Path Finding for Typescript

This notebook demonstrates different ways on how path finding algorithms can be utilized for code analysis.

Path algorithms in Graphs are famous for e.g. finding the fastest way from one place to another. How can these be applied to static code analysis and how can the results be interpreted?

One promising algorithm is All Pairs Shortest Path. It shows dependencies from a different perspective and provides an overview on how directly or indirectly dependencies are connected to each other. The longest shortest path has an additional meaning: It is also known as the **Graph Diameter** and is very useful as a metric for the complexity of the Graph (or Subgraphs). The longest path (for directed acyclic graphs) can uncover the longest existing (worst case) dependency chains as long as there are no cycles in the Graph.

References

- jqassistant
- Neo4j Python Driver
- All Pairs Shortest Path
- Longest Path for DAG (neo4j)
- Graph Diameter

What GPT-4 has to say about it

All pairs shortest path

Interpreting the results of the "all pairs shortest path" algorithm on a graph of statically analyzed code modules and their dependencies involves understanding the structure and implications of the paths between nodes (modules) in the graph. Here are some specific steps and insights to consider:

1. **Graph Structure**: Each node represents a code module, and edges indicate dependencies. A directed edge from module A to module B implies that A depends on B.

2. **Shortest Paths**: The results will give you the shortest path lengths between all pairs of modules. This helps identify:

- **Direct Dependencies**: A length of 1 indicates a direct dependency.
- **Transitive Dependencies**: A length greater than 1 shows indirect dependencies. For example, if the path from A to C is 2, it could mean A → B → C, indicating A indirectly depends on C via B.

3. **Module Isolation**: If a module has very long paths to others, it might be more isolated. This could signal potential issues in the code structure, suggesting that module might be overly complex or decoupled from the rest of the system.

4. **Critical Paths**: Identify the shortest paths that connect key modules (e.g., core functionalities). These paths can highlight the most crucial dependencies that, if modified, might have extensive impacts on the system.

5. **Cycle Detection**: If any pairs have paths that loop back to themselves with a length not equal to 0, it indicates a cycle. Cycles can complicate dependency management, potentially leading to recursive dependencies, which can be problematic in terms of maintainability.

6. **Refactoring Opportunities**: By examining the lengths of paths, you might identify modules that could benefit from refactoring to decrease dependency complexity. For example, a module that has dependencies on many others (with longer path lengths) might be a candidate for breaking into smaller, more manageable components.

7. **Performance Considerations**: In large systems, long paths could impact performance. If certain modules are far from frequently accessed modules, consider whether they can be optimized for speed.

8. **Visual Representation**: Creating a visual representation of the graph with the shortest paths highlighted can be immensely helpful. Tools like Graphviz or D3.js can illustrate these relationships clearly, aiding in your analysis.

By focusing on these aspects, you can glean actionable insights from the results of the all pairs shortest path algorithm in the context of your statically analyzed code modules and their dependencies.

Graph diameter (shortest longest path)

The longest shortest path in a dependency graph (often referred to in graph theory as the "diameter" of the graph) represents the maximum distance (in terms of the number of edges or dependencies) between any two nodes (modules) in the graph. Here’s how you can interpret this metric in the context of statically analyzed code modules and their dependencies:

1. **Network Complexity**: The longest shortest path indicates the overall complexity of the network of dependencies. A longer path suggests a more complicated interrelationship among modules. For example, a longest shortest path of 6 could indicate that there is at least one pair of modules in your system that rely on a chain of 6 other modules to communicate or function together.

2. **Potential Bottlenecks**: If the longest shortest path is significant, it may suggest potential bottlenecks in your architecture. For instance, if a core module at the beginning of a long path is slow or error-prone, it could affect numerous other modules dependent on it, resulting in systemic performance issues.

3. **Critical Communication Points**: The endpoints of the longest shortest path can be seen as critical communication points within your codebase. Understanding these connections can help identify

which modules should be prioritized for testing and monitoring, especially during changes.

4. **Isolation and Coupling**: A long longest shortest path might indicate that some modules are isolated and far removed from others, which can suggest low cohesion. This can be a sign that the architecture might benefit from refactoring to reduce unnecessary dependencies or to improve modularity.

5. **Refactoring Opportunities**: If the longest shortest path is disproportionately long, it may highlight areas in the codebase where modules are too tightly coupled. This situation presents an opportunity for refactoring to create more independent modules or components that can interact with fewer dependencies.

6. **Impact of Changes**: Modules that lie along or are endpoints of the longest shortest paths are likely to have a significant impact on the overall system. Changes to them should be approached with caution and accompanied by rigorous testing.

7. **Cycle Detection**: In some cases, a long shortest path can indicate the presence of cycles in the graph. If there are paths that seem to loop back on themselves, it suggests potential design flaws that could lead to recursion or infinite loops, complicating maintenance.

8. **Architectural Decisions**: The longest shortest path can inform architectural decisions by providing insights into which dependencies might need to be revised or eliminated. For instance, if certain modules are consistently part of the longest path, it could justify investing resources in redesigning their interactions.

In summary, interpreting the longest shortest path provides a comprehensive view of the interdependencies among modules in your system, focusing on complexity, potential bottlenecks, and opportunities for improvement in architecture and design.

Longest path

1. **Complex Dependencies**: Longest paths indicate modules that have extensive dependencies before reaching another module. For instance, if you find a path like A → B → C → D with a length of 4, it highlights a complex chain of dependencies. This can suggest that changes in module A might have far-reaching implications across the system.

2. **Potential Bottlenecks**: Modules located at the beginning of long paths could be performance bottlenecks. If a frequently used module is several layers deep in dependencies (e.g., A → B → C), it may slow down the entire system. Optimizing or refactoring these modules could improve performance.

3. **Maintenance Challenges**: Long paths may indicate parts of the code that are difficult to maintain or understand. For example, if module A requires multiple intermediary modules (B, C, D) for its functionality, developers may struggle to trace how changes propagate, leading to potential bugs.

4. **Risk of Change**: A module with a long dependency path can be more risky to modify. If A has dependencies on several modules down the line (e.g., A → B → C → D → E), any changes to A

could inadvertently affect E, which might be critical or sensitive. This insight can help prioritize testing and reviews around such modules.

5. **Decoupling Opportunities**: Identifying the longest paths can highlight areas where you might want to break up dependencies. If there's a long chain that could be simplified (e.g., by creating intermediary modules or interfaces), it may lead to a more modular and maintainable architecture.
6. **Redundancy**: Long paths may also reveal redundancy in dependencies. For instance, if multiple modules depend on a series of others (A → B → C → D and A → B → E), it could indicate unnecessary coupling that can be streamlined.
7. **System Understanding**: Long paths can help map out the architecture of your codebase. Understanding which modules are pivotal in long chains can provide insights into the overall design and help inform architectural decisions.
8. **Documentation & Knowledge Transfer**: If certain modules consistently appear in the longest paths, they may require better documentation. Ensuring that their roles and the reasons for their lengthy dependencies are well understood can facilitate knowledge transfer among team members.

By focusing on the longest paths in your dependency graph, you can uncover areas requiring attention for optimization, maintenance, and improved system architecture.

1. Typescript Modules

1.1 All pairs shortest path

Use "All Pairs Shortest Path" algorithm to get the shortest path distance between all pairs of dependent Typescript packages. It shows how many Packages have a direct dependency (distance 1), how many are reachable with one dependency in between (distance 2), and so on...

1.1.1 Create a projection of all Typescript module dependencies

Creates a in-memory projection of "TS:Module" nodes and their "DEPENDS_ON" relationships as a preparation to run the Graph algorithms. The weight property is not used for now (September 2024) but

may be needed for other algorithms/variants some time.

Projected Graph statistics for Typescript module dependencies

	nodeCount	relationshipCount	density	sizeInBytes	degreeDistribution.min	degreeDistribution.mean	degreeDistribution.max	degreeDistribution.p50	degreeDistri
0	5	7	0.35	2495512	0	1.4	3	1	

1.1.2 All pairs shortest path in total

First, we'll have a look at the overall/total result of the all pairs shortest path algorithm for all dependencies.

