

1. Intro. OK, you’ve heard about SIGGRAPH; what’s this?

GRAPH-SIG is an experimental program to find potential equivalence classes in automorphism testing. Given a graph G and a vertex v_0 , we compute “signatures” of all vertices such that, if there’s an automorphism that fixes v_0 and takes v to v' , then v and v' will have the same signature.

I plan to generalize the idea, but in this test case I just proceed as follows: First I compute level 0 signatures, which are just the distances from v_0 . Then, given level k signatures σ_k , I compute signatures $\sigma_{k+1}(v) = \prod_{u \sim v} (x - \sigma_k(u))$, where x is a random integer and the multiplication is done mod 2^{64} . We keep going until reaching a round where no class is further refined.

My tentative name for these signatures is “lookahead invariants.”

(Notes for the future: If there’s an automorphism that takes v_0 into v'_0 , then the multiset of signatures computed with respect to v_0 will be the same as the multiset computed with respect to v'_0 , after each round. Also we can generalize to automorphisms that fix k vertices, by defining level 0 signatures as the ordered sequence of distances from v_0, \dots, v_{k-1} . Universal hashing schemes conveniently map such an ordered sequence into a single number.)

```
#define maxn 100      /* upper bound on vertices in the graph */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
#include "gb_flip.h"
long sg[maxn]; /* new signatures found in current class */
Vertex *hd[maxn], *tl[maxn]; /* subdivisions of current class */
main(int argc, char *argv[])
{
    register int i, j, k, r, change;
    register Graph *g;
    register Vertex *u, *v;
    register Arc *a, *b;
    register long x, s;
    Vertex *v0, *prev, *head;
    <Process the command line 2>;
    <Make the initial signatures 3>;
    for (change = 1, r = 1; change; r++) {
        change = 0;
        <Do round r 5>;
    }
}
```

2. \langle Process the command line 2 $\rangle \equiv$

```

if (argc  $\neq$  3) {
    fprintf(stderr, "Usage: %s foo.gb v0\n", argv[0]);
    exit(-1);
}
g = restore_graph(argv[1]);
if ( $\neg$ g) {
    fprintf(stderr, "I couldn't reconstruct graph %s!\n", argv[1]);
    exit(-2);
}
if (g  $>$  maxn) {
    fprintf(stderr, "Recompile me: %g->n=%ld, %maxn=%d!\n", g, maxn);
    exit(-3);
}
gb_init_rand(0); /* the seed doesn't matter much */
for (v = g-vertices; v < g-vertices + g-n; v++)
    if (strcmp(v-name, argv[2])  $\equiv$  0) break;
if (v  $\equiv$  g-vertices + g-n) {
    fprintf(stderr, "I can't find a vertex named '%s'!\n", argv[2]);
    exit(-9);
}
v0 = v;

```

This code is used in section 1.

3. Vertices with the same signature are linked cyclically. As mentioned above, we start by simply computing distances from v_0 .

```

#define sig w.I /* signature of a vertex */
#define link u.V /* link field in a circular list */
#define tag v.I /* to what extent have we processed the vertex? */
 $\langle$  Make the initial signatures 3  $\rangle \equiv$ 
    printf("Initial round:\n");
    for (v = g-vertices; v < g-vertices + g-n; v++) v-sig = -1, v-tag = 0;
    v0-sig = 0, v0-link = v0, k = 1, v = v0;
    while (v) {
        prev = head =  $\Lambda$ ;
        while (1) {
            printf(" %s dist %ld\n", v-name, v-sig);
             $\langle$  Set signature of all v's unseen neighbors to k 4  $\rangle$ ;
            v-tag = k;
            v = v-link;
            if (v-tag) break;
        }
        if (prev  $\equiv$   $\Lambda$ ) break; /* all vertices reachable from v0 have been seen */
        head-link = prev; /* close the cycle */
        v = prev, k++;
    }

```

This code is used in section 1.

4. $\langle \text{Set signature of all } v\text{'s unseen neighbors to } k \text{ 4} \rangle \equiv$

```

for ( $a = v\text{-arcs}$ ;  $a$ ;  $a = a\text{-next}$ ) {
   $u = a\text{-tip}$ ;
  if ( $u\text{-sig} < 0$ ) {
     $u\text{-sig} = k$ ;
    if ( $prev \equiv \Lambda$ )  $head = u$ ;
    else  $u\text{-link} = prev$ ;
     $prev = u$ ;
  }
}

```

This code is used in section 3.

