Deckard: Scalable and Accurate Tree-based Detection of Code Clones

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, Stephane Glondu
The Problem

- Find “similar” code in large code bases, often referred to as **Clone Detection**.

- Many applications in software engineering
Clone Detection Techniques

- **String-based**

- **Token-based**

- **Tree-based**

- **Semantic-based**
  - E.g., Komondooor & Horwitz (2001)
Limitations of Existing techniques

- String-based, token-based
  - Sensitive to code format
  - E.g., $x = a*b+c$ vs. $z = c+a*b$

- Tree-based, semantic-based
  - Not scalable
Deckard’s Goal

- More robust against code modification
- Scalable to million-line programs
Tree Similarity and Code Clones

- Well defined tree similarity problem
  - Tree editing distance $\delta(T_1, T_2)$:
    - $T_1$ and $T_2$ are $\sigma$-similar if $\delta(T_1, T_2) < \sigma$

- Code clones based on $\delta$

Clone pair, if $\delta(T_1, T_2) < \sigma$
Not Scalable

A clone detection directly based on the definition would not scale:

- Computing tree editing distance is expensive

$$\text{Minimal \# of edit operations} \quad O (|T_1| \times |T_2| \times d_1 \times d_2) \quad (\text{Zhang et al. 1989})$$

where $d_i$ is the minimum of the depth of $T_i$ and the number of leaves of $T_i$

- Need to compare many pairs of subtrees
The Idea — Numerical Vectors

- Numerical vectors are much easier to compare than trees
  - Suppose $v_1 = <x_1, \ldots, x_n>$ and $v_2 = <y_1, \ldots, y_n>$
  - Hamming distance (or the $l_1$ norm): $H(v_1, v_2) = \sum_{i=1}^{n} |x_i - y_i|$  
  - Euclidean distance (or the $l_2$ norm): $D(v_1, v_2) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$

- Characterize parse trees as numerical vectors
  - Cluster vectors based on numerical distances
  - Code corresponding to vectors within a same cluster are clones
Deckard’s High Level Architecture
Outline

Define characteristic vectors for trees → Relate $H, D$ with $\delta$ → Generate vectors

Language Description → Parse-tree builder generator → Source Repository → Parse-tree builder → Parse-tree → Vector Generator

Clone Report → Post Processor → Clones → Vector Clustering → Vector Database

Evaluate Deckard → Implement Deckard → Cluster vectors with Locality-Sensitive Hashing
Characteristic Vectors for Trees

Definitions

- **q-level atomic patterns**
  - Labeled complete binary trees of height q
  - E.g., 2-level
    
    ![Diagram of 2-level tree]
    
    - At most $|L|^{2^q-1}$ patterns, where $L$ is the set of labels

- **q-level vector** for a tree or tree-forest $T$
  - $<b_1, \ldots, b_k>$, where $k \leq |L|^{2^q-1}$
  - Each $b_i$ counts # of $i$-th pattern in $T$
An Sample Characteristic Vector

```c
for ( int i = 0; i < n; i++ )
x[i] = 0;
```

1-level patterns:

```
<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```
An Sample Characteristic Vector

\[
\text{for ( int } i = 0; i < n; i++ ) \]
\[
x[i] = 0;
\]

1-level patterns:
\[
<\text{id, lit, assign}_e, \text{incr}_e, \text{array}_e, \text{cond}_e, \text{expr}_s, \text{decl}, \text{for}_s>
\]
Relate $H(T_1, T_2)$, $D(v_1, v_2)$, and $\delta(v_1, v_2)$

- **Theorem 1 (Yang et al. 2005):**
  If $\delta(T_1, T_2) = k$, then $H(v_1, v_2) \leq (4q - 3)k$

- **Theorem 2:** for integer vectors,
  $\sqrt{H(v_1, v_2)} \leq D(v_1, v_2) \leq H(v_1, v_2)$

- **Theorem 3:** for tree characteristic vectors,
  $\frac{\sqrt{H(v_1, v_2)}}{4q - 3} \leq \frac{D(v_1, v_2)}{4q - 3} \leq \frac{H(v_1, v_2)}{4q - 3} \leq k$
Map Trees to Numerical Vectors (1)

