Edgeworth: Authoring Diagrammatic Math Problems using Program Mutation

Hwei-Shin Harriman
hwei.shin.harriman@gmail.com
Olin College of Engineering
USA

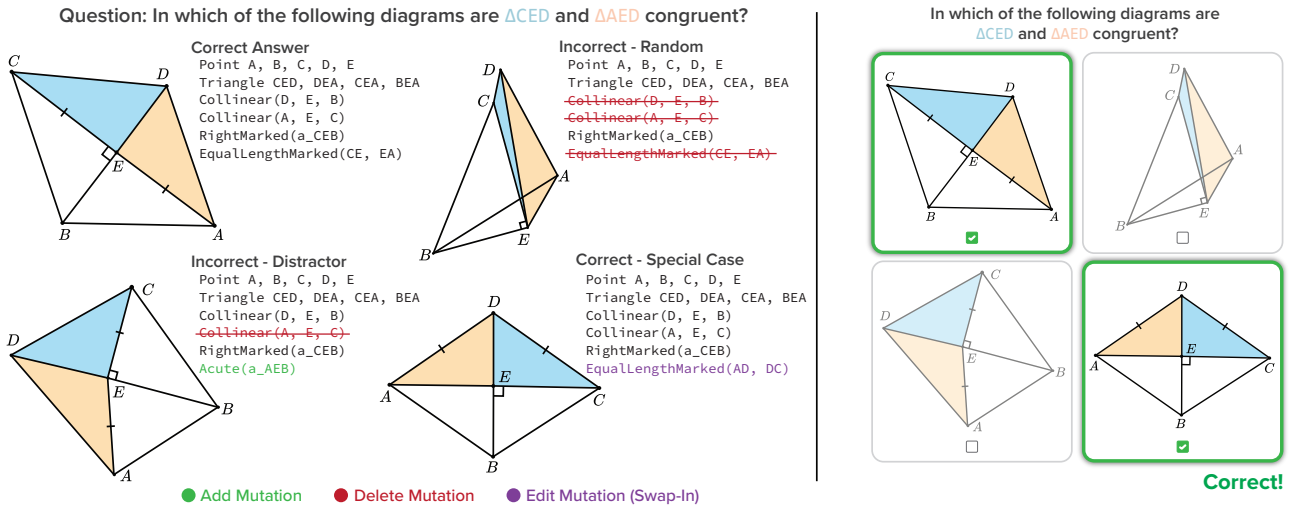

Figure 1. Left: Given a prompt that includes declarative mathematical statements representing a correct diagram, Edgeworth synthesizes examples and counter-examples by mutating the prompt statements. Right: Edgeworth makes it easy to generate diagrammatic multiple-choice questions.

Abstract

Building connections between mathematical expressions and their visual representations increases conceptual understanding and flexibility. However, students rarely practice visualizing abstract mathematical relationships because developing diagrammatic problems is challenging for problem authors. We introduce Edgeworth, a system that automatically generates correct and incorrect diagrams by mutating a given prompt program. We show that Edgeworth can produce diagrammatic problems with minimal author input by recreating problems in a widely used geometry textbook.

1 Problem and Motivation

Solving diagrammatic problems builds connections between mathematical expressions and visual representations, which is known to increase conceptual understanding and flexibility [7]. However, authoring such problems remains a tedious, manual process. Even creating simple shapes can be challenging, so problem authors often resort to recycling course content, providing multiple similar examples, or re-purposing diagrams found online [10]. We interviewed content authors and found that the challenges compound when considering the issue of scale, as many existing tools lack support for high-level tweaking of diagrams or are difficult to use for non-programmers. As a result, even making minor semantic edits to batches of diagrams can quickly become a large undertaking.

Our goal is to create multiple choice problems like the example shown in Figure 1: Right. Edgeworth accomplishes this goal by taking prompt programs and mutating them in various ways, as shown in Figure 1: Left. We evaluated a prototype of Edgeworth (Figure 2) by generating contrasting cases for diagrammatic problems covering the concepts of high school geometry. Contrasting cases (refer to Section 2.1) are similar examples that highlight defining features of a concept. We successfully generated multiple contrasting cases per problem using less than 30 lines of code and a few configuration parameters.

2 Background

Kellman et al. [9] demonstrate that solving questions that map between symbolic and visual representations increases intuition about how the structures equate. They also show that multiple-choice questions enable rapid practice, and that solving these types of problems exposes students to contrasting cases. To automatically generate diagrams that are contrasting cases, Edgeworth combines program mutation with notation-to-diagram translation.

