ETH zürich

Department of Computer Science
Programming, Education, and
Computer-Human Interaction

Merlin: A Markup Language for Algorithm Animations

Shu Wang

Master Thesis, D-INFK

Oct. 2024

Supervisor: Prof. Dr. April Wang

ABSTRACT

Instructors often create visualizations like sketches, diagrams, and animations to help learners grasp abstract programming concepts and form accurate mental models. However, creating these visualizations, particularly step-by-step animations, is often a time-consuming task. In this paper, we explore the design of a markup language and a customized code editor for creating algorithm animations. To inform the design, we analyzed 400 examples from an online coding platform, examining their structure, common elements, and creation process. Based on the findings, we developed Merlin, a declarative language for creating algorithm animations, and Merlin-Lite, a higher-level, simplified version that enables users to define these animations with more concise and minimal code. Additionally, we designed Merlin-Editor that further reduces the burden of fine-grained editing through direct manipulation. We demonstrated the tool's expressiveness through a range of examples and collected usability feedback through a qualitative evaluation with instructors.

ACKNOWLEDGMENT

I am very grateful to my supervisor, Prof. Wang, for her full help and guidance. With her help, I explored the integration of multiple fields such as human-computer interaction, programming language design and web application development in my thesis. I am grateful to Peach Lab for providing me with an excellent platform to support me in completing my thesis research. I am also very grateful to Alice, Di, and other users who participated in the user study for their valuable suggestions and detailed feedback.

1 Introduction

With the increasing importance of computational thinking [10] in education, schools are introducing algorithmic concepts to students at all ages [19]. Even for students who are not expected to become professional programmers, learning algorithmic thinking helps develop problem-solving and logical reasoning skills [8]. The subject of computer science by its nature, requires abstract thinking, which can be difficult for novices who are not used to this method of thinking. Research shows that visual learners can benefit from seeing concepts through shapes, diagrams, and animations. They can better understand abstract ideas when they are made more concrete through visual aids [15]. Visualizing algorithms in shapes and diagrams also help students form mental models that guide their problem-solving process, allowing them to better understand the structure and the flow of the algorithm. In addition to static diagrams, instructors sometimes use *algorithm animations* – a series of step-by-step visualizations that show the change of the values and the data structure over time. These animations are particularly helpful for understanding a dynamic process, such as recursion or iterations.

Despite its usefulness, creating algorithm visualizations is a tedious and time-consuming process since the instructors need to balance precision, clarity, styling, scalability, and reusability. In addition to directly sketching out the visualization on whiteboard or ASCII code [13], existing visualization tools can be classified into two approaches — direct manipulation tools and markup languages. Direct manipulation approaches, such as using GUI-based graphic editing tools like PowerPoint Slides [7] and Figma [2], allow instructors to drag and drop shapes and elements to create visual representations of algorithms. While this approach offers flexibility and more expressiveness in creating and customizing diagrams, it can be a lot of clicking and moving around to ensure that elements like arrays are visually aligned with conventions. Moreover, reusability becomes an issue when instructors need to recreate or adapt visuals for each new program. On the other hand, markup language-based approaches like Mermaid.js [3], Penrose [24], LaTeX's TikZ [5], allow instructors to define algorithm visualizations through structured text notations. These tools separate content from representation, providing a level of abstraction for the instructor so that they can focus on objects and structures rather than visual style. While this approach offers reusability and structured editings, mastering a markup language is a steep learning curve, requiring users to learn the syntax of the notation language. Additionally, modifying and debugging visualizations often require instructors to navigate between the notation code and the rendered graph, which can be overwhelming.

Compared to static algorithm visualization, algorithm animation shows how values and states change over time, and unavoidably, lead to more efforts to create. For example, in a simple Nth Fibonacci Number problem, to illustrate how to infer the 5th number, the instructor need to specify the array 5 times — each time with different values. Either the direct manipulation approach or the markup language approach would be cumbersome for creating the animation for such a simple algorithm problem. Another type of tools such as visuAlgo [12], PythonTutor [11], and Algorithm Visualizer [1] use runtime values to capture the state of the algorithm during the execution and maps it to a visual representation. However, the premise of this approach is that the instructor needs to develop a concrete programming solution beforehand, making it less flexible to create lightweight algorithm animations to illustrate problem walkthrough.

In this paper, we propose a blended approach that allows direct manipulation on the algorithm animations through a GUI interface, while having a markup system for tracking the notation of the animations. Inspired by B2 [23] and Mage [14], we believe that the design of a fluid interaction between code and graphical work can not only benefit data analysis tasks but also algorithm animation tasks. Such a blended approach enables users to adjust specific elements of the animation visually, while leveraging the markup system for ensuring reusability and consistency. This approach can also serve as a middle-layer tool for LLM (Large Language Model) to generate initial algorithm animations from natural language descriptions, which can further lower the learning curve of bootstrap-

ping an animation. The fluid interaction between the code and the GUI editor enables instructors to collaboratively edit the generated animation, ensuring that it is pedagogically sound and visually accurate.

Our goal is to design a system that simplifies the creation of algorithm animations. To approach this, we first conducted a content analysis of 400 expert-created algorithm animations collected from a well-known online coding learning community, identifying common patterns and structures in these animations. Based on this analysis, we developed the Merlin system, a comprehensive tool for creating algorithm animations in educational settings. The Merlin system consists of three components: two custom markup languages — Merlin and Merlin-Lite — and the visual editing tool, Merlin-Editor. Inspired by the design of Vega and Vega-Lite, Merlin and Merlin-Lite aims to ease the description of data structure evolution in animated algorithm visualizations through a two-fold level of abstractions. Merlin provides the first abstraction layer that separates the content from styling. Merlin-Lite provides a higher level of abstraction by separating data from animation. To further ease the process of editing, we designed Merlin-Editor, a mixed-initiative code editor for the Merlin languages. It allows direct manipulation of the visual elements through a GUI interface, ensuring consistency and synchronization between the diagram's notation and the interactive graphical interface for the visual representation.

2 Related Work

As computational thinking has become increasingly emphasized in education, many tools and methods have been developed to facilitate the teaching and learning of abstract programming concepts. Visualization plays a crucial role in this process by providing learners with intuitive, concrete, and vivid representations of algorithms and data structures that are otherwise difficult to grasp through text-based explanations alone [20]. Existing work covers a wide range of approaches, from static diagrams [24] to dynamic, interactive visualizations [18]. These tools not only enhance comprehension by making abstract concepts more concrete but also support the development of powerful mental models for problem-solving and logical reasoning. In this section, we review the literature on visualization tools used in education, explore current tools and methods used to create algorithmic visualizations and discuss how our proposed hybrid approach addresses the limitations of existing solutions.

2.1 Visualization Tools in Education

Visualization tools play an important role in education, especially in areas involving complex and abstract concepts, such as stem subjects. Research has shown that visual representations such as diagrams, animations, and interactive tools can make abstract concepts more concrete and easier to grasp for students of all learning styles (especially visual learners) [22, 17]. Many types of visualization tools are used in educational settings, from simple static images to interactive and dynamic visualizations. Static visualizations, such as textbook diagrams, which provide basic representations of concepts. Penrose [24] and Edgeworth [16] are representative of this type, and they provide visualizations of concepts from STEM disciplines such as set Wayne diagrams, atomic arrangements, etc. However, this approach often lacks interactivity and the ability to show changes over time [22]. Dynamic visualizations overcome this limitation by focusing more on the representation of processes as well as state transitions. However, dynamic visualizations represented by various types of instructional videos (e.g., 3blue1brown, a YouTube video channel targeted at machine learning education) tend to be costly to produce and lack customization of the content: it is the same visualization for different learners. There is a growing shift towards tools that are interactive. Interactive visualization tools such as Vega-lite, Tableau, and Google Data Studio [21] allow learners to edit examples and view the execution of their visualizations in real-time via a GUI, which helps to form a mental model of how the method

works. Among them, vega-lite, which evolved from vega, further simplifies user operations, resulting in an interactive visualization that combines lightweight code editing and generation with drag-and-drop buttons. However, while these tools are effective for specific use cases, they are often inflexible for educators who want to create custom animations or adapt visualizations to different educational needs.

2.2 Visual Aids for Programming Education

Creating algorithm visualizations is an important task for instructors who want to provide more targeted and effective learning experiences. Existing tools for creating these visualizations can be roughly divided into two main approaches: direct manipulation tools and code-based tools [6]. Direct manipulation tools such as PowerPoint Slides [4], Figma [2], and other GUI-based graphics editing software allow users to create visual representations of algorithms by directly manipulating shapes and elements on a canvas. The benefit of this approach is its flexibility and expressiveness; teachers can customize any style of visual elements to their specific needs and style. However, this flexibility comes at a cost: creating visualizations using direct manipulation tools is often a time-consuming process involving many steps to ensure visual alignment and consistency, especially when the content you want to visualize is complex [6]. In addition, direct manipulation tools are often poorly reusable, meaning that instructors may need to recreate or heavily modify visualizations every time they want to show a different example or algorithm, which makes it impossible to create and export animations [6]. Another approach is code-based tools, such as Mermaid.js, Penrose, and TikZ in LaTeX, which use text-based markup languages to define algorithm visualizations. This approach has several advantages, including separating content from presentation, allowing for more structured editing and easier reuse of visual components. Instructors can focus on defining the logical structure of a visualization rather than on style or layout details, which can save time and effort. However, these tools also have significant disadvantages. Learning an additional code language involves a steep learning curve, and making changes or debugging visualizations can be challenging because it often requires navigating between code and rendered output to make adjustments. We are also aware of a new type of runtime value-based tools, such as visuAlgo, PythonTutor, and Algorithm Visualizer, [9] that use runtime values to generate visual representations of algorithms by capturing the state of the algorithm during execution. These tools are very effective for demonstrating existing algorithms that can execute code to produce visual output. However, they are not as effective for more flexible and custom algorithms because these visualizations may not require or need to fully implement the code steps. This approach limits flexibility and is less effective for creating custom animations for different teaching needs.

3 Content Analysis

To understand users' needs for algorithm animation and the patterns of creating algorithm animation, we collected 400 examples from 3100 problems on online coding websites and manually coded them. We summarized 20 hierarchical patterns as shown in Table 1.

