TheorieLearn:
Autograded Resources for Theoretical Computer Science

SIIP Implementation and Exploration Proposal

Jeff Erickson, Carl Evans, Yael Gertner, Brad Solomon
Department of Computer Science

Abstract

We propose to develop resources on the PrairieLearn platform to support the teaching of algorithms, data structures, and other theoretical aspects of computer science, at several different levels of the computer science curriculum. The proposed project extends an existing effort to develop scaffolding exercises for CS 374 and expands this effort to include both scaffolding and assessments in CS 173, CS 225, CS 277, CS 401, CS 403, and possibly other related classes at Illinois and elsewhere. We anticipate the development of new elements, new question types, and other software infrastructure that will be useful for a much larger set of PrairieLearn users. We also propose to use our development effort to support and motivate research in theoretical computer science education.

Background

Pedagogy / Philosophy

Algorithms classes are traditionally taught by showing students several classical algorithms—usually designed using a common technique—proving those algorithms correct, analyzing their running times, and then asking students to design and analyze new algorithms using similar techniques. Unfortunately, this approach is inconsistent with the learning goals of these classes, because it does not expose the process of designing new algorithms. Students in these traditional classes see only a polished fait accompli, with no indication of where the algorithm came from; they are expected to develop a skill—algorithm design—which they are never actually taught.

A common complaint of students in many classes is a lack of "worked examples" to study from, especially before exams. In fact, in a typical semester, CS 374 provides complete solutions and grading rubrics for over 100 problems from labs, homeworks, and previous exams, including at least one solved problem on every homework handout. (We also provide several dozen study problems for each exam, but without solutions.) Nevertheless, the students' complaints have merit, because *solutions are only the finished product*.

Almost all coursework in CS 374 consists of open-ended algorithm design and proof questions, most of which require about half a page (on exams) to a page (in homeworks) of semi-structured English to answer. Typical examples include the following:

- Describe a regular expression for the set of all binary strings with an even number of `0`s and an odd number of 1s.
- Prove that the language $\{0^n 1^{2n} \mid n \geq 0\}$ is not regular.
- Describe and analyze an algorithm to determine whether any number appears more than $n/4$ times in a given array of n numbers.

- Describe and analyze an algorithm to find the longest common subsequence of three given strings.
- Describe an algorithm to find the shortest walk in a given graph *G* with colored edges, from vertex *s* to vertex *t*, in which no three consecutive edges have the same color.
- Prove that it is NP-hard to determine the maximum number of scoops of ice cream that can be balanced on a single cone with no "yucky" pairs of flavors touching each other.

Most problems have multiple correct solutions; in some cases, using significantly different techniques. The freeform nature of these questions is a significant strength of the course, but it does come at a cost. Despite promising work using natural language processing to grade simpler narrative questions,[1] automatically grading narrative work is impossible; almost all work in CS 374 must be graded by human beings—in practice, graduate TAs and undergraduate CAs. Ensuring that TAs and CAs provide timely and consistent feedback is one of the most significant challenges of teaching 374, and this challenge has grown as enrollments have increased.

These freeform narrative problems are the heart of the course, and we have no plans to replace them. But students significantly benefit from more structured scaffolding activities that focus on components of the problem-solving process, that provide targeted feedback, and that can help students gain confidence in their own problem-solving abilities. To that end, for the last two years Jeff has been managing a team of undergraduates to develop PrairieLearn resources, first to support CS 374, and more recently to support related computer science classes, especially CS 225. **We propose to continue this long-term effort to develop PrairieLearn resources to support classes across the CS curriculum that teach theoretical computer science topics.** Our current focus is on CS 225, CS 277, CS 374, and CS 401/403, but in the long run we anticipate collaborating with instructors in other classes that teach related material, both at Illinois and elsewhere.

History

Since August 2021 Jeff has managed and funded a team of students to develop PrairieLearn resources that guide CS 374 students through the design/solution process for many different types of problems. The CS 374 PrairieLearn development effort was originally spearheaded by undergraduate Jason Xia in Spring 2021, with the encouragement of instructors Chandra Chekuri and Patrick Lin; Jason continued to play a significant leadership role on the team until his graduation in 2022.[2] PhD student Eliot Robson joined the team in Fall 2021 as a liaison TA from CS 374 and quickly became the project's technical manager. Since Spring 2021, a total of thirteen undergraduate developers have been part of the team for at least one semester.

