Important: Before reading GB_ECON, please read or at least skim the programs for GB_GRAPH and GB_IO.

1. Introduction. This GraphBase module contains the *econ* subroutine, which creates a family of directed graphs related to the flow of money between industries. An example of the use of this procedure can be found in the demo program ECON_ORDER.

⟨ `gb_econ.h` 1 ⟩ ≡
  extern Graph $*econ()$;

See also section 5.

2. The subroutine call $econ(n, omit, threshold, seed)$ constructs a directed graph based on the information in `econ.dat`. Each vertex of the graph corresponds to one of 81 sectors of the U.S. economy. The data values come from the year 1985; they were derived from tables published in *Survey of Current Business* **70** (1990), 41–56.

If $omit = threshold = 0$, the directed graph is a "circulation"; that is, each arc has an associated *flow* value, and the sum of arc flows leaving each vertex is equal to the sum of arc flows entering. This sum is called the "total commodity output" for the sector in question. The flow in an arc from sector j to sector k is the amount of the commodity made by sector j that was used by sector k, rounded to millions of dollars at producers' prices. For example, the total commodity output of the sector called `Apparel` is 54031, meaning that the total cost of making all kinds of apparel in 1985 was about 54 billion dollars. There is an arc from `Apparel` to itself with a flow of 9259, meaning that 9.259 billion dollars' worth of apparel went from one group within the apparel industry to another. There also is an arc of flow 44 from `Apparel` to `Household furniture`, indicating that some 44 million dollars' worth of apparel went into the making of household furniture. By looking at all arcs that leave the `Apparel` vertex, you can see where all that new apparel went; by looking at all arcs that enter `Apparel`, you can see what ingredients the apparel industry needed to make it.

One vertex, called `Users`, represents people like you and me, the non-industrial end users of everything. The arc from `Apparel` to `Users` has flow 42172; this is the "total final demand" for apparel, the amount that didn't flow into other sectors of the economy before it reached people like us. The arc from `Users` to `Apparel` has flow 19409, which is called the "value added" by users; it represents wages and salaries paid to support the manufacturing process. The sum of total final demand over all sectors, which also equals the sum of value added over all sectors, is conventionally called the Gross National Product (GNP). In 1985 the GNP was 3999362, nearly 4 trillion dollars, according to `econ.dat`. (The sum of all arc flows coming out of all vertices was 7198680; this sum overestimates the total economic activity, because it counts some items more than once—statistics are recorded whenever an item passes a statistics gatherer. Economists try to adjust the data so that they avoid double-counting as much as possible.)

Speaking of economists, there is another special vertex called `Adjustments`, included by economists so that GNP is measured more accurately. This vertex takes account of such things as changes in the value of inventories, and imported materials that cannot be obtained within the U.S., as well as work done for the government and for foreign concerns. In 1985, these adjustments accounted for about 11% of the GNP.

Incidentally, some of the "total final demand" arcs are negative. For example, the arc from `Petroleum and natural gas production` to `Users` has flow −27032. This might seem strange at first, but it makes sense when imports are considered, because crude oil and natural gas go more to other industries than to end users. Total final demand does not mean total user demand.

#define *flow* $a.I$ /* utility field a specifies the flow in an arc */

3. If *omit* $= 1$, the `Users` vertex is omitted from the digraph; in particular, this will eliminate all arcs of negative flow. If *omit* $= 2$, the `Adjustments` vertex is also omitted, thereby leaving 79 sectors with arcs showing inter-industry flow. (The graph is no longer a "circulation," of course, when *omit* > 0.) If `Users` and `Adjustments` are not omitted, `Users` is the last vertex of the graph, and `Adjustments` is next-to-last.

If *threshold* $= 0$, the digraph has an arc for every nonzero *flow*. But if *threshold* > 0, the digraph becomes more sparse; there is then an arc from j to k if and only if the amount of commodity j used by sector k exceeds *threshold*/65536 times the total input of sector k. (The total input figure always includes value added, even if *omit* > 0.) Thus the arcs go to each sector from that sector's main suppliers. When $n = 79$, *omit* $= 2$, and *threshold* $= 0$, the digraph has 4602 arcs out of a possible $79 \times 79 = 6241$. Raising *threshold* to 1 decreases the number of arcs to 4473; raising it to 6000 leaves only 72 arcs. The *len* field in each arc is 1.

