Incremental clone detection for IDEs using dynamic suffix arrays

Jakob Konrad Hansen

University of Oslo

2023

Outline

Motivation

- Duplicated code is generally considered harmful to software quality
- Code clone detection, analysis and management is therefore important
- Incremental clone detection algorithms have not been thoroughly researched
- Incremental algorithms are useful in use-cases such as in IDEs

Our contribution

- CCDetect-LSP: An incremental clone detection tool for IDEs
- Uses a novel application of dynamic extended suffix arrays for clone detection
- Language- and IDE agnostic via Tree-sitter and LSP

Code clones

Definition (Code snippet)

A code snippet is a piece of contiguous source code in a larger software system.

Definition (Code clone)

A code clone is a code snippet which is equal or similar to another code snippet. The two code snippets are both code clones, and together they form a clone pair. Similarity is determined by some metric such as number of equal lines of code.

Clone types

- Code clones are classified into four types
  - Type-1: Syntactically identical
  - Type-2: Structurally identical
  - Type-3: Structurally similar
  - Type-4: Functionally similar (generally)

Clone type examples: type-1 and type-2

```
for (int i = 0; i < 10; i++) {
    print(i);
}
```

```
for (int i = 0; i < 10; i++) {
    // A comment

    print(i);
}
```

Figure: Type-1 clone pair

```
for (int i = 0; i < 10; i++) {
    print(i);
}
```

```
for (int j = 5; j < 20; j++) {
    print(j);
}
```

Figure: Type-2 clone pair

Clone type examples: type-3 and type-4

```
for (int i = 0; i < 10; i++) {
    print(i);
}
```

```
for (int i = 0; i < 10; i++) {
    print(i);
    print(i*2);
}
```

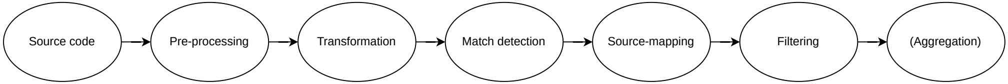
Figure: Type-3 clone pair

```
print((n*(n-1))/2)
```