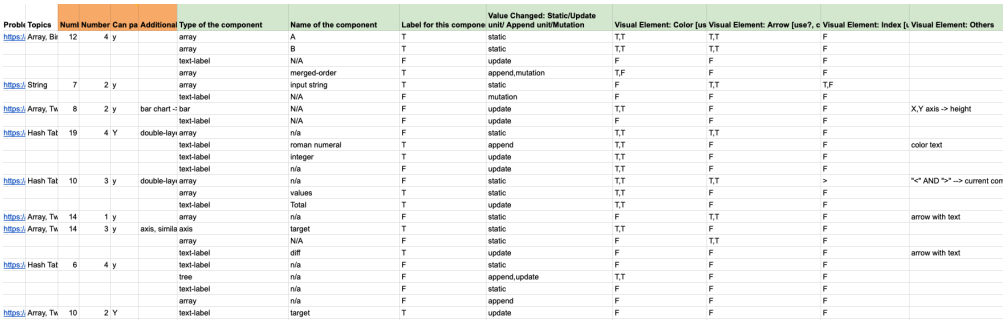
3.1 Method

We adopted the method of manual open coding (as shown in Fig 1)to specify several animation characteristics, which can cover the performance characteristics and creation methods of visual elements. In order to eliminate the uncertainty brought by open code, we asked different researchers to code the same problem and compared the coding results to unify the standards. We divided all samples into two groups, with capacities of 50 and 250 respectively. We used the former group for the unification process: we determined the open-code schema in advance and asked

Object	Type	Frequency	Proportion
page	inheritance	very high	55%
	evolution	very high	80%
	mutation	low	4%
component	array	very high	52%
	matrix	high	25%
	graph	high	15%
	stack	high	12%
	queue	low	2%
	tree	high	18%
	linked list	median	6%
	bar chart	very low	<1%
	star chart	very low	<1%
	text label	very high	75%
	hash map	low	2%
unit	color	very high	90%
	arrow within single unit	very high	65%
	arrow between units	high	33%
	arrow with text label	high	33%
	index	median	8%
	axis	very low	<1%

Table 1: A summary of the Leetcode algorithm animation samples (sample size = 300)

two researchers to independently open-code the 40 samples in this group. Afterward, we compared the open-code results given by the two researchers. We found a small number of differences in coding between researchers. On this basis, we unified the details of the open-code schema, eliminated the differences, and asked two researchers to independently code the remaining 10 samples in the first group. The coding results this time were almost completely the same. We then finished open-coding the 250 samples in the second group with each researcher responsible for the non-overlapping parts. After a review of the open-code results with both researchers, we then quantified the frequency of these patterns, reporting the percentage of occurrences across the dataset. The results are summarized in Table 1, which summarizes the common components and patterns in the current examples.



Probl	Topics	Numl	Number	Can pa	Additional	Type of the component	Name of the component	Label for this compone	Value Changed: Static/Update unit/ Append unit/Mutation	Visual Element: Color [us	Visual Element: Arrow [use?, c	Visual Element: Index [u	Visual Element: Others
https://	Array, Bi	12	4	y		array	A	T	static	T,T	T,T	F	
						array	B	T	static	T,T	T,T	F	
						text-label	N/A	F	update	F	F	F	
						array	merged-order	T	append,mutation	T,F	F	F	
https://	String	7	2	y		array	input string	T	static	F	T,T	T,F	
						text-label	N/A	F	mutation	F	F	F	
https://	Array, Tw	8	2	y	bar chart -> bar		N/A	F	update	T,T	F	F	X,Y axis -> height
						text-label	N/A	F	update	F	F	F	
https://	Hash Tat	19	4	Y	double-layı	array	n/a	F	static	T,T	T,T	F	
						text-label	roman numeral	T	append	T,T	F	F	color text
						text-label	integer	T	update	T,T	F	F	
						text-label	n/a	F	update	T,T	F	F	
https://	Hash Tat	10	3	y	double-layı	array	n/a	F	static	T,T	T,T	>	"<" AND ">" --> current con
						array	values	T	static	T,T	F	F	
						text-label	Total	T	update	T,T	F	F	
https://	Array, Tw	14	1	y		array	n/a	F	static	F	T,T	F	arrow with text
https://	Array, Tw	14	3	y	axis, simila	axis	target	T	static	T,T	F	F	
						array	N/A	F	static	F	T,T	F	
						text-label	diff	T	update	F	F	F	arrow with text
https://	Hash Tat	6	4	y		text-label	n/a	F	static	F	F	F	
						tree	n/a	F	append,update	T,T	F	F	
						text-label	n/a	F	static	F	F	F	
						array	n/a	F	append	F	F	F	
https://	Array, Tw	10	2	Y		text-label	target	T	update	F	F	F	

Figure 1: The Open Code Sheet for Content Analysis

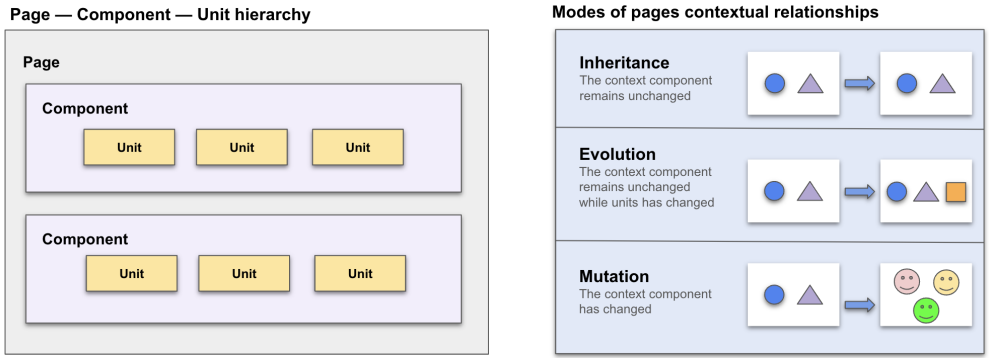


Figure 2: The hierarchical structure and three different modes of pages' contextual relationships

3.2 Result

We found that almost all animation samples conform to the three-layer structure of "page-component-unit" as shown in Figure 2, that is, a complete animation consists of several pages, a page has several components, and each component consists of several units. Here a unit refers to the smallest element of animation, such as a node or an edge in a tree, and a component is a whole composed of several units, which usually conforms to a certain data structure, for example, a tree or an array can be counted as a component. The following content analysis of algorithm visualization samples is based on these three levels.

For pages, we summarize them into three modes based on the contextual relationship: inheritance, evolution, and mutation (as Figure 2). Inheritance means that the components of the page remain unchanged compared to the previous and next pages; evolution means that the components of the page have changed, but are still related to the previous and next pages; mutation means a complete change: starting from a certain page, the visual elements of the previous and next pages are completely unrelated. These three patterns can cover all the inter-page relationships we observe. Among them, inheritance and evolution are the most common. Most visualizations are organized in these two ways. Mutation often appears in a mixed form with the other two modes instead of appearing alone.

For components, we also counted their frequency of occurrence in these 300 samples. Multiple components may appear in the same sample, and we calculate them separately. For example, if two arrays and one tree appear in a sample, we will count two arrays and one tree instead of one array and one tree. The eleven components listed here can cover all the samples we observed. There can be coverage relationships between some components. For example, the stack can often be covered by array, but we divide it into a separate category to facilitate statistics of its frequency of occurrence. It is worth noting that we found that only the following seven components are enough to cover more than 90% of the samples: array, matrix, graph, tree, text label, linked list, and stack. In addition, components are consistent with common data structures. We only observe very limited (3 out of 300) examples beyond common data structures, i.e., the start chart.

For units, we mainly analyze the operations or visual effects performed on each unit. We found that all examples can be covered by the enumerated six formal unit operations. Although they may appear visually diverse, unit operations can be eventually classified into one of these six categories. For example, drawing a graphic next to a unit to emphasize it is actually equivalent to arrows and coloring. Unit operations are mostly used as a supplement to the component for visual emphasis. Through the combination of components and unit operations, algorithm visualization can express complex processes.

4 System Design

Merlin-Editor [1] is implemented as a live web application consisting of the following components: (1) Two code editors for editing Merlin and Merlin-Lite, which can support customized language highlights, auto-complete, code format, etc. (2) A rendering interface that displays animation results in real-time and can synchronize user updates in the animation, and an API that allows users to export animation results as animations or statically. (3) A GUI operation panel, the display content changes according to the selected unit, allowing users to modify the visual results through direct operations and render them simultaneously.

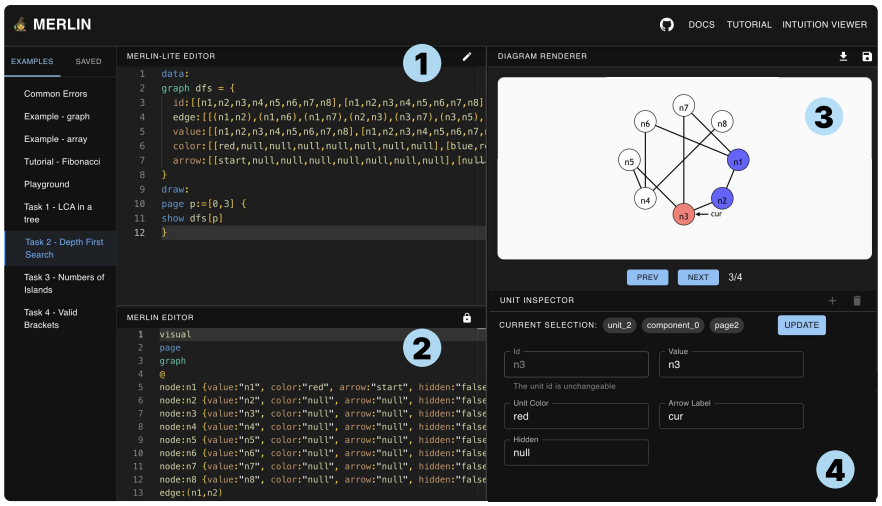


Figure 3: User Interface of Merlin System

4.1 A demo of Merlin-Editor

To demonstrate the user experience of Merlin-Editor, we use a demo described as follows to simulate how instructors use it in real-life programming teaching. Ada, the teaching assistant of the course Data Structure and Algorithms will give a tutorial session of DFS tomorrow and now she wants to make an animation of how DFS works in a graph.

Ada first initializes the first page of the animation. She defines the structure of the graph, such as nodes and edges, using Merlin-Lite. As Ada completes the description of the graph structure in the Merlin-Lite editor, the Merlin code is generated automatically and shown below.

Ada then clicks the add a page button six times, which generates a new six-page animation using the first page she created as a template. To demonstrate the principle of the DFS algorithm, Ada intends to use different colors and arrows with text labels to emphasize the status of the current step of DFS. So she clicks on each visual unit she wants to edit (such as the nodes in the graph), modifies them to the desired color through the GUI editing tool, and adds arrows and labels. When she makes a change, Ada will click the update button to see the new visual changes immediately and check whether the rendering effect is aligned as she wishes.

Ada quickly completes the animation of this example, which has a total of six pages. When Ada switches from the first page to check, she feels that the one step of backtracking in DFS might be confusing to students, so she decides to add some text explanations. To add text explanation, Ada changed the mermaid code part and added a text description to explain the backtracking in the graph.