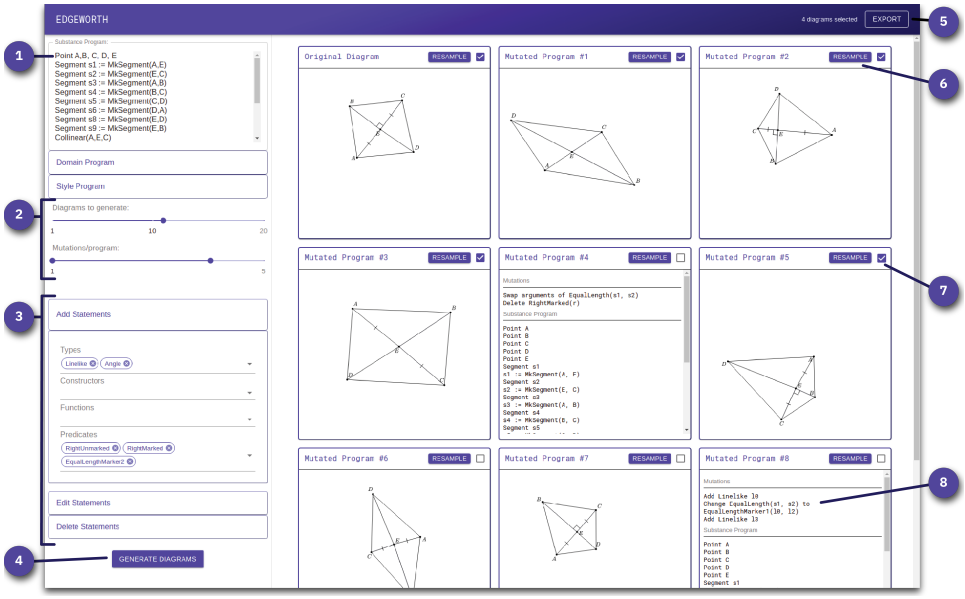

Figure 2. Edgeworth provides a novel problem authoring workflow in a web interface. Creating a problem in Edgeworth involves the following steps : 1. Edit Substance prompt program; 2. Adjust configuration parameters; 3. Select statements of interest; 4. Generate Edgeworth diagrams; 5. Export selected diagrams as SVG format; 6. Resample diagram layout; 7. Select diagram(s) to export with checkboxes; 8. Review mutations applied.

2.1 Using Contrasting Cases to Improve Representational Fluency

Representational fluency is the ability to quickly understand a visual representation and use it to solve domain-specific tasks [15]. Marton [11] showed that contrasting cases—close examples that help people notice features they might otherwise overlook—help students identify key aspects of a particular representation. Early on, students benefit from discerning instances and noninstances that differ only in one dimension of variation.

In Edgeworth, we consider four types of contrasting cases (illustrated in Figure 1: Left). There are correct answers that are semantically equivalent to the prompt program as well as random incorrect answers that are clearly invalid. There are also "distractors," answers that appear correct but are not, and "correct special cases," answers that show a valid alternate representation of the answer in a different or "less obvious" way than the correct answer.

2.2 Program Mutation

Program mutation has long been used as a method to reveal unforeseen characteristics in software programs. It is commonly used in mutation testing [8] and program repair [12] to create new software tests by exploring all possible pathways through a program and uncovering hard-to-find bugs. Mutation is typically accomplished using predefined "mutation operators" to alter programs at the abstract syntax tree (AST) level. The type and specifics of each operator vary between use cases and languages.

2.3 Notation-to-Diagram Translation

Penrose [18] is a platform which allows users to create diagrams by writing mathematical statements in plain text. This text is split into three parts: Substance, Domain, and Style. Substance programs contain mathematical statements and define the semantic content of a diagram. The allowable range of statements is entirely bounded by a Domain schema. Lastly, Style programs define how each of the mathematical statements in Domain should be visually represented. Together, the three programs are used by Penrose to render diagrams with a one-to-one correspondence between the content defined in the Substance file and the visual semantics of the generated diagram. Domain and Style programs can be shared between diagrams, so problem authors can focus on defining the content of their diagrams by using familiar mathematical notation in Substance files.

Penrose diagrams are not limited to any one particular domain of math or science. Ye et al. [18] demonstrated the system's flexibility by creating diagrams in the fields of geometry, set theory, and linear algebra.

