(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), based on my current draft in prefascicle 12a.

I've included all the bells and whistles regarding the output of minimal links between and within strong components. Extra memory references for these features are tallied separately from the *mems* of the basic procedure.

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"
  unsigned long long mems;
  unsigned long long xmems;
  int comps;
  int n;
  Graph *gg;
  ⟨Subroutines 4⟩;
  main(int argc, char *argv[])
  {
    register int p, lowv;
    register Graph *g;
    register Vertex *t, *u, *v, *w, *root, *sink, *settled;
    register Arc *a, *b;
    ⟨Process the command line 2⟩;
    ⟨Do the algorithm 5⟩;
    ⟨Say farewell 11⟩;
  }
```

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);
}
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. I use the fact that GraphBase graphs provide *extra_n* vertices, so that it's OK for me to store something in $g\to vertices + g\to n$, 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 $g\to n$ 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 from y.V    /* FROM */
#define symlink(u)
        ((u) ≡ gg→vertices + n ? "END" : ((u) < gg→vertices + n) ∧ ((u) ≥ gg→vertices) ? (u)→name : "??")
```

```
⟨ Subroutines 4 ⟩ ≡
  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→from) fprintf(stderr, "␣from="O"s", symlink(v→from));
      }
    }
    fprintf(stderr, "\n");
  }
```

See also section 10.

This code is used in section 1.

5. ⟨ Do the algorithm 5 ⟩ ≡

```
   $sent\to low = 0$;
t1: for ($w = g\to vertices$; $w < sent$; $w$++) $o, w\to par = \Lambda$;
   $p = 0$;     /* at this point $w = sent$ */
   $sink = sent, settled = \Lambda$;
t2: if ($w \equiv g\to vertices$) goto done;
   if ($o, (--w)\to par \neq \Lambda$) goto t2;
   $v = w, v\to par = sent, root = v$;
t3: $o, a = v\to arcs$;
   $oo, lowv = v\to low = ++p, v\to link = sent$;
t4: if ($a \equiv \Lambda$) goto t7;
t5: $o, u = a\to tip, a = a\to next$;
t6: if ($o, u\to par \equiv \Lambda$) {
      $oo, u\to par = v, v\to arc = a, v = u$;
      goto t3;
   }
   if ($u \equiv root \wedge p \equiv g\to n$) ⟨ Prepare to terminate early, and goto t8 6 ⟩;
   if ($o, u\to low < lowv$) $oo, lowv = v\to low = u\to low, v\to link = u$;
   goto t4;
t7: $o, u = v\to par$;
   if ($o, v\to link \equiv sent$) goto t8;
   if ($v\to link \neq \Lambda$) printf("␣inner␣"O"s->"O"s\n", $v\to name, v\to link\to name$);
   ⟨ Adjust $u\to low$ with respect to its tree child $v$ 7 ⟩;
   $o, v\to link = sink, sink = v$;
   goto t9;
t8: ⟨ Produce a new strong component whose leader is $v$ 8 ⟩;
t9: if ($u \equiv sent$) goto t2;
   $oo, v = u, a = v\to arc, lowv = v\to low$;
   goto t4;
done: ⟨ Print links between components 9 ⟩;
```

This code is used in section 1.

6. ⟨ Prepare to terminate early, and **goto** *t8* 6 ⟩ ≡

```
   {
      if ($v \neq root$) printf("␣inner␣"O"s->"O"s\n", $v\to name, root\to name$);
      while ($v \neq root$) $oo, v\to link = sink, sink = v, v = v\to par$;
      $o, u = sent, lowv = 1$;
      goto t8;
   }
```

This code is used in section 5.

7. At this point *lowv* is `LOW(`v`)`; it might or might not have been stored in $v\to low$. If $u\to link \neq sent$, step *t6* may have set $u\to link$ to a vertex that's a nontree child of u responsible for $u\to low$.

Three cases arise: If $lowv > u\to low$, we do nothing. If $lowv < u\to low$, we set $u\to low = lowv$; we also set $u\to link = \Lambda$, because this will avoid printing a redundant inner link. (The value of `LOW(`u`)` is inherited from v.)

In the remaining case, $lowv = u\to low$, I thought at first that it was legitimate to set $u\to link = \Lambda$ if $u\to link \neq sent$, reasoning that there was no reason for u to publish an inner arc to $u\to link$ because v already had provided a sufficient inner arc. That was fallacious, because v might have copied u's low pointer, and was relying on it by simply giving an inner link to u. (Consider $1 \longrightarrow 2$, $2 \longrightarrow 1$, $2 \longrightarrow 3$, $3 \longrightarrow 2$, $3 \longrightarrow 1$.)

```
⟨ Adjust u→low with respect to its tree child v 7 ⟩ ≡
  if (o, lowv < u→low) oo, u→low = lowv, u→link = Λ;
```

This code is used in section 5.

8. 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 8 ⟩ ≡
  comps++;
  printf("strong␣component␣"O"s:\n", v→name);
  if (sink→low < lowv) oo, v→rep = v, ox, v→link = settled, settled = v;      /* singleton component */
  else {
    ox, v→link = settled, settled = sink;
    while (o, sink→low ≥ lowv) {
      ox, printf("␣tree␣"O"s->"O"s\n", sink→par→name, sink→name);
      o, sink→rep = v, t = sink;
      o, sink = sink→link;
    }
    o, v→rep = v;
    ox, t→link = v;
  }
```

This code is used in section 5.

9. I've basically copied this from ROGET_COMPONENTS §17.

```
⟨ Print links between components 9 ⟩ ≡
  for (v = g→vertices; v < sent; v++) v→from = Λ;
  for (v = settled; v; ox, v = v→link) {
    oox, u = v→rep, u→from = u;
    for (ox, a = v→arcs; a; ox, a = a→next) {
      oox, w = a→tip→rep;
      if (ox, w→from ≠ u) {
        ox, w→from = u;
        printf("␣link␣"O"s␣to␣"O"s:␣"O"s->"O"s\n", u→name, w→name, v→name, a→tip→name);
      }
    }
  }
```

This code is used in section 5.

10. Here's a subroutine that might be useful when debugging. (For example, I can say '*print_stack*(*sink*)' or '*print_stack*(*settled*)'.)

```
⟨ Subroutines 4 ⟩ +≡
  void print_stack(Vertex *top)
  {
    register Vertex *v;
    for ($v = top$; $v \ge gg\to vertices \land v < gg\to vertices + n$; $v = v\to link$) fprintf(stderr, "␣"O"s", v→name);
    if ($v \ne \Lambda \land v \ne gg\to vertices + n$) fprintf(stderr, "␣(bad␣link!)\n");
    else fprintf(stderr, "\n");
  }
```

11. ⟨ Say farewell 11 ⟩ ≡

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

This code is used in section 1.

12. Index.

TARJAN-STRONG