[1] The website link of Merlin-Editor: https://eth-peach-lab.github.io/merlin/

Fibonacci in Merlin-Lite

```
data:
array arr1 = {
  structure:[[1],[1,1],[1,1,2]]
  value:[[1],[1,1],[1,1,2]]
  color:[[null],[null,null],[null,null,null]]
}
draw:
page p:=[0,2] {
show arr1[p]
}
```

Fibonacci in Merlin

```
visual
page
array
@
1 {color:"null", arrow:"null"}
@
page
array
@
1 {color:"null", arrow:"null"}
1 {color:"null", arrow:"null"}
@
page
array
@
1 {color:"null", arrow:"null"}
1 {color:"null", arrow:"null"}
2 {color:"null", arrow:"null"}
@
```

Algorithm Animation

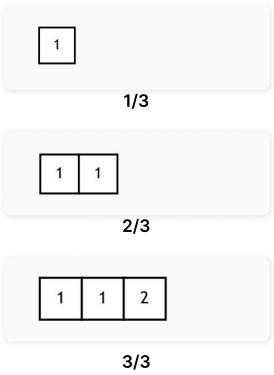


Figure 4: The different descriptions of Fibonacci array by Merlin(right) and Merlin-Lite(left)

Finally, Ada finished all the editing and was satisfied with the animation she just created. She was ready to embed it into the slides she prepared. So she clicked the export button to export the animation in gif or SVG format so that students could see an animation of the DFS algorithm's demonstration.

4.2 Merlin and Merlin-Lite

Merlin and Merlin-Lite are markup declarative languages we created for specifically algorithms animations. Merlin is derived from Mermaid and Merlin-Lite is a simplified version of Merlin, focusing on the data structure and page description. The relationship between them is a two-fold structure inspired by Vega and Vega-lite: Merlin is more descriptive by explicitly defining each unit attribute; while Merlin-Lite is a high-level encapsulation based on Merlin, giving up some freedom but with better generalization. Figure 4 shows the difference between Merlin and Merlin-Lite when describing the same object.

In Merlin, users need to describe each unit following the page-component-unit level. For each object, users should first assign them a unique ID and add optional attribute values wrapped by {}. Merlin is a completely descriptive language that specifies the attributes of each object and generates the object instance by describing these attributes.Merlin's syntax schema always conforms to this abstract paradigm:

```
page_id
component_id {attribute1:"value1", ...}
@
unit_id {attribute1:"value1", attribute2:"value2", ...}
unit_id {attribute1:"value1", attribute2:"value2", ...}
@
```

Figure 5 shows an example of graph component type in Merlin: all the nodes and edges must be listed, as well as their attributes such as color or value. In total we support seven component types based on the content analysis as Table 1, which can cover over 85% of all algorithm animations. In Table 2 we list some the component type examples by Merlin and its each component's supported attribute.

Graph in Merlin

```
graph
@
node:n1 {value:"n1", color:"red", arrow:"start", hidden:"false"}
node:n2 {value:"n2", color:"null", arrow:"null", hidden:"false"}
node:n3 {value:"n3", color:"null", arrow:"null", hidden:"false"}
node:n4 {value:"n4", color:"null", arrow:"null", hidden:"false"}
node:n5 {value:"n5", color:"null", arrow:"null", hidden:"false"}
node:n6 {value:"n6", color:"null", arrow:"null", hidden:"false"}
node:n7 {value:"n7", color:"null", arrow:"null", hidden:"false"}
node:n8 {value:"n8", color:"null", arrow:"null", hidden:"false"}
edge:(n1,n2)
edge:(n1,n6)
edge:(n1,n7)
edge:(n2,n3)
edge:(n3,n7)
edge:(n3,n5)
edge:(n4,n5)
edge:(n4,n6)
edge:(n4,n8)
@
```

Rendered Graph

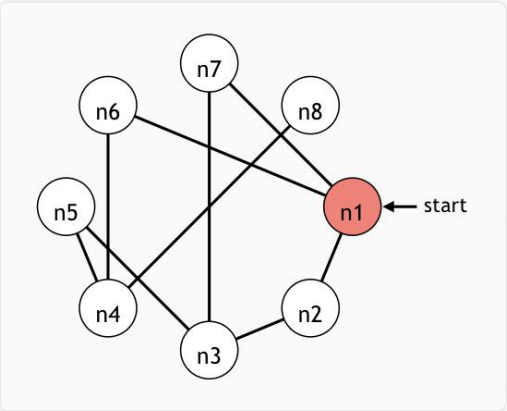


Figure 5: Merlin syntax example: graph

Supported Component	Component Attributes	Unit Attributes
array	title, text label, index	color, arrow, value
stack	title, text label, index	color, arrow, value
matrix	title, text label, index	color, arrow, value
linked list	title, text label, index	id, color, arrow, value, hidden
tree	title, text label, index	id, color, arrow, value, hidden
graph	title, text label, index	id, color, arrow, value, hidden
text	title, text label	

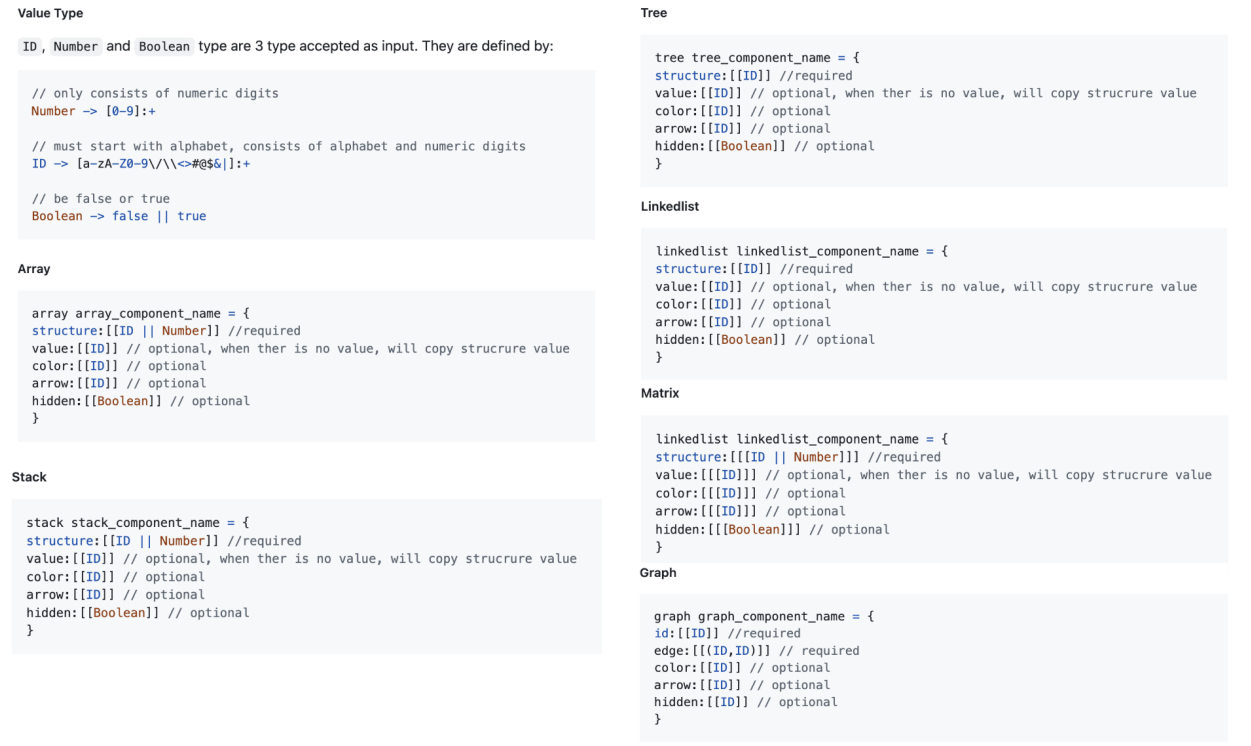
Table 2: Summary of supported components type in Merlin

The implementation of Merlin is based on an extension of the open-source project Mermaid[3]. Mermaid has its own domain-specific language, using langium as the language engine and d3.js as the visualization tool. Based on Mermaid, Merlin completes the expansion of Mermaid domain-specific language by changing langium's configuration file and adding new syntax rules. In addition, Merlin also uses d3.js to draw its own visual instance. Through such an extension, users can call the original Mermaid API to use Merlin functions.

Based on Merlin, Merlin-Lite is developed as a simplified version with a focus only on the data structure and page structure themselves. Merlin-Lite is also a descriptive language, but compared to Merlin's completely specific description method, it has more general descriptions. For example, instead of describing each unit on each page, in Merlin-Lite we first load the data and then use control statements to generate the content of each page. As shown in Figure 4, Merlin-Lite requires significantly less code work than Merlin for the same Fibonacci process. This benefits from the introduction of control flow in Merlin-Lite syntax schema, and will be of great contribution to users in quickly generating visual templates. Merlin-Lite's syntax schema always conforms to this abstract paradigm:

```
data: // load the data
    component_type component_id  = {
        attribute1 : [attribute1 description]
        attribute2 : [attribute2 description]
        ......
    }
    component_type component_id  = {
        attribute1 : [attribute1 description]
        attribute2 : [attribute2 description]
        ......
    }
    ......
opt: // operations on the loaded data
    page page_index {
        show component_id  component_idx
        show component_id component_idx
        ......
    }
    page page_index {
        show component_id  component_idx
        show component_id component_idx
        ......
    }
    ......
```

Figure 6 shows the syntax schema defined in Merlin-Lite, which shows a similarity with object-oriented language and reduces the difficulty of describing data using Merlin. In the editor, users can choose to switch between Merlin or Merlin-Lite to make animations: when they need to quickly produce graphics, they use Merlin-Lite with high-level instructions, which describes data structures with less code and duplication; when they need to make more fine-grained adjustments or go beyond the data structure, they use Merlin (such as when Ada adds a text la-



Value Type

`ID` , `Number` and `Boolean` type are 3 type accepted as input. They are defined by:

```
// only consists of numeric digits
Number -> [0-9]:+

// must start with alphabet, consists of alphabet and numeric digits
ID -> [a-zA-Z0-9\/\\<>#@$&|]:+

// be false or true
Boolean -> false || true
```

Array

```
array array_component_name = {
structure:[[ID || Number]] //required
value:[[ID]] // optional, when ther is no value, will copy strucrure value
color:[[ID]] // optional
arrow:[[ID]] // optional
hidden:[[Boolean]] // optional
}
```

Stack

```
stack stack_component_name = {
structure:[[ID || Number]] //required
value:[[ID]] // optional, when ther is no value, will copy strucrure value
color:[[ID]] // optional
arrow:[[ID]] // optional
hidden:[[Boolean]] // optional
}
```

Tree

```
tree tree_component_name = {
structure:[[ID]] //required
value:[[ID]] // optional, when ther is no value, will copy strucrure value
color:[[ID]] // optional
arrow:[[ID]] // optional
hidden:[[Boolean]] // optional
}
```

Linkedlist

```
linkedlist linkedlist_component_name = {
structure:[[ID]] //required
value:[[ID]] // optional, when ther is no value, will copy strucrure value
color:[[ID]] // optional
arrow:[[ID]] // optional
hidden:[[Boolean]] // optional
}
```

Matrix

```
linkedlist linkedlist_component_name = {
structure:[[[ID || Number]]] //required
value:[[[ID]]] // optional, when ther is no value, will copy strucrure value
color:[[[ID]]] // optional
arrow:[[[ID]]] // optional
hidden:[[[Boolean]]] // optional
}
```

Graph

```
graph graph_component_name = {
id:[[ID]] //required
edge:[[(ID,ID)]] // required
color:[[ID]] // optional
arrow:[[ID]] // optional
hidden:[[ID]] // optional
}
```

Figure 6: Simplified data schema in Merlin-Lite

bel). By combining the two languages for different purposes, users can use the Merlin-Editor to create algorithm animations faster and more efficiently in a variety of scenarios.

