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

1. Intro. This is an implementation of Tarjan's algorithm for strong components (Algorithm 7.4.1.2T), together with his algorithm for weak components (Algorithm 7.4.1.2W), based on my current drafts of those algorithms in prefascicle 12a.

The digraph to be analyzed should be named on the command line. If you'd also like to delete some of its arcs, you can name them on the command line too, by saying '`-`u `--`v' to delete $u \longrightarrow v$.

```
#define o  mems++      /* count one memory reference */
#define oo  mems += 2
#define ooo  mems += 3
#define ox  xmems++      /* count one extra memory reference */
#define oox  xmems += 2
#define O  "%"      /* used for percent signs in format strings */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
  int debugging;
  unsigned long long mems;
  unsigned long long xmems;
  int comps, wcomps;
  int n;
  long int wpsr;      /* the current value of wp→srank */
  int c1p;      /* are we in Case 1′? */
  Graph *gg;
  Vertex *settled;
  ⟨ Subroutines 6 ⟩;
  main(int argc, char *argv[])
  {
    register int p, lowv;
    register Graph *g;
    register Vertex *t, *u, *v, *w, *root, *sink, *wp, *prev;
    register Arc *a, *b;
    ⟨ Process the command line 2 ⟩;
    ⟨ Do the algorithm 9 ⟩;
    ⟨ Say farewell 16 ⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
if (argc & 1) {
  fprintf(stderr, "Usage: "O"s foo.gb [-U --V]*\n", argv[0]);
  exit(-1);
}
gg = g = restore_graph(argv[1]);
if (¬g) {
  fprintf(stderr, "I couldn't reconstruct graph "O"s!\n", argv[1]);
  exit(-2);
}
n = g→n;
⟨ Optionally delete arcs 3 ⟩;
(g→vertices + n)→u.V = g→vertices;
if ((g→vertices + g→n)→u.I ≤ n) {
  fprintf(stderr, "Vertex pointers come too early in memory!!\n");
  exit(-666);
}
⟨ Extend the graph 4 ⟩;
printf("Strong components of "O"s", g→id);
for (p = 2; p < argc; p += 2) printf(" "O"s "O"s", argv[p], argv[p + 1]);
printf(":\n");
```

This code is used in section 1.

3. ⟨ Optionally delete arcs 3 ⟩ ≡

```
for (p = 2; p < argc; p += 2) {
  if (argv[p][0] ≠ '-' ∨ argv[p + 1][0] ≠ '-' ∨ argv[p + 1][1] ≠ '-') {
    fprintf(stderr, "improper command-line arguments "O"s "O"s!\n", argv[p], argv[p + 1]);
    exit(-3);
  }
  for (v = g→vertices; v < g→vertices + n; v++)
    if (strcmp(v→name, argv[p] + 1) ≡ 0) {
      for (b = Λ, a = v→arcs; a; b = a, a = a→next) {
        if (strcmp(a→tip→name, argv[p + 1] + 2) ≡ 0) break;
      }
      if (¬a) v = g→vertices + n;
      else if (b) b→next = a→next; else v→arcs = a→next;
      break;
    }
  if (v ≡ g→vertices + n) {
    fprintf(stderr, "I don't see the arc "O"s->"O"s!\n", &argv[p][1], &argv[p + 1][2]);
    exit(-4);
  }
}
```

This code is used in section 2.

4. Each vertex of a GraphBase graph has six utility fields. But the algorithms implemented here need nine. So we allocate space for n more vertices, effectively giving each vertex five more fields to play with. (We use up one field to provide an associated vertex $v→ext$, which has six fields available.)

⟨ Extend the graph 4 ⟩ ≡

```
t = gb_typed_alloc(n + 1, Vertex, g→aux_data);
for (v = g→vertices; v ≤ g→vertices + n; v++, t++) v→ext = t;
```

This code is used in section 2.

5. Here's how the utility fields are assigned to the book's names for those fields.

I use the fact that GraphBase graphs provide *extra_n* vertices, so that it's OK for me to store something in $g\rightarrow vertices + g\rightarrow n$, which Algorithm T calls `SENT`. (The extra vertices show up in the space for vertices that's allocated on the first line of '`.gb`' format; the value of $g\rightarrow n$ on the second line is smaller.)