The constructed graph will have $\min(n, 81 - omit)$ vertices. If n is less than $81 - omit$, the n vertices will be selected by repeatedly combining related sectors. For example, two of the 81 original sectors are called '`Paper products, except containers`' and '`Paperboard containers and boxes`'; these might be combined into a sector called '`Paper products`'. There is a binary tree with 79 leaves, which describes a fixed hierarchical breakdown of the 79 non-special sectors. This tree is pruned, if necessary, by replacing pairs of leaves by their parent node, which becomes a new leaf; pruning continues until just n leaves remain. Although pruning is a bottom-up process, its effect can also be obtained from the top down if we imagine "growing" the tree, starting out with a whole economy as a single sector and repeatedly subdividing a sector into two parts. For example, if *omit* $= 2$ and $n = 2$, the two sectors will be called `Goods` and `Services`. If $n = 3$, `Goods` might be subdivided into `Natural Resources` and `Manufacturing`; or `Services` might be subdivided into `Indirect Services` and `Direct Services`.

If *seed* $= 0$, the binary tree is pruned in such a way that the n resulting sectors are as equal as possible with respect to total input and output, while respecting the tree structure. If *seed* > 0, the pruning is carried out at random, in such a way that all n-leaf subtrees of the original tree are obtained with approximately equal probability (depending on *seed* in a machine-independent fashion). Any *seed* value from 1 to $2^{31} - 1 = 2147483647$ is permissible.

As usual in GraphBase routines, you can set $n = 0$ to get the default situation where n has its maximum value. For example, either $econ(0,0,0,0)$ or $econ(81,0,0,0)$ produces the full graph; $econ(0,2,0,0)$ or $econ(79,2,0,0)$ produces the full graph except for the two special vertices.

```
#define MAX_N 81 /* maximum number of vertices in constructed graph */
#define NORM_N MAX_N − 2 /* the number of normal SIC sectors */
#define ADJ_SEC MAX_N − 1 /* code number for the Adjustments sector */
```

4. The U.S. Bureau of Economic Analysis and the U.S. Bureau of the Census have assigned code numbers 1–79 to the individual sectors for which statistics are given in `econ.dat`. These sector numbers are traditionally called Standard Industrial Classification (SIC) codes. If for some reason you want to know the SIC codes for all sectors represented by vertex v of a graph generated by *econ*, you can access them via a list of **Arc** nodes starting at the utility field *v*→*SIC_codes*. This list is linked by *next* fields in the usual way, and each SIC code appears in the *len* field; the *tip* field is unused.

The special vertex `Adjustments` is given code number 80; it is actually a composite of six different SIC categories, numbered 80–86 in their published tables.

For example, if $n = 80$ and *omit* $= 1$, each list will have length 1. Hence *v*→*SIC_codes*→*next* will equal Λ for each v, and *v*→*SIC_codes*→*len* will be v's SIC code, a number between 1 and 80.

The special vertex `Users` has no SIC code; it is the only vertex whose *SIC_codes* field will be null in the graph returned by *econ*.

```
#define SIC_codes z.A /* utility field z leads to the SIC codes for a vertex */
```

5. The total output of each sector, which also equals the total input of that sector, is placed in utility field *sector_total* of the corresponding vertex.

```
#define sector_total y.I   /* utility field y holds the total flow in and out */
```

```
⟨ gb_econ.h 1 ⟩ +≡
#define flow a.I   /* definitions of utility fields in the header file */
#define SIC_codes z.A
#define sector_total y.I
```