4.3 GUI for Refinement

We provide a GUI editing panel to provide direct operation, through which users can modify the visual unit in a code-free manner. This mode of operation is similar to common creativity tools, such as PowerPoint or Figma. What users see is what they get, and any edits will be rendered immediately in real-time. In particular, for algorithm animation, we define different component types and the attributes each component possesses. The supported components and attributes are based on our analysis of existing animations on online programming problem-solving websites to ensure that our tools are expressive enough to produce a variety of algorithm animations. For example, when Ada uses GUI to modify her DFS animation, she clicks on a graph node, and the GUI panel will display the current attributes status of this unit: ID, color, value, text-label, and hidden. Ada inputs blue as color and current node as text label. Then she clicks the update button and saw that the selected unit was visually updated based on her input. Ada was not satisfied with the change, so she changed the color to red to make it more eye-catching. During the whole process, Ada does not need to switch between code and visual units. She can only focus on animation, which makes it very fast for her to perform such attribute modification operations. With GUI editing, we save users from having to switch their attention back and forth between and visual elements, making it very convenient to make repetitive and minor adjustments, such as changing colors or deleting nodes in a graph.

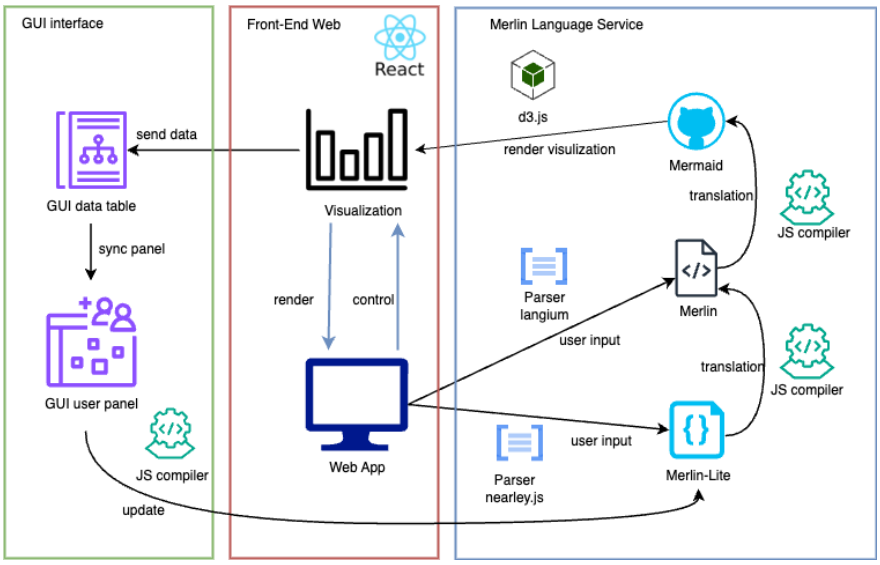


Figure 7: An overview of tech stack in Merlin System

4.4 Exporting and Sharing Algorithm Animations

Merlin-Editor supports users to export animations in an animated way: after the user completes an animation with multiple pages, Merlin-Editor can integrate multiple pages into a single SVG file with inline scripts, supporting a slides-like presentation method; multi-page animations can also be integrated into a single file that automatically plays, such as gif format. Therefore, users can generate the desired animation or static pictures according to different teaching scenarios. For example, in a lecture, if an instructor wants to introduce his students the concept of the stack, he will choose to use a gif to show the continuous pop and push process; while in an exercise class, if a lecturer needs to explain the backtracking algorithm to confused students, he will choose to use static pages so he can stop at key steps, annotate on the visual elements and explain them in more detail. By supporting static and animated animation, users do not need to reprocess the results after using Merlin-Editor. The output of Merlin-Editor can be directly embedded into various educational scenarios, such as course slides, tutorial PDFs, exercise class handouts, online quizzes, etc. This gives instructors enough flexibility and convenience, allowing them to have a one-stop experience greatly improves their efficiency in preparing teaching materials.

5 Implementation

Merlin, Merlin-Lite, and Merlin-Editor are mainly written in Javascript based on React.js, with adopting several popular third-party-libraries such as Markdown editor, d3.js, mermaid, nearly-parser, MUI and etc. The tech-stack overview is shown in Figure 7, which can be roughly divided into the following three parts: markup language, code editor and interactive interface.

5.1 Markup Language

In this system, we implement two types of markup languages: Merlin and Merlin-Lite, which is a high-level encapsulation of Merlin. Merlin language is implemented based on mermaid grammar, and the grammar is defined and parsed through langium. It does not have a compiler, and the parsed input will be used to interact with d3.js

to generate SVG graphics. Merlin-Lite is developed based on Merlin and we adopt nearley.js as its parser which brings a more flexible domain-specific-language design and higher performance of the parser process. We write a mini-compiler for Merlin-Lite using JavaScript to translate the Merlin-Lite into Merlin. We implement and keep the interactive interface using Merlin and Merlin-Lite, so that users can switch freely between coarse-grained but more efficient Merlin-Lite and fine-grained but more-operational Merlin.

5.2 Code Editor

In order to achieve a user experience similar to mainstream editors (such as vscode), we adopted monaco-editor to build the code editing area. Benefiting from the extensibility of Monaco-editor framework, we support several popular features in the existing code editors such as customized Merlin and Merlin-Lite grammar highlights, editor shortcuts, bracket matching, auto-complete and etc. Users with former coding experience can transfer their coding habits into our system with no gaps.

5.3 Interactive Interface and Renderer

The interactive interface with users is based on React.js and MUI.js. We use the react framework to unify all third-party libraries such as Monaco-editor, nearey.js, and mermaid.js, and use react hooks for state management to ensure that users can always see the latest real-time rendering effects. MUI.js provides our system with a modern UI/UX design, which has a similar layout and UI elements with mainstream code-editing tools, saving users from having to learn a new interaction logic.

6 Case Studies on Coverage

In this section, we will explore the Merlin Editor's coverage of algorithm animation to see whether it can meet the instructor's teaching needs and produce modular visual content that meets the requirements. This section will be divided into the following contents: collection and classification of existing algorithm animation examples, case studies of various algorithm animation contents, and a short conclusion.

6.1 Data Analysis of Animation Examples

We selected Leetcode, a popular programming problem-solving website, as the example source of the case study. We used a web crawler to crawl about 3100 questions and their editorial parts by the time the thesis was drafted and analyzed the algorithm animation parts they contained. The results are shown in the following figures. Based on these results, we conduct the case study for coverage by verifying whether Merlin can re-cover the most popular topic from existing Leetcode problems.

On the LeetCode website, visualizations for algorithms are always presented in the form of slides with several pages. Figure 8 shows the proportion of questions with visualization slides. We further explore the relationship between the topic of the question itself and whether there is a visualization slide, as shown in Figure 9. The proportion of LeetCode problems with visual slides is not large overall. For some special topics (such as mathematics), this proportion is significantly low due to the high diversity of content, lack of a unified template, and difficulty of presenting, etc. Figure 10 reveals the visual slides' number for different algorithm topics, providing evidence on selecting the topics for the case study in the following sections.

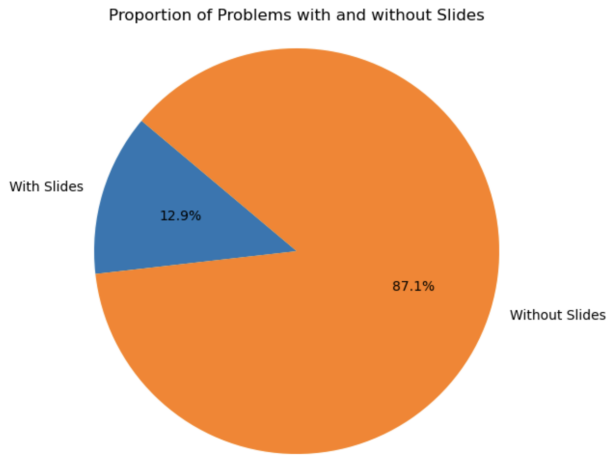


Figure 8: Leetcode Problems with/without visualization slides ratio

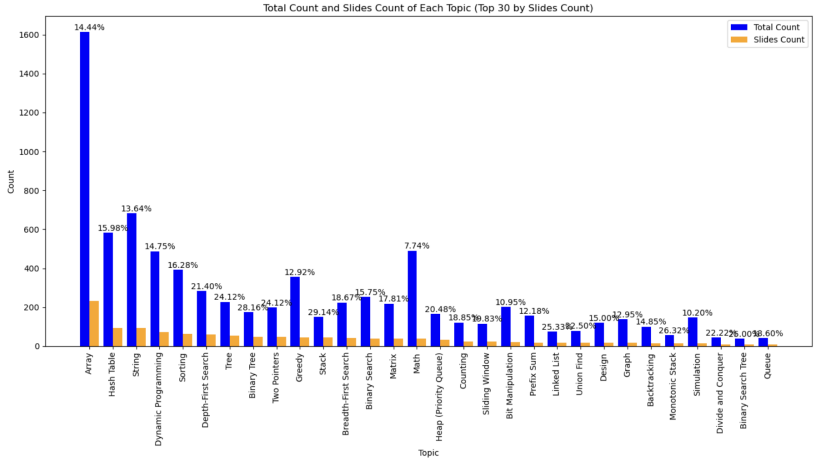


Figure 9: Ratio of algorithmic problems on different topics with visual slides from Leetcode

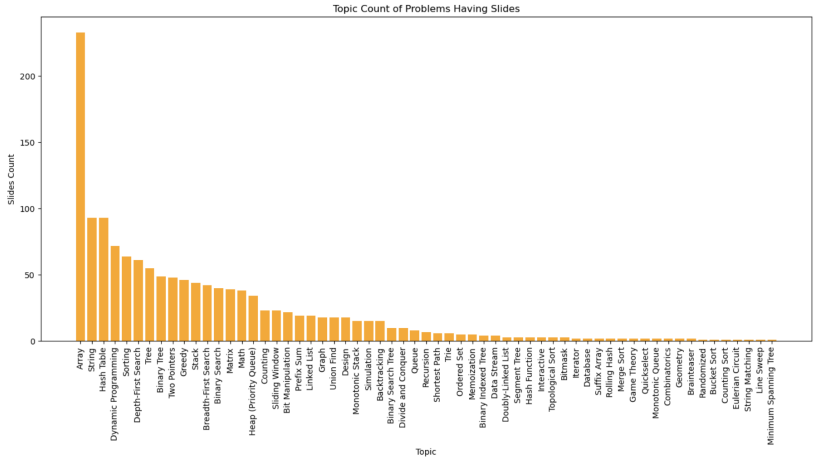


Figure 10: The number of visual slides for each topic of Leetcode problems

6.2 Case Studies

In this section, we will discuss algorithm topics of different types based on their visualization presentation. We will select some common topics and their representative examples, and compare them with the visualizations we reconstructed using Merlin to analyze Merlin's expressive power and its coverage of algorithm topics.

