Important: Before reading FOOTBALL, please read or at least skim the program for GB_GAMES.

1. Introduction. This demonstration program uses graphs constructed by the GB_GAMES module to produce an interactive program called `football`, which finds preposterously long chains of scores to "prove" that one given team might outrank another by a huge margin.

The program prompts you for a starting team. If you simply type ⟨return⟩, it exits; otherwise you should enter a team name (e.g., '`Stanford`') before typing ⟨return⟩.

Then the program prompts you for another team. If you simply type ⟨return⟩ at this point, it will go back and ask for a new starting team; otherwise you should specify another name (e.g., '`Harvard`').

Then the program finds and displays a chain from the starting team to the other one. For example, you might see the following:

```
Oct 06:  Stanford Cardinal 36, Notre Dame Fighting Irish 31 (+5)
Oct 20:  Notre Dame Fighting Irish 29, Miami Hurricanes 20 (+14)
Jan 01:  Miami Hurricanes 46, Texas Longhorns 3 (+57)
Nov 03:  Texas Longhorns 41, Texas Tech Red Raiders 22 (+76)
Nov 17:  Texas Tech Red Raiders 62, Southern Methodist Mustangs 7 (+131)
Sep 08:  Southern Methodist Mustangs 44, Vanderbilt Commodores 7 (+168)
    ⋮
Nov 10:  Cornell Big Red 41, Columbia Lions 0 (+2188)
Sep 15:  Columbia Lions 6, Harvard Crimson 9 (+2185)
```

The chain isn't necessarily optimal; it's just this particular program's best guess. Another chain, which establishes a victory margin of +2279 points, can in fact be produced by modifying this program to search back from Harvard instead of forward from Stanford. Algorithms that find even better chains should be fun to invent.

Actually this program has two variants. If you invoke it by saying simply '`football`', you get chains found by a simple "greedy algorithm." But if you invoke it by saying '`football` ⟨number⟩', assuming UNIX command-line conventions, the program works harder. Higher values of ⟨number⟩ do more calculation and tend to find better chains. For example, the simple greedy algorithm favors Stanford over Harvard by only 781; `football 10` raises this to 1895; the example above corresponds to `football 4000`.

2. Here is the general program layout, as seen by the C compiler:

```
#include "gb_graph.h"     /* the standard GraphBase data structures */
#include "gb_games.h"     /* the routine that sets up the graph of scores */
#include "gb_flip.h"      /* random number generator */
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 10 ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 7 ⟩
  main(argc, argv)
      int argc;     /* the number of command-line arguments */
      char *argv[];     /* an array of strings containing those arguments */
  {
    ⟨ Scan the command-line options 3 ⟩;
    ⟨ Set up the graph 5 ⟩;
    while (1) {
      ⟨ Prompt for starting team and goal team; break if none given 6 ⟩;
      ⟨ Find a chain from start to goal, and print it 9 ⟩;
    }
    return 0;     /* normal exit */
  }
```

3. Let's deal with UNIX-dependent stuff first. The rest of this program should work without change on any operating system.

```
⟨ Scan the command-line options 3 ⟩ ≡
  if (argc ≡ 3 ∧ strcmp(argv[2], "-v") ≡ 0) verbose = argc = 2;     /* secret option */
  if (argc ≡ 1) width = 0;
  else if (argc ≡ 2 ∧ sscanf(argv[1], "%ld", &width) ≡ 1) {
    if (width < 0) width = −width;     /* a UNIX user might have used a hyphen */
  } else {
    fprintf(stderr, "Usage: %s [searchwidth]\n", argv[0]);
    return −2;
  }
```

This code is used in section 2.

4. ⟨ Global variables 4 ⟩ ≡

```
  long width;     /* number of cases examined per stratum */
  Graph *g;     /* the graph containing score information */
  Vertex *u, *v;     /* vertices of current interest */
  Arc *a;     /* arc of current interest */
  Vertex *start, *goal;     /* teams specified by the user */
  long mm;     /* counter used only in verbose mode */
```

See also sections 11, 20, and 29.

