

Pedagogical Execution Traces (PETs) for Educational Algorithm Visualization

Markus Weninger

Institute for System Software

Johannes Kepler University

Linz, Austria

markus.weninger@jku.at

Abstract—Algorithm visualizations can help novices learn, but many students still struggle to build accurate mental models without explicit guidance. *Pedagogical Execution Traces* (PETs) address this gap by enriching traces with embedded explanations, visual highlights, and interactive questions. Since many algorithm visualizations already rely on event traces, this work-in-progress idea paper adapts PETs to trace-based algorithm visualization. We motivate the approach through existing visualizations, refine PET design goals for algorithm-specific event traces and coordinated views, present early applications to Dijkstra’s algorithm, and discuss ongoing work on pedagogical evaluation and LLM-assisted visualization annotations.

Index Terms—Algorithm Visualization, Execution Traces, Software Engineering Education, Pedagogical Annotations, Interactive Learning, Dynamic Analysis, Large Language Models

I. INTRODUCTION

Many concepts in software development are difficult to learn because they are dynamic and abstract. As a consequence, novices often struggle to reason about execution flow, data structure states, recursion, and the results of individual steps [1]–[5]. Algorithm visualization (AV) addresses this by making algorithm behavior perceptible and explorable, which can support more accurate mental models and improve learning [3], [6], [7]. However, visualizing an algorithm does not automatically explain it [8]. In live classes, instructors can point to relevant elements, connect visible changes to the underlying algorithm, and ask learners to predict next steps. In self-paced learning settings, the same visualization often merely becomes a replayable animation without pedagogical context. Prior work thus emphasizes that visualizations should be embedded in explanation and active engagement rather than consumed passively [7], [9]. Manual annotations can provide such guidance [10]–[12], but annotations tied to fixed screen positions are costly to author and brittle to maintain [13]–[15].

We previously introduced *Pedagogical Execution Traces* (PETs) for guided trace-based debugging [13]. PETs enrich traces with *Pedagogical Annotations* (PAs), such as explanations, highlights, and interactive questions, that are bound to trace states and semantic program entities such as variables or heap objects. In algorithm visualization, these annotation targets become algorithm-specific semantic entities such as vertices or edges. Since algorithm visualizations also commonly rely on traces, preserving pedagogical intent within

the trace can turn simple algorithm visualizations into guided learning artifacts.

In this paper, we adapt PETs from general execution traces to algorithm-oriented event traces. We summarize the historical and technical background of algorithm visualization, refine the PET design for this setting, present early applications to Dijkstra’s algorithm, and discuss potential future AI-assisted authoring using large language models [16]–[18].

II. ALGORITHM VISUALIZATION

History and Related Work: Algorithm visualization has a long history [19]. One of the most prominent early algorithm visualizations is *Sorting Out Sorting* [20], [21], a video visualizing various sorting algorithms using animated bars representing array elements. It influenced many subsequent algorithm visualization efforts [22] and similar videos [23]. The BALSA algorithm visualization system by Brown and Sedgewick [24] (which led to Brown’s 1987 ACM Distinguished Dissertation *Algorithm Animation* [25]) inspired many other early systems [26], [27]. These systems already employed various visualization techniques such as color and sound [28] – the latter still being experimented with today [29]. Around the same time, *software visualization* tools, often in the form of debuggers, also emerged [30], [31], anticipating later educational visualization and debugging systems such as Jeliot [32], JIVE [33], jGRASP [34], Python Tutor [35], [36], and JavaWiz [37]. Algorithm visualization has since evolved into a rich field with diverse techniques, ranging from comic strips [38] to 3D visualizations [39].

Traces: Nearly all these approaches use some kind of *trace* to build their visualizations. While debugger-like tools such as JavaWiz [37] typically rely on *execution traces* (i.e., sequences of recorded events or program states from which the stack, heap, and static fields can be reconstructed at arbitrary points in time), tools that focus on algorithm visualization instead rely on *event traces*. As early as the 1980s, BALSA introduced the *interesting event* paradigm [24], which remains relevant up to this day: in algorithm visualization, not all steps in an algorithm need to be visualized but only those where significant changes or operations take place. Thus, in order to animate an algorithm, a programmer must first implement the algorithm in some programming language and modify it by inserting special procedure calls (i.e., tracing)

at the places where interesting events happen. These events are then encoded and collected to form the event trace. The trace can then be processed and visualized with step-by-step, bidirectional navigation, a common design principle in trace-based visualization systems [40]–[43].

