

1-27-2023

CodeBase Relationship Visualizer: Visualizing Relationships Between Source Code Files

Jesse Hines
jessehines@southern.edu

Follow this and additional works at: https://knowledge.e.southern.edu/mscs_reports



Part of the [Graphics and Human Computer Interfaces Commons](#), and the [Systems Architecture Commons](#)

Recommended Citation

Hines, Jesse, "CodeBase Relationship Visualizer: Visualizing Relationships Between Source Code Files" (2023). *MS in Computer Science Project Reports*. 7.
https://knowledge.e.southern.edu/mscs_reports/7

This Thesis is brought to you for free and open access by the School of Computing at Knowledge Exchange. It has been accepted for inclusion in MS in Computer Science Project Reports by an authorized administrator of Knowledge Exchange. For more information, please contact jspears@southern.edu.

CODEBASE RELATIONSHIP VISUALIZER: VISUALIZING RELATIONSHIPS
BETWEEN SOURCE CODE FILES

by

Jesse Hines

A THESIS

Presented to the Faculty of
The School of Computing at Southern Adventist University
In Partial Fulfilment of Requirements
For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Dr. Halterman

Collegedale, Tennessee

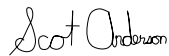
January, 2023

CODEBASE RELATIONSHIP VISUALIZER: VISUALIZING
RELATIONSHIPS BETWEEN SOURCE CODE FILES

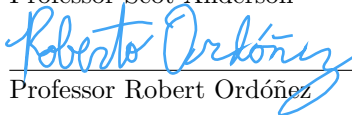
Approved by:



Professor Richard Halterman, Adviser



Professor Scot Anderson



Professor Robert Ordóñez

Date Approved 1/27/2023

CODEBASE RELATIONSHIP VISUALIZER: VISUALIZING RELATIONSHIPS BETWEEN SOURCE CODE FILES

Jesse Hines, M.S.

Southern Adventist University, 2023

Adviser: Richard Halterman, Ph.D.

Understanding relationships between files and their directory structure is a fundamental part of the software development process. However, it can be hard to grasp these relationships without a convenient way to visualize how files are connected and how they fit into the directory structure of the codebase. In this paper we describe CodeBase Relationship Visualizer (CBRV), a Visual Studio Code extension that interactively visualizes the relationships between files. CBRV displays the relationships between files as arrows superimposed over a diagram of the codebase's directory structure. CBRV comes bundled with visualizations of the stack trace path, a dependency graph for Python codebases, and a hyperlink graph for HTML and Markdown. CBRV also exposes an API that can be used to create visualizations for multiple different relationships. CBRV is a convenient and easy-to-use tool that offers a "big picture" perspective on the relationships within a codebase.

Contents

Contents	v
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Problem Statement	3
1.2 Project Overview	3
1.3 Motivation	5
1.4 Outline	5
2 Background	7
2.1 The Power of Interactive Visualizations	7
2.2 Similar Tools	8
2.2.1 Algorithm Visualizations	8
2.2.2 Relationship Visualizations	9
3 Implementation & Design	11
3.1 The Visualization	12
3.1.1 Directory Structure Diagram	12

3.1.2	Connections	13
3.1.3	Interaction	14
3.1.4	Include & Exclude	15
3.2	The API	17
3.3	Minimal Implementation	17
3.4	Stack Trace Visualization	17
3.5	Dependency Visualization	20
3.6	Hyperlink Visualization	20
3.7	Architecture	22
4	Tasks Delineation & Deliverables	25
5	Testing & Evaluation	27
6	Future Work	29
7	Conclusion	31
A	Detailed Requirements	33
A.1	Non-Functional	33
A.2	Functional	34
A.2.1	The Visualization	34
A.2.1.1	Directory Structure Diagram	34
A.2.1.2	Connections	35
A.2.1.3	Interaction	37
A.2.1.4	Include/Exclude	38
A.2.2	API	39
A.2.3	Minimal Implementation	39

A.2.4	Stack Trace Visualization	39
A.2.5	Dependency Visualization	41
A.2.6	Hyperlink Visualization	42
B	Acceptance Testing	43
B.1	Sample codebases	43
B.2	Acceptance Tests	44
B.2.1	Basic diagram	44
B.2.2	Basic interaction	45
B.2.3	Filtering	47
B.2.4	Hyperlink Visualization and Connections	48
B.2.5	Real-time updates	50
B.2.6	Symlinks	50
B.2.7	Large Codebase	52
B.2.8	Stack Trace Visualization	52
B.2.9	Dependency Visualization	55
B.2.10	Empty Codebases	56
	Bibliography	57

List of Figures

3.1	The architecture of CBRV	11
3.2	A screenshot of the CBRV Hyperlink Visualization	12
3.3	A screenshot of dynamic zoom (on the VSCode docs [1] codebase)	14
3.4	A screenshot showing connection on hover (on the VSCode docs [1] codebase)	16
3.5	A screenshot of the CBRV Stack Trace Visualization	18
3.6	A screenshot of the CBRV Stack Trace Visualization with recursion shown as a merged connection with tooltip	19
3.7	A screenshot of the dependency visualization (on the ShellAdventure [2] codebase)	21

List of Tables

3.1	Summary of the protocol between the main extension and the webview	23
4.1	Summary of tasks for the CBRV project	26

Chapter 1

Introduction

You have just started a new job and been dropped into a massive codebase. There are thousands of files of source code, dozens of legacy systems, all interacting with each other in specific ways and you, of course, have no idea what is going on. But you are supposed to fix this bug. You dig through the code, running a debugger and trying to find where exactly everything is going all wrong. How does this piece of code relate to the others? You are in a stack trace, how exactly did we get here, and from where, and why? What parts of the code are relevant? Is this file over here actually used anywhere anymore? What is its purpose? And of course, you are terrified to change anything until you understand all these connections since a tiny change can easily affect all kinds of things.

Confusion when working on a new codebase is a nearly universal experience among software developers. Understanding relationships between systems is a fundamental part of the software development process. When working on any new codebase, trying to understand the relationships between all of its parts is vital, and often the most difficult part of the process [3]. A developer has to know which components could be affected by their changes. They need to have a mental

model of the codebase so that they can see the “big picture” and where this piece fits in with all the others in the intricate puzzle that is software. Even on a familiar codebase with a single developer, upon returning to a project after a year or so they will often encounter the same problem and have to relearn the relationships in the codebase.

An important, but often neglected, relationship in a codebase is the actual layout of its files and folders on the filesystem. Without a mental model of the directory structure of the codebase, a developer will most likely be plagued with import resolution issues, configuration issues, compiling issues, and more. Understanding the directory structure is necessary to navigate it and understand the overall organization and design of the codebase. Logical relationships need to be understood within the context of the directory structure of the codebase.

Fortunately, dynamic visualizations and diagrams are powerful tools to aid in the understanding of complex relationships in a codebase [4]. A good software visualization can help a developer in building a strong mental model of a codebase and the relationships within it. Hand-made architecture diagrams are great tools, but reality usually deviates from the original plan and a hand-made architecture diagram will fail to capture every relationship. Dynamically generated visualizations help overcome some of the weaknesses of pre-made visualizations. A dynamic visualization can show the actual state of the codebase, not just the original plan, and show more complex relationships and information, even including runtime information.

1.1 Problem Statement

Understanding relationships between files and their directory structure is a fundamental part of the software development process. We need a convenient way to visualize these relationships and how they fit within the directory structure of the codebase. Few existing visualizations place relationships between files within the context of their directory structure. To address this gap, we created CodeBase Relationship Visualizer (CBRV), a Visual Studio Code extension that interactively visualizes the relationships between files. CBRV displays the relationships between files as arrows superimposed over a diagram of the codebase's directory structure. CBRV comes bundled with visualizations of the stack trace path, a dependency graph, and a hyperlink graph for HTML and Markdown. CBRV also exposes an API that can be used to create visualizations for multiple different relationships. CBRV is a convenient and easy to use tool that offers a "big picture" perspective on the relationships within a codebase.

1.2 Project Overview

CBRV displays the directory structure of a codebase as nested circles in a circle-packing diagram, with folders represented by circles containing other folder and file circles. The directory structure diagram is overlaid with the relationships between the files, such as dependencies or references depending on the specific visualization, rendered as lines or arrows connecting the files. The circle-packing display of the filesystem and connections is inspired by Wattenberger's Repo Visualizer [5], but is interactive, zoomable, and adaptable to any type of relationship between files.

CBRV is distributed as an extension for Microsoft's popular editor/IDE Visual Studio Code [6] and available for install from the VSCode extension marketplace [7]. The CBRV source is published under the GNU GPL and available on GitHub [8]. CBRV includes 4 bundled visualizations which make use of the the CBRV API:

1. Minimal Implementation

This visualization is a minimal example of using the CBRV API. It has no connections, and just shows the file structure diagram.

2. Stack Trace Visualization

This visualization shows the path of the stack trace through the codebase. It connects to a debugging session in any language that has a VSCode debugger. It shows a line tracing the call stack through the files on top of the CBRV diagram at each breakpoint. The goal of this visualization is to help orient a developer while debugging with a deep stack trace, to show an overview of what path the code followed to get here.

3. Dependency Visualization

This visualization shows a dependency graph of imports in Python codebases, visualizing which parts of the codebase depend on each other.

4. Hyperlink Visualization

This visualization shows a graph of the hyperlinks connecting HTML and/or Markdown files.

