Building Code Cities using the Language Server Protocol

Master's Thesis Presentation

Falko Galperin

Faculty 3—Mathematics & Computer Science
University of Bremen

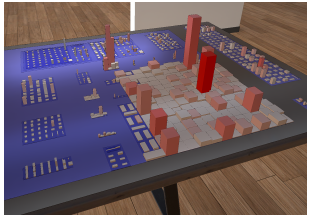
April 29, 2025

Overview

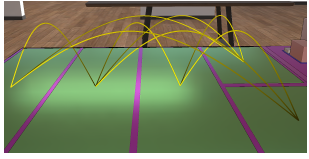
- **Code Cities** can help developers understand complex software systems
  - Often limited to few languages
- The **Language Server Protocol** (LSP) specifies how *Language Servers* can provide language-specific features to IDEs
  - Many Language Servers are available

Goal

☞ Integrate LSP information into code-city implementation SEE



Sample code city (SEE)



Edges connecting nodes (SEE)

Research Questions

Research Question 1

How can LSP be integrated into SEE to generate code cities?

Research Question 2

What is the scalability of this integration?

Research Question 3

Are code cities a suitable means to present LSP information to developers as compared to IDEs + tables (on the dimensions of speed, accuracy, and usability)?

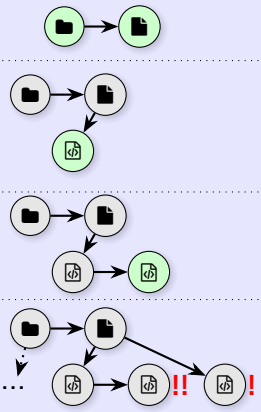
Language Server Protocol

- Open-source specification managed by Microsoft
- JSON-RPC messages sent between Language Client and Language Server
- > 260 Language Servers, > 60 Language Clients
- Specifies **capabilities** to be used by implementers
  - Examples: Go to definition, hover, document symbols
  - We will use only (some) “read-only” capabilities

Part I: Node Synthesis

For each document…

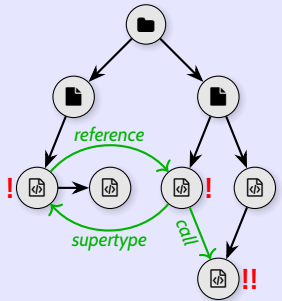
1. Add a node for the document (and its directory).
2. For each symbol in that document…
   - 2.1 Add a child node for the symbol.
   - 2.2 If there are contained symbols, go to 2.1 for each one.
3. Retrieve diagnostics for document and attach to corresponding nodes.



Part II: Edge Synthesis

For each node, connect edge to…

1. …**definition**, if it exists.
2. …**declaration**, if it exists.
3. …**type definition**, if it exists.
4. …**implementation**, if it exists.
5. …any **references**.
6. …any **outgoing calls** using call hierarchy.
7. …any **supertypes** using type hierarchy.

