Software Similarity Analysis

© April 28, 2011 Christian Collberg
Duplicates are the result of **copy-paste-modify** programming.
Clone detection

- Duplicates are the result of **copy-paste-modify** programming.
- Problem during maintenance — all copies of bugs need to be fixed.
Detection phase: locating similar pieces of code in a program.
Clone detection

- Detection phase: locating similar pieces of code in a program.
- Abstraction phase: clones are extracted out into functions.
Clone detection algorithm — Finding Clones

**Detect**\((P, \text{threshold}, \text{minsize})\):  

1. Build a representation \(rep\) of \(P\) from which it is convenient to find clone pairs. Collect code pairs that are sufficiently similar and sufficiently large to warrant their own abstraction:

\[
\begin{align*}
res &\leftarrow \emptyset \\
rep &\leftarrow \text{convenient representation of } P \\
&\text{for every pair of code segments } f, g \in rep, f \neq g \text{ do} \\
&\quad \text{if similarity}(f, g) > \text{threshold} \land \\
&\quad \quad \text{size}(f) \geq \text{minsize} \land \text{size}(g) \geq \text{minsize} \text{ then} \\
&\quad res \leftarrow res \cup \langle f, g \rangle
\end{align*}
\]
Clone detection algorithm — Replace Clones

**DETECT**\((P, \text{threshold, minsize})\):

2. Break out the code-pairs found in the previous step into their own function and replace them with parameterized calls to this function:

   for every pair of code segments \(f, g \in res\) do
   
   \(h(r) \leftarrow\) a parameterized version of \(f\) and \(g\)
   
   \(P \leftarrow P \cup h(r)\)
   
   replace \(f\) with a call to \(h(r_1)\) and \(g\) with \(h(r_2)\)

3. Return \(res, P\)
Attack model

- We don’t expect programmers to be malicious!
We don’t expect programmers to be malicious!

The code becomes naturally “obfuscated” because of the specialization process.
We don’t expect programmers to be malicious!

The code becomes naturally “obfuscated” because of the specialization process.

The programmer renames variables and replace literals with new values in the copied code.
We don’t expect programmers to be malicious!
The code becomes naturally “obfuscated” because of the specialization process.
The programmer renames variables and replace literals with new values in the copied code.
More complex changes are unusual.
What has this to do with software protection?

- Skype binary was protected by adding several hundred hash functions.
What has this to do with software protection?

- Skype binary was protected by adding several hundred hash functions.
- Could a clone detector have found them?
Hand in a verbatim copy of a friend's program.
Plagiarism of programming assignments

- Hand in a verbatim copy of a friend’s program.
- Or, make radical changes to the program to hide the origin of the code.
Plagiarism of programming assignments

- Hand in a verbatim copy of a friend’s program.
- Or, make radical changes to the program to hide the origin of the code.
- “Borrowing” one or more difficult functions from a friend.
Plagiarism of programming assignments

- Hand in a verbatim copy of a friend’s program.
- Or, make radical changes to the program to hide the origin of the code.
- “Borrowing” one or more difficult functions from a friend.
- Fishing code out of the trash can.
Plagiarism of programming assignments

- Hand in a verbatim copy of a friend’s program.
- Or, make radical changes to the program to hide the origin of the code.
- "Borrowing" one or more difficult functions from a friend.
- Fishing code out of the trash can.
- Nabbing code off the printer.
Plagiarism of programming assignments

- Hand in a verbatim copy of a friend’s program.
- Or, make radical changes to the program to hide the origin of the code.
- “Borrowing” one or more difficult functions from a friend.
- Fishing code out of the trash can.
- Nabbing code off the printer.
- Outsource the assignments to an unscrupulous third party (“programming-mills”).
Plagiarism detection