3 Related Work

It is well known in the field of STEM education that students benefit from multiple worked examples [13] as well as repeated practice of concepts [5, 17]. As a result, there are many authoring tools for large-scale generation of math problems. Polozov et al. [14] describe a technique for generating personalized mathematical word problems. Their approach combines specifications from a tutor and student to

define the mathematical and narrative requirements respectively, which allows them to auto-generate word problems with a cohesive, personalized narrative. ASSISTment [16] allows the author to create a question, answer, and hints to a particular math problem with template variables throughout. By calculating and swapping out variables in the template, they are able to auto-generate alternate problems. CTAT [2] maintains high-level context about a student's learning habits and understanding, concepts that should be learned, and the types of problems that correlate to those concepts. Like ASSISTment, it supports mass production of similar problems by allowing the author to define variables that can be swapped for a range of values. Images can also be inserted into CTAT problems but they cannot be altered via mass production.

These approaches specialize in auto-generated, personalized tutoring content for a variety of problem types, but none are able to handle diagrammatic problems. On the other hand, Edgeworth has no contextual information about learning objectives and does not try to adjust diagrams to a student's needs. Edgeworth could be integrated with these approaches to provide a more holistic learning experience.

There are also several other implementations of problem and solution generation for various topics in math, such as natural deduction proofs [1] and algebra [3]. They are specialized to handle written or symbolic problems within their relevant mathematical domain, and do not handle diagrams at all.

Gulwani et al. [6] describe a solution generation method for geometric constructions, and show that they can iteratively apply ruler and compass operations to reach a desired geometric solution. Their approach deals with a specific area of geometry, whereas Edgeworth can handle diagram generation in any domain of math.

4 Uniqueness of the Approach

Edgeworth takes a short Penrose Substance program as a prompt, and uses Penrose to create diagrams with a one-to-one representation. Because of the direct relationship between prompt and diagram, changes made to one will alter the semantics of the other. This allows us to use program mutation to alter the prompt in various ways to produce diagrammatic contrasting cases.

Edgeworth takes advantage of the fact that Substance is a high-level domain-specific language (DSL) bounded by Domain to drastically reduce the search space when executing mutations. Using a small set of basic mutation operators, Edgeworth generates meaningful mutated diagrams from only a prompt Substance program and a few configuration parameters. These mutated diagrams are able to cover the four answer types discussed in Section 2.1.

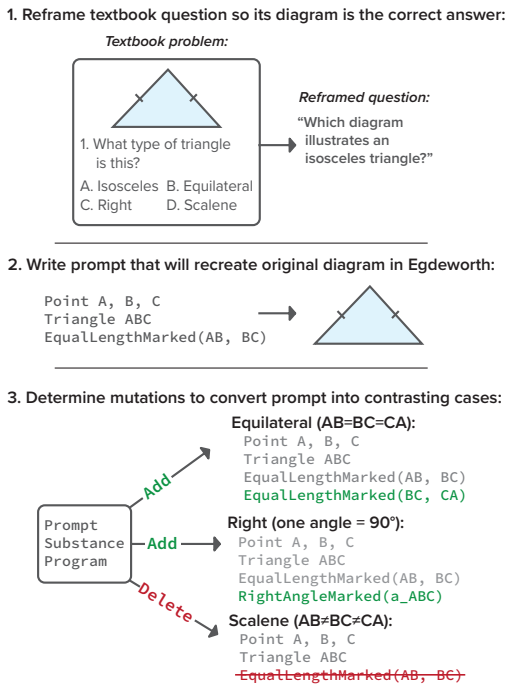

Figure 3. The process used to build our set of mutation operators (top to bottom), repeated across all entries in our representative subset of 24 textbook problems. If a mutation that we had not defined was encountered multiple times in the third step, it was added to our set of operators.

While we demonstrate the potential of Edgeworth by creating Euclidean geometry diagrams in Section 5, Edgeworth's usefulness is by no means limited to this domain. Our approach is domain agnostic—our mutation operators make simple changes to Substance prompt programs, so any Penrose domain works with Edgeworth.

4.1 Defining Mutation Operators

We built our set of mutation operators by examining problems in a widely used high school geometry textbook [4]. The domain of Euclidean geometry was selected because the topics are well represented using diagrams. We examined all 53 diagrammatic problems in the chapter review sections and sampled 24 problems to make up our representative subset. The remaining 29 problems repeated conceptual content.

For each textbook problem, we reframed the question so that the diagram accompanying the original problem could be considered a correct answer (refer to Figure 3). We then wrote the Substance prompt that would recreate the textbook diagram. Then, we considered possible answers to the posed question, including both distractors and correct special cases (i.e. contrasting cases), and determined what mutations would be required to convert the prompt into the proposed mutant. If multiple cases required a mutation operator we had not yet defined, it was added to our set.