This code is used in section 2.

5. An arc from u to v in the graph generated by *games* has a *len* field equal to the number of points scored by u against v. For our purposes we want also a *del* field, which gives the difference between the number of points scored by u and the number of points scored by v in that game.

```
#define del a.I /* del info appears in utility field a of an Arc record */
⟨ Set up the graph 5 ⟩ ≡
  g = games(0_L, 0_L, 0_L, 0_L, 0_L, 0_L, 0_L, 0_L);
    /* this default graph has the data for the entire 1990 season */
  if (g ≡ Λ) {
    fprintf(stderr, "Sorry, can't create the graph! (error code %ld)\n", panic_code);
    return −1;
  }
  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs; a; a = a→next)
      if (a→tip > v) {  /* arc a + 1 is the mate of arc a iff a→tip > v */
        a→del = a→len − (a + 1)→len;
        (a + 1)→del = −a→del;
      }
```

This code is used in section 2.

6. Terminal interaction. While we're getting trivialities out of the way, we might as well take care of the simple dialog that transpires between this program and the user.

```
⟨ Prompt for starting team and goal team; break if none given 6 ⟩ ≡
  putchar('\n');    /* make a blank line for visual punctuation */
restart:    /* if we avoid this label, the break command will be broken */
  if ((start = prompt_for_team("Starting")) ≡ Λ) break;
  if ((goal = prompt_for_team("␣␣␣Other")) ≡ Λ) goto restart;
  if (start ≡ goal) {
    printf("␣(Um,␣please␣give␣me␣the␣names␣of␣two␣DISTINCT␣teams.)\n");
    goto restart;
  }
```

This code is used in section 2.

7. The user must spell team names exactly as they appear in the file `games.dat`. Thus, for example, '`Berkeley`' and '`Cal`' don't work; it has to be '`California`'. Similarly, a person must type '`Pennsylvania`' instead of '`Penn`', '`Nevada-Las Vegas`' instead of '`UNLV`'. A backslash is necessary in '`Texas A\&M`'.

```
⟨ Subroutines 7 ⟩ ≡
  Vertex *prompt_for_team(s)
      char *s;    /* string used in prompt message */
  { register char *q;    /* current position in buffer */
    register Vertex *v;    /* current vertex being examined in sequential search */
    char buffer[30];    /* a line of input */
    while (1) {
      printf("%s␣team:␣", s);
      fflush(stdout);    /* make sure the user sees the prompt */
      fgets(buffer, 30, stdin);
      if (buffer[0] ≡ '\n') return Λ;    /* the user just hit ⟨return⟩ */
      buffer[29] = '\n';
      for (q = buffer; *q ≠ '\n'; q++) ;    /* scan to end of input */
      *q = '\0';
      for (v = g→vertices; v < g→vertices + g→n; v++)
        if (strcmp(buffer, v→name) ≡ 0) return v;    /* aha, we found it */
      printf("␣(Sorry,␣I␣don't␣know␣any␣team␣by␣that␣name.)\n");
      printf("␣(One␣team␣I␣do␣know␣is␣%s...)\n", (g→vertices + gb_unif_rand(g→n))→name);
    }
  }
```

See also section 13.

This code is used in section 2.

8. Greed. This program's primary task is to find the longest possible simple path from *start* to *goal*, using *del* as the length of each arc in the path. This is an NP-complete problem, and the number of possibilities is pretty huge, so the present program is content to use heuristics that are reasonably easy to compute. (Researchers are hereby challenged to come up with better heuristics. Does simulated annealing give good results? How about genetic algorithms?)

Perhaps the first approach that comes to mind is a simple "greedy" approach in which each step takes the largest possible *del* that doesn't prevent us from eventually getting to *goal*. So that's the method we will implement first.