```java
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**

![Subtree diagram]
Map Trees to Numerical Vectors (1)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**

![Diagram of a tree with numerical vectors assigned to nodes]
Map Trees to Numerical Vectors (1)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;
```

- `<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>`

- **Subtree vectors:**

![Diagram showing a for loop and its subtree vectors]
Map Trees to Numerical Vectors (1)

```plaintext
for ( int i = 0; i < n; i++ )
x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**

![Diagram showing a for loop with subtree vectors assigned numerical values.](image)
Map Trees to Numerical Vectors (1)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**
Map Trees to Numerical Vectors (1)

```c
for (int i = 0; i < n; i++)
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**

```
for (decl) 1,1,0,0,0,0,1,0
; cond_e 2,0,0,0,0,1,0,0
; incr_e 2,0,0,0,0,1,0,0
for (decl) 2,1,1,0,1,0,1,0
int id = primary_e
; primary_e
; primary_e
; primary_e
++ primary_e
expr_s 2,1,1,0,1,0,0,0
assign_e 2,1,1,0,1,0,0,0
; primary_e
array_e 2,0,0,0,1,0,0,0
=} primary_e

[ primary_e ]

primary_e [ ]

primary_e [ ]
```

20
Map Trees to Numerical Vectors (1)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**

```plaintext
for ( int i = 0; i < n; i++ )
    x[i] = 0;
```

```plaintext
<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```
Map Trees to Numerical Vectors (1)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- **Subtree vectors:**

  ![Tree Diagram]
Map Trees to Numerical Vectors (2)

- Forest vectors
  - For sequences of program elements
  - Why needed?
    - S1; S2; S3; S4;
    - S0; S5; S2; S3; S6
Map Trees to Numerical Vectors (2)

```plaintext
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

- Forest vectors:
Map Trees to Numerical Vectors (2)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;
```

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>

- Forest vectors:
Map Trees to Numerical Vectors (2)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;
```

Forest vectors:

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>

Diagram of forest vectors with numerical vector values.
Map Trees to Numerical Vectors (2)