4.2 System Design

Edgeworth takes a Substance prompt program and synthesizes numerous alternate Substance files, thereby altering the meaning of the resulting diagrams. The prompt undergoes a number of mutations which are randomly selected from our set of mutation operators. As long as a mutation does not cause a compilation error it is considered a valid mutation by Edgeworth, and random mutations are chosen from the valid subset.

Depending on the scope of the Domain schema, mutating a random statement can break the semantics of the diagram even though the compilation of the mutated program is valid. To address this, Edgeworth also accepts a set of configuration parameters, which allow the problem author to adjust the number of mutations per generated program, and constrain the statements that can be added, deleted, or replaced from within the Domain schema. Thus, we are able to quickly generate diverse variations of the source diagram from a very small input.

It is up to the problem author to select the final subset of diagrams that they wish to include in their question. To facilitate this, we implemented a React web app interface (Figure 2) that allows the author to view the generated diagrams as well as their corresponding Substance program, and select the final subset that they wish to use.

4.2.1 User Experience. The basic Edgeworth experience involves authoring entirely within a React web app interface. We demonstrate the workflow for a small example where a problem author is creating diagrams in the domain of set theory.

1. **Choosing a Domain:** The author wants to make diagrams illustrating subsets in set theory. They begin by opening the set theory Domain and Style programs which have already been authored by an expert Edgeworth user.
2. **Authoring a Prompt:** The author writes a short Substance program, which is checked against the Domain for syntax errors as they progress. For example:
   ```
   Set A, B
   IsSubset(B, A)
   ```
3. **Adjusting Configuration Parameters:** The author chooses the number of diagrams to generate, maximum number of mutations per diagram, and statements of interest. They would like Edgeworth to generate contrasting cases for subsets, so they include `IsSubset` in their statements of interest.
4. **Generating Mutated Diagrams:** Edgeworth takes this information to generate new diagrams with mutated Substance programs, for instance the following mutant where `Equal(B, A)` has replaced `IsSubset(B, A)`:
   ```
   Set A, B
   Equal(B, A)
   ```
5. **Diagram Selection:** The author may examine all the mutated diagrams and their corresponding Substance programs. Some diagrams may be satisfactory from a Substance perspective, but the spacing, layout, or orientation of the diagram could make it unusable. In these cases, the author can use the "Resample" button to re-optimize the layout. The author can also select diagrams to export.

5 Results and Contributions

There were two main questions that we wanted to answer in our evaluation of Edgeworth: does our selected set of mutation operators provide sufficient coverage in our chosen domain of math, and how good is our system at generating contrasting cases?

5.1 Evaluating Mutation Operators

Our examination of high school Euclidean geometry problems uncovered six mutation operators spanning three main types: adds, deletes, and updates. The six operators are:

- **Add** Appends a statement to the Substance program.
- **Cascading Delete** Removes a random statement and all other references to that statement.
- **Swap Arguments** Reorders the arguments passed into a statement. e.g., if `A` and `B` are `Triangles`:
  `Similar(A, B)` → `Similar(B, A)`
- **Swap-In Arguments** Replaces the arguments passed into a statement with other arguments defined in scope. e.g., if `A, B, C, D` are `Points`:
  `s := MkSegment(A, B)` → `s := MkSegment(C, D)`
- **Replace Statement Name** Replaces a statement with a different statement that takes the same type of arguments and has the same return type. e.g., given that `T` is a `Triangle`:
  `Equilateral(T)` → `Scalene(T)`
- **Type Change** Replaces a statement with a new one that takes the same number and type of arguments, but does not necessarily return a value of the same type. e.g., if `E` is an `Angle`:
  `Segment s := Bisector(E)` → `RightAngleMarked(E)`

We identified at least three contrasting cases for each of the 24 problems in our representative set, and determined that it was theoretically possible to mutate the prompt into each one using these six operators. During this process we found that the most common mutations were adds and deletes. While it is possible to mutate any prompt into a contrasting case with only these two operators, the probability of generating a meaningful case is significantly reduced. Edit mutations reduce the search space and make the mutations more readable to the authors. Interestingly, regardless of the length of the prompt Substance program (almost all prompts were fewer than 20 lines), prompts could be converted into contrasting cases with four or fewer mutations, and most were converted in only one or two. Given that

good answers to multiple-choice questions should all appear plausible at first glance it makes sense that it would only take a few choice mutations to produce meaningful results.