1.3 Motivation

There are of course many existing tools to visualize various relationships in a codebase. CBRV adds a new entry to the repertoire of visualization tools for developers, and complements the existing tools. We think that it offers a unique perspective of a codebase, allowing developers to visualize the codebase's directory tree at the same time as the relationships in the code.

We also want CBRV to be easily accessible and easy to use. While there are hundreds of fascinating software visualizations out there, getting one to run on a modern system is a much more daunting task [9]. A visualization is of no use if no one can figure out how to run it anymore. Making CBRV open source and available on the VSCode marketplace will make it easy for anyone to install. And exposing the API will make it easily adaptable to visualize new relationships.

1.4 Outline

This paper reviews the existing tools for software visualization in [Chapter 2](#) and describes our implementation and design in [Chapter 3](#). [Chapter 4](#) lists the project tasks and the final deliverables presented. [Chapter 5](#) outlines the test results for the project. [Chapter 6](#) describes directions for future work and [Chapter 7](#) presents our conclusions. Detailed requirements for the CBRV visualization and acceptance testing can be found in [Appendices A](#) and [B](#) respectively.

Chapter 2

Background

There is lots of research on how visualizations helps in program comprehension and on various software visualization tools. We will briefly look at the theory behind software visualization, and relevant software visualization tools.

2.1 The Power of Interactive Visualizations

Sight, such as examining source code and diagrams, is the primary means that software developers use to understand a codebase [3]. Creating and studying diagrams of a codebase can greatly aid in understanding its overall structure. However, software visualizations work best when their users can interact and engage with them [10]. Passively watching a visualization rarely helps unless users engage in the process [11]. Additionally, combining and overlaying different styles of visualizations into one can be a powerful tool, allowing users to see a new perspective on how the different aspects interact [4]. Making interactive and responsive visualizations that users can play around with will greatly increase their usefulness as comprehension aids.

2.2 Similar Tools

The concept of making software visualizations is certainly not a new one. CBRV builds upon and adds to the existing set of tools developers can use to visualize their software.

2.2.1 Algorithm Visualizations

Visualizations of computer algorithms date back at least to 1975 when Baecker described his system for making “teaching films containing animated representations of the execution of computer programs” [12]. Since then the field has grown expansively, with countless tools providing different visualizations of software, sporting visualizations ranging from 3D program flowcharts [13] to a visual debugger based on Space Invaders [14].

Existing tools for visualizing and debugging algorithms include DataDisplay-Debugger [15], which can connect to multiple different debuggers for different languages and shows data structures as graphs. Another powerful algorithm visualization tool is visualgo.net, which contains dozens of online interactive visualizations of different algorithms such as quick sort or binary tree insertion [16]. If visualgo does not have a pre-made visualization for an algorithm, algorithm-visualizer.org allows users to create algorithm visualizations by instrumenting their code [17].

Dieterichs’ Debug Visualizer [18] is also a fascinating algorithm visualization. Like CBRV, Debug Visualizer is a VSCode extension. It shows various data structures graphically in a side pane while debugging a program in VSCode. Debug Visualizer can show multiple different visualizations, graphs of linked data structures, array data structures, as well as line, scatter plots, and tables. Debug

Visualizer has full support for Javascript and TypeScript, but can be made to work in most other languages by instrumenting code with JSON formatted calls to its API.

2.2.2 Relationship Visualizations

More directly related to CBRV, there are many visualizations of the relationships between software components. TraceVis by Pieter Deelen [19] shows dynamic interactions between Java classes, a very similar concept to CBRV, especially considering that classes and files are nearly synonymous in Java. TraceVis shows both static and runtime information such as how often one class uses another and which classes construct others. It displays all this using a graph visualization. TraceVis is limited to Java applications, and does not appear to have been updated to work on anything past Java 5. A very similar tool to TraceVis was presented by Bertuli et. al. [20] which visualizes runtime relationships between object oriented classes with graphs and charts. Another tool, “Program Explorer” [21] also visualizes class hierarchy, function calls, variable access, and object usage for large C++ programs by combining static and runtime information.

Sourcetrail [22] is a useful open source, cross platform “source explorer.” Sourcetrail can analyze C, C++, and Java codebases. It only takes into account static information, but can show interactive and navigable inheritance trees, call graphs, and more. Unfortunately, Sourcetrail was discontinued in the latter part of 2021 and is currently unmaintained.

Of particular note is Wattenberger’s Repo Visualizer [5]. Repo Visualizer is a GitHub action which generates a static SVG of a GitHub repository as a circle packing diagram. This diagram allows developers to see the overall directory

structure of a codebase at a glance, and easily see information such as which folders have the most content, what types of files are in the repository and where they are. Wattenberger also described displaying dependencies between files as potential future work. CBRV was heavily inspired by Repo Visualizer but makes an interactive visualization that can be run on any codebase and adds the ability to portray many different relationships using a similar visualization style.

CBRV adds a new tool to the developer's toolbox of software visualizations, offering a different perspective on the structure and relationships in a code base, and offering visualizations for multiple different relationships via the CBRV API. Additionally, making CBRV a VSCode extension greatly increases its availability and ease of install and use compared to most existing software visualizations.

Chapter 3

Implementation & Design

The CBRV project consists of the CBRV API, which renders the directory structure diagram and the connections between the files, and 4 bundled visualizations using the CBRV API: one to visualize just the file structure, one to visualize a stack trace, one to visualize dependencies, and one to visualize a hyperlink graph. Third party extensions can also use the CBRV API to create visualizations of more relationships. A simple architecture diagram of CBRV can be seen in [Figure 3.1](#).

