In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission for extensive copying of my thesis for scholarly purposes may be granted by the Dean of the Graduate School, Dean of my academic division, or by the University Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

Redux: Design and Implementation of a Reusable Web-Based Visualization System for Algorithmic and Logical Problem Solving

by

Andrija Sevaljevic

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in the Department of Computer Science

Idaho State University

Spring 2026

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of Andrija Sevaljevic find it satisfactory and recommend that it be accepted.

Paul Bodily,
Major Advisor

Leslie Kerby,
Committee Member

Maryam Bagherian,
Graduate Faculty Representative

Table of Contents

List of Figures

List of Tables

Redux: Design and Implementation of a Reusable Web-Based Visualization System for Algorithmic and Logical Problem Solving

Thesis Abstract – Idaho State University (2026)

Building and maintaining interactive visualization tools for computer science education has historically required significant developer expertise, limiting the breadth of topics such tools can cover and their longevity over time. Redux is an open-source web application originally developed to address the lack of visualization tools for NP-completeness education. This thesis presents substantial extensions to Redux that transform it from a narrowly focused NP-completeness tool into a general-purpose platform for interactive computer science education. Our contributions include a standardized backend API, support for multiple visualization types per problem, automated transitive gadget highlighting, step-by-step algorithm execution, and a suite of reusable visualization templates. Together, these contributions lower the barrier to contribution such that most students can add new visualizations with minimal effort, enabling Redux to grow as a crowd-sourced knowledge base spanning a broad range of computer science topics. Redux is accessible online via `redux.portneuf.cose.isu.edu`.

Keywords: computer science education, algorithm visualization, web-based learning, NP-completeness, extensible platform

Chapter 1

Introduction

Engagement and understanding among students in computer science (CS) curriculum is greatly enhanced through the use of interactive visualizations inside the classroom [23]. Despite this fact, only one fifth of teachers report regular use of such visualization tools in their programming courses [15]. Without such tools, educators typically resort to traditional lectures, pencil-and-paper methods, and static slides — approaches that fail to provide the interactive engagement shown to improve student comprehension [23].

The scarcity of visualization tools in CS classrooms is not simply a matter of awareness — it reflects a deeper structural problem. Developing and maintaining high-quality interactive visualizations requires significant time, developer expertise, and ongoing effort that most educators cannot reasonably sustain [31]. The computing education community has repeatedly identified these barriers as a root cause of limited tool adoption, and has called for shared repositories and common design practices to mitigate them [2]. Existing tools tend to be narrowly scoped to a single topic or problem type (sorting visualizers, graph traversal tools, finite automata simulators, step-through interpreters for a specific language subset etc.) making them difficult to adapt, extend, or reuse across different courses and curricula [27]. As a result, the burden of creating such tools falls on a small number of developers, limiting both the breadth of topics covered and the longevity of the tools themselves.

What is needed is not simply another visualization tool, but a platform — one that is broad enough to accommodate a wide range of CS topics, extensible enough that students and educators can contribute new content with minimal effort, and adaptable enough to

evolve alongside the curriculum it supports [2, 8]. This thesis presents Redux, an open-source web application developed at Idaho State University, as such a platform [21].

Prior development on Redux included support for visualizing problems and their corresponding solutions, with each problem associated with a single visualization. This design provided a clear and structured way to represent problem–solution relationships, but did not yet explore how multiple complementary visualizations could be used to represent more complex or multi-faceted computer science concepts. In addition, earlier implementations of Redux focused primarily on problems related to NP-completeness. This focus enabled in-depth exploration within a specific class of computational problems, but did not yet extend to the broader range of topics in computer science, such as data structures, general algorithms, and logical systems. Expanding beyond this scope enables the system to support a wider variety of educational and exploratory use cases.

While these prior design choices established a strong foundation, they leave open questions regarding how to support multiple complementary visualizations within a single problem and how to extend the system beyond a narrow class of computational topics. Addressing these considerations requires a more general and flexible approach to interactive visualization design. Accordingly, this thesis is guided by the following research questions:

RQ1: How can a single extensible web-based platform be designed to support diverse visualization types across multiple computer science domains?

RQ2: What strategies can be employed to reduce the effort required to add new problem visualizations to an interactive visualization platform?

RQ3: What features contribute to effective pedagogical visualization tools for computer science education?

This work demonstrates that an extensible, web-based visualization platform can effectively support diverse computer science domains by providing reusable templates, a standardized API, and low-barrier contribution workflows—while incorporating pedagogically

effective features such as step-by-step solution tracing and transitive gadget highlighting. We present generalized transitive gadget highlighting that handles logic across chained reductions, as well as a step-by-step solution tracing feature that allows users to follow algorithm execution incrementally. We further contribute a suite of reusable visualization templates for sets, graphs, and Boolean satisfiability problem types, alongside a standardized API that automatically invokes the appropriate templates and parses problem data. Together, these contributions lower the barrier to contribution such that most students can add new visualizations with minimal effort, enabling Redux to grow as a crowd-sourced knowledge base spanning a broad range of computer science topics [1].

The remainder of this thesis is organized as follows. Chapter 2 surveys related work in CS education visualization, existing algorithm visualization tools, and prior Redux development. Chapter 3 describes the overall system architecture of Redux, including the structure of the front-end visualization framework and the backend API. Furthermore, it presents the specific contributions of this thesis in detail, covering each new feature and template. Chapter 4 evaluates these contributions and discusses their implications for CS education. Chapter 5 concludes the thesis with a summary of findings, limitations, and directions for future work.

Chapter 2

Related Work

Interactive algorithm and data structure visualization tools have been an active area of research in CS education for over two decades [27]. This chapter surveys the most relevant work in four areas: CS education and visualization research, existing algorithm visualization tools, web-based education platforms, and prior Redux development, highlighting in each case how existing work informed Redux's design and where gaps remain that this thesis addresses.

2.1 CS Education and Visualization Research

The educational value of interactive visualizations in CS courses has been well-established in the literature [23]. Research has shown that learning gains increase as the level of student engagement with a visualization increases — with students who actively construct and discuss their own representations of algorithms showing the greatest improvement [14]. Despite this, adoption of visualization tools among educators remains low [15], with research consistently pointing to barriers such as the time required to learn new tools, the expertise needed to develop them, and the difficulty of finding tools that fit a given course's needs [15, 31].

Existing visualization tools tend to be narrowly scoped and difficult to reuse across courses and curricula [26]. Most tools are designed with a specific topic or course in mind, meaning that educators must seek out, learn, and maintain a different tool for each subject they wish to visualize — a burden that contributes directly to the low adoption rates observed in practice [31]. Furthermore, the development of such tools is typically driven by individual researchers or small teams, resulting in tools that are well-suited to their original purpose but

rarely designed with extensibility or longevity in mind [1]. The broader computing education community has identified this pattern of isolated, duplicated effort as a systemic problem and has called for shared community software infrastructure to address it [2]. As CS curricula continue to evolve and expand, the need for a visualization platform that can grow alongside them — one that is broad in scope, easy to contribute to, and sustainable over time — becomes increasingly apparent.

2.2 Existing Algorithm Visualization Tools

Algorithm visualization tools have become an increasingly important part of CS education, offering students interactive and visual ways to engage with otherwise abstract and inaccessible concepts [14]. As the breadth of CS curricula has expanded, so too has the demand for tools that can meaningfully support instruction across a diverse range of topics — from foundational data structures to advanced theoretical subjects such as NP-completeness and quantum computing. Numerous such tools have been developed to address specific areas of the curriculum, each reflecting distinct design goals, pedagogical philosophies, and intended audiences. This section surveys the most relevant existing tools, examining their strengths, their limitations, and the gaps that collectively motivated the development of Redux as a more general-purpose and extensible alternative.

JFLAP[1] is one of the most widely adopted tools in CS education, providing interactive visualizations for formal languages and automata theory — including DFAs, NFAs, pushdown automata, and Turing machines [25]. Students using JFLAP reported greater engagement and improved understanding of course concepts across a two-year study at fourteen universities. However, JFLAP is scoped exclusively to automata and formal languages, has been in a stable, largely unchanged state since its 2018 release, and is not designed to serve as a general-purpose platform for broader CS topics [1]. As shown in Figure 2.1, JFLAP operates

[1] `https://jflap.org`

as a desktop application rather than a web-based tool, requiring installation and limiting accessibility compared to browser-based alternatives.

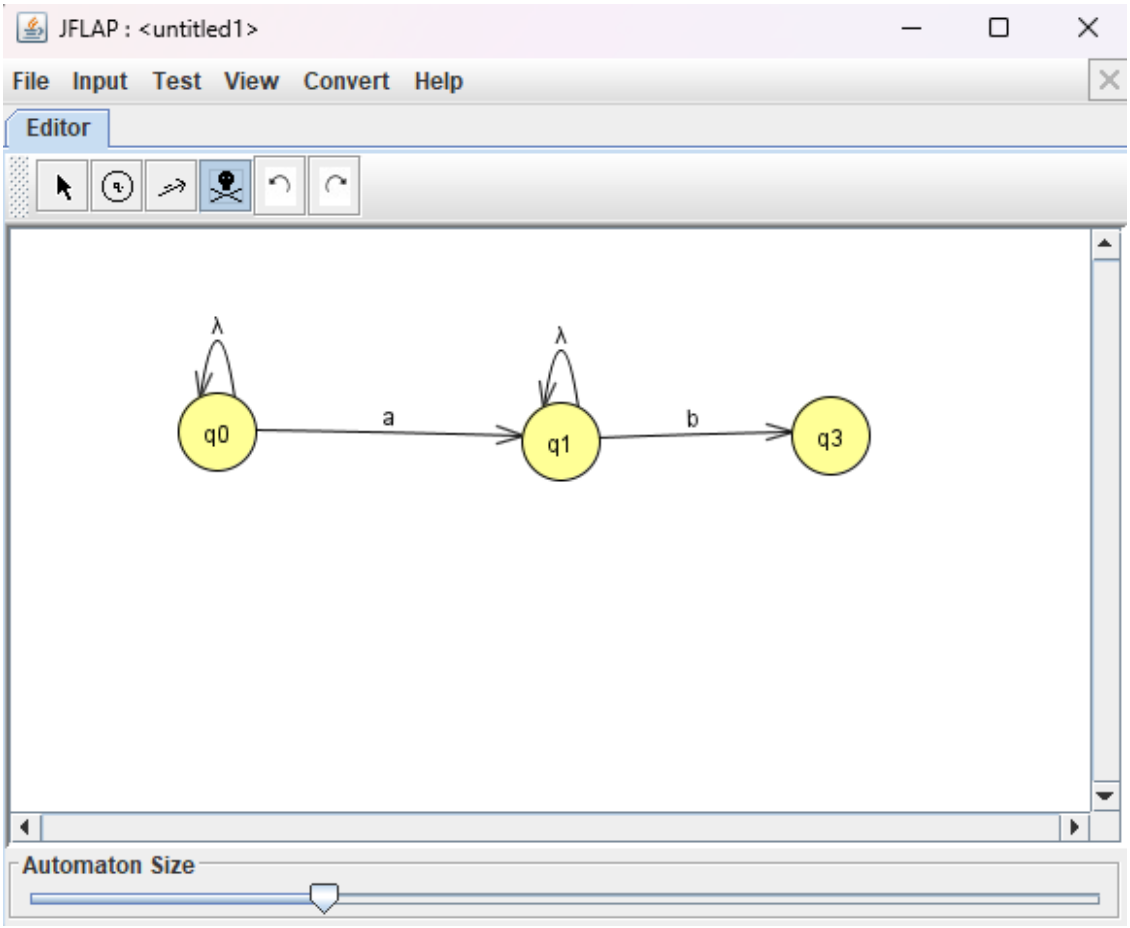


Figure 2.1: **JFLAP desktop interface.** The screenshot shows a user-constructed DFA with three states in JFLAP's desktop interface, which supports creating and modifying automata as well as simulating their execution step by step.

VisuAlgo[2] provides animations of a wide range of data structures and algorithms and has been used by millions of students worldwide [12]. One of the main strengths of VisuAlgo is that it covers an impressive breadth of topics, including sorting, graph algorithms, and dynamic programming, making it one of the more comprehensive algorithm visualization tools available (see Figure 2.2). VisuAlgo is centrally maintained by a small core team of two project advisors, a Software Engineer at Google and an associate professor at National University of Singapore, alongside students rather than being open to external contribution

[2] `https://visualgo.net`

from students or educators, which naturally ties the tool's growth and topic coverage to the capacity of its core team [11]. Additionally, VisuAlgo focuses on classical algorithms and data structures, and does not cover theoretical CS topics such as NP-completeness or logic-based problem solving.