Independently, Yael Gertner was already developing PrairieLearn resources for the theory courses CS 401 and 403 that are part of the department's new iCAN certificate program. The iCAN program and its component courses are designed with the goal of broadening participation in CS and are aimed at college graduates who wish to enter the computing field but have non-computing backgrounds; the needs and goals of iCAN courses are different from classes taken by our undergraduate majors. Immediate feedback and extra practice with solutions are especially important for these courses.

More recently, thanks to SIIP startup funding, we have expanded our development efforts to CS 225, a sophomore-level data structures class, which is a prerequisite for CS 374, and which recently underwent a long-planned revision. Historically, CS 225 included a significant amount of theoretical content—in particular, induction proofs, running-time recurrences, and algorithm analysis—reinforcing material taught in the

[1] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. Autograding "Explain in Plain English" questions using NLP. *Proc. 52nd SIGCSE*, 1163–1169, 2021.

[2] Unfortunately for us, Jason graduated in May 2022 and is now working at Duolingo.

prerequisite discrete math course CS 173. Over roughly the last decade, as enrollment in 225 grew from 800 students per year to over 1500, all manually grading was replaced with auto-grading, and reinforcement of theoretical content all but disappeared. In Fall 2021, the introductory programming sequence was updated to include a new programming studio course CS 126, which absorbed several weeks of C++ instruction from CS 225, leaving room to reintroduce more theory. Primarily in response to this change (and thanks to support from SIIP), two of the regular instructors for CS 225, Carl Evans and Brad Solomon, joined the development effort in Fall 2022.

Design Goals

Most of our exercises for CS 374 are organized into ”guided problem sets”, each containing a series of exercises related to a single problem or skill. Guided problem sets are not intended to replace written homeworks or exams, but rather to replicate the kind of interactive leading questions that a student might be asked in a discussion/lab section or in office hours.

The design goals for these guided problem sets reflect existing goals for other components of the course, including lectures, labs, and grading rubrics. Most importantly, for each type of problem, guided problem sets should reinforce the solution *process* recommended for that type of problem in other parts of the course. Said differently, we want to provide students with *working* examples, not just more *worked* examples. Guided problem sets should also support multiple correct solutions, recognize and reward progress toward *any* correct solution, explicitly detect common mistakes, award partial credit using the same rubrics as manually graded homeworks and exams, and provide helpful narrative feedback.

Whenever possible, we avoid questions that invite blind exploration, especially multiple choice questions; we want solving the problems to be a learning process, not a process of elimination. We also aim to provide partial credit that rewards progress and targeted feedback that guides students toward correct solutions, instead of merely grading questions as correct or incorrect.

Finally, as a general rule, we also avoid free-form programming questions, in part because it is difficult to automatically grade code on any other basis than correctness on a finite set of test inputs, and we want to recognize and reward *progress toward* correct solutions. Turning well-designed algorithms into practically efficient code is an important skill, but that skill is not the focus of CS 374. We want students to focus instead on the structure, correctness, and efficiency of algorithms without worrying about (more strongly, while staying deliberately agnostic about) low-level implementation details or specific language syntax.

Our design goals for CS 225 are quite different. Here we not only require formative exercises that help students develop mastery and confidence in the theoretical course material, we also need summative assessments. Experience strongly suggests that students will learn material that has no effect on the final course grade, and even students who engage with more theoretical topics out of intrinsic interest benefit from feedback on their efforts. On the other hand, manually grading 1200 freeform induction proofs or data structure design problems is completely infeasible, especially on a short enough schedule for the feedback to be useful, and especially when almost all the TAs with theory expertise are busy in CS 374. We aim to build exercises that reward understanding, not just memorization; that offer suitable partial credit and targeted feedback; that do not succumb to blind exploration; and that are parametrized to inhibit cheating. Building exercises that meet these constraints is a significant challenge, with enormous impact on the rest of the course; in a sense, our success determines what theory *can* be taught in CS 225.

Progress in 2022–23

Thanks to support from the SIIP program as a startup project, we made progress on several different fronts during the 2022–23 academic year, including several new guided problem sets in CS 374 and CS 401/403, a small number of assessments (offered as practice exercises) in CS 225, and several new interactive PrairieLearn elements. We have contributed exercises to other classes at Illinois that are not formal participants in our project; elements, bug fixes, and feature requests to the main PrairieLearn codebase, and significant updates to other open-source projects. In the 2022-23 academic year alone, our resources have been used by almost 2000 Illinois students.