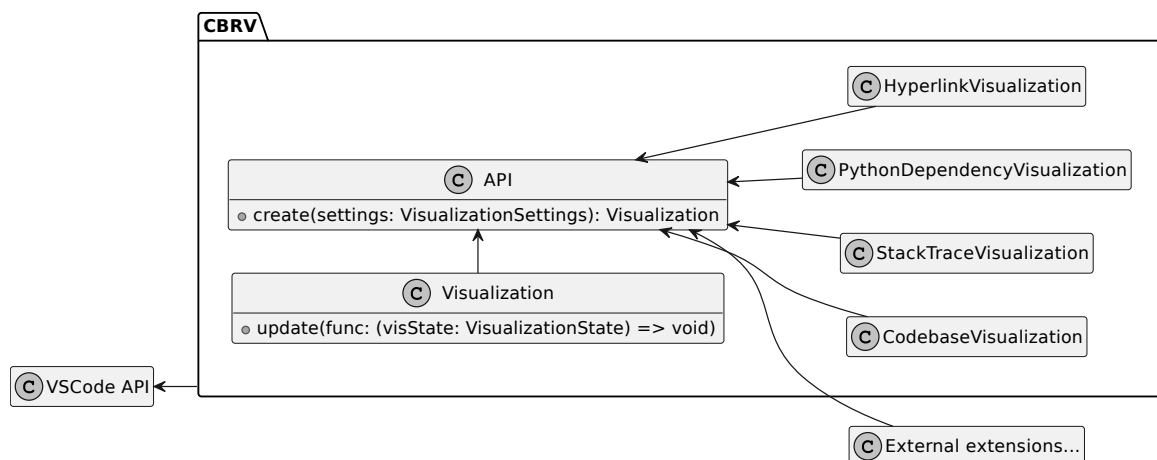


Figure 3.1: The architecture of CBRV

3.1 The Visualization

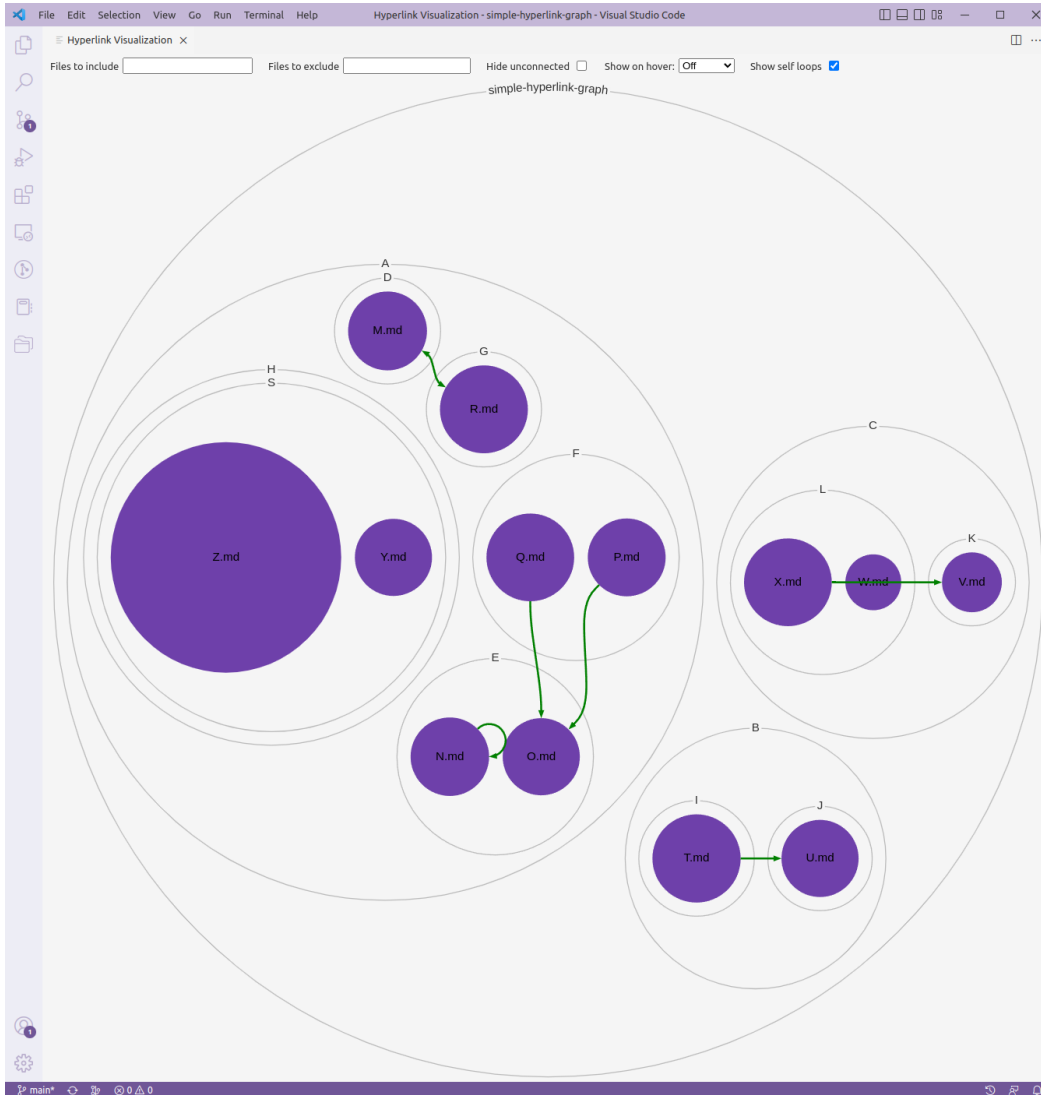


Figure 3.2: A screenshot of the CBRV Hyperlink Visualization

3.1.1 Directory Structure Diagram

The base of the visualization is the diagram of the codebase directory structure, which displays the directory structure of a codebase as nested circles in a padded circle-packing diagram, as seen in [Figure 3.2](#). Folders are represented by circles

containing other folder and file circles, such that each file in the codebase is a “leaf node” of the diagram. Each folder and file is labeled with their filename if there is room on the visualization. If there is not enough space, the name is be omitted or cropped to fit. This diagram displays from the root of the currently open VSCode workspace.

Like in RepoVisualizer [5], other attributes of the circles are used to visualize attributes of the files. Files are color-coded by their file extension, allowing for a quick overview of the distribution of different file types in the codebase. Each circle’s area is proportional to its respective file’s size, though clamped to a minimum and maximum bound to avoid tiny and empty files being nearly invisible or a very large file crowding out everything else.

3.1.2 Connections

An arbitrary set of connections can be defined via the API, representing dependencies, references, or some other relationship. Each connection has a source and a destination file and can optionally reference line numbers within the source and destination files as well. CBRV only supports connections between files, not connections between directories. The directory structure diagram is overlaid with these connections, rendered as lines connecting their respective files. The connections can be either directed, which will be rendered as an arrow, or undirected, which will be rendered as a simple line. The connections can also have color, weight, and a tooltip defined.

Connections are able to “go out of the diagram” or be “self loops”. If a connection only has one of source or destination defined, the rendered connection will start from the edge of the screen and then connect to a single file. This can

be useful to indicate the starting location in a stack trace (as in [Figure 3.5](#)) or a reference to something external to the current codebase. If a connection's source and destination are the same file it is rendered as a self loop on the diagram (as seen on file `N.md` in [Figure 3.2](#)).

If there are multiple connections between the same two files those connections may be merged. The API can set which connections to merge, whether connections going opposite directions into a double-arrowed connection, and connection weight and color (See `VisualizationSettings.mergeRules` in the API docs [\[23\]](#)).

3.1.3 Interaction

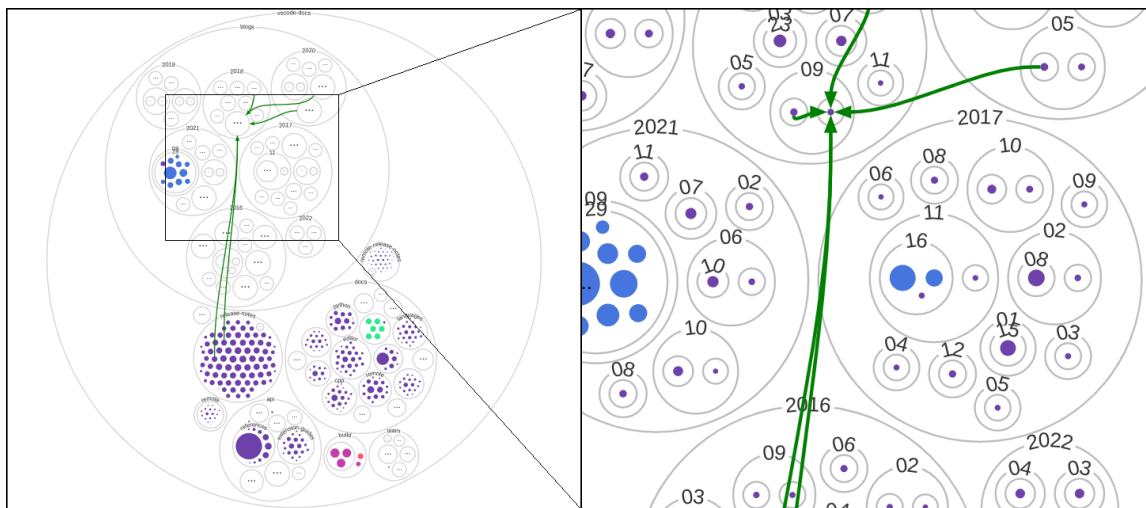


Figure 3.3: A screenshot of dynamic zoom (on the VSCode docs [\[1\]](#) codebase)

CBRV is an interactive visualization, allowing users to get much more information out of it than with a static visualization. The diagram is zoomable and draggable, using the mouse wheel to zoom and click and drag to pan. If the directory structure is large and deeply nested the visualization may not be able to display all the files at once. In this case, the visualization dynamically hides

the content of the deeply nested folders. As seen in [Figure 3.3](#), when the user zooms in the hidden content is displayed as room becomes available on the screen. Connections to or from a file that is hidden because of high levels of nesting instead connect to the first ancestor that is visible.

Symlinks within the codebase are handled by making the circle for the symlink show a symlink icon, and clicking it highlights and jump the screen to the resolved file.

Each visualization using the CBRV API can provide context aware information by adding custom tooltips on the connections. Additionally, each folder and file has a tooltip which displays the path to the folder relative to the workspace root. The visualization can also configure settings to only show connections when hovering over a connected file. An example of this can be seen in [Figure 3.4](#). This is important if a visualization has a large number of connections, as the visualization can easily get too crowded to be helpful.

Double-clicking on a file circle opens that file in a VSCode editor tab so that the user can view and edit it. Double-clicking a folder circle shows that folder in the VSCode file explorer. There is also a right-click context menu available on folders and circles. It has the options “Open in new Editor” (for files), “Reveal in Explorer”, “Copy Path”, and “Copy Relative Path” by default. Each visualization can add additional items to the context menu if desired.

3.1.4 Include & Exclude

The visualization includes “Include” and “Exclude” inputs and a “Hide Unconnected” checkbox. These are used to filter the files and folders displayed in the visualization. Users can type in a comma separated list of file paths with wildcards