- Make pair-wise comparisons between all the programs handed in by the students:

\[
\langle P_1, P_2 \rangle = 70\% \\
\langle P_1, P_3 \rangle = 20\% \\
\langle P_2, P_3 \rangle = 10\%
\]
Detect \((U, threshold)\):

\[
\begin{align*}
res & \leftarrow \emptyset \\
\text{for each pair of programs } f,g & \text{ do} \\
\quad sim & \leftarrow \text{similarity}(f,g) \\
\quad \text{if } sim > threshold & \text{ then} \\
\quad \quad res & \leftarrow res \cup \langle f, g, sim \rangle \\
res & \leftarrow res \text{ sorted on similarity} \\
\text{return } res
\end{align*}
\]
The student needs the code to look “reasonable.”
The student needs the code to look “reasonable.”
General-purpose obfuscation — probably not a good idea.
The student needs the code to look “reasonable.”
General-purpose obfuscation — probably not a good idea.
Renaming `windowSize` to `sizeOfWindow` — OK.
The student needs the code to look “reasonable.”
General-purpose obfuscation — probably not a good idea.
Renaming `windowSize` to `sizeOfWindow` — OK.
Renaming `windowSize` to `x93` — not OK.
• The student needs the code to look “reasonable.”
• General-purpose obfuscation — probably not a good idea.
• Renaming `windowSize` to `sizeOfWindow` — OK.
• Renaming `windowSize` to `x93` — not OK.
• Replace a while-loop with a for-loop — OK.
The student needs the code to look “reasonable.”

General-purpose obfuscation — probably not a good idea.

Renaming `windowSize` to `sizeOfWindow` — OK.

Renaming `windowSize` to `x93` — not OK.

Replace a while-loop with a for-loop — OK.

Unroll the for-loop — not OK.
Algorithm ssEFM

p. 631

AST-based clone detection

```
var a
var b
var c
int 9
```

```
int 5
```

```
int 7
```
Look for clones in this program:

\[(5 + (a + b)) \times (7 + (c + 9))\]

Parse and build an AST $S$:
An inefficient clone detector... 

- Construct all **tree patterns**.
An inefficient clone detector... 

- Construct all tree patterns.
- A tree pattern is a subtree of $S$ where one or more subtrees have been replaced with a wildcard.
An inefficient clone detector.

- Construct all **tree patterns**.

- A tree pattern is a subtree of $S$ where one or more subtrees have been replaced with a wildcard.
An inefficient clone detector... 

- Construct all **tree patterns**.
- A tree pattern is a subtree of $S$ where one or more subtrees have been replaced with a wildcard.
- We’ll color the ASTs themselves blue and the tree patterns pink.
Some of the tree patterns
What’s a clone in an AST?

What’s a clone in the context of an AST?
What’s a clone in an AST?

- What’s a clone in the context of an AST?
- Simply a tree pattern for which there’s more than one match!
What’s a clone in an AST?

- Simply a tree pattern for which there’s more than one match!
- Which patterns would make a good clone?
  1. has a large number of nodes
  2. occurs a large number of times in the AST
  3. has few holes
Which patterns would make good clones?

- This pattern seems like it might make a good choice
Which patterns would make good clones?

- This pattern seems like it might make a good choice

- It matches two large subtrees of $S$: 
Extract clones!

Now you can extract the clones and turn them into macros:

```c
#define CLONE(x,y,z) (((x)+((y)+(z))))
CLONE(5,a,b) * CLONE(7,c,9)
```
A slow algorithm...

- Build a *clone table*, a mapping from each pattern to the locations in $S$ where it occurs:
A slow algorithm...

- Build a *clone table*, a mapping from each pattern to the locations in $S$ where it occurs:
- Sort the table with largest patterns, most number of occurrences, fewest number of holes first!
A heuristic algorithm…

- Won’t scale: exponential number of tree patterns.
A heuristic algorithm...

- Won’t scale: exponential number of tree patterns.
- Idea: iteratively grow larger tree patterns from smaller ones.
A heuristic algorithm. . .