6.2.1 Case Study of 1d-structure topics

Algorithmic problems expressed using one-dimensional data structures are the most common type, including but not limited to the following topics: arrays, hash tables, stacks, linked lists, queues, etc. They constitute the majority of algorithmic problems, accounting for more than 70% of all problems. For this type of problem, the visualization method is to describe the data structure and change the values in the data structure step by step as the algorithm evolves. Algorithm visualizations of 1d-structures are usually straightforward and have few variations, which leads instructors to prefer presenting the complete process.

Figure 11 and figure 12 are comparisons of the slides provided by LeetCode and the visualization produced by instructors using our Merlin. Apart from minor stylistic differences (such as the position of arrows), the visualizations produced by Merlin are almost identical to existing slides and can almost completely cover all the information and content in the original slides. In the appendix, we provide a complete Merlin visualization demonstration of the example *3Sum*.

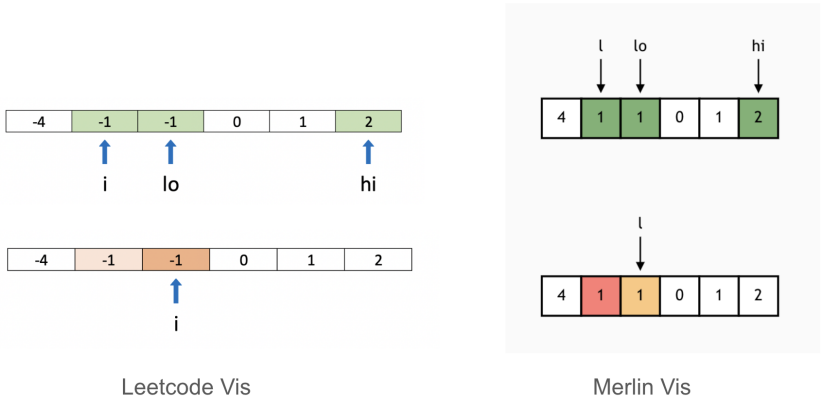


Figure 11: The Comparison between Leetcode existing visualization slides (left) and Merlin visualization (right) for problem 3sum

6.2.2 Case Study of 2d-structure topics

Algorithmic topics that use 2D structures are matrices and dynamic programming. Although there are more complex data structures, instructors present them in the same straightforward way as 1D structures: changing the values of variables as the algorithm evolves. However, due to the more variables and more complex structures, instructors are less likely to show the entire process in all demonstrations. The use of Merlin-Editor allows instructors to draw these 2D data structures more quickly, further making it easy to present the complete algorithm process.

We selected the classic LeetCode problem, *the number of islands*, as a representative of this topic. Figure 13 shows the comparison between the existing LeetCode visualization slides and the animation made by instructors

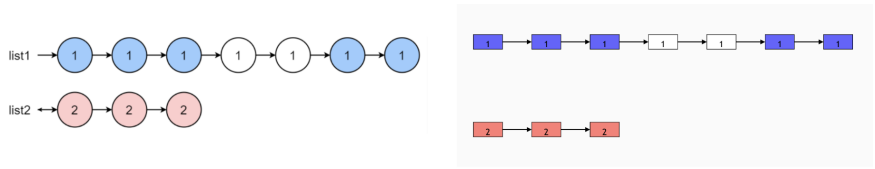


Figure 12: The Comparison between Leetcode existing visualization slides (left) and Merlin visualization (right) for problem merge in linked list

using Merlin. The visualizations made by Merlin are well suited for the graphical representation of this topic and even have more advantages regarding less time consumption. The full demonstration of this example by Merlin can be found in the appendix as well.

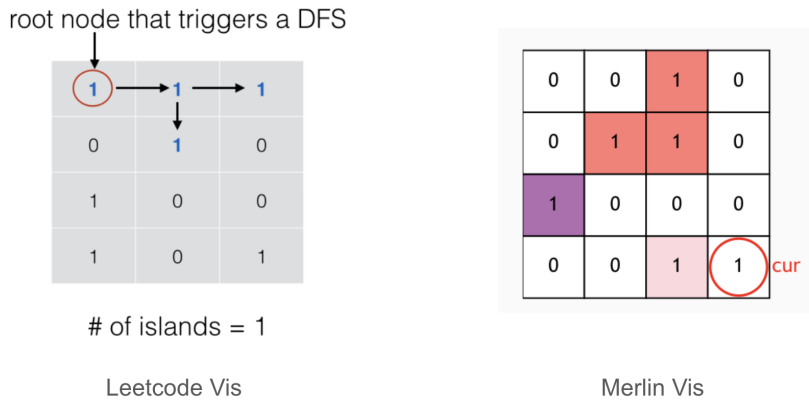


Figure 13: The Comparison between Leetcode existing visualization slides (left) and Merlin visualization (right) for problem numbers of islands

6.2.3 Case Study of graph topics

Graph topics refer to algorithm types that involve node and edge data structures, including topics such as graph, tree, topological sort, etc. When creating visualizations for such topics, creators usually capture a few key fragments rather than the entire process. At the same time, for such objects, creators usually use small data structures (such as a binary tree with a depth of 3 instead of a binary tree with a depth of 5), which is also because the visualization of such algorithms is relatively complex and time-consuming.

We use Leetcode problem *find the lowest common ancestor in a binary tree* as an example, as shown in the figure 14. Leetcode's existing visualization slides can be perfectly reproduced with Merlin. In fact, for this type of problem, instructors often use hand drawing to reduce their production time while Merlin is a good tool to supplement their

visualization method with ease and higher efficiency. The full visual process of this algorithm problem by Merlin can be found in the appendix.

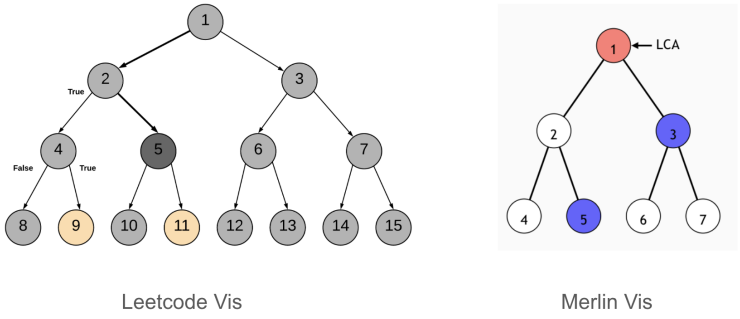


Figure 14: The Comparison between Leetcode existing visualization slides (left) and Merlin visualization (right) for problem LCA in a binary tree

6.2.4 Case Study of other topics

In the case study, although the above three general types can cover more than 80% of the collected examples, there are still some cases that are difficult to reconstruct using Merlin. They are either more special data structures or mixed presentations. Due to their complex and changeable characteristics, it is difficult for us to find a general solution to the problems they represent, and therefore it is difficult to design a visual domain-specific language for these problems.

We have summarized several types of these cases that are difficult to recover by Merlin. The first type is due to rare or special data structures. Merlin's implementation does not currently support this type of data structure, but if this type of problem is crucial for instructors in the future, Merlin's support is technically feasible. The example is shown as figure 15. Horizontal bars appear very rarely in all cases (2/400), so we didn't design Merlin to support them, but it is technically possible when necessary. Another case in which Merlin cannot recover well is hybrid data structures. Currently, Merlin only supports basic data structures such as trees, graphs, and stacks, and its support for their variants or fusions is not flexible enough. As shown in the figure 16, in the example *Rotten Orange* there is a mixed tree/graph structure (right part), and our tools are not yet strong enough to express this part. In addition, Merlin currently only supports basic elements and structures, such as circles and squares, but no more vivid visual elements or even custom graphics, such as apples and oranges uploaded by users. The visualization with more vivid visual elements shown in Figure 17 cannot be recovered well by Merlin at present.

6.3 Results of Coverage Study

Through the above case study, we found that Merlin have very good expressive ability and can cover most of the topics in Figure 9. For various data structure problems, Merlin provides visualization grammar and rendering support, which meets the instructors' teaching requirements in education settings. By comparing the visualization made from Merlin, we point out that the limitations of Merlin lie in the lack of more customized content, such as more flexible text labels, handwriting-like annotation functions, and insufficient support for a few individual topics

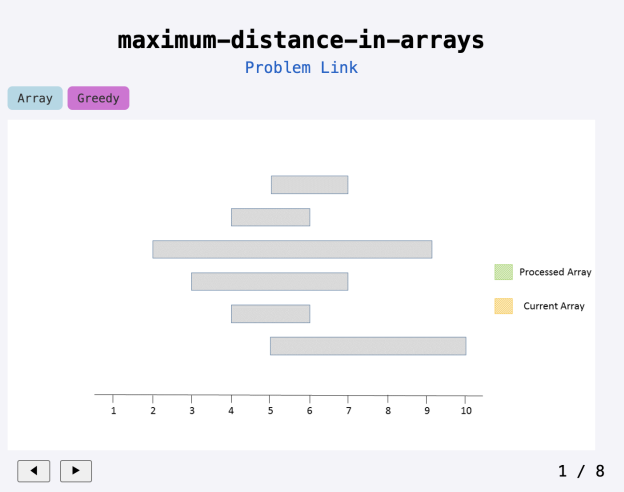


Figure 15: Special visual case of bar chart from Leetcode problem maximum distance in arrays

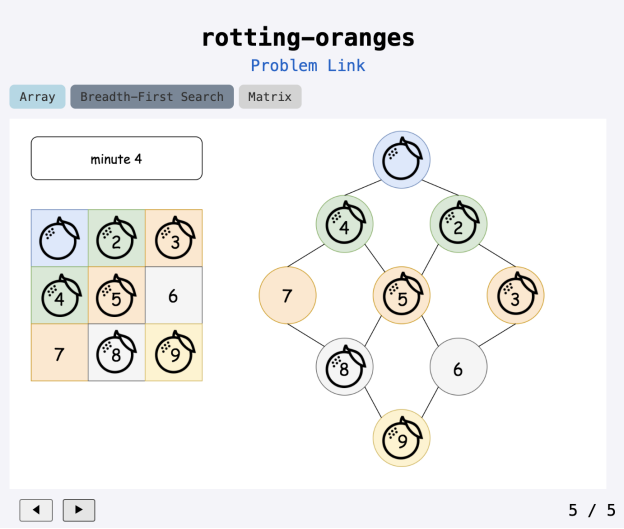


Figure 16: Special visual case of mixed data structure from Leetcode problem rotten oranges

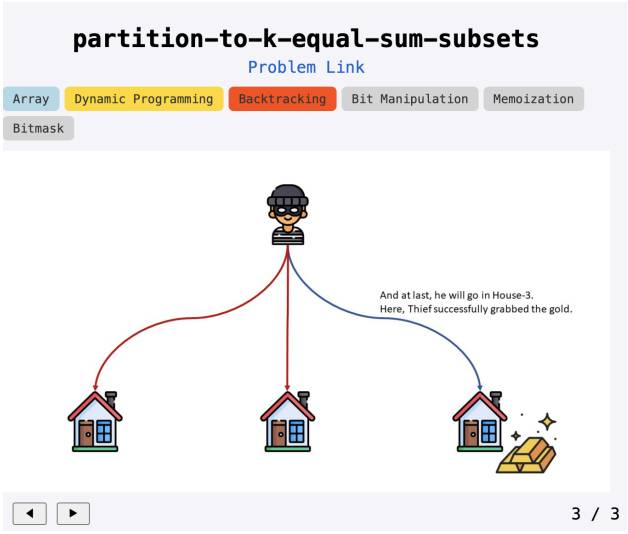


Figure 17: Special visual case of complicated visual elements from Leetcode problem partition to k equal sum subsets

(such as bar charts under the math topic). In the future study, various and flexible support of these functions should be added into Merlin system.