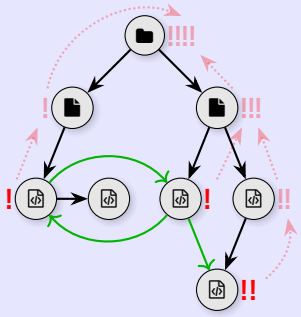


Part III: Aggregation

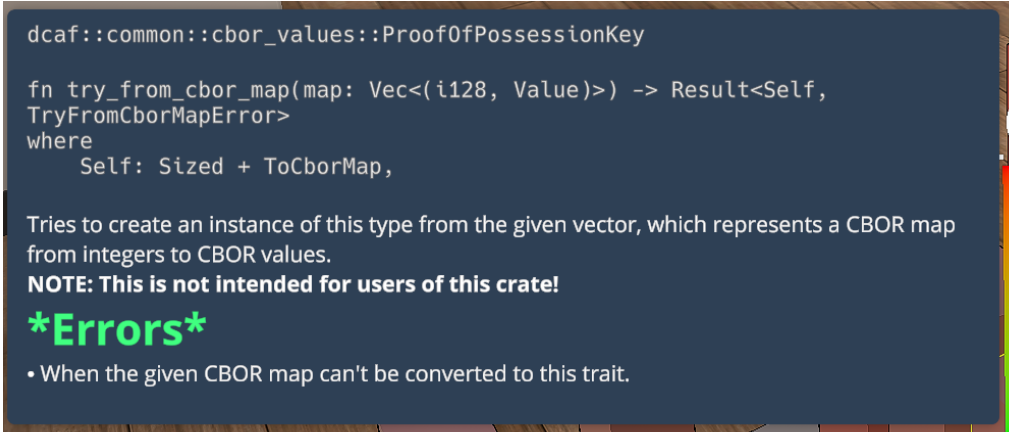
For each root node…

1. Aggregate LOC upwards.
2. Aggregate diagnostic counts upwards.

Return constructed graph.

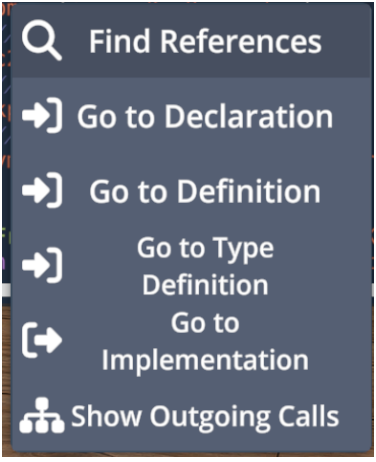


Hover Tooltip

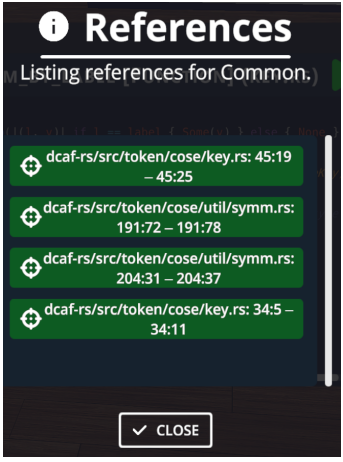


A hover tooltip with information collected from the `rust-analyzer` Language Server.

Navigation

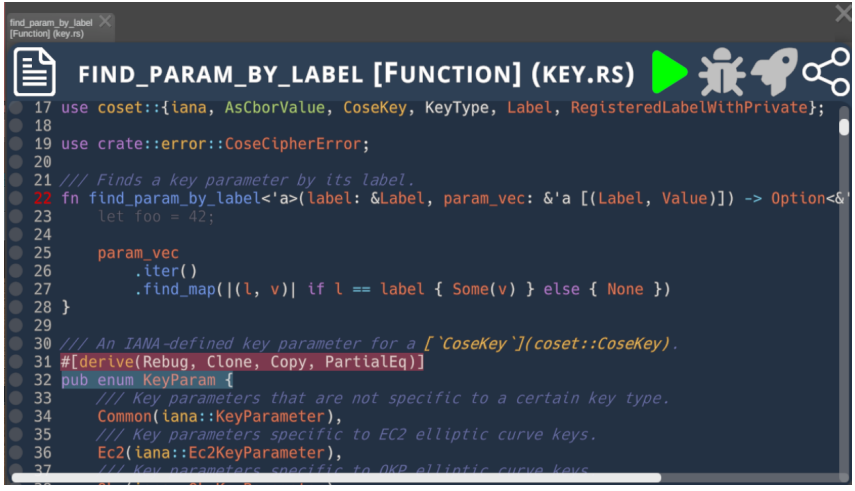


Menu shown when right-clicking an element.