A trace thus separates what happened in the algorithm from how this behavior is rendered. A sorting trace may record comparisons and swaps, which one view renders as highlighted bars while another represents them in a table or alongside pseudocode. Yet, most existing tools simply replay recorded behavior but do not encode pedagogical intent within the trace. For example, a sorting algorithm trace may record which elements have been compared or swapped (i.e., the *interesting events*), but they lack explanations of *why* these events are interesting or how they relate to the underlying algorithm. This is where our concept of *Pedagogical Execution Traces* (PETs) comes into play.

III. ADAPTING PETs TO ALGORITHM VISUALIZATION

This section details the adaptation of our PET concept [13] from debugger-oriented execution traces to algorithm-oriented event traces. The central idea is to preserve not only the sequence of algorithmic events, but also the pedagogical intent that helps learners understand which events are important and how their underlying entities relate to the algorithm’s behavior.

A. Motivation for PETs in Algorithm Visualization

We motivate our work based on experience collected across multiple Bachelor’s and Master’s theses on various trace-based educational algorithm visualizations (including sorting algorithms, string-matching algorithms, as well as graph and graph-flow algorithms – see <https://univiz.org>). All these tools first build a trace from instrumented source code, transform this trace into an intermediate representation, and finally transform that representation into the final visualization.

Burgstaller [44] developed a visualization tool for Dijkstra’s algorithm and collected 103 student responses on a 5-point Likert scale regarding helpfulness. 77 rated his tool as “(Very) Helpful” while only 2 considered it “Less Helpful”. Similarly, when asked whether they preferred the interactive tool visualizations over traditional slides, 78 students rated the tool “(Much) Better” than slides, while only 3 preferred slides. Yet, even though students praised the visualizations, they also expressed a desire for more explanations and guidance. Some students explicitly mentioned that they would have liked to see “speech bubbles as in the slide set” (see Figure 1). Thus, our PET concept aims to add such explanations, as well as highlights and questions, to existing interactive visualizations.

B. The General Concept

The basic premise of PETs is that when trace-based tools are used in educational contexts, the lecturer typically has a certain intent about what the visualization should teach, i.e., the lecturer has a mental model in mind that the students should comprehend by looking at the visualization. Thus, trace-based educational tools should not only preserve what

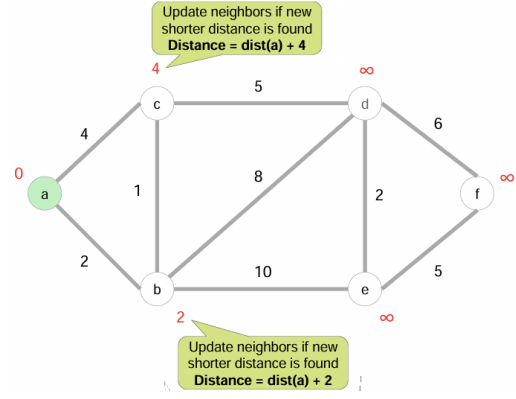


Fig. 1. Students expressed their desire for explanations in interactive algorithm visualizations similar to speech bubbles used on our lecture slides about Dijkstra’s algorithm.

happened during an execution, but also why particular states matter for learning. Unfortunately, the visualization alone does not necessarily communicate the instructor’s pedagogical intent: which state transition is conceptually important, which visual element deserves attention, or which prediction a learner should be able to make before continuing.

PETs address this gap by enriching execution traces with *Pedagogical Annotations* (PAs). A PA is attached to a specific entity at a specific state or transition in the trace and is rendered by the visualization when the learner reaches the corresponding point. In this way, PETs turn a trace from a simple replayable record of algorithm behavior into a guided learning artifact. This framing is particularly suitable for algorithm visualization. Prior work has shown that the pedagogical value of algorithm visualizations depends strongly on how learners engage with them, not only on whether an animation is available [45], [46]. Passive viewing may help learners observe an execution, but stronger forms of engagement ask learners to respond, predict, change inputs, construct explanations, or present their understanding [7], [47], [48]. PETs operationalize such engagement directly inside the trace: explanations can clarify important steps, highlights can direct attention to relevant entities, and questions can prompt learners to predict the next state before the visualization continues [49]. Following the original PET concept [13], PAs can be described by a small set of properties:

- **Action:** the pedagogical operation – either explaining, highlighting, or asking a question.
- **View:** an optional restriction to a particular visualization component, e.g., a graph view, array view, or table view.
- **Target:** the semantic entity to which the annotation is anchored, such as a certain vertex, edge, array cell, queue entry, table row, or pseudocode line.
- **Payload:** explanation text, highlight style, or a question with corresponding answer options.

