# 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
- LSP architecture and functionality + demo
- Initial clone detection
- Incremental clone detection

4 Evaluation

5 Discussion

6 Conclusion

# 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);
```

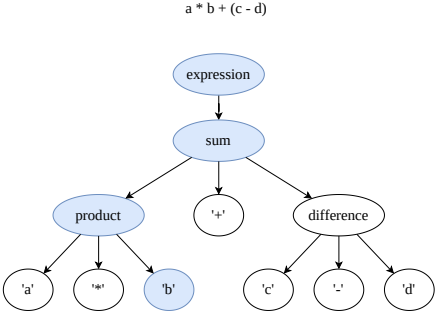
Figure: Type-4 clone pair

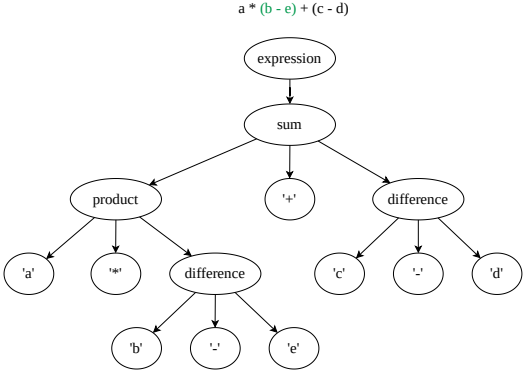
# Clone detection

# 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



# Suffixes

## Definition

A suffix of a string $S$ is any nonempty substring which reaches the end of $S$. The suffix at position $i$ of $S$ is denoted $\text{Suffix}(S, i)$.

# Suffix tree

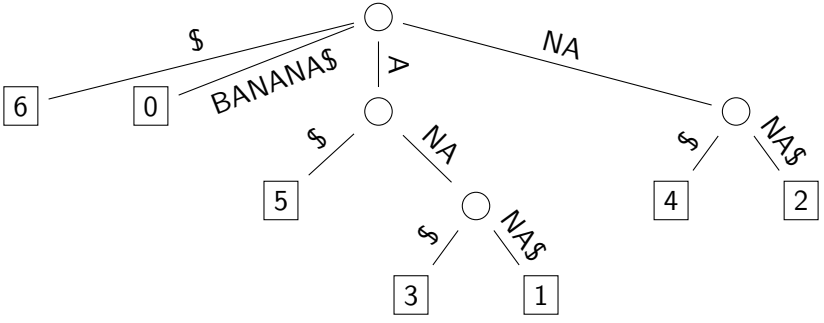



Figure: Suffix tree for $S = \text{BANANA\$}$

# Suffix array

**Definition (Suffix array)**

Let $S$ be a string of length N. The suffix array SA of $S$ is an array of length N where SA$[i] = n$ if Suffix$(S, n)$ is the ith lexicographically smallest suffix in $S$.

**Definition (LCP array)**

Let S be a string of length N and SA be the suffix array of S. The LCP array of S is an array of length N where LCP[i] = n if Suffix$(S, SA[i])$ and Suffix$(S, SA[i-1])$ has a maximal common prefix of length n. LCP[0] is undefined or 0.

# 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

# Implementation: LSP architecture and functionality

- The Language Server Protocol (LSP) facilitates IDE agnostic tooling
- CCDetect-LSP is implemented as an LSP server
  - List clones
  - Display clones inline with code
  - Jump between matching clones
  - Incremental updates on each edit

# LSP

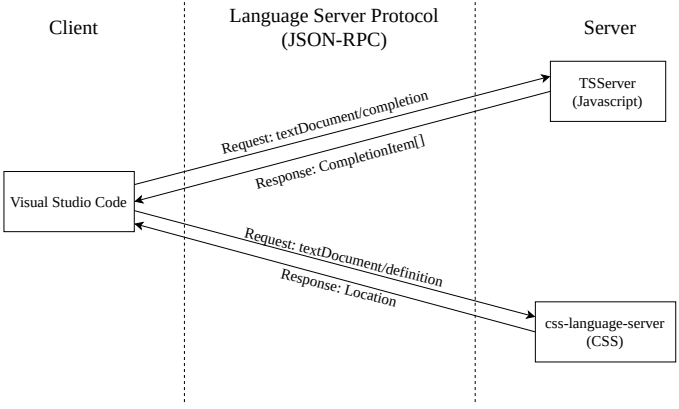



Figure: Example LSP server communication

# CCDetect-LSP architecture

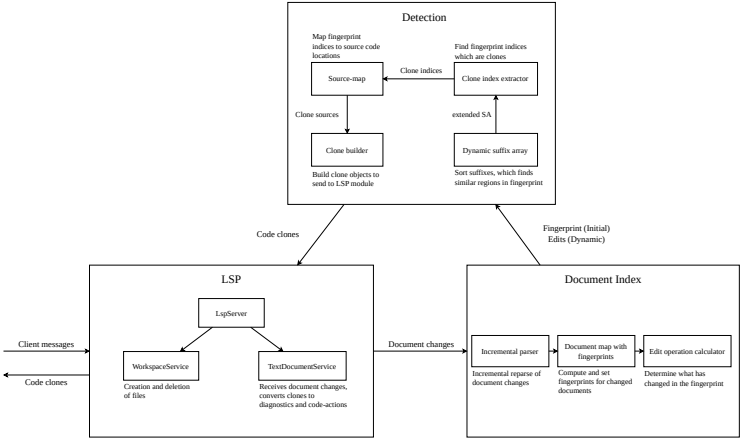



Figure: Architecture of CCDetect-LSP

# 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

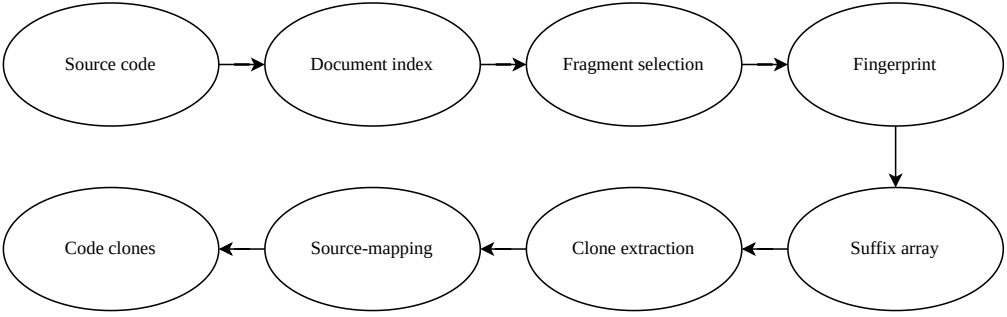



Figure: Overview of detection algorithm phases

# 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

# 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

- Example here

# 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

# Phase 4: Clone extraction

# Phase 5: Source-mapping

# Implementation: Incremental clone detection

# Results

# Discussion

# Conclusion