(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. Given a graph, this program computes a Monte Carlo estimate of the number of simple paths from a given source vertex to a given target vertex. The estimate is simply the product of the number of choices at every step.

(I don't have time today to make this program work for digraphs. But I think such an extension would be pretty easy. The main change would be to create a table of inverse arcs.)

Everything is quite straightforward, except that there's an interesting algorithm to update the currently shortest distances from each unused vertex to the target. This algorithm guarantees that the procedure won't get stuck in a dead end.

The first three command-line parameters are the same as those of SIMPATH, a companion program by which the exact number of paths can be calculated via ZDD techniques (if the graph isn't too large). The fourth parameter is a seed for the random numbers. Additional parameters are ignored except that they increase the verbosity of output.

```
#define infty 9999    /* infinity (or close enough) */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
#include "gb_flip.h"
  int seed;    /* seed for the random number generator */
  int vbose;    /* level of verbosity */
  ⟨Subroutines 5⟩
  main(int argc, char *argv[])
  {
    register int d, k, l;
    register Graph *g;
    register Arc *a, *b;
    register Vertex *u, *v, *vv, *head, *tail;
    register double e;
    Vertex *source = Λ, *target = Λ;
    ⟨Process the command line 2⟩;
    ⟨Initialize the table of distances 3⟩;
    if (source→dist ≡ infty) {
      fprintf(stderr, "There's␣no␣path␣from␣%s␣to␣%s!\n", source→name, target→name);
      exit(−99);
    }
    ⟨Do a random walk and print the estimate 4⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
if (argc < 5 ∨ sscanf(argv[4], "%d", &seed) ≠ 1) {
   fprintf(stderr, "Usage:␣%s␣foo.gb␣source␣target␣seed␣[verbose]␣[extraverbose]\n", argv[0]);
   exit(−1);
}
vbose = argc − 5;
g = restore_graph(argv[1]);
if (¬g) {
   fprintf(stderr, "I␣can't␣input␣the␣graph␣%s␣(panc␣code␣%ld)!\n", argv[1], panic_code);
   exit(−2);
}
if (g→n > infty) {
   fprintf(stderr, "Sorry,␣that␣graph␣has␣%ld␣vertices;␣", g→n);
   fprintf(stderr, "I␣can't␣handle␣more␣than␣%d!\n", infty);
   exit(−2);
}
for (v = g→vertices; v < g→vertices + g→n; v++) {
   if (strcmp(argv[2], v→name) ≡ 0) source = v;
   if (strcmp(argv[3], v→name) ≡ 0) target = v;
   for (a = v→arcs; a; a = a→next) {
      u = a→tip;
      if (u ≡ v) {
         fprintf(stderr, "Sorry,␣the␣graph␣contains␣a␣loop␣%s--%s!\n", v→name, v→name);
         exit(−4);
      }
      b = (v < u ? a + 1 : a − 1);
      if (b→tip ≠ v) {
         fprintf(stderr, "Sorry,␣the␣graph␣isn't␣undirected!\n");
         fprintf(stderr, "(%s->%s␣has␣mate␣pointing␣to␣%s)\n", v→name, u→name, b→tip→name);
         exit(−5);
      }
   }
}
if (¬source) {
   fprintf(stderr, "I␣can't␣find␣source␣vertex␣%s␣in␣the␣graph!\n", argv[2]);
   exit(−6);
}
if (¬target) {
   fprintf(stderr, "I␣can't␣find␣target␣vertex␣%s␣in␣the␣graph!\n", argv[3]);
   exit(−7);
}
gb_init_rand(seed);
```

This code is used in section 1.

3. Ye olde breadth-first search.

```
#define dist u.I
#define stamp v.V
#define link w.V
⟨Initialize the table of distances 3⟩ ≡
  for (v = g→vertices; v < g→vertices + g→n; v++) v→dist = infty, v→stamp = Λ;
  for (target→dist = 0, head = tail = target; ; head = head→link) {
    d = head→dist;
    for (a = head→arcs; a; a = a→next) {
      v = a→tip;
      if (v→dist ≡ infty) v→dist = d + 1, tail→link = v, tail = v;
    }
    if (head ≡ tail) break;
  }
```

This code is used in section 1.

4.
```
#define choice(k) ((g→vertices + (k))→z.V)
⟨Do a random walk and print the estimate 4⟩ ≡
  for (v = source, e = 1.0, l = 0; v ≠ target; l++, v = vv, e *= d) {
    axe(v);
    for (a = v→arcs, d = 0; a; a = a→next) {
      u = a→tip;
      if (u→dist < infty) choice(d) = u, d++;
    }
    if (d ≡ 0) {
      fprintf(stderr, "Oh␣oh,␣I␣goofed.\n");
      exit(−666);
    }
    k = gb_unif_rand(d);
    vv = choice(k);
    if (vbose) fprintf(stderr, "␣%s␣(%d␣of␣%d)\n", vv→name, k + 1, d);
  }
  printf("length␣%d,␣est␣%20.1f\n", l, e);
```

This code is used in section 1.

5. The interesting part. When a vertex v becomes part of the path, the distance of every other vertex u will increase unless u has a path of the same length to *target* that doesn't go through v.

We start by identifying all "tarnished" vertices whose distances must change, in order of their current distances from the *target*. At the same time we build a queue of "resource" vertices, whose distances are known to be unchangeable. (This list of resources isn't complete. But it is large enough so that every tarnished vertex that can still get to the target will have a shortest path through at least one resource.)