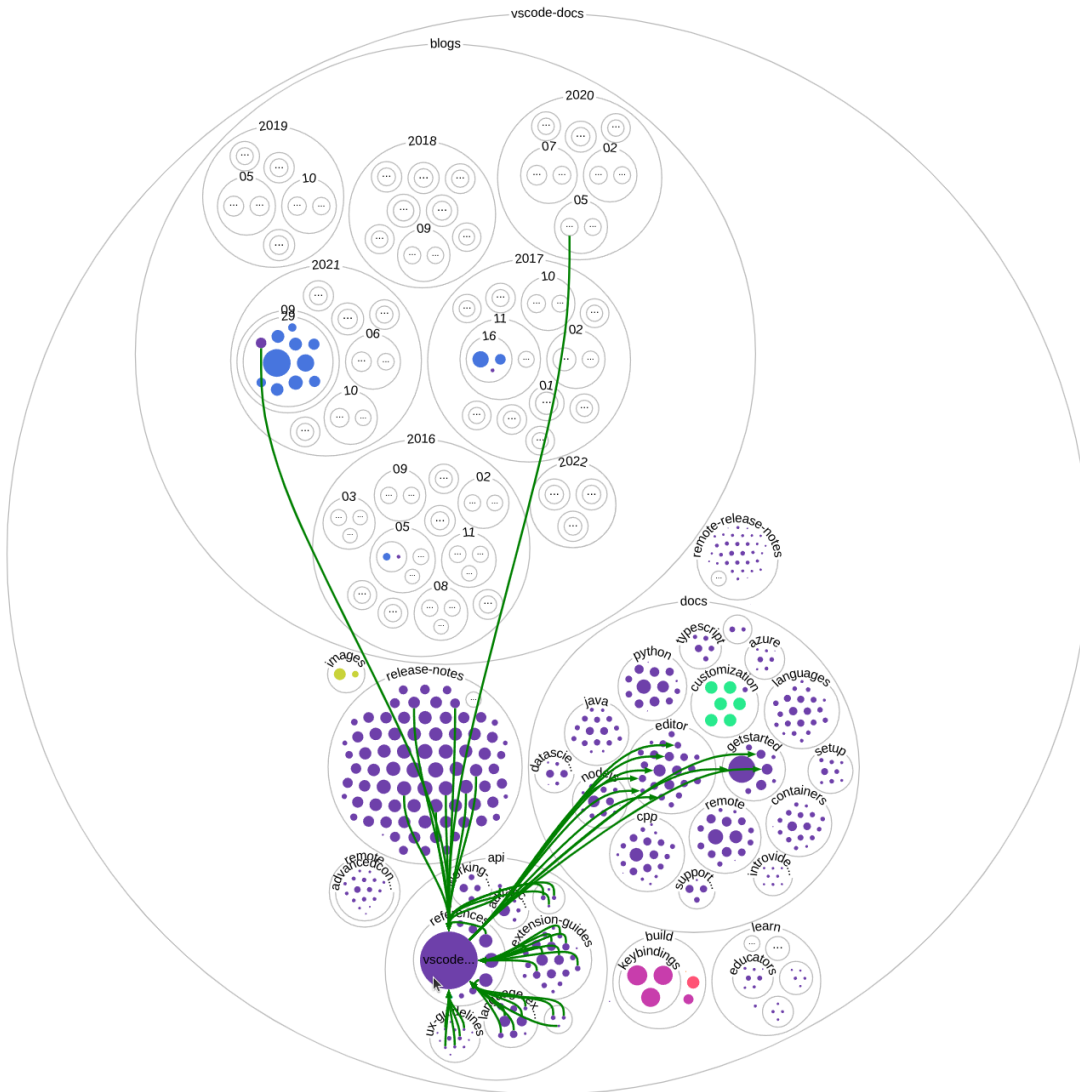


Figure 3.4: A screenshot showing connection on hover (on the VSCode docs [1] codebase)

and the visualization automatically updates. The “Hide Unconnected” checkbox toggles whether to show all files or only ones with connections to them.

Connections that reference an excluded file are not shown in the visualization. And, if all the files in a folder are excluded, that folder is not shown in the visualization either.

3.2 The API

VSCoDe allows extensions to expose an API via the `exports` property. CBRV exports its API using this method and can be accessed from other extensions as seen below.

```
let cbrvAPI = vscode.extensions
    .getExtension('jesse-r-s-hines.codebase-relationship-visualizer')
    .exports;
let vis = cbrvAPI.create({
  // ...
});
```

Other extensions can use the `create` method of the API to create a visualization with an arbitrary set of connections. The `create` method returns a `Visualization` object which can be then used to update the visualization dynamically.

Documentation for the CBRV API is published on GitHub pages [23].

3.3 Minimal Implementation

The CBRV extension has a command “Visualize your codebase” accessible via the VSCoDe Command Palette. This command opens a new VSCoDe tab displaying the directory structure diagram. This visualization is just a minimal example of using the CBRV API with no connections specified and all settings left as default. This command is useful if all the user wants is to get an overview of the directory structure without worrying about additional relationships.

3.4 Stack Trace Visualization

The command “Visualize the stack trace during a debugger session” brings up the Stack Trace Visualization. This visualization uses the CBRV API to display the

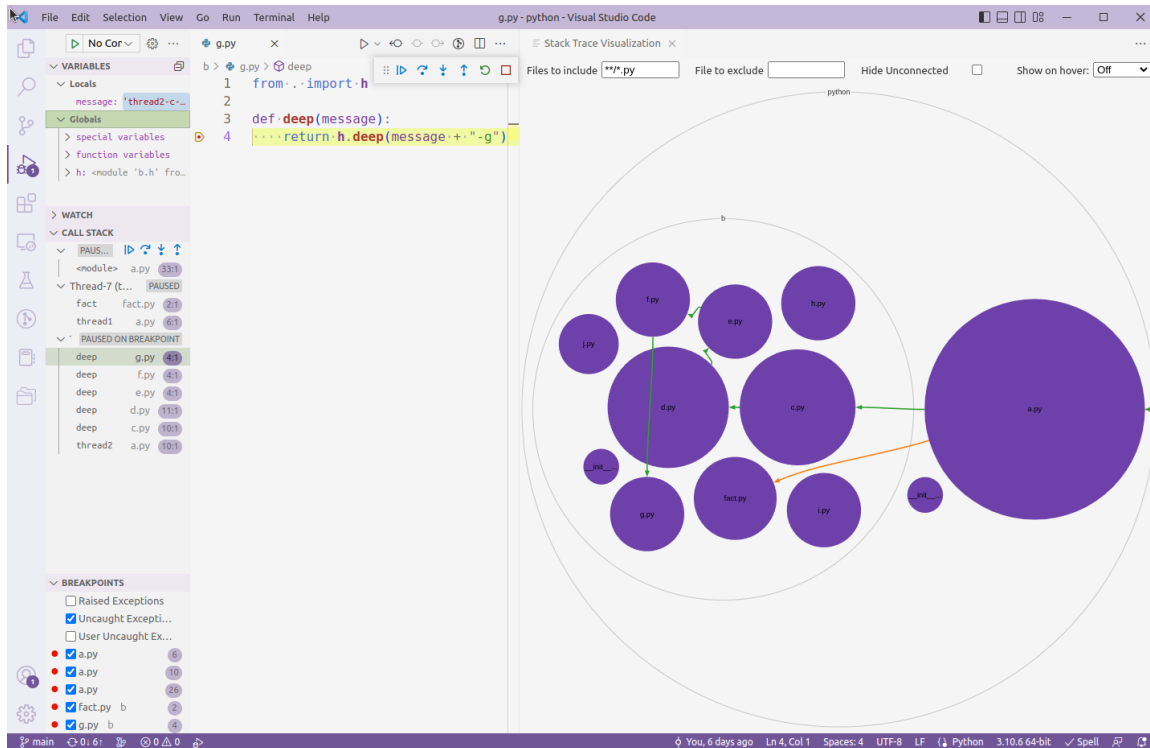


Figure 3.5: A screenshot of the CBRV Stack Trace Visualization

stack trace as a line over the directory structure diagram, as seen in Figures 3.5 and 3.6.

VSCoDe uses the Debug Adapter Protocol (DAP) [24] for its debugger interface. Most VSCoDe debugging extensions translate their captured debugging information into the Debug Adapter Protocol. This architecture makes it easy to make VSCoDe debugging extensions for multiple languages that can all use the same UI. It also allows extensions to extend debugging functionality in any language that has VSCoDe debugger that uses the DAP. VSCoDe has a very large ecosystem of extensions and has debuggers for most popular programming languages.

The Stack Trace Visualization uses the DAP to connect to any active VSCoDe debugging session in any language that has a VSCoDe debugger. Arrows jump

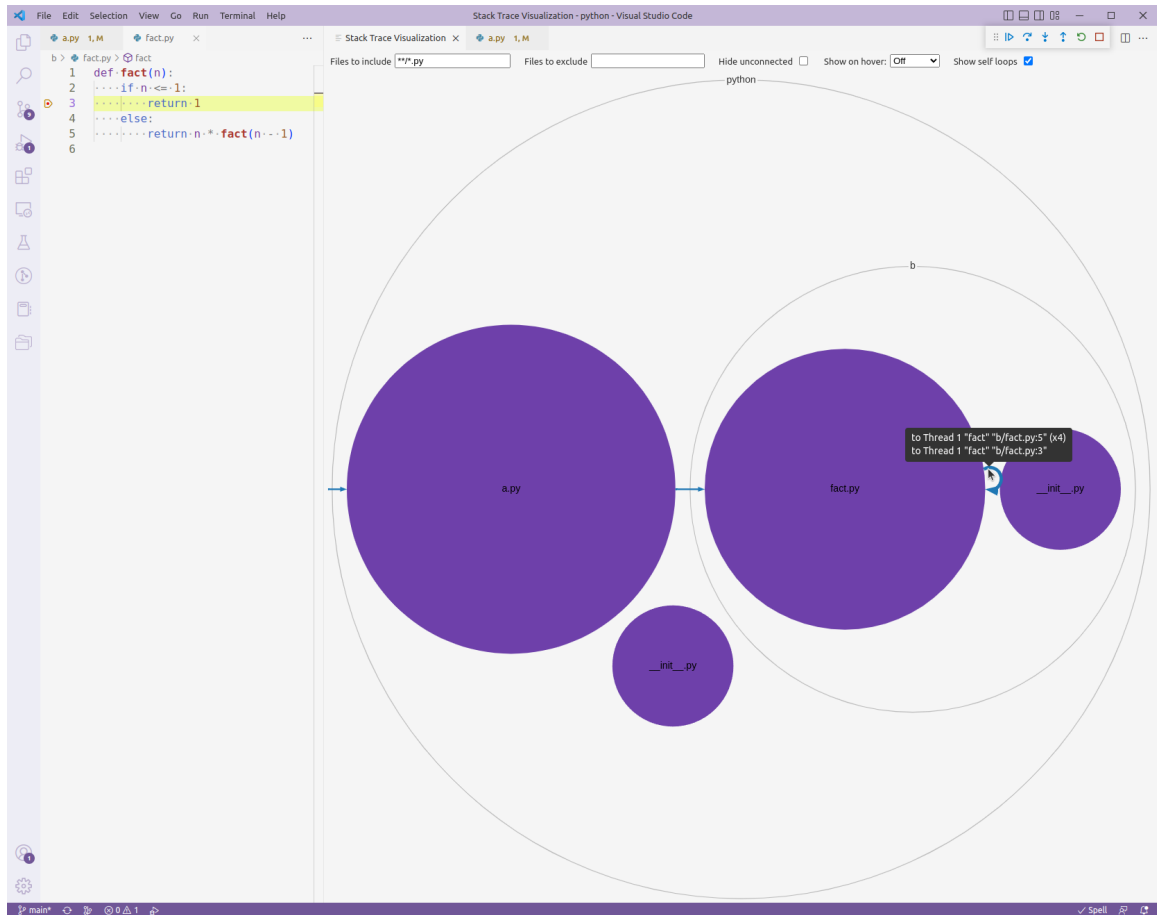


Figure 3.6: A screenshot of the CBRV Stack Trace Visualization with recursion shown as a merged connection with tooltip

between the files in the current stack trace, displaying a line following the current path of execution through the code base. The first file in the stack trace has an arrow going into it starting from outside the view window as seen in [Figure 3.5](#).