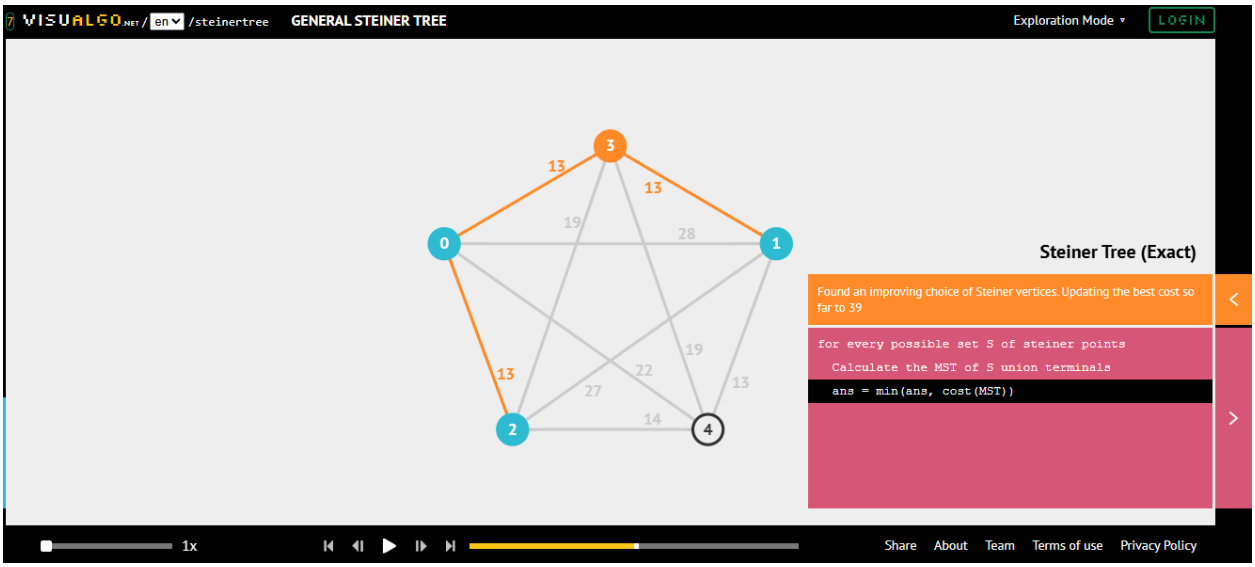


Figure 2.2: **VisuAlgo Steiner Tree visualization.** The visualization shows a Steiner tree on a weighted undirected graph with 5 vertices [13]. The platform is interactive, allowing users to modify the graph by adding vertices and edges directly. A step-by-step playback bar at the bottom of the screen enables users to scrub through algorithm execution at adjustable speeds, while a side panel displays the corresponding pseudocode with the current line highlighted.

AlViE[3] represents one of the few prior efforts to apply algorithm visualization specifically to NP-completeness education [4, 6]. By incorporating visualizations of four well-known NP-completeness reductions, AlViE demonstrated that interactive tools can meaningfully improve student understanding of this notoriously challenging topic, yielding positive preliminary student evaluations (see Figure 2.3). AlViE focused on a fixed set of four problems, suited to its goals as a proof of concept, and does not extend to arbitrary problem instances or reductions reachable via transitivity. Furthermore, AlViE is no longer actively maintained. Redux builds directly on AlViE's contribution, extending the approach to support problem instances and the broader set of reductions reachable via transitivity.

[3] `https://github.com/piluc/AlViE`

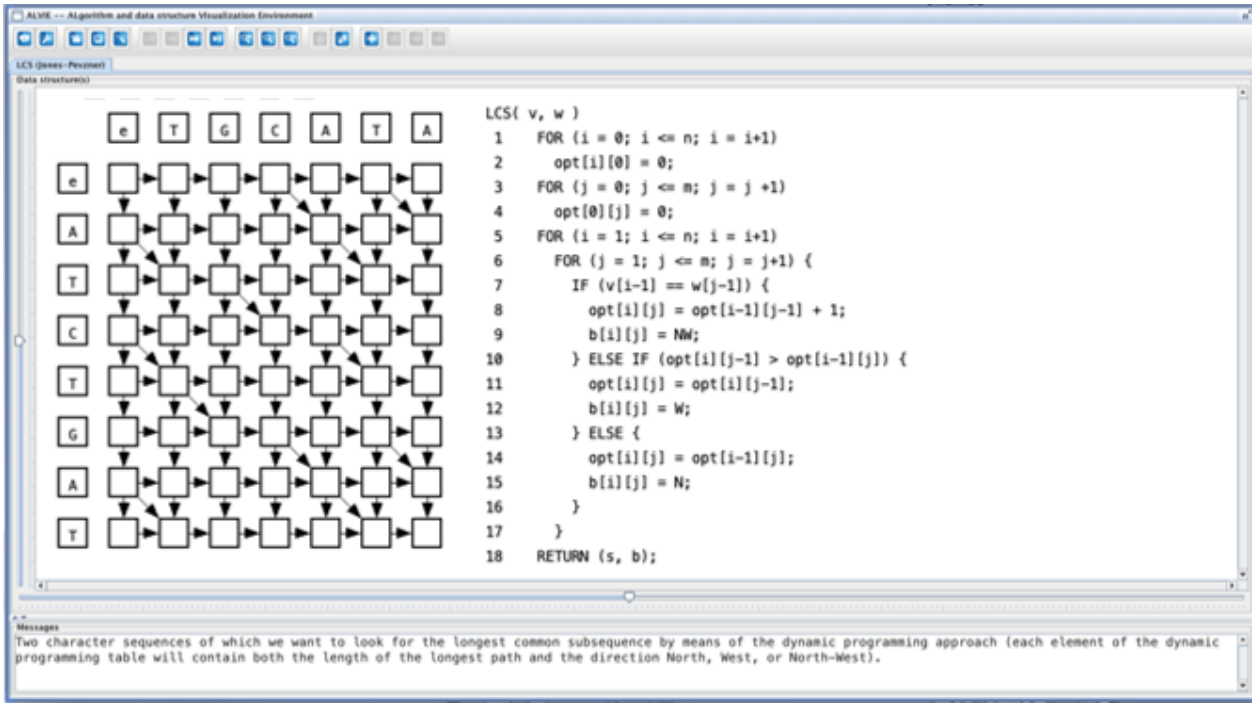


Figure 2.3: **AlViE desktop interface.** The figure is visualizing the Longest Common Subsequence algorithm, with a dynamic programming table rendered as a directional arrow grid, accompanied by pseudocode and a plain-language description panel. [5].

Python Tutor[4] is a code visualization tool that has reached over ten million users across more than 180 countries, making it one of the most widely used pieces of research software developed within a university lab [10]. Research on its design offers directly relevant lessons for Redux in terms of good software design: sustainable CS education software must prioritize user experience, long-term architectural decisions, and workflows that enable broad contribution. These principles directly informed Redux's shift toward a standardized API and reusable template system designed to lower the barrier to contribution. Python Tutor is intentionally focused on code execution visualization, which naturally leaves room for tools like Redux to address more abstract theoretical CS concepts. Figure 2.4 illustrates how Python Tutor makes program execution explicit, highlighting design principles that influenced Redux's approach to visualized data flow and reusable workflows.

[4] `https://pythontutor.com`

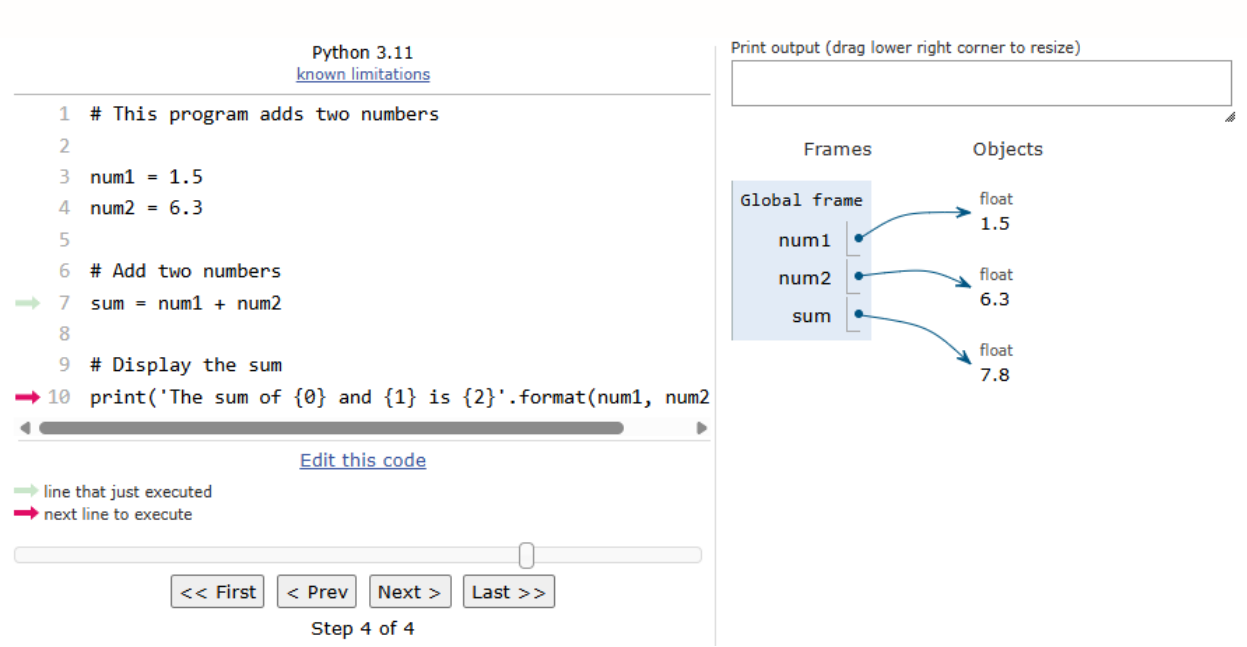


Figure 2.4: **Python Tutor code example.** The screenshot from Python Tutor shows a simple program that adds two numbers. The left panel displays the code with the current execution step highlighted, while the right panel visualizes the program's memory state, including variable names and values. This illustrates Python Tutor's approach to making program execution explicit, a design philosophy that inspired Redux's emphasis on clear, reusable workflows and visualized data flow.

2.3 Web-Based Education Platforms

Web-based platforms that combine visualizations with interactive exercises and automated assessment have demonstrated strong educational outcomes in CS courses (such as **VisuAlgo**) [8]. **OpenDSA**[5] is an open-source interactive eTextbook platform developed at Virginia Tech, designed to provide students with a complete, self-contained learning environment for data structures and algorithms [16]. Rather than functioning as a standalone visualization tool, OpenDSA integrates algorithm visualizations directly into course modules alongside written explanations, exercises, and automated assessment — creating a cohesive instructional experience that goes beyond what a visualization tool alone can offer. Its modular, community-extensible architecture allows educators to assemble custom course

[5] `https://opendsa-server.cs.vt.edu`

materials from a shared library of components, and the platform has been adopted across numerous universities as a primary course resource. More recent work has extended this approach to game-based contexts, combining algorithm visualization with interactive game mechanics to improve motivation and comprehension in data structures courses [29]. While user-defined problem instances are not a central feature of OpenDSA, select visualizations support arbitrary input. A fixed input interactive visualization implemented in OpenDSA can be seen in Figure 2.5. Redux shares this commitment to extensibility but differs in important ways: whereas OpenDSA is structured as an eTextbook scoped to data structures and algorithms, Redux is designed as a general-purpose knowledge base spanning a broad range of CS topics, with a lightweight contribution model that requires minimal developer expertise.

