

Incremental clone detection for IDEs using dynamic suffix arrays

Jakob Konrad Hansen

University of Oslo

2023

Outline

- 1 Motivation and contribution
- 2 Background
 - Code clone theory
 - Preliminary algorithms and data structures
- 3 Implementation
 - Features
 - Initial clone detection
 - Incremental clone detection
- 4 Evaluation
- 5 Discussion
- 6 Conclusion
- 7 Questions or demo?

Motivation

- Duplicated code is generally considered harmful to software quality
- Software often contains 10 – 15% duplicated code
- 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.

Clone types: 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);  
}
```

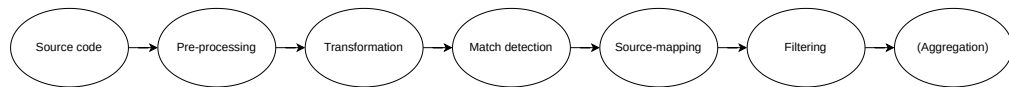
Figure: Type-1 clone pair

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

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

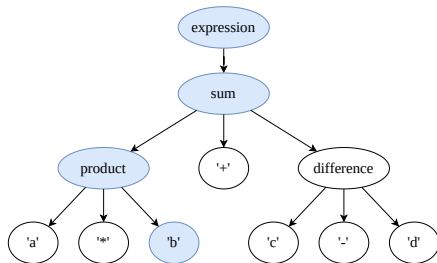
Figure: Type-2 clone pair

Clone detection

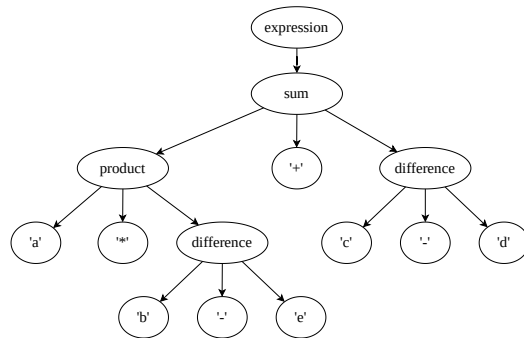


Parsing and incremental parsing

$a * b + (c - d)$



$a * (b - e) + (c - d)$



Suffix array

Index	Suffix
0	BANANAS\$
1	ANANAS\$
2	NANAS\$
3	ANAS\$
4	NAS\$
5	AS\$
6	\$

(a) Suffixes

Index	Suffix
6	\$
5	AS\$
3	ANAS\$
1	ANANAS\$
0	BANANAS\$
4	NAS\$
2	NANAS\$

(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	Index	CS	L	F
0	BANANA\$	6	\$BANANA	0	$Rank_A(0) + C[A] = 0 + 1 = 1$
1	ANANA\$B	5	A\$BANAN	1	$Rank_N(1) + C[N] = 0 + 5 = 5$
2	NANA\$BA	3	ANA\$BAN	2	$Rank_N(2) + C[N] = 1 + 5 = 6$
3	ANA\$BAN	1	ANANA\$B	3	$Rank_B(3) + C[B] = 0 + 4 = 4$
4	NA\$BANA	0	BANANA\$	4	$Rank_{\$}(4) + C[\$] = 0 + 0 = 0$
5	A\$BANAN	4	NA\$BANA	5	$Rank_A(5) + C[A] = 1 + 1 = 2$
6	\$BANANA	2	NANA\$BA	6	$Rank_A(6) + C[A] = 2 + 1 = 3$

(d) Cyclic shifts

(e) Sorted cyclic shifts
and BWT

(f) LF function

Table: $S = \text{BANANA\$}$, $\text{BWT} = \text{ANNB\$AA}$

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

The screenshot shows a Java IDE with a code editor and a problem list. The code editor displays a snippet from `BufferedImage` with annotations indicating detected clones. The problem list at the bottom shows the detected clones as errors.