7 User Study

To evaluate the usability of the Merlin system and understand users' habits of making algorithm visualizations, we conducted a lab study with 8 participants. In this study, participants will first undergo a short training session, then be asked to complete 4 tasks selected from real-life scenarios, and finally undergo an interview about their experience using the Merlin system.

7.1 Participants

We recruited 8 participants (4 male, 4 female) from local universities through a mailing list. All of them have at least one year of programming experience and are familiar with basic data structures and algorithms. Among the 8 participants, 5 participants (P1 – P5) had served as teaching assistants for at least one programming course and had experience in preparing courseware and giving exercise sessions. The remaining three participants (P6 – P8) were STEM majors who had no experience as teaching assistants in programming courses but had frequently produced algorithm animations (daily or weekly). All participants mentioned that they have rich prior experience in consuming and creating algorithm animations. Commonly used animation tools mentioned include PowerPoint slides, tableau, markdown, canvas, d3, etc. None of the participants had experience using code-based runtime animation tools, such as Algorithm Visualizer. Table 3 summarizes the background of all the participants.

7.2 Study Protocol

The user research consists of three parts: a tutorial session (15 minutes), a task session (45 minutes), and an interview session (30 minutes). In the tutorial session, participants are introduced to the Merlin system, covering the editing and syntax of both Merlin-Lite and Merlin, as well as the use of the GUI editing panel. At the end of the tutorial, users are given a playground environment to experiment with creating their own algorithm animations. The task

Participant	Role	Major	Experience of Coding (years)	Familiar Languages
1	TA	Computer Science	3–5	C++, MATLAB, Python
2	TA	Computer Science	>5	C++, Java, TypeScript, Python
3	TA	Computer Science/Math	1–3	Java, JavaScript, Python
4	TA	Computer Science/Math	3–5	C++, JavaScript, Python
5	TA	Computer Science	>5	C++, Java, Python, Golang
6	Student	Data Science	3–5	C++, Python, MATLAB
7	Student	Computer Science	3–5	C++, Java, Python
8	Student	Math	1–3	Python

Table 3: Overview of Participants Background in User Study

session involves four programming animation tasks sourced from online problem-solving platforms. These tasks are arranged in increasing order of difficulty and animation complexity, ranging from modifying a provided single-page animation to creating a complete algorithm animation from scratch. During this session, participants have access to the Merlin system documentation, which addresses all aspects of its use, and they are encouraged to create animations independently. However, they are also allowed to ask questions if needed. Following the task session, a one-on-one interview is conducted to gather feedback on the participants' experiences with the Merlin system and the comparison with other animation tools. The specific interview questions are provided in the Appendix.

7.3 Results

After completing the user study, we recorded and organized the participants' task completion performance, interview responses, and questionnaires about the Merlin system, and conducted qualitative and quantitative analysis. The results and findings are shown in the following sections.

7.4 Quantitative Results

Through the user survey submitted by the participants, we compiled a bar chart as shown in Figure 18. Figure 18 illustrates user ratings for various attributes of the Merlin system on a scale from 1 (Strongly Disagree) to 7 (Strongly Agree). The attributes cover the system's expressiveness, utility, ease of use, and ease of learning. The color gradient, ranging from red (low scores) to green (high scores), visually represents the distribution of responses across these categories.

7.4.1 Expressiveness of Animation

Ratings for the expressiveness of the Merlin system varied, with one participant rating it below 4, indicating that they felt that the Merlin system was not expressive enough. However, the vast majority (7/8) rated it between 5 and 7, with half (4/8) rating it strongly (a score of 7). In addition to the individual interviews, participants also told us the reasons for their ratings: participants who gave high ratings felt that the Merlin system was adequate for most animation scenarios in programming courses, while participants who gave low ratings felt that the Merlin system's animations were currently limited to specific graph templates, which were not attractive enough for them. *"I wish I could annotate or take notes on the graphs freely"*, said P6.

There was also a clear distinction between animated and static visualizations when evaluating the usefulness of the system for creating animations. "Helpful for making animated visualizations" received highly positive ratings, with 6/8 responses in the 7 range, indicating that users strongly agreed that the system was effective in this regard.

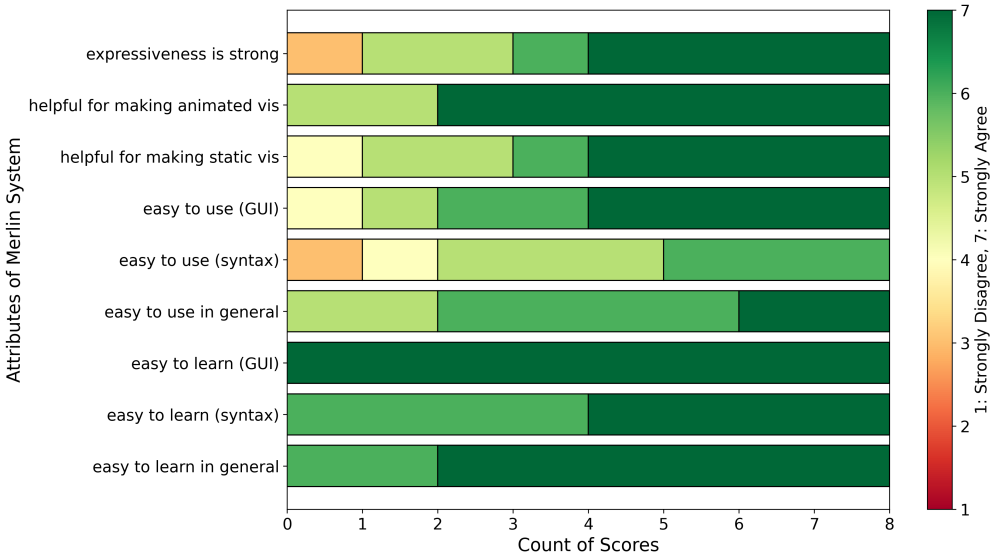


Figure 18: Participants' ratings on Merlin System user experience

In contrast, the distribution of "helpful for making static visualizations" is more even, with 5/8 ratings between 5 and 7, but the ratings in the middle range are scattered, indicating that users have a moderate level of agreement. These results suggest that the strength of the system lies particularly in its support for animated visualizations.

7.4.2 Ease of Learning

The learnability of the Merlin system was assessed across GUI, syntax, and overall learning experience. "easy to learn (GUI)" stood out with high consensus, with absolutely 8/8 users rating it 7, indicating that the graphical interface is a significant advantage and is easy to learn for users with basic coding knowledge. The "easy to learn (syntax)" attribute, while generally strongly positive, also showed some variation. 4/8 users rated it 7, indicating that most users found the syntax very easy to learn. The remaining participants (4/8) rated it 6, indicating that a subset of users might face minor difficulties, perhaps due to complex syntax or insufficient documentation. Finally, the "easy to learn in general" attribute showed strong positive feedback, with about 6/8 of the scores in the 7 range. This suggests that the system as a whole is considered easy to learn, reinforcing the overall positive perception of the system's learnability.

7.4.3 Ease of use

The usability of the Merlin system was evaluated across three dimensions: graphical user interface (GUI), syntax, and general usability. The attribute "easy to use (GUI)" received mostly high scores, with 6/8 rated from 6 to 7, indicating that most users found the GUI easy to use. The attribute "ease of use (syntax)" showed a more even distribution of scores across the range, with 2/8 of the responses ranging from 1 to 4 and the remaining participants ranging from 5 to 6, none rate it as highest 7. This distribution suggests that people have different opinions on the ease of use of the system's syntax, and further improvements or better user support are needed to improve the experience. So far the current negative feedback focuses on the readability of the markup language. In terms of overall ease of use, the attribute "ease of use overall" showed positive opinions, with 6/8 of users giving scores of 6 and 7. Nevertheless, the remaining users gave moderate scores (3 and 4), indicating that while many users found the

system generally easy to use, there is still room for improvement.

7.5 Qualitative Results

We conducted one-on-one interviews with each participant and asked participants about their experience using the Merlin system and how it compares to other animation tools (see the Appendix for detailed questions). By organizing what participants share during the interview, we obtained the following interesting findings:

7.5.1 Different needs by TAs and learners

TAs and learners put forward significantly different needs and expectations, which is also reflected in their overall scores. The average score of TAs in the 7.4 section reaches 6.3, which is much higher than the average score of 5 by learners. Compared with learners, they pay special attention to the efficiency of making algorithm animations. When asked, "In which scenario would you consider using the Merlin system", each of them mentioned that the Merlin system allows them to make some standard animations very quickly. P2 said, *"I need to make a lot of visualizations, and there will be dozens of pages in the slides used for the exercise. With Merlin I can finish it in 20 minutes."* However, learners' expectations were more towards expressiveness and interactivity. *"I need to be able to annotate the graphs freely, just like I can do with GoodNote on my iPad now."*, P7 said. When asked about the features that need to be added in the future, TA participants all focused on improving efficiency, such as a more convenient editor and enabling import templates created by others; while learner participants proposed more interactive possibilities, such as joining online peer-learning and freer content editing. When we told P4 about this interesting finding about the different perspectives of TAs and the learners, he commented as follows, which summarizes the reason for this finding. *"I don't need Merlin to help me understand the algorithm. I already understand it, I just use Merlin to make the visualization. It is very fast and I like it."*

7.5.2 Direct Manipulation vs. Coding-based Manipulation

Merlin system provides a hybrid visualization method of direct manipulation and coding-based manipulation. By observing the participants' behavior in the task session and their answers in the interview, we found that different participants have significantly different preferences for the way to make animations. We found that heavy programmers (such as those with more than 5 years of programming experience) showed a preference for code-based manipulation: when they can use Merlin markup language and GUI to make visualizations, they always choose to use markup language; other participants showed a preference for a hybrid mode, switching between markup language and GUI operation. P3 explained why on this point, *"When making structural changes or creating frameworks, I will use markup language. But when I fine-tune the unit, I will use GUI, which is more direct and convenient."*. We also introduced the third mode represented by the algorithm visualizer, the run-time value mode, to the participants. The algorithm visualizer has no gui-editing module and is a completely coding-based animation. Interestingly, when we asked whether they would prefer to use the algorithm visualizer visualization method, even the heavy coder participants who previously showed full coding-based manipulation said no. *"The algorithm visualizer requires much more coding than Merlin. I have to write a complete algorithm for a picture, but in Merlin I only need to give a small piece of code. Although I use code to control visualization most of the time, the GUI makes it much easier for me to adjust visual elements."* P1 explained to us why he did not think the algorithm visualizer model was better.