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        sum++;
    }
}
print(sum);
```

Figure: Type-4 clone pair

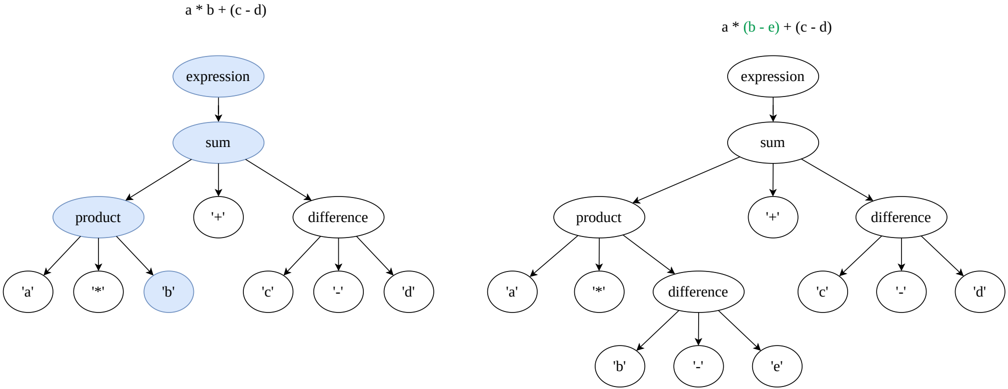
Clone detection



Clone matching techniques

- Text-based detection
  - Match based on raw source code
- Token-based detection
  - Match based on tokens
- Syntactic detection
  - Match based on AST
- Hybrid detection
  - Combine multiple approaches

Parsing and incremental parsing



Suffix tree

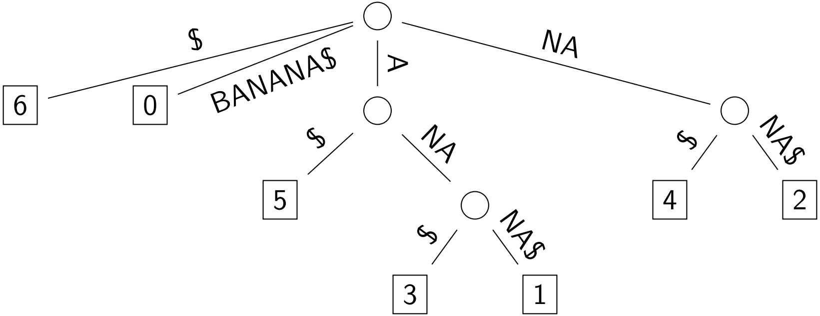


Figure: Suffix tree for $S = \text{BANANA\$}$

Suffix array

Index	Suffix
0	BANANA$
1	ANANA$
2	NANA$
3	ANA$
4	NA$
5	A$
6	$

(a) Suffixes

Index	Suffix
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

(b) Sorted suffixes

Index	SA	ISA	LCP
0	6	4	0
1	5	3	0
2	3	6	1
3	1	2	3
4	0	5	0
5	4	1	0
6	2	0	2

(c) SA, ISA and LCP

Burrows-Wheeler transform

Index	CS
0	BANANA$
1	ANANA$B
2	NANA$BA
3	ANA$BAN
4	NA$BANA
5	A$BANAN
6	$BANANA

(d) Cyclic shifts

Index	CS
6	$BANAN**A**
5	A$BANA**N**
3	ANA$BA**N**
1	ANANA$**B**
0	BANANA**$**
4	NA$BAN**A**
2	NANA$B**A**

(e) Sorted cyclic shifts and BWT

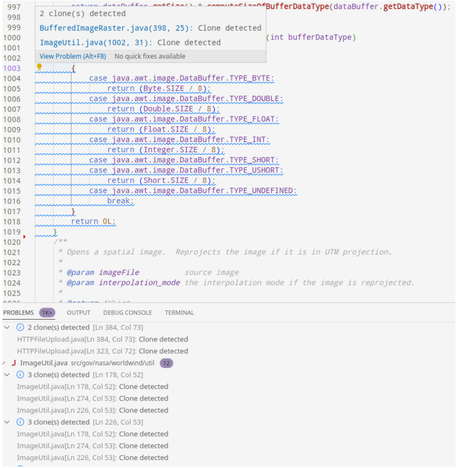
L	F
0	$Rank_A(0) + C[A] = 0 + 1 = 1$
1	$Rank_N(1) + C[N] = 0 + 5 = 5$
2	$Rank_N(2) + C[N] = 1 + 5 = 6$
3	$Rank_B(3) + C[B] = 0 + 4 = 4$
4	$Rank_\$(4) + C[\$] = 0 + 0 = 0$
5	$Rank_A(5) + C[A] = 1 + 1 = 2$
6	$Rank_A(6) + C[A] = 2 + 1 = 3$

(f) LF function

Table: S = BANANA$, BWT = ANNB$AA

CCDetect-LSP features

- CCDetect-LSP is implemented as an LSP server
  - List clones
  - Display clones inline with code
  - Jump between matching clones
  - Incremental updates on each edit



Implementation: Initial clone detection

- Algorithm which initially detects type-1 and optionally type-2 clones
- Pipeline of 5 phases, returns a list of clones
- Uses an extended suffix array for match detection
- Starting point: Assume documents are indexed

Detection algorithm overview

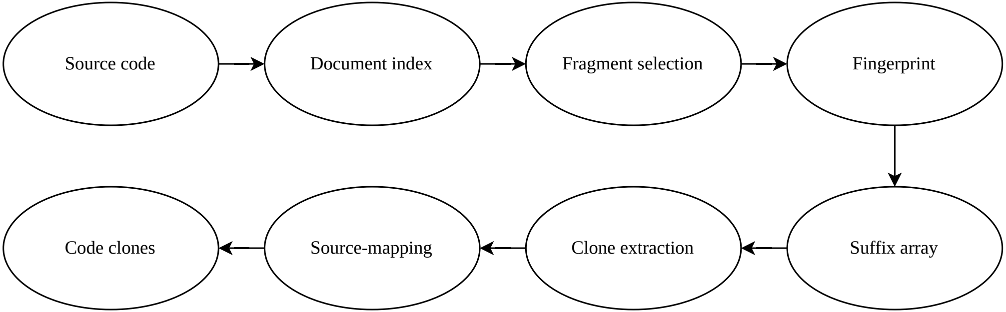


Figure: Overview of detection algorithm phases

Phase 1: Fragment selection

- Parse files using Tree-sitter
- Use a configurable Tree-sitter query to “capture” nodes
- Extract and store the tokens of captured nodes