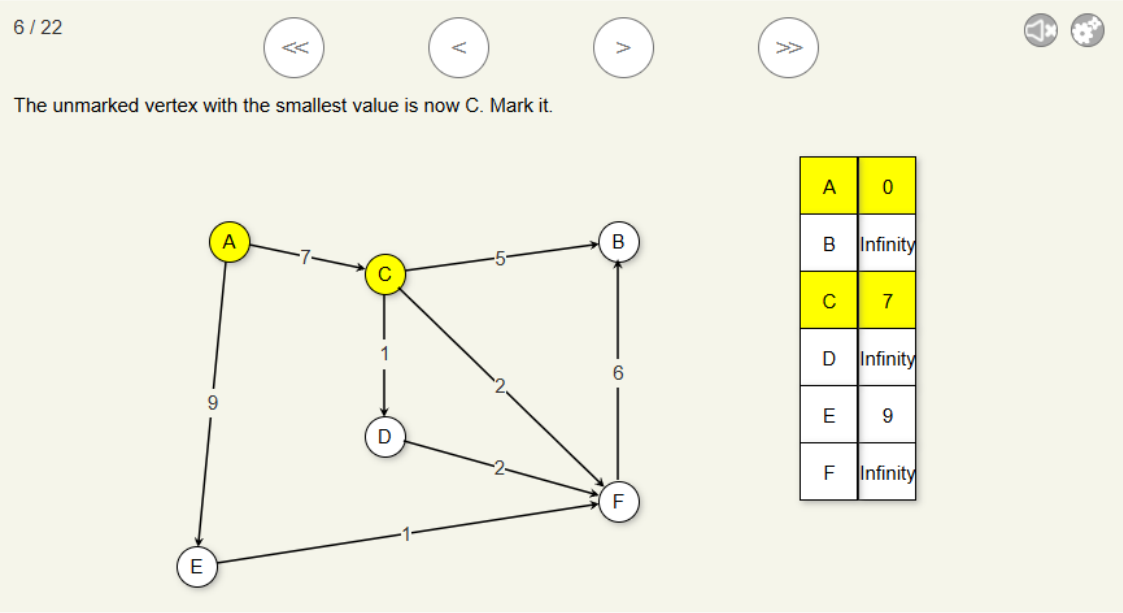


Figure 2.5: **OpenDSA Shortest Path.** A screenshot taken from OpenDSA's lecture 18.5 on Shortest Path algorithms shows step 6 of 22 of a slideshow visualization of Dijkstra's shortest path algorithm. Visited vertices and their finalized distances are highlighted in yellow, while the distance table on the right tracks the current shortest known path to each vertex.

Tool	User input	Web-based	Extensible	Maintained	Multi-viz
VisuAlgo	×	×		×	∼
JFLAP	×		∼	∼	
AlViE	×		×		∼
PythonTutor	×	×		×	
OpenDSA	∼	×	×	×	∼

Table 2.1: Comparison of algorithm visualization tools features

2.4 Prior Redux Development

Redux was originally developed at Idaho State University as an open-source web application providing a dynamic, interactive interface for NP-completeness education [21]. The tool allows users to visualize problem instances, mapping reductions, solutions, and gadgets — including those reachable via transitivity — atop a growing knowledge base of NP-complete problems. Early work laid the conceptual groundwork for this kind of visual, reduction-centric pedagogy; Marchetti and Bodily first demonstrated the viability of the approach by visualizing the 3SAT-to-CLIQUE reduction process, showing that reduction walkthroughs could make abstract theoretical concepts more accessible to students [20]. Surveys of students using Redux confirmed this direction, indicating that the tool helped them better understand mapping reductions, that they preferred it over manual methods, and that it made learning more enjoyable [21].

Alongside Redux, parallel work explored crowd-sourcing as a mechanism for tackling complex, real-world problems at scale. The KAMI system investigated how distributed human contributors could be organized to solve problems that resist purely automated solutions [19]. This crowd-sourcing philosophy directly informs Redux's design: rather than relying on a static, centrally-maintained knowledge base, the system is architected to grow through community contribution, allowing new problems, reductions, and gadgets to be added collaboratively over time.

Underpinning Redux's functionality is a suite of supporting tools developed alongside it. The SPADE library provides programmatic parsing and verification of discrete data structures,

enabling Redux to rigorously validate problem instances and reductions contributed by users [24]. Together, these contributions form an ecosystem of tools that support Redux's broader mission: making abstract CS concepts not just comprehensible, but interactively explorable.

However, the original Redux architecture presented several limitations that constrained its broader applicability — and understanding why these limitations matter is essential to appreciating the scope of what this thesis addresses. Interactive visualization tools have long been recognized as valuable in computer science education, yet building and maintaining them has historically required significant developer expertise, meaning most topics across the CS curriculum go without dedicated visual support. Redux was developed to fill this gap for NP-completeness, but its architecture inadvertently reproduced the same problem it set out to solve: a tool that is difficult to extend is a tool that cannot grow. The first concrete limitation was that Redux supported only a single visualization type per problem. In a domain as rich as NP-completeness — and indeed across computer science broadly — a single visual representation forces a single pedagogical perspective. Students who struggle to grasp a concept from one angle have no alternative view to fall back on. Figure 2.6 illustrates this constraint, showing the original Redux interface in which only a single formula view and graph are available to the student regardless of the problem being studied. The second limitation was that transitive gadget highlighting was incomplete across certain chained reductions. Transitivity is not a peripheral feature of NP-completeness; it is the very mechanism by which the theory holds together, and a tool that cannot faithfully trace a full transitive chain risks actively misleading students about how the theory works. The third and most consequential limitation was that contributing new content required significant developer expertise. This is not merely an inconvenience — it is an existential threat to the long-term viability of any crowd-sourced educational platform. A knowledge base that can only grow when a skilled developer finds time to extend it will stagnate, and in an academic environment where students cycle through and faculty attention is divided, sustainability depends entirely on lowering that barrier.

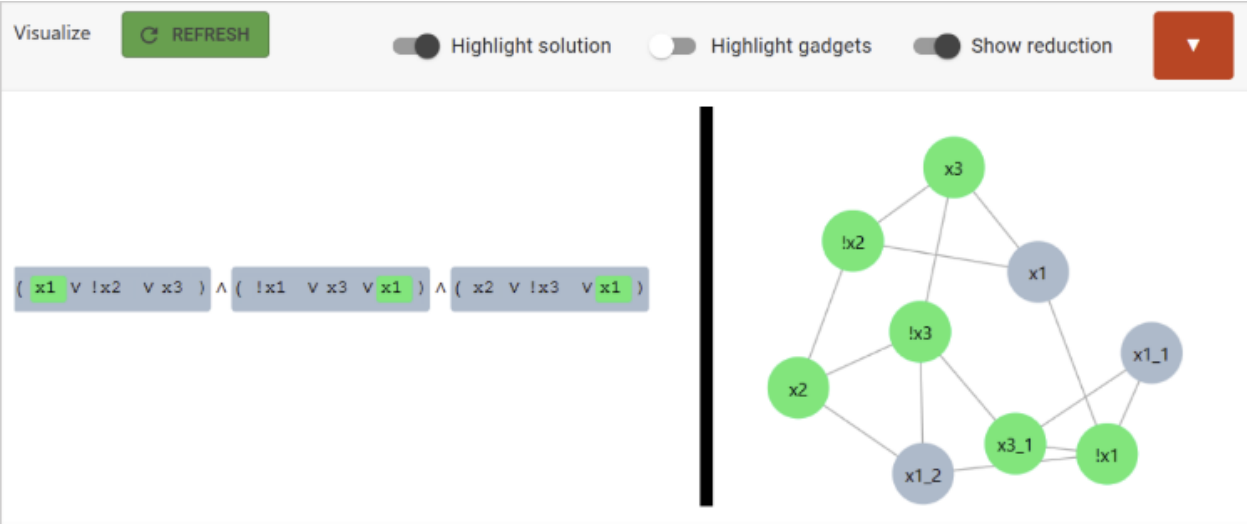


Figure 2.6: **Original Redux visualization interface.** Previous versions of Redux were restricted to a single visualization type per problem. Here the 3SAT formula (left) and its corresponding CLIQUE graph (right) are the only views available, with no mechanism for alternative representations, taken from [21].

Taken together, these limitations meant that Redux could not fulfill its deeper potential: not just a tool for NP-completeness, but a living, extensible platform for interactive computer science education at large. The philosophical groundwork for this vision was laid in part by KAMI [19], which demonstrated the power of crowd-sourcing as a mechanism for tackling problems too large and complex for any single contributor to solve alone. That same philosophy motivates Redux's evolution — a knowledge base that grows through community contribution, spanning the breadth of computer science rather than a single topic. Realizing this vision, however, required more than a philosophical commitment; it required the technical infrastructure to support it. This thesis provides that infrastructure directly: a standardized backend API, support for multiple visualization types per problem, automated transitive gadget highlighting, step-by-step algorithm execution, and a suite of reusable visualization templates that together lower the barrier to contribution such that most students can add new visualizations with minimal effort. Supporting work such as Spade [24], which provides programmatic parsing and verification of discrete data structures, addresses exactly this need by ensuring that crowd-sourced contributions can be validated rigorously without

demanding expert oversight. This thesis builds on these foundations, transforming Redux into a general-purpose platform that most students can contribute to with minimal effort — a crowd-sourced knowledge base built to span the breadth of computer science, and built to last.

Chapter 3

Contributions

This chapter describes the overall structure of the Redux system as extended by this thesis. Section 3.1 provides a high-level overview of the client-server architecture. Section 3.2 describes the organization of the knowledge base. Section 3.3 details the backend REST API and the standardized naming schema introduced by this thesis. Section 3.4 describes the frontend visualization framework and the reusable template system. Section 3.5 summarizes the end-to-end interaction flow through the system.

3.1 Overview

Redux is a web application built on a client-server architecture, where a React-based frontend communicates with a C# backend via a REST API, providing easy and effective data pipelines [7, 22]. The backend serves as the knowledge base, housing all problems, algorithms, visualizations, and reductions, while the frontend is responsible for rendering visualizations and managing user interaction. Redux is accessible online via `redux.portneuf.cose.isu.edu`, with the frontend[1] and backend[2] available as open-source repositories.

The original Redux system established the foundational architecture of a web-based knowledge base for NP-complete problems, providing a dynamic interface for visualizing problem instances, reductions, and solutions [21]. This thesis extends that foundation by restructuring the backend API with a standardized naming schema [17], introducing a reusable frontend template system, and redesigning the visualization pipeline to support multiple

[1]Redux GUI: `https://github.com/ReduxISU/Redux_GUI`
[2]Redux API: `https://github.com/ReduxISU/Redux`

visualization types per problem. These changes preserve the core architecture of Redux while substantially improving its extensibility and ease of contribution.

The Redux knowledge base is organized as a file system in which each problem occupies its own dedicated folder. This structure is designed to be self-contained and modular, allowing new contributions to be easily implemented independently [2, 30]. Each problem folder contains a central problem class alongside four subfolders:

- **Solvers** — contains algorithm files that solve the problem, each implementing a specific solution strategy
- **Verifiers** — contains files that verify whether a given solution is correct
- **Reductions** — contains reduction files that map one problem to another, alongside gadget maps used for transitive gadget highlighting
- **Visualizations** — contains visualization files that define how a problem instance and its solution should be displayed

Such a folder structure makes it practical to view and organize information in our knowledge base. Figure 3.1 shows an example of this structure for the NPC_CLIQUE problem.

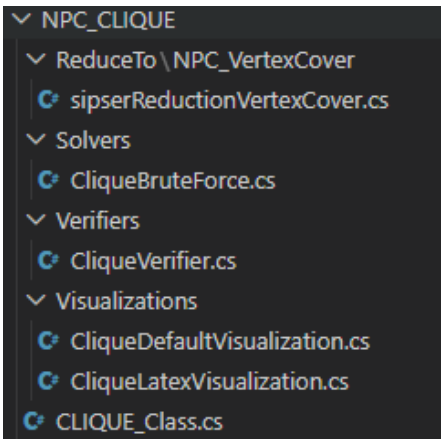


Figure 3.1: **Example problem folder structure.** The folder structure of the NPC_CLIQUE problem in Redux, containing a central problem class alongside dedicated subfolders for solvers, verifiers, and visualizations. Adding a new contribution only requires creating a new file inside its respective folder.

This modular folder structure means that contributors can add new solvers, reductions, or visualizations to an existing problem without touching any other part of the system. At the center of each problem folder is a main problem class file that serves as the authoritative definition of the problem — specifying how instances are parsed, providing definitions and links to reference sources, and declaring the default solvers, verifiers, and visualizations associated with that problem. Similarly, an entirely new problem can be introduced by creating a new folder that follows the same structure, making it straightforward for new contributors to extend the system without touching unrelated parts, and increasing the longevity of the open-source tool [11]. To further lower the barrier to contribution, Redux provides a template download system through its web interface: for any file type — problem class, visualization, verifier, reduction, or solver — a contributor can download a pre-populated template with the class name and required method signatures already filled in, leaving only the implementation to be written. This means a student with no prior knowledge of the Redux codebase can begin contributing a new problem or visualization within minutes of visiting the site.

