# Expanding the Horizons of Autograding: Innovative Questions at UBC

Jeffrey Niu*
yinian@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Jessica Wong†
jhmwong@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Charlie Lake
clake13@alum.ubc.ca
University of British Columbia
Vancouver, Canada

Justin Rahardjo
jdr213@alum.ubc.ca
University of British Columbia
Vancouver, Canada

Hedayat Zarkoob
hzarkoob@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Oluwakemi Ola
kemiola@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Patrice Belleville
patrice@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Karina Mochetti
mochetti@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Meghan Allen
meghana@cs.ubc.ca
University of British Columbia
Vancouver, Canada

Firas Moosvi
firas.moosvi@ubc.ca
University of British Columbia
Vancouver, Canada

Steven Wolfman
wolf@cs.ubc.ca
University of British Columbia
Vancouver, Canada

## Abstract

The popularity of autograding has grown due to increasing class sizes and the need to reduce grading load while ensuring quality. Autograding has conventionally been used for multiple choice and fill in the blank questions, or to check code correctness. In this work, we discuss the use of autograders at UBC and some non-conventional autograding implementations in our curricula. We reflect upon our autograder use in our courses and discuss the benefits, implications, and considerations of this pedagogical choice.

## CCS Concepts

• **Social and professional topics → Student assessment**.

## Keywords

Automated Assessment, Computer Science Education

**ACM Reference Format:**
Jeffrey Niu, Jessica Wong, Charlie Lake, Justin Rahardjo, Hedayat Zarkoob, Oluwakemi Ola, Patrice Belleville, Karina Mochetti, Meghan Allen, Firas Moosvi, and Steven Wolfman. 2025. Expanding the Horizons of Autograding: Innovative Questions at UBC. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701892

*co-first author
†co-first author

*SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0531-1/25/02
https://doi.org/10.1145/3641554.3701892

## 1 Introduction

Autograders (AGs) have been used for decades [23] and their use has recently proliferated to reduce grading load in large classrooms [3]. AGs help improve consistency in grading [15], improve student learning through providing immediate feedback and enabling iteration over problems [15, 16], increase student autonomy through providing feedback flexibly [15], and encourage students to start assignments early through automated feedback schedules [9].

AGs have conventionally graded multiple choice questions [15], fill in the blank questions [13], or code correctness [23]. While autograding the functional correctness of coding questions has been well-documented, many additional factors, such as style and syntax, play into whether a student's code is optimal in the context of a course. These factors and other aspects of a student's learning can be assessed through non-conventional AG questions.

This experience report discusses effective, innovative ways we have already incorporated autograding into courses we teach. These questions extend beyond conventional autograding, i.e., multiple choice and fill in the blank questions, and questions that examine code functionality. Section 2 briefly discusses other published work about non-conventional uses of autograding. In Section 3, we describe eight examples of non-conventional autograding use at UBC. We then share our reflections on the benefits and drawbacks of integrating these innovative autograding approaches into our curricula. Finally, Section 5 briefly presents our views on the future of autograding before we conclude in Section 6.

## 2 Related Work

### 2.1 Autograding Reasoning

Computer science reasoning skills appear across many courses and are valuable to assess students' analysis of complex mathematical problems that are not well-suited to conventional autograding approaches. One way of evaluating these reasoning skills is through Parsons problems [24]. Students are provided with a list of code snippets, possibly including distractors, and are asked to order the relevant code fragments to obtain a program that solves a given problem. Variants of Parsons problems have implementations that are immediately autograded to provide feedback [7, 30].

AG proof blocks (Parsons proof problems) scaffold students to make the jump to full proofs [25]. Similarly, the "Incredible Proof Machine" is an interactive theorem prover that allows students to logically connect assumptions, theorems, and conclusions across varying types of proofs which helps students visualize connections between different parts of the proof [6]. Zhao et al. [33] explored autograding of complete induction proofs using NLP methods. While students benefited from this AG's immediate feedback, they expressed a lack of trust towards it [33], and it cannot autograde all proof questions relevant to computer science courses.

## 2.2 Autograding Code

Code can be assessed via various metrics. Dynamic analysis involves running the code and checking the outputs and efficiency [2, 23]. Static analysis examines non-functional elements of code quality such as software complexity and code structure, style, and syntax [23]. For example, code segments can be autograded by converting them into a standardized pseudocode format whose structure and logic is evaluated relative to an answer key [20]. One potential benefit (or drawback) of this process is it does not evaluate syntax. Providing feedback on style and structure is important; Singh et al. [27] investigated automated feedback generation by comparing code structure with a reference solution. They generated and shared a minimal set of potential corrections as feedback.