In essence, this leads to a new tracing method such as `trace(action, view, target, payload)` that can be called from the instrumented source code to add pedagogical annotations to the trace, which can then be rendered by the visualization. The renderer has to be

extended to recognize the new trace entries and render the corresponding annotations in the visualization using methods such as `addSpeechBubble(target, view, text)`, `addHighlight(target, view)` or `addQuestion(target, view, text, answers)`.

C. Design Goals for PETs in Algorithm Visualization

To effectively preserve pedagogical intent within an algorithmic event trace, we identified three main design goals (DGs) when adapting PETs to algorithm visualizations:

DG1: Annotations Must Support Novice Comprehension:

To mirror in-class guidance, e.g., explanations a lecturer might provide or questions a lecturer might ask, we reuse the same three PET interaction types identified for guided debugging:

- **Explain:** Contextual “speech bubbles” embedded directly within the visualizations. These bubbles are attached to specific entities (e.g., a specific vertex, edge or array entry) at the moment they become relevant.
- **Highlight:** Visual highlights (e.g., colored borders or dimming the surrounding view) to direct the learner’s attention to a specific graphical element. This is crucial for guiding the eye during complex state changes (e.g., linked list operations).
- **Ask (Questions):** Assessments embedded within the trace. These can take the form of single-choice, multiple-choice, or short free-text questions. Crucially, “Ask” annotations can act as *execution gates*, requiring the student to predict the next state (e.g., “Which variable will change in the next step?”) before allowing them to continue stepping through the trace.

DG2: Do Not Disrupt the Original Visualization: PETs should integrate seamlessly with existing visualizations without requiring significant changes to the underlying trace generation or rendering logic. This means that PET annotations should be optional and non-intrusive, allowing instructors to add pedagogical intent without having to redesign their visualizations from scratch. We achieve this in three ways:

- The original trace logic remains unchanged, as PET annotations are added as new, separate entries in the trace.
- The visualization rendering logic is designed to recognize and render PET annotations without requiring modifications to the existing visualization views (e.g., as an overlay on top of the existing visualization).
- PA visualization can be toggled on and off. This allows instructors to use the original non-annotated visualization in live teaching, or students to first explore the visualization on their own before revealing PET guidance.

DG3: Binding Annotations to Algorithmic Entities: Many algorithm visualizations present an algorithm through several coordinated views, such as a graph, a table, a queue, or pseudocode. While previous work, for example by Sirkiä and Sorva [10], bound annotations to screen coordinates, our most important design decision is to instead bind annotations to algorithm-specific semantic entities. Whereas the original PET concept binds annotations to program entities such as

variables or heap objects, an algorithm visualization and its PET instrumentation agree on a common set of higher-level annotatable algorithmic entities, such as vertices or edges. This makes PETs layout-independent and enables the same trace to be rendered in different views. For example, in a Dijkstra visualization, vertex B may appear as a node in a graph, as a row or column in a distance table, and as an entry in a priority queue. A single PA can therefore be attached to vertex B and rendered in any view in which that vertex appears, or restricted to a specific view where appropriate, ensuring consistency and reducing authoring effort. A PET can thus express the pedagogical intent once, while each view decides how to render the corresponding annotation.

IV. EARLY APPLICATIONS OF PETs TO ALGORITHM VISUALIZATIONS

Because this is a work-in-progress idea paper, our investigation of PETs for algorithm visualization is still ongoing. To provide preliminary insight into their potential, we present a PET-based visualization of Dijkstra’s algorithm. It finds the shortest path from a source vertex to all other vertices in a weighted graph [50]. We use a graph visualization to depict edge selection and a tabular view to summarize the shortest distances found so far and the processed nodes. PETs now guide students’ attention to specific visual elements. For example, the following PET annotation guides attention when deciding which node to process next, highlighting the corresponding column in the tabular view, as shown in Figure 2:

```
{ action: "highlight", view: "TableView",
  target: { kind: "localMin" },
  payload: { color: "#b7f7c2", opacity: 0.55 } }
```

Current Vertex	Already Visited	Not Yet Visited	d(A)	d(B)	d(C)	d(D)	d(E)	d(F)	d(G)	Local Min
D	D	A B C E F G	Infinity via -	Infinity via -	Infinity via -	.	3 via D E	2 via D F	Infinity via -	2
F	D F	A B C E G	Infinity via -	Infinity via -	Infinity via -	.	3 via D E	.	Infinity via -	3

Fig. 2. Highlights via background color in the Table View.