If there are duplicate arrows (i.e. recursion) the arrows are merged into a single connection as seen in [Figure 3.6](#). The arrows are only merged if they have the same source and destination files. The width of the connection and a tooltip on the connection indicate the depth of the recursion. The stack traces internal to a file are displayed as self loops.

Stack frames in files that are not included in the visualization are skipped. For example, the stack trace `main.py:8 → tutorial.py:78 → excluded.py:4 → tutorial_docker.py:34` the visualization would display arrows from `main.py → tutorial.py → tutorial_docker.py`.

The visualization is linked with the debugger in real time, as the user steps through the debugger the visualization updates to match the current stack trace.

In the case of multithreading, multiple stack trace lines are rendered, each color-coded by thread. This can be seen in [Figure 3.5](#) where two threads are shown by green and orange stack trace lines. The extension is only able to show stack traces for threads that are currently stopped on a breakpoint in the debugger.

3.5 Dependency Visualization

The command “Visualize the dependencies between Python files” brings up the Dependency Visualization. This visualization uses the CBRV API to display dependencies between files in Python codebases. It displays a directed arrow for any imports in `.py` files. It only show connections to/from a file on hover over that file by default. This visualization is very similar to the one proposed in RepoVisualizer [5]. The visualization uses the handy pydeps [25] package to extract the dependencies from the source files. A screen shot of the Dependency Visualization can be seen in [Figure 3.7](#).

3.6 Hyperlink Visualization

The command “Visualize a hyperlink graph” brings up the Hyperlink Visualization. This visualization uses the CBRV API to display references between HTML and

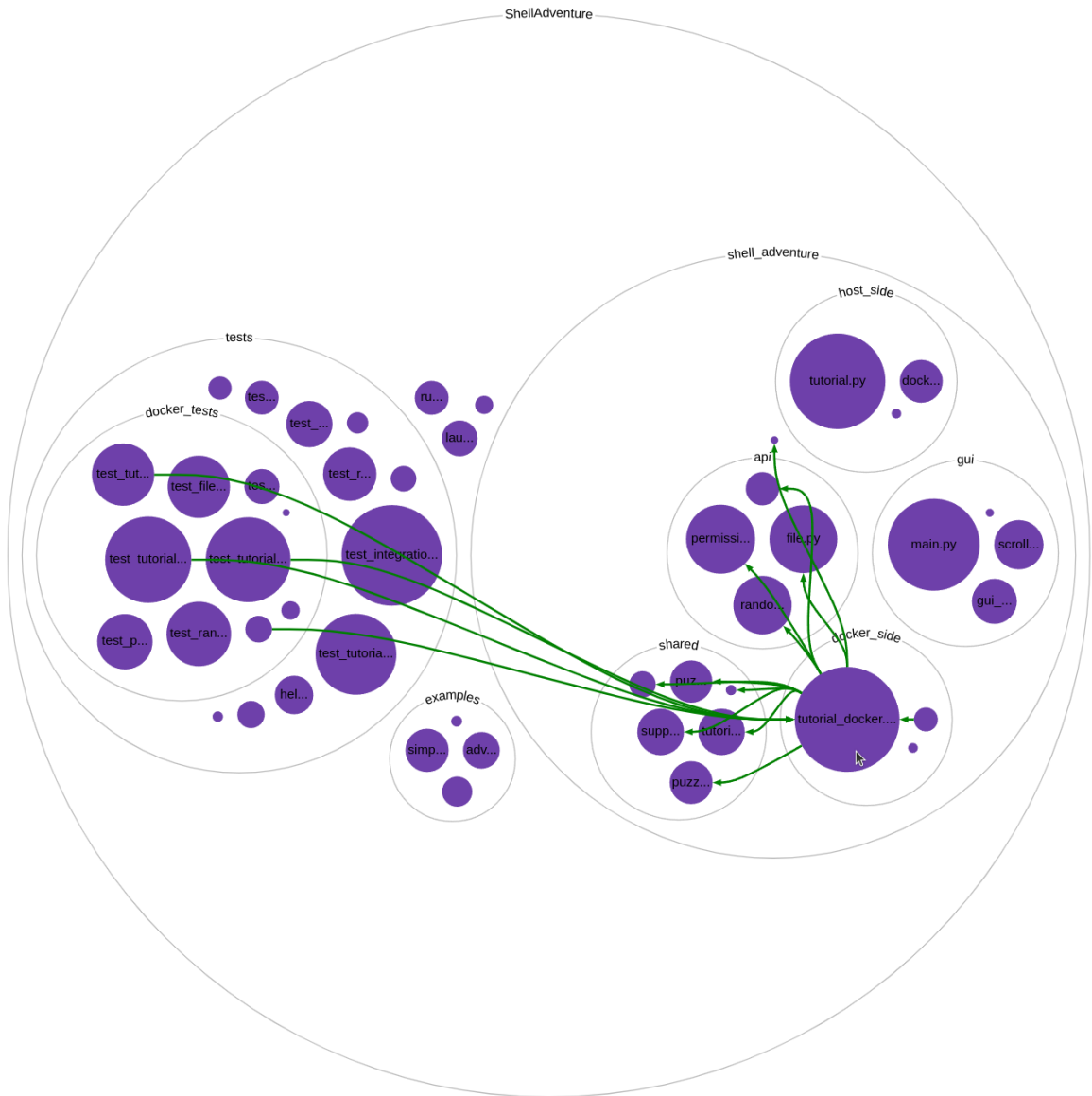


Figure 3.7: A screenshot of the dependency visualization (on the ShellAdventure [2] codebase)

Markdown files. It displays a directed arrow for any href in an HTML document or links in a Markdown document that reference another file in the workspace. The visualization only shows connection to/from a file on hover over that file by default to prevent the visualization getting overly crowded. Screenshots of the

Hyperlink Visualization can be seen in [Figures 3.2](#) and [3.4](#).

3.7 Architecture

Most of VSCode's extension API consists of hooks into standard VSCode functionalities. These are very useful and sufficient for most tasks. But, they are not flexible enough to implement extensions such as CBRV which need to build their own interface and have complex visual aspects. For these kinds of extensions, VSCode provides "webviews" which give extension writers a panel where they can use the full suite of web development tools. However, for security and performance reasons, VSCode webviews are run in a sandboxed environment, and communication in and out of them is limited to JSON message passing [26].

The sandboxing of the webview requires CBRV to be split into two sections, almost resembling a client server architecture. The extension proper runs in the normal VSCode environment. It contains the API and code to launch, update, and manage the CBRV visualization. The rendering of the diagram runs within the sandboxed webview. The two communicate using a simple messaging protocol, summarized in [Table 3.1](#). Since messages must JSON serializable, we are unable to send complex objects like functions into the webview. This forces the API to do things such as merge rules using configuration objects rather than callbacks.

Message	Source	Arguments	Response	Description
ready	webview	none	set	The webview is loaded
tooltip-request	webview	id, content	tooltip-set	Fetch the results of the tooltip callback
tooltip-set	extension	id, content	none	Return the results of the tooltip callback
open	webview	file	none	The user requested to open a file
reveal	webview	file	none	The user requested to reveal a folder in the explorer
context-menu	webview	action, file	none	The user selected a context menu action
update-settings	webview	settings	set/none	The user changed settings via the ui, triggers a set with the new file list if updating filters
set	extension	settings, codebase, connection	none	Update the settings, file list, and/or connection list

Table 3.1: Summary of the protocol between the main extension and the webview

The visualization itself is implemented using the powerful D3 [27] visualization library. D3 contains many useful tools for visualization, including circle packing logic, SVG curve path calculations, color scaling, and much more. D3 is low level and very flexible, making it perfect for implementing the combination circle-packing and link diagram required for CBRV.

Chapter 4

Tasks Delineation & Deliverables

Work progressed on the project between October 2021 and December 2022. A summary of the major tasks for the project along with rough time estimates is listed in [Table 4.1](#). Many of the tasks were worked on in parallel. Detailed requirements are listed in [Appendix A](#).