3.2 API Restructuring

The backend is implemented in C# and exposes a REST API — a standardized communication pattern in which the frontend sends HTTP requests to specific endpoints and receives structured data in return [17]. The React frontend queries to retrieve problem data, solutions, and visualization instructions. For example, to retrieve the solvers available for a given problem, the frontend might call `/api/NPC_EXACTCOVER/getSolvers`, which returns a list of available solver names. A key design contribution of this thesis is the restructuring of this API around a consistent naming schema that standardizes how data is requested and returned across all problems, following established design patterns for extensible APIs [7].

Each visualization file on the backend contains an API constructor, which takes a problem instance expressed in mathematical notation — parsed and verified using Spade [24] — and transforms it into a structured, frontend-friendly JSON object. Prior to this thesis,

the API used generic placeholder fields such as `attribute1`, `attribute2`, and `attribute3` whose meaning varied silently from problem to problem, requiring implicit knowledge to interpret correctly and making the frontend brittle and difficult to extend. As shown in Figure 3.2, the standardized schema replaces these ambiguous fields with semantically named, self-documenting equivalents — such as `color`, `weight`, and `directed` — that are consistent across all problems. This standardization effort, led by Michael Crapse for solvers, verifiers, reductions, and problem classes, and extended by Russell Phillips and the author for visualization API calls, eliminated over 10,000 lines of redundant and inconsistent code across the codebase.

```
Before:
{
  "nodes": [
    { "name": "1", "id": "1",
      "attribute1": "red", "attribute2": "green", "attribute3": "" }
  ],
  "links": [
    { "source": "1", "target": "2",
      "attribute1": "green", "attribute2": "directed" }
  ]
}

After:
{
  "nodes": [
    { "name": "1", "id": "1", "color": "green",
      "outline": "red", "delay": "", "dashed": "", "additional": "" }
  ],
  "links": [
    { "source": "1", "target": "2", "color": "green", "dashed": "",
      "delay": "", "weight": "1", "weighted": false, "directed": true }
  ]
}
```

Figure 3.2: **Graph API response.** Example API response structure before and after standardization. The original schema used generic `attribute1`, `attribute2` fields whose meaning varied by the selected problem, requiring implicit knowledge to interpret. The standardized schema replaces these with semantically named fields — `color`, `weight`, `directed` — that are self-documenting and consistent across all graph problems.

As seen in Figure 3.2, in the original schema a graph node carries only `attribute1` and `attribute2` with no indication of what those fields represent. In the standardized schema, the same node carries explicitly named fields — `color`, `outline`, `weight`, `directed` — whose purpose is immediately clear to any contributor without needing to consult additional documentation.

```
[
  { "nodes": [ {"name":"1","color":"Background","id":"1"}, ... ],
    "links": [ {"source":"4","target":"1","color":"Background","directed":false}, ... ]
  },  // index 0: initial state

  ...,  // intermediate steps

  { "nodes": [ {"name":"1","color":"Background","id":"1"},
               {"name":"2","color":"Solution",  "id":"2"}, ... ],
    "links": [ {"source":"4","target":"1","color":"Background","directed":false},
               {"source":"4","target":"3","color":"Solution",  "directed":false}, ... ]
  }   // index n: solution state
]
```

Figure 3.3: **Step-by-step API information.** A real API response for a graph visualization problem. Each element of the array represents one step of the algorithm's execution: index 0 is the initial unsolved state, the final index is the fully colored solution state, and intermediate indices capture each incremental transition. The two objects shown are taken directly from an Exact Cover problem instance, where nodes and edges are progressively colored to distinguish solution edges (`"Solution"`) from background edges (`"Background"`).

The API constructor assembles the full step-by-step solution trace, packaging each incremental step of the algorithm's execution as a structured JSON object and collecting all steps into an ordered array. As shown in Figure 3.3, index 0 holds the initial state, the final index holds the solution state, and all intermediate states occupy the indices between. The entire array is returned to the frontend in a single API response, eliminating the need for

repeated polling or sequential requests. Prior to this refactor, the frontend issued a separate API request each time the user navigated to a different step — including switching between the initial and solution states — resulting in redundant network calls for data that was already known at solve time. This batching approach follows best software practice and has been shown to lower latency and server processing overhead compared to many fine-grained calls, improving performance and scalability under concurrent request loads [18].

3.3 Frontend Visualization Framework

The frontend is implemented in React and organized around a central visualization component, `VisualizeRowMemo`, which manages the entire visualization section of the page. As illustrated in Figure 3.4, when a user selects a problem and chooses a visualization type, `VisualizeRowMemo` issues a single HTTP request to the backend REST API, which returns the complete step array in one response. Each element of this array represents one state of the algorithm's execution — index 0 holds the initial unsolved state, the final index holds the fully colored solution state, and all intermediate indices capture each incremental transition between them. This design eliminates the repeated per-step network requests that characterized the prior implementation, in which the frontend would issue a new API call each time the user navigated to a different step — including switching between the initial and solution states — resulting in redundant round trips for data that was already known at solve time. By batching the entire trace into a single response, the system reduces latency and server processing overhead, improving both responsiveness and scalability under concurrent request loads [18]. Once the response is received, `VisualizeRowMemo` parses the structured JSON object and extracts the requested step by index, passing it forward into the rendering pipeline.

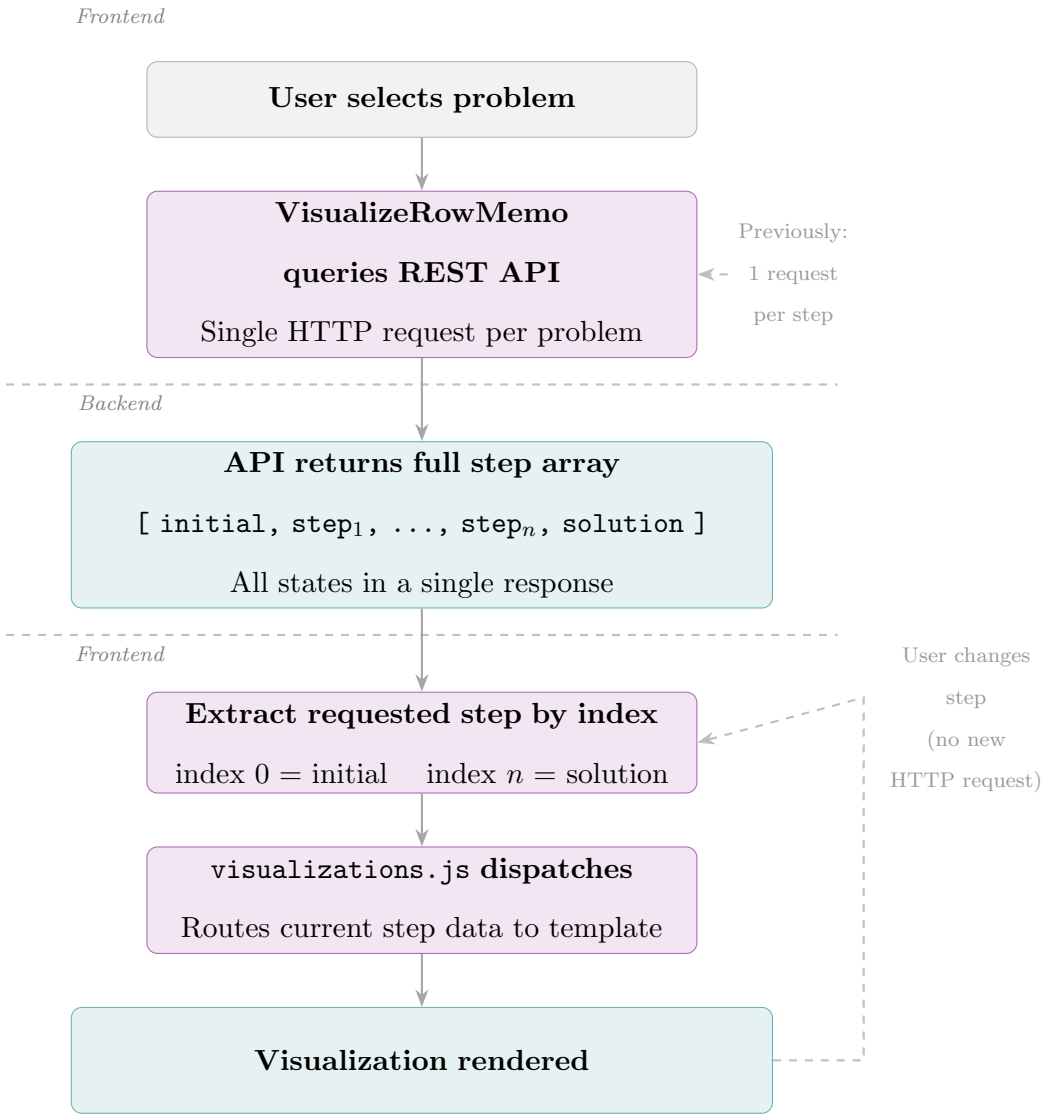


Figure 3.4: **Visualization rendering pipeline.** Frontend–backend interaction for visualization requests requires a single API call which returns the full step array; navigating between steps requires no further network requests.

The rendering pipeline routes visualization requests through `visualizations.js`, which reads the visualization name specified in the backend visualization file and dispatches the request to the appropriate template. This dispatch mechanism is designed for extensibility — adding a new visualization type requires only implementing the template component and registering its name in `visualizations.js`. Together, the backend visualization files and the frontend rendering pipeline form a clean separation of concerns: the backend defines

what data to present and *how* it is structured, while the frontend determines *how* that data is rendered visually. This decoupling ensures that changes to one layer do not necessitate modifications to the other, making the system straightforward to maintain and extend. This thesis introduces four reusable frontend templates that cover the most common visualization types encountered in CS problems:

- **D3 Sets** — renders set-based problem instances and solutions using D3.js, suitable for problems whose primary structure involves collections of elements
- **D3 Graphs** — renders graph-based problem instances and solutions using D3.js, supporting directed and undirected graphs with weighted edges and configurable node styling
- **LaTeX Graphs** — renders graph-based visualizations using TikZJax, providing publication-quality typeset output for problems where precise mathematical notation is important
- **SAT Templates** — renders Boolean satisfiability problem types, supporting clause and literal representations across a range of SAT variants

Each template adheres to the standardized field naming schema established by the backend API, ensuring that data flows predictably from the knowledge base through the API to the rendered visualization [17]. This consistency is what allows new templates to be added with minimal effort — contributors need only follow the established naming conventions rather than reverse-engineering the data pipeline.

3.4 Multiple Visualization Types Per Problem

The original Redux system supported only a single visualization type per problem. This was a significant limitation, as many CS problems have multiple natural visual representations that serve different pedagogical purposes [23]. For example, a graph problem might be best

understood through a standard node-link diagram in one context and a matrix representation in another.

New contributions introduce support for **multiple visualization types** per problem. Each problem folder on the backend can now contain multiple visualization files, each defining a distinct visual representation of the same problem. The user is presented with a dropdown menu on the frontend, as shown in Figure 3.5, allowing them to select their preferred visualization type, and the system dynamically loads the appropriate template and API constructor for the selected type.

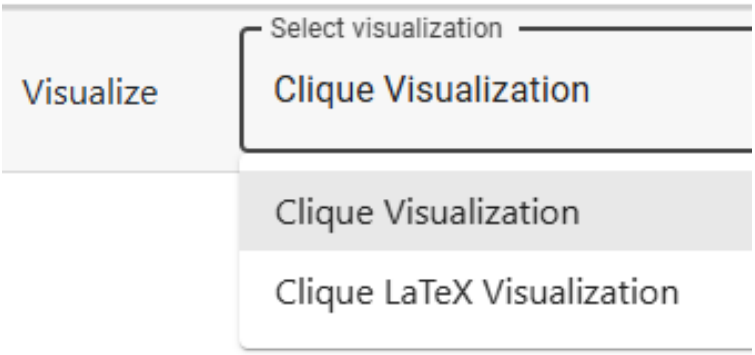


Figure 3.5: **Visualization dropdown menu.** The dropdown menu allowing users to select their preferred visualization type for a given problem, enabling the same underlying data to be rendered in multiple formats.