- Won’t scale: exponential number of tree patterns.
- Idea: iteratively grow larger tree patterns from smaller ones.
- Step 1:
We specialize, and the new pattern becomes larger (but only has 2 matches):
We specialize, and the new pattern becomes larger (but only has 2 matches):

After two more steps of specialization, we’re done:
They found this clone 10 times over some Java classes:

```java
for(int i=0; i<?1; i++)
    if (?2[i] != ?3[i])
        return false;
```

The strength of the algorithm is that it allows structural matching: holes can accept any subtree.
Graph-based analysis
p. 635
Programs are graphs!

- Control-flow graphs!
Programs are graphs!

- Control-flow graphs!
- Dependence graphs!
Programs are graphs!

- Control-flow graphs!
- Dependence graphs!
- Inheritance graphs!
Programs are graphs!

- Control-flow graphs!
- Dependence graphs!
- Inheritance graphs!
- Can program similarity be computed over graph representations of programs?
Unfortunately...

- Sub-graph isomorphism is NP-complete.
Unfortunately...

- Sub-graph isomorphism is NP-complete.
- Fortunately, graphs computed from programs are not general graphs.
Unfortunately…

- Sub-graph isomorphism is NP-complete.
- Fortunately, graphs computed from programs are not general graphs.
- Control-flow graphs will not be arbitrarily large.
Unfortunately...

- Sub-graph isomorphism is NP-complete.
- Fortunately, graphs computed from programs are not general graphs.
- Control-flow graphs will not be arbitrarily large.
- Call-graphs tend to be very sparse.
Unfortunately...

- Sub-graph isomorphism is NP-complete.
- Fortunately, graphs computed from programs are not general graphs.
- Control-flow graphs will not be arbitrarily large.
- Call-graphs tend to be very sparse.
- Heuristics can be very effective in approximating subgraph isomorphism.
Algorithm $SSKH$

PDG-based clone detection
The nodes of a PDF are the statements of a function.
The nodes of a PDF are the statements of a function. There’s an edge $m \rightarrow n$ if
1. $n$ is data-dependent on $m$, or
2. $n$ is control-dependent on $m$. 
The nodes of a PDF are the statements of a function.

There's an edge $m \rightarrow n$ if

1. $n$ is data-dependent on $m$, or
2. $n$ is control-dependent on $m$.

Semantics-preserving reordering of the statements of a function won’t affect the graph.
$S_0$: int $k = 0$;
$S_1$: int $s = 1$;
$S_2$: while ($k < w$) {
  $S_3$: if ($x[k] == 1$)
    $S_4$: $R = (s*y) \% n$;
    else
    $S_5$: $R = s$;
  $S_6$: $s = R*R \% n$;
  $S_7$: $L = R$;
  $S_8$: $k = k + 1$;
}
Program Dependence Graph
Build a PDG for each function of the program
ssKH: basic idea

- Build a PDG for each function of the program
- Compute two isomorphic subgraphs by slicing backwards along dependency edges starting with every pair of *matching nodes*. 
Build a PDG for each function of the program
Compute two isomorphic subgraphs by slicing backwards along dependency edges starting with every pair of matching nodes.
Two nodes are matching if they have the same syntactic structure.
ssKH: basic idea

- Build a PDG for each function of the program
- Compute two isomorphic subgraphs by slicing backwards along dependency edges starting with every pair of *matching nodes*.
- Two nodes are matching if they have the same syntactic structure.
- Repeat until no more nodes can be added to the slice.
A (contrived) example

```
a_1:  a = g(8);
b_1:  b = z*3;
a_2:  while (a<10)
     a_3:  a = f(a);
b_2:  while (b<20)
     b_3:  b = f(b);
a_4:  if (a==10) {
     a_5:     printf("foo\n");
     a_6:     x=x+2;
    }
b_4:  if (b==20) {
     b_5:     printf("bar\n");
     b_6:     y=y+2;
     b_7:     printf("baz\n");
    }
```

- Two similar pieces of code have been intertwined within the same function.
A (contrived) example

```plaintext
a1: a = g(8)
a2: while(a<10)
a3: a = f(a)
a4: if(a==10)
a5: printf("foo\n")
a6: x = x + 2

b1: b = z * 3
b2: while(b<20)
b3: b = f(b)
b4: if(b==20)
b5: printf("bar\n")
b6: y = y + 2
b7: printf("baz\n")
```
Algorithm: Step 1-3.

