GraphAlg Playground: An Online Platform for Learning and Experimenting with the GraphAlg Language

Daan de Graaf
d.j.a.d.graaf@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

Robert Brijder
r.brijder@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

Soham Chakraborty
s.s.chakraborty@tudelft.nl
Delft University of Technology
Delft, Netherlands

George Fletcher
g.h.l.fletcher@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

Bram van de Wall
a.a.g.v.d.wall@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

Nikolay Yakovets
hush@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

ABSTRACT

The GraphAlg language for graph algorithms enables native support for user-defined graph analytics workloads in databases. In this demonstration, we present a web-based playground for writing and executing GraphAlg programs in the web browser, including an interactive tutorial explaining its key concepts. The playground runs inside the user's web browser without any installation, and is freely available under a permissive license as a reusable library. We present two demonstration scenarios of the publicly available playground website, showing how new users can learn to program in GraphAlg using the tutorial, while expert users can use the playground to prototype and validate their algorithms.

PVLDB Reference Format:
Daan de Graaf, Robert Brijder, Soham Chakraborty, George Fletcher, Bram van de Wall, and Nikolay Yakovets. GraphAlg Playground: An Online Platform for Learning and Experimenting with the GraphAlg Language. PVLDB, 19(12): 4546 - 4549, 2026.
doi:10.14778/3827998.3828062

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://wildarch.dev/graphalg.

1 INTRODUCTION

Consider a data scientist analyzing citation networks to identify influential papers using PageRank, or a fraud analyst running community detection on transaction graphs. Despite graph databases being the natural home for such data, these users cannot express these fundamental algorithms in Cypher [4] or GQL [12]. Instead, they must export gigabytes of data and wrestle with format conversions. This "data wrangling" wastes engineering effort and risks correctness when copies diverge from the source of truth.

Existing approaches to graph analytics support in databases suffer from various problems. Algorithm packages such as the Neo4j Graph Data Science Library [15] offer only fixed implementations that rarely match exact requirements (e.g., Neo4j's PageRank lacks sink redistribution). Other systems integrate general-purpose programming languages or separate algorithm languages [15, 18] for maximum flexibility, but this impedes cost-based optimization inside algorithms. Another popular approach is to extend existing query languages with recursion [6, 8, 9, 14]. This approach inherits the cost-based optimizations from the original query languages, but leaves high-level optimization across loop iterations to the programmer. Finally, the existing systems that both support loops *and* apply comprehensive optimization [10, 16, 17] are standalone solutions that do not integrate with graph data management systems. Figure 2 provides a visual overview of the database-native approaches according to their expressive power and the scope of automated optimization.

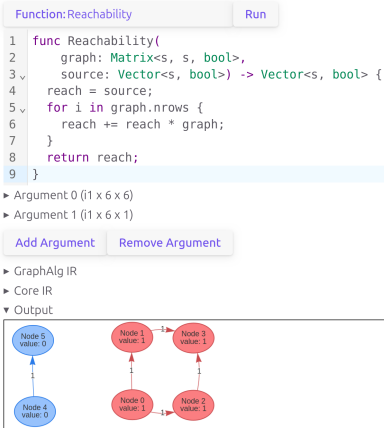

Figure 1: A GraphAlg program in the playground.

GraphAlg [3] is a domain-specific language for graph algorithms, based on linear algebra, that addresses this gap. GraphAlg satisfies four critical requirements: (1) **Expressive**: users implement arbitrary algorithms by composing matrix operations; (2) **Designed for graph algorithms**: programs require 2–10× less code than SQL or Java; (3) **Fully integrated**: algorithms embed directly in queries and execute on the native graph storage; (4) **Optimizable**: the GraphAlg compiler performs loop optimization, after which programs compile to query plans to leverage existing database optimizations. We integrated GraphAlg into the AvantGraph [22]

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828062

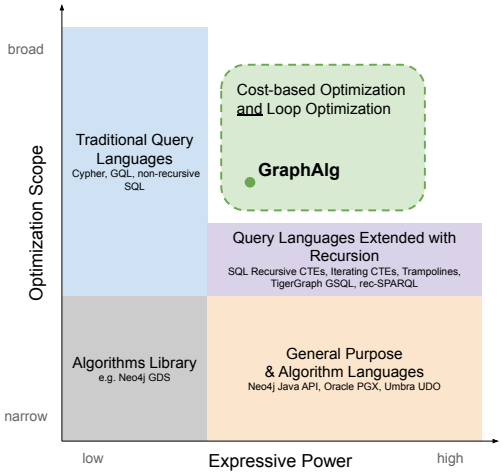

Figure 2: Existing approaches to in-database graph analytics along two dimensions: the complexity of algorithms they support (horizontal) and how automatically they choose efficient execution strategies (vertical). GraphAlg aims toward the upper-right corner, carefully balancing expressive power with a high degree of automated optimization.