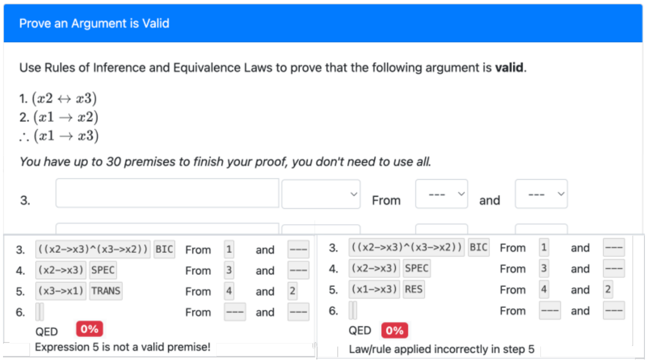
## 2.3 Autograding Diagrams

In the context of object oriented programming, UML diagrams are commonly used to show relationships between types or objects. Fill in the blank UML diagrams have been explored with moderate success to ease grading [18]. More complex systems such as SD4ED allow students to develop UML sequence diagrams with an interactive tool [1]. Algorithmic work has also been done to autograde UML diagrams, allowing feedback via a similarity score between the students' UML diagram and the answer key [17].

General graph problems have been used in conjunction with autograding. For instance, the "Online Judge" system automates the generation and grading of graph questions [31]. Graph autogenerators have also been explored on other platforms such as PrairieLearn [30]. Tools such as these are promising due to the potential for question randomization providing students with virtually unlimited variants. One caveat of using diagram questions in general (not just autograded diagrams) is the accessibility challenges it may pose to some students. Care must be taken to ensure that diagram questions are compatible with text-to-speech and screen-reader systems [28].

## 2.4 AI and AI-Assisted Autograding

In recent years, commercial educational technology companies have successfully incorporated AI-assisted grading into their various products [26]. For instance, using computer vision to group written work together [14] and machine learning to develop adaptive assessments based on student performance on specific questions [21].



Prove an Argument is Valid

Use Rules of Inference and Equivalence Laws to prove that the following argument is **valid**.

1. $(x2 \leftrightarrow x3)$
2. $(x1 \rightarrow x2)$
$\therefore (x1 \rightarrow x3)$

*You have up to 30 premises to finish your proof, you don't need to use all.*

3. From --- and ---

```
3. ((x2->x3)^(x3->x2)) BIC    From 1   and ---
4. (x2->x3) SPEC              From 3   and ---
5. (x3->x1) TRANS             From 4   and 2
6.                            From --- and ---
   QED 0%
   Expression 5 is not a valid premise!
```

```
3. ((x2->x3)^(x3->x2)) BIC    From 1   and ---
4. (x2->x3) SPEC              From 3   and ---
5. (x1->x3) RES               From 4   and 2
6.                            From --- and ---
   QED 0%
   Law/rule applied incorrectly in step 5
```

**Figure 1: Question and answers about argument validity.**

There are also clear paths for large language models (LLMs) as autograding assessment tools. Automatic short textual answers grading (ASAG) is the lowest hanging fruit [12] but more sophisticated approaches include automated parsing of code specifications to generate test cases and evaluating student responses to "explain in plain English" questions [12]. In some cases, LLMs can reduce manual grading by separating clearly correct or incorrect responses from those the model has low confidence in evaluating correctly.

Other work on AI-assisted autograding has explored fully automated essay scoring systems using methods such as neural networks, ontology based techniques, and regression and classification models [4]. AI systems can be used in more sophisticated ways as well; for instance Bayesian inference has previously been used to facilitate fair peer-grading at scale [8, 32].

# 3 Autograding Examples

## 3.1 Autograding Reasoning

*3.1.1 AG-Proofs.* In our first-year course on Models of Computation, students are given an argument and must reach a given conclusion by using a specified set of logical equivalence laws and rules of inference. At each step, they can use only one law or rule and they must clearly state which premise(s) it is applied to. Our AG provides students with immediate feedback as they practice these types of proofs. Each new fact derived in the proof goes into a box, as do the law or rule and the premise(s) used to deduce it (Figure 1). The AG can provide feedback on the location of syntax issues (e.g., missing brackets) and incorrect deductions, and also identify instances where a law/rule was not used with a given premise correctly. Figure 1 shows possible student submissions and feedback. Instead of giving somewhat arbitrary partial marks as we did when grading on paper, we allow multiple attempts. This approach not only evaluates the students' ability to reach the conclusion but also encourages mastery-learning by leading them to analyze their answers critically to correct any mistakes.

*3.1.2 AG-Parsons.* The initial application of Parsons problems and most of the subsequent research has been focused on programming, but they can be used for any problem where a number of statements need to be reordered correctly to reach a solution. The following question is used in our upper-level operating systems course:

"Copy-on-write allows processes to share data for reading, up until the point that one process writes the data. We implement this by mapping the same physical page into the address space of two or more processes. Imagine processes $X$ and $Y$ both have physical page $P$ mapped into their address space. If $X$ tries to write to page $P$, we create a copy of the page, replace the mapping in $X$ with a mapping to the new page, and then let $X$ write to the new page. In the area below, order the steps that the hardware and software must take to implement copy-on-write."