Besides *highlight* annotations, PETs also support *explain* and *ask* (question) annotation actions. For example, the following PET annotation describes a question with multiple answer options anchored to an edge in the graph view:

```
{ action: "ask",
  view: "GraphView",
  target: edge(fromId, toId),
  payload: { q: ..., answers: [...], gate: true } }
```

Figure 3 shows the corresponding visualization of this question. By changing the `view` in the PET annotation, we could choose the visualization component in which the question is shown (or omit `view` entirely to show the question in multiple views). `target` anchors the question to a specific element, here the edge under consideration. Setting `gate` to `true` prevents the learner from advancing until the question has been answered correctly.

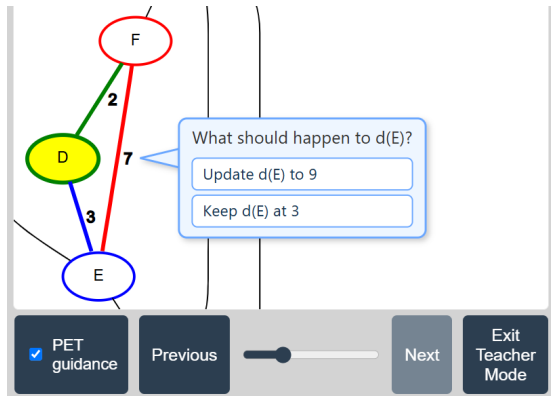


Fig. 3. Questions can act as *execution gates* (see the disabled “Next” button) that require active student participation.

V. ONGOING AND FUTURE WORK

PETs open several avenues for future research, ranging from pedagogical evaluations to advanced AI integrations.

A. Student Interaction Styles

To measure the pedagogical impact of PETs, we plan to conduct user studies comparing cognitive load, engagement, and retention across three scenarios: live instructor presentations, passive video recordings, and active PET stepping. We hypothesize that PETs’ self-paced nature and “Ask” execution gates will foster active learning and increase retention.

B. AI-Assisted Visualization Navigation

We want to explore whether LLM support can be embedded into educational visualizations, for example by allowing students to ask questions about the current state of the algorithm and having the LLM answer these questions *visually*. As an initial evaluation of this, we are currently conducting a study on optimizing LLM inputs (source code, raw trace, and images) to maximize explanation quality while minimizing latency. Additionally, we are comparing student ratings of AI-generated versus human-authored annotations.

We further envision stronger LLM integration into algorithm visualizations via the Model Context Protocol (MCP). Rather than statically generating PETs (with fixed pedagogical annotations at given trace locations), LLMs could autonomously run the visualization tool (either alone or in cooperation with a student or lecturer), dynamically step through the algorithm, highlight relevant visualization elements, and inject dynamic real-time annotations [51] based on perceived student struggles, effectively acting as a personalized, interactive tutor.

VI. CONCLUSION

Algorithm visualizations can make dynamic algorithm behavior visible, but visibility alone does not guarantee understanding. Especially in self-paced learning settings, students may observe state changes without recognizing why a transition is important or what reasoning step they should perform next. In this work-in-progress idea paper, we adapted *Pedagogical Execution Traces* (PETs) as a lightweight pedagogy layer for trace-based algorithm visualizations.

PETs enrich existing execution or event traces with *Pedagogical Annotations* such as explanations, highlights, and interactive questions. By binding these annotations to trace states and semantic algorithmic entities rather than to screen coordinates, PETs can preserve pedagogical intent while remaining compatible with different visual representations of the same algorithm. We discussed how this idea applies to educational algorithm visualization, derived design goals for PET-enhanced visualizations, and presented early applications to existing trace-based visualizations of Dijkstra’s algorithm.

Ongoing work focuses on evaluating the pedagogical impact of PETs in comparison to live explanations, passive recordings, and unguided interaction. We also see substantial potential in LLM-assisted authoring, where LLMs help generate, refine, or adapt PET annotations for existing visualizations. Ultimately, we envision PETs as a step toward algorithm visualizations that do not merely replay what an algorithm does, but actively guide learners through the underlying reasoning.