```

997 2 clone(s) detected
998 BufferedImageRaster.java(398, 25): Clone detected
1000 ImageUtil.java(1002, 31): Clone detected
1002 View Problem (Alt+F8) No quick fixes available
1003 {
1004     case java.awt.image.DataBuffer.TYPE_BYTE:
1005         return (Byte.SIZE / 8);
1006     case java.awt.image.DataBuffer.TYPE_DOUBLE:
1007         return (Double.SIZE / 8);
1008     case java.awt.image.DataBuffer.TYPE_FLOAT:
1009         return (Float.SIZE / 8);
1010     case java.awt.image.DataBuffer.TYPE_INT:
1011         return (Integer.SIZE / 8);
1012     case java.awt.image.DataBuffer.TYPE_SHORT:
1013     case java.awt.image.DataBuffer.TYPE_USHORT:
1014         return (Short.SIZE / 8);
1015     case java.awt.image.DataBuffer.TYPE_UNDEFINED:
1016         break;
1017 }
1018 return 0L;
1019 /**
1020  * Opens a spatial image. Reprojects the image if it is in UTM projection.
1021  *
1022  * @param imageFile source image
1023  * @param interpolation_mode the interpolation mode if the image is reprojected.
1024  *
1025  */

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

- 2 clone(s) detected [Ln 384, Col 73]
 - HTTPFileUpload.java(Ln 384, Col 73): Clone detected
 - HTTPFileUpload.java(Ln 323, Col 72): Clone detected
- ImageUtil.java src/gov/inasa/worldwind/util 12
 - 3 clone(s) detected [Ln 178, Col 52]
 - ImageUtil.java(Ln 178, Col 52): Clone detected
 - ImageUtil.java(Ln 274, Col 53): Clone detected
 - ImageUtil.java(Ln 226, Col 53): Clone detected
 - 3 clone(s) detected [Ln 226, Col 53]
 - ImageUtil.java(Ln 178, Col 52): Clone detected
 - ImageUtil.java(Ln 274, Col 53): Clone detected
 - ImageUtil.java(Ln 226, Col 53): Clone detected

