Software Similarity Analysis

Clone detection

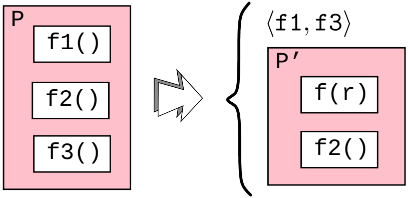
- Duplicates are the result of copy-paste-modify programming.

Clone detection

- Duplicates are the result of copy-paste-modify programming.
- Problem during maintenance — all copies of bugs need to be fixed.

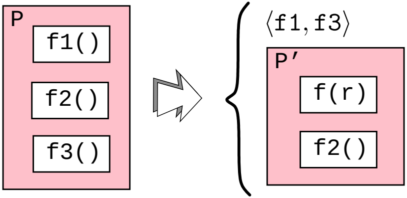
Clone detection

- 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, *threshold*, *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 warrent their own abstraction:

$res \leftarrow \emptyset$
$rep \leftarrow$ `convenient representation of` P
`for every pair of code segments` $f, g \in rep, f \neq g$ `do`
    `if similarity(`f, g`) >` *threshold* `&&`
            $size(f) \geq minsize$ `&&` $size(g) \geq minsize$ `then`
        $res \leftarrow res \cup \langle f, g \rangle$

Clone detection algorithm — Replace Clones

DETECT(P, $threshold$, $minsize$):

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!

Attack model

- We don't expect programmers to be malicious!
- The code becomes naturally "obfuscated" because of the specialization process.

Attack model

- 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.

Attack model

- 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 dector have found them?

Plagiarism of programming assignments

- 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

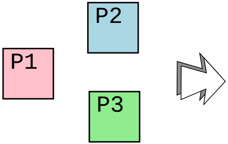
- Hand in a verbatim copy of a friend's program.
- Or, make radical changes to the program to hide the origin of the code.
- "Borrowing" one or more difficult functions from a friend.
- Fishing code out of the trash can.
- Nabbing code off the printer.

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:



$$\begin{cases} \langle \mathrm{P1}, \mathrm{P2} \rangle = 70\% \\ \langle \mathrm{P1}, \mathrm{P3} \rangle = 20\% \\ \langle \mathrm{P2}, \mathrm{P3} \rangle = 10\% \end{cases}$$

Plagiarism detection — Algorithm

$\text{DETECT}(U, \mathit{threshold})$:

```
res ← ∅
for each pair of programs f, g do
    sim ← similarity(f, g)
    if sim > threshold then
        res ← res ∪ ⟨f, g, sim⟩
res ← res sorted on similarity
return res
```

Attack model

- The student needs the code to look "reasonable."

Attack model

- The student needs the code to look “reasonable.”
- General-purpose obfuscation — probably not a good idea.

Attack model

- The student needs the code to look "reasonable."
- General-purpose obfuscation — probably not a good idea.
- Renaming `windowSize` to `sizeOfWindow` — OK.

Attack model

- 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.

Attack model

- 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.