Big-O Input and Multistage Exercises

As a minor but still important contribution, we developed the big-o-input element, which evaluates expressions using asymptotic ("Big O") notation, most commonly in reporting the running times of algorithms. Our element symbolically compares student input to a reference solution using the SymPy Python library, and then provides feedback and partial credit targeted to common errors, such as upper bounds that are too small and therefore incorrect, upper bounds that are correct but loose, and expressions with unnecessary constant factors or lower-order terms. The element properly supports O( ), Θ( ), Ω( ), o( ), and ω( ) expressions, providing necessary feedback and partial credit for each expression type. The question writer only has to describe a reference solution; the element automatically handles all grading and feedback. Our element has been incorporated into the main PrairieLearn codebase, and it is already being used by at least five different CS courses at Illinois.

We are also developing templates to support multistage exercises, and we have already deployed a few prototype examples in CS 225. In these examples, the exercise presents a description of a data structure and asks students to perform a series of (randomly generated) update operations on that data structure. Depending on the question configuration, each operation is revealed only after the student has correctly answered the previous stage ("homework mode"), or after a fixed number of attempts ("exam mode"). Our current prototypes rely heavily on custom Python code, but we plan to provide a lightweight element that would allow authors to create multistage exercises with this simple narrative structure by specifying (or generating) a series of questions and answers.

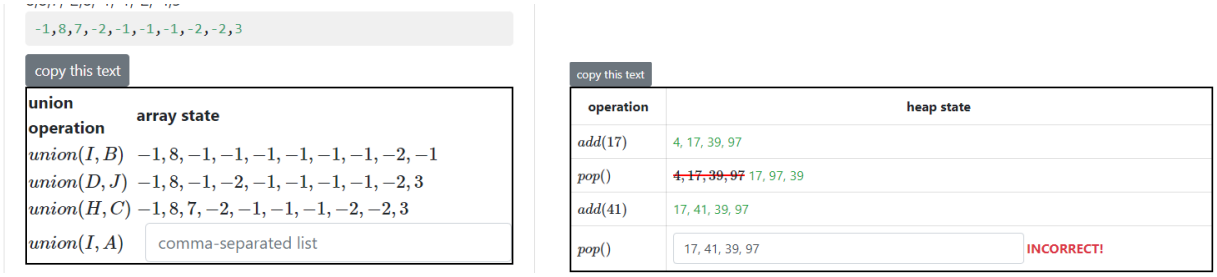

Figure 1. Multistage data structure exercises.

Order/Proof Blocks

One of the most natural PrairieLearn tools for theory classes is the Order Blocks (pl-order-block) element developed by Seth Poulson and others at Illinois.[3,4,5] In previous years, we developed Order Blocks exercises for CS 374 that as students learn to assemble basic induction proofs, that ask students to sort functions by asymptotic ("big-Oh") growth rates, and that ask students to assemble pseudocode descriptions of NP-hardness reductions (which the grading code translates into Python). More recently, we have used Order Blocks to ask students to develop correctness proofs of greedy algorithms, and to describe correct evaluation orders for multidimensional dynamic programming algorithms. According to Seth Poulsen,[6] CS 374 is the first course anywhere to use Proof Blocks for any topic more advanced than introductory proofs.

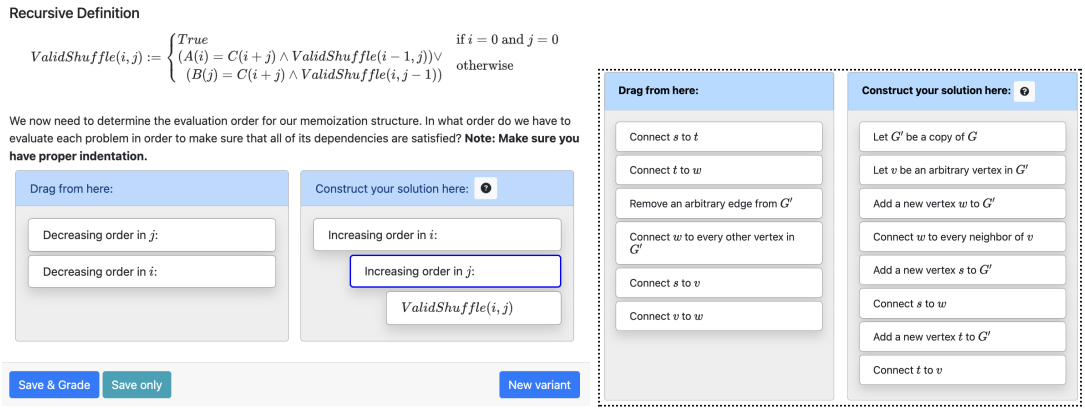

Figure 2. Pseudocode blocks.