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

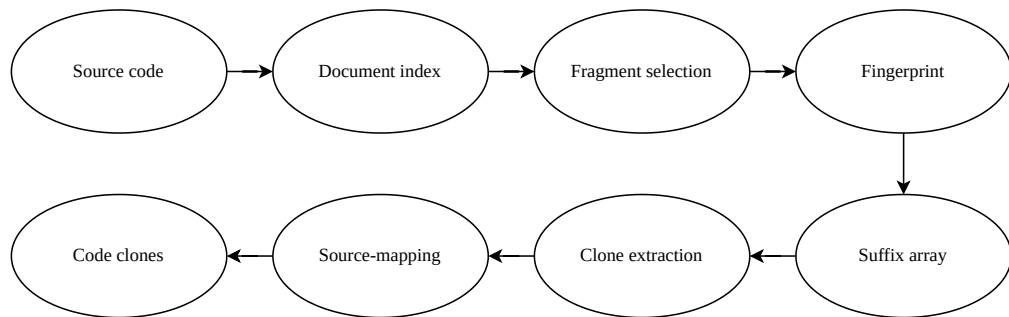


Figure: Overview of detection algorithm phases

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
- For type-2 detection, hash the token type instead

```
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, 1, 2, 3, 14, 5, 3, 6, 7, 8, 9, 6, 15, 11, 1, 0]

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

Phase 4: Clone extraction

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

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 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
- Binary-search speeds this process up

Implementation: Incremental clone detection

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

Phase 1 and 2: Fragment selection and fingerprinting

- Fragment selection
 - Store the AST of the opened files
 - If range available, incrementally parse the changed file
 - Fragment selection unchanged