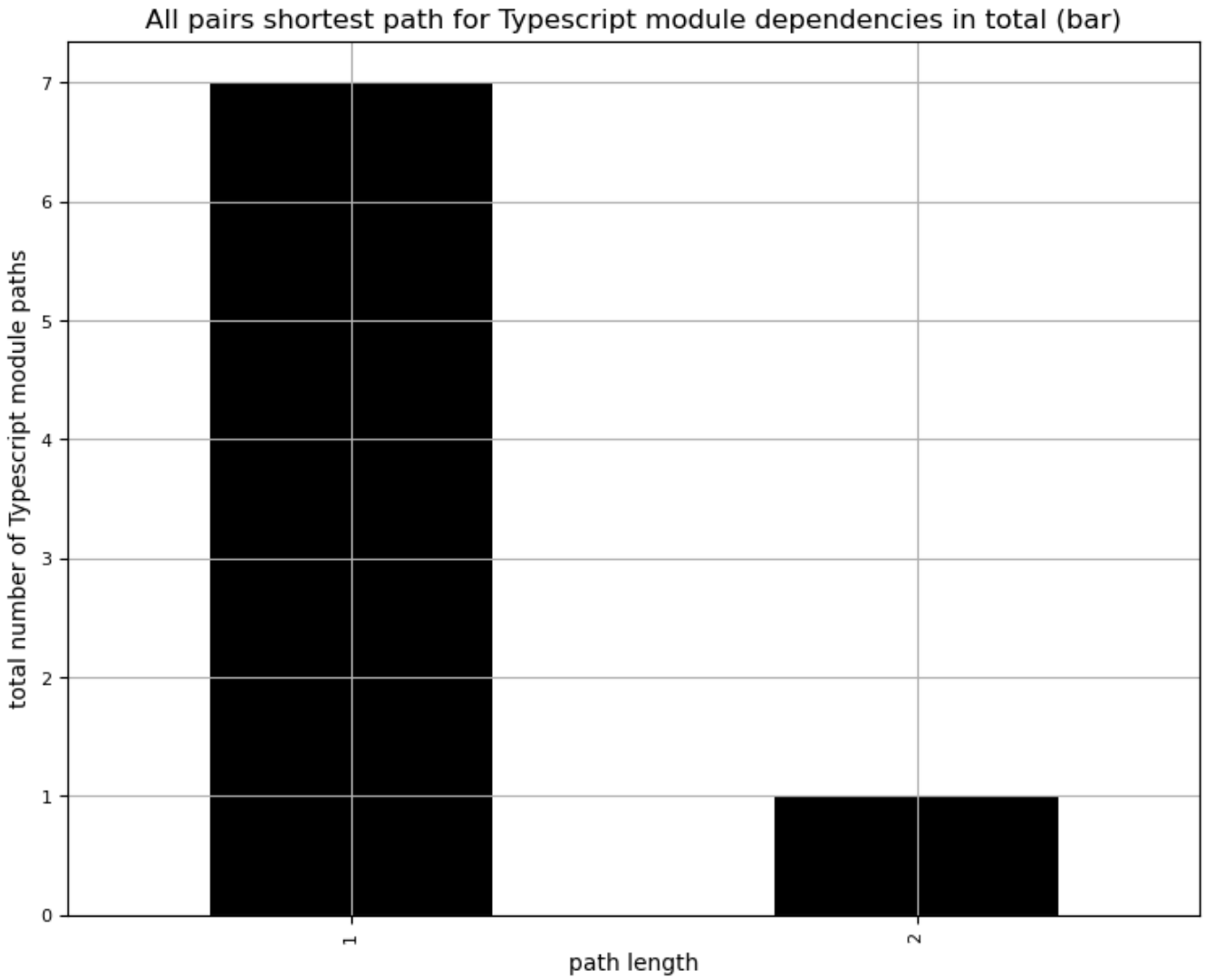
All pairs shortest path in total - Longest shortest path (Graph diameter)

```
The diameter (longest shortest path) of the projected module dependencies Graph is: 2
```

All pairs shortest path in total - Path count per length - Table

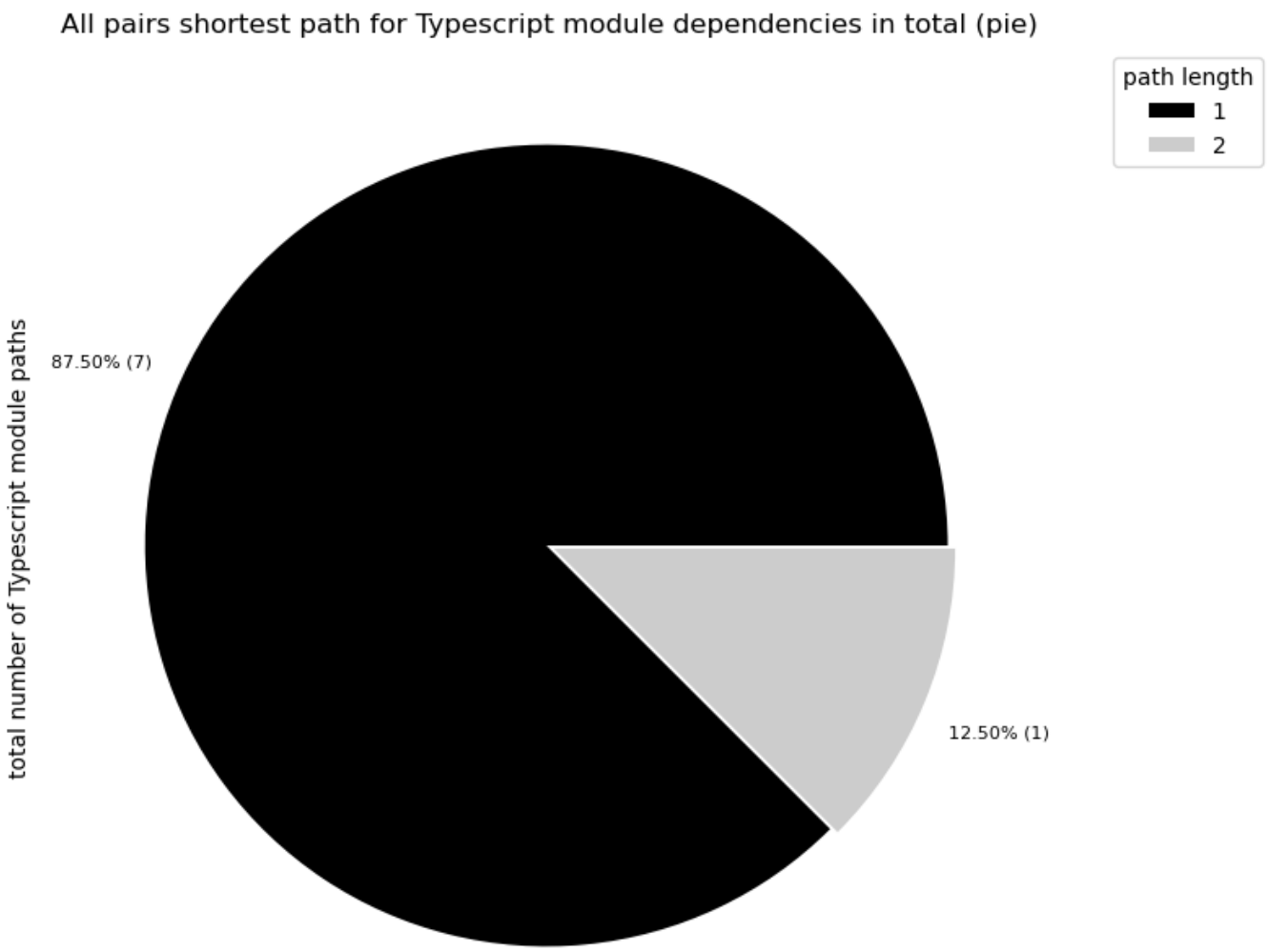
	index	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
0	0	1	7	4	3
1	4	2	1	1	1

All pairs shortest path in total - Path count per length - Bar chart



All pairs shortest path in total - Path count per length - Pie chart

```
<Figure size 640x480 with 0 Axes>
```



1.1.3 All pairs shortest path in detail

The following table shows the first 10 rows with all details of the query above. It contains the results of the "all pairs shortest path" algorithm including the project the source node belong to and if the target node is the same or not. The main intuition here is to show how the data is structured. It provides the basis for tables and charts shown in following sections below, that filter and group the data accordingly.

	sourceProject	sourceScan	isDifferentTargetProject	isDifferentTargetScan	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
0	react-router	react-router-6.26.2	True	False	1	7	4	3
1	react-router-dom	react-router-6.26.2	False	False	1	7	4	3
2	react-router-dom	react-router-6.26.2	True	False	1	7	4	3
3	react-router-native	react-router-6.26.2	True	False	1	7	4	3
4	react-router-native	react-router-6.26.2	True	False	2	1	1	1

1.1.4 All pairs shortest path for each project

In this section we'll focus only on pairs of nodes that both belong to the same project, filtering out every line that has `isDifferentTargetProject==False`. The first ten rows are shown in a table followed by charts that show the distribution of shortest path distances across different projects in stacked bar charts (absolute and normalized).

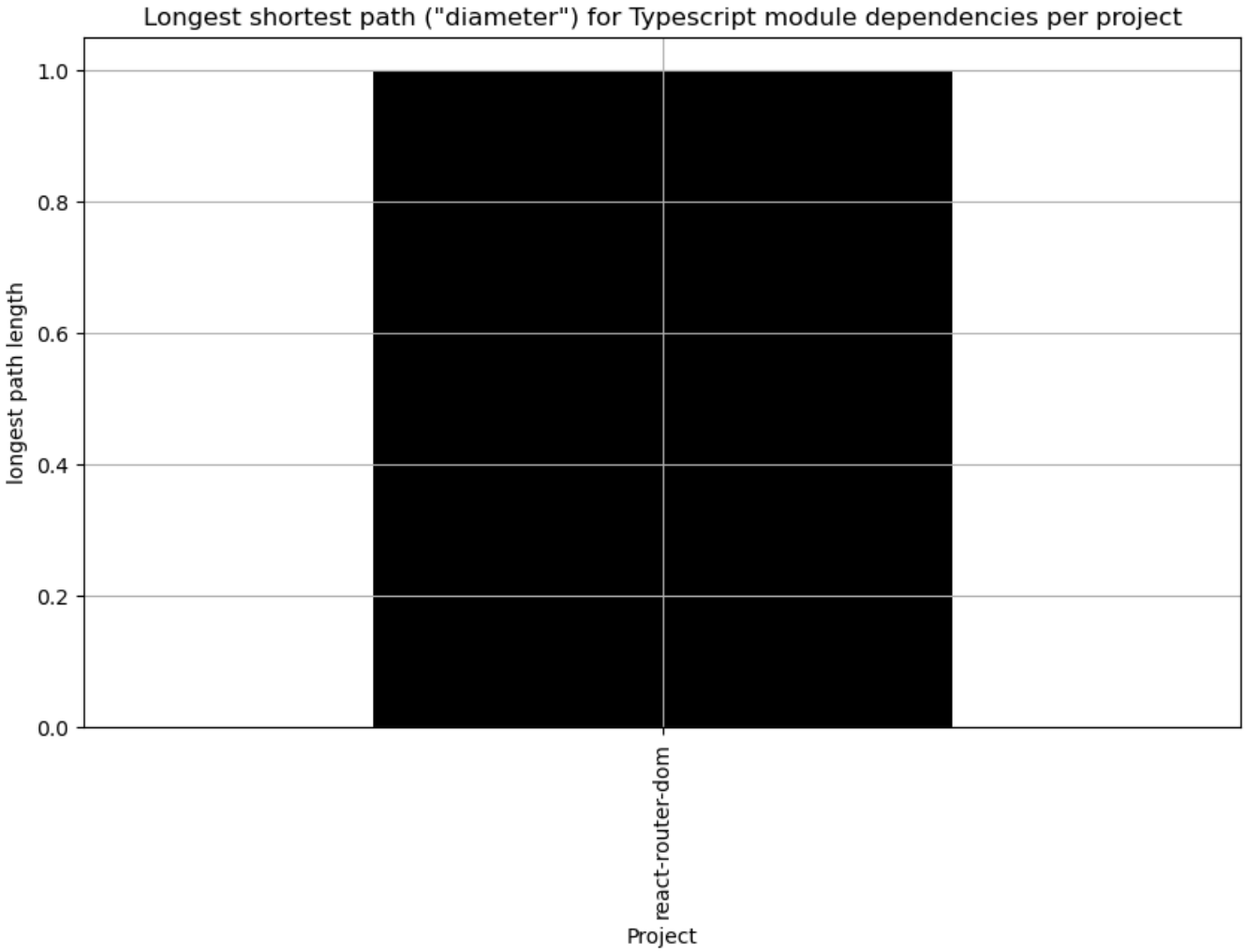
Note: It is possible that a (shortest) path could have nodes in between that belong to different projects. Therefore, the data of each project isn't perfectly isolated. However, it shows how the dependencies interact across projects "in real life" while still providing a decent isolation of each project.

