgery). Such software needs to have its behavior validated and potentially formally proved correct. As a result, formal methods may be more important in the future. This would require greater mathematical skills and ability in graduates.

Mathematical and Statistical Foundations (MSF)

Faculty and students alike have strong opinions about how much and what mathematics should be included in the CS curriculum. Generally, faculty, who themselves have strong theoretical training, are typically concerned about poor student preparation and motivation to learn mathematics, while students complain about not seeing applications and wonder what any of the mathematics has to do with the software jobs they seek. Even amongst faculty, there is recurring debate on whether calculus should be required of computer science students, especially in light of the impact calculus failure rates have on broadening participation. Yet, at the same time, the discipline has itself undergone a significant mathematical change – machine learning, robotics, data science, and quantum computing all demand a different kind of mathematics than is typically covered in a standard discrete structures course. The combination of changing mathematical demands and inadequate student preparation or motivation, in an environment of enrollment-driven strain on resources, has become a key challenge for CS departments. Some recommendations that have been presented for the treatment of mathematics in computer science programs follow.

- Requiring *PreCalculus* as a prerequisite for discrete mathematics will ensure that students enter computer science with some degree of comfort with symbolic mathematics and functions.
- Studies show that students are motivated when they see applications of mathematics. It is recommended that minor programming assignments and demonstrations of applications of mathematics be included in computer science courses.
- Institutions should adopt preparatory options to ensure sufficient background without lowering standards in mathematics. Theory courses can be moved further back in the curriculum to accommodate first-year preparation. Where possible, institutions can offer online self-paced tutoring systems alongside regular coursework.
- What is clear, when looking forward to the next decade, is that exciting high-growth areas of computer science will require a strong background in linear algebra, probability, and statistics (preferably calculus-based). And as much of this material as possible should be included in the standard curriculum.
- Educators and institutions are often under pressure to help every student succeed, many of whom struggle with mathematics. While pathways, including computer science-adjacent degrees or tracks, are sometimes created to steer students around mathematics requirements towards software-focused careers, educators should be equally direct in explaining the importance of sufficient mathematical preparation for graduate school and for the very topical areas that excite students.

The better approach is to invest in remediation courses to sufficiently prepare as many students as possible.

Networking and Communication (NC)

- To what extent should students learn the very low-level details associated with networking or should higher levels of abstraction be the norm in teaching?
- Cutting-edge technologies promise to significantly affect networking. Generative AI might benefit the generation of networked configurations, security assessments, and capacity planning. Quantum computing may significantly affect the teaching and practice of networking. And the same goes for emerging communication technologies like 5G.

Operating Systems (OS)

- How do we teach operating systems as a cohesive set of functions when OS functions are increasingly embedded in architectures or distributed within software development frameworks?
- Educators should continue to emphasize the importance of operating systems knowledge in distributed, parallel, and secure applications.
- Is there value in having students recreate operating system functions as a pedagogical approach or should the focus be on a student's ability to reason about the performance of off-the-shelf library modules that compose applications?

Parallel and Distributed Computing (PDC)

- Should parallel and distributed programming be infused across a curriculum?

Software Development Fundamentals (SDF)

- Students will still need to be able to produce code. How they go about doing so may change rapidly and dramatically with improvements in the capabilities of generative AI. Students should also be able to read, critique, and verify the correctness of code — abilities that students will need if generative AI is used to write code.

- Assessment practices in introductory programming courses will need to be adapted to take into account the availability of generative AI. How remains to be seen. It is foreseeable that currently popular assessment approaches such as drill-and-practice, many-small-problems, and written Explain in Plain English (EiPE) assessments, will need to change or be utilized differently. A focus on what today would be considered 'alternative' means of assessment may rise in prominence—for instance, oral examination, code modification, and other difficult-to-accomplish (with generative AI) assessment schemes.
- High-level approaches to teaching and learning introductory programming may need to be dramatically rethought. Approaches similar to studio models of learning from the fine arts where students design, show, explain, and critique work made in the studio are a way to engage students with topics that let them express their own ideas and vision while learning about fundamental topics.

- The order in which Software Development Fundamentals (SDF) topics are discussed may need to be reconsidered. For instance, starting with syntax and creating increasingly complex programs from scratch may be replaced by concepts-first approaches where modularity is considered from the beginning, and syntax and other more basic constructs such as conditionals and repetition are learned during the process.
- New skills such as prompting (prompt engineering) and the use of other generative AI tool requirements/features may in the near future be considered to be basic programming skills. How the industry adopts generative AI may be a leading driver in such arenas.