- 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
5	B	A
3	N	B

Inserted
 $A \rightarrow B$

(b) After change and insert

Order	F	L
7	\$	A
6	A	N
1	A	B
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\$ \rightarrow BABNANA\$

Dynamic extended suffix array

Dynamic extended suffix array

- Updating suffix array is slow ($O(n)$), new data structure needed

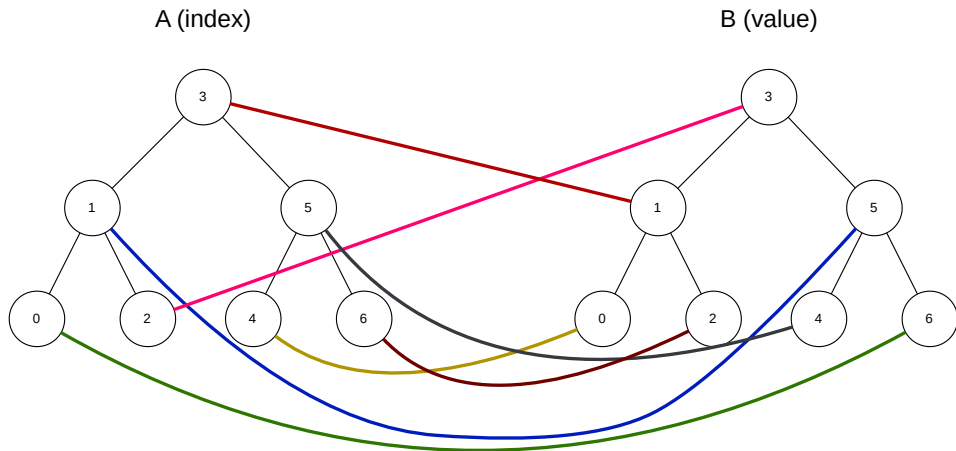


Figure: Dynamic permutation for the permutation $[6, 5, 3, 1, 0, 4, 2]$.

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	<u>1</u>	<u>3</u>	<u>0</u>	<u>N</u>	<u>0</u>	2
New LCP	0	0	1	1	0	1	0	2

Phase 4 and 5: Clone extraction and source-mapping

- Similar to the initial detection
- Store nodes with LCP values \geq threshold
- 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 = 0.999956915122792  
      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	LCP_{avg}	$LCP_{\geq 100}$