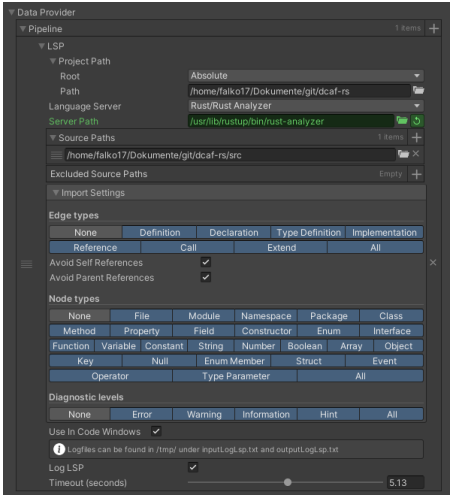
Selecting an LSP reference to navigate to.

Code Windows



A code window with enabled LSP integration.

Generating Cities



Unity Editor UI for the LSP graph provider.

Research Question 1

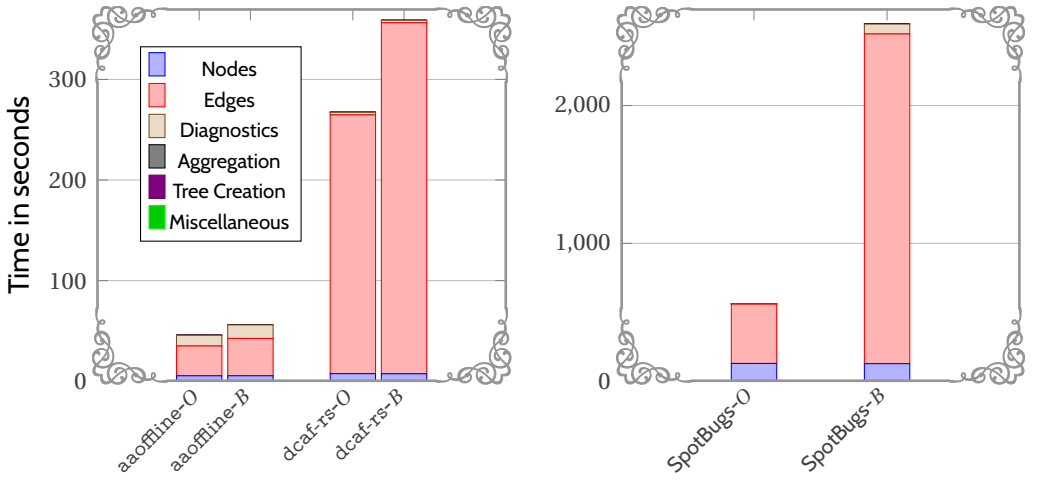
How can LSP be integrated into SEE to generate code cities?

☞ Answered in previous slides

Evaluation Setup

- Ran generation algorithm on six projects in total: two per Java, Rust, LaTeX
- Measured average execution time across multiple runs
- *Only* generation algorithm included in measurement
  - E. g., no layouting or visual rendering of nodes
- Later followed up in paper submitted to ICSME 2025

Evaluation Results—Time Breakdown



Generation time for Rust (left) and Java (right) projects, broken down by parts of the algorithm. The suffix *O* denotes the optimized version of the algorithm, while *B* refers to the brute-force version.

Evaluation Results—Conclusion

- Most important metric by far: Number of edges
- Optimized variant always more performant than brute-force approach
  - $\mathcal{O}(|E| \cdot \log |V|)$ vs. $\Theta(|E| \cdot |V|)$
- Time taken only comparable within the same Language Server

Research Question 2

What is the scalability of this integration?

☞ Feasible until ≈ 200 kLOC (but very dependent on configuration and Language Server). Similar results for many more language servers and projects as part of ICSME 2025 paper.

