1. Introduction. This GraphBase program is intended to be used only when the Stanford GraphBase is being installed. It invokes the most critical subroutines and creates a file that can be checked against the correct output. The testing is not exhaustive by any means, but it is designed to detect errors of portability—cases where different results might occur on different systems. Thus, if nothing goes wrong, one can assume that the GraphBase routines are probably installed satisfactorily.

The basic idea of TEST_SAMPLE is quite simple: We generate a graph, then print out a few of its salient characteristics. Then we recycle the graph and generate another, etc. The test is passed if the output file matches a "correct" output file generated at Stanford by the author.

Actually there are two output files. The main one, containing samples of graph characteristics, is the standard output. The other, called `test.gb`, is a graph that has been saved in ASCII format with *save_graph*.

```
#include "gb_graph.h"    /* we use the GB_GRAPH data structures */
#include "gb_io.h"    /* and the GraphBase input/output routines */
  ⟨ Include headers for all of the GraphBase generation modules 2 ⟩
  ⟨ Private variables 7 ⟩
  ⟨ Procedures 13 ⟩
int main( )
  { Graph *g, *gg; long i; Vertex *v;    /* temporary registers */
    printf("GraphBase␣samples␣generated␣by␣test_sample:\n");
    ⟨ Save a graph to be restored later 6 ⟩;
    ⟨ Print samples of generated graphs 3 ⟩;
    return 0;    /* normal exit */
  }
```

2. ⟨ Include headers for all of the GraphBase generation modules 2 ⟩ ≡

```
#include "gb_basic.h"    /* we test the basic graph operations */
#include "gb_books.h"    /* and the graphs based on literature */
#include "gb_econ.h"    /* and the graphs based on economic data */
#include "gb_games.h"    /* and the graphs based on football scores */
#include "gb_gates.h"    /* and the graphs based on logic circuits */
#include "gb_lisa.h"    /* and the graphs based on Mona Lisa */
#include "gb_miles.h"    /* and the graphs based on mileage data */
#include "gb_plane.h"    /* and the planar graphs */
#include "gb_raman.h"    /* and the Ramanujan graphs */
#include "gb_rand.h"    /* and the random graphs */
#include "gb_roget.h"    /* and the graphs based on Roget's Thesaurus */
#include "gb_save.h"    /* and we save results in ASCII format */
#include "gb_words.h"    /* and we also test five-letter-word graphs */
```

This code is used in section 1.

3. The subroutine *print_sample*(g, n) will be specified later. It prints global characteristics of g and local characteristics of the nth vertex.

We begin the test cautiously by generating a graph that requires no input data and no pseudo-random numbers. If this test fails, the fault must lie either in GB_GRAPH or GB_RAMAN.

⟨ Print samples of generated graphs 3 ⟩ ≡

```
  print_sample(raman(31L, 3L, 0L, 4L), 4);
```

See also sections 4, 5, 8, 9, 10, and 11.

This code is used in section 1.

4. Next we test part of GB_BASIC that relies on a particular interpretation of the operation '$w \gg= 1$'. If this part of the test fails, please look up 'system dependencies' in the index to GB_BASIC, and correct the problem on your system by making a change file `gb_basic.ch`. (See `queen_wrap.ch` for an example of a change file.)

On the other hand, if TEST_SAMPLE fails only in this particular test while passing all those that follow, chances are excellent that you have a pretty good implementation of the GraphBase anyway, because the bug detected here will rarely show up in practice. Ask yourself: Can I live comfortably with such a bug?

```
⟨ Print samples of generated graphs 3 ⟩ +≡
  print_sample(board(1L, 1L, 2L, −33L, 1L, −#40000000L − #40000000L, 1L), 2000);
    /* coordinates 32 and 33 (only) should wrap around */
```

5. Another system-dependent part of GB_BASIC is tested here, this time involving character codes.

```
⟨ Print samples of generated graphs 3 ⟩ +≡
  print_sample(subsets(32L, 18L, 16L, 0L, 999L, −999L, #80000000L, 1L), 1);
```

6. If `test.gb` fails to match `test.correct`, the most likely culprit is *vert_offset*, a "pointer hack" in GB_BASIC. That macro absolutely must be made to work properly, because it is used heavily. In particular, it is used in the *complement* routine tested here, and in the *gunion* routine tested below.

```
⟨ Save a graph to be restored later 6 ⟩ ≡
  g = random_graph(3L, 10L, 1L, 1L, 0L, Λ, dst, 1L, 2L, 1L);
    /* a random multigraph with 3 vertices, 10 edges */
  gg = complement(g, 1L, 1L, 0L);    /* a copy of g, without multiple edges */
  v = gb_typed_alloc(1, Vertex, gg→data);    /* we create a stray vertex too */
  v→name = gb_save_string("Testing");
  gg→util_types[10] = 'V';
  gg→ww.V = v;    /* the stray vertex is now part of gg */
  save_graph(gg, "test.gb");    /* so it will appear in test.gb (we hope) */
  gb_recycle(g); gb_recycle(gg);
```

This code is used in section 1.

