A Novel Self-Paced Model for Teaching Programming

Jeff Offutt, Paul Ammann, Kinga Dobolyi, Chris Kauffman, Jaime Lester, Upsorn Praphamontripong, Huzefa Rangwala, Sanjeev Setia, Pearl Wang, and Liz White
George Mason University, Fairfax VA, USA
{offutt,pammann,kdobolyi,jlester2,upraphamsetia,pwang,white}@gmu.edu {kauffman,rangwala}@cs.gmu.edu

ABSTRACT

The Self-Paced Learning Increases Retention and Capacity (SPARC) project is responding to the well-documented surge in CS enrollment by creating a self-paced learning environment that blends online learning, automated assessment, collaborative practice, and peer-supported learning. SPARC delivers educational material online, encourages students to practice programming in groups, frees them to learn material at their own pace, and allows them to demonstrate proficiency at any time. This model contrasts with traditional course offerings, which impose a single schedule of due dates and exams for all students. SPARC allows students to complete courses faster or slower at a pace tailored to the individual, thereby allowing universities to teach more students with the same or fewer resources. This paper describes the goals and elements of the SPARC model as applied to CS1. We present results so far and discuss the future of the project.

ACM Classification Keywords

K.3.2 Computer and Information Science Education: Computer science education

Author Keywords

Scaling CS1; Active learning; Gender and diversity; Self-pacing; Online learning; Collaboration; Peer learning

1. INTRODUCTION

This paper introduces a project that is attempting to significantly scale our ability to teach introductory programming to more students with the same or fewer resources by re-inventing the way we teach CS1 (CS 112 at George Mason). In addition to responding to the recent enrollment surge [3] by increasing the capacity of our CS1 courses, we have additional goals of retaining more students, increasing learning with less effort, and reducing or eliminating cheating. The Self-Paced Learning Increases Retention and Capacity (SPARC[1]) project blends self-pacing, separation of practice and assessment, collaborative and peer learning, automated grading, and flipped classrooms to create a more efficient and effective way to teach introductory computer science.

[1]http://sparc.cs.gmu.edu/

L@S'17, April 20-21, 2017, Cambridge, MA, USA

ISBN ???.

DOI: ???

George Mason University is a 34,000-student state-supported university. As at many universities, our introductory computing courses are under great stress. Explosive enrollment growth has doubled our undergraduate enrollment in the last five years, with our introductory classes tripling since 2012. Mason also accepts significant numbers of transfer students, mostly from community colleges, including 80 transfer CS majors in fall 2016. Although great news for the labor shortage in the software industry, this dramatic national enrollment increase puts enormous stress on universities in a time of declining state funding.

The increase in enrollment in CS1 and CS2 has not been matched by increased TA support, so current TAs are overloaded. Some TAs report spending as much as 50% more hours per week than they are officially required to work. Furthermore, there is a ripple effect—more TAs are being pulled into our introductory classes every semester, leaving less support for upper-level and graduate courses.

A particular concern is when students fail. From fall 2009 through summer 2015, 37.7% of our students failed CS1 at least once. We consider a failure to be a C- or below, since students must have a C or better to take the follow-on course (CS2). This includes most students who are accused of plagiarism. National studies report that average six-year graduation rates across higher-education institutions is only 59% and have remained relatively stable over the last 15 years [2]. Requiring students to repeat classes, or worse, leave college without graduating, has high human and monetary costs. It can even deprive students of the economic benefits of a college degree, which can exceed $1 million lifetime and even higher in STEM fields [1].

In a pure economic sense, failing students are also expensive to the system as they consume extra resources. If they change majors, leave the university, or otherwise do not re-take the course, then the resources they used when taking the course are lost with little benefit. If they repeat the course, then they consume course resources twice. Worse, most students who fail learned part of the course material the first time, so they are in effect required to learn that material twice, at a cost to both students and the university.

Computer Science majors at our university are a diverse group comprised in fall 2015 of 46% European-Americans and 26% Asian-Americans. Several other ethnicities constitute the remainder. Although the number of female students is increasing, only 15% of Mason's CS student majors are female. Worse, the percentage of female and minority students decreases through the four years of college.

Mason's CS1 instructors catch up to 20% of students cheating. Most are copying programs from classmates or other sources. This is only what we find—it is impossible to know how many cheating students are not caught. Cheating imposes huge costs. Although alerts are raised through automatic similarity checking programs, instructors still have to deal with each case individually, spending time and emotional energy. As enrollment continues to increase, these costs are becoming unsustainable.