Scaffolded Writing (SIGCSE 2023)

Over a year ago we developed a scaffolded writing element that allows students to generate sentences from a hidden context-free grammar, one token at a time. The interface closely resembles the auto-complete feature of several mobile messaging apps; as the student enters their sentence, they are presented with a list of all possible next tokens. To design a problem for this element, the instructor specifies a grammar that can generate both correct and incorrect answers—ideally multiple correct answers and incorrect answers that cover common student mistakes—as well as grading code to provide feedback and partial credit that reflects progress toward a correct solution.

We first deployed this tool to support the design of dynamic programming algorithms. An important step in the design process of these algorithms is clearly specifying the underlying recursive function that the algorithm will evaluate. The underlying grammars generate multiple correct solutions (for example, specifying subproblems by either prefixes or suffixes of the input array), specifications that lead to correct but slower algorithms, several common errors (for example, not describing explicitly how the output value depends on input parameters), and a few stylistic errors (for example, naming the recursive function "DP"). Feedback and partial

---

[3] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. Proof blocks: Autogradable scaffolding activities for learning to write proofs. Preprint, August 2021, arXiv:2106.11032.

[4] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. Evaluating proof blocks as exam questions. *Proc. ICER 2021*, 157–168, 2021. Reprinted in ACM Inroads 13(1): 41–51, 2022.

[5] See https://www.proofblocks.org/.

[6] Personal communication

credit follow the same grading rubric used for dynamic programming problems on written homeworks and exams. A detailed description of this tool was published at SIGCSE 2023.[7]

More recently, we have used the same scaffolded writing tool for graph transformation problems, which arise both in the design of efficient graph algorithms and in NP-hardness proofs. These problems give students practice writing "landmark sentences" that describe the precise relationship between the data given to the transformation algorithm and the graph output by the transformation algorithm. Behind the scenes, we have implemented constraint-based graders that makes it easier for authors to write new constrained writing problems, at least of the same types. We have deployed these exercises in both CS 225 and CS 374.

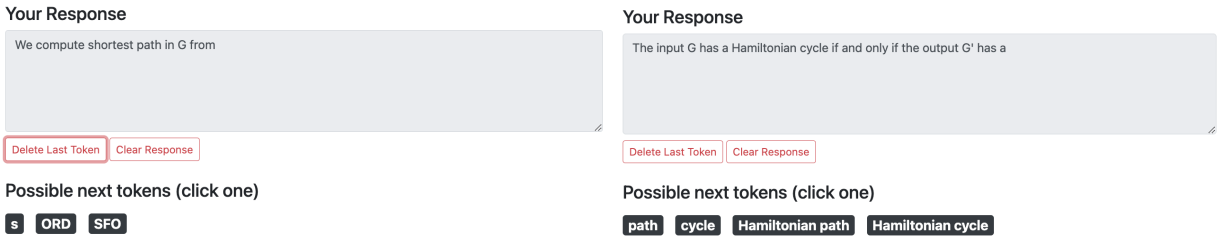

Figure 3. Examples of scaffolded writing, one from CS 225 and one from CS 374.

Automata and Binary Tree Editors

One of the skills we teach early in our algorithms class is designing and drawing deterministic and non-deterministic automata. Online tools for drawing and simulating automata (also known as finite state machines or FSMs), such as JFLAP[8] and Automata Tutor,[9] have existed for many years. While these tools are capable, we could not integrate them into PrairieLearn. Instead, we adapted a lightweight open-source browser-based FSM editor[10] for the front-end interface and an open-source `automata` Python library[11] for the back-end grading code. We have extended the `automata` library to meet our needs, and those extensions have since been incorporated back into the original open-source project.

Our automata editor provides complete freedom to draw, label, and edit states and transitions, including the start state and accepting states. For deterministic automata, students can declare a hidden dump/trash state to simplify their design. When the student submits, the grading code compares the language accepted by the submitted FSM to the target language, and automatically provides counterexamples if the submitted machine is incorrect. Question authors only need to specify the desired type of automaton (deterministic or nondeterministic), a maximum state limit, the input alphabet, and a formal description of one correct automaton. We emphasize that scores and feedback are based on *the language accepted by* the student's submission; every correct FSM is graded as such.

More recently we have modified the automata editor into an editor for binary search trees. Our binary tree editor currently only supports moving nodes, labeling nodes, and adding and deleting leaves. These limited operations are already enough to support exercises about insertions and deletions in binary search trees, which we have already deployed as practice exercises in CS 225. We plan to continue developing this tool to

---

[7] Jason Xia and Craig Zilles. Using context-free grammars to scaffold and automate feedback in precise mathematical writing. *Proc. 54th SIGCSE*, 479–485, 2023.