Study Design

- User study comparing SEE + LSP with VSCode + LSP
- Most LSP capabilities are hard to evaluate
  - ☞ Hence, we focus on the generated city itself

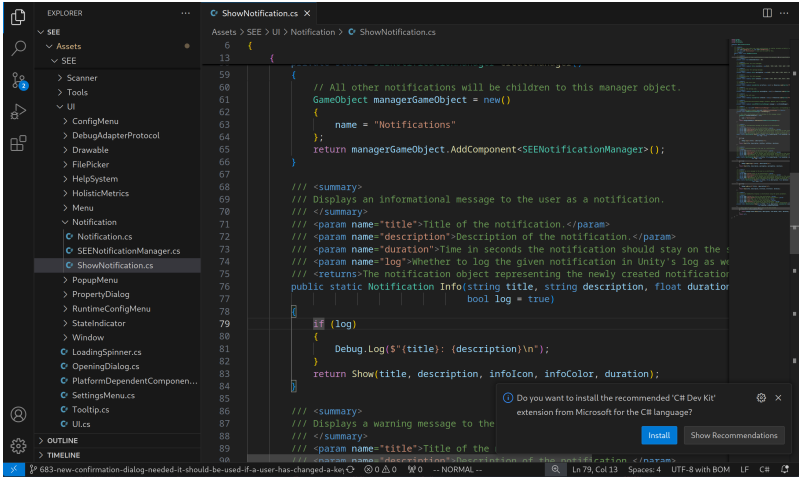
We measure five dependent variables:

1. **Correctness**
2. **Speed**
3. **Usability**, differentiating between:
   - 3.1 **SUS** (post-study)
   - 3.2 **ASQ** (post-task), measuring:
     - 3.2.1 Perceived **complexity**
     - 3.2.2 Perceived **effort**

Research Question 3

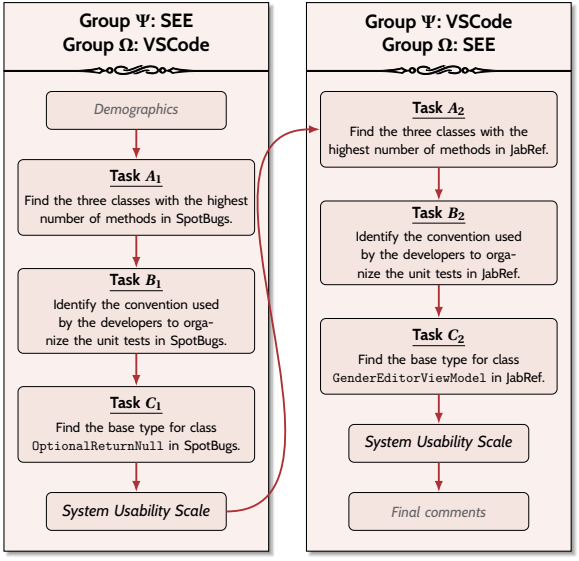
Are code cities a suitable means to present LSP information to developers as compared to IDEs + tables (on the dimensions of speed, accuracy, and usability)?

VSCode



Screenshot of the main UI of VSCode.

Tasks

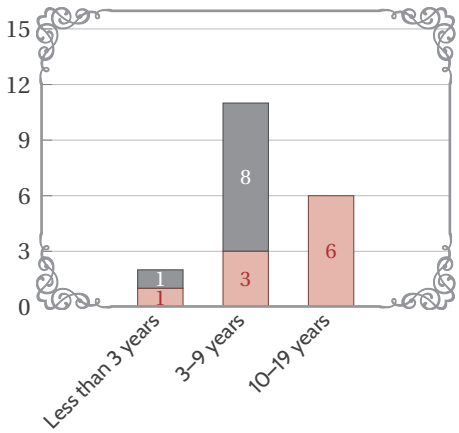


- Within-subject study
- Three tasks per condition
- *JabRef* and *SpotBugs* used as object systems
- ASQ after each task
- SUS after each condition

Study Results

- $N = 19$ participants
- Significance level of $\alpha = 0.05$
- Used *Mann-Whitney U test* as statistical test (with some exceptions)

"How long have you been programming?"