The following deliverables have been provided:

- CBRV API extension, along with the 4 built-in visualizations, is available on the VSCode marketplace
 - <https://marketplace.visualstudio.com/items?itemName=jesse-r-s-hines.codebase-relationship-visualizer>
- The source is published on GitHub
 - <https://github.com/jesse-r-s-hines/CodeBaseRelationshipVisualizer>
- API documentation is hosted on GitHub pages
 - <https://jesse-r-s-hines.github.io/CodeBaseRelationshipVisualizer>

- This final project report

#	Milestone	Time Estimate
1	Research & Prototyping	100
2	Proposal	50
3	Interactive Directory Structure Diagram	75
4	Context Menu & Actions from Diagram	5
5	Rendering Basic Connections	75
6	Customizing and Merging Connections	35
7	Dynamically Updating Connections and Files	20
8	Include/Exclude Settings	10
9	Positioning Algorithm	5
10	Stack Trace Visualization - Basic	20
11	Stack Trace Visualization - Multithreading	2
12	Dependency Visualization	20
13	Hyperlink Visualization	30
14	Testing & Evaluation	65
15	Polishing & Bugfixes	65
16	API Documentation	8
17	Publish & Project Report	15
Totals		600

Table 4.1: Summary of tasks for the CBRV project

Chapter 5

Testing & Evaluation

CBRV has an automated testing suite that can be run using `npm run test`. It passes, and tests any non-graphical portions of the code, including geometric logic, connection merging logic, and getting hyperlink and dependency graphs. However, since CBRV is primarily a visualization project, much of the code cannot easily be tested automatically. The visualization rendering itself was tested using manual acceptance testing.

There are several sample codebases provided for testing in the CBRV source code under the `test/sample-codebases` directory. These include a small markdown codebase, a python codebase, a codebase containing many types of symlinks, as well as a copy of the official VSCode docs repository [1] to test how CBRV scales to a large codebase.

There is a detailed list of manual acceptance tests that were performed in the CBRV GitHub repository as well as in [Appendix B](#).

Chapter 6

Future Work

We believe that CBRV shows promise for helping developers understand relationships between software files. Because of the API CBRV exposes, anyone can make their own extensions that uses the CBRV API to make visualizations for different types of relationships. Some potential extensions include expanding the dependency visualization to work with different languages or showing variable usage graphs.

Future work on the CBRV extension itself could include improvements to the file and connection positioning to reduce connections crossing each other. Additionally more transitions and animations when the file system or connection list is updated could make the visualization look smoother.

We did not find a practical way to implement the “Editor Decorations” described in the original proposal via the VSCode extension API. Future work could explore different methods to implement editor decorations, or other alternatives for showing relationships within a single file. Another area for improvement would be to make the include/exclude interface match more closely with VSCode’s built-in search functionality. This would likely require modifying the VSCode API itself to

30

expose hooks into the search functionality.

Chapter 7

Conclusion

Comprehending relationships within a codebase is a vital part of the software development process. CodeBase Relationship Visualizer (CBRV) offers an easy to install and extendable tool to display relationships between files in a codebase superimposed over a diagram of the codebase's directory structure. It allows developers to get a big picture of the relationships within a codebase and how the relationships fit into the overall directory structure of the codebase. The Stack Trace, Dependency, and Hyperlink Visualizations use the CBRV API to make useful visualizations for developers working on a different types of codebases. The extension can be installed by anyone via the VSCode marketplace [7] and the source is published under the GNU GPL on GitHub [8]. We believe that CBRV is a valuable addition to the repertoire of tools for visualizing and understanding codebases.

Appendix A

Detailed Requirements

This appendix shows a detailed list of requirements for the CBRV project. Most requirements have a reference to the corresponding task from [Table 4.1](#).

A.1 Non-Functional

1. The project will be a Visual Studio Code extension
2. The project will support VSCode 1.70.0+ [\[6\]](#)
3. The project will be made available on the VSCode extension marketplace [\[28\]](#)
4. The project will be open source and available on Github [\[29\]](#)
5. The project will be written using TypeScript 4.9 [\[30\]](#)

A.2 Functional

A.2.1 The Visualization

A.2.1.1 Directory Structure Diagram

6. (3) Will display the directory structure of a codebase using a circle packing diagram
 - Will display nested circles
 - Each folder is a circle containing other circles
 - Each file is a “leaf” of the diagram, at the most deeply nested level
 - Circles will have padding between them (as in [Figure 3.2](#))
7. (3) Will display files and folders starting from the root of the current workspace
8. (3) File and folder circles will display their names if they fit
 - The name will display inside the circle for files
 - The name will display along the circumference of the circle for folders (positioned near the top of the circle)
 - If the name will not fit, either hide the name or clip it to fit and add an ellipsis
9. (3) The color of file circles will be based on their file extension
10. (3) Folder circles will just be outlines
11. (3) The area of file circles be proportional to their actual file size
 - Displayed size will have min and max cap, to ensure small files are visible and large files do not use up to much screen space

- This prevents images and such being over emphasized, when text source files are what we are interested in
12. (3) Empty folders will not be shown
 13. (3) Symlinks will link to actual location in diagram
 - Symlinks that link to files within the visualization will show a symlink icon and clicking them will jump and highlight the canonical location of the file
 - Symlink circles' size will be fixed, not based on the linked file size
 - Symlink circles' color will be based on the linked file extension
 14. (7) The diagram will automatically update to match changes to the filesystem

A.2.1.2 Connections

15. (5) Connections can connect two files, from and to
 - Connections can optionally have line numbers within the files
 - Connections can only be between files, not directories
16. (5) The connections specified will render overlaying the directory structure diagram
 - Connections will render as lines or arrows connecting two file circles
17. (5) If files are hidden because of deep nesting (see [Req. 26](#)) connections will connect to the first ancestor folder that is shown
 - Connections between files within the folder will become self loops

- The connections will render normally when the user zooms in enough to see the individual files
 - See [Figure 3.3](#)
18. (5) Connections can go “out of the visualization”
- If connection from is omitted, the line will start from “outside” the visualization
 - If connection to is omitted, the line will end “outside” the visualization
 - See [Figure 3.5](#)
19. (5) Connections can have self-loops
- A connection between lines in a single file will render as a loop connecting the file to itself
 - The `showSelfLoops` option can be used to hide these loops instead
20. (5) Connections to a missing file will not be shown
21. (5) Connections will always connect to the canonical path of a file in case of symlinks
22. (6) Connections can be customized
- Will render as arrows if `directed` is set in the API, as lines otherwise
 - Connections’ color, and weight can be specified
 - Connections can have a tooltip
23. (6) Duplicate connections may be merged based on merge options instead of drawing multiple duplicate connections

- Merging can be disabled or configured in API settings
- The API can configure which connections to merge, whether to merge opposite direction arrows into one double-headed arrow, and how to merge properties of the connections

24. (7) Connections in an existing visualization can be updated via the API

A.2.1.3 Interaction

25. (3) The display will be zoomable and draggable

- Mouse wheel to zoom, drag and pan to move
- Also bind keyboard shortcuts Ctrl +, Ctrl -, and arrows

26. (3) The display will dynamically display content based on zoom level

- If the directory structure is deeply nested, only show folder content for the levels that fit on the screen
- The lowest folder that fits will show without content but with an ellipsis indicator to indicate that it is collapsed
- The size of the folder without content will match the size of the total content but capped the same as files (see [Req. 11](#))
- As the user zooms in to the display, more deeply nested files and folders will be displayed

27. (4) On hover over a folder or file circle path relative to workspace root will display in a tooltip

28. (4) Double-clicking a file circle will open it in a VSCode editor

29. (4) Double-clicking on a folder circle will show it in the VSCode file explorer

30. (4) Context menu

- A context menu (on right click) should be available on folders and circles
- It should have the options:
 - Reveal in Explorer
 - Open in New Editor (only for files)
 - Copy Path
 - Copy Relative Path
- The context menu will also include any additional commands listed via the API

31. (6) If `showOnHover` is set it will hide all connections but those connected to the hovered file/folder

- Can be set to show only connections “in”, “out”, or “both”

A.2.1.4 Include/Exclude

32. (8) There will be inputs for “Include” and “Exclude”

- Will filter files displayed in the visualization
- Specified as comma separated list of file paths with wildcards

33. (8) There will a toggle for “Hide Unconnected”

- Will not show if there are no connections
- Will hide any files/folders that do not have any connections to them

34. (8) Visualization will dynamically update to match new Include/Exclude settings
35. (8) Folders will not be shown if all contents are excluded
36. (8) Connections to files that are excluded will not be displayed
37. (8) Default Include/Exclude settings can be specified via the API

A.2.2 API

38. The API will be used by other extensions to display connections between files
39. See <https://jesse-r-s-hines.github.io/CodeBaseRelationshipVisualizer> for API documentation

A.2.3 Minimal Implementation

40. (3) CBRV will have a command “Visualize your codebase”, accessible via the VSCode Command Palette
 - The command will open up a new VSCode tab displaying the directory structure diagram
 - It will just be a minimal example of using the API with no connections specified and all settings left as default

A.2.4 Stack Trace Visualization

41. (10) The Stack Trace Visualization extension will have a command “Visualize the stack trace during a debugger session”, accessible via the VSCode Command Palette

- Command will open up a new VSCode tab containing the visualization for the current workspace and debug session
 - Attempts to connect to any active debugging session
 - If there is no active debugging session, it will notify the user that they need to start a session and bring up the visualization without any debug info
 - Once a debug session starts it should connect it to the visualization automatically
42. (10) Will use the Debug Adapter Protocol [24] to get debug information
- This will make it work with any language that has a VSCode debugger
43. (10) The visualization will be linked with the debugger real-time
- As the user steps through the debugger the visualization will update
44. (10) Will display arrows between files representing the stack trace
- Arrows will jump between the files in the current stack trace, displaying a line following the current path of execution through the code base
 - The stack trace line can loop back on itself if there are circular dependencies in the code
45. (10) The first file in the stack trace will have an arrow going into it starting from outside of the view window
46. (10) Multiple consecutive stack frames within the same file will not be displayed on the main visualization
47. (10) If there are duplicate arrows (i.e. recursion) merge the connections