9. ⟨ Find a chain from *start* to *goal*, and print it 9 ⟩ ≡

```
⟨ Initialize the allocation of auxiliary memory 12 ⟩;
if (width ≡ 0) ⟨ Use a simple-minded greedy algorithm to find a chain from start to goal 17 ⟩
else ⟨ Use a stratified heuristic to find a chain from start to goal 19 ⟩;
⟨ Print the solution corresponding to cur_node 15 ⟩;
⟨ Recycle the auxiliary memory used 14 ⟩;
```

This code is used in section 2.

10. We might as well use data structures that are more general than we need, in anticipation of a more complex heuristic that will be implemented later. The set of all possible solutions can be viewed as a backtrack tree in which the branches from each node are the games that can possibly follow that node. We will examine a small part of that gigantic tree.

⟨ Type declarations 10 ⟩ ≡

```
typedef struct node_struct {
  Arc *game;     /* game from the current team to the next team */
  long tot_len;     /* accumulated length from start to here */
  struct node_struct *prev;     /* node that gave us the current team */
  struct node_struct *next;     /* list pointer to node in same stratum (see below) */
} node;
```

This code is used in section 2.

11. ⟨ Global variables 4 ⟩ +≡

```
Area node_storage;     /* working storage for heuristic calculations */
node *next_node;     /* where the next node is slated to go */
node *bad_node;     /* end of current allocation block */
node *cur_node;     /* current node of particular interest */
```

12. ⟨ Initialize the allocation of auxiliary memory 12 ⟩ ≡

```
next_node = bad_node = Λ;
```

This code is used in section 9.

13. ⟨ Subroutines 7 ⟩ +≡

```
node *new_node(x, d)
    node *x;    /* an old node that the new node will call prev */
    long d;    /* incremental change to tot_len */
{
  if (next_node ≡ bad_node) {
    next_node = gb_typed_alloc(1000, node, node_storage);
    if (next_node ≡ Λ) return Λ;    /* we're out of space */
    bad_node = next_node + 1000;
  }
  next_node→prev = x;
  next_node→tot_len = (x ? x→tot_len : 0) + d;
  return next_node++;
}
```

14. ⟨ Recycle the auxiliary memory used 14 ⟩ ≡

```
gb_free(node_storage);
```

This code is used in section 9.

15. When we're done, *cur_node→game→tip* will be the *goal* vertex, and we can get back to the *start* vertex by following *prev* links from *cur_node*. It looks better to print the answers from *start* to *goal*, so maybe we should have changed our algorithm to go the other way.

But let's not worry over trifles. It's easy to change the order of a linked list. The secret is simply to think of the list as a stack, from which we pop all the elements off to another stack; the new stack has the elements in reverse order.

⟨ Print the solution corresponding to *cur_node* 15 ⟩ ≡

```
next_node = Λ;    /* now we'll use next_node as top of temporary stack */
do { register node *t;
  t = cur_node;
  cur_node = t→prev;    /* pop */
  t→prev = next_node;
  next_node = t;    /* push */
} while (cur_node);
for (v = start; v ≠ goal; v = u, next_node = next_node→prev) {
  a = next_node→game;
  u = a→tip;
  ⟨ Print the score of game a between v and u 16 ⟩;
  printf("␣(%+ld)\n", next_node→tot_len);
}
```

This code is used in section 9.

16. ⟨ Print the score of game a between v and u 16 ⟩ ≡

```
{ register long d = a→date;      /* date of the game, 0 means Aug 26 */
  if (d ≤ 5) printf("␣Aug␣%02ld", d + 26);
  else if (d ≤ 35) printf("␣Sep␣%02ld", d − 5);
  else if (d ≤ 66) printf("␣Oct␣%02ld", d − 35);
  else if (d ≤ 96) printf("␣Nov␣%02ld", d − 66);
  else if (d ≤ 127) printf("␣Dec␣%02ld", d − 96);
  else printf("␣Jan␣01");      /* d = 128 */
  printf(":␣%s␣%s␣%ld,␣%s␣%s␣%ld", v→name, v→nickname, a→len, u→name, u→nickname, a→len − a→del);
}
```

This code is used in section 15.