	sourceProject	sourceScan	isDifferentTargetProject	isDifferentTargetScan	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
1	react-router-dom	react-router-6.26.2	False	False	1	7	4	3

All pairs shortest path for each project - Longest shortest path (Diameter) for each project

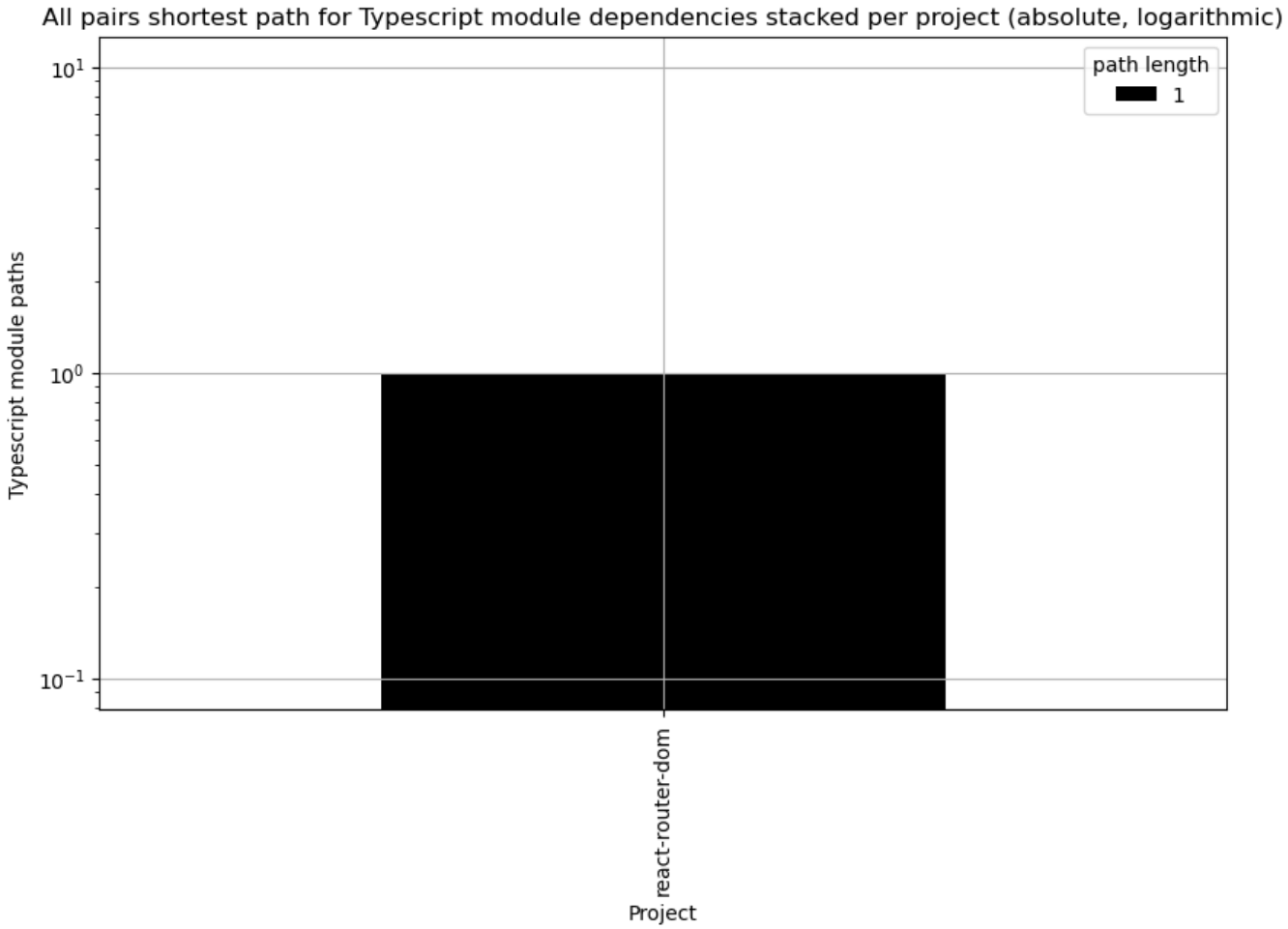
Shows the top 20 projects with the longest shortest path (=Graph Diameter).

```
sourceProject
react-router-dom    1
Name: distance, dtype: int64
```



All pairs shortest path for each project - Bar chart (absolute)

```
<Figure size 640x480 with 0 Axes>
```

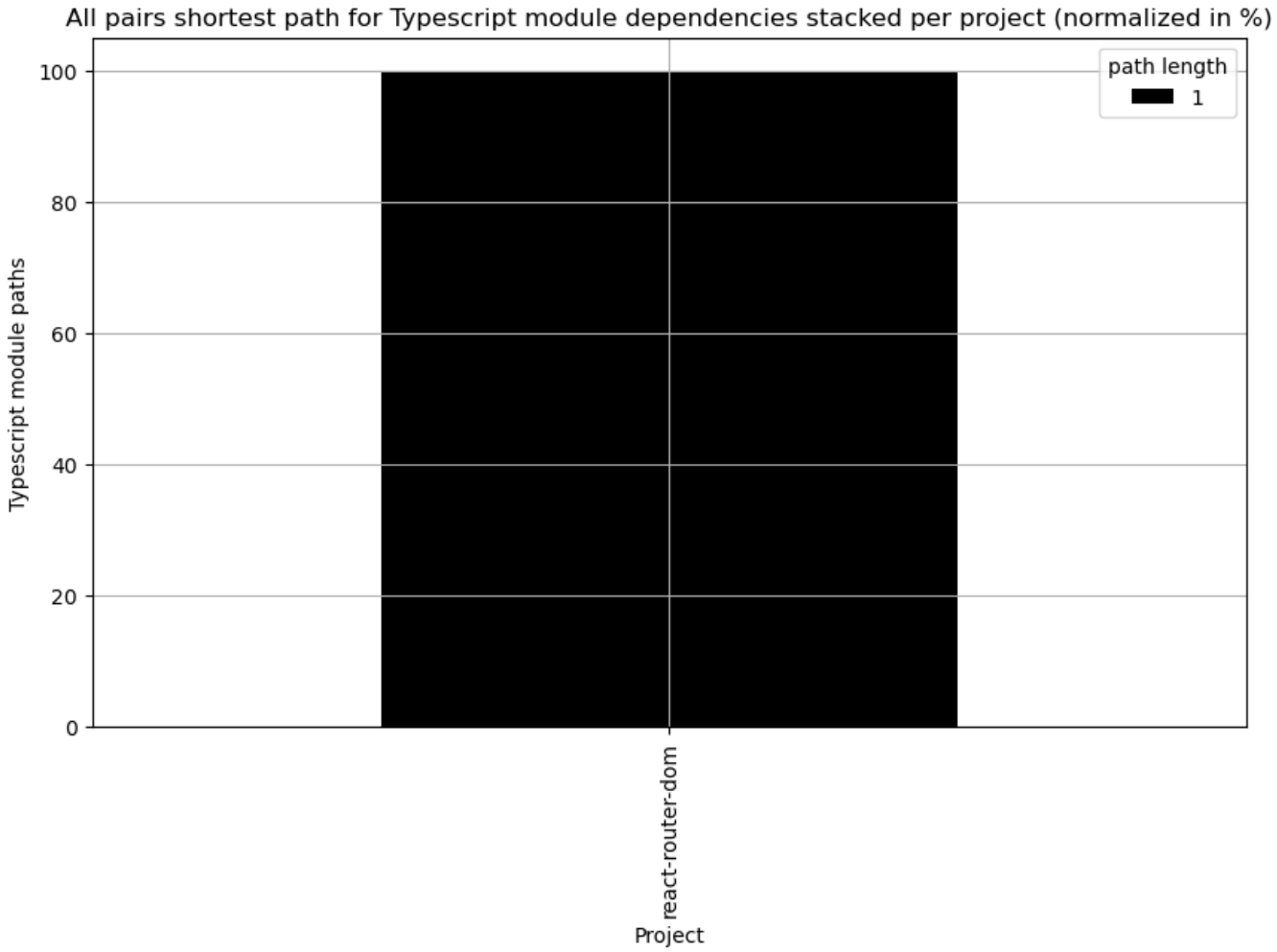


All pairs shortest path for each project - Bar chart (normalized)

Shows the top 50 projects with the highest number of dependency paths stacked by their length.

distance	1
sourceProject	
react-router-dom	100.0

```
<Figure size 640x480 with 0 Axes>
```



1.1.5 All pairs shortest path for each scan

In this section we'll focus only on pairs of nodes that both belong to the same scan, filtering out every line that has `isDifferentTargetScan==False`. The first ten rows are shown in a table followed by charts that show the distribution of shortest path distances across different scans in stacked bar charts (absolute and normalized).