6. If the *econ* routine encounters a problem, it returns Λ (NULL), after putting a nonzero number into the external variable *panic_code*. This code number identifies the type of failure. Otherwise *econ* returns a pointer to the newly created graph, which will be represented with the data structures explained in GB_GRAPH. (The external variable *panic_code* is itself defined in GB_GRAPH.)

```
#define panic(c) { panic_code = c; gb_trouble_code = 0; return Λ; }
```

7. The C file gb_econ.c has the following overall shape:

```
#include "gb_io.h"   /* we will use the GB_IO routines for input */
#include "gb_flip.h"   /* we will use the GB_FLIP routines for random numbers */
#include "gb_graph.h"   /* and of course we'll use the GB_GRAPH data structures */
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 11 ⟩
  ⟨ Private variables 12 ⟩
  Graph *econ(n, omit, threshold, seed)
      unsigned long n;   /* number of vertices desired */
      unsigned long omit;   /* number of special vertices to omit */
      unsigned long threshold;   /* minimum per-64K-age in arcs leading in */
      long seed;   /* random number seed */
  { ⟨ Local variables 8 ⟩
    gb_init_rand(seed);
    init_area(working_storage);
    ⟨ Check the parameters and adjust them for defaults 9 ⟩;
    ⟨ Set up a graph with n vertices 10 ⟩;
    ⟨ Read econ.dat and note the binary tree structure 14 ⟩;
    ⟨ Determine the n sectors to use in the graph 17 ⟩;
    ⟨ Put the appropriate arcs into the graph 25 ⟩;
    if (gb_close() ≠ 0) panic(late_data_fault);
        /* something's wrong with "econ.dat"; see io_errors */
    gb_free(working_storage);
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);   /* oops, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

8. ⟨ Local variables 8 ⟩ ≡

```
  Graph *new_graph;   /* the graph constructed by econ */
  register long j, k;   /* all-purpose indices */
  Area working_storage;   /* tables needed while econ does its thinking */
```

See also section 13.

This code is used in section 7.

9. ⟨ Check the parameters and adjust them for defaults 9 ⟩ ≡

```
if (omit > 2) omit = 2;
if (n ≡ 0 ∨ n > MAX_N − omit) n = MAX_N − omit;
else if (n + omit < 3) omit = 3 − n;   /* we need at least one normal sector */
if (threshold > 65536) threshold = 65536;
```

This code is used in section 7.

10. ⟨ Set up a graph with n vertices 10 ⟩ ≡

```
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);   /* out of memory before we're even started */
sprintf(new_graph→id, "econ(%lu,%lu,%lu,%ld)", n, omit, threshold, seed);
strcpy(new_graph→util_types, "ZZZZIAIZZZZZZZ");
```

This code is used in section 7.

11. The economic tree. As we read in the data, we construct a sequential list of nodes, each of which represents either a micro-sector of the economy (one of the basic SIC sectors) or a macro-sector (which is the union of two subnodes). In more technical terms, the nodes form an extended binary tree, whose external nodes correspond to micro-sectors and whose internal nodes correspond to macro-sectors. The nodes of the tree appear in preorder. Subsequently we will do a variety of operations on this binary tree, proceeding either top-down (from the beginning of the list to the end) or bottom-up (from the end to the beginning).

Each node is a rather large record, because we will store a complete vector of sector output data in each node.

⟨ Type declarations 11 ⟩ ≡

```
typedef struct node_struct {   /* records for micro- and macro-sectors */
  struct node_struct *rchild;   /* pointer to right child of macro-sector */
  char title[44];   /* "Sector␣name" */
  long table[MAX_N + 1];   /* outputs from this sector */
  unsigned long total;   /* total input to this sector (= total output) */
  long thresh;   /* flow must exceed thresh in arcs to this sector */
  long SIC;   /* SIC code number; initially zero in macro-sectors */
  long tag;   /* 1 if this node will be a vertex in the graph */
  struct node_struct *link;   /* next smallest unexplored sector */
  Arc *SIC_list;   /* first item on list of SIC codes */
} node;
```

This code is used in section 7.

12. When we read the given data in preorder, we'll need a stack to remember what nodes still need to have their *rchild* pointer filled in. (There is a no need for an *lchild* pointer, because the left child always follows its parent immediately in preorder.)

⟨ Private variables 12 ⟩ ≡

