Important: Before reading BOOK_COMPONENTS, please read or at least skim the program for GB_BOOKS.

1. Bicomponents. This demonstration program computes the biconnected components of GraphBase graphs derived from world literature, using a variant of Hopcroft and Tarjan's algorithm [R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing* **1** (1972), 146–160]. Articulation points and ordinary (connected) components are also obtained as byproducts of the computation.

Two edges belong to the same biconnected component—or "bicomponent" for short—if and only if they are identical or both belong to a simple cycle. This defines an equivalence relation on edges. The bicomponents of a connected graph form a free tree, if we say that two bicomponents are adjacent when they have a common vertex (i.e., when there is a vertex belonging to at least one edge in each of the bicomponents). Such a vertex is called an *articulation point*; there is a unique articulation point between any two adjacent bicomponents. If we choose one bicomponent to be the "root" of the free tree, the other bicomponents can be represented conveniently as lists of vertices, with the articulation point that leads toward the root listed last. This program displays the bicomponents in exactly that way.

2. We permit command-line options in typical UNIX style so that a variety of graphs can be studied: The user can say '`-t`⟨title⟩', '`-n`⟨number⟩', '`-x`⟨number⟩', '`-f`⟨number⟩', '`-l`⟨number⟩', '`-i`⟨number⟩', '`-o`⟨number⟩', and/or '`-s`⟨number⟩' to change the default values of the parameters in the graph generated by *book*(t, n, x, f, l, i, o, s).

When the bicomponents are listed, each character in the book is identified by a two-letter code, as found in the associated data file. An explanation of these codes will appear first if the `-v` or `-V` option is specified. The `-V` option prints a fuller explanation than `-v`; it also shows each character's weighted number of appearances.

The special command-line option `-g`⟨filename⟩ overrides all others. It substitutes an external graph previously saved by *save_graph* for the graphs produced by *book*.

```
#include "gb_graph.h"    /* the GraphBase data structures */
#include "gb_books.h"    /* the book routine */
#include "gb_io.h"    /* the imap_chr routine */
#include "gb_save.h"    /* restore_graph */
⟨ Preprocessor definitions ⟩
⟨ Global variables 7 ⟩
⟨ Subroutines 4 ⟩
main(argc, argv)
    int argc;    /* the number of command-line arguments */
    char *argv[];    /* an array of strings containing those arguments */
{ Graph *g;    /* the graph we will work on */
  register Vertex *v;    /* the current vertex of interest */
  char *t = "anna";    /* the book to use */
  unsigned long n = 0;    /* the desired number of vertices (0 means infinity) */
  unsigned long x = 0;    /* the number of major characters to exclude */
  unsigned long f = 0;    /* the first chapter to include */
  unsigned long l = 0;    /* the last chapter to include (0 means infinity) */
  long i = 1;    /* the weight for appearances in selected chapters */
  long o = 1;    /* the weight for appearances in unselected chapters */
  long s = 0;    /* the random number seed */
  ⟨ Scan the command-line options 3 ⟩;
  if (filename) g = restore_graph(filename);
  else g = book(t, n, x, f, l, i, o, s);
  if (g ≡ Λ) {
    fprintf(stderr, "Sorry, can't create the graph! (error code %ld)\n", panic_code);
    return -1;
  }
  printf("Biconnectivity analysis of %s\n\n", g→id);
  if (verbose) ⟨ Print the cast of selected characters 5 ⟩;
  ⟨ Perform the Hopcroft-Tarjan algorithm on g 12 ⟩;
  return 0;    /* normal exit */
}
```