DATA AVAILABILITY STATEMENT

To support open science and reproducibility, the presented tool is available at <https://univiz.org> and its source code is archived at <https://doi.org/10.5281/zenodo.20729675>.

REFERENCES

- [1] A. V. Robins, J. Rountree, and N. Rountree, “Learning and Teaching Programming: A Review and Discussion,” *Comput. Sci. Educ.*, vol. 13, no. 2, pp. 137–172, 2003. [Online]. Available: <https://doi.org/10.1076/csed.13.2.137.14200>
- [2] E. Lahtinen, K. Ala-Mutka, and H. Järvinen, “A study of the difficulties of novice programmers,” in *SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITICSE 2005)*, 2005, pp. 14–18. [Online]. Available: <https://doi.org/10.1145/1067445.1067453>
- [3] J. Sorva, “Notional machines and introductory programming education,” *ACM Trans. Comput. Educ.*, vol. 13, no. 2, pp. 8:1–8:31, 2013. [Online]. Available: <https://doi.org/10.1145/2483710.2483713>
- [4] Y. Qian and J. Lehman, “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review,” *ACM Trans. Comput. Educ.*, vol. 18, no. 1, pp. 1:1–1:24, 2017. [Online]. Available: <https://doi.org/10.1145/3077618>
- [5] Z. Ullah, A. Lajis, M. Jamjoom, A. H. Altalhi, A. A. Al-Ghamdi, and F. Saleem, “The effect of automatic assessment on novice programming: Strengths and limitations of existing systems,” *Comput. Appl. Eng. Educ.*, vol. 26, no. 6, pp. 2328–2341, 2018. [Online]. Available: <https://doi.org/10.1002/cae.21974>
- [6] L. Ma, J. Ferguson, M. Roper, I. Ross, and M. Wood, “Improving the mental models held by novice programmers using cognitive conflict and jeliot visualisations,” in *SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITICSE 2009)*, 2009, p. 166–170. [Online]. Available: <https://doi.org/10.1145/1562877.1562931>
- [7] T. L. Naps, G. Röbling, V. L. Almstrum, W. P. Dann, R. Fleischer, C. D. Hundhausen, A. Korhonen, L. Malmi, M. F. McNally, S. H. Rodger, and J. Á. Velázquez-Iturbide, “Exploring the Role of Visualization and Engagement in Computer Science Education,” *SIGCSE Bull.*, vol. 35, no. 2, pp. 131–152, 2003. [Online]. Available: <https://doi.org/10.1145/782941.782998>
- [8] A. Moreno and M. Joy, “Jeliot 3 in a Demanding Educational Setting,” in *Program Visualization Workshop (PVW)*, vol. 178. Elsevier, 2006, pp. 51–59. [Online]. Available: <https://doi.org/10.1016/j.entcs.2007.01.033>
- [9] G. Banerjee, S. Murthy, and S. Iyer, “Effect of active learning using program visualization in technology-constrained college classrooms,” *Res. Pract. Technol. Enhanc. Learn.*, vol. 10, no. 1, p. 15, 2015. [Online]. Available: <https://doi.org/10.1186/s41039-015-0014-0>
- [10] T. Sirkiä and J. Sorva, “Tailoring animations of example programs,” in *Koli Calling Conf. on Computing Education Research*. ACM, 2015, pp. 147–151. [Online]. Available: <https://doi.org/10.1145/2828959.2828965>