- Duplicate arrows have the same source and destination line numbers
 - Tooltip on connection should show number of times of recursion
48. (10) Stack frames in files that are not part of the visualization will not be displayed
- E.g. if stack trace goes `fileA.py` → `excludedFile.py` → `fileB.py`, we just display `fileA.py` → `fileB.py`
49. (11) Multithreading will show multiple color coded stack trace lines
- Display a stack trace for each stopped thread in the debugger
 - Connections will be color coded by thread
 - Connections from different threads will never merge

A.2.5 Dependency Visualization

50. (12) The Dependency Visualization extension will have a command “Visualize the dependencies between python files”, accessible via the VSCode Command Palette
- Command will open up a new VSCode tab containing the dependency graph for the current workspace
51. (12) Will display dependencies between Python files
- Show a directed arrow for any `import` statement in a `.py` file
52. (12) Will only show references to/from a file on hover over that file (see [Req. 31](#))
53. (12) Will merge all connections that go between the same files

- Merge opposite direction connections into one double-headed arrow

A.2.6 Hyperlink Visualization

54. (13) The extension will have a command “Visualize a hyperlink graph”, accessible via the VSCode Command Palette
- Command will open up a new VSCode tab containing the hyperlink graph for the current workspace
55. (13) Will display a directed arrow for any link that points to another file in the workspace
- Any href on a `a`, `link`, or `area` tag in HTML
 - `[example](http://example.com)`, `<http://example.com>`, and raw `http://example.com` links in Markdown
56. (13) Will only show references to/from a file on hover over that file [Req. 31](#)
57. (13) Will merge all connections that go between the same files
- Merge opposite direction connections into one double-headed arrow
 - Will display internal links (i.e. `#Header1`) as connections within the file preview
58. (13) If a link goes to an external URL or to an excluded file do not show it

Appendix B

Acceptance Testing

This appendix shows a list of manual acceptance tests for the CBRV project.

B.1 Sample codebases

These sample codebases are CBRV repository under `test/sample-codebases`.

1. `minimal`

A codebase that's just a set of files and directories. Contains some long names to show name cropping. Used to test the minimal codebase visualization.

2. `python`

A simple python codebase containing a stack trace 8 levels deep and recursion cycles within a single file and between two files. Used to test the stack trace and dependency visualizations.

3. `simple`

A simple markdown codebase to test the hyperlink visualization and basic connections and iterations.

4. symlinks

A codebase that contains internal and external symlinks to test symlink rendering and handling.

5. symlink

A codebase where the root folder is a symlink to `simple-hyperlink-graph`.

6. vscode

This sample codebase is taken from the official VSCode docs repository [1]. It used to test the hyperlink visualization and test how CBRV scales to a large and interconnected codebase.

7. empty

A codebase that is just an empty folder. Since `git` can't commit empty directories this codebase will have to be created manually by the tester.

B.2 Acceptance Tests

B.2.1 Basic diagram

1. Open the minimal codebase in VSCode
2. Run the "Visualize your codebase" command
 - (View > Command Palette or Ctrl + Shift + P) and type and select "Visualize your codebase"
3. A circle packing diagram containing 7 leaf files should be shown
4. File names should show on files and folders

5. deoxyribonucleicAcid folder and Supercalifragilisticexpialidocious file should have their names cropped
 - May depend on your screen and font size settings
6. E.txt should be the largest circle, deoxyribonucleicAcid/I should be the smallest
7. Files should be color coded by extension (should be 4 different colors shown)

B.2.2 Basic interaction

1. Open the minimal codebase in VSCode
2. Run the “Visualize your codebase” command
3. Zoom and pan should work use mouse wheel and click and drag
4. Use keyboard shortcuts `ctrl +`, `ctrl -` to zoom and pan
 - May need to click on the svg to make sure its focused first
5. Hover over the smallest file
 - It should show deoxyribonucleicAcid/I in a tooltip
6. Double click A/F.md
 - The file should open in a VSCode editor panel
7. Double click A
 - The folder should be selected in the explorer panel
8. Right click on A/E.txt

- A context menu should show containing:
 - Reveal in Explorer
 - Open in editor
 - Copy Path
 - Copy Relative Path
9. Use the context menu on A/E.txt to “Reveal in Explorer”
- Should select A/E.txt in vscode explorer
10. Use the context menu on A/E.txt to “Open”
- Should open A/E.txt in the editor
11. Use the context menu on A/E.txt to “Copy Path”
- Should copy full path `.../sample-codebases/minimal/A/E.txt` to clipboard
12. Use the context menu on A/E.txt to “Copy Relative Path”
- Should copy A/E.txt to clipboard
13. Right click on A
- A context menu should show containing:
 - Reveal in Explorer
 - Copy Path
 - Copy Relative Path
14. Use the context menu on A to “Reveal in Explorer”

- Should select A in vscode explorer
15. Use the context menu on A to “Copy Path”
 - Should copy full path `.../sample-codebases/minimal/A` to clipboard
 16. Use the context menu on A to “Copy Relative Path”
 - Should copy A to clipboard

B.2.3 Filtering

1. Open the minimal codebase in VSCode
2. Run the “Visualize your codebase” command
3. Type `A/**` in “Files to Include” input, press Enter
 - Diagram should change to only include `E.txt`, `F.txt`, `G.txt`
4. Type `**/*.txt` in “Files to Exclude”
 - Diagram should change to only include `G.md`
5. Clear both inputs
 - View should go back to what it was before showing everything
6. Type `**/*.txt, **/*.md` into “Files to Include”
 - Diagram should change to include only `C.txt`, `D.txt`, `E.txt`, `F.txt`, `G.txt`
7. Clear both inputs
8. Type `A/**` in “Files to Exclude”

- Folder A and all contents should be hidden

B.2.4 Hyperlink Visualization and Connections

1. Open the `simple-hyperlink-graph` codebase in VSCode
2. Run the “Visualize a hyperlink graph” command
 - Should show the codebase diagram, but no connections at first
3. Hover over `0.md`
 - Connections for `Q.md` and `P.md` should show
4. Hover over `P.md`
 - Connection from `P.md` *rightarrow* `0.md` should show
5. Change the “Show on hover” dropdown to “in only”
6. Hover over `P.md` again
 - No connection should show
7. Hover over `0.md`
 - Connections for `Q.md` and `P.md` should show.
8. Change the “Show on hover” dropdown to “out only”
9. Hover over `P.md`
 - Connection from `P.md` *rightarrow* `0.md` should show
10. Hover over `0.md`

- No connections should show
11. Change the “Show on hover” dropdown to “off”
 - Connections across the diagram should appear.
 12. One double headed arrow should show between M.md and R.md
 13. A single self loop connection should show on N.md
 14. Uncheck the “Show self loops” checkbox
 - The loop on N.md should disappear
 15. Check the “Show self loops” checkbox
 - The loop on N.md should come back
 16. Check “Hide unconnected”
 - View should change to only include files with connections
 17. Uncheck show self loops
 - N.md' should disappear
 18. Check show shelf loops
 - N.md' should reappear
 19. Uncheck “Hide unconnected”
 - All files should show again
 20. Hover over the connection between M.md and R.md
 - Tooltip should show containing

- "A/D/M.md" -> "A/D/R.md"
- "A/D/R.md" -> "A/D/M.md" x2

B.2.5 Real-time updates

1. Open the simple-hyperlink-graph codebase in VSCode
2. Run the "Visualize a hyperlink graph" command
3. Open A/F/Q.md and delete the first line.
 - Link between Q.md and O.md should disappear.
4. Undo the line change
 - Link between Q.md and O.md should come back.
5. Delete or move folder A out of the codebase
 - Visualization should immediately update to remove the A folder
6. Put A back
 - `git reset --hard HEAD` or copy A back into codebase
 - Visualization should immediately update to show A again

B.2.6 Symlinks

1. Open the symlinks codebase in VSCode
2. Run the "Visualize a hyperlink graph" command
3. Symlinks files should show as colored circles with an arrow icon inside

4. link folder should show as a circle outline with an arrow icon inside
5. externalLink.md and E.txt should show as different colors
 - The color is based on the resolved filepath's extension
6. Zoom in until you can only see link.md in the screen
7. Double-click link.md
 - View should jump to center on and fit B.md
 - B.md should flash
8. Zoom out all the way and double-click on link folder
 - A should flash
 - View shouldn't change
9. Double-click on external.md
 - Nothing should happen
10. Hover over B.md
 - Connections should show to C.md and link.md
11. Open the symlink-root codebase in VSCode
12. Run the "Visualize a hyperlink graph" command
 - Should show a codebase containing several files
13. Hovering over P.md
 - Should show connection to O.md

B.2.7 Large Codebase

1. Open the `vscode-docs` codebase in VSCode
2. Run the “Visualize a hyperlink graph” command
3. Deep folders should have their contents hidden and replaced with ellipses
 - Exact results will depend on your screen size
4. Zoom in on one of the folders with hidden content
 - Contents should show up as you zoom.
5. Hover over hidden folders and regular files
 - Connections show on hover, and connect to the first visible parent folder of their target

B.2.8 Stack Trace Visualization

1. Open the `python` codebase in VSCode
2. Install the “Python” extension from the VSCode marketplace for python debugging if you don’t already have it
3. Run the “Visualize the stack trace during a debugger session” command
 - Notification saying “No active debug session” should show
4. Type `**/*.py` in “Files to Include”
 - `.pyc` files should be hidden
5. Add breakpoints (Click in gutter) at