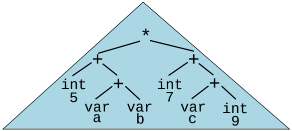
Attack model

- 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

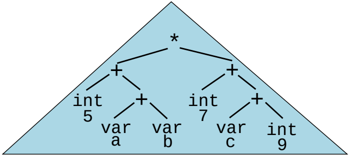


SSEFM: AST-based clone detection

Look for clones in this program:

```
(5 + (a + b)) * (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.

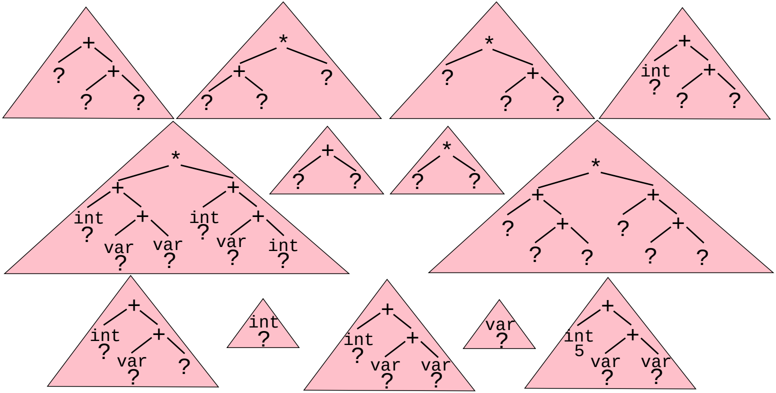
An inefficient clone detector. . .

- 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?

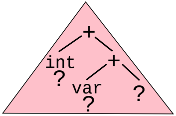
- What's a clone in the context of an AST?
- Simply a tree pattern for which there's more than one match!

What's a clone in an AST?

- What's a clone in the context of 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

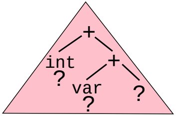
Which patterns would make good clones?

- This pattern seems like it might make a good choice

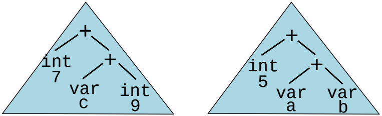


Which patterns would make good clones?

- 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:

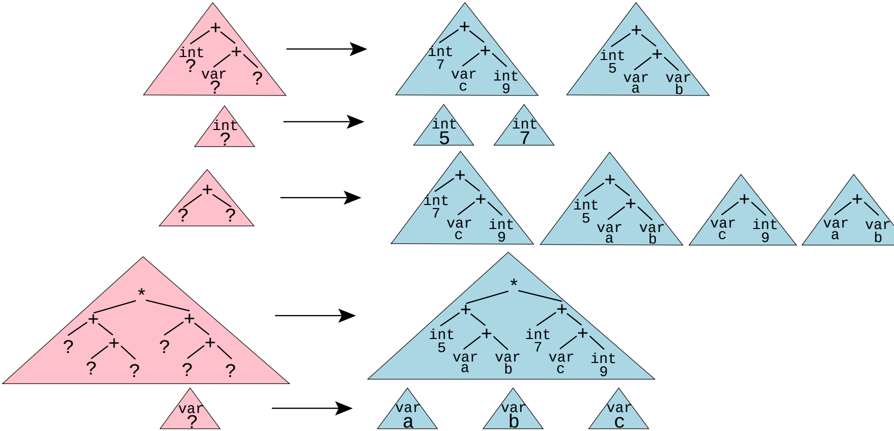
```
#define CLONE(x,y,z) ((x)+((y)+(z)))
CLONE(5,a,b) * CLONE(7,c,9)
```

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!


+
int
?
+
var
?
?
+
int
7
+
var
c
int
9
+
int
5
+
var
a
var
b
int
?
int
5
int
7
+
?
?
+
int
7
+
var
c
int
9
+
int
5
+
var
a
var
b
+
var
c
int
9
+
var
a
var
b
*
+
?
+
?
?
+
?
+
?
?
*
+
int
5
+
var
a
var
b
+
int
7
+
var
c
int
9
var
?
var
a
var
b
var
c

A heuristic algorithm. . .

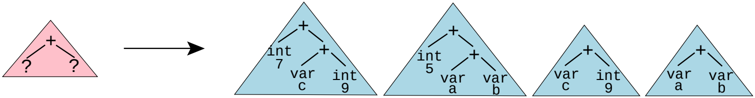
- Won't scale: exponential number of tree patterns.

A heuristic algorithm...

- Won't scale: exponential number of tree patterns.
- Idea: iteratively *grow* larger tree patterns from smaller ones.

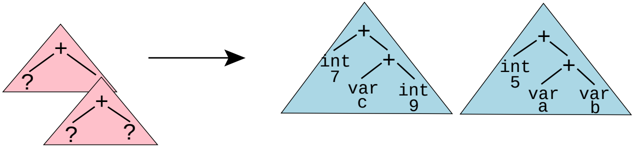
A heuristic algorithm...

- Won't scale: exponential number of tree patterns.
- Idea: iteratively *grow* larger tree patterns from smaller ones.
- Step 1:



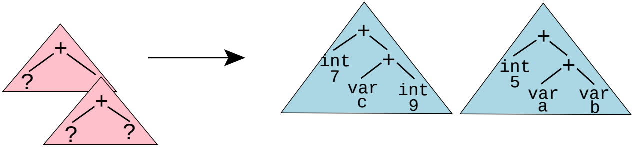
Step 2-3

- We specialize, and the new pattern becomes larger (but only has 2 matches):

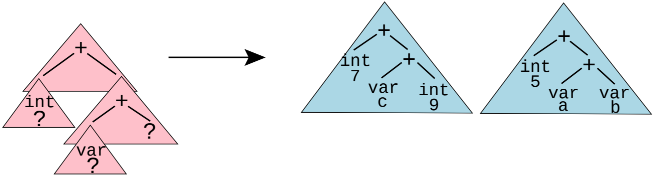


Step 2-3

- 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:

```
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.