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)

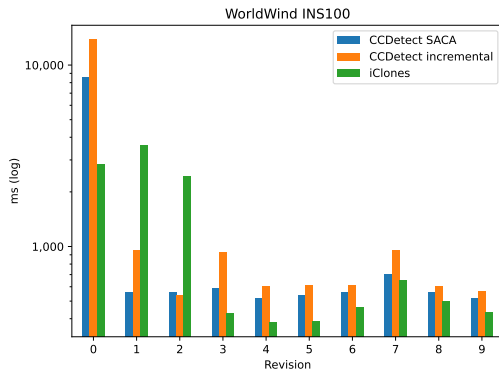
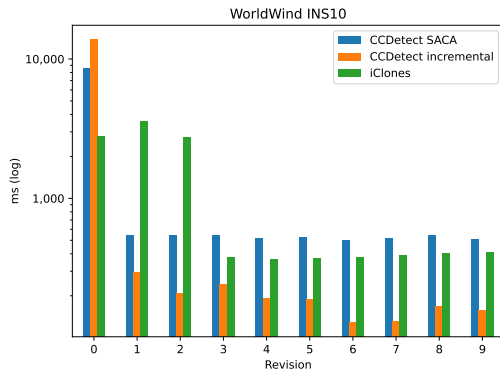


Figure: WorldWind performance benchmark

neo4j (1MLOC)

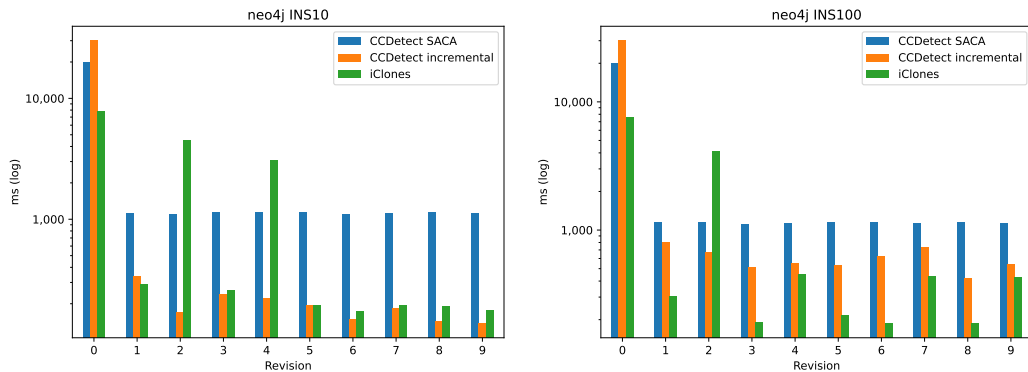


Figure: neo4j performance benchmark

graal (2.2MLOC)

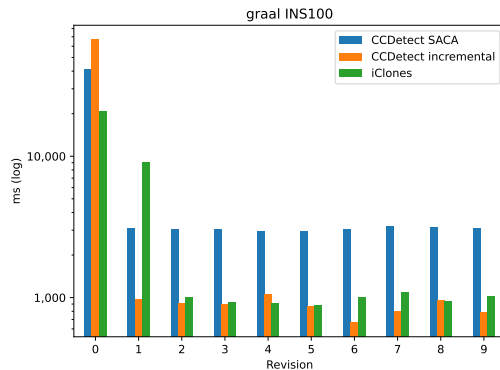
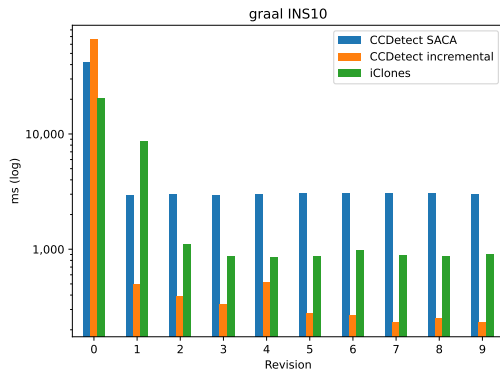


Figure: graal performance benchmark

flink (2.3MLOC)

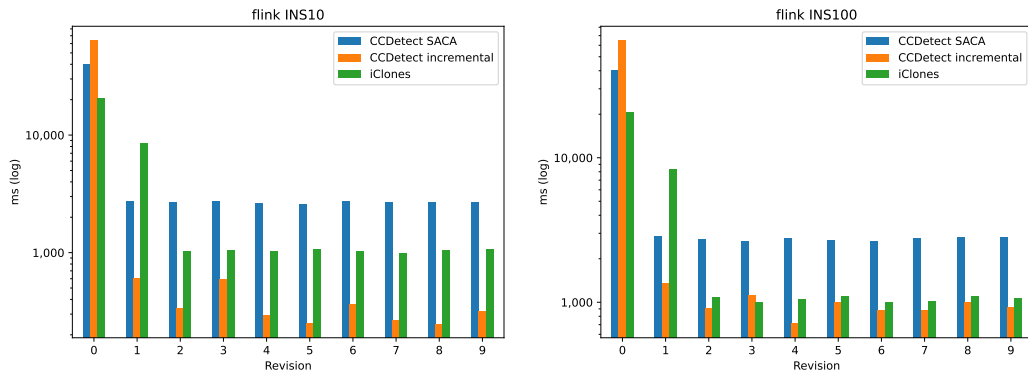


Figure: flink performance benchmark

elasticsearch (3.2MLOC)

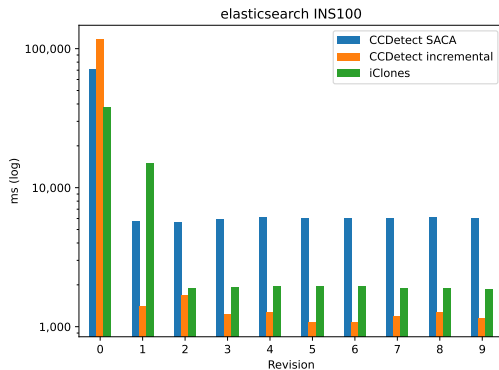
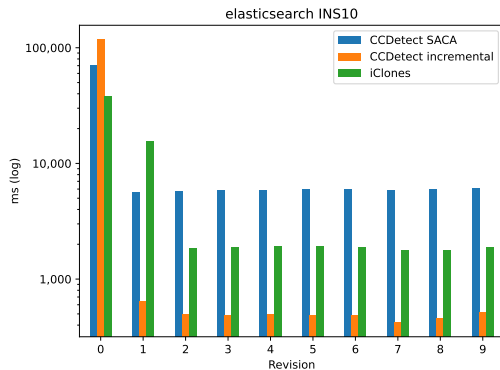


Figure: elasticsearch performance benchmark

intellij-community (5.8MLOC)

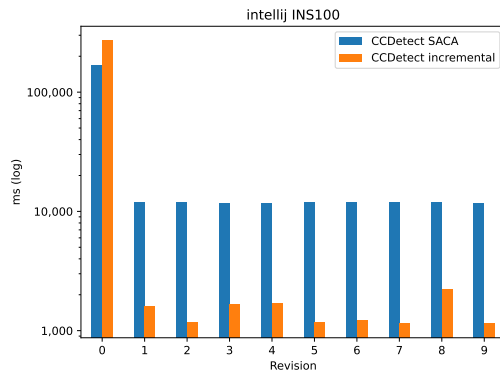
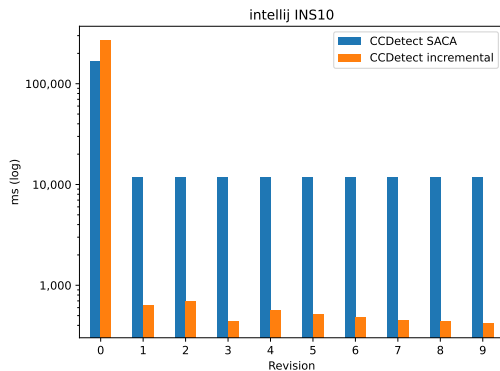


Figure: intellij-community performance benchmark

elasticsearch (3.2MLOC)

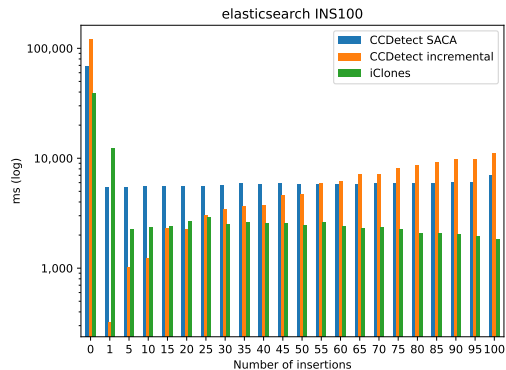
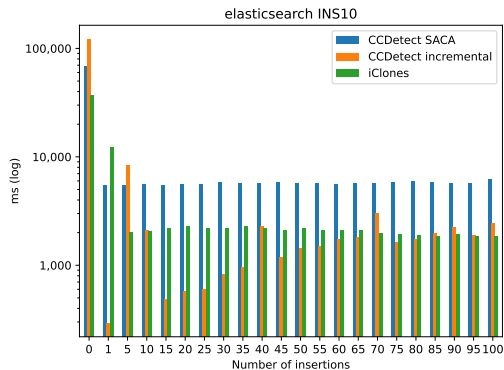