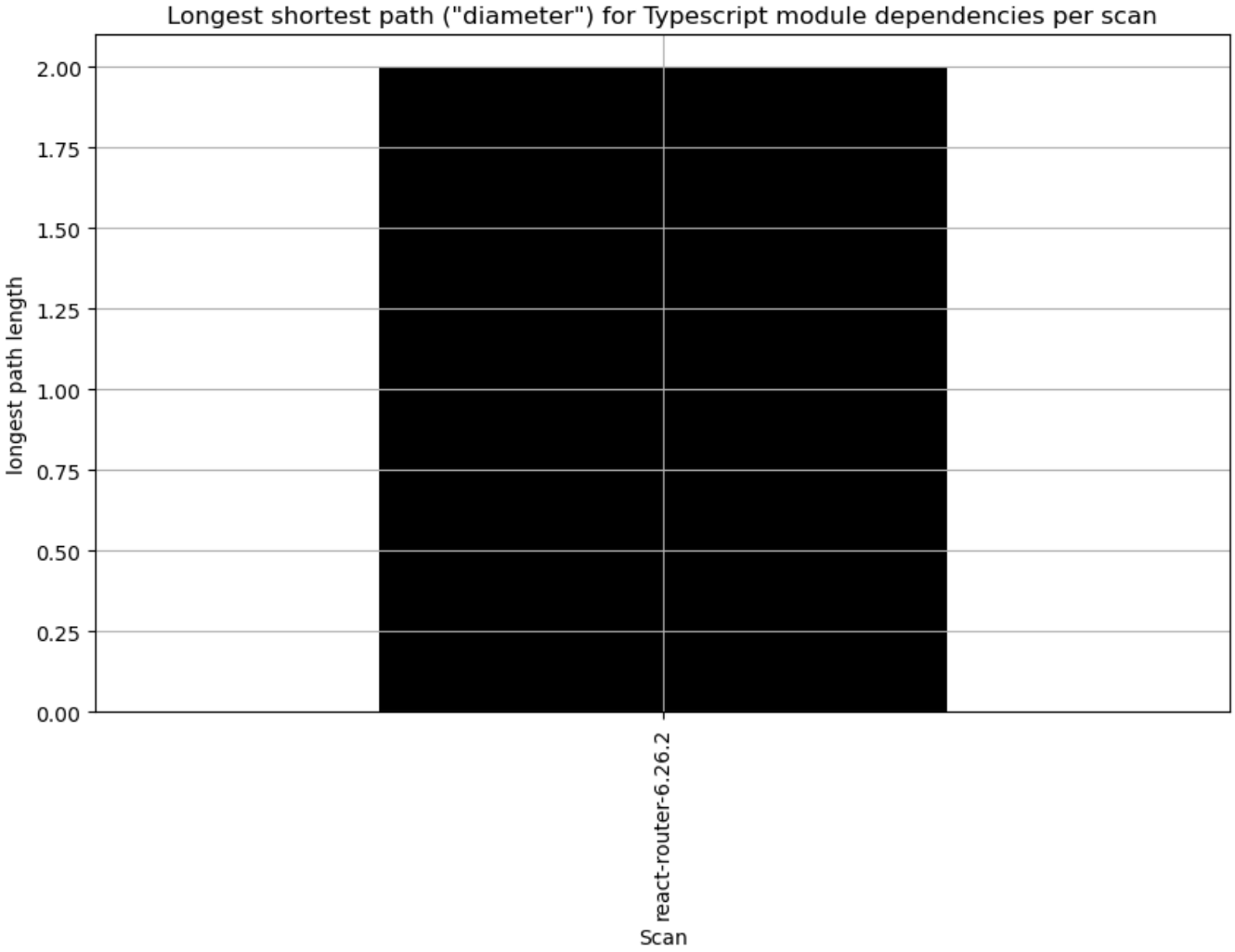
Note: It is possible that a (shortest) path could have nodes in between that belong to different scans. Therefore, the data of each scan isn't perfectly isolated. However, it shows how the dependencies interact across scans "in real life" while still providing a decent isolation of each scan.

	sourceScan	distance	pairCount	sourceNodeCount	targetNodeCount
0	react-router-6.26.2	1	4	2	2
1	react-router-6.26.2	2	1	1	1

All pairs shortest path for each scan - Longest shortest path (Diameter) for each scan

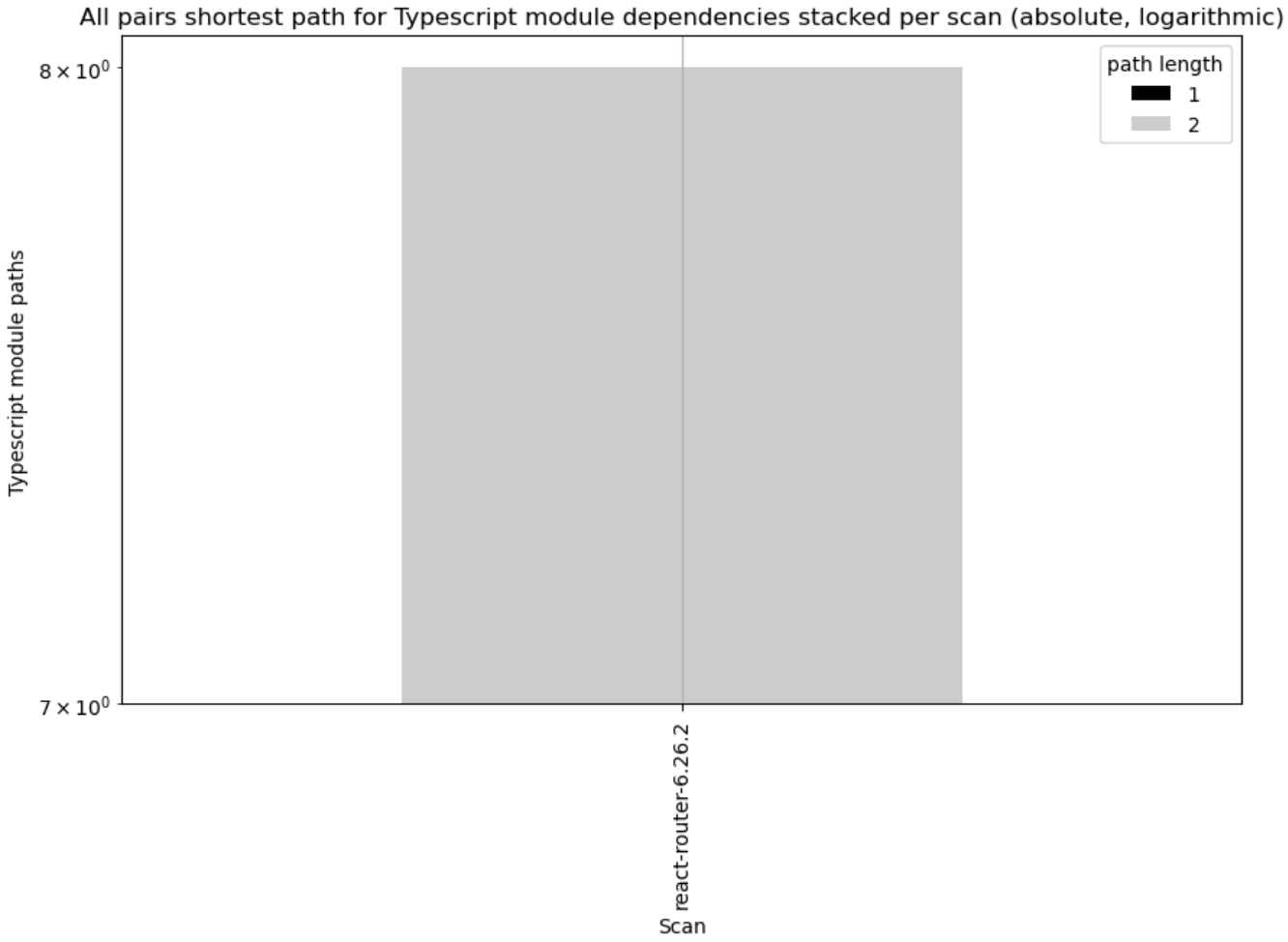
Shows the top 20 scans with the longest shortest path (=Graph Diameter).

```
sourceScan
react-router-6.26.2    2
Name: distance, dtype: int64
```



All pairs shortest path for each scan - Bar chart (absolute)

```
<Figure size 640x480 with 0 Axes>
```