[8] Susan H. Rodger and Thomas W. Finley, *JFLAP: An Interactive Formal Languages and Automata Package*. Jones & Bartlett, 2006.

[9] Loris D'Antoni, Martin Helfrich, Jan Kretinsky, Emanuel Ramneantu, and Maximilian Weininger. Automata Tutor v3. *Proc. 32nd CAV*, 3–14, 2020. LNCS 12225, Springer.

[10] Evan Wallace. Finite state machine designer. Github repository, 2015. https://github.com/evanw/fsm

[11] Caleb Evans. Automata. Github repository, 2022. https://github.com/caleb531/automata

support highlighting nodes and edges (for example, to highlight search paths) and performing rotations (to support questions about AVL and red-black trees, for example). We also anticipate using this tool for questions not only about binary search trees, but also about binary heaps, Huffman codes, and recursion trees.

Student Survey (ASEE 2023)

We designed our auto-graded scaffolding exercises to support students in learning the course objectives. So that students would be motivated to engage with these exercises throughout the course, we designed them to be easy and enjoyable to use. We also focused on making sure the content of these exercises closely matched the required content of the class, so students felt that these exercises are valuable to improving their competency in the class. All three of these factors— ease of use, enjoyability of exercises, and a clear connection to increased competency—are correlated with improving motivation.

To evaluate the success of our new exercises, we directly surveyed students in the Fall 2022 offering of CS 374 about their experience working with both the new exercises and the traditional written homeworks. We administered the survey at the last week of class, after they had ample engagement with the exercises. Our survey asked the students to express their agreement on a standard 5-point Likert scale, first to 14 statements about the PrairieLearn guided problem sets, and then to the same 14 statements about the traditional written homeworks.

Figure 4 on the following page, which is taken from our upcoming ASEE 2023 paper,[12] summarizes the 260 responses we received. For each statement, responses for PrairieLearn guided problem sets are shown immediately above responses for written homework. As we hoped, the survey results revealed that students found the guided problem sets more enjoyable and less stressful than written homeworks, and gave students more confidence in their own mastery of the course material.

We are surveying CS 374 students again this semester (Spring 2023), and we anticipate repeating the survey in Fall 2023 and Spring 2024, so that we have responses for sections taught by all four regular instructors. We are also collecting grade information and PrairieLearn data for students who have consented to have their information used for research purposes, and we are planning a more detailed analysis of this combined data for future publication.

Goals for 2023–24

Our efforts over the next academic year will focus primarily on CS 225 and CS 374. For CS 225, our main goal is to design and deploy new types of assessment exercises. As mentioned earlier in the proposal, there is a tension between what we would *like* to teach (in particular, to better prepare students for CS 374) and what we can practically assess on PrairieLearn. We are optimistic about potential applications of our prototype binary-tree builder to problems involving about binary search trees, AVL trees and other balanced binary search trees, and Huffman codes—all topics that are already covered from a more practical viewpoint—as well as more purely theoretical topics like recursion trees. In the long run we plan to generalize the element further to handle arbitrary *graphs*, opening up many more possibilities for new assessments. More broadly, we expect close discussion between Jeff (channeling the other tenure-track theory faculty) and Carl and Brad (regular instructors for CS 225) to inform the evolution of both CS 225 and CS 374, beyond their uses of PrairieLearn.

[12] Jeff Erickson, Jason Xia, Eliot Wong Robson, Tue Do, Aidan Glickman, Zhuofan Jia, Eric Jin, Jiwon Lee, Patrick Lin, Steven Pan, Samuel Ruggerio, Tomoko Sakurayama, Andrew Yin, Yael Gertner, and Brad Solomon. Auto-graded scaffolding exercises for theoretical computer science. To appear in *Proc. ASEE 2023.*