```
(method_declaration) @method (constructor_declaration) @constructor
```

Phase 2: Fingerprinting

- Consistently hash each token value with an increasing integer counter
- Store the fingerprint of each fragment in the document index
- For type-2 detection, hash the token type instead

Phase 2: Fingerprinting

```
public class Math() {
    public int multiplyByTwo(int param) {
        return param * 2;
    }

    public int addTwo(int param) {
        return param + 2;
    }
}
```

Token	Fingerprint
public	2
int	3
multiplyByTwo	4
(	5
param	6
)	7
{	8
return	9
*	10
2	11
;	12
}	13
addTwo	14
+	15

[2, 3, 4, 5, 3, 6, 7, 8, 9, 6, 10, 11, <u>1</u>, 2, 3, 14, 5, 3, 6, 7, 8, 9, 6, 15, 11, <u>1</u>, <u>0</u>]

Figure: Example fingerprint of Java source-code

Phase 3: Suffix array construction

- Concatenate the fingerprints of each document in the index
- Construct SA, ISA and LCP array of the full fingerprint
- Uses "Induced sorting variable-length LMS-substrings" (SA-IS) algorithm
- LCP algorithm slightly modified

F:[2, 3, 4, 5, 3, 6, 7, 8, 9, 6, 10, 11, 1, 2, 3, 14, 5, 3, 6, 7, 8, 9, 6, 15, 11, 1, 0]
SA:[26,25,12,0,13,1,4,17,14,2,3,16,5,18,9,22,6,19,7,20,8,21,10,24,11,15,23]
ISA:[3,5,9,10,6,12,16,18,20,14,22,24,2,4,8,25,11,7,13,17,19,21,15,26,23,1,0]
LCP: [0, 0, 0, 0, 2, 0, 1, 6, 1, 0, 0, 7, 0, 5, 1, 1, 0, 4, 0, 3, 0, 2, 0, 0, 1, 0, 0]

Phase 4: Clone extraction

- Use SA, ISA and LCP to find clone positions
- Linear scan through the fingerprint
- Skip contained clones

Phase 5: Source-mapping

- Map the positions of clones back to the original source-code
- Find the correct file and position of an index in the fingerprint

Implementation: Incremental clone detection

- Convert the algorithm to an incremental one
- Input is now the file which has changed and potentially the range
- Dynamic suffix array with edit operations as input

Phase 1: Update document index and fragment selection

- Store the AST of the opened files
- If range available, incrementally parse the changed file
- Mark changed files
- Fragment selection unchanged

Phase 2: Fingerprinting

- Each document stores its fingerprint
- Only need to fingerprint (and fragment select) changed files

Phase 2.5: Edit operations

- Input to phase 3: Edit operations
- How to determine edit operations?
- Edit distance algorithm!
- "Batched" operations preferred

		D	E	M	O	C	R	A	T
	0	1	2	3	4	5	6	7	8
R	1	1	2	3	4	5	5	6	7
E	2	2	1	2	3	4	5	6	7
P	3	3	2	2	3	4	5	6	7
U	4	4	3	3	3	4	5	6	7
B	5	5	4	4	4	4	5	6	7
L	6	6	5	5	5	5	5	6	7