17. We can't just move from v to any adjacent vertex; we can go only to a vertex from which *goal* can be reached without touching v or any other vertex already used on the path from *start*.

Furthermore, if the locally best move from v is directly to *goal*, we don't want to make that move unless it's our last chance; we can probably do better by making the chain longer. Otherwise, for example, a chain between a team and its worst opponent would consist of only a single game.

To keep track of untouchable vertices, we use a utility field called *blocked* in each vertex record. Another utility field, *valid*, will be set to a validation code in each vertex that still leads to the goal.

```
#define blocked  u.I
#define valid  v.V
```

⟨ Use a simple-minded greedy algorithm to find a chain from *start* to *goal* 17 ⟩ ≡

```
{
  for (v = g→vertices; v < g→vertices + g→n; v++) v→blocked = 0, v→valid = Λ;
  cur_node = Λ;
  for (v = start; v ≠ goal; v = cur_node→game→tip) { register long d = −10000;
    register Arc *best_arc;      /* arc that achieves del = d */
    register Arc *last_arc;      /* arc that goes directly to goal */
    v→blocked = 1;
    cur_node = new_node(cur_node, 0_L);
    if (cur_node ≡ Λ) {
      fprintf(stderr, "Oops,␣there␣isn't␣enough␣memory!\n"); return −2;
    }
    ⟨ Set u→valid = v for all u to which v might now move 18 ⟩;
    for (a = v→arcs; a; a = a→next)
      if (a→del > d ∧ a→tip→valid ≡ v)
        if (a→tip ≡ goal) last_arc = a;
        else best_arc = a, d = a→del;
    cur_node→game = (d ≡ −10000 ? last_arc : best_arc);      /* use last_arc as a last resort */
    cur_node→tot_len += cur_node→game→del;
  }
}
```

This code is used in section 9.

18. A standard marking algorithm supplies the final missing link in our algorithm.

```
#define link  w.V
⟨ Set u→valid = v for all u to which v might now move 18 ⟩ ≡
  u = goal;      /* u will be the top of a stack of nodes to be explored */
  u→link = Λ;
  u→valid = v;
  do {
    for (a = u→arcs, u = u→link; a; a = a→next)
      if (a→tip→blocked ≡ 0 ∧ a→tip→valid ≠ v) {
        a→tip→valid = v;      /* mark a→tip reachable from goal */
        a→tip→link = u;
        u = a→tip;      /* push it on the stack, so that its successors will be marked too */
      }
  } while (u);
```

This code is used in section 17.

19. Stratified greed. One approach to better chains is the following algorithm, motivated by similar ideas of Pang Chen [Ph.D. thesis, Stanford University, 1989]: Suppose the nodes of a (possibly huge) backtrack tree are classified into a (fairly small) number of strata, by a function h with the property that $h(\text{child}) < h(\text{parent})$ and $h(\text{goal}) = 0$. Suppose further that we want to find a node x that maximizes a given function $f(x)$, where it is reasonable to believe that $f(\text{child})$ will be relatively large among nodes in a child's stratum only if $f(\text{parent})$ is relatively large in the parent's stratum. Then it makes sense to restrict backtracking to, say, the top w nodes of each stratum, ranked by their f values.

The greedy algorithm already described is a special case of this general approach, with $w = 1$ and with $h(x) = -(\text{length of chain leading to } x)$. The refined algorithm we are about to describe uses a general value of w and a somewhat more relevant stratification function: Given a node x of the backtrack tree for longest paths, corresponding to a path from *start* to a certain vertex $u = u(x)$, we will let $h(x)$ be the number of vertices that lie between u and *goal* (in the sense that the simple path from *start* to u can be extended until it passes through such a vertex and then all the way to *goal*).