Such an approach required changes to both the backend, where the problem folder structure was extended to support multiple visualization files, and the frontend, where the visualization selection interface and dispatch mechanism were updated accordingly. The result is a more flexible and expressive system that can accommodate the full range of visual representations a problem may require. This flexibility matters because the most effective visualization of a problem is not always self-evident, and different representations can serve fundamentally different pedagogical goals. A node-link diagram may make the structure of a graph problem immediately intuitive, while an adjacency matrix representation better exposes properties such as symmetry or sparsity that are critical to understanding certain algorithms. More strikingly, when a classical problem is approached through an

unconventional algorithm, the traditional visualization may become actively misleading. Consider SAT: classically represented as a boolean formula over variables and clauses, its standard visualization reflects propositional logic. When solved using Grover's algorithm, however, the meaningful unit of analysis shifts to quantum states and circuit operations, making a quantum circuit visualization far more appropriate and informative than its classical counterpart. Beyond such algorithm-driven mismatches, supporting multiple visualization types creates an open-market dynamic for visual representations, inviting contributors to compete on the quality and clarity of their designs. This is particularly valuable in the context of Redux, which supports not only a wide variety of problem domains but also fundamentally different categories of visualization, including step-by-step animations, gadget highlighting, and solution highlighting. The breadth of these possibilities means that for virtually any problem in the system, there is room for a contributor to devise a representation that is clearer, more intuitive, or more revealing than what currently exists, turning visualization itself into an active and ongoing area of contribution rather than a one-time implementation detail.

3.5 Transitive Gadget Highlighting

Mapping reductions are a cornerstone of NP-completeness theory [28]. A mapping reduction from problem Π_1 to problem Π_2, written $\Pi_1 \leq_p \Pi_2$, is a polynomial-time computable function f such that $x \in \Pi_1$ if and only if $f(x) \in \Pi_2$. This machinery underlies NP-completeness proofs: if $\Pi_1 \leq_p \Pi_2$ and Π_2 is NP-complete, then Π_1 is NP-complete as well. Crucially, reductions compose transitively — if $\Pi_1 \leq_p \Pi_2$ and $\Pi_2 \leq_p \Pi_3$, then $\Pi_1 \leq_p \Pi_3$ — meaning that NP-hardness propagates along arbitrarily long reduction chains. This transitivity is central to how NP-completeness proofs are structured in practice, yet the correspondence between problem elements across such chains is rarely made explicit. Redux was conceived precisely to address this gap, providing interactive visualizations that allow students to trace

how elements of one problem map to their counterparts through each step of a reduction chain.

Gadget highlighting is one of Redux's most distinctive features, allowing users to visually trace the correspondence between elements of one problem and their counterparts in a reduced problem. In the original Redux system, transitive gadget highlighting — the ability to highlight gadgets across chains of reductions — was incomplete, failing to properly handle cases that arose when reductions were composed transitively.

Since then, contributions have corrected and generalized the transitive gadget highlighting system to handle all edge cases across arbitrarily long chains of NP-complete reductions. The gadget map, stored in the reductions folder of each problem, defines the correspondence between elements in the source and target problems. The updated highlighting system traverses these gadget maps transitively, correctly propagating highlights across all intermediate reductions in a chain regardless of their length or structure.

Let $\mathcal{R} = \{r_1, r_2, \ldots, r_k\}$ be a sequence of reductions where each r_i maps instances of problem Π_i to instances of Π_{i+1}. For each r_i, we define a gadget mapping correspondence $\varphi_i : G_i \to \mathcal{P}(G_{i+1})$, where G_i is the set of gadgets at step i and $\mathcal{P}(\cdot)$ denotes the powerset, accommodating one-to-one, one-to-many, and many-to-many gadget mappings.

We interpret each φ_i as a finitely-supported set-valued mapping (equivalently, a relation $\varphi_i \subseteq G_i \times G_{i+1}$), where each gadget $g \in G_i$ is associated with a finite set of successor gadgets in G_{i+1}. We require that $\varphi_i(g) \neq \emptyset$ for all relevant g, ensuring that every highlighted gadget propagates forward through the reduction chain. We further extend φ_i to subsets $S \subseteq G_i$ by lifting it to the powerset level via union:

$$\varphi_i(S) := \bigcup_{g \in S} \varphi_i(g),$$

which makes φ_i a union-preserving operator on $\mathcal{P}(G_i)$. In particular, this implies the homomorphism property

$$\varphi_i(S_1 \cup S_2) = \varphi_i(S_1) \cup \varphi_i(S_2),$$

ensuring that simultaneous or collective highlights decompose consistently into gadget-wise propagation.

The transitive highlight map is then the composition

$$\Phi = \varphi_k \circ \cdots \circ \varphi_1, \qquad \Phi : G_1 \to \mathcal{P}(G_{k+1}),$$

where composition is understood in the lifted (powerset) sense. Concretely, each φ_i acts not on individual gadgets alone but on subsets via its union extension, and composition corresponds to repeated relational propagation across layers of reductions. That is, for $\varphi_{i+1} \circ \varphi_i$ we define

$$(\varphi_{i+1} \circ \varphi_i)(g) := \varphi_{i+1}(\varphi_i(g)) = \bigcup_{g' \in \varphi_i(g)} \varphi_{i+1}(g'),$$

for all $g \in G_i$. This construction ensures that the composition corresponds to a forward reachability operator over the induced bipartite gadget correspondence graph between successive reduction levels.

Equivalently, the lifted composition can be understood as a recursive expansion of reachable gadget sets. Defining $\Phi_1 := \varphi_1$, we obtain

$$\Phi_{i+1}(g) := \varphi_{i+1}(\Phi_i(g)),$$

which iteratively accumulates all descendants of g across reduction layers. Hence $\Phi = \Phi_k$ represents the full transitive closure of gadget correspondence along the reduction chain, mapping each source gadget to the complete set of semantically induced gadgets in the final problem instance.

Highlighting is anchored to the default instance G_0: for any gadget $g \in g_0$, the set $\Phi(g)$ contains exactly those gadgets in the final reduced problem that are semantically induced by g. A single highlight color is assigned to g and propagated transitively through Φ, keeping the origin of each highlighted gadget visually traceable across the full chain. Figure 3.6 illustrates the result of this process for a reduction from 3SAT (through Clique, and then Vertex Cover) to Feedback Arc Set.

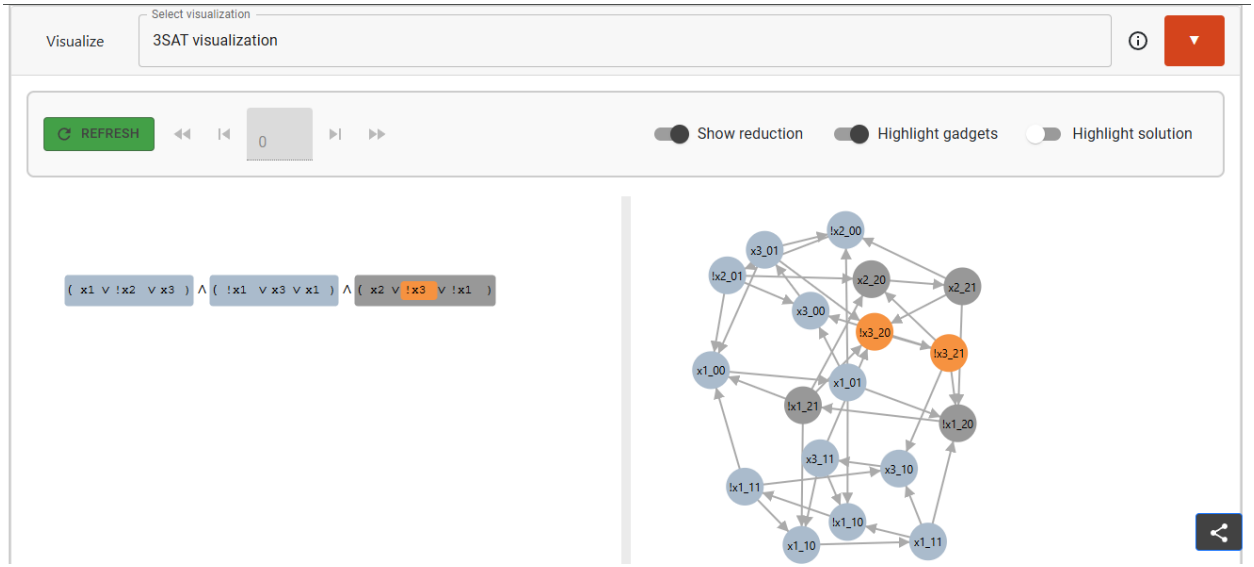


Figure 3.6: Transitive gadget highlighting in Redux for a 3SAT instance. The highlighted literal `!x3` in the SAT representation (left) is propagated to its corresponding nodes in the reduced graph (right), shown in orange. The *Highlight gadgets* toggle is enabled, demonstrating the corrected transitive highlighting system across the reduction chain.

3.6 Step-by-Step Solution Tracing

Prior to this work, Redux displayed only the initial problem instance and the final solved state of a problem, with no way for users to follow the intermediate steps of an algorithm's execution. Such a design choice made it difficult for students to develop intuition for how algorithms actually work, as the transition from problem to solution was opaque. Research in algorithm visualization pedagogy has consistently shown that active, incremental engagement with algorithm execution — rather than passive observation of a final result — is key to developing genuine algorithmic understanding [14]. Studies indicate that, outside of algorithm

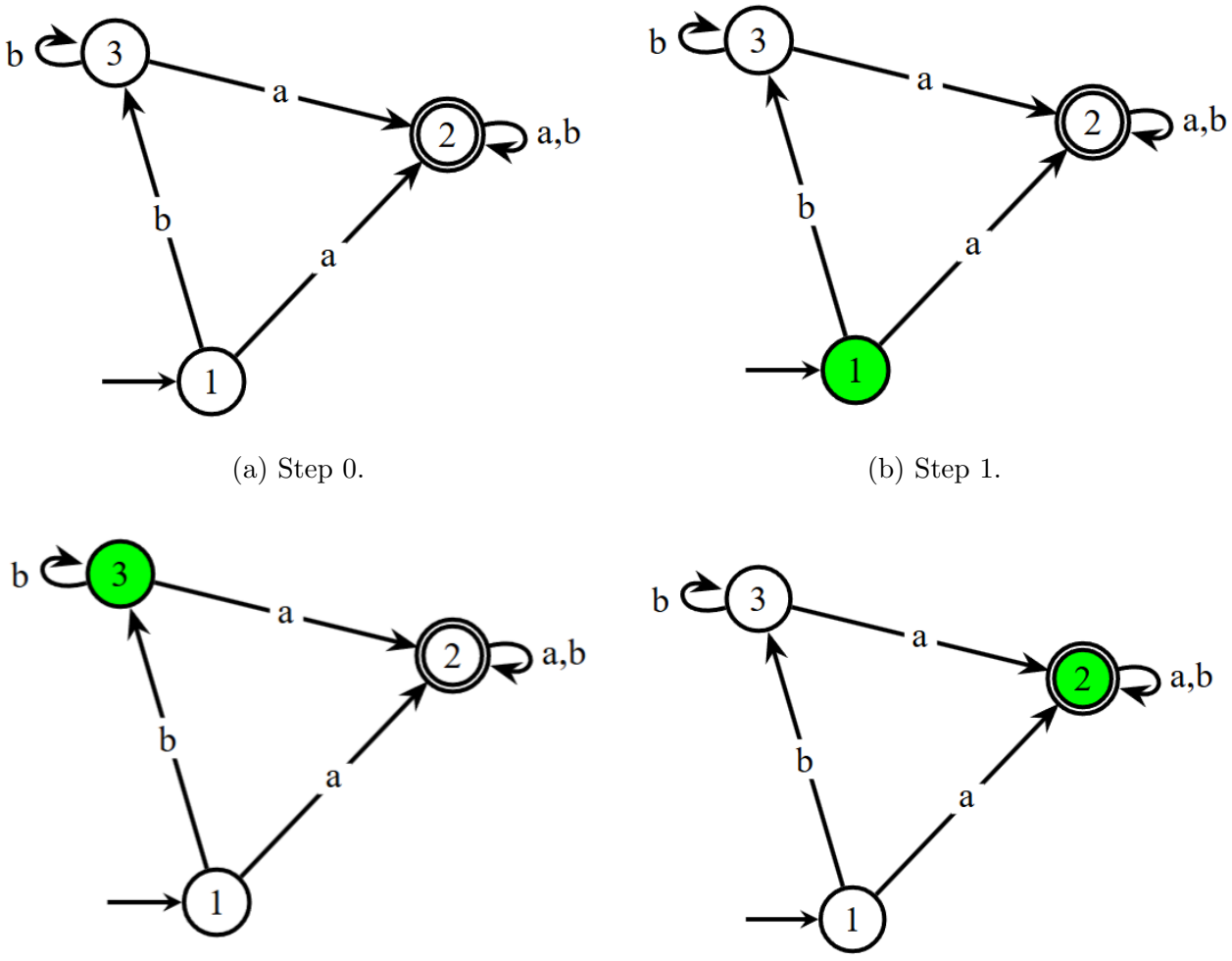
visualizations, interactive learning substantially improves comprehension and engagement [29].