Our novel educational pedagogy is being designed and implemented by a diverse team of faculty and students. The team includes CS1 instructors, an educational specialist, a programmer, our department chair and undergraduate studies associate chair, and very experienced instructors.

Challenges

The overarching goal of the project is to improve the capacity of CS1 while maintaining, and hopefully improving, educational quality. For the purposes of this paper, we define a *traditional* introductory CS course as CS1 that lasts for a term (usually semester or quarter); assigns for-credit programs that are done out-of-class with specific deadlines, while allowing little or no collaboration; have class sizes dictated by resources including classroom size and instructors; and uses most of the class time for instructor presentations. We present several challenges faced when using this traditional pedagogy, each of which leads to unnecessary use of student time and university resources.

Challenge 1: Using practice problems for assessment wastes instructor and TA resources. Students need a lot of programming practice, and they need to be assessed on their programming skills. Traditional pedagogy mix practice and assessments by giving students programming assignments that are used for practice as well as for grades (assessment). Many students learn programming best collaboratively, where students work together and help each other develop those skills [10]. Using the same assignments for both practice and assessment forces students to choose between working alone (which can lead to unnecessary struggles and less learning) and working together (cheating). The traditional pedagogy requires instructors and TAs to spend time searching for and penalizing such cheating. Besides wasting time, this inhibits student learning, and can discourage some from pursuing a CS major, especially under-represented minority and female students, who have been found to learn better with collaborative approaches [7, 9].

Challenge 2: Traditional introductory CS courses do not adequately support students with diverse learning styles. It is well known that students learn material at widely varying paces: some faster than the traditional curriculum and some slower. However, the traditional pedagogy requires all students to move at exactly the same pace. This puts slower learners at a severe disadvantage while making classes painfully slow and unchallenging for faster learners.

Challenge 3: Traditional introductory CS courses do not adequately support students with diverse backgrounds. Students start these courses with dramatically different amounts of computing knowledge, programming skills, and general academic knowledge. Because of prior preparation, fewer students every year need all the material in these courses. While many students test out of one or both courses completely, every year hundreds of moderately-prepared students are required to take an entire introductory course, even though they only need part of it. Repeating material is not only a poor use of their time, it is a poor use of university resources. In contrast, less prepared students rarely receive enough instruction, hands-on teaching, and skills practice to master the course material. These students often fail CS1, not because they cannot learn it, but because they cannot keep up. Neither student population is well served by the traditional *one-size-fits-none* approach to early CS education.

The traditional model of 3-hour, semester-long, lock-step courses forces students into boxes that all move at the same speed, regardless of how fast the students master course material. This has been called a "conveyor-belt" or "factory" education model [5, 6]. The conveyor-belt model often drives talented students away, and we suggest that it reduces capacity and throughput in introductory CS classes.

Pedagogical Elements

Our overall goal in this project is to address these challenges with an approach that reduces overall costs without sacrificing educational quality. This is accomplished through several major elements:

1. Separating individual assessment activities from collaborative practice activities (*challenge 1*).
2. Permitting students to demonstrate mastery of skills at any time (*challenge 2 & 3*).
3. Allowing students to pace their own learning, decreasing the need to retake courses (*challenge 2 & 3*).
4. Moving lectures online and using contact hours for interactive teaching (*challenge 2 & 3*).
5. Automating, as much as possible, evaluations of students' programming skills and computing knowledge (*challenge 1*).

2. THE SPARC MODEL

We are creating an innovative teaching model of **self-paced** introductory programming courses. Students periodically demonstrate competency with skills, similar to earning martial arts black-belts. While the self-pacing aspect of martial arts studies are important, we explicitly downplay competition to encourage retention and diversity. Flipped learning [11] has been used successfully in prior introductory CS courses. Our self-paced learning model adapts best practices from these different learning paradigms.

The rest of this section describes the major elements in the SPARC model for teaching.

Practice assignments: The courses is divided into 10 *stages*. For each stage, students are given many practice assignments to be done collaboratively, at their own pace, and using any resources desired. Students are not just **allowed**, they are **strongly encouraged** to help each other. Instructors, GTAs, and undergraduate TAs are in classrooms, hold office hours

in public places, and are available through online discussion boards (we use *piazza.com*).