- `b/fact.py:3`
 - `b/e.py:3`
 - `b/j.py:3`
 - `a.py:26`
6. Open `a.py` in a split panel so you can see both the visualization and the editor
 7. Open “Run and Debug” panel in VSCode’s side panel
 8. Focus `a.py` editor view
 9. Click “Run and Debug” in the side panel and select “Python File”
 - Debugger should stop at `b/fact.py:3`
 - Connections: out of screen *rightarrow* `a.py` *rightarrow* `fact.py` *rightarrow* `fact.py` (self loop)
 10. Click continue
 - Debugger should stop at `b/e.py:3`
 - Connections: out of screen *rightarrow* `a.py` *rightarrow* `c.py` *rightarrow* `d.py` *rightarrow* `e.py`
 11. Type `b/d.py` in “Files to Exclude”
 - Connections: out of screen *rightarrow* `a.py` *rightarrow* `c.py` *rightarrow* `e.py`
 12. Clear “Files to Exclude”

- Connections: out of screen *rightarrow* a.py *rightarrow* c.py *rightarrow* d.py *rightarrow* e.py

13. Click continue

- Debugger should stop at b/j.py:3
- Connections: out of screen *rightarrow* a.py *rightarrow* c.py *rightarrow* d.py *rightarrow* e.py *rightarrow* f.py *rightarrow* g.py *rightarrow* h.py *rightarrow* i.py *rightarrow* j.py

14. Remove breakpoint at b/j.py:3 (click the red dot in the gutter)

15. Click continue

- Debugger should stop at a.py:26
- Connections: out of screen *rightarrow* a.py

16. Click continue.

17. Should stop at either thread1 b/fact.py:3 or thread2 b/j.py:3

18. In the debugger side panel select the other thread

- If you are stopped on b/fact.py select the thread that starts at thread2 in the stack trace,
- If stopped on b/e.py select the thread that starts on thread1

19. Click continue again

- Should stop at the other thread.
- You should now see two lines in the stack trace of different colors

- Out of screen *rightarrow* a.py *rightarrow* c.py *rightarrow* d.py
rightarrow e.py
- Out of screen *rightarrow* a.py *rightarrow* fact.py *rightarrow* fact.py
(self loop)

20. Click stop

- Debugging session will end to end the debugging session,
- Connections should clear

B.2.9 Dependency Visualization

1. Open the python codebase in VSCode
2. Run the “Visualize the dependencies between python files” command
 - If this is your first time running the command, it should show an “Installing pydeps...” progress notification
 - Wait for any progress notifications to complete
3. Hover over a.py
 - Connections should show to b/c.py, b/__init__.py and b/fact.py
4. Hover over b/h.py
 - Connections should show to b/i.py, b/__init__.py, __init__.py
 - Connections should show from b/g.py

B.2.10 Empty Codebases

1. Open empty codebase (may need to create an empty folder)
2. Run the “Visualize your codebase” command
 - Should just show an empty circle
3. Run the “Visualize a hyperlink graph” command
 - Should just show an empty circle
4. Run the “Visualize the stack trace during a debugger session” command
 - Should just show an empty circle
5. Run the “Visualize the dependencies between python files” command
 - Should just show an empty circle

Bibliography

- [1] Microsoft, "Microsoft/vscode-docs: Public documentation for visual studio code." [Online]. Available: <https://github.com/microsoft/vscode-docs> (document), 3.3, 3.4, 5, 6
- [2] J. Hines, "Jesse-r-s-hines/shelladventure: A tool for making tutorials to teach the linux command line." [Online]. Available: <https://github.com/jesse-r-s-hines/ShellAdventure> (document), 3.7
- [3] Y.-G. Guéhéneuc, "Taupe: Towards understanding program comprehension," in *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '06. USA: IBM Corp., 2006, pp. 1–es. [Online]. Available: <https://doi.org/10.1145/1188966.1188968> 1, 2.1
- [4] J. Seyster, "Techniques for visualizing software execution," Stony Brook University, Tech. Rep., 2008. 1, 2.1
- [5] A. Wattenberger, "Visualizing a codebase," 2021. [Online]. Available: <https://next.github.com/projects/repo-visualization> 1.2, 2.2.2, 3.1.1, 3.5
- [6] Microsoft, "Visual studio code - code editing. redefined," 2021. [Online]. Available: <https://code.visualstudio.com/> 1.2, 2

- [7] J. Hines, “Codebase relationship visualizer.” [Online]. Available: <https://marketplace.visualstudio.com/items?itemName=jesse-r-s-hines.codebase-relationship-visualizer> 1.2, 7
- [8] —, “Jesse-r-s-hines/codebaserelationshipvisualizer: A vscode extension to display relationships between files in a codebase, overlaid on a circle packing diagram of the file structure.” [Online]. Available: <https://github.com/jesse-r-s-hines/CodeBaseRelationshipVisualizer> 1.2, 7
- [9] C. D. HUNDHAUSEN, S. A. DOUGLAS, and J. T. STASKO, “A meta-study of algorithm visualization effectiveness,” *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1045926X02902375> 1.3
- [10] T. L. Naps, J. R. Eagan, and L. L. Norton, “JhavÉ—an environment to actively engage students in web-based algorithm visualizations,” in *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE ’00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 109–113. [Online]. Available: <https://doi.org/10.1145/330908.331829> 2.1
- [11] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. A. Velázquez-Iturbide, “Exploring the role of visualization and engagement in computer science education,” *SIGCSE Bull.*, vol. 35, no. 2, pp. 131–152, jun 2002. [Online]. Available: <https://doi.org/10.1145/782941.782998> 2.1
- [12] R. Baecker, “Two systems which produce animated representations of the execution of computer programs,” in *Proceedings of the Fifth SIGCSE Technical*

- Symposium on Computer Science Education*, ser. SIGCSE '75. New York, NY, USA: Association for Computing Machinery, 1975, pp. 158–167. [Online]. Available: <https://doi.org/10.1145/800284.811152> 2.2.1
- [13] T. Okamura, B. Shizuki, and J. Tanaka, “Execution visualization and debugging in three-dimensional visual programming,” in *Proceedings. Eighth International Conference on Information Visualisation, 2004. IV 2004.*, 2004, pp. 167–172. [Online]. Available: <https://doi.org/10.1109/IV.2004.1320140> 2.2.1
- [14] S. Deitz and U. Buy, “From video games to debugging code,” in *Proceedings of the 5th International Workshop on Games and Software Engineering*, ser. GAS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 37–41. [Online]. Available: <https://doi.org/10.1145/2896958.2896964> 2.2.1
- [15] A. Zeller and D. Lütkehaus, “Ddd—a free graphical front-end for unix debuggers,” *SIGPLAN Not.*, vol. 31, no. 1, pp. 22–27, jan 1996. [Online]. Available: <https://doi.org/10.1145/249094.249108> 2.2.1
- [16] “Visualgo.net.” [Online]. Available: <https://visualgo.net/> 2.2.1
- [17] “Algorithm visualizer.” [Online]. Available: <https://algorithm-visualizer.org/> 2.2.1
- [18] H. Dieterichs, “Hediet/vscode-debug-visualizer: An extension for vs code that visualizes data during debugging.” [Online]. Available: <https://github.com/hediet/vscode-debug-visualizer> 2.2.1
- [19] P. Deelen, F. van Ham, C. Huizing, and H. van de Wetering, “Visualization of dynamic program aspects,” in *2007 4th IEEE International Workshop on*

- Visualizing Software for Understanding and Analysis*, 2007, pp. 39–46. [Online]. Available: <https://doi.org/10.1109/VISSOF.2007.4290698> 2.2.2
- [20] R. Bertuli, S. Ducasse, and M. Lanza, “Run-time information visualization for understanding object-oriented,” *4th International Workshop on Object-Oriented Reengineering*, pp. 10–19, 12 2003. 2.2.2
- [21] D. B. Lange and Y. Nakamura, “Program explorer: A program visualizer for c++,” in *Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, ser. COOTS’95. USA: USENIX Association, 1995, p. 4. 2.2.2
- [22] E. Gräther and M. Langkabel, “Sourcetrail - free and open-source interactive source explorer.” [Online]. Available: <https://github.com/CoatiSoftware/Sourcetrail> 2.2.2
- [23] J. Hines, “Codebaserelationshipvisualizer api docs.” [Online]. Available: <https://jesse-r-s-hines.github.io/CodeBaseRelationshipVisualizer/> 3.1.2, 3.2
- [24] Microsoft, “Dap,” 2021. [Online]. Available: <https://microsoft.github.io/debug-adapter-protocol/> 3.4, 42
- [25] TheBjron, “Thebjorn/pydeps: Python module dependency graphs.” [Online]. Available: <https://github.com/thebjorn/pydeps> 3.5
- [26] Microsoft, “Vscode webview,” Mar 2022. [Online]. Available: <https://code.visualstudio.com/api/extension-guides/webview> 3.7
- [27] M. Bostock, “Data-driven documents.” [Online]. Available: <https://d3js.org/> 3.7

- [28] “Visual studio marketplace.” [Online]. Available: <https://marketplace.visualstudio.com/vscode> 3
- [29] Microsoft. [Online]. Available: <https://github.com/> 4
- [30] —, “Typescript.” [Online]. Available: <https://www.typescriptlang.org/> 5