3. ⟨ Scan the command-line options 3 ⟩ ≡

```
while (--argc) {
  if (strncmp(argv[argc], "-t", 2) ≡ 0) t = argv[argc] + 2;
  else if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) ;
  else if (sscanf(argv[argc], "-x%lu", &x) ≡ 1) ;
  else if (sscanf(argv[argc], "-f%lu", &f) ≡ 1) ;
  else if (sscanf(argv[argc], "-l%lu", &l) ≡ 1) ;
  else if (sscanf(argv[argc], "-i%ld", &i) ≡ 1) ;
  else if (sscanf(argv[argc], "-o%ld", &o) ≡ 1) ;
  else if (sscanf(argv[argc], "-s%ld", &s) ≡ 1) ;
  else if (strcmp(argv[argc], "-v") ≡ 0) verbose = 1;
  else if (strcmp(argv[argc], "-V") ≡ 0) verbose = 2;
  else if (strncmp(argv[argc], "-g", 2) ≡ 0) filename = argv[argc] + 2;
  else {
    fprintf(stderr, "Usage: %s [-ttitle] [-nN] [-xN] [-fN] [-lN] [-iN] [-oN] [-sN] [-v] [-gfoo]\n",
        argv[0]);
    return -2;
  }
}
if (filename) verbose = 0;
```

This code is used in section 2.

4. ⟨ Subroutines 4 ⟩ ≡

```
char *filename = Λ;    /* external graph to be restored */
char code_name[3][3];
char *vertex_name(v, i)    /* return (as a string) the name of vertex v */
    Vertex *v;
    char i;    /* i should be 0, 1, or 2 to avoid clash in code_name array */
{
  if (filename) return v→name;    /* not a book graph */
  code_name[i][0] = imap_chr(v→short_code/36);
  code_name[i][1] = imap_chr(v→short_code % 36);
  return code_name[i];
}
```

This code is used in section 2.

5. ⟨ Print the cast of selected characters 5 ⟩ ≡

```
{
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    if (verbose ≡ 1) printf("%s=%s\n", vertex_name(v, 0), v→name);
    else printf("%s=%s, %s [weight %ld]\n", vertex_name(v, 0), v→name, v→desc,
        i * v→in_count + o * v→out_count);
  }
  printf("\n");
}
```

This code is used in section 2.

6. The algorithm. The Hopcroft-Tarjan algorithm is inherently recursive. We will implement the recursion explicitly via linked lists, instead of using C's runtime stack, because some computer systems bog down in the presence of deeply nested recursion.

Each vertex goes through three stages during the algorithm. First it is "unseen"; then it is "active"; finally it becomes "settled," when it has been assigned to a bicomponent.

The data structures that represent the current state of the algorithm are implemented by using five of the utility fields in each vertex: *rank*, *parent*, *untagged*, *link*, and *min*. We will consider each of these in turn.

7. First is the integer *rank* field, which is zero when a vertex is unseen. As soon as the vertex is first examined, it becomes active and its *rank* becomes and remains nonzero. Indeed, the kth vertex to become active will receive rank k.

It's convenient to think of the Hopcroft-Tarjan algorithm as a simple adventure game in which we want to explore all the rooms of a cave. Passageways between the rooms allow two-way travel. When we come into a room for the first time, we assign a new number to that room; this is its rank. Later on we might happen to come into the same room again, and we will notice that it has nonzero rank. Then we'll be able to make a quick exit, saying "we've already been here." (The extra complexities of computer games, like dragons that might need to be vanquished, do not arise.)

```
#define rank z.I /* the rank of a vertex is stored in utility field z */
⟨ Global variables 7 ⟩ ≡
  long nn; /* the number of vertices that have been seen */
```

See also sections 8, 10, 13, and 20.

This code is used in section 2.

8. The active vertices will always form an oriented tree, whose arcs are a subset of the arcs in the original graph. A tree arc from u to v will be represented by $v\rightarrow parent \equiv u$. Every active vertex has a parent, which is usually another active vertex; the only exception is the root of the tree, whose *parent* is a dummy vertex called *dummy*. The dummy vertex has rank zero.

In the cave analogy, the "parent" of room v is the room we were in immediately before entering v the first time. By following parent pointers, we will be able to leave the cave whenever we want.

```
#define parent y.V /* the parent of a vertex is stored in utility field y */
⟨ Global variables 7 ⟩ +≡
  Vertex dummy; /* imaginary parent of the root vertex */
```

9. All edges in the original undirected graph are explored systematically during a depth-first search. Whenever we look at an edge, we tag it so that we won't need to explore it again. In a cave, for example, we might mark each passageway between rooms once we've tried to go through it.

In a GraphBase graph, undirected edges are represented as a pair of directed arcs. Each of these arcs will be examined and eventually tagged.