5. Now comes the fun part. As we pass from σ_{r-1} to σ_r , each equivalence class becomes one or more classes.

#define *oldsig* *z.I*

$\langle \text{Do round } r \text{ 5} \rangle \equiv$

```

printf("Round_%d:\n", r);
for ( $v = g\text{-vertices}$ ;  $v < g\text{-vertices} + g\text{-n}$ ;  $v++$ )  $v\text{-oldsig} = v\text{-sig}$ ;
 $k++$ ; /*  $k$  is a unique stamp to identify this round */
 $x = (gb\_next\_rand() \ll 1) + 1$ ; /* pseudorandom number used for new signatures */
for ( $v = g\text{-vertices}$ ;  $v < g\text{-vertices} + g\text{-n}$ ;  $v++$ )
  if ( $v\text{-tag} > 0$ ) {
    if ( $v\text{-tag} \equiv k$ ) continue;
    if ( $v\text{-link} \equiv v$ ) {
      printf("%s_is_fixed\n", v-name); /* class of size 1 */
       $v\text{-tag} = -k$ ; /* we needn't pursue it further */
      continue;
    }
    for ( $j = 0$ ;  $v\text{-tag} \neq k$ ;  $v = u$ ) {
       $u = v\text{-link}$ ;
       $\langle \text{Compute } s = \sigma_r(v) \text{ 6} \rangle$ ;
      printf("%s_%lx\n", v-name, s);
       $v\text{-sig} = s$ ;
      for ( $i = 0$ ,  $sg[j] = s$ ;  $sg[i] \neq s$ ;  $i++$ ) ;
      if ( $i \equiv j$ )  $hd[j] = tl[j] = v$ ,  $j++$ ; /* a new cyclic list begins */
      else  $v\text{-link} = tl[i]$ ,  $tl[i] = v$ ; /* continue building an existing list */
       $v\text{-tag} = k$ ;
    }
    for ( $i = 0$ ;  $i < j$ ;  $i++$ )  $hd[i]\text{-link} = tl[i]$ ; /* complete the cycles */
    if ( $j > 1$ )  $change = 1$ ;
  }
}

```

This code is used in section 1.

6. $\langle \text{Compute } s = \sigma_r(v) \text{ 6} \rangle \equiv$

```

for ( $s = 1$ ,  $a = v\text{-arcs}$ ;  $a$ ;  $a = a\text{-next}$ )  $s *= x - a\text{-tip-oldsig}$ ;

```

This code is used in section 5.

7. Index.

a: [1](#).

Arc: [1](#).

arcs: [4](#), [6](#).

argc: [1](#), [2](#).

argv: [1](#), [2](#).

b: [1](#).

change: [1](#), [5](#).

exit: [2](#).

fprintf: [2](#).

g: [1](#).

gb_init_rand: [2](#).

gb_next_rand: [5](#).

Graph: [1](#).

hd: [1](#), [5](#).

head: [1](#), [3](#), [4](#).

i: [1](#).

j: [1](#).

k: [1](#).

link: [3](#), [4](#), [5](#).

main: [1](#).

maxn: [1](#), [2](#).

name: [2](#), [3](#), [5](#).

next: [4](#), [6](#).

oldsig: [5](#), [6](#).

prev: [1](#), [3](#), [4](#).

printf: [3](#), [5](#).

r: [1](#).

restore_graph: [2](#).

s: [1](#).

sg: [1](#), [5](#).

sig: [3](#), [4](#), [5](#).

stderr: [2](#).

strcmp: [2](#).

tag: [3](#), [5](#).

tip: [4](#), [6](#).

tl: [1](#), [5](#).

u: [1](#).

v: [1](#).

Vertex: [1](#).

vertices: [2](#), [3](#), [5](#).

v0: [1](#), [2](#), [3](#).

x: [1](#).

- ⟨ Compute $s = \sigma_r(v)$ 6 ⟩ Used in section 5.
- ⟨ Do round r 5 ⟩ Used in section 1.
- ⟨ Make the initial signatures 3 ⟩ Used in section 1.
- ⟨ Process the command line 2 ⟩ Used in section 1.
- ⟨ Set signature of all v 's unseen neighbors to k 4 ⟩ Used in section 3.

GRAPH-SIG-V0

	Section	Page
Intro	1	1
Index	7	4