7.6 Summary of Key Findings

The above qualitative and quantitative analysis of the Merlin System has brought us meaningful inspiration. Here we summarize the findings:

- Participants generally believe that the Merlin System is very easy to learn, and its similarity with other programming languages can help them get started quickly.
- Participants have different preferences for direct manipulation and coding-based manipulation, and most of them like mixed operation modes; in addition, even deep programmers think that the direct manipulation provided by GUI is useful.
- TA participants and learner participants have different expectations for the Merlin System. TA participants give Merlin System a higher score because they see it as an efficient and convenient visualization tool; while learner participants want a more free-expression and interactive learning tool.
- Participants think that Merlin system has significant advantages in making animated visualizations.

8 Discussion

Our findings provide valuable insights into the use and potential of the Merlin system for algorithm animation in educational settings. Through user study, content analysis, and system evaluation, we identified the key benefits of the Merlin system, making it a versatile tool for educators and learners. At the same time, our research identified a number of areas for improvement to better meet user needs and expectations. In this section, we discuss the implications of our findings for the design and development of animation tools, explore potential enhancements to the Merlin system, and propose directions for future research. By reflecting on the identified strengths and limitations, we aim to provide a comprehensive perspective on how Merlin can evolve to more effectively support diverse educational roles and scenarios.

8.1 Advantages and Limitations

The interviews with various educational roles revealed several notable advantages of the Merlin system. A significant benefit is its dual-mode approach that integrates both code-based and direct manipulation animation. This flexibility allows users to switch between modes fluidly, which proved particularly useful when handling complex animations. Participants consistently highlighted that the combination of Merlin-Lite (coding-based) and GUI editing (direct manipulation) enhanced their workflow, enabling faster completion of visual tasks. Moreover, the Merlin system was praised for its ease of use; following a brief 15-minute training, participants were capable of independently creating algorithm animations and applying personalized adjustments without difficulty.

Despite these positive findings, the user studies also identified several areas needing improvement. One major limitation is the system's expressiveness. Many participants desired a more flexible, drag-and-drop interface similar to what is offered by general-purpose tools like PowerPoint, where any shape can be created and positioned freely. Currently, Merlin-Lite is limited to predefined visual elements and fixed positions, without a drag-and-drop feature, which restricts creative possibilities and user control over the design.

Another challenge lies in the system's user interface (UI) and interaction logic. The current UI is somewhat basic, lacking features like customized graphic highlighting and automatic positioning, which become problematic when animations and associated code increase in complexity. Furthermore, the Merlin system currently supports input

from different levels—Merlin-Lite, Merlin, and GUI editing—which sometimes results in data inconsistencies that confuse users. These inconsistencies suggest the need for refined UI design and more seamless integration between different input modes to avoid user frustration.

8.2 Integrate Large-Language Model

We envision integrating large-scale language models (LLMs) into programming education through the Merlin system to enhance user experience and streamline the process of creating animations. In this approach, Merlin-Lite would act as an intermediary layer that bridges natural language input with visual elements. For example, users could enter a natural language prompt such as "Generate a 12-page Fibonacci sequence," and the LLM would generate the corresponding Merlin-Lite code automatically. This code would then produce a preview of the 12-page animation directly in the Merlin-Editor.

This integration allows users to easily adjust and refine the animation using the Merlin-Editor, without needing to manually write Merlin-Lite code. By using the Merlin system as an intermediate layer, we can mitigate the uncertainty typically associated with LLM-generated content and reduce the dependency on high-quality prompts. Furthermore, this approach provides users with significant flexibility to customize and modify the generated animations directly, eliminating the need to repeatedly fine-tune prompts to achieve the desired output.

Overall, integrating LLMs with the Merlin system could greatly simplify the creation of algorithm animations, making the process more accessible to both educators and learners by combining the power of natural language input with the flexibility of direct manipulation in the editor.

8.3 Customization and User-Defined Visualizations

Our user research highlighted the need for greater customization and flexibility in visualization tools. Users expressed a strong desire for the ability to create highly customized visualizations that go beyond the limitations of existing templates. This indicates a demand for more sophisticated support within the Merlin system, such as enabling user-defined components, visual styles, drags-and-pull operations and interaction methods. By providing these enhanced customization capabilities, Merlin could better accommodate diverse user needs, ranging from specific educational scenarios to personal preferences for visualization aesthetics and functionality. Furthermore, the research suggests that integrating Merlin with other educational and development tools could significantly enhance its usability and effectiveness. Potential integration with environments such as integrated development environments, online coding platforms, or learning management systems would allow users to visualize algorithms within their existing workflows. This seamless integration could reduce the cognitive load associated with switching between tools and improve the overall learning and development experience. Future work could explore these integration possibilities to expand the use cases for Merlin and make it a more versatile tool for both educators and developers.

8.4 Instructor vs. Student Perspectives

Our user study revealed that instructors (including teaching assistants) and students have distinct expectations and needs when it comes to visualization tools. Instructors and teaching assistants prioritize tools that enhance efficiency, speed, and flexibility while also maintaining a high level of expressiveness to reduce the time required for lesson preparation. On the other hand, students are more focused on interactivity, integration with code, and more personalized content. They seek visualization tools that not only help in visualizing algorithms but also support practical coding tasks, such as using visual aids for code debugging and learning.

To address these differing needs, future research may focus on enhancing the Merlin system to better serve both groups. For instructors, improvements could center around efficiency and ease of use, such as introducing an online material library with algorithm visualization templates created by other users, which can be easily imported and customized. Additionally, providing support for more flexible rich text content could facilitate easier annotation and explanation for teaching purposes.

For students, the focus could be on enhancing interactivity and practical learning experiences. This might include adding features for collaborative learning within development groups, integrating visualizations with multiple programming languages, and incorporating guided coding exercises. By tailoring specific features to the needs of each educational role, the Merlin system can become a versatile tool that allows both instructors and students to achieve their unique educational objectives.

8.5 Reflection on DSL Design

In this work, we designed domain-specific language Merlin and Merlin-Lite aims to enable users to have a more efficient control flow of producing visual algorithm animation via coding. We used the third-party open-source library nearley.js as the DSL engine for the parser and built AST functions. During DSL design, our main considerations were lower learning difficulty and compilation efficiency. For the former, we implemented it by imitating other popular programming languages and achieved good results in user studies. However, we also found that imitating other popular programming languages may cause confusion to users, especially when we imitate more than one popular programming language; We improved the compilation efficiency by limiting the flexibility of DSL. For example, we limited the use of spaces, which can greatly reduce grammatical ambiguity or fuzziness, thereby achieving higher parser and compilation efficiency. However, through user studies, we found that this method is somewhat counter-intuitive, especially when we expect Merlin to be similar to other popular programming languages: users will bring their habit of using spaces when using other programming languages to Merlin, which will cause unpredictable compilation errors. Therefore, the DSL design of Merlin and Merlin-Lite needs further polishing and improvement. We hope to provide users with an experience as similar as possible to the programming languages they already know without compromising too much compilation efficiency.

8.6 Various Requirements of Online Editor

Similar to the different perspectives of instructors and students, we found that users also have different expectations and requirements for online editors. In general, most users still share common needs: users hope that the Merlin-Editor can have more visual aids, such as highlighting selected elements, flexible control panels, etc. The differences in user needs mainly lie in whether they focus more on coding methods or GUI methods. Users who focus on coding methods hope to get an experience more similar to the VSCode code editor. One user even mentioned that if the code editor is comprehensive enough, he doesn't need a GUI interface anymore. Users who focus on the GUI experience are completely the opposite. Their ideal experience is more like PowerPoint slides with built-in code modules: users have absolutely free customization experience and can use code modules to reduce repetitive operations. For Merlin, we need to further define our target audience and usage needs in order to better improve the user experience of Merlin-Editor.

9 Conclusion

To explore the key characteristics of algorithm visualization, we conducted a content analysis of 400 visualizations from a popular online coding platform, gaining insights into common design elements and methods used by algorithm visualization creators. Based on these findings, we developed the Merlin system, a comprehensive tool for visualizing algorithms in educational contexts, which includes the custom markup languages Merlin and Merlin-Lite, and the visual editing tool Merlin-Editor. The Merlin system combines programming-based and direct-manipulation visual editing, providing users with a flexible and user-friendly experience.

To evaluate the system's capabilities, we conducted a user study. Our evaluation demonstrated that the Merlin system can effectively produce a wide range of algorithm visualizations, covering most examples found in our analysis. The user study, involving teaching assistants and students, revealed key insights into the different needs of programming educators and novice users, as well as valuable feedback on the system's user experience, such as its interactive logic, learning curve, and desired features.

These findings underscore the potential of the Merlin system to support diverse educational scenarios by providing a flexible tool that accommodates both programming-based and direct-manipulation editing approaches, which also reveal the different needs of visualization tools from the instructor's perspective and the student's perspective. Meanwhile, our content analysis offers valuable insights into the design considerations for future visualization tools, emphasizing the shared pattern and visual elements widely adopted by creators. Together, our content analysis and user study provide a comprehensive understanding of how algorithm visualizations are created and used in educational settings, guiding future research and development of more effective, user-centered visualization tools in both instructor-centered and learner-centered settings.

A Merlin and Merlin-Editor Demo Website

The Merlin and Merlin-Editor demo versions have been deployed to the Internet and any user can access and try them out through this link. From the demo website, you can see more details by looking through docs and tutorials.

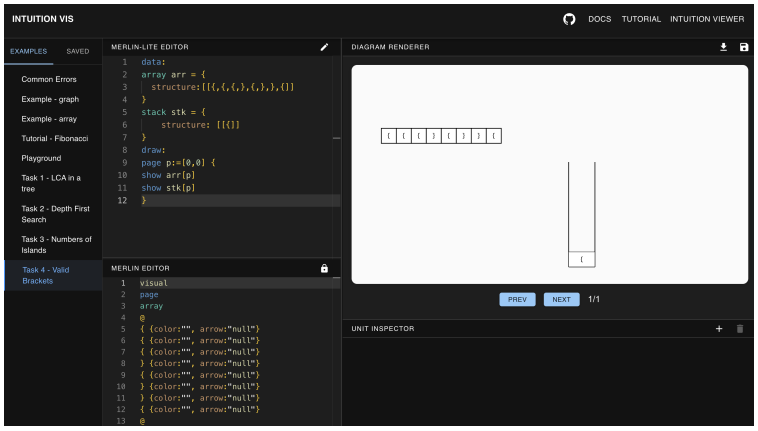


Figure 19: The web demo of Merlin

B Algorithm Visualization Viewer Demo Website

The cases mentioned and collected in section *Case Study* are organized in a single website. It can be accessed through this link. Users can check specific case by searching topics or titles.