7.

```
⟨ Private variables 7 ⟩ ≡
  static long dst[] = {#20000000, #10000000, #10000000};
    /* a probability distribution with frequencies 50%, 25%, 25% */
```

See also section 12.

This code is used in section 1.

8. Now we try to reconstruct the graph we saved before, and we also randomize its lengths.

```
⟨ Print samples of generated graphs 3 ⟩ +≡
  g = restore_graph("test.gb");
  if (i = random_lengths(g, 0L, 10L, 12L, dst, 2L))
    printf("\nFailure␣code␣%ld␣returned␣by␣random_lengths!\n", i);
  else {
    gg = random_graph(3L, 10L, 1L, 1L, 0L, Λ, dst, 1L, 2L, 1L);    /* same as before */
    print_sample(gunion(g, gg, 1L, 0L), 2);
    gb_recycle(g); gb_recycle(gg);
  }
```

9. Partial evaluation of a RISC circuit involves fairly intricate pointer manipulation, so this step should help to test the portability of the author's favorite programming tricks.

```
⟨ Print samples of generated graphs 3 ⟩ +≡
  print_sample(partial_gates(risc(0L), 1L, 43210L, 98765L, Λ), 79);
```

10. Now we're ready to test the mechanics of reading data files, sorting with GB_SORT, and heavy randomization. Lots of computation takes place in this section.

⟨ Print samples of generated graphs 3 ⟩ +≡
```
print_sample(book("homer",500L,400L,2L,12L,10000L,-123456L,789L),81);
print_sample(econ(40L,0L,400L,-111L),11);
print_sample(games(60L,70L,80L,-90L,-101L,60L,0L,999999999L),14);
print_sample(miles(50L,-500L,100L,1L,500L,5L,314159L),20);
print_sample(plane_lisa(100L,100L,50L,1L,300L,1L,200L,50L*299L*199L,200L*299L*199L),1294);
print_sample(plane_miles(50L,500L,-100L,1L,1L,40000L,271818L),14);
print_sample(random_bigraph(300L,3L,1000L,-1L,0L,dst,-500L,500L,666L),3);
print_sample(roget(1000L,3L,1009L,1009L),40);
```

11. Finally, here's a picky, picky test that is supposed to fail the first time, succeed the second. (The weight vector just barely exceeds the maximum weight threshold allowed by GB_WORDS. That test is ultraconservative, but eminently reasonable nevertheless.)

⟨ Print samples of generated graphs 3 ⟩ +≡
```
print_sample(words(100L,wt_vector,70000000L,69L),5);
wt_vector[1]++;
print_sample(words(100L,wt_vector,70000000L,69L),5);
print_sample(words(0L,Λ,0L,69L),5555);
```

12. ⟨ Private variables 7 ⟩ +≡
```
static long wt_vector[] = {100,-80589,50000,18935,-18935,18935,18935,18935,18935};
```

13. Printing the sample data. Given a graph g in GraphBase format and an integer n, the subroutine *print_sample*(g, n) will output global characteristics of g, such as its name and size, together with detailed information about its nth vertex. Then g will be shredded and recycled; the calling routine should not refer to it again.

```
⟨ Procedures 13 ⟩ ≡
  static void pr_vert( );     /* a subroutine for printing a vertex is declared below */
  static void pr_arc( );      /* likewise for arcs */
  static void pr_util( );     /* and for utility fields in general */
  static void print_sample(g, n)
      Graph *g;     /* graph to be sampled and destroyed */
      int n;     /* index to the sampled vertex */
  {
    printf("\n");
    if (g ≡ Λ) {
      printf("Ooops, we just ran into panic code %ld!\n", panic_code);
      if (io_errors) printf("(The I/O error code is 0x%lx)\n", (unsigned long) io_errors);
    } else {
      ⟨ Print global characteristics of g 18 ⟩;
      ⟨ Print information about the nth vertex 17 ⟩;
      gb_recycle(g);
    }
  }
```

See also sections 14, 15, and 16.

This code is used in section 1.

14. The graph's *util_types* are used to determine how much information should be printed. A level parameter also helps control the verbosity of printout. In the most verbose mode, each utility field that points to a vertex or arc, or contains integer or string data, will be printed.

```
⟨ Procedures 13 ⟩ +≡
  static void pr_vert(v, l, s)
      Vertex *v;     /* vertex to be printed */
      int l;     /* ≤ 0 if the output should be terse */
      char *s;     /* format for graph utility fields */
  {
    if (v ≡ Λ) printf("NULL");
    else if (is_boolean(v)) printf("ONE");     /* see GB_GATES */
    else {
      printf("\"%s\"", v→name);
      pr_util(v→u, s[0], l − 1, s);
      pr_util(v→v, s[1], l − 1, s);
      pr_util(v→w, s[2], l − 1, s);
      pr_util(v→x, s[3], l − 1, s);
      pr_util(v→y, s[4], l − 1, s);
      pr_util(v→z, s[5], l − 1, s);
      if (l > 0) { register Arc *a;
        for (a = v→arcs; a; a = a→next) {
          printf("\n␣␣␣␣");
          pr_arc(a, 1, s);
        }
      }
    }
  }
```