I	7	7	6	6	6	6	6	6	7
C	8	8	7	7	7	6	7	7	7
A	9	9	8	8	8	7	7	7	8
N	10	10	9	9	9	8	8	8	8

Table: REPUBLICAN → DEMOCRAT

Optimize edit distance

- Standard algorithm memory usage is too high
- Need to optimize
  - Compare new/old fingerprint of changed document only
  - Remove trivial part at each end of matrix
  - Hirschberg's algorithm

		F	I	N	I	S	H	I	N	G
	0	1	2	3	4	5	6	7	8	9
F	1	0	1	2	3	4	5	6	7	8
A	2	1	1	2	3	4	5	6	7	8
S	3	2	2	2	3	3	4	5	6	7
C	4	3	3	3	3	4	4	5	6	7
I	5	4	3	4	3	4	5	4	5	6
N	6	5	4	3	4	4	5	5	4	5
A	7	6	5	4	4	5	5	6	5	5
T	8	7	6	5	5	5	6	6	6	6
I	9	8	7	6	5	6	6	6	7	7
N	10	9	8	7	6	6	7	7	6	7
G	11	10	9	8	7	7	7	8	7	6

Table: FASCINATING → FINISHING
ASCINAT → INISH

Phase 3: Dynamic suffix array

- Update suffix array based on edit operations
- "Four-stage algorithm for updating a Burrows-Wheeler transform"
- Updates to the BWT correlates with updates to the SA and ISA
- Inserting a single character leads to:
  - A new character in the BWT
  - A changed character in the BWT
  - 0 or more reordering of characters

Phase 3: Dynamic suffix array

Order	F	L
6	$	A
5	A	N
3	A	N
1	A	B
0	B	$
4	N	A
2	N	A

(a) Original BWT

Order	F	L	
7	$	A	
6	A	N	
4	A	N	
1	A	B	
0	B	$	
2	**B**	**A**	Inserted
5	B	A	
3	**N**	**B**	$A \rightarrow B$

(b) After change and insert

Order	F	L	
7	$	A	
6	A	N	
1	**A**	**B**	$\updownarrow$
4	**A**	**N**	
0	B	$	
2	B	A	
5	B	A	
3	N	B	

(c) After reordering

Table: Updating BWT dynamically for the string BANANA$ → BABNANA$

Dynamic extended suffix array

- Updating suffix array is slow
  - Inserting: $O(n)$
  - Incrementing: $O(n)$
- We therefore change the data structure which stores SA, ISA and LCP

Dynamic extended suffix array

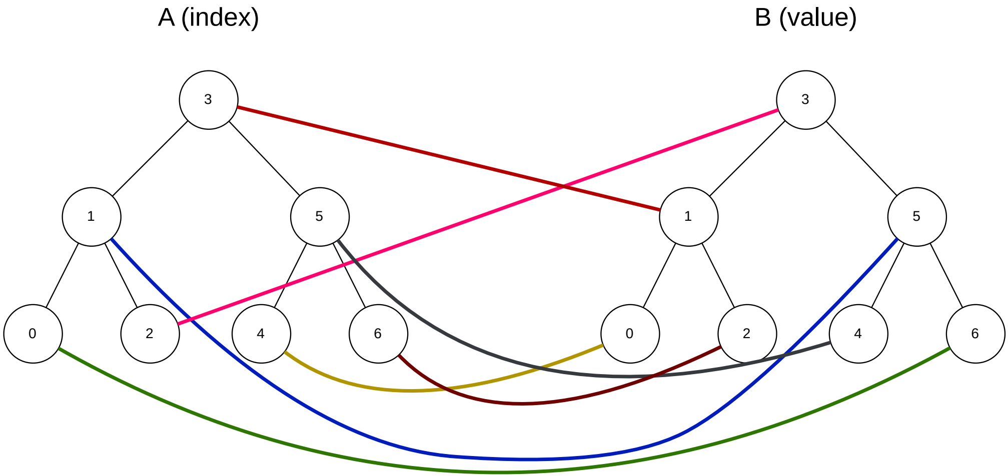


Figure: Dynamic permutation for the permutation $[6, 5, 3, 1, 0, 4, 2]$.

Updating LCP values

- SA updates correlate with LCP values which need to be updated

			$\leftrightarrow$			INS		
BWT	A	N	B	N	\$	A	A	B
SA	7	6	4	1	0	2	5	3
Old LCP	0	0	$\underline{1}$	$\underline{3}$	$\underline{0}$	$\underline{\text{N}}$	$\underline{0}$	2
New LCP	0	0	1	1	0	1	0	2

Phase 4 and 5: Clone extraction and source-mapping

- Very similar to the initial detection
- Accessing SA, ISA and LCP is now a bit slower, but this is optimized

Evaluation

- CCDetect-LSP evaluation:
  - Verify correctness with BigCloneEval
  - Benchmark performance
  - Benchmark memory usage
  - Tested multiple languages and IDEs

BigCloneBench

- A large database of clones in a Java dataset
- BigCloneEval can evaluate detection tools on BigCloneBench