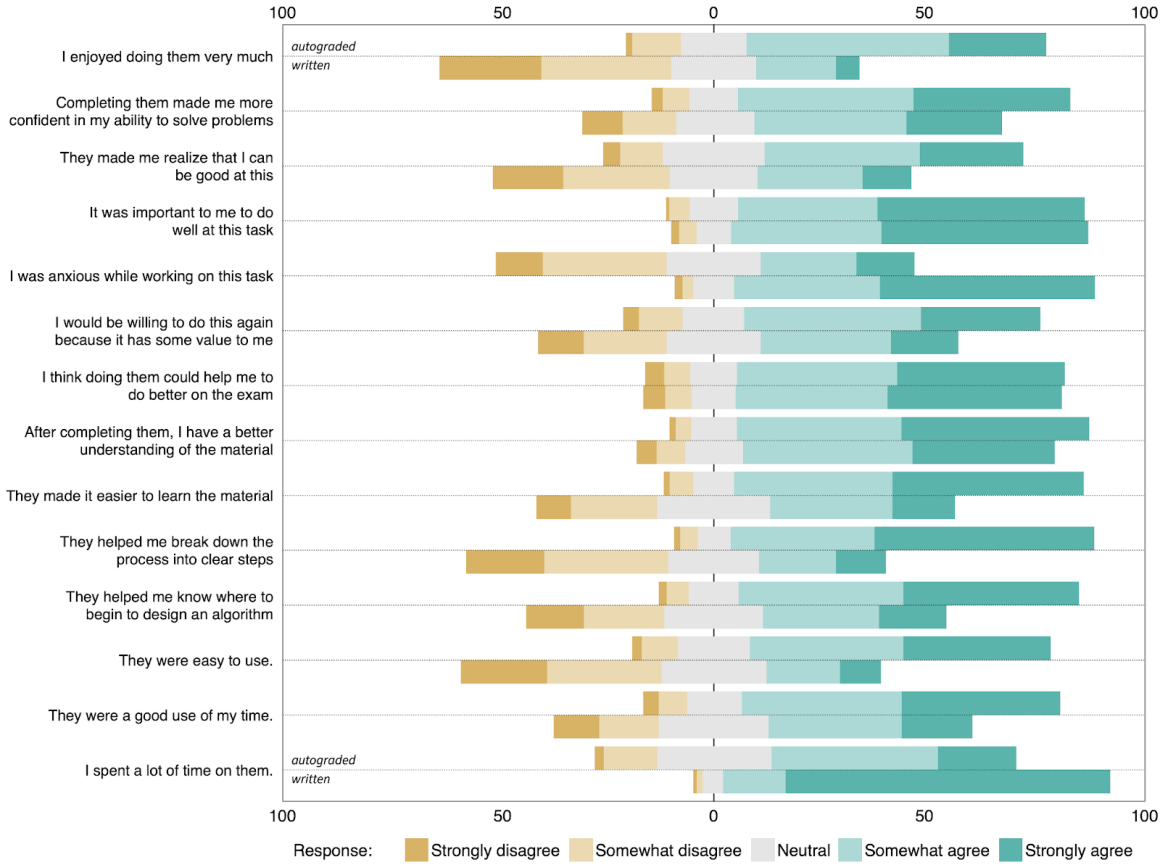

Figure 4. Summary of Fall 2022 responses to our survey

We have already deployed guided problem sets that cover all topics in CS 374, but for most of these topics, we only have a few fully developed exercises. We plan to focus on building more exercises of the types we already have, both to provide students with additional opportunities for practice ("*working* examples") and to give instructors more choice about which exercises to assign for credit. Before each exam in CS 374, the instructors distribute an "exam fodder" document containing dozens of problems for each topic covered on that exam, of similar scope and difficulty to actual exam questions, *without* solutions. (Indeed, many of the fodder problems are taken directly from old exams; however, instructors generally create new problems for each exam.) Over time, we would like to implement most of these hundreds of fodder problems as guided problem sets.

We will continue to discuss future development specifically for CS 401/403 and CS 277. CS 401/403 already uses several guided problem sets taken directly from 374, together with exercises developed independently by Yael Gertner. CS 277 has not adopted any PrairieLearn resources, in part because the course and its intended audience are still very new; the first freshmen were admitted to the X+Data Science majors only this year, for Fall 2023 admission. We do not expect to deploy significant new resources specifically for those classes during the 2023–24 academic year. The team feels that we need more time to understand how to adapt the materials to the needs of students in these classes.

We are planning for more systematic assessments of our resources. A major design goal for our assessments in CS 225 is to better prepare students for CS 374. Instead of waiting for a cohort of students to work through both classes, we are tentatively planning to use some of the 225 assessments as intake quizzes in the Fall 2023

offering of CS 374. At least initially, we will give anyone who completes the intake quizzes full credit—after all, we are assessing the *assessments* more than we are assessing the students—although we will still give the students feedback and collect partial credit scores.