The `REP` field in Algorithm T has two forms, either a small integer or an offset vertex. Here we simply use the vertex itself, calling it '*rep*' in a field shared with the integer '*low*' field. That is safe, because of the test on vertex pointers made above.

```
#define sent (g→vertices + g→n)
#define par u.V /* PARENT in the book */
#define low v.I /* LOW (when REP equals LOW) */
#define rep v.V /* v′ (when REP equals SENT + v′) */
#define link w.V /* LINK */
#define arc x.A /* ARC */
#define srank y.I /* SRANK */
#define ext z.V
#define hit ext→u.I /* HIT */
#define whit ext→v.I /* WHIT */
#define wlink ext→w.V /* WLINK */
#define src ext→x.V /* SRC */
```

6. The following debugging-oriented subroutine clarifies the conventions used here.

```
#define symlink(u)
        ((u) ≡ gg→vertices + n ? "END" : ((u) < gg→vertices + n) ∧ ((u) ≥ gg→vertices) ? (u)→name : "??")
⟨ Subroutines 6 ⟩ ≡
  void print_vert(Vertex *v)
  {
    register int k;
    register Vertex *u;
    register Arc *a;
    if (¬v) fprintf(stderr, "NULL");
    else if (v ≡ gg→vertices + n) fprintf(stderr, "SENT");
    else if (v < gg→vertices ∨ v > gg→vertices + n) fprintf(stderr, "␣(out␣of␣range)");
    else {
      fprintf(stderr, ""O"s:", v→name);
      u = v→par;
      if (¬u) fprintf(stderr, "␣(unseen)");
      else {
        fprintf(stderr, "␣parent="O"s", symlink(u));
        k = v→low, u = v→rep;
        if (k ≤ n) fprintf(stderr, "␣low="O"d", k);
        else fprintf(stderr, "␣rep="O"s", u→name);
        if (v→link) fprintf(stderr, "␣link="O"s", symlink(v→link));
        if (v→arc) fprintf(stderr, "␣arc="O"s", symlink(v→arc→tip));
        if (v→rep ≡ v) ⟨ Print fields used in strong component leaders 7 ⟩;
      }
    }
    fprintf(stderr, "\n");
  }
```

See also section 8.

This code is used in section 1.

7. ⟨ Print fields used in strong component leaders 7 ⟩ ≡

```
{
  fprintf(stderr, "␣srank="O"ld", v→srank);
  if (v→hit) fprintf(stderr, "␣hit");
  if (v→whit) fprintf(stderr, "␣whit");
  fprintf(stderr, "␣wlink="O"s", symlink(v→wlink));
  fprintf(stderr, "␣src="O"s", symlink(v→src));
}
```

This code is used in section 6.

8. ⟨ Subroutines 6 ⟩ +≡

```
void print_settled(void)
{
  register Vertex *w;
  for (w = settled; w; w = w→link) print_vert(w);
}
```

9. ⟨ Do the algorithm 9 ⟩ ≡

```
  sent→low = 0, sent→hit = 0, sent→srank = wpsr = n;
  wp = prev = sent;
t1: for (w = g→vertices; w < sent; w++) o, w→par = Λ, oox, w→hit = w→whit = 0;
  p = 0;      /* at this point w = sent */
  sink = sent, settled = Λ;
t2: if (w ≡ g→vertices) goto done;
  if (o, (--w)→par ≠ Λ) goto t2;
  v = w, v→par = sent, root = v;
t3: o, a = v→arcs;
  oo, lowv = v→low = ++p, v→link = sent;
t4: if (a ≡ Λ) goto t7;
t5: o, u = a→tip, a = a→next;
t6: if (o, u→par ≡ Λ) {
    oo, u→par = v, v→arc = a, v = u;
    goto t3;
  }
  if (u ≡ root ∧ p ≡ g→n) ⟨ Prepare to terminate early, and goto t8 10 ⟩;
  if (o, u→low < lowv) oo, lowv = v→low = u→low, v→link = Λ;
  goto t4;
t7: o, u = v→par;
  if (o, v→link ≡ sent) goto t8;
  ⟨ Adjust u→low with respect to its tree child v 11 ⟩;
  o, v→link = sink, sink = v;
  goto t9;
t8: ⟨ Produce a new strong component whose leader is v 12 ⟩;
t9: if (u ≡ sent) goto t2;
  oo, v = u, a = v→arc, lowv = v→low;
  goto t4;
done:
```

This code is used in section 1.

10. ⟨ Prepare to terminate early, and **goto** *t8* 10 ⟩ ≡

```
{
  while (v ≠ root) oo, v→link = sink, sink = v, v = v→par;
  o, u = sent, lowv = 1;
  goto t8;
}
```

This code is used in section 9.

11. At this point *lowv* is `LOW(v)`; it might or might not have been stored in *v*→*low*.

⟨ Adjust *u*→*low* with respect to its tree child *v* 11 ⟩ ≡

```
if (o, lowv < u→low) oo, u→low = lowv, u→link = Λ;
```

This code is used in section 9.

12. The *settled* stack retains the links of the items removed from the *sink* stack, followed by *v*, followed by its former contents.