```
static node *stack[NORM_N + NORM_N];
static node **stack_ptr;   /* current position in stack */
static node *node_block;   /* array of nodes, specifies the tree in preorder */
static node *node_index[MAX_N + 1];   /* which node has a given SIC code */
```

See also section 26.

This code is used in section 7.

13. ⟨ Local variables 8 ⟩ +≡

```
register node *p, *pl, *pr;   /* current node and its children */
register node *q;   /* register for list manipulation */
```

14. ⟨ Read econ.dat and note the binary tree structure 14 ⟩ ≡

```
node_block = gb_typed_alloc(2 * MAX_N − 3, node, working_storage);
if (gb_trouble_code) panic(no_room + 1);   /* no room to copy the data */
if (gb_open("econ.dat") ≠ 0) panic(early_data_fault);
    /* couldn't open "econ.dat" using GraphBase conventions */
⟨ Read and store the sector names and SIC numbers 15 ⟩;
for (k = 1; k ≤ MAX_N; k++) ⟨ Read and store the output coefficients for sector k 16 ⟩;
```

This code is used in section 7.

15. The first part of `econ.dat` specifies the nodes of the binary tree in preorder. Each line contains a node name followed by a colon, and the colon is followed by the SIC number if that node is a leaf.

The tree is uniquely specified in this way, because of the nature of preorder. (Think of Polish prefix notation, in which a formula like '$+x+xx$' means '$+(x,+(x,x))$'; the parentheses in Polish notation are redundant.)

The two special sector names don't appear in the file; we manufacture them ourselves.

The program here is careful not to clobber itself in the presence of arbitrarily garbled data.

⟨ Read and store the sector names and SIC numbers 15 ⟩ ≡

```
stack_ptr = stack;
for (p = node_block; p < node_block + NORM_N + NORM_N − 1; p++) { register long c;
  gb_string(p→title, ':');
  if (strlen(p→title) > 43) panic(syntax_error);   /* sector name too long */
  if (gb_char() ≠ ':') panic(syntax_error + 1);   /* missing colon */
  p→SIC = c = gb_number(10);
  if (c ≡ 0)   /* macro-sector */
    *stack_ptr++ = p;   /* left child is p + 1, we'll know rchild later */
  else {   /* micro-sector; p + 1 will be somebody's right child */
    node_index[c] = p;
    if (stack_ptr > stack) (*−−stack_ptr)→rchild = p + 1;
  }
  if (gb_char() ≠ '\n') panic(syntax_error + 2);   /* garbage on the line */
  gb_newline();
}
if (stack_ptr ≠ stack) panic(syntax_error + 3);   /* tree malformed */
for (k = NORM_N; k; k−−)
  if (node_index[k] ≡ 0) panic(syntax_error + 4);   /* SIC code not mentioned in the tree */
strcpy(p→title, "Adjustments"); p→SIC = ADJ_SEC; node_index[ADJ_SEC] = p;
strcpy((p + 1)→title, "Users"); node_index[MAX_N] = p + 1;
```

This code is used in section 14.

16. The remaining part of `econ.dat` is an 81×80 matrix in which the kth row contains the outputs of sector k to all sectors except `Users`. Each row consists of a blank line followed by 8 data lines; each data line contains 10 numbers separated by commas. Zeroes are represented by `""` instead of by `"0"`. For example, the data line

`8490,2182,42,467,,,,,,`

follows the initial blank line; it means that sector 1 output 8490 million dollars to itself, \$2182M to sector 2, ..., \$0M to sector 10.

⟨ Read and store the output coefficients for sector k 16 ⟩ ≡