Correctness

- Answers manually checked for correctness (to catch, e. g., misspellings)
- *Fisher's exact test* used due to binary results
- No significant differences for any of the tasks

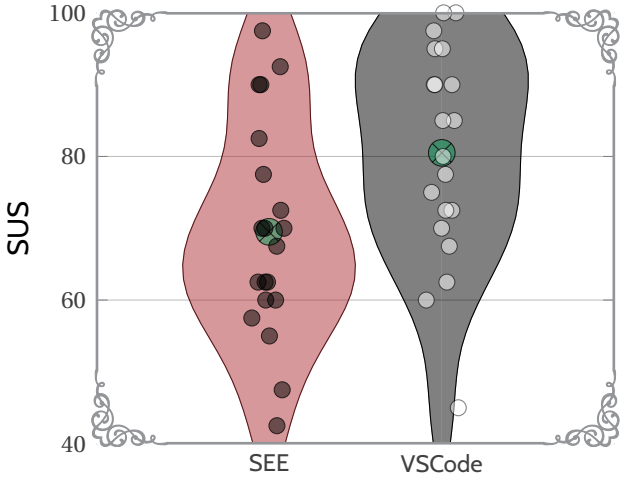
Time

- Time measured automatically
- Only correct answers included in analysis
- Significant differences for task C (reach base class) in favor of VSCode
  - ☞ Potential cause: following edges in SEE requires more navigation/interaction than just repeatedly Ctrl-clicking in VSCode

Usability: ASQ

- Two questions asked after each task:
  - "Overall, I am satisfied with the ease of completing the tasks in this scenario."
  - "Overall, I am satisfied with the amount of time it took to complete the tasks in this scenario."
- Complexity (Ease): No significant differences for any task
- Effort (Time): Significant differences for tasks A_2 and C_1 in favor of VSCode
  - Participants indeed took longer to solve tasks A_2 and C_1 in SEE

Usability: SUS

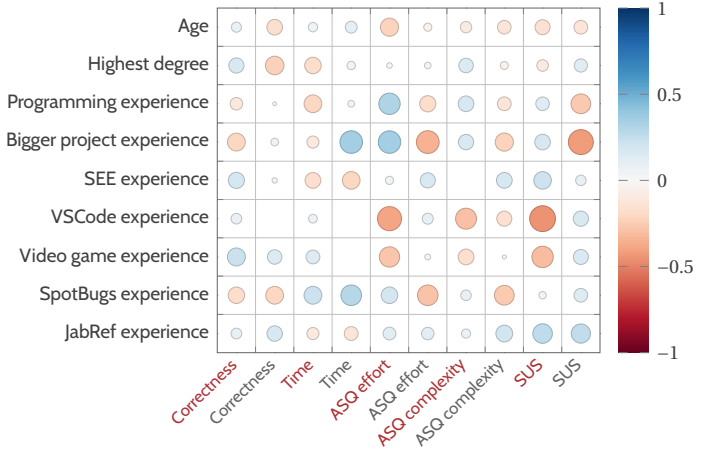


- Ten questions asked about each system
- *Wilcoxon signed-rank test* used due to dependent samples
- Significant difference in favor of VSCode
  - SUS score falls in line with previous usability experiments involving SEE

The Effects of Experience

- We want to find correlations between independent and dependent variables
  - Especially to check if, e. g., experience may bias our results
- *Kendall's coefficient of rank correlation* τ_b used as statistical test
  - However, at 90 tests, we run into the multiple comparisons problem
  - Hence, *Benjamini-Yekutieli procedure* used to fix False Discovery Rate (FDR) at 0.05
- Result: No significant correlations for any pair of variables (after FDR correction)

The Effects of Experience—Heatmap



Correlations (τ_b) between independent and dependent variables. Dependent variables for SEE are marked in red and those for VSCode in gray.

Summary

Significant differences between the variables, all in favor of VSCode.

Variable	A_1	B_1	C_1	A_2	B_2	C_2
Correctness	—	—	—	—	—	—
Time	—	—	$p \approx 0.0015$	—	—	$p \approx 0.0247$
ASQ: Complexity	—	—	—	—	—	—
ASQ: Effort	—	—	$p \approx 0.0142$	$p \approx 0.00531$	—	—
SUS				$p \approx 0.02116$		

Research Question 3

Are code cities a suitable means to present LSP information to developers as compared to IDEs + tables (on the dimensions of speed, accuracy, and usability)?