- [11] T. Sirkiä, “Jsvee & Kelm: Creating and Tailoring Program Animations for Computing Education,” in *Working Conf. on Software Visualization (VISOFT)*. IEEE Computer Society, 2016, pp. 36–45. [Online]. Available: <https://doi.org/10.1109/VISOFT.2016.24>
- [12] —, “Jsvee & Kelm: Creating and Tailoring Program Animations for Computing Education,” *J. Softw. Evol. Process.*, vol. 30, no. 2, 2018. [Online]. Available: <https://doi.org/10.1002/smr.1924>
- [13] M. Weninger, “Towards Guided Omniscient Debugging in Education using Pedagogical Execution Traces (PETs),” in *Workshop on Future Debugging Techniques (DEBT 2026)*, 2026. [Online]. Available: https://ssw.jku.at/General/Staff/Weninger/Papers/Weninger_DEBT_26_Preprint.pdf
- [14] M. D. Rahman, B. Doppalapudi, G. J. Quadri, and P. Rosen, “A Survey on Annotations in Information Visualization: Empirical Studies, Applications and Challenges,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 31, no. 12, pp. 10439–10456, 2025.
- [15] C. Lai, Z. Lin, R. Jiang, Y. Han, C. Liu, and X. Yuan, “Automatic Annotation Synchronizing with Textual Description for Visualization,” in *ACM Conf. on Human Factors in Computing Systems (CHI 2020)*, 2020, p. 1–13. [Online]. Available: <https://doi.org/10.1145/3313831.3376443>
- [16] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, “Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models,” in *ACM Int’l Conf. on Comp. Edu. Research (ICER 2022)*, 2022. [Online]. Available: <https://doi.org/10.1145/3501385.3543957>
- [17] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, and A. Hellas, “Comparing Code Explanations Created by Students and Large Language Models,” in *Conf. on Innovation and Technology in Computer Science Education (ITiCSE 2023)*, 2023, p. 124–130. [Online]. Available: <https://doi.org/10.1145/3587102.3588785>
- [18] D. Lohr, H. Keuning, and N. Kiesler, “You’re (Not) My Type-Can LLMs Generate Feedback of Specific Types for Introductory Programming Tasks?” *J. Comput. Assist. Learn.*, vol. 41, no. 1, 2025. [Online]. Available: <https://doi.org/10.1111/jcal.13107>
- [19] J. T. Stasko, M. H. Brown, J. B. Domingue, and B. A. Price, Eds., *Software Visualization*. MIT Press, Feb. 1998.
- [20] R. M. Baecker and D. Sherman, “Sorting out Sorting,” 30-minute color sound film, 1981, presented at ACM SIGGRAPH 1981; excerpted in ACM SIGGRAPH Video Review 7, 1983.
- [21] R. M. Baecker, “Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science,” in *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.
- [22] D. Furcy, T. L. Naps, and J. Wentworth, “Sorting out sorting: the sequel,” in *13th Annual SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITiCSE 2008)*, 2008, pp. 174–178. [Online]. Available: <https://doi.org/10.1145/1384271.1384319>
- [23] T. Bingmann, “The Sound of Sorting,” 2013, accessed 2026-06-06. [Online]. Available: <https://panthema.net/2013/sound-of-sorting/>
- [24] M. H. Brown and R. Sedgewick, “Techniques for Algorithm Animation,” *IEEE Softw.*, vol. 2, no. 1, pp. 28–39, 1985.
- [25] M. H. Brown, *Algorithm Animation*, ser. ACM Distinguished Dissertations. MIT Press, 1988. [Online]. Available: <https://mitpress.mit.edu/books/algorithm-animation>
- [26] T. D. Collins, “The Visualization of Genetic Algorithms - Related Work,” Knowledge Media Institute, The Open University, Tech. Rep. KMI-95-12, 1995. [Online]. Available: <https://kmi.open.ac.uk/publications/pdf/kmi-95-12.pdf>
- [27] —, “The Application of Software Visualization Technology to Evolutionary Computation: A Case Study in Genetic Algorithms,” Ph.D. dissertation, The Open University, 1998. [Online]. Available: <https://oro.open.ac.uk/28579/>
- [28] M. H. Brown and J. Hershberger, “Color and Sound in Algorithm Animation,” in *IEEE Workshop on Visual Languages (VL 1991)*, 1991.
- [29] C. Armenti, M. Raglianti, and M. Lanza, “Sonifying and Visualizing the Heartbeat of Evolving Software Systems,” in *IEEE Working Conf. on Software Visualization (VISOFT 2025)*, 2025, pp. 1–12.
- [30] T. G. Moher, “PROVIDE: A Process Visualization and Debugging Environment,” *IEEE Trans. Software Eng.*, vol. 14, no. 6, 1988.
- [31] H. Lieberman and C. Fry, “Bridging the Gulf Between Code and Behavior in Programming,” in *ACM Conf. on Human Factors in Computing Systems (CHI 1995)*, 1995, pp. 480–486.
- [32] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, “Visualizing Programs with Jeliot 3,” in *Work. Conf. on Advanced Visual Interfaces (AVI)*, 2004, pp. 373–376.