The problem then states several assumptions and provides the following list of items to order:

(1) The OS sets RW bit in $Q$'s PTE for VA to 1.
(2) A fault occurs due to a page missing from physical mem.
(3) The OS sets the RW bit in $R$'s PTE for VA to 0.
(4) Hardware takes a fault due to permissions.
(5) The OS copies the contents of $P$ to $P2$.
(6) The OS copies the contents of $P2$ to $P$.
(7) The OS sets the phys. page number in $Q$'s PTE for VA to $P2$.
(8) The OS writes $P2$ to disk.
(9) The OS finds $P2$, an unused page of memory.
(10) Process $Q$ issued a store to address VA.

Items 2, 3, 6, 8 and 10 are distractors and do not appear in any correct solution (for instance, 4, 9, 5, 7, 1). The question includes a list of 29 possible distractors, five of which are chosen at random when the problem is presented to each student. This question tests student's reasoning about how to implement *copy-on-write*.

## 3.2 Autograding Code

*3.2.1 AG-Structure.* In an introductory programming course for non-majors, we teach a systematic program design process [11]; students design data types that model real-world information. We designed an AG to assess whether students follow the process taught in class.

Each data type includes a template function outlining the structure of a one-argument function that operates on the data type. When students design a function, we expect they will modify the input data type's template function to inform the structure of their function. Students are taught that the template function's structure will provide consistency across functions, thereby improving their program's readability and maintainability.

To emphasize the importance of the design process, and because the template function is not executable, the AG uses regular expressions to check whether students have deviated from the template function's structure. Solutions that are functionally correct will not receive many marks unless they follow the design process.

*3.2.2 AG-Efficiency.* Another programming question used in a computer hardware and operating systems course involves both code correctness and, unlike most common uses of autograding, code efficiency. The students are given a small C program that computes a convolution of a large image in order to apply a filter to it. This C program is correct, but intentionally inefficient. Students are then asked to use their knowledge of how caches work to improve the code. The faster solution must obviously still be correct, so submissions that produce incorrect output do not receive any marks. The grade given to a correct submission depends entirely on how fast it runs compared to the original program. In order to get full marks, students need to develop an implementation that takes only 4% of the time of the original/baseline implementation.

## 3.3 Autograding Diagrams

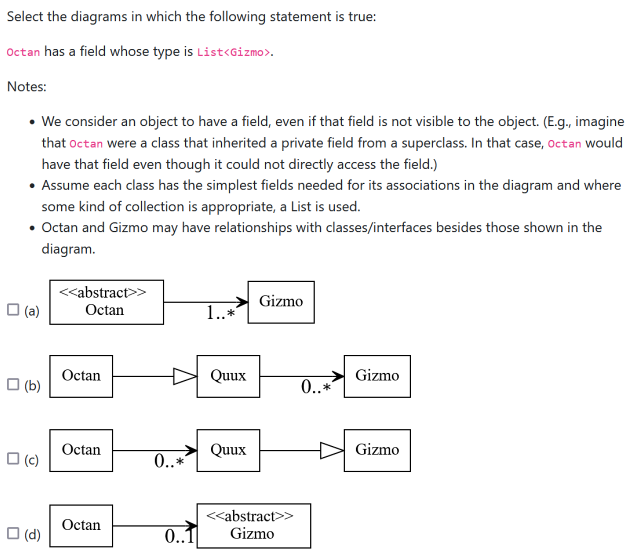
*3.3.1 AG-UML.* In a second-year course on software construction, students learn object-oriented programming. UML class diagrams play an important role in the learning objectives for this course, but they are laborious to create, even for paper-based exams. We have designed a UML problem generator that helps us create new problems more efficiently. It allows for randomization that can deter cheating and encourages mastery-learning when students revisit randomized variants of the same question repeatedly.

In the UML problem generator, nodes and edges can be specified. A node specification is *[not] (concrete | abstractClass | interface) <Name>* where *Name* is a name beginning with an uppercase letter. The *not* is optional. For example:

- concrete SomeConcreteClass
- not interface SomeClassMayBeAbstract
- not concrete DefinitelyAbstract

An edge specification is *[mutual] <Name1> (<type>) <Name2>* where *Name1* and *Name2* are legal node names and the edge goes from *Name1* to *Name2*. If the edge is *mutual*, then it is bidirectional. The *type* is an edge type. The specific edge types are: (1) *implements*, *extends*; (2) *depends*; and (3) *aggregates*, *associates*.