```
{ register long s = 0;   /* row sum */
  register long x;   /* entry read from econ.dat */
  if (gb_char() ≠ '\n') panic(syntax_error + 5);   /* blank line missing between rows */
  gb_newline();
  p = node_index[k];
  for (j = 1; j < MAX_N; j++) {
    p→table[j] = x = gb_number(10); s += x;
    node_index[j]→total += x;
    if ((j % 10) ≡ 0) {
      if (gb_char() ≠ '\n') panic(syntax_error + 6);   /* out of synch in input file */
      gb_newline();
    } else if (gb_char() ≠ ',') panic(syntax_error + 7);   /* missing comma after entry */
  }
  p→table[MAX_N] = s;   /* sum of table[1] through table[80] */
}
```

This code is used in section 14.

17. Growing a subtree. Once all the data appears in *node_block*, we want to extract from it and combine it as specified by parameters n, *omit*, and *seed*. This amalgamation process effectively prunes the tree; it can also be regarded as a procedure that grows a subtree of the full economic tree.

⟨Determine the n sectors to use in the graph 17⟩ ≡
```
{ long l = n + omit − 2;   /* the number of leaves in the desired subtree */
  if (l ≡ NORM_N) ⟨Choose all sectors 18⟩
  else if (seed) ⟨Grow a random subtree with l leaves 21⟩
  else ⟨Grow a subtree with l leaves by subdividing largest sectors first 19⟩;
}
```
This code is used in section 7.

18. The chosen leaves of our subtree are identified by having their *tag* field set to 1.

⟨Choose all sectors 18⟩ ≡
```
for (k = NORM_N; k; k−−) node_index[k]→tag = 1;
```
This code is used in section 17.

19. To grow the l-leaf subtree when $seed = 0$, we first pass over the tree bottom-up to compute the total input (and output) of each macro-sector; then we proceed from the top down to subdivide sectors in decreasing order of their total input. This process provides a good introduction to the bottom-up and top-down tree methods we will be using in several other parts of the program.

The *special* node is used here for two purposes: It is the head of a linked list of unexplored nodes, sorted by decreasing order of their *total* fields; and it appears at the end of that list, because $special\to total = 0$.

⟨Grow a subtree with l leaves by subdividing largest sectors first 19⟩ ≡
```
{ register node *special = node_index[MAX_N];   /* the Users node at the end of node_block */
  for (p = node_index[ADJ_SEC] − 1; p ≥ node_block; p−−)   /* bottom up */
    if (p→rchild) p→total = (p + 1)→total + p→rchild→total;
  special→link = node_block; node_block→link = special;   /* start at the root */
  k = 1;   /* k is the number of nodes we have tagged or put onto the list */
  while (k < l) ⟨If the first node on the list is a leaf, delete it and tag it; otherwise replace it by its two
        children 20⟩;
  for (p = special→link; p ≠ special; p = p→link) p→tag = 1;   /* tag everything on the list */
}
```
This code is used in section 17.

20. ⟨If the first node on the list is a leaf, delete it and tag it; otherwise replace it by its two children 20⟩ ≡
```
{
  p = special→link;   /* remove p, the node with greatest total */
  special→link = p→link;
  if (p→rchild ≡ 0) p→tag = 1;   /* p is a leaf */
  else {
    pl = p + 1; pr = p→rchild;
    for (q = special; q→link→total > pl→total; q = q→link) ;
    pl→link = q→link; q→link = pl;   /* insert left child in its proper place */
    for (q = special; q→link→total > pr→total; q = q→link) ;
    pr→link = q→link; q→link = pr;   /* insert right child in its proper place */
    k++;
  }
}
```
This code is used in section 19.

21. We can obtain a uniformly distributed l-leaf subtree of a given tree by choosing the root when $l = 1$ or by using the following idea when $l > 1$: Suppose the given tree T has subtrees T_0 and T_1. Then it has $T(l)$ subtrees with l leaves, where $T(l) = \sum_k T_0(k)T_1(l-k)$. We choose a random number r between 0 and $T(l)-1$, and we find the smallest m such that $\sum_{k\le m} T_0(k)T_1(l-k) > r$. Then we proceed recursively to compute a random m-leaf subtree of T_0 and a random $(l-m)$-leaf subtree of T_1.

A difficulty arises when $T(l)$ is 2^{31} or more. But then we can replace $T_0(k)$ and $T_1(l-k)$ in the formulas above by $\lceil T_0(k)/d_0\rceil$ and $\lceil T_1(k)/d_1\rceil$, respectively, where d_0 and d_1 are arbitrary constants; this yields smaller values $T(l)$ that define approximately the same distribution of k.

The program here computes the $T(l)$ values bottom-up, then grows a random tree top-down. If node p is not a leaf, its *table*[0] field will be set to the number of leaves below it; and its *table*[l] field will be set to $T(l)$, for $1 \le l \le$ *table*[0].

The data in `econ.dat` is sufficiently simple that most of the $T(l)$ values are less than 2^{31}. We need to scale them down to avoid overflow only at the root node of the tree; this case is handled separately.

We set the *tag* field of a node equal to the number of leaves to be grown in the subtree rooted at that node. This convention is consistent with our previous stipulation that *tag* $= 1$ should characterize the nodes that are chosen to be vertices.