Figure: Elasticsearch performance benchmark with increasing number of edits

Memory usage

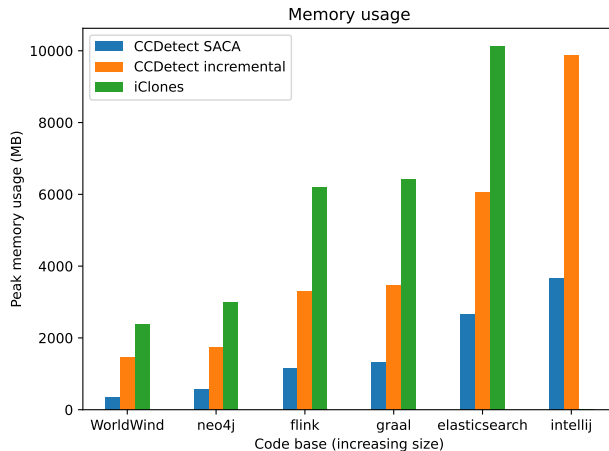


Figure: Peak memory usage of each tool when running the DEL10 test for each code base.

Language and IDE agnostic

- CCDetect-LSP 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?
 - Live manual/automatic refactoring of code clones
 - Clone information used by other tools/refactorings
 - Simultaneous editing
- Future work?
 - Type-3 clones
 - Refactoring clones
 - Compressing data structures
 - Optimal edit operations

Questions?

- Questions?
 - Demo?
 - Go through anything in more detail?
 - LSP communication and architecture?
 - Anything else?

LSP

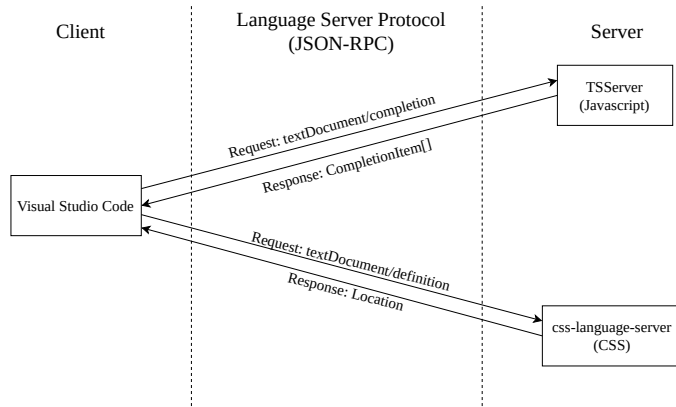


Figure: Example LSP server communication

LSP architecture

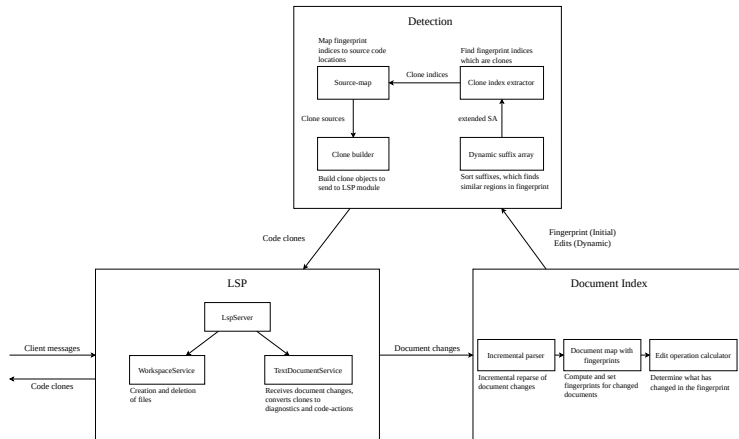


Figure: Architecture of CCDetect-LSP

Clone types: 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