We will continue to maintain close communication both with the staff for all relevant courses. Jeff is teaching CS 374 in Fall 2023, Brad is teaching CS 225 in Fall 2023 and Spring 2024; Yael is continuing to teach CS 401 and 403; and the CS 374 instructor for Spring 2024, Timothy Chan, has already agreed to continue using our resources. We will also continue working closely with the core PrairieLearn team, both to promote broadly useful components up to the main code base, and to advocate (and serve as guinea pigs) for major feature requests (such as parameterized questions and question/element sharing).

Ultimately, we want the resources we develop to be available to as many people as possible, with as little friction as possible. We plan to broaden communication with other classes at Illinois and elsewhere. Although our exercises are tailored to specific audiences and learning, we believe that both the pedagogical architecture of guided problem sets and the software infrastructure that we are building is applicable much more broadly. Our automata builder and `big-o-input` element have already been adopted by CS 173 (discrete math); we also see potential applications of our resources in CS 357 (numerical methods), CS 361 (probability and statistics), and CS 421 (programming languages and compilers), all of which already use PrairieLearn. We have also had a few preliminary conversations with other instructors at Illinois (Ben Cosman for CS 173) and at other universities (Cinda Heeren at UBC, Seth Poulsen soon to join Utah State) which we hope to develop into stronger collaborations.

Finally, we plan to use our development effort as a springboard for more computer science education research. For example, we have already deployed a survey asking students about their interaction with our PrairieLearn resources in CS 374. A majority of students who have responded to the survey reported greater motivation from the PrairieLearn guided problem sets than from traditional written homework. We plan to further explore the data to identify reasons for why certain students do not experience increased motivation and if there are ways to further improve the tool to make it more engaging to *all* students.

As another example, we hypothesize that students who engage with our guided problem sets are better prepared to design similar problems in homeworks and exams. We plan to test this hypothesis with specific exercises, initially by correlating the number of attempts to completion with exam scores for similar problems. We plan to follow this up with interviews where we ask students to "think aloud" as they solve these problems. This will allow us to understand common student mistakes, identify situations where students succeed only after multiple attempts, and help us add feedback that might hint at a path for success. Other researchers have published similar studies about student misconceptions of dynamic programming,[13,14] but so far without *actionable* outcomes.

Finally, Jeff and Yael (along with Ben Cosman, Geoffrey Herman, and Seth Poulsen) are in very early discussions to join a $2 million multi-university NSF proposal, led by Diana Franklin at the University of Chicago, to study theoretical computer science education. The precise scope of that project is still under discussion, but we are optimistic that both projects will be strengthened by our collaboration.

---

[13] Shamama Zehra, Aishwarya Ramanathan, Larry Yueli Zhang, and Daniel Zingaro. Student misconceptions of dynamic programming. *Proc. 49th SIGCSE*, 556–561, 2018.

[14] Michael Shindler, Natalia Pinpin, Mia Markovic, Frederick Reiber, Jee Hoon Kim, Giles Pierre Nunez Carlos, Mine Dogucu, Mark Hong, Michael Luu, Brian Anderson, Aaron Cote, Matthew Ferland, Palak Jain, Tyler LaBonte, Leena Mathur, Ryan Moreno, and Ryan Sakuma. Student misconceptions of dynamic programming: A replication study. *Computer Science Education* 32(3):288–312, 2022.

Organization and Budget

The team will include up to eight undergraduate hourly researchers and two graduate research assistantships. This is larger than the team has been during the startup phase;this expansion is justified by the transition from planning to active development in CS 225, increasing opportunities to interact with other classes and instructors at Illinois and elsewhere, and an increased focus on computer science education research.

Undergraduates will author new exercises, prototype new types of exercises, develop new interactive elements, and participate in computer science education research. By default, new undergraduate team members will focus primarily on writing exercises that follow existing structures, under the guidance of more experienced developers, but we expect their responsibilities to broaden as they become more familiar with the existing codebase and the project's pedagogical goals. Several past and current team members have expressed interest in continuing on the project. To attract new team members, we will advertise and collect applications for undergraduate researchers through the Computer Science Department's undergraduate-hiring portal. Past experience suggests that we will attract a large pool of qualified applicants.

Graduate research assistants will be technical managers for the software development effort and actively participate in CS education research. CS theory PhD student Eliot Robson (advised by Sariel Har-Peled) has already been working as a technical manager for the project for over a year, strictly on a volunteer basis; his leadership and contributions have been crucial for the success of the project so far. We anticipate Eliot's continued leadership in this role. CS education PhD student Hongxuan Chen (advised by Geoffrey Herman) has also expressed interest in joining the project.

In addition to these paid positions, we may include additional students who are interested in independent study or senior-thesis credit.