With the introduction of **step-by-step solution tracing**, a new feature is available to students, educators and researchers, that allows users to follow the execution of a solution algorithm incrementally. Each visualization file on the backend now defines not only the initial and solved states of a problem, but also a sequence of intermediate steps that represent the algorithm's execution from start to finish. These steps are assembled by the API constructor and packaged into the API response alongside the initial and solved states.

On the frontend, the `VisualizerRowMemo` component parses the step sequence from the API response and exposes navigation controls that allow the user to step forward and backward through the algorithm's execution. At each step, the visualization updates to reflect the current state of the algorithm, with colors and highlights changing to draw the user's attention to the relevant elements. The `delay` field in the graph template API, for example, controls the timing of node and edge highlights during animated transitions between steps, allowing contributors to choreograph the visual progression of their algorithm with fine-grained control.

Such an addition required coordinated changes across both layers of the system. On the backend, the step sequence format was defined and standardized, requiring each visualization file to encode not only the initial and final states of a problem but also every intermediate transition the algorithm traverses on its way to a solution. On the frontend, a step navigation interface was implemented within `VisualizerRowMemo`, exposing forward and backward controls that allow the user to move through the algorithm's execution at their own pace. The rendering pipeline was updated accordingly to redraw the visualization at each step, updating colors, highlights, and labels to reflect the current algorithmic state and draw the user's attention to the elements most relevant to that moment in the execution. The result is a substantially more educational tool that allows students to develop genuine algorithmic intuition by observing each decision an algorithm makes as it executes, rather

than being presented only with a finished solution whose derivation remains opaque [23]. Figure 3.7 shows the step-by-step tracing interface applied to four problem instances — a Deterministic Finite Automata (DFA), a Steiner Tree, an Exact Cover, and a SAT instance — demonstrating the breadth of problem types that the tracing system supports across the reusable template suite.



(a) Step 0.

(b) Step 1.

(c) Step 2.

(d) Solution step.

Figure 3.7: **Step-by-step solution tracing.** The screenshot taken from Redux shows a Deterministic Finite Automata (DFA) problem instance. Directed edges carry transition labels (`a`, `b`), with nodes indicating states. This visualization demonstrates the template's applicability beyond NP-complete problems to automata theory. The visualized instance in mathematical notation is $((\{1,2,3\},\{a,b\},\{(1,a,2),(1,b,3),(2,a,2),(2,b,2),(3,a,2),(3,b,3)\},1,\{2\}),a)$.

3.7 Reusable Frontend Templates

Prior to this contribution, adding a new visualization to Redux required mastering an entirely separate skillset from standard software development: namely, the D3.js graphics library [3] and the intricacies of the Redux rendering pipeline [31]. D3.js is a low-level visualization library with a steep learning curve — even developers already proficient in JavaScript, HTML, and CSS can expect to spend upwards of 15 hours producing a single animated, interactive plot from scratch. For the many students and developers with little to no JavaScript experience, the barrier is substantially higher. Compounding this, the prior Redux pipeline provided no clear integration point for new visualizations, as rendering logic was largely hard-coded with no designated extension mechanism, leaving contributors to reverse-engineer the system before writing a single line of visualization code.

The contribution of **reusable frontend templates** eliminates that barrier entirely. New visualizations now require only a single template added to `visualizations.js` (Figure 3.4), with no knowledge of D3.js internals or the broader rendering pipeline required. Crucially, a set of standardized templates is provided out of the box — so for the common case of adding a new problem instance without needing a novel visualization type, no new code needs to be written at all. This reduces the candidate pool for visualization contributors from the rare few with specialized graphics programming experience to essentially any student or developer with basic programming literacy, dramatically broadening who can meaningfully extend Redux.

All four templates consume the standardized API fields described in the previous section and are registered by name in `visualizations.js`, which dispatches incoming visualization requests to the appropriate template based on the visualization name specified in the backend visualization file. This dispatch mechanism acts as a single, centralized extension point for the entire rendering pipeline — contributors never need to modify the core visualization infrastructure to introduce a new visual representation. Adding a new template requires only implementing the template component and registering its name; the rest of

the pipeline, including API querying, step extraction, and state management, is handled automatically by the existing infrastructure. This design ensures that the cost of extending the system remains constant regardless of how many templates already exist, making Redux's visualization framework as easy to extend on its hundredth contribution as it was on its first.

3.7.1 D3 Graph Template

The force-directed graph template renders relational data as a dynamic node-link diagram using D3's force simulation engine. Nodes are positioned iteratively by simulating physical forces — including charge repulsion, link attraction, and collision detection — until the layout reaches equilibrium.

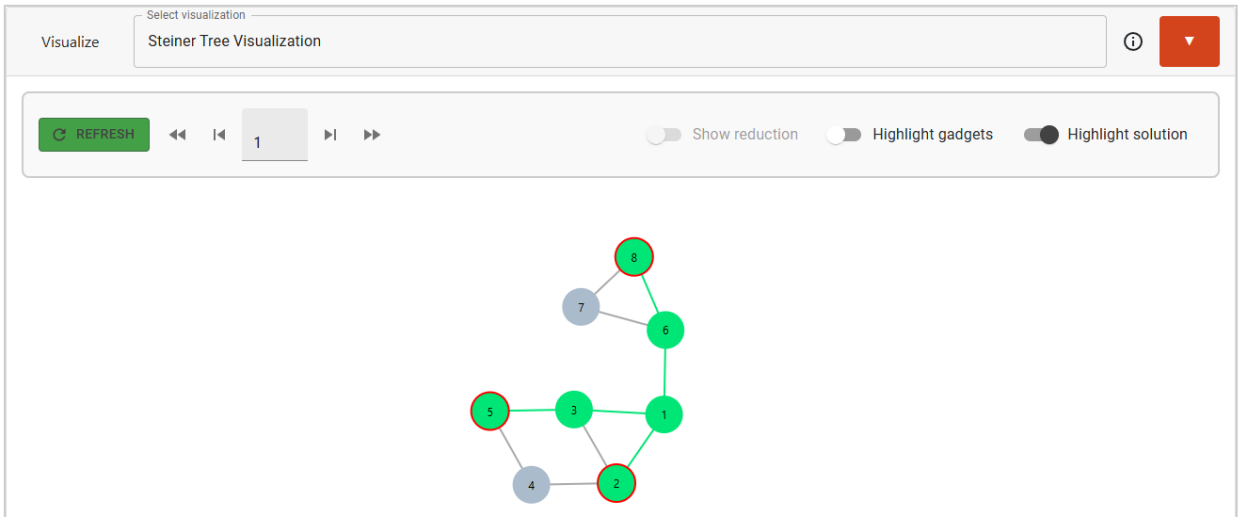


Figure 3.8: The D3 graph template rendering a Steiner Tree problem instance. Terminal nodes are highlighted in green using the `color` field and our target nodes we want to connect have a red outline using the `outline` field. The edges are highlighted using the `color` field.

The template accepts a standardized graph specification consistent with the API described in Table 3.1, translating node and edge definitions into an interactive SVG visualization with configurable force parameters.

Type	Field	Description
Node	`id`	Unique node identifier
Node	`name`	Label displayed inside the node
Node	`color`	Node fill color from palette
Node	`outline`	Border color for visual distinction
Node	`delay`	Highlight delay (ms) for animation steps
Node	`dashed`	Whether node border is dashed
Node	`additional`	Optional extra node metadata
Link	`id`	Unique link ID (e.g., `source-target`)
Link	`source`	Source node identifier
Link	`target`	Target node identifier
Link	`color`	Link color from palette
Link	`dashed`	Whether link is dashed
Link	`delay`	Highlight delay (ms)
Link	`weight`	Numerical weight of the link
Link	`weighted`	Whether weight is displayed
Link	`directed`	Whether link has direction (arrow)
Link	`attribute1`	Optional extra metadata field
Link	`attribute2`	Optional extra metadata field

Table 3.1: Graph template API fields (nodes and links)

3.7.2 D3 Set Template

The D3 set template renders set-based problem instances and solutions using D3.js. A particularly notable design feature of this template is its support for recursive set structures — sets may contain nested sets to arbitrary depth, enabling the representation of complex mathematical objects such as sets of tuples or sets of sets. The template distinguishes between

ordered tuples, rendered with parentheses, and unordered sets, rendered with curly braces, based on the `isOrdered` field.

Figure 3.9: The D3 set template rendering an Exact Cover problem instance. The universal set $\{1, 2, 3, 4\}$ is displayed alongside candidate subsets $\{1, 2, 3\}$, $\{2, 3\}$, and $\{4, 1\}$. Each set is rendered using curly brace notation, with nested elements displayed as individual labeled tiles. Both sets and their constituent elements are individually highlightable via the gadget system — selecting any set or element propagates the highlight to all semantically corresponding components across the reduction chain, consistent with the transitive map Φ defined in Section 3.5.

The template consumes a standardized API object containing a recursive list structure. The fields are described in Table 3.2. Figure 3.9 shows the D3 set template rendering an Exact Cover problem instance, where the universal set and candidate subsets are displayed as nested set structures using the recursive list representation.

Type	**Field**	**Description**
Set	`id`	Unique set identifier for highlighting
Set	`isOrdered`	Whether set is shown as an ordered tuple (parentheses)
Set	`list`	Elements or nested sets (recursive structure)
Element	`id`	Unique element identifier for highlighting
Element	`value`	Display label of the element
Element	`isValue`	Whether entry is a primitive value (not a set)

Table 3.2: Set template API fields

3.7.3 LaTeX Graph Template

The LaTeX graph template renders graph-based visualizations using TikZJax, a JavaScript port of the TikZ typesetting library [9]. This template is intended for contexts where publication-quality mathematical typesetting is important, producing output that is visually consistent with the notation used in academic textbooks and papers. It is particularly well-suited for problems where precise spatial layout of graph elements is required. While it follows the same structure and API conventions as the D3 graph template, the two differ in how they render the data visually. This is a direct example of the benefit of a standardized API: two visually distinct templates can share the same data format, allowing contributors to focus entirely on the rendering layer without modifying the backend.

3.7.4 D3 SAT Template

The SAT template renders Boolean satisfiability problem types, supporting clause and literal representations across a range of SAT variants. Each clause is rendered as a collection of literals, with the `color` field of each literal used to indicate its current state — for example, whether it has been assigned a satisfying truth value during solution tracing. Negated literals are represented using a leading `!` in the `literal` field, which the template renders using standard logical notation.

Type	**Field**	**Description**
Clause	`id`	Unique clause identifier
Clause	`literals`	Ordered list of literal objects
Literal	`id`	Unique literal ID (e.g., `clause-literal`)
Literal	`literal`	String form; negation uses `!` (e.g., `!x1`)
Literal	`color`	Display color indicating state (e.g., background/solution)

Table 3.3: SAT template API fields

The template consumes a standardized API object containing a list of clauses, each containing a list of literals. The fields are described in Table 3.3.

Figure 3.10 shows the SAT template rendering a satisfiability instance with solution highlighting enabled. Satisfied literals are rendered in green, allowing users to immediately identify which truth assignments satisfy each clause during step-by-step solution tracing.