Here is the top level of the stratified greedy algorithm. We maintain a linked list of nodes for each stratum, that is, for each possible value of h. The number of nodes required is bounded by w times the number of strata.

```
⟨ Use a stratified heuristic to find a chain from start to goal 19 ⟩ ≡
  {
    ⟨ Make list[0] through list[n − 1] empty 21 ⟩;
    cur_node = Λ;      /* Λ represents the root of the backtrack tree */
    m = g→n − 1;      /* the highest stratum not yet fully explored */
    do {
      ⟨ Place each child x of cur_node into list[h(x)], retaining at most width nodes of maximum tot_len on
          each list 27 ⟩;
      while (list[m] ≡ Λ) ⟨ Decrease m and get ready to explore another list 23 ⟩;
      cur_node = list[m];
      list[m] = cur_node→next;      /* remove a node from highest remaining stratum */
      if (verbose) ⟨ Print "verbose" info about cur_node 24 ⟩;
    } while (m > 0);      /* exactly one node should be in list[0] (see below) */
  }
```

This code is used in section 9.

20. The calculation of $h(x)$ is somewhat delicate, and we will defer it for a moment. But the list manipulation is easy, so we can finish it quickly while it's fresh in our minds.

```
#define MAX_N 120      /* the number of teams in games.dat */
⟨ Global variables 4 ⟩ +≡
  node *list[MAX_N];      /* the best nodes known in given strata */
  long size[MAX_N];      /* the number of elements in a given list */
  long m, h;      /* current lists of interest */
  node *x;      /* a child of cur_node */
```

21. ⟨ Make *list*[0] through *list*[$n-1$] empty 21 ⟩ ≡

```
  for (m = 0; m < g→n; m++) list[m] = Λ, size[m] = 0;
```

This code is used in section 19.

22. The lists are maintained in order by *tot_len*, with the largest *tot_len* value at the end so that we can easily delete the smallest.

When $h = 0$, we retain only one node instead of *width* different nodes, because we are interested in only one solution.

```
⟨ Place node x into list[h], retaining at most width nodes of maximum tot_len 22 ⟩ ≡
  if ((h > 0 ∧ size[h] ≡ width) ∨ (h ≡ 0 ∧ size[0] > 0)) {
    if (x→tot_len ≤ list[h]→tot_len) goto done;      /* drop node x */
    list[h] = list[h]→next;      /* drop one node from list[h] */
  } else size[h]++;
  { register node *p, *q;      /* node in list and its predecessor */
    for (p = list[h], q = Λ; p; q = p, p = p→next) if (x→tot_len ≤ p→tot_len) break;
    x→next = p;
    if (q) q→next = x; else list[h] = x;
  }
done: ;
```

This code is used in section 27.

23. We reverse the list so that large entries will tend to go in first.

```
⟨ Decrease m and get ready to explore another list 23 ⟩ ≡
  { register node *r = Λ, *s = list[--m], *t;
    while (s) t = s→next, s→next = r, r = s, s = t;
    list[m] = r;
    mm = 0;      /* mm is an index for "verbose" printing */
  }
```

This code is used in section 19.

24.

```
⟨ Print "verbose" info about cur_node 24 ⟩ ≡
  {
    cur_node→next = (node *)((++mm ≪ 8) + m);      /* pack an ID for this node */
    printf("[%lu,%lu]=[%lu,%lu]&%s␣(%+ld)\n", m, mm,
        cur_node→prev ? ((unsigned long) cur_node→prev→next) & #ff : 0L,
        cur_node→prev ? ((unsigned long) cur_node→prev→next) ≫ 8 : 0L,
        cur_node→game→tip→name, cur_node→tot_len);
  }
```

This code is used in section 19.