Figure 2 shows an exam question that uses the UML generation tool. The exam question shown also uses randomization; we specify a relationship between two or more types and then present a list of randomly generated UML diagrams. Students must select all of the UML diagrams that are consistent with the described types and their relationship(s).



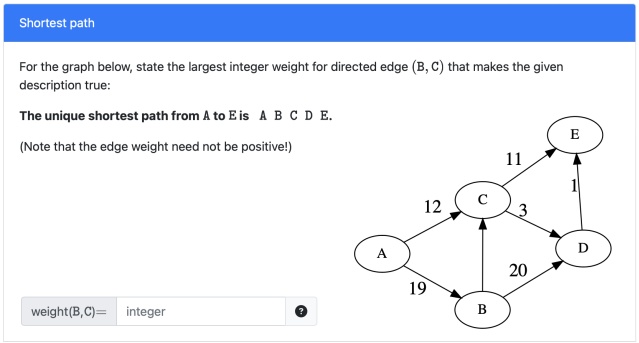
Select the diagrams in which the following statement is true:

`Octan` has a field whose type is `List<Gizmo>`.

Notes:

- We consider an object to have a field, even if that field is not visible to the object. (E.g., imagine that `Octan` were a class that inherited a private field from a superclass. In that case, `Octan` would have that field even though it could not directly access the field.)
- Assume each class has the simplest fields needed for its associations in the diagram and where some kind of collection is appropriate, a List is used.
- Octan and Gizmo may have relationships with classes/interfaces besides those shown in the diagram.

**Figure 2: Sample UML question.**



Shortest path

For the graph below, state the largest integer weight for directed edge (B, C) that makes the given description true:

**The unique shortest path from A to E is A B C D E.**

(Note that the edge weight need not be positive!)

weight(B,C)= integer

**Figure 3: A variant of a graph question where students need to determine an edge weight to satisfy a condition.**

*3.3.2 AG-Graphs.* In a second-year programming and algorithms course intended for non-computer science majors, students are taught how to parse graphs and apply algorithms. These classes of problems are essential to develop students' algorithmic thinking skills. One important example is applying Dijkstra's algorithm to find the shortest path between two nodes in a directed graph. Anecdotally, students do not perceive this to be a particularly difficult concept, but over several terms students score lower on these questions compared to other questions in this unit (higher discrimination index). To give students plenty of practice opportunities, we used GraphViz notation [10] to develop virtually unlimited directed graph diagrams for students to practice on, as shown in Figure 3. Variants of this problem include giving students incomplete diagrams and asking them to draw edges to create graphs that are consistent with the provided specification. Creating these problems requires significant upfront effort, but with appropriately set constraints, near limitless randomization is possible by varying edge weights and the conditions that need to be satisfied.

## 3.4 AI and AI-assisted Autograding

*3.4.1 AG-Adaptive.* In our CS1 course, where students use guided data and function definitions, we use an adaptive AG. The course follows the "How to Design Programs" [11] approach, which specifies a five-step recipe for defining functions. The platform is a key player in this approach, using multiple-choice questions at each recipe step to support students' learning. The scenarios are semi-adaptive, with the student's choice determining the next question, hint, or support they are given. If a student answers a question incorrectly, instead of a fixed score of 0, the platform steps in, providing supportive hints to illuminate the errors and then redirect the student to a similar question that helps to re-assess the learning goal before proceeding to the next step in the design process. The structured approach is beneficial in isolating each step of the design process and providing multiple support structures and questions to ensure that misconceptions are corrected. The structure of a scenario can be visualized as a network of nodes connected by edges, similar to a graph. This structure allows for numerous connections, leading to different pathways through the problem. Figure 4 illustrates a simple scenario with one main question. The main question is the starting point, leading to three possible options. Depending on the option chosen, the next step could be a clarifying question, immediate support before retrying the question, or the end of the scenario. A typical scenario would have five main questions (i.e., one per recipe step), each question would have four options, and each option would have zero to two clarifying questions. The platform resets the problem if it determines that students are trying to subvert the learning process. The scenario-based platform breaks down programming problems into discrete steps; the semi-adaptive nature also provides support for all students as needed.

