Important: Before reading ROGET_COMPONENTS, please read or at least skim the program for GB_ROGET.

**1. Strong components.** This demonstration program computes the strong components of GraphBase graphs derived from Roget's Thesaurus, using a variant of Tarjan's algorithm [R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing* **1** (1972), 146–160]. We also determine the relationships between strong components.

Two vertices belong to the same strong component if and only if they are reachable from each other via directed paths.

We will print the strong components in "reverse topological order"; that is, if $v$ is reachable from $u$ but $u$ is not reachable from $v$, the strong component containing $v$ will be listed before the strong component containing $u$.

Vertices from the *roget* graph are identified both by name and by category number.

```
#define specs(v) (filename ? v - g→vertices + 1L : v→cat_no), v→name
        /* category number and category name */
```

**2.** We permit command-line options in UNIX style so that a variety of graphs can be studied: The user can say '-n⟨number⟩', '-d⟨number⟩', '-p⟨number⟩', and/or '-s⟨number⟩' to change the default values of the parameters in the graph *roget*$(n, d, p, s)$. Or '-g⟨filename⟩' to change the graph itself.

```
#include "gb_graph.h"   /* the GraphBase data structures */
#include "gb_roget.h"   /* the roget routine */
#include "gb_save.h"    /* restore_graph */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 5 ⟩
  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 */
    unsigned long n = 0;   /* the desired number of vertices (0 means infinity) */
    unsigned long d = 0;   /* the minimum distance between categories in arcs */
    unsigned long p = 0;   /* 65536 times the probability of rejecting an arc */
    long s = 0;   /* the random number seed */
    char *filename = Λ;   /* external graph substituted for roget */
    ⟨ Scan the command-line options 3 ⟩;
    g = (filename ? restore_graph(filename) : roget(n, d, p, s));
    if (g ≡ Λ) {
      fprintf(stderr, "Sorry,␣can't␣create␣the␣graph!␣(error␣code␣%ld)\n", panic_code);
      return −1;
    }
    printf("Reachability␣analysis␣of␣%s\n\n", g→id);
    ⟨ Perform Tarjan's algorithm on g 10 ⟩;
    return 0;   /* normal exit */
  }
```

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

```
while (--argc) {
  if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) ;
  else if (sscanf(argv[argc], "-d%lu", &d) ≡ 1) ;
  else if (sscanf(argv[argc], "-p%lu", &p) ≡ 1) ;
  else if (sscanf(argv[argc], "-s%ld", &s) ≡ 1) ;
  else if (strncmp(argv[argc], "-g", 2) ≡ 0) filename = argv[argc] + 2;
  else {
    fprintf(stderr, "Usage:␣%s␣[-nN][-dN][-pN][-sN][-gfoo]\n", argv[0]);
    return −2;
  }
}
```

This code is used in section 2.

**4.** Tarjan's 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 strong component.

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.

**5.** 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 $k$th vertex to become active will receive rank $k$. When a vertex finally becomes settled, its rank is reset to infinity.

It's convenient to think of Tarjan's algorithm as a simple adventure game in which we want to explore all the rooms of a cave. Passageways between the rooms allow one-way travel only. 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 enter 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 5 ⟩ ≡

```
long nn;   /* the number of vertices that have been seen */
```

See also sections 8 and 11.

This code is used in section 2.

**6.** 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 $v$→*parent* ≡ $u$. Every active vertex has a parent, which is usually another active vertex; the only exception is the root of the tree, whose *parent* is Λ.

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.

As soon as a vertex becomes settled, its *parent* field changes significance. Then $v$→*parent* is set equal to the unique representative of the strong component containing vertex $v$. Thus two settled vertices will belong to the same strong component if and only if they have the same *parent*.

```
#define parent y.V   /* the parent of a vertex is stored in utility field y */
```

**7.** All arcs in the original directed graph are explored systematically during a depth-first search. Whenever we look at an arc, 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.

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

**#define** *untagged* $x.A$ /* the *untagged* field points to an **Arc** record, or $\Lambda$ */

**8.** The algorithm maintains two special stacks: *active_stack* contains all the currently active vertices, and *settled_stack* contains all the currently settled vertices. Each vertex has a *link* field that points to the vertex that is next lower on its stack, or to $\Lambda$ 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 5 ⟩ +≡
  **Vertex** **active_stack*; /* the top of the stack of active vertices */
  **Vertex** **settled_stack*; /* the top of the stack of settled vertices */

**9.** 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 $v\to$*min* points to the active vertex $u$ of smallest rank having the following property: Either $u \equiv v$ or 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). 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.

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

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

**10.** Depth-first search explores a graph by systematically visiting all vertices and seeing what they can lead to. In Tarjan's 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 strong component. Indeed, this condition turns out to be true if and only if $v\to min \equiv v$; 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 $v\to min$ is used to update the value of $u\to min$. In both cases, $v$ becomes mature and the new current vertex will be the parent of $v$. Notice that only the value of $u\to min$ needs to be updated, when the arc from $u$ to $v$ matures; all other values $w\to min$ stay the same, because a newly mature arc has no mature predecessors.