```c
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>
```

Forest vectors:
for ( int i = 0; i < n; i++ )
    x[i] = 0;

<id, lit, assign_e, incr_e, array_e, cond_e, expr_s, decl, for_s>

- All vectors:
Vector Clustering

- Have millions of generated vectors
- Need to efficiently cluster “close” vectors together
  - Hashing

- Locality-sensitive hashing (LSH)
  - A family of hash functions, s.t., with high probability,
    - Similar vectors hashed to a same hash value
    - Distant vectors hashed to different hash values
LSH-based Clone Detection (1)

- Rely on LSH to construct hash tables for vectors.

\[ v_1, v_i, \ldots, v_n \]

\[ h(v_1), h(v_i), \ldots, h(v_n) \]
LSH-based Clone Detection (2)

- Query LSH for vector clusters

![Diagram showing LSH and vector clusters]
**LSH-based Clone Detection (2)**

- Query LSH for vector clusters

![Diagram showing LSH-based clone detection](image)

- Close neighbors of \( v_i \):
  - \( v_1, v_k, v_{j'}, v_{j'}, \ldots \)
  - \( v_n, v_{m}, \ldots, v_y \)

- LSH output:
  - \( h(v_1), h(\ldots), h(v_n) \)
LSH-based Clone Detection (2)

- Query LSH for vector clusters

- Post-process to reduce “false” clones
  - Small clones
  - Overlapping/duplicate clones
Implementation

- Modified bison for constructing parse trees
- 1-level characteristic vectors for trees
- Group vectors according to their sizes
  - Able to handle size-sensitive clones
  - Less stress on LSH
  - Easy to parallelize
Evaluation Results

- Comparison with two state-of-the-art tools
  - Tree-based: CloneDR (for Java)
    - An evaluation version of a commercial tool
  - Token-based: CP-Miner (for C)

- Evaluation metrics
  - Clone quantity (% of cloned code)
  - Clone quality (% of false clones)
  - Scalability
## Test Programs

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th># files</th>
<th># lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux kernel 2.6.16</td>
<td>C</td>
<td>7,988</td>
<td>5,287,090</td>
</tr>
<tr>
<td>JDK 1.4.2</td>
<td>Java</td>
<td>8,534</td>
<td>2,418,767</td>
</tr>
<tr>
<td>GCC 3.3.6</td>
<td>C</td>
<td>7,828</td>
<td>1,246,322</td>
</tr>
<tr>
<td>PostgreSQL 8.1.0</td>
<td>C</td>
<td>725</td>
<td>606,015</td>
</tr>
<tr>
<td>Derby 10.0.2.1</td>
<td>Java</td>
<td>1,418</td>
<td>523,939</td>
</tr>
<tr>
<td>Apache 2.2.0</td>
<td>C</td>
<td>537</td>
<td>269,287</td>
</tr>
</tbody>
</table>
# Test Programs

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th># files</th>
<th># lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux kernel 2.6.16</td>
<td>C</td>
<td>7,988</td>
<td>5,287,090</td>
</tr>
<tr>
<td>JDK 1.4.2</td>
<td>Java</td>
<td>8,534</td>
<td>2,418,767</td>
</tr>
<tr>
<td>GCC 3.3.6</td>
<td>C</td>
<td>7,828</td>
<td>1,246,322</td>
</tr>
<tr>
<td>PostgreSQL 8.1.0</td>
<td>C</td>
<td>725</td>
<td>606,015</td>
</tr>
<tr>
<td>Derby 10.0.2.1</td>
<td>Java</td>
<td>1,418</td>
<td>523,939</td>
</tr>
<tr>
<td>Apache 2.2.0</td>
<td>C</td>
<td>537</td>
<td>269,287</td>
</tr>
</tbody>
</table>
Detected More Clones (1)

JDK 1.4.2 (Deckard vs. CloneDR)

Cloned LoC (#/10^5)

- minimum clone size: 50 tokens, stride: 4
- CloneDR w/ optimal parameters

Similarity

1.0 0.9999 0.999 0.99 0.95 0.9
Detected More Clones (2)

Linux kernel 2.6.16 (Deckard vs. CP-Miner)

Gap: # of statement insertions, deletions, and modifications

Graph showing the relation between similarity and LOC in clones.
Clone Quality (1)

- Examined 100 clone clusters
  - Randomly selected from JDK 1.4.2
  - Used parameters
    - Similarity: 1.0
    - Minimum clone size in tokens: 50
    - Stride: 4

- Among the 100 clusters
  - 93 are clearly real clones
  - The remaining 7 are difficult to assess
    - But they are structurally similar
Clone Quality (2)

```java
if ......
....... else if (option.equalsIgnoreCase("basic")) {
    bBasicTraceOn = true;
} else if (option.equalsIgnoreCase("net")) {
    bNetTraceOn = true;
} else if (option.equalsIgnoreCase("security")) {
    bSecurityTraceOn = true;
} else ......
.......

if ......
....... else if (opt.equals("-nohelp")) {
    nohelp = true;
} else if (opt.equals("-splitindex")) {
    splitindex = true;
} else if (opt.equals("-noindex")) {
    createindex = false;
} else ......
.......`
Scalability (1)

JDK 1.4.2 (Deckard vs. CloneDR)

- Running Time (min)
- Similarity

- minimum clone size: 50 tokens; stride 4
- CloneDR w/ optimal parameters
Scalability (2)

Linux kernel 2.6.16 (Deckard vs. CP-Miner)

Gap: # of statement insertions, deletions, and modifications

![Graph showing running time vs. similarity for Linux kernel 2.6.16 with Deckard and CP-Miner. The graph includes data points for minimum clone size: 50 tokens; stride 4, CP-Miner min size: 50 tokens; Gap 0, and CP-Miner min size: 50 tokens; Gap 1.]
Conclusion

- New algorithm for tree similarity checking
  - Map trees to numerical vectors
  - Cluster similar vectors efficiently

- Application of the algorithm on clone detection
  - Similarity checking on parse trees
  - Promising results: scalable and accurate
Thank you!

Questions?
jiangl@cs.ucdavis.edu