Table 1: Features of online playgrounds for several programming languages and database systems.

	Go	Rust	Umbra	DuckDB	GraphAlg
Execution	Server-side			**Client-side (WASM)**	
Syntax Highlighting	No	**Yes (Client-side)**			
Compiler Diagnostics	When clicking run button				**Real-time**
Error Highlighting	No				**Yes**
Result Rendering	Text (stdout)				**Graph viz.**

graph data management system, where it outperforms DuckDB and Neo4j on PageRank, single-source shortest paths, and connected components benchmarks [3]. We implemented a diverse set of centrality, community detection and path finding algorithms in GraphAlg widely considered as representative for graph analysis in general [11].

To streamline the process of learning the language, we propose the GraphAlg *playground*: an online platform for learning and experimenting with GraphAlg. We adopt a fully client-side implementation based on WebAssembly [7] that eliminates network overhead, enabling real-time linting and error diagnostics. Table 1 compares the GraphAlg playground to other online interfaces for programming languages and database systems where users can write and execute programs from their browser without installing any software [5, 19–21]. The playground supports the full GraphAlg language, so any valid GraphAlg program runs in it; its only limitation is an unoptimized WebAssembly runtime, making it suited to learning and small-scale experimentation rather than the production workloads AvantGraph handles.

The GraphAlg playground includes an *interactive* tutorial that explains key language concepts, starting from basic syntax and gradually increasing complexity up to complete, practical algorithms such as PageRank. Every part of the tutorial is accompanied

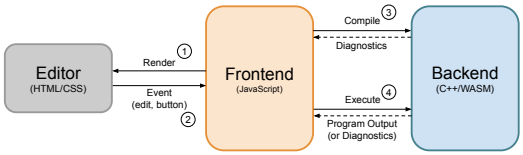

Figure 3: Overview of the GraphAlg Playground Architecture.

by interactive GraphAlg code snippets that can be modified and executed directly from the user's web browser.

The GraphAlg playground also serves experienced programmers already familiar with the language. The same editor with syntax highlighting and real-time linting provides a platform for developing new algorithms. Programmers can upload input data (e.g., a small test graph), run the algorithm, and visually inspect output matrices rendered as graphs. This rapid feedback loop accelerates algorithm prototyping without requiring a full database setup.

The entire playground, including the compiler, runtime, and tutorial, is freely available under a permissive open-source license [2].

Our demonstration consists of two scenarios: (1) Using the playground as an experienced GraphAlg programmer to modify and test an algorithm, and (2) Working through the tutorial as a developer new to GraphAlg.

2 GRAPHALG IN A NUTSHELL

GraphAlg is a domain-specific language for graph algorithms based on linear algebra. Matrix operations naturally express graph computation: multiplying a vector of "reached" vertices by the adjacency matrix yields the vertices reachable in one hop. This correspondence facilitates concise definitions of algorithms.

The language provides high-level operations (matrix multiplication, aggregation, element-wise application) with an imperative style similar to Python. For example, the reachability algorithm shown in Figure 1 is just a loop that repeatedly multiplies the frontier vector by the adjacency matrix and accumulates results. A powerful feature is support for *semirings* [13], which generalize arithmetic operations: using the tropical semiring (with min for addition and + for multiplication) transforms a reachability algorithm into single-source shortest paths with minimal code changes. This expressiveness allows GraphAlg to implement algorithms including PageRank, connected components, community detection, and breadth-first search.

Programs compile to relational algebra via a formal core derived from MATLANG [1], enabling integration into existing query pipelines. In AvantGraph, GraphAlg programs embed directly in Cypher queries and are optimized together with the surrounding query, enabling cross-boundary optimizations impossible in systems with separate algorithm pipelines.

3 PLAYGROUND ARCHITECTURE

The GraphAlg playground architecture (Figure 3) consists of two components: a *Frontend* (JavaScript) that renders interactive code

editors and handles user input, and a *Backend* (C++/WebAssembly) that parses, compiles, and executes GraphAlg programs.

Written in JavaScript, the Frontend is included as a script in an HTML document. Upon loading, it scans the document for GraphAlg code markers and (1) renders interactive editors in their place. This approach allows the tutorial to be written in Markdown, with code snippets automatically becoming interactive editors. The Frontend also downloads and initializes the Backend.

The Backend is compiled to WebAssembly so it runs entirely client-side, avoiding the need for server infrastructure. This eliminates hosting costs and scalability concerns, and keeps user code fully private since it never leaves the browser. The Backend uses the same open-source GraphAlg compiler library [2] as AvantGraph, ensuring programs validated in the playground work identically in production. We illustrate the components working together through two key features.