The cave analogy helps to clarify the situation: If there's no way out of the subcave starting at $v$ unless we come back through $v$ itself, and if we can get back to $v$ from all its descendants, then room $v$ and its descendants will become a strong component. Once such a strong component 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 $v\to min \equiv v$, so it will always define a new strong component at the moment it matures. Then the depth-first search will terminate, since $v$ has no parent. But Tarjan's 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 Tarjan's algorithm on g 10 ⟩ ≡
  ⟨ Make all vertices unseen and all arcs untagged 12 ⟩;
  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 strong components of all unseen
          vertices reachable from vv 13 ⟩;
  ⟨ Print out one representative of each arc that runs between strong components 17 ⟩;
```

This code is used in section 2.

**11.**
```
⟨ Global variables 5 ⟩ +≡
  Vertex *vv;      /* sweeps over all vertices, making sure none is left unseen */
```

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

```
⟨ Make all vertices unseen and all arcs untagged 12 ⟩ ≡
  for (v = g→vertices + g→n − 1; v ≥ g→vertices; v−−) {
    v→rank = 0;
    v→untagged = v→arcs;
  }
  nn = 0;
  active_stack = settled_stack = Λ;
```

This code is used in section 10.

**13.** 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 strong components of all unseen vertices reachable from $vv$ 13 ⟩ ≡

```
{
  v = vv;
  v→parent = Λ;
  ⟨ Make vertex v active 14 ⟩;
  do ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and
        calling it v 15 ⟩ while (v ≠ Λ);
}
```

This code is used in section 10.

**14.** ⟨ Make vertex $v$ active 14 ⟩ ≡

```
v→rank = ++nn;
v→link = active_stack;
active_stack = v;
v→min = v;
```

This code is used in sections 13 and 15.

**15.** 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 $v$ 15 ⟩ ≡

```
{ 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 14 ⟩;
    }
  } else {    /* all arcs from v are tagged, so v matures */
    u = v→parent;    /* prepare to backtrack in the tree */
    if (v→min ≡ v) ⟨ Remove v and all its successors on the active stack from the tree, and mark them
          as a strong component of the graph 16 ⟩
    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 13.

**16.** 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 $v$. Tarjan's 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.

Another fundamental property of Tarjan's algorithm is more subtle: *There is always a way to get from any active vertex to the current vertex.* This follows from the fact that all mature active vertices $u$ have *u→min→rank* < *u→rank*. If some active vertex does not lead to the current vertex $v$, let $u$ be the counterexample with smallest rank. Then $u$ isn't an ancestor of $v$, hence $u$ must be mature; hence it leads to the active vertex *u→min*, from which there *is* a path to $v$, contradicting our assumption.

Therefore $v$ and its active descendants are all reachable from each other, and they must belong to the same strong component. Moreover, if *v→min* = $v$, this component can't be made any larger. For there is no arc from any of these vertices to an unseen vertex; all arcs from $v$ and its descendants have already been tagged. And there is no arc from any of these vertices to an active vertex that is below $v$ on the stack; otherwise *v→min* would have smaller rank than $v$. Hence all arcs, if any, that lead from these vertices to some other vertex must lead to settled vertices. And we know from previous steps of the computation that the settled vertices all belong to other strong components.

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 *v→rank*. Hence their removal does not affect the validity of the *u→min* value for any vertex $u$ that remains active.

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

```
#define infinity g→n    /* infinite rank (or close enough) */
```

⟨ Remove $v$ and all its successors on the active stack from the tree, and mark them as a strong component of the graph 16 ⟩ ≡

```
{ register Vertex *t;    /* runs through the vertices of the new strong component */
  t = active_stack;
  active_stack = v→link;
  v→link = settled_stack;
  settled_stack = t;    /* we've moved the top of one stack to the other */
  printf("Strong component `%ld %s'", specs(v));
  if (t ≡ v) putchar('\n');    /* single vertex */
  else {
    printf(" also includes:\n");
    while (t ≠ v) {
      printf(" %ld %s (from %ld %s; ..to %ld %s)\n", specs(t), specs(t→parent), specs(t→min));
      t→rank = infinity;    /* now t is settled */
      t→parent = v;    /* and v represents the new strong component */
      t = t→link;
    }
  }
  v→rank = infinity;    /* v too is settled */
  v→parent = v;    /* and represents its own strong component */
}
```

This code is used in section 15.

**17.** After all the strong components have been found, we can also compute the relations between them, without mentioning any cross-connection more than once. In fact, we built the *settled_stack* precisely so that this task could be done easily without sorting or searching. If only the strong components themselves were of interest, this part of the algorithm wouldn't be necessary.

For this step we use the name *arc_from* for the field we previously called *untagged*. The trick here relies on the fact that all vertices of the same strong component appear together in *settled_stack*.

```
#define arc_from x.V    /* utility field x will now point to a vertex */
```

⟨ Print out one representative of each arc that runs between strong components 17 ⟩ ≡

```
printf("\nLinks␣between␣components:\n");
for (v = settled_stack; v; v = v→link) { register Vertex *u = v→parent;
  register Arc *a;
  u→arc_from = u;
  for (a = v→arcs; a; a = a→next) { register Vertex *w = a→tip→parent;
    if (w→arc_from ≠ u) {
      w→arc_from = u;
      printf("%ld␣%s␣->␣%ld␣%s␣(e.g.,␣%ld␣%s␣->␣%ld␣%s)\n", specs(u), specs(w), specs(v),
          specs(a→tip));
    }
  }
}
```

This code is used in section 10.

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

April 9, 2003 at 12:54

# ROGET_COMPONENTS






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