In the second phase we use those resources to find the new shortest paths for all the tarnished ones that aren't dead.

```
⟨Subroutines 5⟩ ≡
  void axe(Vertex *vv)
  {
    register Vertex *u, *v, *w, *tarnished, *resource, *rtail, *stack, *bot;
    register Arc *a, *b;
    register int d;

    d = vv→dist;
    vv→dist = infty + 1;
    ⟨Do phase 1, identifying tarnished and resource vertices 6⟩;
    ⟨Do phase 2, updating the nondead tarnishees 10⟩;
  }
```

This code is used in section 1.

6. Fortuitously, we can maintain the *tarnished* list as a stack, not a queue, because all of its items are at the same distance. The previous *tarnished* list is traversed while we're building a new one.

```
⟨Do phase 1, identifying tarnished and resource vertices 6⟩ ≡
  for (resource = Λ, tarnished = vv, vv→link = Λ; tarnished; d++) {
    for (v = tarnished, tarnished = stack = Λ; v; v = v→link) {
        /* vertices on the new tarnished list will have former distance d + 1 */
      for (a = v→arcs; a; a = a→next) {
        u = a→tip;
        if (u→dist < d) continue;      /* this can happen only when v = vv */
        if (u→dist ≥ infty) continue;      /* u is gone or already on the tarnished list */
        if (u→dist ≡ d) ⟨Make u a resource 7⟩
        else ⟨If u is tarnished, put it on the list 8⟩;
      }
    }
    ⟨Append stack to resource 9⟩;
  }
```

This code is used in section 5.

7. This code applies when v, which a tarnished vertex formerly at distance d, is adjacent to u, which is an untarnished vertex still at distance d. (It's a scenario that would be impossible in a bipartite graph, but this program is supposed to be more general.)

Every vertex in the resource queue currently has distance d or less, so we can maintain that list in order of distance by appending u at the end.

We stamp a vertex with *vv* when it goes into the resource queue, so that vertices don't get queued twice.

```
⟨Make $u$ a resource 7⟩ ≡
  {
    if ($u\to stamp \neq vv \land v \neq vv$) {
      if ($vbose > 1$) fprintf(stderr, "␣early␣resource␣%s␣at␣dist␣%d\n", $u\to name, d$);
      $u\to stamp = vv$;
      if ($resource \equiv \Lambda$) $resource = rtail = u$;
      else $rtail\to link = u, rtail = u$;
    }
  }
```

This code is used in section 6.

8. At this point $u\to dist = d + 1$. Vertex u is tarnished if and only if none of its neighbors has distance d.

If u isn't tarnished, we will want it to be a resource, because it provides a potential lifeline to v. (However, we don't make it a resource if $v = vv$, because we don't care about resuscitating *vv*.) In such cases we put u on *stack* temporarily, because (in nonbipartite graphs) the *resource* list isn't yet ready for distance $d + 1$.

```
⟨If $u$ is tarnished, put it on the list 8⟩ ≡
  {
    if ($u\to dist \neq d + 1$) {
      fprintf(stderr, "Confusion:␣%s␣at␣distance␣%ld,␣not␣%d!\n", $u\to name, u\to dist, d + 1$);
      exit($-999$);
    }
    for ($b = u\to arcs$; $b$; $b = b\to next$) {
      $w = b\to tip$;
      if ($w\to dist \equiv d$) goto okay;
    }
    if ($vbose > 1$) fprintf(stderr, "␣tarnished␣%s␣at␣dist␣%d\n", $u\to name, d + 1$);
    $u\to link = tarnished, u\to dist = infty, tarnished = u$;
    continue;
  okay: if ($u\to stamp \neq vv \land v \neq vv$) {
      if ($vbose > 1$) fprintf(stderr, "␣resource␣%s␣at␣dist␣%d\n", $u\to name, d + 1$);
      $u\to stamp = vv$;
      if ($stack \equiv \Lambda$) $stack = bot = u$;
      else $u\to link = stack, stack = u$;
    }
  }
```

This code is used in section 6.

9. I'm intentionally living a bit dangerously by leaving the *link* field undefined at the tail end of the resource queue. Therefore I'm documenting that fact here: This program assumes that *rtail* is undefined when $resource = \Lambda$, and that *rtail*→*link* is undefined when $resource \neq \Lambda$.

```
⟨ Append stack to resource 9 ⟩ ≡
  if (stack) {
    if ($resource \equiv \Lambda$) $resource = stack, rtail = bot$;
    else $rtail\to link = stack, rtail = bot$;
  }
```

This code is used in section 6.

10. During phase 2, newly updated vertices become resources. Once again it's possible to keep them on two stacks of equal-distance vertices.

```
⟨ Do phase 2, updating the nondead tarnishees 10 ⟩ ≡
  if ($\neg resource$) return;
  $rtail\to link = \Lambda$;
  for ($stack = \Lambda$; $resource \neq \Lambda \lor stack \neq \Lambda$; ) {
    if (stack) {    /* now $resource\to dist \geq stack\to dist$, if resource exists */
      for ($d = stack\to dist, v = stack, stack = \Lambda$; $v$; $v = v\to link$) ⟨ Update the neighbors of $v$ 11 ⟩;
    }
    else $d = resource\to dist$;
    while ($resource \land resource\to dist \equiv d$) {
      $v = resource, resource = v\to link$;
      ⟨ Update the neighbors of $v$ 11 ⟩;
    }
  }
```

This code is used in section 5.

11. ⟨ Update the neighbors of v 11 ⟩ ≡

```
  for ($a = v\to arcs$; $a$; $a = a\to next$) {
    $u = a\to tip$;
    if ($u\to dist \equiv infty$) {
      if ($vbose > 1$) fprintf(stderr, "␣updated␣%s␣at␣dist␣%d\n", $u\to name, d + 1$);
      $u\to dist = d + 1, u\to link = stack, stack = u$;    /* no need to stamp $u$ */
    }
  }
```

This code is used in section 10.

12. Index.

WALK-RANDOM