```
-- Recall Per Clone Type (type: numDetected / numClones = recall) --
                 Type-1: 23209 / 23210 = 0.9999569151227920
                 Type-2: 3542 / 3547 = 0.9985903580490555
         Type-2 (blind): 242 / 245 = 0.9877551020408163
    Type-2 (consistent): 3300 / 3302 = 0.9993943064809206
```

Figure: BigCloneEval evaluation report for CCDetect-LSP

Performance evaluation

- CCDetect-LSP was evaluated on multiple codebases
- Performance compared against SACA detection and iClones
- Incremental updates were randomly generated
- 10×10 or 10×100 tokens inserted/deleted

Code base	**LOC**	**Clones detected**	$\mathbf{LCP_{avg}}$	$\mathbf{LCP_{\geq 100}}$
WorldWind	550KLOC	1517	18	63967
neo4j	1MLOC	1313	9	27557
graal	2.2MLOC	2012	28	154452
flink	2.3MLOC	4729	13	155754
elasticsearch	3.2MLOC	9986	14	289511
intellij-community	5.8MLOC	3585	19	336190

Table: Properties of code bases

WorldWind (550KLOC)

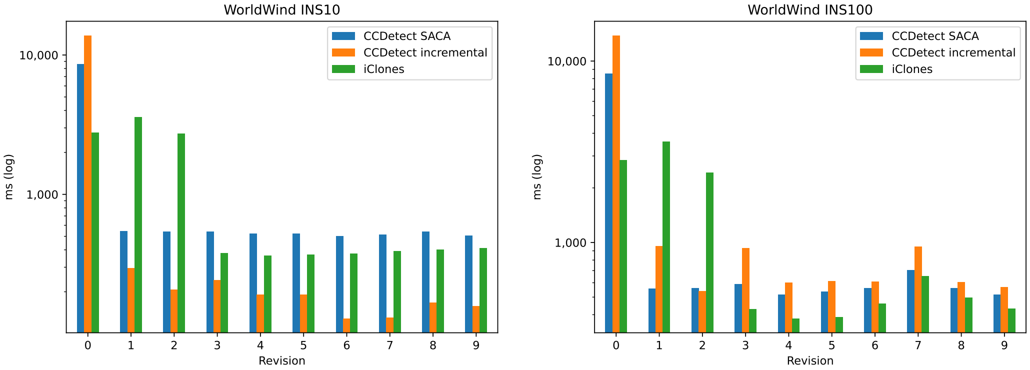


Figure: WorldWind performance benchmark

neo4j (1MLOC)

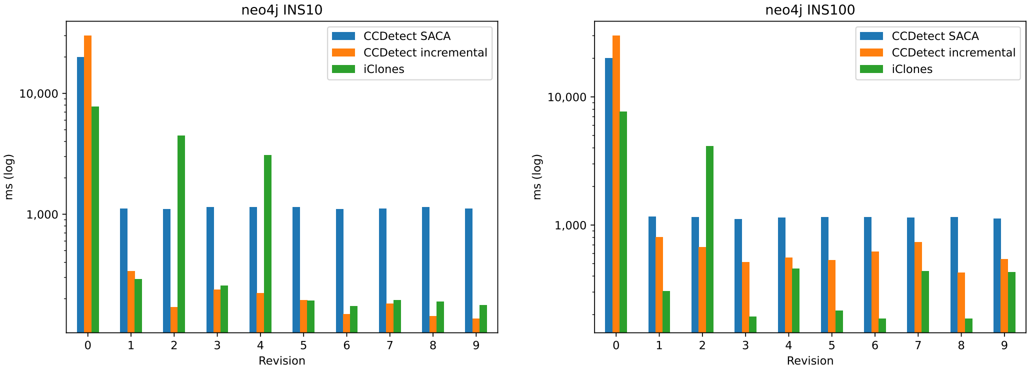


Figure: neo4j performance benchmark

graal (2.2MLOC)

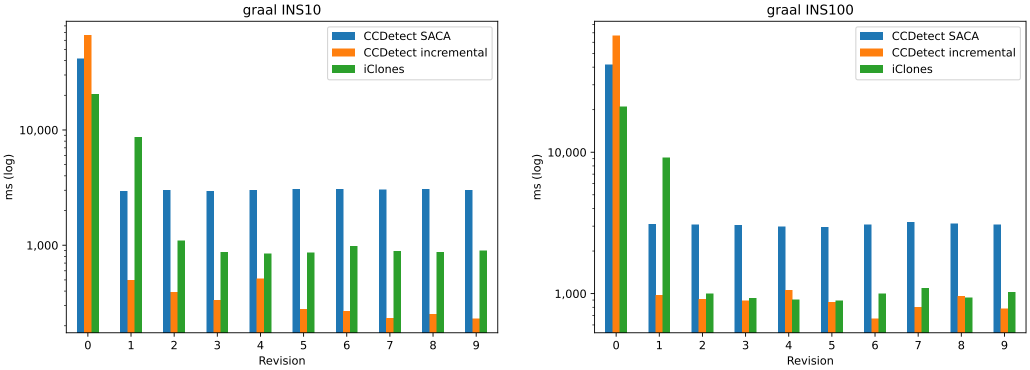


Figure: graal performance benchmark

flink (2.3MLOC)

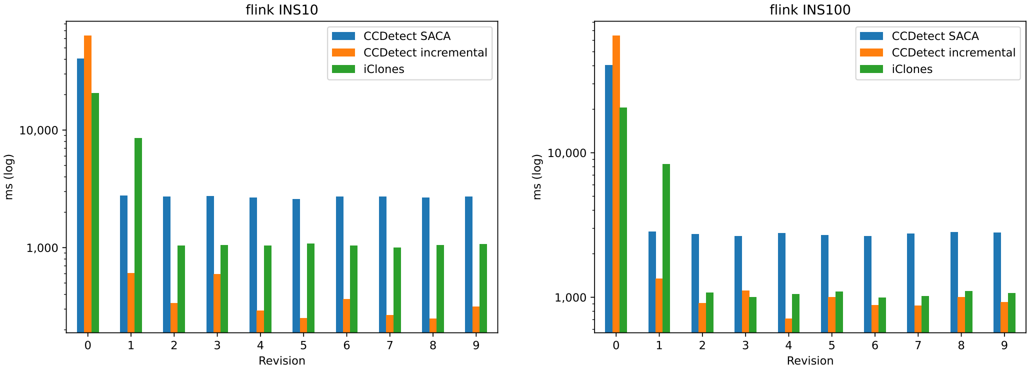


Figure: flink performance benchmark

elasticsearch (3.2MLOC)

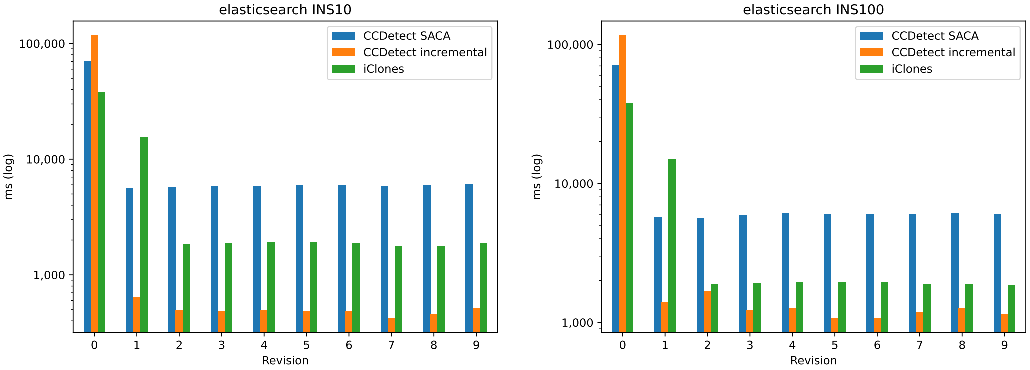


Figure: elasticsearch performance benchmark

intellij-community (5.8MLOC)

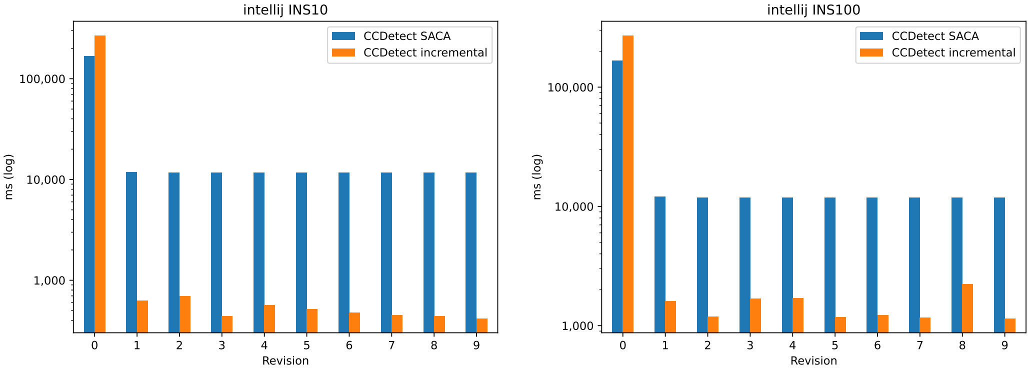
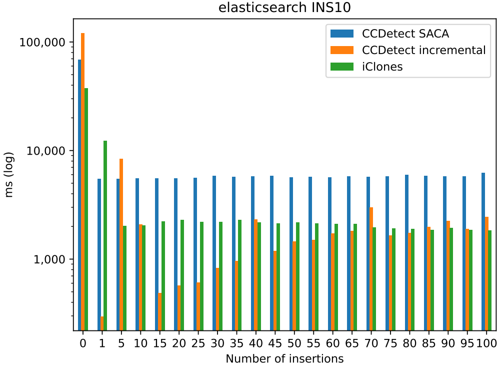