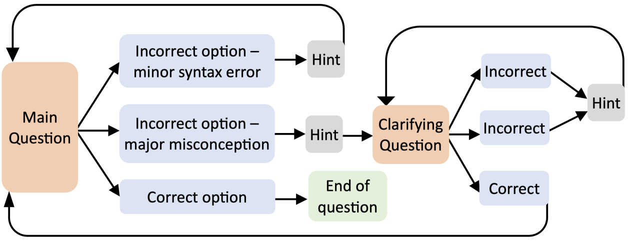


**Figure 4: The structure of a one-question scenario.**

*3.4.2 AG-Peer-Grading.* In a fourth-year course about the relationship between information technology and society, students complete an assigned reading and write a mini-essay in response each week. They are then asked to grade their peers' written work with each essay receiving 2-5 grades and each student submitting 2-5 grades. Peer grading provides faster feedback and helps students learn the course material better by thinking critically about each other's work. However, without a proper way to evaluate the quality of the peer evaluations, many students are not motivated to put in the effort required to give high-quality feedback. Historically, students' grading performance in this course was partially assessed by TAs, but this process was time-consuming and we often lacked sufficient resources to grade everyone. We therefore switched to an AI-based algorithm that autogrades students' peer grading performance. This algorithm takes in peer grading data and uses Bayesian inference to aggregate the student-assigned grades, resulting in a score that indicates how accurately each student graded their peers. This approach helps aggregate students' noisy grades to estimate grading performance and essay grades, detect undesired grading behavior, and direct TA efforts where they are most needed.

# 4 Discussion

## 4.1 Implications

*4.1.1 Equity in Grading.* AGs can improve equity in assessment by improving grading consistency and by enabling alternative grading practices. Consistent manual grading can be challenging, especially when different graders are assigned to the same question. For proofs or coding questions, evaluating how close an incomplete answer is to the correct solution is subjective. Transitioning these types of questions from manual to autograding can result in more consistent grading, ensuring that identical answers receive identical grades. Also, the same feedback is given to all students with a given answer, offering them the same learning opportunity. This consistency builds students' trust in their final grades and reduces the time spent on regrade requests caused by human error.

Autograding allows students to make several attempts and critically analyze their incorrect answers based on feedback, giving them a chance to understand and correct mistakes. For instance, AG-Proofs and AG-Structure showcase questions used in all-or-nothing grading systems where partial marks are replaced by multiple attempts. In AG-Structure, students receive feedback beyond what a typical compiler or interpreter might provide. These features align well with alternative grading systems such as Mastery Learning [5] and Specification Grading [22].

*4.1.2 Problem Authenticity.* In order for students to engage with questions genuinely and not treat them solely as hoops that they need to jump through for marks, students must be convinced the questions are (at least somewhat) *authentic*. This means the questions are not artificially constrained by the technology used (here, the AG), and that they might even be problems students would need to solve in situations beyond the context of the course.

Rule of inference proofs (AG-Proofs), while not a question type that would arise outside of this course, are authentic for two reasons: (1) because this is precisely the type of question we would have asked on a paper evaluation, and (2) because the answer is either correct or not, just like a proof in a research paper. The questions about UML diagrams (AG-UML), directed graphs (AG-Graphs), and virtual memory (AG-Parsons) are additional examples of question types that frequently appeared on paper evaluations. Adaptive problem presentation (AG-Adaptive) mimics the way an instructor or teaching assistant would guide a student through a problem, while the Bayesian inference AG in AG-Peer-Grading also efficiently executes the same tasks that would have taken teaching assistants a great deal of time in the past.

*4.1.3 Developing Metacognitive Skills.* Students often race through course materials without taking time to slow down and pay attention to the nuances of the content. From a constructivist perspective, activities that enable students to focus on how they are developing their solutions are helpful for learning [19]. One of the benefits of our AGs is an increased focus on improving students' metacognition by designing autograded questions that allow students to slow down and pay attention to how they construct knowledge. In AG-Proofs, the autograded questions allow students to practice creating proofs where they must justify each step explicitly. Externalizing thought processes in early theory courses facilitates the active construction that enables students to isolate and identify the reasoning involved at each step. In AG-Adaptive, the staged programming builds metacognitive skills as students can think about how they approach the question and confront their misconceptions in a low-stakes environment. Furthermore, the scaffolding provided by a staged-adaptive process tailors the learning activity to help students construct their understanding of the material and achieve their learning outcomes. A key aspect of metacognition is externalizing and examining mental models. In AG-UML, the focus on UML diagram-type questions focuses students on the relationships and interactions between components of their course project. While their projects are small in this CS2 course, AG-UML allows them to see benefits of non-textual models of their software systems.

*4.1.4 Emphasizing Process over Correctness.* Encouraging students to focus on the problem solving process enables them to develop transferable skills to identify and solve problems beyond those they have already encountered. In order to encourage students to focus on the process, we have structured questions to steer students towards thinking about the process they took to reach the answer.

In AG-Structure, we reward students who follow the systematic design process, while penalizing students who write functions in an ad-hoc manner. In AG-Adaptive, each step of the recipe is treated as a unique sub-problem. This approach reinforces systematic programming principles and emphasizes the importance of attention to detail in the early design stages to avoid mistakes.

The randomization in AG-UML and AG-Graphs also drives students to consider their process as students can attempt different variations. These variations discourage students from guessing and instead encourage them to construct a generalizable process for understanding UML diagrams and graph algorithms.