Software Engineering (SE)

- Are undergraduate programs in computing that rely on a *single* team project beneficial? Teamwork is ubiquitous in industry, but no teams are formed entirely out of people with similar backgrounds who have no prior experience working on a team. The heterogeneous background and experience level of real teams is fundamentally different from the comparative homogeneity of students in a class.
- Do students have sufficient opportunity to practice with open-ended problems, where the choice of tools and approach are critical? In a software engineering context, most work involves evaluating tradeoffs – between space and time, between speed-of-implementation and runtime optimization, between Do-it-Yourself (DIY) and Commercial-off-the-shelf (COTS) or Open-Source Software (OSS) approaches. Are students given enough opportunities to practice the critical decision-making skills necessary to succeed in a professional environment?
- Software developed in a team setting (software engineering rather than programming) is more likely to have an impact on society for good or ill. At the same time, teamwork means no single person may be responsible for the impact of the software—the larger the project, the greater the potential impact, but the less responsible any team member feels about the impact. How do we instill among students a proper sense of responsibility for the whole solution regardless of the size of one's contribution to it?
- Formal methods for software validation will pay off (substantially) later in students' careers, but it is a perfect-but-infrequent solution. The current approach to validation focuses on attempts to get high-fidelity evidence (unit tests) over proofs and other formal methods, which is more immediately useful but fails to expose students to interesting long-term ideas. Given the finite resources provided for software engineering education, educators must strike the right balance between these two approaches.

Security (SEC)

- An ongoing pedagogical consideration is inculcating a security mindset among students, so that security is a goal from the start and not an afterthought.

Society, Ethics, and the Profession (SEP)

- Is an introduction to ethical thinking and an awareness of social issues and their emerging professional responsibilities sufficient for our graduates to act ethically and responsibly? If not, does that put the burden on instructors to not only lead discussion about the pressing questions of the day but also weigh in on those matters? How should that be done? Will we just be imparting our own biases upon our students? Should this be avoided? If so, how?

- CS curricula regularly require prerequisite courses in mathematics. Should there be parallel prerequisite requirements in moral philosophy to prepare students for future course work in performing ethical analysis?
- How could we weave SEP throughout the curriculum in practice? Is this realistic? How much coordination would it take? How less optimal is it to have a standalone ethics course only? Is there another model in between these two extremes (neglecting the extreme of not having any coordinated or targeted SEP content in our courses)?
- Educators naturally possess real or perceived authority when it comes to technical issues. Many educators believe they lack this authority when it comes to SEP topics. How can CS instructors be empowered to effectively incorporate SEP topics into their courses?
- How can we effectively impart the core values of diversity, equity, inclusion, and accessibility? How is this best done in a computer science context? How can we effectively impart the core values and skills of being a computing professional into our students' education? Are engineered "toy" projects a suitable context for these? Are study abroad opportunities/work-placements/internships better? Are they worth the cost? Should we put more focus on efforts to have more programs/degrees contain study abroad opportunities/placements/internships?
- Should software developers be licensed like engineers, architects, and medical practitioners? This is an older debate but given the impact of software systems (akin to safe bridges, buildings, etc.), maybe it is time to revisit this question.
- What would a set of current SEP case studies look like and how could they be employed effectively?
- How can collateral learning be leveraged to improve the learning and appreciation of societal, ethical, and professional topics?

Specialized Platform Development (SPD)

- Computing is no longer limited to traditional desktop applications. Students need to learn how to develop software solutions for various platforms, including web, mobile, IoT devices, and emerging platforms like virtual reality (VR) and augmented reality (AR). A well-rounded curriculum should provide opportunities for learning on multiple platforms.
- With the increasing demand for cross-platform apps, educators should teach students how to develop applications that work seamlessly across different operating systems and devices using technologies such as React Native, Flutter, or Progressive Web Apps (PWAs).
- Cloud platforms and services, such as AWS, Azure, and Google Cloud, have become integral to modern platform development. Students need to learn how to deploy, scale, and manage applications in the cloud.

Systems Fundamentals (SF)

- How deeply should instructors elaborate on the design principles of computer systems in undergraduate courses?

- What role should generative AI play in system-related knowledge areas, not only systems fundamentals, but also architecture and organization, network and communication, operating systems, and parallel and distributed computing?
- Should instructors link knowledge units from systems-related knowledge areas with those from applications-related knowledge areas? And if so, how?