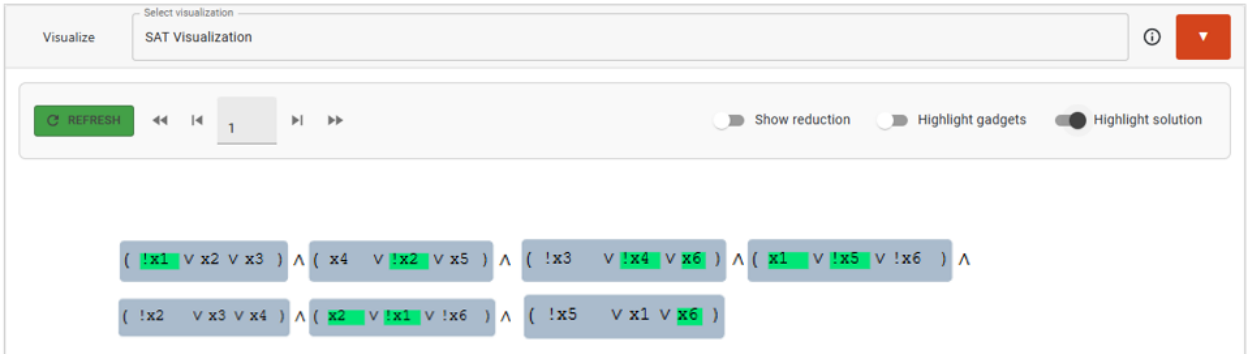


Figure 3.10: The SAT template rendering a satisfiability instance with the *Highlight solution* toggle enabled (step 1). Literals assigned a satisfying truth value are highlighted in green, while unassigned literals remain in the default background color. Negated literals (prefixed with !) are rendered using standard logical notation, and clauses are separated by conjunction symbols (∧).

3.7.5 Impact

Collectively, the four templates presented in this work — force-directed graphs, set visualization, and SAT — enable the visualization of problems spanning complexity classes from P through NP-Complete, including all 21 of Karp's canonical NP-Complete problems. Beyond this benchmark, the templates extend naturally to several dozen additional problems across complexity classes including NP-hard, PSPACE, P, and co-NP, reflecting the broad expressive power of graph, set, and satisfiability representations in theoretical computer science. This cross-class coverage is not incidental but a consequence of deliberate design: the three representational primitives underlying these templates are precisely the structures through which a large portion of classical complexity theory is defined, making the framework a natural pedagogical instrument for exploring the boundaries between complexity classes.

Chapter 4

Evaluation

This chapter evaluates the contributions of this thesis by examining how well they address the two research questions posed in the introduction. Since the primary goal of this thesis is to demonstrate that Redux can serve as a general-purpose, extensible, and easy-to-contribute-to visualization platform, the evaluation is grounded in a real-world practical assignment in which graduate students contributed new visualizations to Redux as part of a quantum computing course at Idaho State University.

4.1 Evaluation Approach

Rather than measuring the effectiveness of Redux through a controlled user study, our work adopts a demonstration-based evaluation approach. This is appropriate given the nature of the contributions — the central claims of this thesis are architectural and systemic, concerning whether Redux can be extended to new domains by contributors with modest development experience, and whether the redesigned architecture meaningfully reduces the effort required to do so [31]. A real-world contribution by students who had no prior involvement in Redux's development provides direct and concrete evidence for these claims.

4.2 Case Study: Quantum Computing Visualizations

To evaluate the extensibility of the redesigned Redux architecture, we incorporated the tool in a CS 6680 quantum computing (QC) course, taught by Dr. Paul Bodily. The course taught QC concepts over a 16 week period. Within the second half of the semester students

were introduced to Redux, worked in groups and were given access to the codebase and the existing frontend templates as reference material. Before taking the course, no prior knowledge of Redux's internal architecture was assumed or required. Likewise, students had little to no previous experience with QC.

The class of 10 consisted of master's and PhD students, all with a specialization in computer science. For the group project, students were invited to express their preferences regarding which aspect of the codebase they wished to contribute to, after which the course instructor formed three groups of three to four students, balancing individual preferences with complementary expertise. The resulting groups were: the problem class group, responsible for defining and encoding computational problems; the solutions group, responsible for implementing solution procedures; and the visualizations group, responsible for developing the visual templates discussed in this work.

Table 4.1 summarizes the contributions made by the student group. In roughly 6 weeks, 10 graduate CS students — none of whom had prior experience with Redux — implemented 6 new problems, 8 solution algorithms, and 6 visualizations, along with 2 reusable quantum circuit templates built using Q.js and D3.js. The group project was introduced on October 30th and final projects were submitted and presented on December 18th. The breadth of these contributions, spanning quantum computing problems such as Bernstein-Vazirani, Deutsch, Deutsch-Jozsa, and Simon's Problem, demonstrates that Redux's standardized API and visualization templates meaningfully lower the barrier to contribution, enabling a random sample of graduate students to extend the platform substantially in a short timeframe.

Table 4.1: Student contributions to Redux over a 6-week group project.

Category	**Contribution**
Problems	Bernstein-Vazirani
	Deutsch
	Deutsch-Jozsa
	Prime Factorization
	Simon's Problem
	Sudoku
Solutions	Bernstein-Vazirani Classical Solver
	Bernstein-Vazirani Quantum API Solver
	Deutsch Classical Solver
	Deutsch Quantum API Solver
	Deutsch-Jozsa Classical Solver
	Deutsch-Jozsa Quantum API Solver
	Shor's Quantum API Solver
	Simon's Problem Classical Solver
Visualizations	Bernstein-Vazirani Quantum Circuit (Q.js)
	Bernstein-Vazirani Quantum Circuit (D3.js)
	Deutsch Quantum Circuit (Q.js)
	Deutsch Quantum Circuit (D3.js)
	Deutsch-Jozsa Quantum Circuit (Q.js)
	Deutsch-Jozsa Quantum Circuit (D3.js)
Templates	Quantum Circuit Template (Q.js)
	Quantum Circuit Template (D3.js)

By the end of the semester, one group of students had successfully implemented and integrated quantum circuit visualization templates into Redux. The students built two templates — one using Q.js and one using D3.js — developing their own API schema suited to the structure of quantum circuit problems rather than being constrained to the conventions

of the existing templates. The quantum circuit visualizations were fully integrated into the Redux knowledge base, extending the platform into an entirely new domain of computer science well beyond its original NP-completeness focus. Notably, the student-implemented Python solvers accompanying these contributions further expanded Redux's technical scope, extending beyond the platform's original constraint of being exclusively implemented in C#. Figure 4.1 shows the Deutsch-Jozsa quantum circuit rendered using the D3-based template, with the visualization type dropdown visible at the top demonstrating that both the D3 and Q.js templates are available for the same problem.

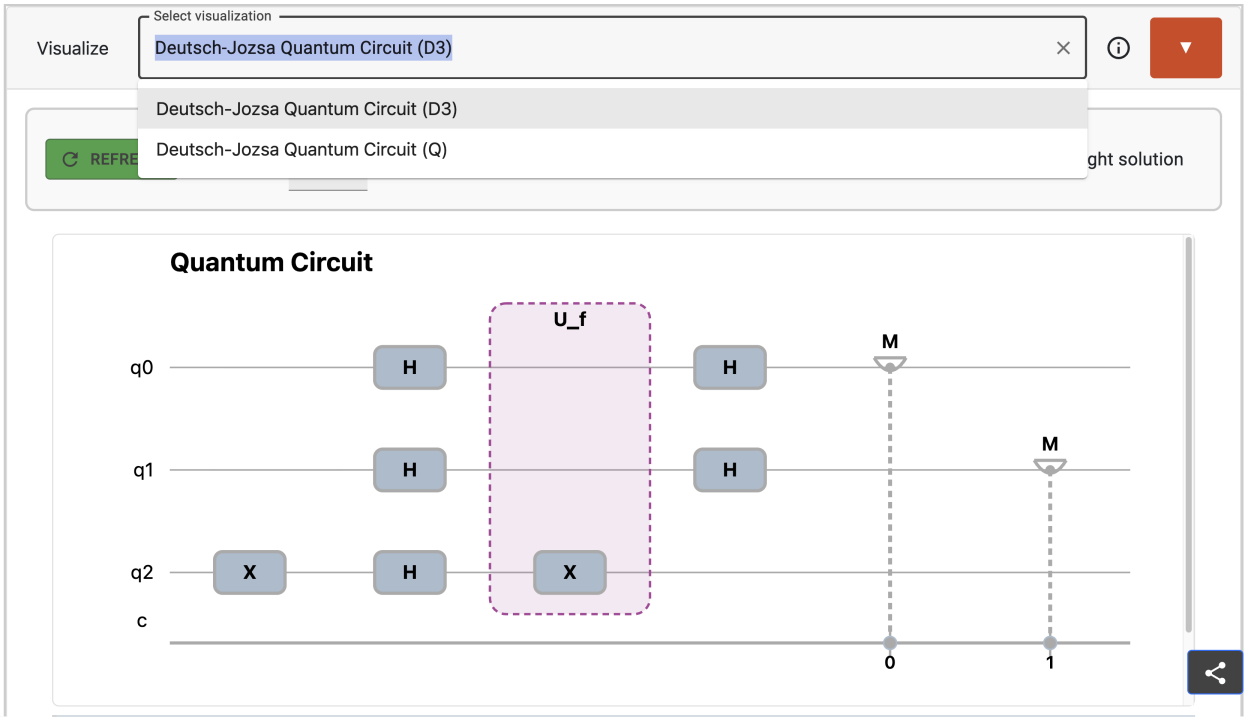


Figure 4.1: The student-contributed quantum circuit visualization rendering the Deutsch-Jozsa algorithm. Qubits q_0, q_1, and q_2 are shown with Hadamard (H) and Pauli-X (X) gates, the oracle U_f highlighted with a dashed boundary, and measurement operations (M) on the classical register c. The dropdown at the top shows two available visualization types for the same problem — the D3-based template (selected) and the Q.js-based template — directly demonstrating the multiple visualization types per problem, a contribution of this thesis.

This outcome is significant for several reasons. First, the students were able to contribute meaningfully to a production codebase within a single semester without requiring deep familiarity with Redux's underlying architecture [2]. Second, their decision to develop their

own API schema demonstrates that Redux's architecture is flexible enough to accommodate entirely new approaches to visualization — contributors are not locked into a single paradigm but can design data representations that best suit their problem domain [17]. Third, the successful integration of quantum circuit visualizations into Redux provides concrete evidence that the platform is capable of spanning a broad range of CS topics, from NP-complete reductions to quantum computing.

4.3 Research Question Responses

RQ1: How can a single extensible web-based platform be designed to support diverse visualization types across multiple computer science domains? Research Statement: A layered architecture that separates problem definition, API construction, and visual rendering into distinct modules enables a single platform to accommodate diverse visualization types across domains without requiring structural modifications to its core system. The integration of quantum circuit visualizations alongside the existing NP-completeness knowledge base provides direct evidence for this statement. Redux now supports visualization types spanning graph problems, set-based problems, Boolean satisfiability, and quantum circuits — a range of topics that spans theoretical computer science, logic, and quantum computing [1]. The modular folder structure, template dispatch mechanism, and standardized API together realize this layered design in practice [7], and the quantum computing case study demonstrates its extensibility: students were able to extend Redux into an entirely new domain within a single semester.

RQ2: What strategies can be employed to reduce the effort required to add new problem visualizations to an interactive visualization platform? Research Statement: A standardized API naming schema, reusable frontend templates, and an automated dispatch mechanism together meaningfully reduce the effort required to add new problem visualizations to an interactive platform. Three concrete strategies emerge from the work presented in this thesis. First, a standardized REST API naming schema [7, 17] ensures

that contributors need only follow established conventions rather than reverse-engineer an ad hoc system. Second, a suite of reusable frontend templates eliminates the need to build rendering logic from scratch for the most common visualization types [2]. Third, a template dispatch mechanism that automatically routes visualization requests by name means that registering a new template requires minimal wiring. The quantum computing case study provides evidence that these strategies are effective in practice: graduate students with no prior Redux experience were able to design their own API schema and integrate new templates within a single semester [10], suggesting that the combination of these strategies meaningfully lowers the barrier to contribution [11].

RQ3: What features contribute to effective pedagogical visualization tools for computer science education? Research Statement: Effective pedagogical visualization tools share a set of core features: incremental, step-by-step execution that mirrors the algorithm's logic; active engagement mechanisms such as gadget highlighting that invite students to trace correspondences rather than passively observe; and support for multiple visual representations of the same concept to accommodate different learning perspectives [8, 10, 25]. The literature consistently identifies active engagement as the primary driver of learning gains in algorithm visualization [14], and the features introduced in this thesis are designed with this principle in mind. Step-by-step solution tracing transforms Redux from a tool that shows what a solution looks like into one that reveals how it is reached, giving students a front-row view of algorithmic decision-making at each incremental step. Transitive gadget highlighting goes further, requiring students to actively trace the correspondence between problem elements across reduction chains — an engagement mode that mirrors the kind of reasoning NP-completeness theory demands. Finally, support for multiple visualization types per problem ensures that students who struggle to grasp a concept from one visual perspective have an alternative representation to fall back on, consistent with research showing that representational variety improves comprehension across diverse learners [23].