☞ While code cities seem suitable to present LSP information, developers work faster and prefer traditional IDEs (at least in the case of SEE vs. VSCode).

Future Work

- Improve usability of SEE
- Improve performance to make it possible for bigger projects to run well
- Add suport for editing-related capabilities to SEE
- Use the Language Server Index Format (LSIF) to construct code cities remotely
- Utilize a registry to automatically download and set up Language Servers in SEE

Thank You!

Any questions?

https://github.com/falko17/masterthesis

Following slides: Appendix

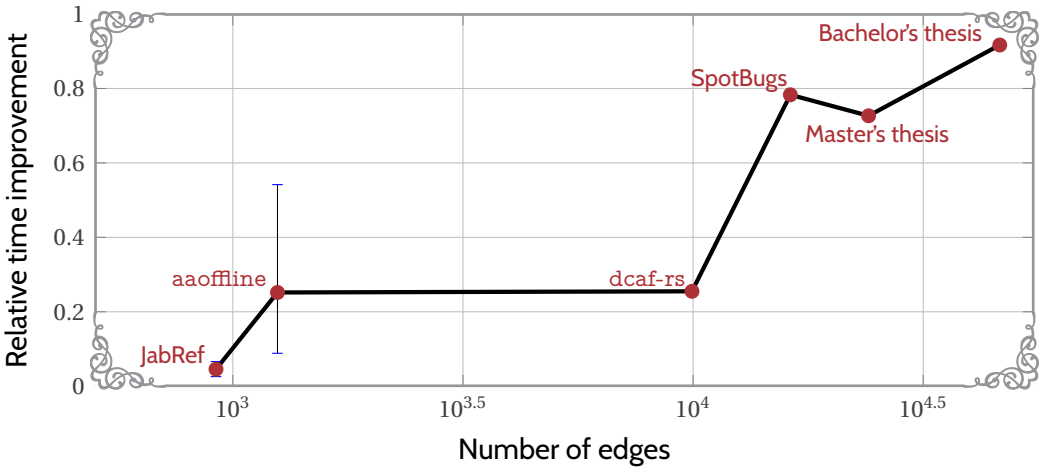
i.e., not part of main presentation

SEE: Graph

Attributed project graph $G = (V, E, a, s, t, \ell)$

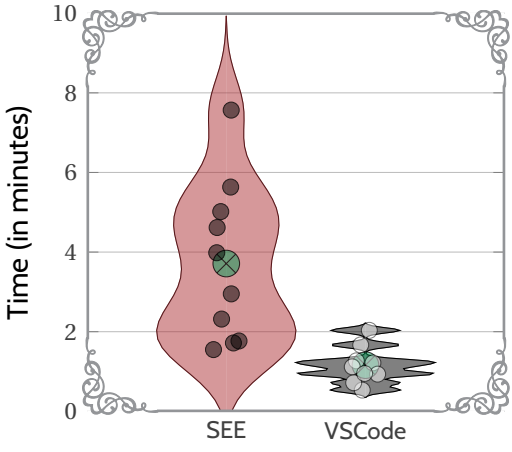
- V: Set of nodes
- E: Set of edges
- $a : (V \times \mathcal{A}_K) \rightharpoonup \mathcal{A}_V$ assigns named attributes to nodes
- $s, t : E \to V$ denotes source/target node of each edge
- $\ell : E \to \Sigma$ for labelling edges
  - Edge label $\texttt{partOf} \in \Sigma$ induces source code hierarchy

Evaluation Results—Optimization Improvement

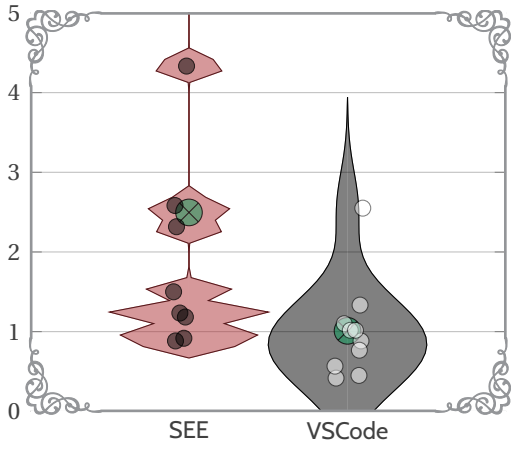


The percentage by which time is reduced when using the optimized instead of the base algorithm.