The algorithm doesn't actually place a tag on its **Arc** records; instead, each vertex v has a pointer $v\rightarrow untagged$ that leads to all hitherto-unexplored arcs from v. The arcs of the list that appear between $v\rightarrow arcs$ and $v\rightarrow untagged$ are the ones already examined.

```
#define untagged x.A /* the untagged field points to an Arc record, or Λ */
```

10. The algorithm maintains a special stack, the *active_stack*, which contains all the currently active vertices. Each vertex has a *link* field that points to the vertex that is next lower on its stack, or to Λ if the vertex is at the bottom. The vertices on *active_stack* always appear in increasing order of rank from bottom to top.

```
#define link w.V /* the link field of a vertex occupies utility field w */
⟨ Global variables 7 ⟩ +≡
  Vertex *active_stack; /* the top of the stack of active vertices */
```

11. Finally there's a *min* field, which is the tricky part that makes everything work. If vertex v is unseen or settled, its *min* field is irrelevant. Otherwise v→*min* points to the active vertex u of smallest rank having the following property: There is a directed path from v to u consisting of zero or more mature tree arcs followed by a single non-tree arc.

What is a tree arc, you ask. And what is a mature arc? Good questions. At the moment when arcs of the graph are tagged, we classify them either as tree arcs (if they correspond to a new *parent* link in the tree of active nodes) or non-tree arcs (otherwise). The tree arcs therefore correspond to passageways that have led us to new territory. A tree arc becomes mature when it is no longer on the path from the root to the current vertex being explored. We also say that a vertex becomes mature when it is no longer on that path. All arcs from a mature vertex have been tagged.

We said before that every vertex is initially unseen, then active, and finally settled. With our new definitions, we see further that every arc starts out untagged, then it becomes either a non-tree arc or a tree arc. In the latter case, the arc begins as an immature tree arc and eventually matures.

The dummy vertex is considered to be active, and we assume that there is a non-tree arc from the root vertex back to *dummy*. Thus there is a non-tree arc from v to v→*parent* for all v, and v→*min* will always point to a vertex whose rank is less than or equal to v→*parent*→*rank*. It will turn out that v→*min* is always an ancestor of v.

Just believe these definitions, for now. All will become clear soon.

```
#define min  v.V    /* the min field of a vertex occupies utility field v */
```

12. Depth-first search explores a graph by systematically visiting all vertices and seeing what they can lead to. In the Hopcroft-Tarjan algorithm, as we have said, the active vertices form an oriented tree. One of these vertices is called the current vertex.

If the current vertex still has an arc that hasn't been tagged, we tag one such arc and there are two cases: Either the arc leads to an unseen vertex, or it doesn't. If it does, the arc becomes a tree arc; the previously unseen vertex becomes active, and it becomes the new current vertex. On the other hand if the arc leads to a vertex that has already been seen, the arc becomes a non-tree arc and the current vertex doesn't change.

Finally there will come a time when the current vertex v has no untagged arcs. At this point, the algorithm might decide that v and all its descendants form a bicomponent, together with $v\rightarrow parent$. Indeed, this condition turns out to be true if and only if $v\rightarrow min \equiv v\rightarrow parent$; a proof appears below. If so, v and all its descendants become settled, and they leave the tree. If not, the tree arc from v's parent u to v becomes mature, so the value of $v\rightarrow min$ is used to update the value of $u\rightarrow min$. In both cases, v becomes mature and the new current vertex will be the parent of v. Notice that only the value of $u\rightarrow min$ needs to be updated, when the arc from u to v matures; all other values $w\rightarrow min$ stay the same, because a newly mature arc has no mature predecessors.

The cave analogy helps to clarify the situation: Suppose we enter room v from room u. If there's no way out of the subcave starting at v unless we come back through u, and if we can get to u from all of v's descendants without passing through v, then room v and its descendants will become a bicomponent together with u. Once such a bicomponent is identified, we close it off and don't explore that subcave any further.