As we have for the last two semesters, the team will meet two or three times each week during the fall and spring semesters: once with all faculty participants (primarily to report technical progress and discuss strategic and administrative issues), and at least once with student developers (primarily to discuss technical issues). Visitors are welcome to attend any of these meetings. In addition to weekly meetings, the team will communicate asynchronously through Slack and through Github issue tracking and pull requests. In particular, student developers review each other's code before changes to the code base are accepted.

The only item in our budget other than student wages is support for student travel to two domestic conferences (for example, two students to SIGCSE, or one student to SIGCSE and another to ASEE).

Our budget is outlined in the table on the follopwing page. **The Department of Computer Science has agreed to match funding from the SIIP program.** Accordingly, all budget items are evenly split between the two funding sources.

About the Target Classes

CS 225 ("Data Structures") is a sophomore-level course covering elementary data structures and algorithms and their implementations. This course is required for all computer science and computer engineering majors, computer science minors, and transfer applications into computer science and computer engineering. The course has a steady-state enrollment of about 1200 students every fall semester and 600 students every spring.

Salaries / Wages	Proposed Budget	Dept. Match	Comments
Graduate Assistants	$42,715	$42,715	2 RAs, each 11 months at 50%, at estimated CS 2023-24 post-prelim RA rate, no overhead or tuition
Undergrad Hourlies	$28,880	$28,880	8 undergrads × $19/hour (2022–23 CS rate for senior URAs) × 10 hours/week × 38 weeks (8/23/23–5/12/24)
Travel	$2,000	$2,000	one student to SIGCSE or ASEE
Totals	**$73,595**	**$73,595**	
Total Project Budget	**$147,190**		

Table 1. Proposed budget

CS 277 ("Algorithms and Data Structures for Data Science") is an introduction to elementary concepts in algorithms and classical data structures, with a focus on their data science applications. This is a new course designed for the new "Data Science + X" degree programs; so far there have been only two pilot offerings, with 30 students in the second offering in Spring 2023. The first Data Science + X students will matriculate in Fall 2023. Based on existing enrollments in the prerequisite class Stat 207, we expect steady-state enrollment in CS 277 to grow to about 100 students per offering by 2024.

CS 374 ("Introduction to Algorithms and Models of Computation") is a junior-level theoretical computer science course, which covers a combination of algorithm design and analysis, automata and formal language theory, and complexity theory. Coursework consists almost entirely of open-ended design and analysis problems. CS 374 is required for all computer science (including CS+X) and computer engineering majors. The course is split into two independent sections, taught by CS and ECE instructors, with steady-state enrollments of 450 and 200 students per semester, respectively.

CS 401 ("Accelerated Fundamentals of Algorithms I") and **CS 403** ("Accelerated Fundamentals of Algorithms II") are part of our new ICAN (Illinois Computing Accelerator for Non-Specialists) graduate certificate program, which is aimed at students with bachelor's degrees in fields other than computer science. So far each of these classes has been taught three times, most recently to a cohort of about 30 students; steady state enrollment is expected to grow to about 40 students per year in each class.

About the Project Team

All five team members are faculty members in the Department of Computer Science.

Jeff Erickson is a full (tenure-track) professor. He is one of the architects and a regular instructor of both CS 374 and the followup algorithms course CS 473. Jeff served as an AE3 Engineering Innovation Fellow from 2017 to 2021; this is his first SIIP team. His research interests are slowly morphing from algorithms to computer science education.

Carl Evans is a teaching assistant professor and one of the regular instructors of CS 225. Carl has also taught CS 126 (software design studio), CS 173 (discrete structures), and CS 341 (system programming; he also served

as a TA for several CS courses as a PhD student at Illinois. Carl has also participated in coordinating the development of CS 128, which was supported by a SIIP grant.

Yael Gertner is a teaching assistant professor. She developed and regularly teaches several courses in the iCAN program, including CS 401 and 403, and she has been actively developing PrairieLearn resources to support these classes. She has also taught CS 173 (discrete structures). Her research interests are in computer science education in the areas of broadening participation in computing and designing interventions to increase students' learning outcomes. She is also part of the ongoing SIIP project "Identifying Student Profiles to Facilitate Learning Outcomes in Introductory Problem-Solving Classes".

Brad Solomon is a teaching assistant professor and one of the regular instructors of CS 225. Brad is the course director and regular instructor for our new course CS 277 (algorithms and data structures for data science); he has also taught CS 173 (discrete structures). His research focuses on developing new algorithms and data structures for the efficient storage, search, and analysis of genomic sequencing data. Brad has not been a member of any other SIIP team.