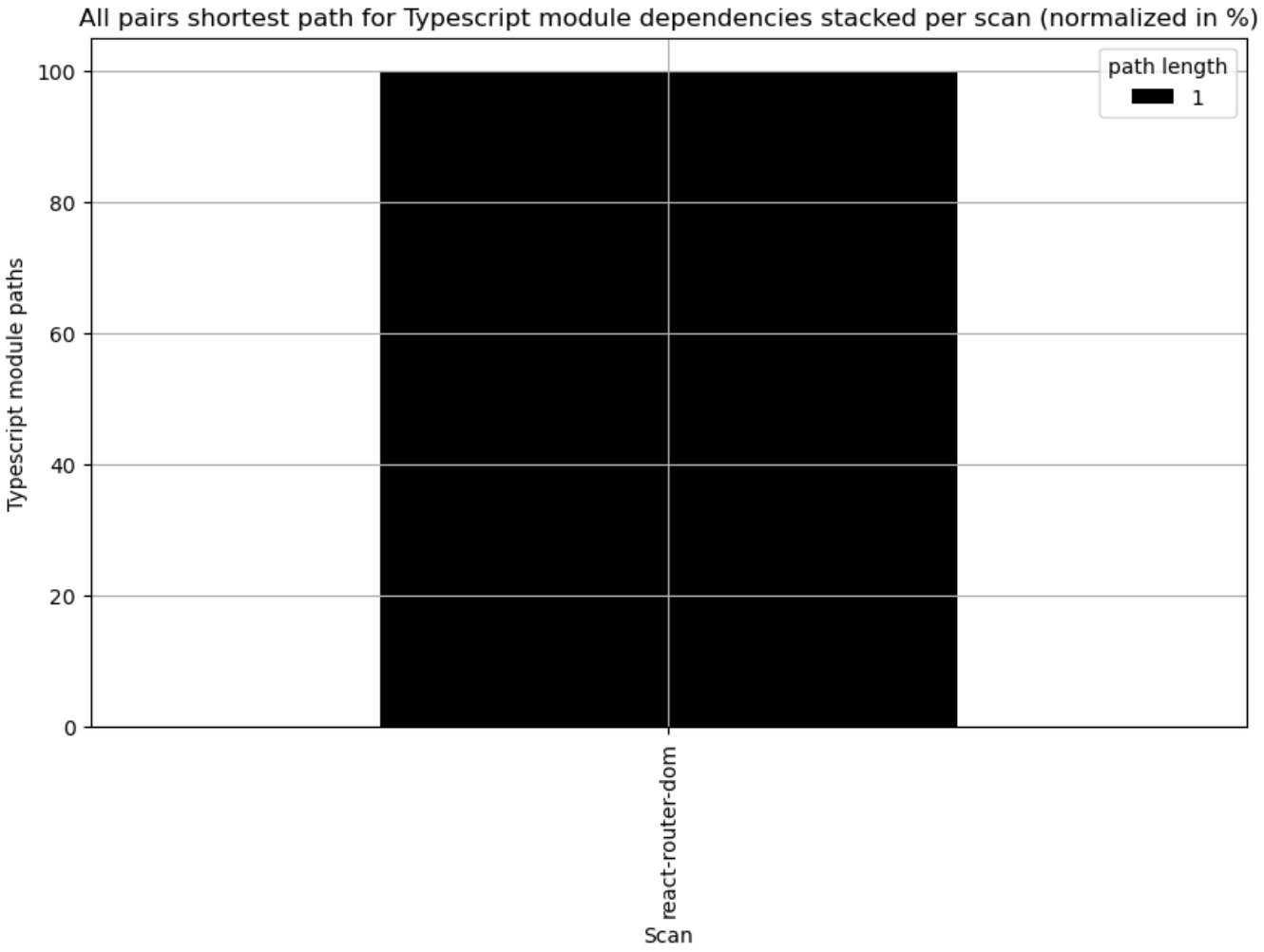


All pairs shortest path for each scan - Bar chart (normalized)

Shows the top 50 scans with the highest number of dependency paths stacked by their length.

distance	1
sourceProject	
react-router-dom	100.0

```
<Figure size 640x480 with 0 Axes>
```



1.2 Longest path

Use Longest Path algorithm to get the longest paths between Typescript packages. It is typically higher than the longest shortest path (diameter) and helps together with it to get a good overview of the complexity.

Note: This algorithm requires a Directed Acyclic Graph (DAG) and will lead to inaccurate results when the Graph contains cycles.

1.2.1 Longest path in total

First, we'll have a look at the overall/total result of the longest path algorithm for all dependencies.

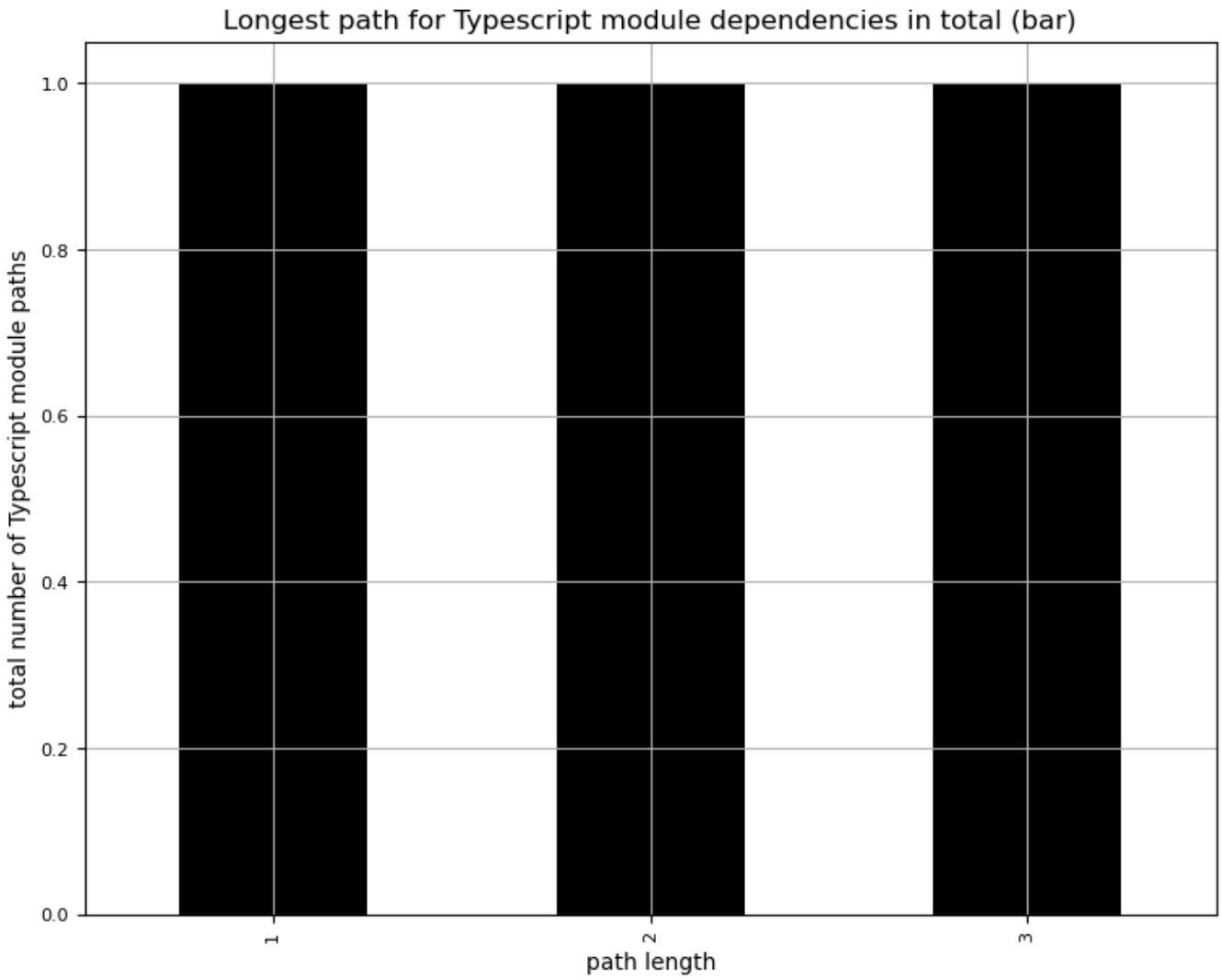
Longest path in total - Max longest path

```
The max. longest path of the projected module dependencies is: 3
```