## 4.2 Reflections

*4.2.1 Students learning the AG.* Students enter our program with limited experience with autograded assessments. We observed that students had to grapple with learning to use the AG, on top of course concepts. This struggle was particularly pronounced for introductory courses as students often lacked the domain background to understand whether an error was due to their answer or to an issue with the AG itself. There is a gap between student expectations of AG behavior and what the AG actually does, which can be frustrating for the students. For example, they can be unhappy that their code must compile for the AG to generate a grade. To someone with domain background, this restriction seems like an obvious expectation, but it is not to introductory students.

There can also be a perception that the AG makes things harder as it impedes the students' ability to submit their work. This is particularly true for courses that are transitioning from paper-based assessments to autograding. In another instance, when attempting AG-Proofs, students were frustrated when they formatted their answers incorrectly across multiple attempts. With manual grading, these frustrations would not arise as improper formatting would only be detected after submission. Providing immediate feedback that pinpoints the line causing the issue helped students learn how the AG works and reduced their frustration. Students also found the all-or-nothing grading scheme with multiple attempts drastically different from a conventional approach with one attempt and partial marks; this led to many requests for grade reconsideration.

Another source of confusion that arose in AG-Structure is that many students expect the AG to work like a conventional AG where there is exactly one correct answer. They asked many questions on how to modify their solution to match *the* correct solution, which does not exist. To address these questions, we redirected students towards the systematic design process to obtain *a* correct solution. Finally, students in AG-Peer-Grading needed to understand that the AG provides an unbiased estimate based on their peers, unlike conventional grading where one grader provides a definitive score.

*4.2.2 Instructors learning the AG.* Most of the AGs discussed in this paper were written, and use questions developed, by one or two people. Their design tried to ensure that future instructors can create new questions based on already existing ones. For instance, the code efficiency checker from AG-Efficiency can be easily integrated

| Example | Pedagogical Value | Benefits | | | |
|---|---|---|---|---|---|
| | | Equity | Authenticity | Metacognition | Process first |
| AG-Proofs | authentic mathematical and logical thinking | X | X | X | |
| AG-Parsons | low-barrier way to assess virtual memory comprehension | | X | | X |
| AG-Structure | assess design and student abilities to follow templates | X | X | | X |
| AG-Efficiency | assess knowledge of caching, run-time efficiency | | X | | |
| AG-UML | authentic diagrams, mastery-oriented practice | | X | X | X |
| AG-Graphs | authentic diagrams, mastery-oriented practice | | X | | X |
| AG-Adaptive | individualized support | | X | X | X |
| AG-Peer-Grading | reduces the need for TAs to monitor peer grading | X | X | | |

**Table 1: Summary of pedagogical value and benefits for each autograder example.**

into another question or course that uses our autograding platform. Similarly, AG-UML and AG-Graphs can be easily modified for different UML or graph questions. In AG-Proofs and AG-Structure, we implemented custom frameworks to support our AG uses. They expose a simple API that future instructors can leverage to create new questions. For AG-Structure we also provided extensive documentation and unit tests to allow future instructors to extend the framework and add further functionality. Other AGs require instructors to understand how to best use them, such as AG-Peer-Grading, which requires extensive hyperparameter tuning.

For instructors, learning how to set up questions to target specific concepts can be difficult. For example, in the adaptive problems in AG-Adaptive, question design requires careful considerations to target the concept requiring reinforcement. Each question also needs to be carefully placed within the graph of questions. Even labelling questions so other instructors can find similar ones can be a difficult task as devising a classification system is non-trivial.

*4.2.3 Recognizing AG pitfalls.* One of the difficulties when designing an AG is ensuring that it accurately reflects the course goals. Both AG-Proofs and AG-Structure can be too strict when evaluating an answer. The custom library used for verifying proof steps in AG-Proofs requires specific formatting, and disallows expressions with more than two operators within a bracket such as $(a \wedge b \wedge c)$. In AG-Structure, creating regular expressions that do not result in false positives or false negatives was difficult. We designed feedback around known limitations and continue working to fix issues and align the AG with the learning objectives.

Randomization within the AG can also create problems. In AG-UML, randomization of surface-level features such as fake names can mislead students, distracting them from the target concept. The AG design needs to take this into account to avoid confusing students and focus on the concepts at hand. As a simple example, the different distractor options in AG-Parsons are only targeted towards virtual memory misconceptions, not extraneous features.

Finally, we can unintentionally commit future instructors to grading decisions because there is a risk that some of these decisions get built into the AG and are non-trivial to change.

## 5 Future of Autograding

We foresee that the scope of autograding will expand in the future. Integrating new computational tools, including LLMs, can enhance autograding frameworks to grade subjective tasks [12]. These tools can further the authenticity of all our questions by prompting students for explanations in addition to their answers. This strategy can be especially useful in AG-UML or AG-Graphs, where the answer format is still multiple choice or fill in the blank.