Assessments: Since our students learn in a less rigid schedule, we replace the traditional assessments such as out-of-class individual programs with anytime assessments. When ready, students appear in scheduled labs to attempt an assessment. Our automated system contains about 10 versions of each assessment (a mix of programming assignments and concept questions), and on each attempt, a student is presented with one version chosen randomly and without replacement. The assessments are supervised and use locked-down computers (no access to the web or other materials unless explicitly authorized). The results are graded immediately with automated tests and an automated coding style assessment. Students have up to five chances to pass each assessment, and are not penalized for repeated attempts. To pass the class, students must eventually pass all assessments with a score of 70% or higher.

Classroom activities: We free instructor resources to teach more students by using the *flipped classroom* model [4]. Most knowledge is delivered online through self-recorded lectures, tutorials, and other resources. Class time is primarily used to practice programming, which has three elements. (1) Students collaboratively practice programming skills with help from peers, the teacher, and TAs. When possible, we prefer rooms that have multiple whiteboards, movable chairs, and desks that facilitate group-work. (2) The teacher and TAs work directly with students to solve problems, answer questions, and offer advice. (3) Students work on material and programs related to their individual current stage, forming groups dynamically of students at the same stage.

Small group discussions are led by instructors and TAs. Discussions are often in the form of *mini-lectures* where the instructor or TAs explain core concepts needed to solve the programming problems. Mini-lectures usually take 10 to 15 minutes and involve small groups—a type of *just-in-time* learning [8]. Instructors and TAs sometimes start discussions based on student questions. Students then collaborate to design algorithms to solve the problems. As students become familiar with the teaching model, they form study groups by themselves.

Assessment labs: Students use our software to register for assessments weekly, where they get a new, random assignment for their current stage. TAs are present to help students understand the assignment, solve syntax problems, understand the language, and facilitate record keeping, but **not** to solve the problem. After completing an assessment, the automatic grading program gives immediate feedback in the form of which automated tests pass and which fail. Students may resubmit the assignment during lab until they pass it. If a student fails to pass that assessment during the lab, the instructor or a TA helps the student understand what was wrong to improve for the next attempt.

Grading: The overall grade is 70% assessment scores, 20% final exam, and 10% participation. Students who do not finish by the end of the term are given an *in-progress* grade, and have 10 weeks after the end of the term to complete.

Software: To support our educational model, the SPARC team is developing custom software that will be shared with the community in an open source model.

3. RESULTS AND PLANS

This project started with a grant in spring 2015 and the team has developed over 100 practice problems and 75 assessment problems with corresponding automated tests for CS 112. At Mason, CS 112 is taught in Python, and includes programming, problem solving, testing and debugging, and the use of program documentation[2]. We have taught four sections to a total of 344 students, including 78 CS majors. CS1 is required by most STEM majors at Mason, and most CS majors took the equivalent course in high school. To demonstrate scalability, we have progressively taught more students and sections with the same instructor. The SPARC model is allowing her to teach more students with less effort.

Our most dramatic result thus far is that the number of students caught cheating has gone from 10% on average, and up to 20%, down to zero. **We have not caught a single student cheating.** Collaboration is encouraged rather than proscribed, and assessments are done individually in an environment engineered to make undetected cheating extremely difficult. This is not just a huge savings in time and effort, we also eliminate the hugely destructive "false positives," that is, we do not erroneously accuse students of cheating.

Our completion rates have been at least the same as, and sometimes dramatically more than, the pass rates in non-SPARC sections. This is partly because of the self-pacing, which is allowing 10% to 25% of students to finish after the end of the semester, and partly because the class environment puts students in a more positive frame of mind. By comparing final exam scores among courses, we have also verified that students who pass SPARC sections are learning at least as much as students who pass non-SPARC sections.

Anecdotally, the instructor reports much more pleasant interactions with students in SPARC sections. She feels more effective as a teacher, and reports higher job satisfaction. She feels this model is more stimulating as she was interacting with students directly rather then talking to them. The TAs report similar views, and said they enjoy working for a SPARC class more than other CS 112 classes, and had better interactions with the students.

We are starting to extend the SPARC model to CS2 (CS 211 at Mason).

We are also computing longitudinal data to assess how the SPARC teaching model affects students as they progress into subsequent courses. We are measuring effort by the instructors, scalability, performance of students, number of students who finish early and late, and retention to compare with students who did not participate in SPARC courses. This will be used to evaluate several experimental hypotheses: (1) SPARC students learn at least as much as students in our regular courses; (2) this model allows us to teach more students with less effort;

[2]A SPARC CS 112 course website can be found at https://cs.gmu.edu/∼kdobolyi/sparc/.