```
⟨Grow a random subtree with l leaves 21⟩ ≡
  {
    node_block→tag = l;
    for (p = node_index[ADJ_SEC] − 1; p > node_block; p−−)    /* bottom up, except root */
      if (p→rchild) ⟨Compute the T(l) values for subtree p 22⟩;
    for (p = node_block; p < node_index[ADJ_SEC]; p++)    /* top down, from root */
      if (p→tag > 1) {
        l = p→tag;
        pl = p + 1; pr = p→rchild;
        if (pl→rchild ≡ Λ) {
          pl→tag = 1; pr→tag = l − 1;
        } else if (pr→rchild ≡ Λ) {
          pl→tag = l − 1; pr→tag = 1;
        } else ⟨Stochastically determine the number of leaves to grow in each of p's children 24⟩;
      }
  }
```

This code is used in section 17.

22. Here we are essentially multiplying two generating functions. Suppose $f(z) = \sum_l T(l)z^l$; then we are computing $f_p(z) = z + f_{pl}(z)f_{pr}(z)$.

⟨ Compute the $T(l)$ values for subtree p 22 ⟩ ≡

```
{
  pl = p + 1; pr = p→rchild;
  p→table[1] = p→table[2] = 1;    /* T(1) and T(2) are always 1 */
  if (pl→rchild ≡ 0) {    /* left child is a leaf */
    if (pr→rchild ≡ 0) p→table[0] = 2;    /* and so is the right child */
    else {    /* no, it isn't */
      for (k = 2; k ≤ pr→table[0]; k++) p→table[1 + k] = pr→table[k];
      p→table[0] = pr→table[0] + 1;
    }
  } else if (pr→rchild ≡ 0) {    /* right child is a leaf */
    for (k = 2; k ≤ pl→table[0]; k++) p→table[1 + k] = pl→table[k];
    p→table[0] = pl→table[0] + 1;
  } else {    /* neither child is a leaf */
    ⟨ Set p→table[2], p→table[3], ... to convolution of pl and pr table entries 23 ⟩;
    p→table[0] = pl→table[0] + pr→table[0];
  }
}
```

This code is used in section 21.

23. ⟨ Set $p\to table[2]$, $p\to table[3]$, ... to convolution of pl and pr table entries 23 ⟩ ≡

```
p→table[2] = 0;
for (j = pl→table[0]; j; j−−) { register long t = pl→table[j];
  for (k = pr→table[0]; k; k−−) p→table[j + k] += t * pr→table[k];
}
```

This code is used in section 22.

24. ⟨ Stochastically determine the number of leaves to grow in each of p's children 24 ⟩ ≡