Figure: intellij-community performance benchmark

elasticsearch (3.2MLOC)



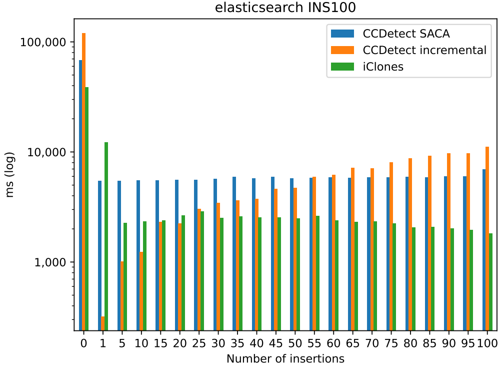


Figure: Elasticsearch performance benchmark with increasing number of edits

Memory usage

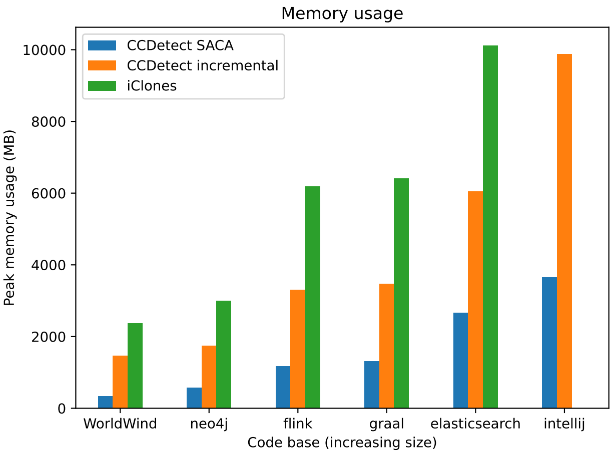


Figure: Peak memory usage of each tool when running the DEL10 test for each code base.

Language and IDE agnostic

- CCDetect tested on 6 languages
  - Java
  - Python
  - C
  - Rust
  - Javascript
  - Go
- And 2 IDEs
  - Neovim
  - VSCode
- Works well in our experience

Discussion

- Performance
  - Incremental detection has best performance if edits are small
- Memory usage
  - SACA detection best, incremental detection better than iClones
  - Memory usage is probably a bottle-neck for practical usage
- Language agnostic