25. Incidentally, it is plausible to conjecture that the stratified algorithm always beats the simple greedy algorithm, but that conjecture is false. For example, the greedy algorithm is able to rank Harvard over Stanford by 1529, while the stratified algorithm achieves only 1527 when *width* = 1. On the other hand, the greedy algorithm often fails miserably; when comparing two Ivy League teams, it doesn't find a way to break out of the Ivy and Patriot Leagues.

26. Bicomponents revisited. How difficult is it to compute the function h? Given a connected graph G with two distinguished vertices u and v, we want to count the number of vertices that might appear on a simple path from u to v. (This is *not* the same as the number of vertices reachable from both u and v. For example, consider a "claw" graph with four vertices $\{u, v, w, x\}$ and with edges only from x to the other three vertices; in this graph w is reachable from u and v, but it is not on any simple path between them.)

The best way to solve this problem is probably to compute the bicomponents of G, or least to compute some of them. Another demo program, BOOK_COMPONENTS, explains the relevant theory in some detail, and we will assume familiarity with that algorithm in the present discussion.

Let us imagine extending G to a slightly larger graph G^+ by adding a dummy vertex o that is adjacent only to v. Suppose we determine the bicomponents of G^+ by depth-first search starting at o. These bicomponents form a tree rooted at the bicomponent that contains just o and v. The number of vertices on paths between u and v, not counting v itself, is then the number of vertices in the bicomponent containing u and in any other bicomponents between that one and the root.

Strictly speaking, each articulation point belongs to two or more bicomponents. But we will assign each articulation point to its bicomponent that is nearest the root of the tree; then the vertices of each bicomponent are precisely the vertices output in bursts by the depth-first procedure. The bicomponents we want to enumerate are B_1, B_2, $\ldots$, B_k, where B_1 is the bicomponent containing u and B_{j+1} is the bicomponent containing the articulation point associated with B_j; we stop at B_k when its associated articulation point is v. (Often $k = 1$.)

The "children" of a given graph G are obtained by removing vertex u and by considering paths from u' to v, where u' is a vertex formerly adjacent to u; thus u' is either in B_1 or it is B_1's associated articulation point. Removing u will, in general, split B_1 into a tree of smaller bicomponents, but $B_2, \ldots, B_k$ will be unaffected. The implementation below does not take full advantage of this observation, because the amount of memory required to avoid recomputation would probably be prohibitive.

27. The following program is copied almost verbatim from BOOK_COMPONENTS. Instead of repeating the commentary that appears there, we will mention only the significant differences. One difference is that we start the depth-first search at a definite place, the *goal*.

```
⟨ Place each child x of cur_node into list[h(x)], retaining at most width nodes of maximum tot_len on each
      list 27 ⟩ ≡
  ⟨ Make all vertices unseen and all arcs untagged, except for vertices that have already been used in steps
      leading up to cur_node 28 ⟩;
  ⟨ Perform a depth-first search with goal as the root, finding bicomponents and determining the number of
      vertices accessible between any given vertex and goal 30 ⟩;
  for (a = (cur_node ? cur_node→game→tip : start)→arcs; a; a = a→next)
    if ((u = a→tip)→untagged ≡ Λ) {      /* goal is reachable from u */
      x = new_node(cur_node, a→del);
      if (x ≡ Λ) {
        fprintf(stderr, "Oops, there isn't enough memory!\n"); return −3;
      }
      x→game = a;
      ⟨ Set h to the number of vertices on paths between u and goal 35 ⟩;
      ⟨ Place node x into list[h], retaining at most width nodes of maximum tot_len 22 ⟩;
    }
```

This code is used in section 19.