Longest path in total - Paths per length - Table

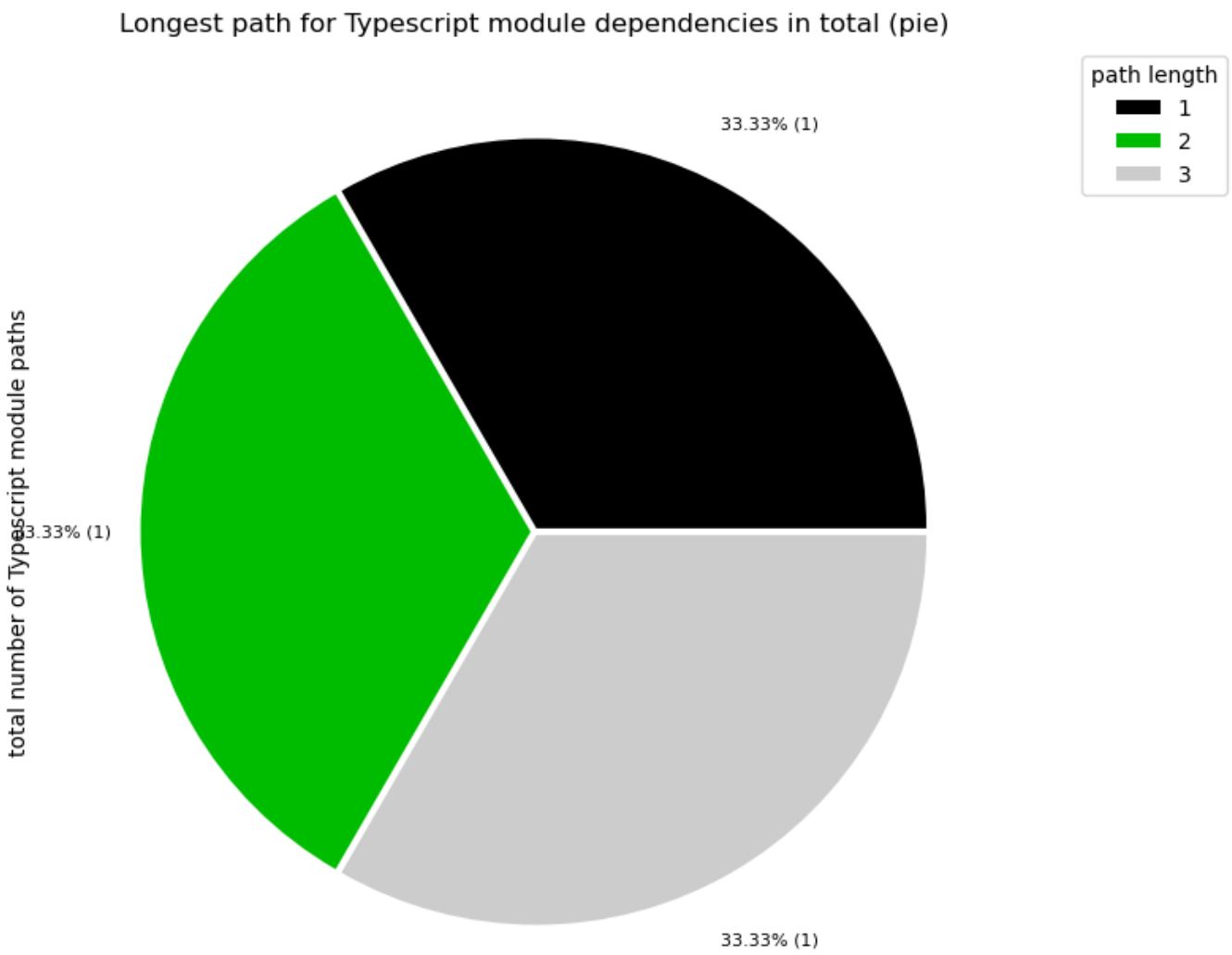
	index	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
0	0	1	1	1	1
1	1	2	1	1	1
2	2	3	1	1	1

Longest path in total - Path count per length - Bar chart



Longest path in total - Path count per length - Pie chart

```
<Figure size 640x480 with 0 Axes>
```



1.2.2 Longest path in detail

The following table shows the first 10 rows with all details of the query above. It contains the results of the "longest path" algorithm including the project the source node belongs to and if the target node is in the same project or not. The main intuition is to show how the data is structured. It provides the basis for tables and charts shown in following sections below, that filter and group the data accordingly.

	sourceProject	sourceScan	isDifferentTargetProject	isDifferentTargetScan	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
0	react-router-dom	react-router-6.26.2	False	False	1	1	1	1
1	react-router-dom	react-router-6.26.2	True	False	2	1	1	1
2	react-router-dom	react-router-6.26.2	True	False	3	1	1	1

1.2.3 Longest path for each project

In this section we'll focus only on pairs of nodes that both belong to the same project, filtering out every line that has `isDifferentTargetProject==False`. The first ten rows are shown in a table

followed by charts that show the distribution of longest path distances across different projects in stacked bar charts (absolute and normalized).

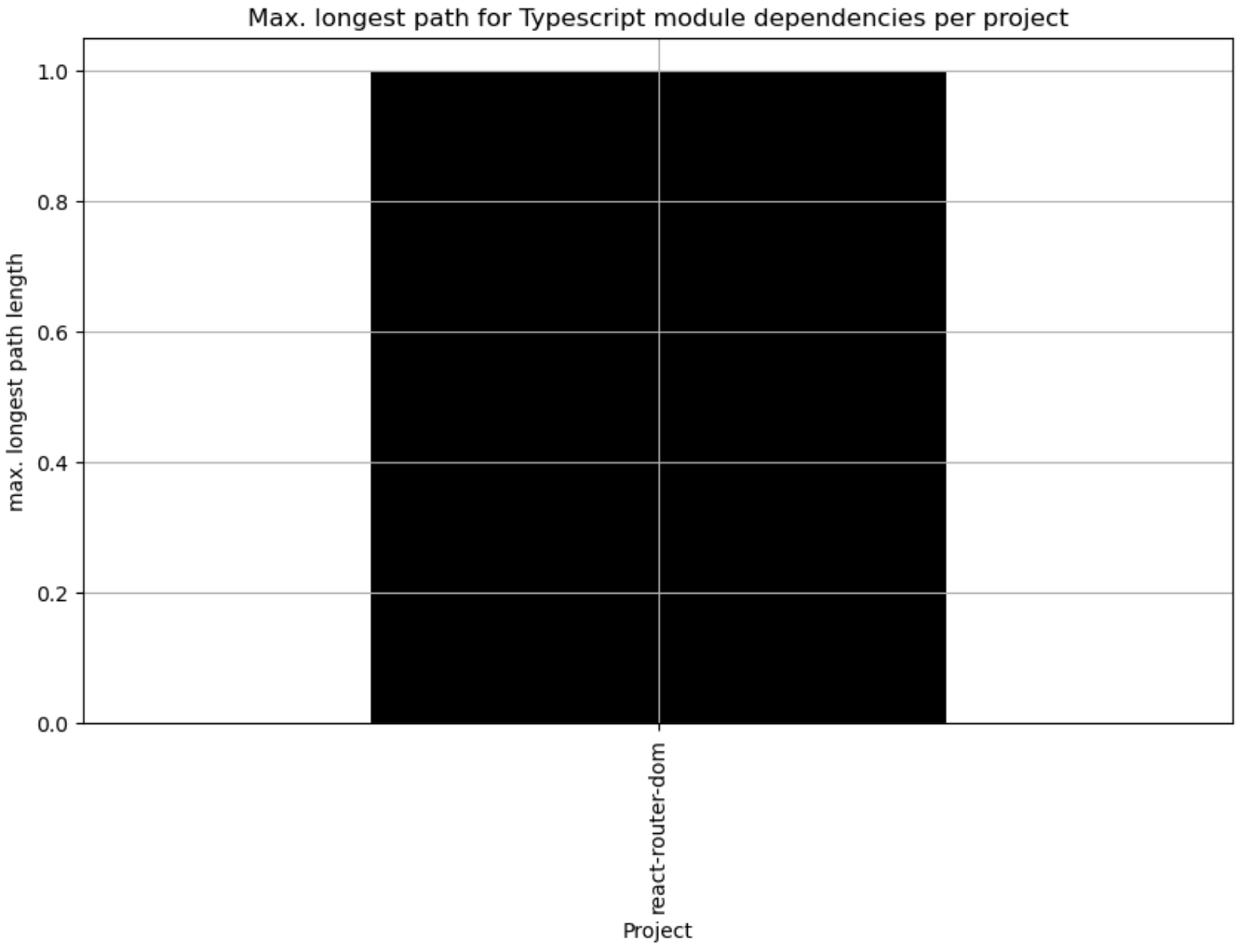
Note: It is possible that a (longest) path could have nodes in between that belong to different projects. Therefore, the data of each project isn't perfectly isolated. However, it shows how the dependencies interact across projects "in real life" while still providing a decent isolation of each project.

	sourceProject	sourceScan	isDifferentTargetProject	isDifferentTargetScan	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
0	react-router-dom	react-router-6.26.2	False	False	1	1	1	1