  - Works well if grammar is correct
- IDE agnostic
  - Functionality should work in most IDEs
  - Setup and configuration depends on IDE

Conclusion

- CCDetect-LSP is a performant incremental clone detection tool
- Language- and IDE agnostic clone detection is achieved
- Practical usage?
  - Clone information used by other tools
  - Live manual refactoring of code clones
- Future work?
  - Type-3 clones
  - Refactoring clones
  - Compressing data structures
  - Optimal edit operations

Questions?

- Questions?
  - Demo?
  - LSP communication and architecture?
  - Anything else?

LSP

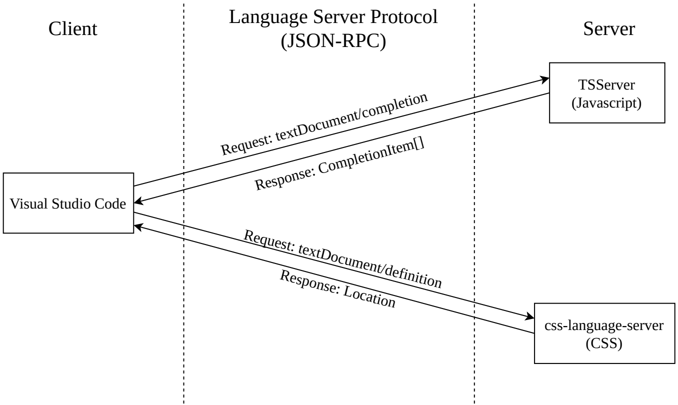


Figure: Example LSP server communication

LSP architecture

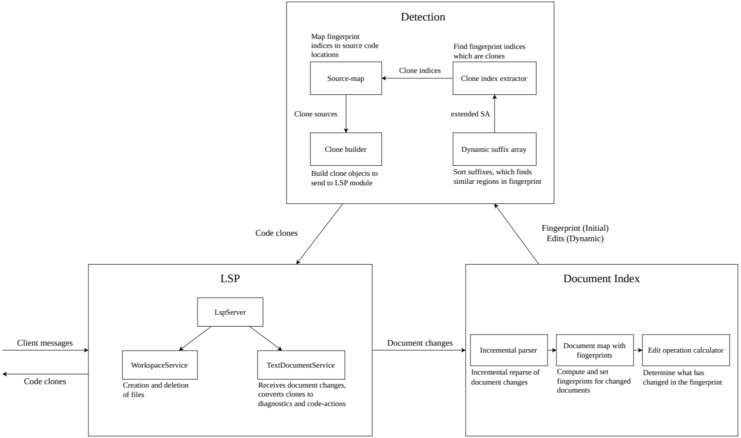


Figure: Architecture of CCDetect-LSP