- [33] P. V. Gestwicki and B. Jayaraman, “JIVE: Java Interactive Visualization Environment,” in *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications Companion (OOPSLA Companion 2004)*, 2004, pp. 226–228.
- [34] J. H. Cross, II and T. D. Hendrix, “jGRASP: A Lightweight IDE with Dynamic Object Viewers for CS1 and CS2,” in *Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*, 2006, p. 356.
- [35] P. J. Guo, “Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education,” in *44th ACM Technical Symposium on Computer Science Education (SIGCSE 2013)*, 2013, pp. 579–584. [Online]. Available: <https://doi.org/10.1145/2445196.2445368>
- [36] —, “Ten million users and ten years later: Python tutor’s design guidelines for building scalable and sustainable research software in academia,” in *34th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 2021, pp. 1235–1251. [Online]. Available: <https://doi.org/10.1145/3472749.3474819>
- [37] M. Weninger, S. Grünbacher, and H. Prähofer, “JavaWiz: A Trace-Based Graphical Debugger for Software Development Education,” in *33rd IEEE/ACM International Conf. on Program Comprehension (ICPC 2025)*, 2025, pp. 1–12. [Online]. Available: <https://doi.org/10.1109/ICPC66645.2025.00023>
- [38] H. Biermann and R. Cole, “Comic Strips for Algorithm Visualization,” New York University, Tech. Rep. 1999-778, 1999. [Online]. Available: <https://cs.nyu.edu/media/publications/TR1999-778.pdf>
- [39] E. Carson, I. Parberry, and B. Jensen, “Algorithm Explorer: Visualizing Algorithms in a 3D Multimedia Environment,” in *38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, I. Russell, S. M. Haller, J. D. Dougherty, and S. H. Rodger, Eds. ACM, 2007, pp. 155–159. [Online]. Available: <https://doi.org/10.1145/1227310.1227367>
- [40] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, “A Systematic Survey of Program Comprehension through Dynamic Analysis,” *IEEE Trans. Software Eng.*, vol. 35, no. 5, pp. 684–702, 2009. [Online]. Available: <https://doi.org/10.1109/TSE.2009.28>
- [41] M. Weninger, L. Makor, and H. Mössenböck, “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor,” in *IEEE Working Conf. on Software Visualization (VISOFT 2020)*, 2020, pp. 110–121.
- [42] —, “Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations,” in *Conf. on Smart Tools and Applications in Graphics (STAG 2020)*, 2020, pp. 63–75.
- [43] T. Herber and M. Weninger, “Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education,” in *IEEE Working Conf. on Software Visualization (VISOFT 2025)*, 2025, pp. 13–24.
- [44] M. Burgstaller, “Interactive and playful visualizations of graph algorithms,” Bachelor’s thesis, Johannes Kepler University Linz, Linz, Austria, Aug. 2025. [Online]. Available: https://ssw.jku.at/Teaching/BachelorTheses/2025/Burgstaller_Marcel.pdf
- [45] J. T. Stasko, “Using Student-Built Algorithm Animations as Learning Aids,” in *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1997)*, 1997, pp. 25–29.
- [46] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, “A meta-study of algorithm visualization effectiveness,” *J. Vis. Lang. Comput.*, vol. 13, no. 3, pp. 259–290, 2002. [Online]. Available: <https://doi.org/10.1006/jvlc.2002.0237>
- [47] M. Hertz and M. Jump, “Trace-based teaching in early programming courses,” in *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2013, pp. 561–566. [Online]. Available: <https://doi.org/10.1145/2445196.2445364>
- [48] N. Myller, R. Bednarik, E. Sutinen, and M. Ben-Ari, “Extending the engagement taxonomy: Software visualization and collaborative learning,” *ACM Trans. Comput. Educ.*, vol. 9, no. 1, pp. 7:1–7:27, 2009. [Online]. Available: <https://doi.org/10.1145/1513593.1513600>
- [49] N. Myller, “Automatic generation of prediction questions during program visualization,” in *Program Visualization Workshop (PVW)*, G. Röbling, Ed., vol. 178. Elsevier, 2006, pp. 43–49. [Online]. Available: <https://doi.org/10.1016/j.entcs.2007.01.034>
- [50] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available: <https://doi.org/10.1007/BF01386390>
- [51] T. Lehtinen, A. L. Santos, and J. Sorva, “Let’s ask students about their programs, automatically,” in *IEEE/ACM International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 467–475. [Online]. Available: <https://doi.org/10.1109/ICPC52881.2021.00054>