5.2 Evaluating System Design

We examined Edgeworth's viability for generating contrasting cases by running trials in the web app. Taking prompt Substance programs created from Section 5.1 and configuration parameters as input, we used Edgeworth to generate batches of 20 diagrams. Each program was mutated one to three times and the correctness of each diagram was determined manually. This process was repeated three times for each of the prompts. We found that Edgeworth easily generates at least one correct or random incorrect answer within the first five programs in any given batch. We were also able to generate one to two correct special cases and one to five distractors per trial. These numbers show promise in Edgeworth's ability to generate contrasting cases, though it is important to note that in order to get these results we had to be intentional about the statements of interest that we selected in the configuration parameters. Due to the broad scope of the Domain (coverage of all the diagrammatic concepts in high school Euclidean geometry) there are certain statements that are better suited for specific problems. Adding too many or too few statements to the configuration parameters often leads to "noisy" diagrams. Choosing the number of mutations that can occur follows similar logic.

5.3 Discussion and Future Work

We implemented Edgeworth as a React app and showed that it is capable of generating contrasting cases on a representative set of problems spanning the topics covered in Euclidean geometry. Our results highlight an exciting step towards problem authors being able to create unique mathematical diagrams with low overhead.

There are a few avenues for future work. Edgeworth currently maintains no high-level understanding about the correctness of any of the diagrams it generates. Analyzing the Substance programs internal data structure using AST search analysis could allow us to infer such information. Saving interesting mutation patterns from prior problems as templates could improve Edgeworth's output quality and reduce its reliance on carefully selected configuration parameters. While our approach is effective when generating related diagrams, it is not optimal if a problem author has only a few specific cases in mind. Intelligent selection of mutations may improve Edgeworth's effectiveness in this case. Additionally, it is important to examine and iterate upon the user experience of Edgeworth as we work towards making it accessible to problem authors who do not have a programming background.

References

[1] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

[2] V. Aleven, B. Mclaren, J. Sewall, and K. Koedinger. The cognitive tutor authoring tools (ctat): Preliminary evaluation of efficiency gains. In *Proc. Intelligent Tutor Students (ITS)*, 2006.

[3] E. Andersen, S. Gulwani, and Z. Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, 2013.

[4] E. B. Burger, D. J. Chard, E. J. Hall, P. A. Kennedy, S. J. Leinwand, F. L. Renfro, D. G. Seymour, and B. K. Wattis. *Holt geometry*. Holt, Rinehart and Winston, 2007.

[5] K. A. Ericsson. *The Influence of Experience and Deliberate Practice on the Development of Superior Expert Performance*, chapter 38, pages 685–706. Cambridge University Press, 2006.

[6] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Proc. Programming Language Design and Implementation (PLDI)*, 2011.

[7] D. Halpern, A. Graesser, and M. Hakel. 25 learning principles to guide pedagogy and the design of learning environments. In *Association of Psychological Science Taskforce on Lifelong Learning at Work and at Home*, 2007.

[8] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[9] P. J. Kellman, C. M. Massey, and J. Y. Son. Perceptual learning modules in mathematics: Enhancing students' pattern recognition, structure extraction, and fluency. *Topics in Cognitive Science*, 2010.

[10] D. Ma'ayan, W. Ni, K. Ye, C. Kulkarni, and J. Sunshine. How domain experts create conceptual diagrams and implications for tool design. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, 2020.

[11] F. Marton. Sameness and difference in transfer. *Journal of the Learning Sciences*, 15(4):499–535, 2006.

[12] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[13] H. Pashler, P. Bain, B. Bottge, A. Graesser, K. Koedinger, M. Mcdaniel, and J. Metcalfe. Organizing instruction and study to improve student learning. *National Center for Education Research*, 2007.

[14] O. Polozov, E. O'Rourke, A. M. Smith, L. Zettlemoyer, S. Gulwani, and Z. Popovic. Personalized mathematical word problem generation. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[15] M. A. Rau. *Conceptual learning with multiple graphical representations: Intelligent tutoring systems support for sense-making and fluency-building processes*. PhD thesis, Carnegie Mellon University, 2013.

[16] L. Razzaq, J. Patvarczki, S. F. Almeida, M. Vartak, M. Feng, N. T. Heffernan, and K. R. Koedinger. The assistment builder: Supporting the life cycle of tutoring system content creation. *IEEE Trans. on Learning Technologies*, 2009.

[17] H. L. Schnackenberg, H. Sullivan, L. Leader, and E. E. K. Jones. Learner preferences and achievement under differing amounts of learner practice. *Educational Technology Research and Development*, 46(2):5–16, 1998.

[18] K. Ye, W. Ni, M. Krieger, D. Ma'ayan, J. Wise, J. Aldrich, J. Sunshine, and K. Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. on Graphics*, 2020.