4.4 Limitations

While the case study provides encouraging evidence for the extensibility of Redux, several limitations of this evaluation should be acknowledged. The evaluation relies on a single group of contributors in a single course, and no formal measures of contribution effort, time, or difficulty were collected [31]. Furthermore, while quantum circuit visualizations represent a meaningful extension of Redux beyond NP-completeness, the platform's ability to support a broader range of CS topics — such as operating systems concepts, network protocols, or machine learning algorithms — remains to be demonstrated.

Chapter 5

Conclusion

This work set out to address a fundamental challenge in computer science education: the difficulty of building, maintaining, and extending interactive visualization tools [2, 31]. While the educational value of such tools is well established [14, 23], the expertise and effort required to develop them has historically limited their adoption, leaving many CS topics — particularly advanced theoretical topics — without adequate visual support. Substantial extensions to Redux — an open-source web application originally developed at Idaho State University for NP-completeness education — have transformed it into a general-purpose platform for interactive CS education that is broad in scope, extensible by design, and accessible to contributors with modest development experience.

5.1 Summary of Contributions

Through our contributions we introduced five core features that address the gaps for interactive CS education. A restructured backend API built around a consistent naming schema replaced the ad hoc wiring of the original system, making the contribution process consistent and predictable. Support for multiple visualization types per problem allows students and educators to explore different visual perspectives on the same concept. A corrected and generalized transitive gadget highlighting system properly handles arbitrarily long reduction chains, closing a correctness gap in the original implementation. Step-by-step solution tracing transforms Redux from a tool that shows what a solution looks like into one that shows how it is reached. Finally, a suite of four reusable frontend templates — covering graphs, sets,

Boolean satisfiability, and LaTeX-typeset graphs — gives contributors a concrete foundation to build on, removing the need to implement rendering logic from scratch.

5.2 Future Work

While these improvements substantially advances Redux as a platform for CS education, several directions for future work remain.

The most immediate opportunity is the continued expansion of the Redux knowledge base. The quantum computing visualizations contributed by graduate students through a university course demonstrate that Redux is capable of spanning diverse CS domains, but the platform's full potential will only be realized as more problems, algorithms, and visualization types are added [1]. Encouraging contribution from students across a wider range of courses and disciplines — including data structures, operating systems, network protocols, and machine learning — would significantly broaden Redux's reach and utility.

A second direction is the development of more sophisticated evaluation infrastructure. The evaluation presented is demonstration-based and relies on a single group of contributors, which, while encouraging, does not provide the statistical rigor of a controlled study [31]. Future work could include structured surveys measuring contributor experience and effort, controlled comparisons of contribution time before and after the architectural redesign, and longitudinal studies tracking student comprehension outcomes when using Redux in the classroom. Such evaluations would provide stronger empirical grounding for the claims made in this thesis and could inform further improvements to the platform.

A third direction is the development of additional frontend templates beyond the four introduced previously. While the existing templates cover a broad range of problem types, many CS topics have visualization needs that fall outside their scope [26, 27]. Templates for tree structures, state machines, matrix representations, and network flow diagrams would substantially expand the range of problems that can be visualized without requiring contributors to build rendering logic from scratch [2]. The quantum circuit templates

contributed by graduate students demonstrate that entirely new template paradigms can be successfully integrated into Redux, and future contributors are encouraged to follow this model.

A fourth direction is the improvement of Redux's onboarding and documentation infrastructure. While the technical barrier to contribution has been lowered significantly, new contributors still require guidance on how to structure their problem folders, implement their API constructors, and register their templates [11]. Comprehensive written documentation, video tutorials, and worked examples would further reduce the onboarding burden and make Redux accessible to an even wider range of contributors.

Finally, future work could explore the integration of Redux with existing CS education platforms such as OpenDSA, potentially allowing Redux's visualizations to be embedded directly within interactive eTextbooks and course management systems [8]. Such integration would dramatically increase Redux's reach and make its visualizations available to students who may never visit the Redux website directly.

5.3 Broader Impact on CS Education

The broader significance of this thesis extends beyond the specific contributions made to Redux. At its core, this thesis is a demonstration that the barriers which have historically limited the adoption of interactive visualization tools in CS education are not insurmountable [2, 31]. By designing a system that separates the concerns of problem definition, API construction, and visual rendering into distinct, well-documented layers, it becomes possible for students and educators — not just experienced developers — to contribute meaningfully to a shared visualization platform [7].

This has implications that reach beyond Redux itself. The architectural patterns introduced in this thesis — standardized API schema, reusable frontend templates, modular knowledge base organization, and automatic template dispatch — represent a general approach to building extensible visualization systems that could be applied to other educational tools

and platforms [1, 17]. As CS curricula continue to evolve and expand into new areas such as quantum computing, artificial intelligence, and cybersecurity, the need for visualization tools that can keep pace with that evolution will only grow [27]. Redux, as extended by this thesis, offers one model for how such tools can be built to last.

Ultimately, the goal of this work is to make CS education more engaging, more accessible, and more effective for students at every level [23]. Interactive visualizations have the power to transform abstract theoretical concepts into tangible, explorable objects — to make the invisible visible. By lowering the barrier to building and contributing such visualizations, this thesis takes a step toward a future in which every CS topic, from sorting algorithms to quantum circuits, has the interactive visual support it deserves.

References

[1] Adam Alami, Raúl Pardo, and Johan Linåker. Free open source communities sustainability: Does it make a difference in software quality? *Empirical Software Engineering*, 29(5):114, 2024. ISSN 1573-7616. doi: 10.1007/s10664-024-10529-6. URL `https://doi.org/10.1007/s10664-024-10529-6`.

[2] Jeremiah Blanchard, John R. Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. Stop reinventing the wheel! promoting community software in computing education. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '22, page 261–292, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9798400700101. doi: 10.1145/3571785.3574129. URL `https://doi.org/10.1145/3571785.3574129`.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185.

[4] Pierluigi Crescenzi. Using AVs to explain NP-completeness. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, page 299, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588209. doi: 10.1145/1822090.1822175. URL `https://doi.org/10.1145/1822090.1822175`.

[5] Pierluigi Crescenzi. AlViE: Java algorithm visualization environment. `https://github.com/piluc/AlViE`, 2024. GNU General Public License v3.0.

[6] Pierluigi Crescenzi, Emma Enström, and Viggo Kann. From theory to practice: NP-completeness for every CS student. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 16–21, 2013. doi: 10.1145/2462476.2465582.

[7] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[8] Eric Fouh, Ville Karavirta, Daniel A. Breakiron, Sally Hamouda, Simin Hall, Thomas L. Naps, and Clifford A. Shaffer. Design and architecture of an interactive etextbook – the opendsa system. *Science of Computer Programming*, 88:22–40, 2014. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2013.11.040. URL https://www.sciencedirect.com/science/article/pii/S016764231300333X. Software Development Concerns in the e-Learning Domain.

[9] Jim Fowler. TikZJax: TikZ running under WebAssembly in the browser, 2019. URL https://github.com/kisonecat/tikzjax.

[10] Philip Guo. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 1235–1251, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386357. doi: 10.1145/3472749.3474819. URL https://doi.org/10.1145/3472749.3474819.

[11] Waqas Haider, Muhammad Ilyas, Shah Khalid, and Sikandar Ali. Factors influencing sustainability aspects in crowdsourced software development: A systematic literature review. *Journal of Software: Evolution and Process*, 36(6):e2630, 2024. doi: https://doi.org/10.1002/smr.2630. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2630.

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

[13] Steven Halim. VisuAlgo: Visualising data structures and algorithms through animation. https://visualgo.net, 2025. National University of Singapore.

[14] CHRISTOPHER D. HUNDHAUSEN and SARAH A. DOUGLAS. Low-fidelity algorithm visualization. *Journal of Visual Languages Computing*, 13(5):449–470, 2002. ISSN 1045-926X. doi: https://doi.org/10.1006/jvlc.2002.0231. URL https://www.sciencedirect.com/science/article/pii/S1045926X02902314.

[15] Essi Isohanni and Hannu-Matti Järvinen. Are visualization tools used in programming education? by whom, how, why, and why not? In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, page 35–40,

New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330657. doi: 10.1145/2674683.2674688. URL https://doi.org/10.1145/2674683.2674688.

[16] Ari Korhonen, Thomas L. Naps, Charles Boisvert, Pilu Crescenzi, Ville Karavirta, Linda Mannila, Bradley Miller, Briana Morrison, Susan H. Rodger, Rocky Ross, and Clifford A. Shaffer. Requirements and design strategies for open source interactive computer science ebooks. In *Proceedings of the ITiCSE Working Group Reports*, ITiCSE-WGR '13, pages 53–72, 2013. doi: 10.1145/2543882.2543886.

[17] Li Li, Wu Chou, Wei Zhou, and Min Luo. Design patterns and extensibility of rest api for networking applications. *IEEE Transactions on Network and Service Management*, 13(1):154–167, 2016. doi: 10.1109/TNSM.2016.2516946.

[18] Amine El Malki and Uwe Zdun. Combining api patterns in microservice architectures: Performance and reliability analysis. Zenodo preprint, 2023. URL https://zenodo.org/records/8066290. Discusses the Request Bundle API pattern to reduce communication overhead by combining multiple operations into a single request.

[19] K. Marchetti and P. Bodily. KAMI: Leveraging the power of crowd-sourcing to solve complex, real-world problems. In *2022 Intermountain Engineering, Technology and Computing (IETC)*, pages 1–4, Orem, UT, USA, 2022. doi: 10.1109/IETC54973.2022.9796945.

[20] K. Marchetti and P. Bodily. Visualizing the 3SAT to CLIQUE reduction process. In *2022 Intermountain Engineering, Technology and Computing (IETC)*, pages 1–5, Orem, UT, USA, 2022. doi: 10.1109/IETC54973.2022.9796851.

[21] Kaden Marchetti, Andrija Sevaljevic, Alex Diviney, Caleb Eardley, Russell Phillips, Rajiv Khadka, Daniel Igbokwe, and Paul Bodily. Redux: An interactive, dynamic knowledge base for teaching np-completeness. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, page 255–261, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706004. doi: 10.1145/3649217.3653544. URL https://doi.org/10.1145/3649217.3653544.

[22] Meta Open Source. React: A JavaScript library for building user interfaces. https://react.dev, 2013. Accessed 2026.

[23] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in

computer science education. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '02, page 131–152, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 9781450374491. doi: 10.1145/960568.782998. URL https://doi.org/10.1145/960568.782998.

[24] R. Phillips and P. M. Bodily. SPADE: A library for programmatic parsing and verification of discrete data structures. In *2025 Intermountain Engineering, Technology and Computing (IETC)*, pages 1–5. IEEE, 2025.

[25] Susan H. Rodger, Eric Wiebe, Kyung Min Lee, Chris Morgan, Kareem Omar, and Jonathan Su. Increasing engagement in automata theory with jflap. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, page 403–407, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605581835. doi: 10.1145/1508865.1509011. URL https://doi.org/10.1145/1508865.1509011.

[26] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. Algorithm visualization: The state of the field. *ACM Trans. Comput. Educ.*, 10(3), August 2010. doi: 10.1145/1821996.1821997. URL https://doi.org/10.1145/1821996.1821997.

[27] Naaz Sibia, Michael Liut, and Carolina Nobre. Exploring the Role of Visualization Tools in Enhancing Computing Education: A Systematic Literature Review. In Jillian Aurisano, Robert S. Laramee, and Carolina Nobre, editors, *EuroVis 2025 - Education Papers*. The Eurographics Association, 2025. ISBN 978-3-03868-273-8. doi: 10.2312/eved.20251027.

[28] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013. ISBN 9781133187790.

[29] Simon Su, Edward Zhang, Paul Denny, and Nasser Giacaman. A game-based approach for teaching algorithms and data structures using visualizations. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1128–1134, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3432520. URL https://doi.org/10.1145/3408877.3432520.

[30] Hongyi Sun, Waileung Ha, Min Xie, and Jianglin Huang. Modularity's impact on the quality and productivity of embedded software development: a case study in a hong kong company. *Total Quality Management Business Excellence*, 26:1–14, 08 2014. doi: 10.1080/14783363.2014.920179.

[31] Keith Tran, John Bacher, Yang Shi, James Skripchuk, and Thomas Price. Overcoming barriers in scaling computing education research programming tools: A developer's perspective. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*, ICER '24, page 312–325, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704758. doi: 10.1145/3632620.3671113. URL https://doi.org/10.1145/3632620.3671113.