- $a_4$ and $b_4$ match. Add them to the slice.
Algorithm: Step 1-3.

- $a_4$ and $b_4$ match. Add them to the slice.
- Consider $a_4$ and $b_4$'s predecessors, $a_3$ and $b_3$. 
Algorithm: Step 1-3.

- $a_4$ and $b_4$ match. Add them to the slice.
- Consider $a_4$ and $b_4$'s predecessors, $a_3$ and $b_3$.
- $a_3$ and $b_3$ match, too. Add them to the slice.
Algorithm: Step 1-3.

- $a_4$ and $b_4$ match. Add them to the slice.
- Consider $a_4$ and $b_4$’s predecessors, $a_3$ and $b_3$.
- $a_3$ and $b_3$ match, too. Add them to the slice.
- Add $a_2$ and $b_2$ to the slice since they match and are predecessors of $a_3$ and $b_3$. 
a₁: a = g(8)

a₂: while (a < 10)

a₃: a = f(a)

a₄: if (a == 10)

a₅: printf("foo\n")

a₆: x = x + 2

b₁: b = z * 3

b₂: while (b < 20)

b₃: b = f(b)

b₄: if (b == 20)

b₅: printf("bar\n")

b₆: y = y + 2

b₇: printf("baz\n")
Algorithm: Step 4

- $a_5/b_5$ and $a_6/b_6$ really should belong to the clone!
Algorithm: Step 4

- $a_5/b_5$ and $a_6/b_6$ really should belong to the clone!
- But, backwards slice won’t include them.
- $a_5/b_5$ and $a_6/b_6$ really should belong to the clone!
- But, backwards slice won’t include them.
- So, slice forward one step from any predicate in an if- and while-statement.
The PDG after Step 4

```plaintext
a1: a = g(8)

a2: while (a < 10)
    a3: a = f(a)
    a4: if (a == 10)
        a5: printf("foo\n")
    a6: x = x + 2

b1: b = z * 3

b2: while (b < 20)
    b3: b = f(b)
    b4: if (b == 20)
        b5: printf("bar\n")
    b6: y = y + 2
    b7: printf("baz\n")
```

b = z * 3

```
37 / 49
```
```c
#define CLONE(x, c, d, s, p, y)\
    while (x < c) x = f(x);\
    if (x == d) {\
        printf(s);\
        y = y + 2;\
        p = 1;}
    else p = 0;

a = g(8);
b = z * 3;
CLONE(a, 10, 10, "foo\n", p, x)
CLONE(b, 20, 20, "bar\n", p, y)
if (p) printf("baz\n");
```
This algorithm handles

- clones where statements have been reordered,
- clones that are non-contiguous,
- and clones that have been intertwined with each other.
Summary

- This algorithm handles
  - clones where statements have been reordered,
  - clones that are non-contiguous,
  - and clones that have been intertwined with each other.

- Depressing performance numbers. A 11,540 line C program takes 1 hour and 34 minutes to process.
Algorithm \texttt{SSLCHY}  

p. 640

PDG-based plagiarism detection
Uses PDGs, but for plagiarism detection.
Uses PDGs, but for plagiarism detection.
Uses a general-purpose subgraph isomorphism algorithm.
Uses PDGs, but for plagiarism detection.

Uses a general-purpose subgraph isomorphism algorithm.

Uses a preprocessing step to weed out unlikely plagiarism candidates.
What does it mean for one PDG to be considered a plagiarised version of another?
Plagiarised PDGs?

- What does it mean for one PDG to be considered a plagiarised version of another?
- We expect some manner of obfuscation of the code — equality is too strong!
Plagiarised PDGs?