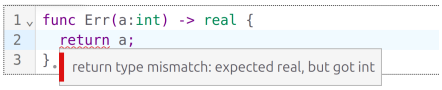

Figure 4: An error diagnostic in the code editor, generated by type checking in the GraphAlg compiler.

Real-time Compiler Feedback. When the user edits code (2), the Frontend requests the Backend to (3) compile the program. If parsing or type checking fails, the Backend returns error diagnostics with source locations. The Frontend (1) underlines errors in the editor; hovering over the underlined text displays the error message in a tooltip (Figure 4). Because compilation happens locally via WebAssembly, feedback is instantaneous. The GraphAlg type system catches errors such as dimension mismatches at compile time; remaining runtime errors are reported with source locations.

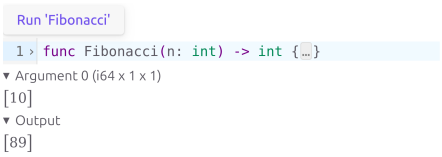

Figure 5: Code editor with rendered argument and output.

Algorithm Execution in the Browser. When the user clicks the *Run* button (2), the Frontend collects input arguments and requests the Backend to (4) execute the program. The Backend compiles and runs the program, returning output to the Frontend. The Frontend (1) renders results below the code (Figure 5). Matrix outputs can be visualized as a graph, making algorithm behavior easy to inspect, and users can upload their own graph to test algorithms on custom inputs.

4 DEMONSTRATION

Our demonstration comprises two scenarios highlighting different use cases for the GraphAlg playground. The first scenario shows developing and testing a GraphAlg algorithm in the playground, then deploying it on a large graph in a production-like setup.

In the second (optional) scenario, attendees work through exercises in the GraphAlg tutorial, which is publicly available and fully web-based, so participants can also continue on their own device later.

4.1 Modifying, Testing and Deploying an Algorithm using the Playground

Alex maintains a scientific knowledge graph with tens of millions of citations using the AvantGraph [22] graph data management system. To find high-impact publications in the graph, they apply PageRank to the citation graph, implemented as a GraphAlg program embedded in a Cypher query.

Alex wants to extend their PageRank implementation to redistribute scores from sink nodes (a common requirement that built-in algorithm libraries do not satisfy [3]). They take the following steps to modify the existing program, test the behavior of the modified program, and finally deploy it on the large graph.

Step 1. Alex opens their browser and navigates to the playground at wildarch.dev/graphalg/playground.

Step 2. Alex uploads a small test graph to the playground (Figure 6).

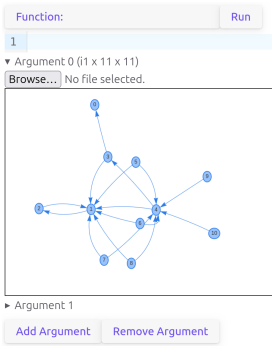

Figure 6: Alex uploads a file containing a small graph to the Playground.

Step 3. Alex writes a sink detection program to verify the test graph has sink nodes; clicking *Run*, the output visualization confirms a sink exists.

Step 4. Alex pastes the original PageRank program into the editor and runs it, noting the score for a well-connected node n; summing the computed scores yields less than one, confirming the rank sink issue.

Step 5. Alex adds sink redistribution logic but forgets a required type cast; real-time type checking immediately underlines the error, which they correct.

Step 6. Alex runs the modified program: the score for node n rises, and the scores now sum to one, confirming correct redistribution.

Step 7. Confident the algorithm is correct, Alex updates the Cypher query and runs it on the large citation graph (Figure 7).

4.2 Learning GraphAlg with the Tutorial

Beau wants to learn GraphAlg and navigates to the tutorial at wildarch.dev/graphalg/tutorial. They work through the twelve-part

```
query = f"""
WITH ALGORITHM "{algorithm}"
CALL PageRank("Product", "CITES")
RETURN row.title AS publication_title, val AS pagerank_score
ORDER BY val DESC
LIMIT 10
"""
execute_query(query)
```

[11]:

	publication_title	pagerank_score
0	Crystalline Serotonin	0.000564
1	The impulses produced by sensory nerve endings	0.000560
2	A Fluorimetric Method for the Determination of...	0.000503
3	SERUM VASOCONSTRICTOR (SEROTONIN)	0.000503
4	Tumor Detection by Nuclear Magnetic Resonance	0.000498
5	The concentration of sympathin in different pa...	0.000496

Figure 7: Alex executes the PageRank algorithm on the large knowledge graph by sending a Cypher query embedding the algorithm to AvantGraph.