Computer vision and NLP tools can expand the scope of what is graded. Tools that can parse and understand scratch work can award partial marks based on the types of errors made. Unlike AG-Proofs, where only fully-correct solutions earn grades, an intelligent grader can assign grades based on an estimated level of understanding.

Future AGs can become increasingly adaptive to students' learning. Our approach in AG-Adaptive requires manual setup of the question network. By incorporating intelligent agents, we can automate question suggestion, parse the gaps in the student's knowledge, and offer useful next steps. LLMs can assist in generating questions and solutions that target student knowledge gaps. However, we must exercise caution when integrating AI tools, especially surrounding privacy and bias. The AI tools we use must be vetted to ensure student data is protected and that their outputs are accurate and impartial. Otherwise, our AGs may leave students with further confusion, or worse, propagate AI tools' biases [29] onto students.

## 6 Conclusion

In this paper, we described how several courses at UBC have adopted various AGs beyond conventional autograding. We took a tool-agnostic approach, providing a variety of questions across courses to highlight novel ways in which we can assess students. Though not a comprehensive look at the full space of autograding, we hope it provides useful perspectives, implications, and reflections on our adoption of these AGs. Approaches to autograding amongst instructors vary: some use the most advanced tools and conform assessments to available question types, while others create bespoke tools to address specific needs. As computer science educators, we can both use and design these tools so it is beneficial to discuss the strengths and limitations of existing tools, as well as the types of questions that can be created. It is our hope that this paper will serve as a catalyst for future discussions.

## Acknowledgments

We gratefully acknowledge Cinda Heeren and Margo Seltzer for designing two of the question types and for their feedback on this paper.

# References

[1] Sohail Alhazmi, Charles Thevathayan, and Margaret Hamilton. 2021. Learning UML Sequence Diagrams with a New Constructivist Pedagogical Tool: SD4ED. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. ACM. https://doi.org/10.1145/3408877.3432521

[2] Maha Aziz, Heng Chi, Anant Tibrewal, Max Grossman, and Vivek Sarkar. 2015. Auto-grading for parallel programs. In *Proceedings of the Workshop on Education for High-Performance Computing (EduHPC2015)*. ACM, Article 3. https://doi.org/10.1145/2831425.2831427

[3] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1062–1068. https://doi.org/10.1145/3408877.3432430

[4] Majdi Beseiso, Omar A. Alzubi, and Hasan Rashaideh. 2021. A novel automated essay scoring approach for reliable higher educational assessments. *Journal of Computing in Higher Education* 33, 3 (June 2021), 727–746. https://doi.org/10.1007/s12528-021-09283-1

[5] Benjamin S. Bloom. 1968. Learning for mastery. *Evaluation comment* 1, 2 (March 1968), 1–12.

[6] Joachim Breitner. 2016. Visual Theorem Proving with the Incredible Proof Machine. In *Interactive Theorem Proving, Lecture Notes in Computer Science*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer International Publishing, 123–139. https://doi.org/10.1007/978-3-319-43144-4_8

[7] Serena Caraco, Nelson Lojo, Michael Verdicchio, and Armando Fox. 2024. Generating Multi-Part Autogradable Faded Parsons Problems From Code-Writing Exercises. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE '24)*. ACM, 179–185. https://doi.org/10.1145/3626252.3630786

[8] Binglin Chen, Matthew West, and Craig Zilles. 2022. Peer-grading "Explain in Plain English": A Bayesian Calibration Method for Categorical Answers. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1 (SIGCSE '22')*. 133–139. https://doi.org/10.1145/3478431.3499409

[9] Paul Denny, Jacqueline Whalley, and Juho Leinonen. 2021. Promoting Early Engagement with Programming Assignments Using Scheduled Automated Feedback. In *Proceedings of the 23rd Australasian Computing Education Conference (ACE '21)*. ACM, 88–95. https://doi.org/10.1145/3441636.3442309

[10] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2002. Graphviz — Open Source Graph Drawing Tools. In *Graph Drawing: 9th International Symposium*. Springer, 483–484.

[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press.

[12] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding "Explain in Plain English" questions using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. ACM. https://doi.org/10.1145/3408877.3432539

[13] Edward F. Gehringer. 2015. Automated and Scalable Assessment: Present and Future. In *2015 ASEE Annual Conference & Exposition*. 26.270.1–26.270.8.