If v is the root of the tree, it always has $v\rightarrow min \equiv dummy$, so it will always define a new bicomponent at the moment it matures. Then the depth-first search will terminate, since v has no real parent. But the Hopcroft-Tarjan algorithm will press on, trying to find a vertex u that is still unseen. If such a vertex exists, a new depth-first search will begin with u as the root. This process keeps on going until at last all vertices are happily settled.

The beauty of this algorithm is that it all works very efficiently when we organize it as follows:

```
⟨ Perform the Hopcroft-Tarjan algorithm on g 12 ⟩ ≡
  ⟨ Make all vertices unseen and all arcs untagged 14 ⟩;
  for (vv = g→vertices; vv < g→vertices + g→n; vv++)
    if (vv→rank ≡ 0)      /* vv is still unseen */
      ⟨ Perform a depth-first search with vv as the root, finding the bicomponents of all unseen vertices
          reachable from vv 15 ⟩;
```

This code is used in section 2.

13. ⟨ Global variables 7 ⟩ +≡

```
Vertex *vv;      /* sweeps over all vertices, making sure none is left unseen */
```

14. It's easy to get the data structures started, according to the conventions stipulated above.

```
⟨ Make all vertices unseen and all arcs untagged 14 ⟩ ≡
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    v→rank = 0;
    v→untagged = v→arcs;
  }
  nn = 0;
  active_stack = Λ;
  dummy.rank = 0;
```

This code is used in section 12.

15. The task of starting a depth-first search isn't too bad either. Throughout this part of the algorithm, variable v will point to the current vertex.

```
⟨ Perform a depth-first search with vv as the root, finding the bicomponents of all unseen vertices reachable
      from vv 15 ⟩ ≡
  {
    v = vv;
    v→parent = &dummy;
    ⟨ Make vertex v active 16 ⟩;
    do ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and
         calling it v 17 ⟩ while (v ≠ &dummy);
  }
```

This code is used in section 12.

16.

```
⟨ Make vertex v active 16 ⟩ ≡
  v→rank = ++nn;
  v→link = active_stack;
  active_stack = v;
  v→min = v→parent;
```

This code is used in sections 15 and 17.

17. Now things get interesting. But we're just doing what any well-organized spelunker would do when calmly exploring a cave. There are three main cases, depending on whether the current vertex stays where it is, moves to a new child, or backtracks to a parent.

```
⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 17 ⟩ ≡
  { register Vertex *u;    /* a vertex adjacent to v */
    register Arc *a = v→untagged;    /* v's first remaining untagged arc, if any */
    if (a) {
      u = a→tip;
      v→untagged = a→next;    /* tag the arc from v to u */
      if (u→rank) {    /* we've seen u already */
        if (u→rank < v→min→rank) v→min = u;    /* non-tree arc, just update v→min */
      } else {    /* u is presently unseen */
        u→parent = v;    /* the arc from v to u is a new tree arc */
        v = u;    /* u will now be the current vertex */
        ⟨ Make vertex v active 16 ⟩;
      }
    } else {    /* all arcs from v are tagged, so v matures */
      u = v→parent;    /* prepare to backtrack in the tree */
      if (v→min ≡ u) ⟨ Remove v and all its successors on the active stack from the tree, and report them
          as a bicomponent of the graph together with u 19 ⟩
      else    /* the arc from u to v has just matured, making v→min visible from u */
       if (v→min→rank < u→min→rank) u→min = v→min;
      v = u;    /* the former parent of v is the new current vertex v */
    }
  }
```

This code is used in section 15.

18. The elements of the active stack are always in order by rank, and all children of a vertex v in the tree have rank higher than v. The Hopcroft-Tarjan algorithm relies on a converse property: *All active nodes whose rank exceeds that of the current vertex v are descendants of v.* (This property holds because the algorithm has constructed the tree by assigning ranks in preorder, "the order of succession to the throne." First come v's firstborn and descendants, then the nextborn, and so on.) Therefore the descendants of the current vertex always appear consecutively at the top of the stack.