28. Setting the *rank* field of a vertex to infinity before beginning a depth-first search is tantamount to removing that vertex from the graph, because it tells the algorithm not to look further at such a vertex.

```
#define rank z.I /* when was this vertex first seen? */
#define parent u.V /* who told me about this vertex? */
#define untagged x.A /* what is its first untagged arc? */
#define min v.V /* how low in the tree can we jump from its mature descendants? */
```

⟨ Make all vertices unseen and all arcs untagged, except for vertices that have already been used in steps leading up to *cur_node* 28 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + g→n; v++) {
  v→rank = 0;
  v→untagged = v→arcs;
}
for (x = cur_node; x; x = x→prev) x→game→tip→rank = g→n; /* "infinite" rank (or close enough) */
start→rank = g→n;
nn = 0;
active_stack = settled_stack = Λ;
```

This code is used in section 27.

29. ⟨ Global variables 4 ⟩ +≡

```
Vertex *active_stack; /* the top of the stack of active vertices */
Vertex *settled_stack; /* the top of the stack of bicomponents found */
long nn; /* the number of vertices that have been seen */
Vertex dummy; /* imaginary parent of goal; its rank is zero */
```

30. The *settled_stack* will contain a list of all bicomponents in the opposite order from which they are discovered. This is the order we'll need later for computing the h function in each bicomponent.

⟨ Perform a depth-first search with *goal* as the root, finding bicomponents and determining the number of vertices accessible between any given vertex and *goal* 30 ⟩ ≡

```
{
  v = goal;
  v→parent = &dummy;
  ⟨ Make vertex v active 31 ⟩;
  do ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and
       calling it v 32 ⟩ while (v ≠ &dummy);
  ⟨ Use settled_stack to put the mutual reachability count for each vertex u in u→parent→rank 34 ⟩;
}
```

This code is used in section 27.

31. ⟨ Make vertex v active 31 ⟩ ≡

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

This code is used in sections 30 and 32.

32. ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 32 ⟩ ≡

```
{ 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 31 ⟩;
      }
    } 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 33 ⟩
      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 30.

33. When a bicomponent is found, we reset the *parent* field of each vertex so that, afterwards, two vertices will belong to the same bicomponent if and only if they have the same *parent*. (This trick was not used in BOOK_COMPONENTS, but it does appear in the similar algorithm of ROGET_COMPONENTS.) The new parent, v, will represent that bicomponent in subsequent computation; we put it onto *settled_stack*. We also reset v→*rank* to be the bicomponent's size, plus a constant large enough to keep the algorithm from getting confused. (Vertex u might still have untagged arcs leading into this bicomponent; we need to keep the ranks at least as big as the rank of u→*min*.) Notice that v→*min* is u, the articulation point associated with this bicomponent. Later the *rank* field will contain the sum of all counts between here and the root.

We don't have to do anything when $v \equiv goal$; the trivial root bicomponent always comes out last.

```
⟨ 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 33 ⟩ ≡
  { if (v ≠ goal) { register Vertex *t;     /* runs through the vertices of the new bicomponent */
      long c = 0;     /* the number of vertices removed */
      t = active_stack;
      while (t ≠ v) {
        c++;
        t→parent = v;
        t = t→link;
      }
      active_stack = v→link;
      v→parent = v;
      v→rank = c + g→n;     /* the true component size is c + 1 */
      v→link = settled_stack;
      settled_stack = v;
    }
  }
```

This code is used in section 32.

34. So here's how we sum the ranks. When we get to this step, the *settled* stack contains all bicomponent representatives except *goal* itself.

```
⟨ Use settled_stack to put the mutual reachability count for each vertex u in u→parent→rank 34 ⟩ ≡
  while (settled_stack) {
    v = settled_stack;
    settled_stack = v→link;
    v→rank += v→min→parent→rank + 1 − g→n;
  }     /* note that goal→parent→rank = 0 */
```

This code is used in section 30.

35. And here's the last piece of the puzzle.

```
⟨ Set h to the number of vertices on paths between u and goal 35 ⟩ ≡
  h = u→parent→rank;
```

This code is used in section 27.

36. Index. Finally, here's a list that shows where the identifiers of this program are defined and used.

August 4, 2025 at 10:31

FOOTBALL

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