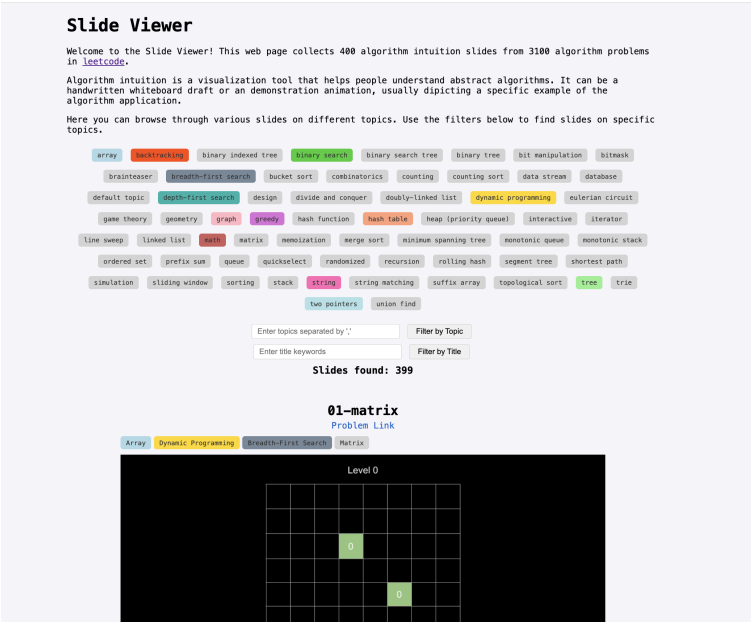


Figure 20: The web demo of collected examples from Leetcode for Case Study

C Visualizations produced by MERLIN in Case Study

Here are the full visualization generated by MERLIN and MERLIN-EDITOR mentioned in *Case Study*.

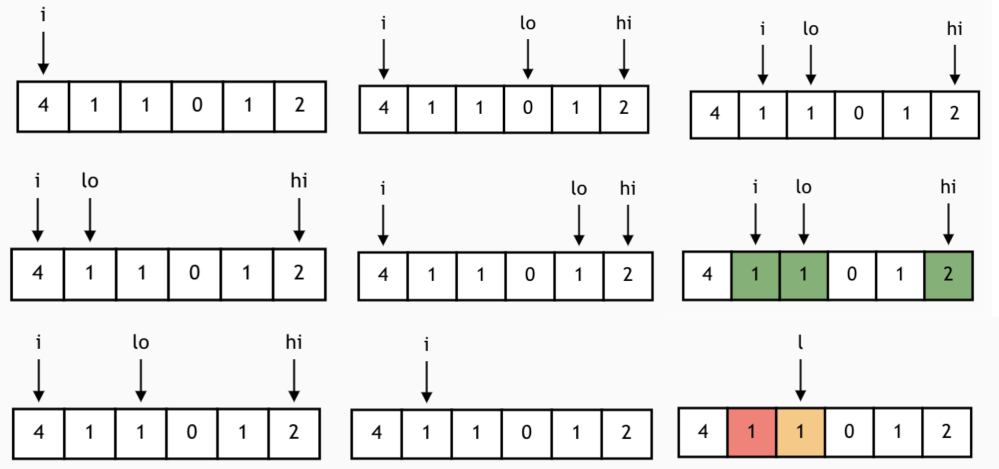


Figure 21: The Visualizations for Leetcode Problem *3Sum* by MERLIN

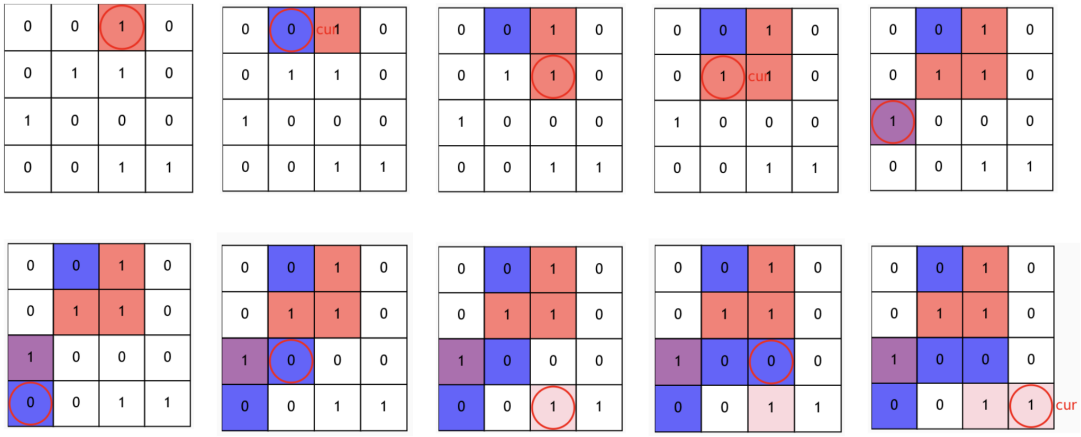


Figure 22: The Visualizations for Leetcode Problem *Numbers-of-Islands* by MERLIN

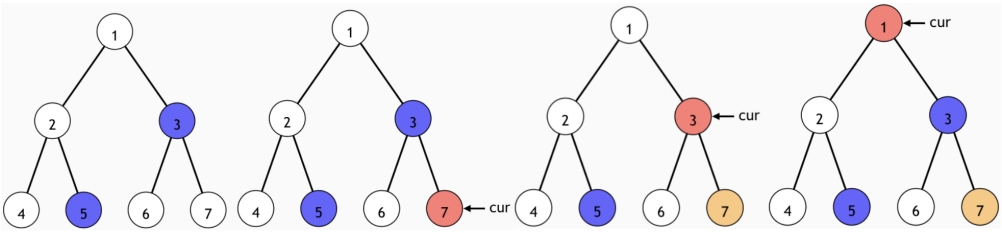


Figure 23: The Visualizations for Leetcode Problem *The LCA in a binary tree* by MERLIN

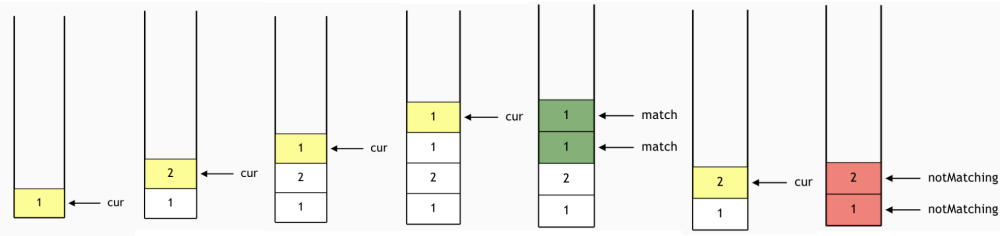


Figure 24: The Visualizations for Leetcode Problem *The Valid Brackets Matching* by Merlin

D Interview Questions of User Study

- How long have you been coding?
- What programming languages are you most familiar with?
- Do you find the Merlin system easy to learn?
- Do you think the markup languages used in the Merlin system are similar to any languages you already know?
- Does this similarity help you get started with the Merlin system?
- Do you think the Merlin system is easy to use overall? What factors influence your opinion?
- Do you find it easy to use the Merlin system for creating simple visualizations?
- Do you find it easy to use the Merlin system for creating complex visualizations?
- Do you find it easy to use the Merlin system for creating algorithm visualizations?
- What features do you think should be added to the Merlin system?
- In what scenarios would you consider using the Merlin system?
- What tool do you most commonly use for similar visualization tasks?
- Under what circumstances would you choose the Merlin system over your preferred tool?
- (After introducing a runtime tool like Algorithm Visualizer) How do you feel about this tool?
- How would you compare the three tools (your preferred tool, Merlin, and Algorithm Visualizer)? Please rank them in order of preference.
- Why did you rank the tools in this order?

References

[1] 2024. Algorithm Visualizer. https://algorithm-visualizer.org/

[2] 2024. Figma. https://www.figma.com/

[3] 2024. Mermaid.js. https://mermaid.js.org/

[4] 2024. PowerPoint Slides. http://www.poker-edge.com/stats.php

[5] 2024. TikZ. https://tikz.net/

[6] Michel Adam, Moncef Daoud, and Patrice Frison. 2019. Direct manipulation versus text-based programming: An experiment report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 353–359.

[7] Wim Blokzijl and Bas Andeweg. 2007. The Effect of Text Slides Compared to Visualizations on Learning and Appreciation in Lectures. In *2007 IEEE International Professional Communication Conference*. 1–9. https://doi.org/10.1109/IPCC.2007.4464074

[8] Fatih Kursat Cansu and Sibel Kilicarslan Cansu. 2019. An overview of computational thinking. *International Journal of Computer Science Education in Schools* 3, 1 (2019), 17–30.

[9] Barnini Goswami, Anushka Dhar, Akash Gupta, and Antriksh Gupta. 2021. Algorithm Visualizer: Its features and working. In *2021 IEEE 8th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)*. IEEE, 1–5.

[10] Shuchi Grover and Roy Pea. 2013. Computational thinking in K–12: A review of the state of the field. *Educational researcher* 42, 1 (2013), 38–43.

[11] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 579–584.

[12] Steven Halim. 2015. Visualgo–visualising data structures and algorithms through animation. *Olympiads in informatics* 9 (2015), 243–245.

[13] Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip Guo, and Haijun Xia. 2024. Taking ASCII Drawings Seriously: How Programmers Diagram Code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16.

[14] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid moves between code and graphical work in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 140–151.

[15] Michael B McGrath and Judith R Brown. 2005. Visual learning for science and engineering. *IEEE Computer Graphics and Applications* 25, 5 (2005), 56–63.

[16] Wode Ni, Sam Estep, Hwei-Shin Harriman, Kenneth R Koedinger, and Joshua Sunshine. 2024. Edgeworth: Efficient and Scalable Authoring of Visual Thinking Activities. In *Proceedings of the Eleventh ACM Conference on Learning@ Scale*. 98–109.

[17] Nazmus Saquib, Rubaiat Habib Kazi, Li-yi Wei, Gloria Mark, and Deb Roy. 2021. Constructing embodied algebra by sketching. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16.

[18] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[19] Amber Settle, Baker Franke, Ruth Hansen, Frances Spaltro, Cynthia Jurisson, Colin Rennert-May, and Brian Wildeman. 2012. Infusing computational thinking into the middle-and high-school curriculum. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. 22–27.

[20] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. 2010. Algorithm Visualization: The State of the Field. *ACM Trans. Comput. Educ.* 10, 3, Article 9 (aug 2010), 22 pages. https://doi.org/10.1145/1821996.1821997

[21] Genifer Snipes. 2018. Google data studio. *Journal of Librarianship and Scholarly Communication* 6, 1 (2018).

[22] Karen L Vavra, Vera Janjic-Watrich, Karen Loerke, Linda M Phillips, Stephen P Norris, and John Macnab. 2011. Visualization in science education. *Alberta Science Education Journal* 41, 1 (2011), 22–30.

[23] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging code and interactive visualization in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 152–165.

[24] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 144–1.