⟨ Produce a new strong component whose leader is *v* 12 ⟩ ≡

```
comps++;
oox, v→srank = n − comps, v→src = prev, prev = v;
printf("strong␣component␣"O"s("O"ld):\n", v→name, v→srank);
ox, v→link = sink, t = v;
while (o, sink→low ≥ lowv) {
  printf("+"O"s\n", sink→name);
  o, sink→rep = v, t = sink;
  o, sink = sink→link;
}
o, v→rep = v;
ox, t→link = settled, settled = v;
⟨ Update weak components 13 ⟩;
```

This code is used in section 9.

13. We perform steps W3 thru W8 of Algorithm 7.4.1.2W at this point. (Variable w in this program plays the role of w' in the book. We are careful not to clobber the values of this program's variables u and w, because they belong to Algorithm 7.4.1.2T.)

This program includes some optimizations that are mentioned in the exercises of Section 7.4.1.2: (1) There's no need to set `WP` $\leftarrow v$ in step W8; our `WP` is really the leader of W_2, not W_1. (2) The `HIT` field needs to be set only in Case 1'.

```
⟨ Update weak components 13 ⟩ ≡
  {
    register Vertex *u, *w, *up, *vp;      /* redeclare u and w for temporary use */
  w2: ⟨ Run through all arcs from strong component v, setting whit and t 14 ⟩;
  w3: if (vp ≡ sent) { wp = sent, wcomps = 1, wpsr = n; goto w6; }
    for (ox, w = v→src, c1p = 1; ox, vp→srank ≥ wpsr; oox, wp = w→wlink, wpsr = wp→srank)
      w = wp, c1p = 0, wcomps−−;
    u = w;
  w4: if (u ≡ sent) { wcomps++; goto w5; }
    if (ox, ¬u→whit) goto w6;
    for (ox, up = u, u = u→src; ox, u→hit; oox, u = u→src, up→src = u) ;
    goto w4;
  w5: ox, wp = w, wpsr = wp→srank;
    if (c1p) ox, c1p = 0, v→src = sent;      /* Case 1 */
  w6: ox, v→wlink = wp;
    printf("␣weak␣to␣"O"s("O"ld)\n", symlink(wp), wpsr);
  w7: ⟨ Run through all arcs from strong component v, resetting whit 15 ⟩;
    if (debugging) print_settled( );
  }
```

This code is used in section 12.

14. ⟨ Run through all arcs from strong component v, setting *whit* and t 14 ⟩ ≡

```
  for (w = settled, vp = sent; ; ox, w = w→link) {
    for (ox, a = w→arcs; a; ox, a = a→next) {
      oox, u = a→tip→rep;
      if (u ≡ v) continue;
      ox, u→whit = 1;
      if (ox, u→srank < vp→srank) vp = u;
    }
    if (w ≡ t) break;
  }
```

This code is used in section 13.

15. ⟨ Run through all arcs from strong component v, resetting *whit* 15 ⟩ ≡

```
  for (w = settled; ; ox, w = w→link) {
    for (ox, a = w→arcs; a; ox, a = a→next) {
      oox, u = a→tip→rep;
      if (u ≡ v) continue;
      ox, u→whit = 0;
      if (c1p ∧ (ox, u→srank < wpsr)) ox, u→hit = 1;
    }
    if (w ≡ t) break;
  }
```

This code is used in section 13.

16. ⟨ Say farewell 16 ⟩ ≡

```
fprintf(stderr, "Altogether␣"O"d␣strong␣component"O"s␣and␣"O"d␣weak␣component"O"s;",
    comps, comps ≡ 1 ? "" : "s", wcomps, wcomps ≡ 1 ? "" : "s");
fprintf(stderr, "␣"O"llu+"O"llu␣mems.\n", mems, xmems);
```

This code is used in section 1.

17. Index.

⟨ Adjust $u\neg low$ with respect to its tree child v 11 ⟩ Used in section 9.
⟨ Do the algorithm 9 ⟩ Used in section 1.
⟨ Extend the graph 4 ⟩ Used in section 2.
⟨ Optionally delete arcs 3 ⟩ Used in section 2.
⟨ Prepare to terminate early, and **goto** *t8* 10 ⟩ Used in section 9.
⟨ Print fields used in strong component leaders 7 ⟩ Used in section 6.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Produce a new strong component whose leader is v 12 ⟩ Used in section 9.
⟨ Run through all arcs from strong component v, resetting *whit* 15 ⟩ Used in section 13.
⟨ Run through all arcs from strong component v, setting *whit* and t 14 ⟩ Used in section 13.
⟨ Say farewell 16 ⟩ Used in section 1.
⟨ Subroutines 6, 8 ⟩ Used in section 1.
⟨ Update weak components 13 ⟩ Used in section 12.

TARJAN-STRONG-AND-WEAK