(3) this model helps retain more students; and (4) this model helps retain more female and minority students.

4. DISCUSSION AND LESSONS LEARNED

We believe the this model of teaching introductory CS courses has potential to increase capacity, retention, and learning. A major effect is that it frees instructors and TAs to be *teachers*, instead of talkers, graders, cops, judges, and managers.

The effect of the undergraduate teaching assistants (UTAs) has been enormous and helpful beyond expectations. They learned, the CS1 students enjoyed having knowledgeable peers to work with, and they all developed useful collaborative skills. UTAs are significantly cheaper than instructors and GTAs, so this is a major accelerator for our goal of scaling our ability to teach more students with fewer resources.

The SPARC model is increasing our capacity to teach more students in several ways. Eliminating cheating frees up substantial instructor time to teach more students. It also means fewer students have to repeat the course, freeing up even more resources. Many students can learn CS1 material, but need more than a semester. In past years, such students failed the course the first time, and either took it again (expensive for them and us!), or gave up (also expensive). Every student who completes past the end of the semester saves money for the university, saves instructor time, saves their time, and increases their confidence. Another savings in instructor time comes from the collaboration, which in effect lets students teach each other. Finally, the automatic grading frees up instructor and TA time. Every hour saved allows the SPARC model to scale to teaching more students.

We continue to face challenges. We need more classrooms that facilitate group work, which our university is slowly creating. Hiring lots of great UTAs is also a challenge that we are still learning how to solve. We are also staffing labs between terms (winter and summer break). This is currently funded out of our grant, but will eventually need to be funded internally.

5. CONCLUSIONS

Ultimate success of this project depends not just on these innovations working for us, but whether this model is used successfully elsewhere.

The effect that we are most happy about is that this approach has completely changed classroom atmosphere and student-teacher relationships. The students feel the teacher wants them to learn, and they believe that they can. When students believe the system is designed to make learning difficult, they become discouraged, lose confidence, and are more likely to quit or cheat. The SPARC teaching model disrupts this dynamic in such a way that we hope will lead to greater retention, especially among female students and under-represented minorities.

The SPARC model also dramatically decreases instructor's workload. We are currently increasing class sizes without making instructors suffer or impacting learning. This effort is already indicating that the SPARC model of teaching can go a long way towards reaching the goal of learning at scale.

ACKNOWLEDGMENTS

We want to thank all of the UTAs for making this course possible. We also want to acknowledge support from our entire department, especially the other CS1 instructors for allowing us to intrude on their classrooms as observers. This project is supported by Google through a CS Capacity Award.

REFERENCES

1. J.R. Abel and R. Deitz. 2014. Do the Benefits of College Still Outweigh the Costs? *Federal Reserve Bank of New York: Current Issues in Economics and Finance* 20, 3 (2014).
2. Susan Aud and Sidney Wilkinson-Flicker. 2013. *The Condition of Education 2013*. US Department of Education, National Center for Education Statistics. NCES 2013-037.
3. L. Barker, T. Camp, E. Walker, and S. Zweben. 2015. Booming Enrollments - What is the Impact? *Computing Research News* 27, 5 (May 2015).
4. J. Campbell, D. Horton, M. Craig, and P. Gries. 2014. Evaluating an inverted CS1. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. 307–312.
5. Oliver Van DeMille. 2009. *A Thomas Jefferson Education: Teaching a Generation of Leaders for the Twenty-First Century*. TJEdOnline.com.
6. Richard A. DeMillo. 2011. *Abelard to Apple: The Fate of American Colleges and Universities*. MIT Press.
7. Julie Krause, Irene Polycarpou, and Keith Hellman. 2012. Exploring formal learning groups and their impact on recruitment of women in undergraduate CS. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, 179–184.
8. Gregor Novak, Evelyn T. Patterson, Andrew D. Gavrin, and Wolfgang Christian. 1999. *Just-In-Time Teaching: Blending Active Learning with Web Technology*. Prentice Hall, Upper Saddle River, NJ.
9. N. Orhun. 2007. An investigation into the mathematics achievement and attitude towards mathematics with respect to learning style according to gender. *International Journal of Mathematical Education in Science and Technology* 38, 3 (2007), 321–333.
10. Leo Porter, Cynthia Bailey Lee, and Beth Simon. 2013. Halving fail rates using peer instruction: A study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, 177–182.
11. R. Rutherfoord and J. Rutherfoord. 2013. Flipping the classroom: Is it for you? *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education* (2013), 19–22.