Time: Significant Differences

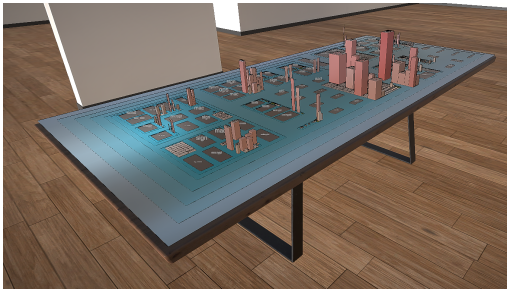


(a) Time taken for task C_1.

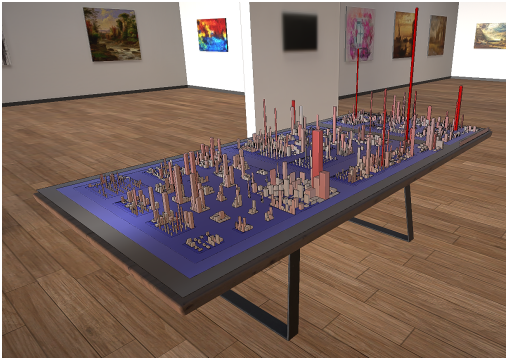


(b) Time taken for task C_2.

Sample Code Cities



Code city for `dcaf-rs`.



Code city for JabRef.

Summary

Research Question 1

How can LSP be integrated into SEE to generate code cities?

☞ Using the algorithm explained in previous slides

Research Question 2

What is the scalability of this integration?

☞ Feasible until ≈ 200 kLOC (but very dependent on configuration and Language Server)

Research Question 3

Are code cities a suitable means to present LSP information to developers as compared to IDEs + tables (on the dimensions of speed, accuracy, and usability)?

☞ While code cities seem suitable to present LSP information, developers work faster and prefer traditional IDEs (at least in the case of SEE vs. VSCode).

All SUS Questions

1. “I think that I would like to use the system frequently.”
2. “I found the system unnecessarily complex.”
3. “I thought the system was easy to use.”
4. “I think that I would need the support of a technical person to be able to use the system.”
5. “I found the various functions in the system were well integrated.”
6. “I thought there was too much inconsistency in the system.”
7. “I would imagine that most people would learn to use the system very quickly.”
8. “I found the system very cumbersome to use.”
9. “I felt very confident using the system.”
10. “I needed to learn a lot of things before I could get going with the system.”

Matching Locations to Nodes

- Need to match returned LSP locations (definition, references, …) to nodes
- Cannot use LUT, since locations are not necessarily the same

☞ Match location to node with “tightest fitting range”
  - That is, smallest node that completely contains given location

- Naïve approach: iterate over all nodes to find best fit
  - $\Theta(|V|)$ per added edge, so in total, $\Theta(|V| \cdot |E|)$

Augmented Interval Trees

- Idea: Augment k-d tree (2 dimensions: line + column) to **interval tree**
- Node in such a tree modeled as $T = (v, l, c, \lambda, \rho)$
  - $v \in V$: node in project graph
    - Starting position of node acts as key
  - $l, c \in \mathbb{N}_0$: maximum line number and character offset in subtree
  - $\lambda, \rho \in \mathcal{T}$: left and right subtree
- Construct such an interval tree for each document

Augmented Interval Trees: Example

Example: Find `someValue`: $(1, 13, 1, 22)$

```
0  |public class Example {
1    |const char someValue = 'A'|;
2
3    |public static long pow(|int num|) {
4      |long result = num * num|;
5      return result;
6    }|
7  }|
```