Suppose v is a mature, active vertex with $v\rightarrow min \equiv v\rightarrow parent$, and let $u = v\rightarrow parent$. We want to prove that v and its descendants, together with u and all edges between these vertices, form a biconnected graph. Call this subgraph H. The parent links define a subtree of H, rooted at u, and v is the only vertex having u as a parent (because all other vertices are descendants of v). Let x be any vertex of H different from u and v. Then there is a path from x to $x\rightarrow min$ that does not touch $x\rightarrow parent$, and $x\rightarrow min$ is a proper ancestor of $x\rightarrow parent$. This property is sufficient to establish the biconnectedness of H. (A proof appears at the conclusion of this program.) Moreover, we cannot add any more vertices to H without losing biconnectivity. If w is another vertex, either w has been output already as a non-articulation point of a previous biconnected component, or we can prove that there is no path from w to v that avoids the vertex u.

19. Therefore we are justified in settling v and its active descendants now. Removing them from the tree of active vertices does not remove any vertex from which there is a path to a vertex of rank less than $u\rightarrow rank$. Hence their removal does not affect the validity of the $w\rightarrow min$ value for any vertex w that remains active.

A slight technicality arises with respect to whether or not the parent of v, vertex u, is part of the present bicomponent. When u is the dummy vertex, we have already printed the final bicomponent of a connected component of the original graph, unless v was an isolated vertex. Otherwise u is an articulation point that will occur in subsequent bicomponents, unless the new bicomponent is the final bicomponent of a connected component. (This aspect of the algorithm is probably its most subtle point; consideration of an example or two should clarify everything.)

We print out enough information for a reader to verify the biconnectedness of the claimed component easily.

```
⟨ Remove v and all its successors on the active stack from the tree, and report them as a bicomponent of
      the graph together with u 19 ⟩ ≡
  if (u ≡ &dummy) {      /* active_stack contains just v */
    if (artic_pt) printf("␣and␣%s␣(this␣ends␣a␣connected␣component␣of␣the␣graph)\n",
          vertex_name(artic_pt, 0));
    else printf("Isolated␣vertex␣%s\n", vertex_name(v, 0));
    active_stack = artic_pt = Λ;
  } else { register Vertex *t;      /* runs through the vertices of the new bicomponent */
    if (artic_pt) printf("␣and␣articulation␣point␣%s\n", vertex_name(artic_pt, 0));
    t = active_stack;
    active_stack = v→link;
    printf("Bicomponent␣%s", vertex_name(v, 0));
    if (t ≡ v) putchar('\n');      /* single vertex */
    else {
      printf("␣also␣includes:\n");
      while (t ≠ v) {
        printf("␣%s␣(from␣%s;␣..to␣%s)\n", vertex_name(t, 0), vertex_name(t→parent, 1),
              vertex_name(t→min, 2));
        t = t→link;
      }
    }
    artic_pt = u;      /* the printout will be finished later */
  }
```

This code is used in section 17.

20. Like all global variables, *artic_pt* is initially zero (Λ).

⟨ Global variables 7 ⟩ +≡

```
Vertex *artic_pt;    /* articulation point to be printed if the current bicomponent isn't the last in its
    connected component */
```

21. Proofs. The program is done, but we still should prove that it works. First we want to clarify the informal definition by verifying that the cycle relation between edges, as stated in the introduction, is indeed an equivalence relation.

Suppose u — v and w — x are edges of a simple cycle C, while w — x and y — z are edges of a simple cycle D. We want to show that there is a simple cycle containing the edges u — v and y — z. There are vertices $a, b \in C$ such that a —* y — z —* b is a subpath of D containing no other vertices of C besides a and b. Join this subpath to the subpath in C that runs from b to a through the edge u — v.

Therefore the stated relation between edges is transitive, and it is an equivalence relation. A graph is biconnected if it contains a single vertex, or if each of its vertices is adjacent to at least one other vertex and any two edges are equivalent.

22. Next we prove the well-known fact that a graph is biconnected if and only if it is connected and, for any three distinct vertices x, y, z, it contains a path from x to y that does not touch z. Call the latter condition property P.

Suppose G is biconnected, and let x, y be distinct vertices of G. Then there exist edges u — x and v — y, which are either identical (hence x and y are adjacent) or part of a simple cycle (hence there are two paths from x to y, having no other vertices in common). Thus G has property P.