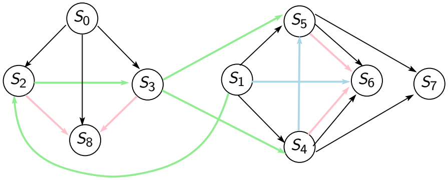
Unfortunately...

- 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

p. 636

PDG-based clone detection



SSKH: PDG-based clone detection

- The nodes of a PDF are the statements of a function.

SSKH: PDG-based clone detection

- 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.

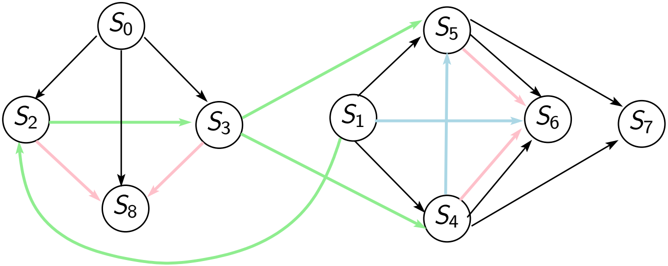
SSKH: PDG-based clone detection

- The nodes of a PDF are the statements of a function.
- There's an edge $m \to n$ 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.

Program Dependence Graph

```
S0 : int k = 0;
S1 : int s = 1;
S2 : while (k < w) {
S3 :    if (x[k] == 1)
S4 :        R = (s*y) % n;
       else
S5 :        R = s;
S6 :    s = R*R % n;
S7 :    L = R;
S8 :    k=k+1;
    }
```

Program Dependence Graph



SSKH: basic idea

- Build a PDG for each function of the program

SSKH: basic idea

- Build a PDG for each function of the program
- Compute two isomorphic subraphs by slicing backwards along dependency edges starting with every pair of *matching nodes*.

SSKH: basic idea

- Build a PDG for each function of the program
- Compute two isomorphic subraphs 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

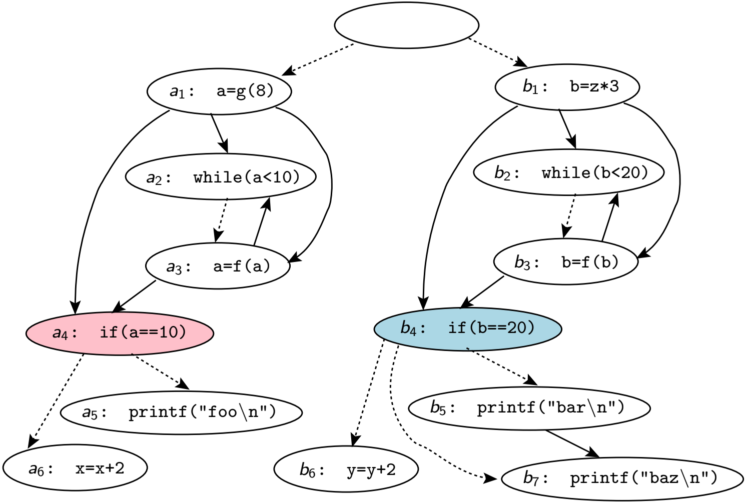
- Build a PDG for each function of the program
- Compute two isomorphic subraphs by slicing backwards along dependency edges starting with every pair of *matching nodes*.
- Two nodes are matching if they have the same syntactic structure.
- Repeat until no more nodes can be added to the slice.

A (contrived) example

```
a1: a = g(8);
b1: b = z*3;
a2: while(a<10)
    a3: a = f(a);
b2: while(b<20)
    b3: b = f(b);
a4: if (a==10) {
    a5: printf("foo\n");
    a6: x=x+2;
}
b4: if (b==20) {
    b5: printf("bar\n");
    b6: y=y+2;
    b7: printf("baz\n");
}
```

- Two similar pieces of code have been intertwined within the same function.

A (contrived) example



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.