[14] Victor H. Gonzalez, Spencer Mattingly, Jessica Wilhelm, and Danielle Hemingson. 2023. Using artificial intelligence to grade practical laboratory examinations: Sacrificing students' learning experiences for saving time? *Anatomical Sciences Education* 17 (2023). Issue 5. https://doi.org/10.1002/ase.2360

[15] Marcelo Guerra Hahn, Silvia Margarita Baldiris Navarro, Luis De La Fuente Valentín, and Daniel Burgos. 2021. A systematic review of the effects of automatic scoring and automatic feedback in educational settings. *IEEE Access* 9 (2021), 108190–108198. https://doi.org/10.1109/ACCESS.2021.3100890

[16] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM. https://doi.org/10.1145/3159450.3159502

[17] Rhaydae Jebli, Jaber El Bouhdidi, and Mohamed Yassin Chkouri. 2024. A Proposed Algorithm for Assessing and Grading Automatically Student UML Diagrams. *International Journal of Modern Education and Computer Science* 16, 1 (Feb. 2024), 37–46. https://doi.org/10.5815/ijmecs.2024.01.04

[18] Andrew Luxton-Reilly, Ewan Tempero, Nalin Arachchilage, Angela Chang, Paul Denny, Allan Fowler, Nasser Giacaman, Igor Kontorovich, Danielle Lottridge, Sathiamoorthy Manoharan, Shyamli Sindhwani, Paramvir Singh, Ulrich Speidel, Sudeep Stephen, Valerio Terragni, Jacqueline Whalley, Burkhard Wuensche, and Xinfeng Ye. 2023. Automated Assessment: Experiences From the Trenches. In *Proceedings of the 25th Australasian Computing Education Conference (ACE '23)*. ACM, 1–10. https://doi.org/10.1145/3576123.3576124

[19] Lauren E. Margulieux, Brian Dorn, and Kristin A. Searle. 2019. *Learning Sciences for Computing Education*. Cambridge University Press, 208–230.

[20] Patrick McDowell, Christine Terranova, and Kuo-Pao Yang. 2019. Design of an Automated Pseudo-Code Grading Tool. *International Journal of Engineering Research and Technology* 8, 12 (Dec. 2019), 47–50. https://doi.org/10.17577/ijertv8is120016

[21] Jorge Leoncio Rivera Muñoz, Federico Moscoso Ojeda, Dina Lizbeth Aparicio Jurado, Percy Fritz Puga Peña, Christian Paolo Martel Carranza, Haydeé Quispe Berríos, Shanda Ugarte Molina, Amanda Rosa Maldonado Farfan, José Luis Arias-Gonzáles, and Mario José Vasquez-Pauca. 2022. Systematic review of adaptive learning technology for learning in higher education. *Eurasian Journal of Educational Research* 98, 98 (2022), 221–233.

[22] Linda Nilson and Claudia J. Stanny. 2015. *Specifications Grading: Restoring Rigor, Motivating Students, and Saving Faculty Time*. Stylus Publishing. https://books.google.ca/books?id=UrCpCwAAQBAJ

[23] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education* 22, 3, Article 34 (June 2022), 40 pages. https://doi.org/10.1145/3513140

[24] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: A fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06)*. Australian Computer Society, Inc., 157–163.

[25] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2021. Evaluating Proof Blocks Problems as Exam Questions. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER '21)*. ACM, 157–168. https://doi.org/10.1145/3446871.3469741

[26] Ido Roll and Ruth Wylie. 2016. Evolution and revolution in artificial intelligence in education. *International Journal of Artificial Intelligence in Education* 26 (2016), 582–599.

[27] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 15–26. https://doi.org/10.1145/2491956.2462195

[28] Márcio Josué Ramos Torres and Regina Barwaldt. 2019. Approaches for diagrams accessibility for blind people: A systematic review. In *2019 IEEE Frontiers in Education Conference (FIE '19)*. 1–7. https://doi.org/10.1109/FIE43999.2019.9028522

[29] Yixin Wan, George Pu, Jiao Sun, Aparna Garimella, Kai-Wei Chang, and Nanyun Peng. 2023. "Kelly is a Warm Person, Joseph is a Role Model": Gender Biases in LLM-Generated Reference Letters. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 3730–3748. https://doi.org/10.18653/v1/2023.findings-emnlp.243

[30] Matthew West, Nathan Walters, Mariana Silva, Timothy Bretl, and Craig Zilles. 2021. Integrating diverse learning tools using the PrairieLearn platform. In *Seventh SPLICE Workshop at SIGCSE*.

[31] Rose Marie Tan Zhao Yun. 2013/2014. "Online Judge" for Data Structures and Algorithms Course. (2013/2014).

[32] Hedayat Zarkoob and Kevin Leyton-Brown. 2024. Mechanical TA 2: Peer Grading With TA and Algorithmic Support. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE '24)*. 1470–1476. https://doi.org/10.1145/3626252.3630891

[33] Chenyan Zhao, Mariana Silva, and Seth Poulsen. 2024. Autograding Mathematical Induction Proofs with Natural Language Processing. *arXiv preprint arXiv:2406.10268* (2024). https://doi.org/10.48550/arXiv.2406.10268