Longest path for each project - Max. longest path for each project

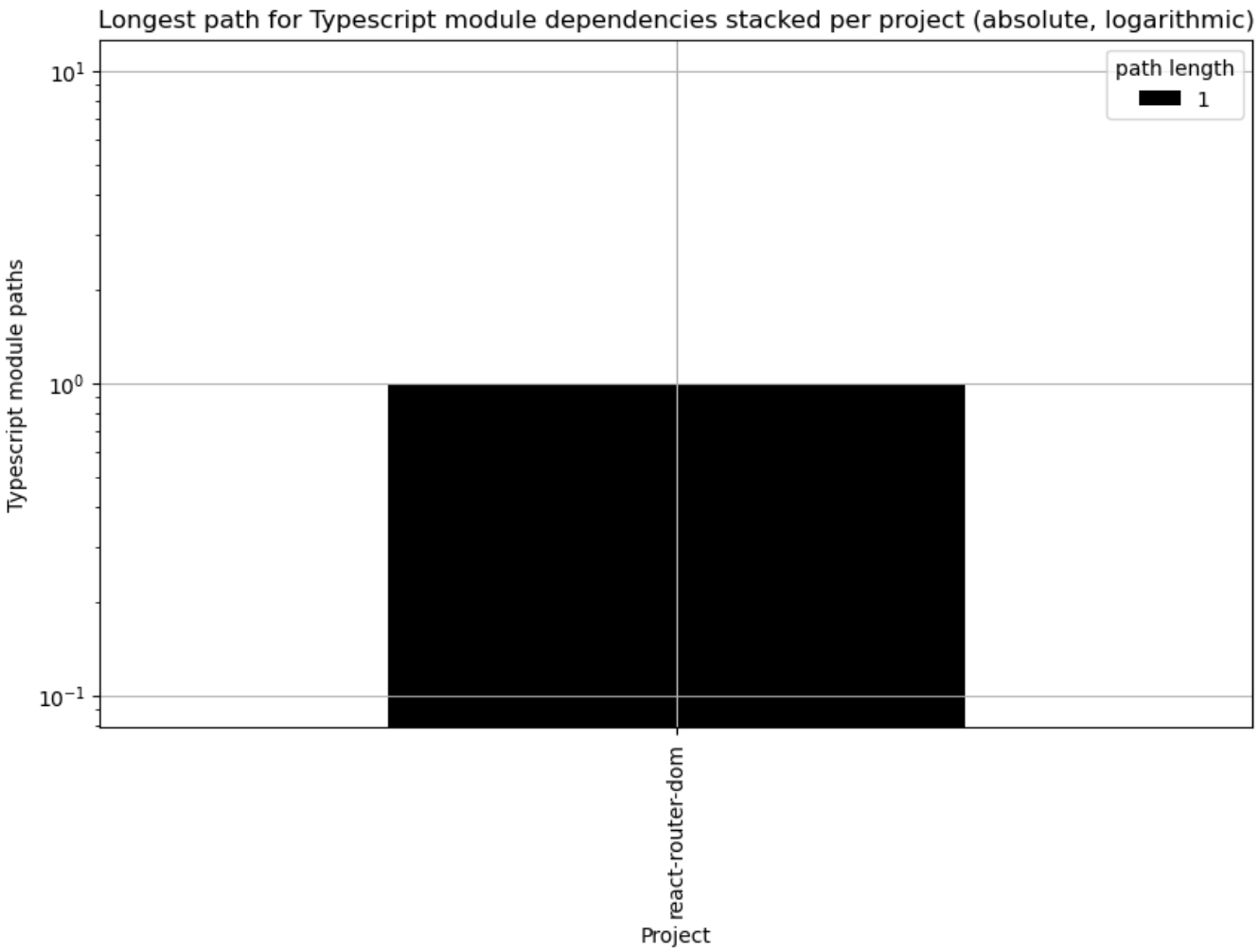
Shows the top 20 projects with their max. longest path.

```
sourceProject
react-router-dom    1
Name: distance, dtype: int64
```



Longest path for each project - Bar chart (absolute)

<Figure size 640x480 with 0 Axes>

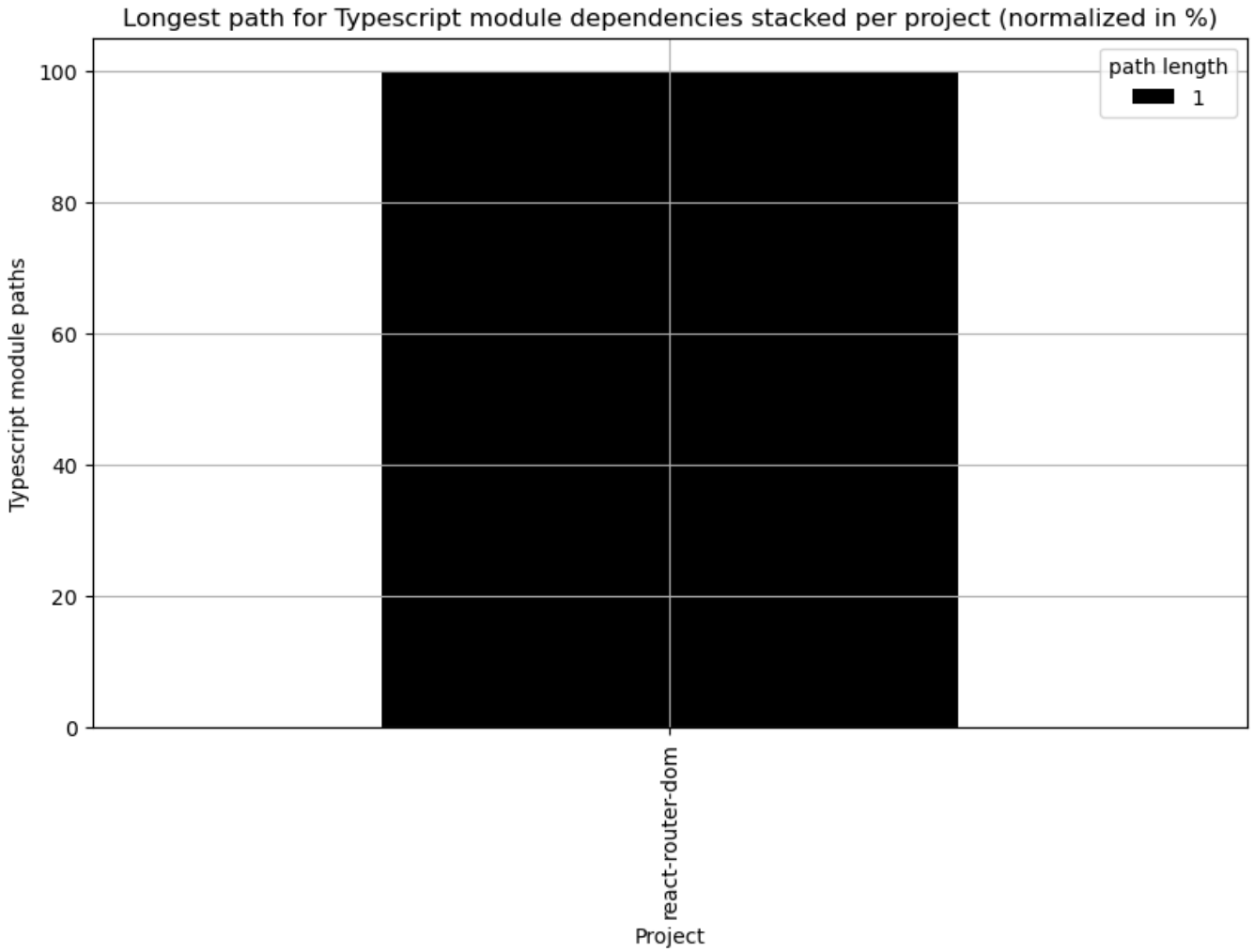


Longest path for each project - Bar chart (normalized)

Shows the top 50 projects with the highest number of dependency paths stacked by their length.

distance	1
sourceProject	
react-router-dom	100.0

<Figure size 640x480 with 0 Axes>



1.2.4 Longest path for each scan

In this section we'll focus only on pairs of nodes that both belong to the same scan, filtering out every line that has `isDifferentTargetScan==False`. The first ten rows are shown in a table followed by charts that show the distribution of longest path distances across different scans in stacked bar charts (absolute and normalized).

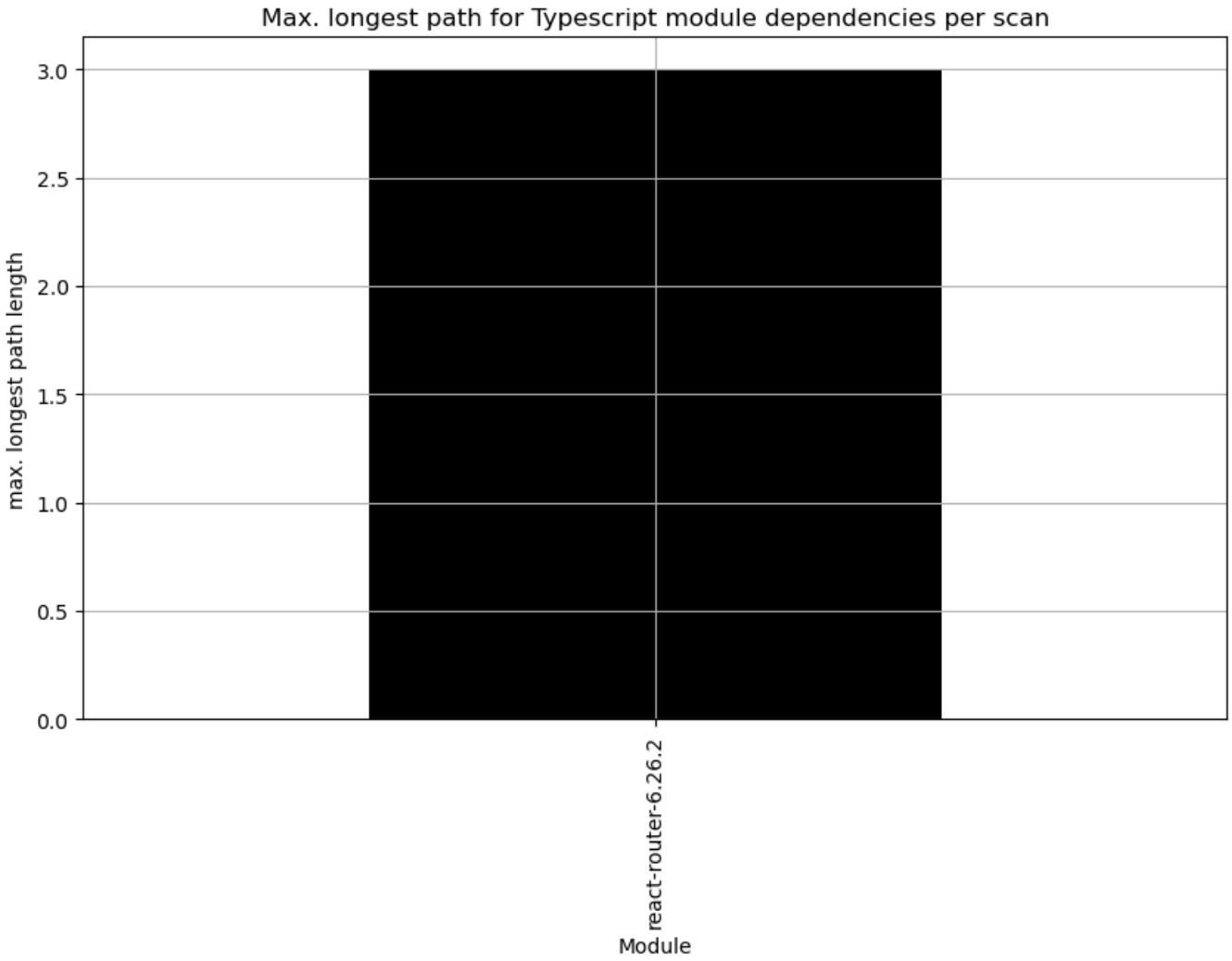
Note: It is possible that a (longest) path could have nodes in-between that belong to different scans. Therefore, the data of each scan isn't perfectly isolated. However, it shows how the dependencies interact across scans "in real life" while still providing a decent amount of isolation of each scan.