15.

```
⟨ Procedures 13 ⟩ +≡
  static void pr_arc(a, l, s)
      Arc *a;     /* non-null arc to be printed */
      int l;     /* ≤ 0 if the output should be terse */
      char *s;     /* format for graph utility fields */
  {
    printf("->");
    pr_vert(a→tip, 0, s);
    if (l > 0) {
      printf(",␣%ld", a→len);
      pr_util(a→a, s[6], l − 1, s);
      pr_util(a→b, s[7], l − 1, s);
    }
  }
```

16. ⟨ Procedures 13 ⟩ +≡

```
static void pr_util(u, c, l, s)
    util u;     /* a utility field to be printed */
    char c;     /* its type code */
    int l;     /* 0 if output should be terse, −1 if pointers omitted */
    char *s;     /* utility types for overall graph */
{
  switch (c) {
  case 'I': printf("[%ld]", u.I); break;
  case 'S': printf("[\"%s\"]", u.S ? u.S : "(null)"); break;
  case 'A':
    if (l < 0) break;
    printf("[");
    if (u.A ≡ Λ) printf("NULL");
    else pr_arc(u.A, l, s);
    printf("]");
    break;
  case 'V':
    if (l < 0) break;     /* avoid infinite recursion */
    printf("[");
    pr_vert(u.V, l, s);
    printf("]");
  default: break;     /* case 'Z' does nothing, other cases won't occur */
  }
}
```

17. ⟨ Print information about the nth vertex 17 ⟩ ≡

```
printf("V%d:␣", n);
if (n ≥ g→n ∨ n < 0) printf("index␣is␣out␣of␣range!\n");
else {
  pr_vert(g→vertices + n, 1, g→util_types);
  printf("\n");
}
```

This code is used in section 13.

18. ⟨ Print global characteristics of g 18 ⟩ ≡

```
printf("\"%s\"\n%ld␣vertices,␣%ld␣arcs,␣util_types␣%s", g→id, g→n, g→m, g→util_types);
pr_util(g→uu, g→util_types[8], 0, g→util_types);
pr_util(g→vv, g→util_types[9], 0, g→util_types);
pr_util(g→ww, g→util_types[10], 0, g→util_types);
pr_util(g→xx, g→util_types[11], 0, g→util_types);
pr_util(g→yy, g→util_types[12], 0, g→util_types);
pr_util(g→zz, g→util_types[13], 0, g→util_types);
printf("\n");
```

This code is used in section 13.

19. Index. We end with the customary list of identifiers, showing where they are used and where they are defined.

a: 14, 15.
Arc: 14, 15.
arcs: 14.
board: 4.
book: 10.
c: 16.
complement: 6.
data: 6.
dst: 6, 7, 8, 10.
econ: 10.
g: 1, 13.
games: 10.
gb_recycle: 6, 8, 13.
gb_save_string: 6.
gb_typed_alloc: 6.
gg: 1, 6, 8.
Graph: 1, 13.
gunion: 6, 8.
i: 1.
id: 18.
io_errors: 13.
is_boolean: 14.
l: 14, 15, 16.
len: 15.
main: 1.
miles: 10.
n: 13.
name: 6, 14.
next: 14.
panic_code: 13.
partial_gates: 9.
plane_lisa: 10.
plane_miles: 10.
pr_arc: 13, 14, 15, 16.
pr_util: 13, 14, 15, 16, 18.
pr_vert: 13, 14, 15, 16, 17.
print_sample: 3, 4, 5, 8, 9, 10, 11, 13.
printf: 1, 8, 13, 14, 15, 16, 17, 18.
raman: 3.
random_bigraph: 10.
random_graph: 6, 8.
random_lengths: 8.
restore_graph: 8.
risc: 9.
roget: 10.
s: 14, 15, 16.
save_graph: 1, 6.
subsets: 5.
tip: 15.
u: 16.
util: 16.
util_types: 6, 14, 17, 18.
uu: 18.
v: 1, 14.
vert_offset: 6.
Vertex: 1, 6, 14.
vertices: 17.
vv: 18.
words: 11.
wt_vector: 11, 12.
ww: 6, 18.
xx: 18.
yy: 18.
zz: 18.

⟨ Include headers for all of the GraphBase generation modules 2 ⟩ Used in section 1.
⟨ Print global characteristics of g 18 ⟩ Used in section 13.
⟨ Print information about the nth vertex 17 ⟩ Used in section 13.
⟨ Print samples of generated graphs 3, 4, 5, 8, 9, 10, 11 ⟩ Used in section 1.
⟨ Private variables 7, 12 ⟩ Used in section 1.
⟨ Procedures 13, 14, 15, 16 ⟩ Used in section 1.
⟨ Save a graph to be restored later 6 ⟩ Used in section 1.

August 4, 2025 at 09:09

TEST_SAMPLE

	Section	Page
Introduction	1	1
Printing the sample data	13	4
Index	19	7

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.