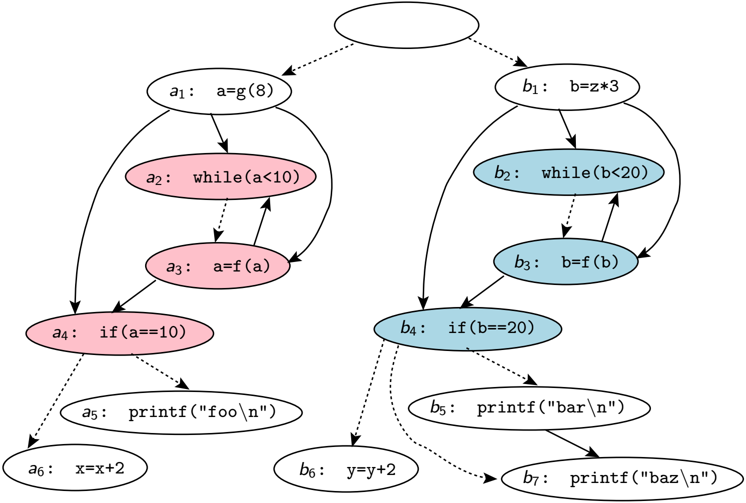
Algorithm: Step 1-3.

- 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.

The PDF after Step 3



Algorithm: Step 4

- a_5/b_5 and a_6/b_6 really should belong to the clone!

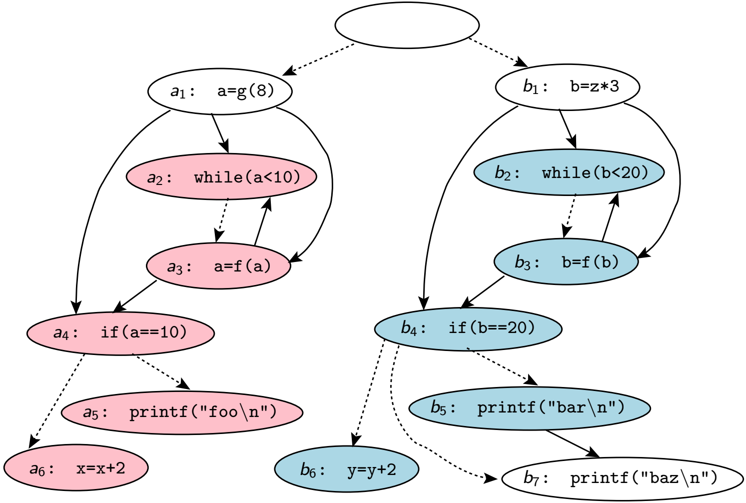
Algorithm: Step 4

- a_5/b_5 and a_6/b_6 really should belong to the clone!
- But, backwards slice won't include them.

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.
- So, slice forward one step from any predicate in an if- and while-statement.

The PDG after Step 4



The extracted clone

```
#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");
```

Summary

- 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

sSLCHY

p. 640

PDG-based plagiarism detection

SSLCHY: PDG-based plagiarism detection

- Uses PDGs, but for plagiarism detection.

SSLCHY: PDG-based plagiarism detection

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

SSLCHY: PDG-based plagiarism detection

- Uses PDGs, but for plagiarism detection.
- Uses a general-purpose subgraph isomorphism algorithm.
- Uses a preprocessing step to weed out unlikely plagiarism candidates.

Plagiarised PDGs?

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

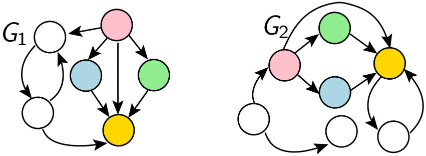
Common Subgraphs

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 ($G = mcs(G_1, G_2)$) 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.

Common Subgraphs — Example

- The colored nodes induce a maximal common subgraph of G_1 and G_2 of four nodes:



Graph similarity and containment

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

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

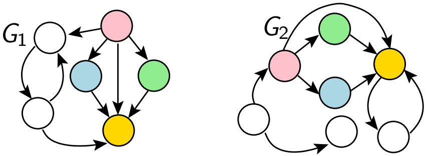
The *containment*(G_1, G_2) of G_1 within G_2 is defined as

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

We say that G_1 is γ-isomorphic to G_2 if

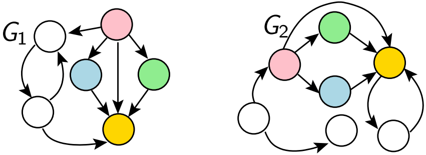
$$containment(G_1, G_2) \geq \gamma, \gamma \in (0, 1].$$

Graph similarity and containment — Example



- *similarity*$(G_1, G_2) = \frac{4}{7}$ and

Graph similarity and containment — Example



- *similarity*$(G_1, G_2) = \frac{4}{7}$ and
- *containment*$(G_1, G_2) = \frac{4}{6}$.

Filtering step

- 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.

Filtering step

- 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 γ-isomorphism test).

Filtering step

- 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 γ-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.

Summary

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

Summary

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

Summary

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

Summary

- 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.

Summary

- 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

Summary

- 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