	sourceProject	sourceScan	isDifferentTargetProject	isDifferentTargetScan	distance	distanceTotalPairCount	distanceTotalSourceCount	distanceTotalTargetCount
0	react-router-dom	react-router-6.26.2	False	False	1	1	1	1
1	react-router-dom	react-router-6.26.2	True	False	2	1	1	1
2	react-router-dom	react-router-6.26.2	True	False	3	1	1	1

Longest path for each scan - Max. longest path for each scan

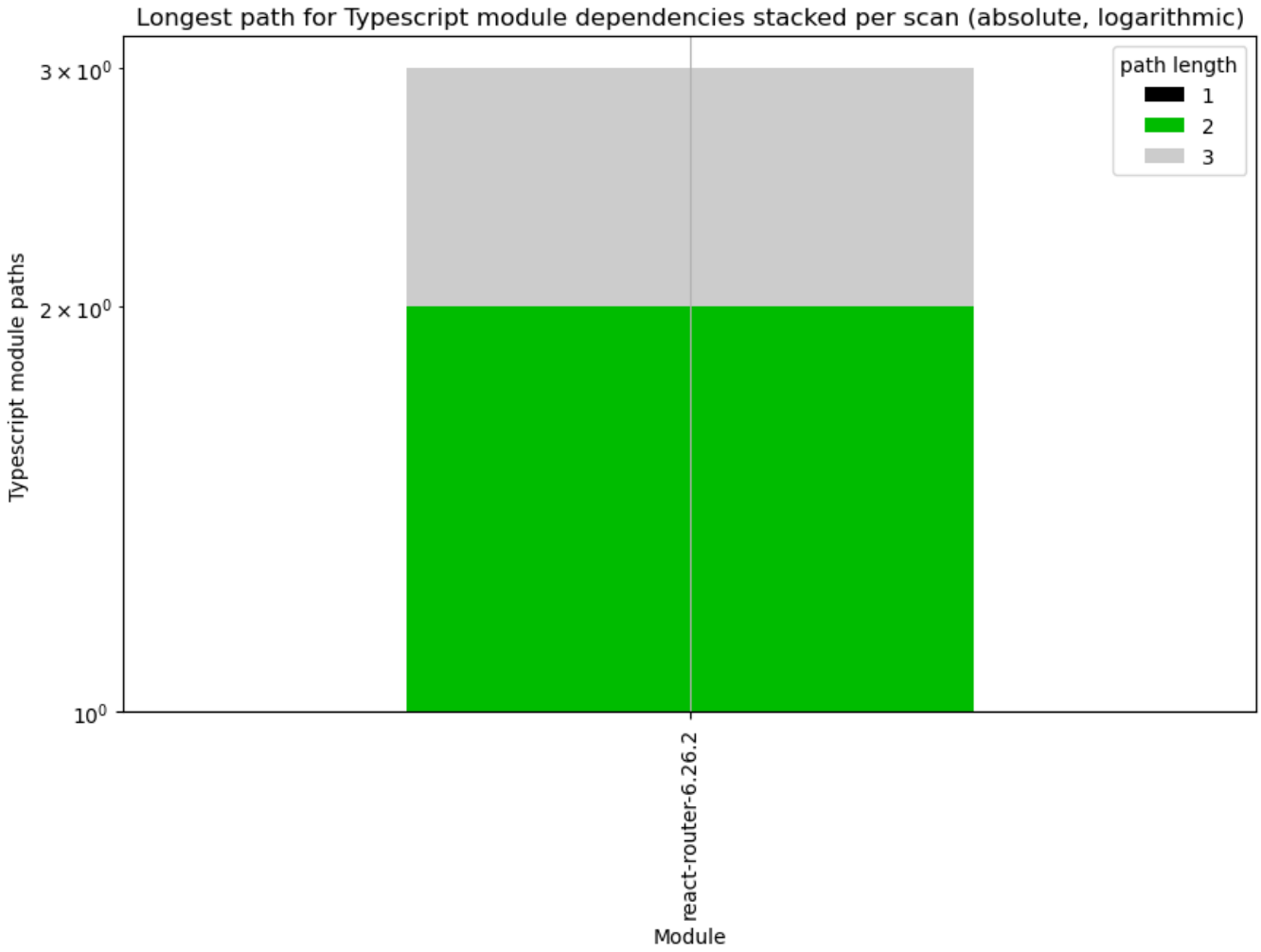
Shows the top 20 scans with their max. longest path.

```
sourceScan
react-router-6.26.2    3
Name: distance, dtype: int64
```



Longest path for each scan - Bar chart (absolute)

```
<Figure size 640x480 with 0 Axes>
```

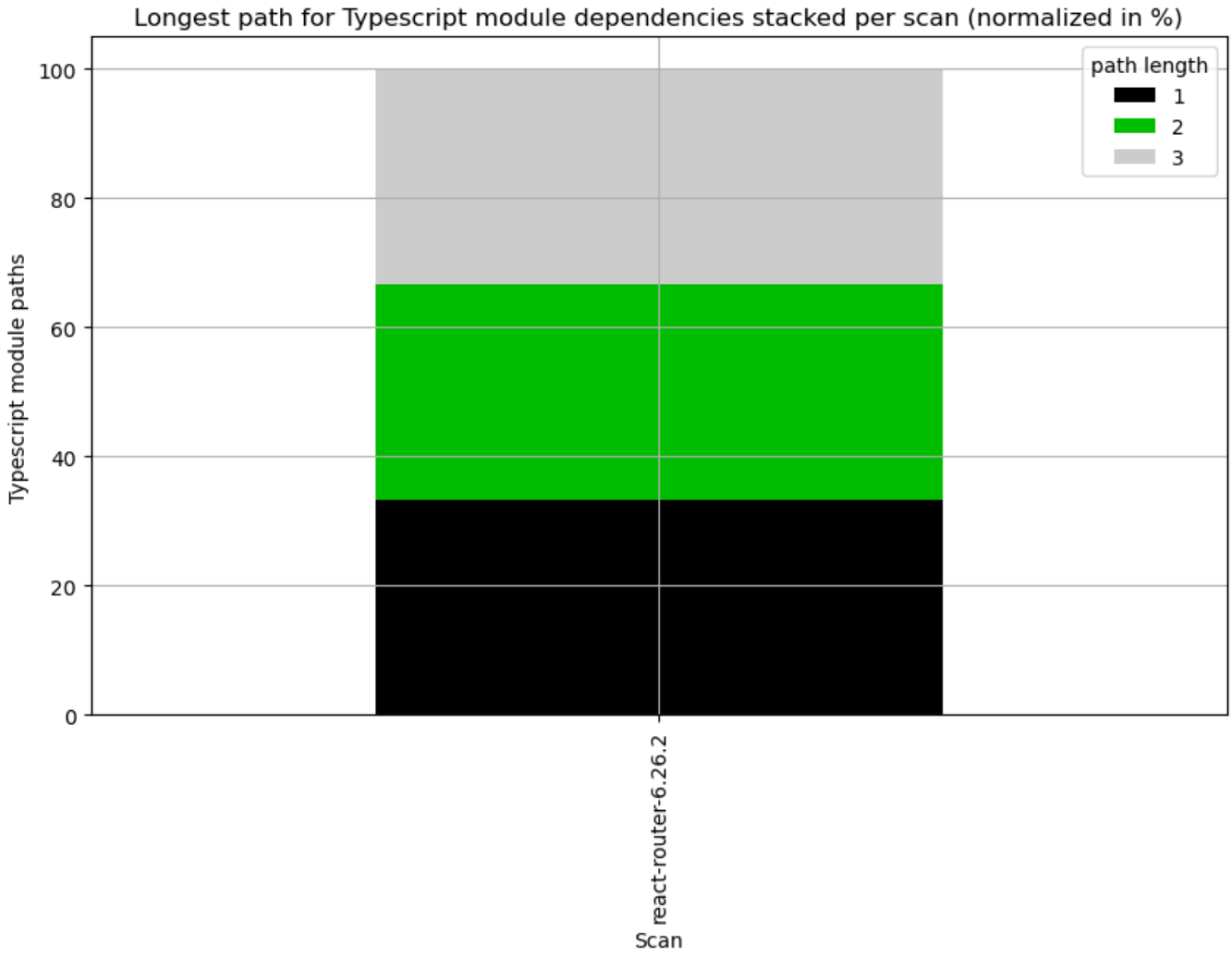


Longest path for each scan - Bar chart (normalized)

Shows the top 50 scans with the highest number of dependency paths stacked by their length.

distance	1	2	3
sourceScan			
react-router-6.26.2	33.333333	33.333333	33.333333

```
<Figure size 640x480 with 0 Axes>
```



3. Summary

3.1 Typescript modules summary

	count	degree density	degree median	degree max	longest shortest path (diameter)	max. longest path
0	5	0.35	1	3	2	3