- What does it mean for one PDG to be considered a plagiarised version of another?
- We expect some manner of obfuscation of the code — equality is too strong!
- The two PDGs should be $\gamma$-isomorphic.
Plagiarised PDGs?

What does it mean for one PDG to be considered a plagiarised version of another?

We expect some manner of obfuscation of the code — equality is too strong!

The two PDGs should be $\gamma$-isomorphic.

Set $\gamma = 0.9$, ("overhauling (without errors) 10% of a PDG of reasonable size is almost equivalent to rewriting the code."
Common Subgraphs

Definition

Common subgraphs Let $G$, $G_1$, and $G_2$ be graphs. $G$ is a common subgraph of $G_1$ and $G_2$ if there exists subgraph isomorphisms from $G$ to $G_1$ and from $G$ to $G_2$.

$G$ is the maximal common subgraph of two graphs $G_1$ and $G_2$ ($G = \text{mcs}(G_1, G_2)$) if $G$ is a common subgraph of $G_1$ and $G_2$ and there exists no other common subgraph $G'$ of $G_1$ and $G_2$ that has more nodes than $G$. 
The colored nodes induce a maximal common subgraph of $G_1$ and $G_2$ of four nodes:
Definition

Graph similarity and containment. Let $|G|$ be the number of nodes in $G$. The similarity $(G_1, G_2)$ of $G_1$ and $G_2$ is defined as

$$similarity(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{\max(|G_1|, |G_2|)}$$

The containment $(G_1, G_2)$ of $G_1$ within $G_2$ is defined as

$$containment(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{|G_1|}.$$

We say that $G_1$ is $\gamma$-isomorphic to $G_2$ if

$$containment(G_1, G_2) \geq \gamma, \gamma \in (0, 1].$$
Graph similarity and containment — Example

\[ \text{similarity}(G_1, G_2) = \frac{4}{7} \] and
Graph similarity and containment — Example

\[ \text{similarity}(G_1, G_2) = \frac{4}{7} \] and
\[ \text{containment}(G_1, G_2) = \frac{4}{6}. \]
Subgraph isomorphism testing is expensive — prune out $\frac{9}{10}$ of all program pairs from consideration:
- Ignore any graph which has too few nodes to be interesting.
Subgraph isomorphism testing is expensive — prune out $\frac{9}{10}$ of all program pairs from consideration:

1. ignore any graph which has too few nodes to be interesting.
2. remove $(g, g')$ from consideration if $|g'| < \gamma |g|$ (would never pass a $\gamma$-isomorphism test).
Subgraph isomorphism testing is expensive — prune out $\frac{9}{10}$ of all program pairs from consideration:

1. Ignore any graph which has too few nodes to be interesting.
2. Remove $(g, g')$ from consideration if $|g'| < \gamma|g|$ (would never pass a $\gamma$-isomorphism test).
3. Remove $(g, g')$ if the frequency of their different node types are too different.

For example, if $g$ consists solely of function call nodes and $g'$ consists solely of nodes representing arithmetic operations, $\Rightarrow$ unlikely related.
A PDG is not affected by
statement reordering,
Summary

- A PDG is not affected by
  1. statement reordering,
  2. variable renaming,
A PDG is not affected by

1. statement reordering,
2. variable renaming,
3. replacing while-loops by for-loops,
A PDG is not affected by

1. statement reordering,
2. variable renaming,
3. replacing while-loops by for-loops,
4. flipping the order of branches in if-statements.
A PDG is not affected by
1. statement reordering,
2. variable renaming,
3. replacing while-loops by for-loops,
4. flipping the order of branches in if-statements.

The PDG is affected by
1. inlining and outlining
A PDG is not affected by
1. statement reordering,
2. variable renaming,
3. replacing while-loops by for-loops,
4. flipping the order of branches in if-statements.

The PDG is affected by
1. inlining and outlining
2. add bogus dependencies to introduce spurious edges