tutorial, interacting with code snippets accompanying each concept. For example, when learning about the Fibonacci sequence (Figure 8), the tutorial suggests modifying the program. Beau experiments by changing the loop bound and observing how results change. When they make a mistake, the editor immediately highlights the error. *Note: Attendees can try the first parts during the session, and complete it later on their own device.*

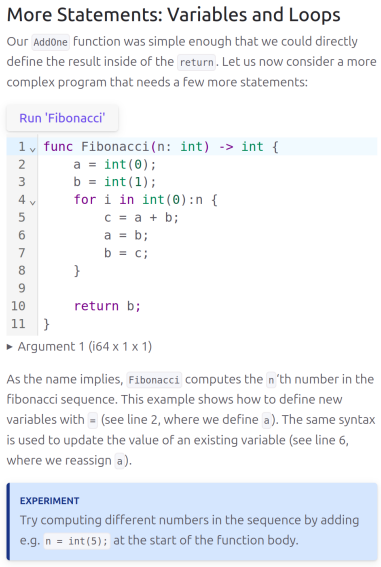

Figure 8: A section of the GraphAlg tutorial with an interactive code snippet and a suggested experiment.

.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon Europe framework programme under grant agreement No. 101058573 as part of the SciLake project.

REFERENCES

[1] Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. 2019. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.* 44, 4 (Oct. 2019), 15:1–15:31. https://doi.org/10.1145/3331445

[2] Daan de Graaf. 2026. wildarch/graphalg. https://github.com/wildarch/graphalg Last accessed: 2026-06-23.

[3] Daan de Graaf, Robert Brijder, Soham Chakraborty, George Fletcher, Bram van de Wall, and Nikolay Yakovets. 2026. Algorithm Support for Graph Databases, Done Right. https://doi.org/10.48550/arXiv.2601.06705 arXiv:2601.06705 [cs].

[4] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[5] Google Inc. [n.d.]. Go Playground - The Go Programming Language. https://go.dev/play/ Last accessed: 2026-06-23.

[6] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural extensions of SQL: understanding their usage in the wild. *Proceedings of the VLDB Endowment* 14, 8 (April 2021), 1378–1391. https://doi.org/10.14778/3457390.3457402

[7] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[8] Denis Hirn and Torsten Grust. 2023. A Fix for the Fixation on Fixpoints. In *Proceedings of the 13th Conference on Innovative Data Systems Research*.

[9] Aidan Hogan, Juan L. Reutter, and Adrián Soto. 2020. In-Database Graph Analytics with Recursive SPARQL. In *The Semantic Web – ISWC 2020*, Jeff Z. Pan, Valentina Tamma, Claudia d'Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne, and Lalana Kagal (Eds.). Springer International Publishing, Cham, 511–528. https://doi.org/10.1007/978-3-030-62419-4_29

[10] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. 1–10. https://doi.org/10.1145/3070607.3070608 arXiv:1703.07342 [cs].

[11] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[12] ISO. 2024. Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html

[13] Jeremy Kepner and John Gilbert (Eds.). 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719918

[14] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. 2016. G-SQL: fast query processing via graph exploration. *Proceedings of the VLDB Endowment* 9, 12 (Aug. 2016), 900–911. https://doi.org/10.14778/2994509.2994510

[15] Neo4J inc. [n.d.]. Graph algorithms - Neo4j Graph Data Science. https://neo4j.com/docs/graph-data-science/2.17/algorithms/ Last accessed: 2026-06-23.

[16] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Q. Ngo. 2024. Optimizing Nested Recursive Queries. *Proc. ACM Manag. Data* 2, 1 (2024), 16:1–16:27. https://doi.org/10.1145/3639271

[17] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *2015 IEEE 31st International Conference on Data Engineering*. 867–878. https://doi.org/10.1109/ICDE.2015.7113340 ISSN: 2375-026X.

[18] Moritz Sichert and Thomas Neumann. 2022. User-defined operators: efficiently integrating custom algorithms into modern databases. *Proceedings of the VLDB Endowment* 15, 5 (Jan. 2022), 1119–1131. https://doi.org/10.14778/3510397.3510408

[19] The DuckDB Developers. [n.d.]. DuckDB Web Shell. https://shell.duckdb.org/ Last accessed: 2026-06-23.

[20] The Rust Developers. [n.d.]. Rust Playground. https://play.rust-lang.org/ Last accessed: 2026-06-23.

[21] The Umbra Developers. [n.d.]. Umbra Online Interface. https://umbra-db.com/interface Last accessed: 2026-06-23.

[22] Wilco van Leeuwen, Thomas Mulder, Bram van de Wall, George Fletcher, and Nikolay Yakovets. 2022. AvantGraph query processing engine. *Proceedings of the VLDB Endowment* 15, 12 (Aug. 2022), 3698–3701. https://doi.org/10.14778/3554821.3554878