Suppose, conversely, that G has property P, and let u — v, w — x be distinct edges of G. We want to show that these edges belong to some simple cycle. The proof is by induction on $k = \min\bigl(d(u,w), d(u,x), d(v,w), d(v,x)\bigr)$, where d denotes distance. If $k = 0$, property P gives the result directly. If $k > 0$, we can assume by symmetry that $k = d(u,w)$; so there's a vertex y with u — y and $d(y,w) = k - 1$. And we have u — v equivalent to u — y by property P, u — y equivalent to w — x by induction, hence u — v is equivalent to w — x by transitivity.

23. Finally, we prove that G satisfies property P if it has the following properties: (1) There are two distinguished vertices u and v. (2) Some of the edges of G form a subtree rooted at u, and v is the only vertex whose parent in this tree is u. (3) Every vertex x other than u or v has a path to its grandparent that does not go through its parent.

If property P doesn't hold, there are distinct vertices x, y, z such that every path from x to y goes through z. In particular, z must be between x and y in the unique path π that joins them in the subtree. It follows that $z \neq u$ is the parent of some node z' in that path; hence $z' \neq u$ and $z' \neq v$. But we can avoid z by going from z' to the grandparent of z', which is also part of path π unless z is also the parent of another node z'' in π. In the latter case, however, we can avoid z by going from z' to the grandparent of z' and from there to z'', since z' and z'' have the same grandparent.

24. Index. We close with a list that shows where the identifiers of this program are defined and used.

a: 17.
active_stack: 10, 14, 16, 19.
Arc: 9, 17.
arcs: 9, 14.
argc: 2, 3.
argv: 2, 3.
artic_pt: 19, 20.
book: 2, 4.
code_name: 4.
desc: 5.
dummy: 8, 11, 12, 14, 15, 19.
f: 2.
filename: 2, 3, 4.
fprintf: 2, 3.
g: 2.
Graph: 2.
Hopcroft, John Edward: 1.
i: 2, 4.
id: 2.
imap_chr: 2, 4.
in_count: 5.
l: 2.
link: 6, 10, 16, 19.
main: 2.
min: 6, 11, 12, 16, 17, 18, 19.
n: 2.
name: 4, 5.
next: 17.
nn: 7, 14, 16.
o: 2.
out_count: 5.
panic_code: 2.
parent: 6, 8, 11, 12, 15, 16, 17, 18, 19.
printf: 2, 5, 19.
putchar: 19.
rank: 6, 7, 11, 12, 14, 16, 17, 19.
restore_graph: 2.
s: 2.
save_graph: 2.
short_code: 4.
sscanf: 3.
stderr: 2, 3.
strcmp: 3.
strncmp: 3.
t: 2, 19.
Tarjan, Robert Endre: 1.
tip: 17.
u: 17.
UNIX dependencies: 2, 3.
untagged: 6, 9, 14, 17.
v: 2, 4.
verbose: 2, 3, 5.
Vertex: 2, 4, 8, 10, 13, 17, 19, 20.
vertex_name: 4, 5, 19.
vertices: 5, 12, 14.
vv: 12, 13, 15.
x: 2.

⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 17 ⟩
Used in section 15.
⟨ Global variables 7, 8, 10, 13, 20 ⟩ Used in section 2.
⟨ Make all vertices unseen and all arcs untagged 14 ⟩ Used in section 12.
⟨ Make vertex v active 16 ⟩ Used in sections 15 and 17.
⟨ Perform a depth-first search with vv as the root, finding the bicomponents of all unseen vertices reachable from vv 15 ⟩ Used in section 12.
⟨ Perform the Hopcroft-Tarjan algorithm on g 12 ⟩ Used in section 2.
⟨ Print the cast of selected characters 5 ⟩ Used in section 2.
⟨ Remove v and all its successors on the active stack from the tree, and report them as a bicomponent of the graph together with u 19 ⟩ Used in section 17.
⟨ Scan the command-line options 3 ⟩ Used in section 2.
⟨ Subroutines 4 ⟩ Used in section 2.

August 4, 2025 at 09:09

BOOK_COMPONENTS

	Section	Page
Bicomponents	1	1
The algorithm	6	4
Proofs	21	10
Index	24	11

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