```
{ register long ss, rr;
  j = 0;    /* we will set j = 1 if scaling is necessary at the root */
  if (p ≡ node_block) {
    ss = 0;
    if (l > 29 ∧ l < 67) {
      j = 1;    /* more than 2^31 possibilities exist */
      for (k = (l > pr→table[0] ? l − pr→table[0] : 1); k ≤ pl→table[0] ∧ k < l; k++)
        ss += ((pl→table[k] + #3ff) ≫ 10) * pr→table[l − k];    /* scale with d_0 = 1024, d_1 = 1 */
    } else
      for (k = (l > pr→table[0] ? l − pr→table[0] : 1); k ≤ pl→table[0] ∧ k < l; k++)
        ss += pl→table[k] * pr→table[l − k];
  } else ss = p→table[l];
  rr = gb_unif_rand(ss);
  if (j)
    for (ss = 0, k = (l > pr→table[0] ? l − pr→table[0] : 1); ss ≤ rr; k++)
      ss += ((pl→table[k] + #3ff) ≫ 10) * pr→table[l − k];
  else
    for (ss = 0, k = (l > pr→table[0] ? l − pr→table[0] : 1); ss ≤ rr; k++)
      ss += pl→table[k] * pr→table[l − k];
  pl→tag = k − 1; pr→tag = l − k + 1;
}
```

This code is used in section 21.

25. Arcs. In the general case, we have to combine some of the basic micro-sectors into macro-sectors by adding together the appropriate input/output coefficients. This is a bottom-up pruning process.

Suppose *p* is being formed as the union of *pl* and *pr*. Then the arcs leading out of *p* are obtained by summing the numbers on arcs leading out of *pl* and *pr*; the arcs leading into *p* are obtained by summing the numbers on arcs leading into *pl* and *pr*; the arcs from *p* to itself are obtained by summing the four numbers on arcs leading from *pl* or *pr* to *pl* or *pr*.

We maintain the *node_index* table so that its non-Λ entries contain all the currently active nodes. When *pl* and *pr* are being pruned in favor of *p*, node *p* inherits *pl*'s place in *node_index*; *pr*'s former place becomes Λ.

```
⟨ Put the appropriate arcs into the graph 25 ⟩ ≡
  ⟨ Prune the sectors that are used in macro-sectors, and form the lists of SIC sector codes 28 ⟩;
  ⟨ Make the special nodes invisible if they are omitted, visible otherwise 30 ⟩;
  ⟨ Compute individual thresholds for each chosen sector 27 ⟩;
  { register Vertex *v = new_graph→vertices + n;
    for (k = MAX_N; k; k−−)
      if ((p = node_index[k]) ≠ Λ) {
        vert_index[k] = −−v;
        v→name = gb_save_string(p→title);
        v→SIC_codes = p→SIC_list;
        v→sector_total = p→total;
      } else vert_index[k] = Λ;
    if (v ≠ new_graph→vertices) panic(impossible);      /* bug in algorithm; this can't happen */
    for (j = MAX_N; j; j−−)
      if ((p = node_index[j]) ≠ Λ) { register Vertex *u = vert_index[j];
        for (k = MAX_N; k; k−−)
          if ((v = vert_index[k]) ≠ Λ)
            if (p→table[k] ≠ 0 ∧ p→table[k] > node_index[k]→thresh) {
              gb_new_arc(u, v, 1L);
              u→arcs→flow = p→table[k];
            }
      }
  }
```

This code is used in section 7.

26. ⟨ Private variables 12 ⟩ +≡

```
  static Vertex *vert_index[MAX_N + 1];      /* the vertex assigned to an SIC code */
```

27. The theory underlying this step is the following, for integers a, b, c, d with $b, d > 0$:

$$\frac{a}{b} > \frac{c}{d} \qquad \iff \qquad a > \left\lfloor \frac{b}{d} \right\rfloor c + \left\lfloor \frac{(b \bmod d)c}{d} \right\rfloor .$$

In our case, $b =$ *p→total* and $c =$ *threshold* $\le d = 65536 = 2^{16}$, hence the multiplications cannot overflow. (But they can come awfully darn close.)

```
⟨ Compute individual thresholds for each chosen sector 27 ⟩ ≡
  for (k = MAX_N; k; k−−)
    if ((p = node_index[k]) ≠ Λ) {
      if (threshold ≡ 0) p→thresh = −99999999;
      else p→thresh = ((p→total ≫ 16) * threshold) + (((p→total & #ffff) * threshold) ≫ 16);
    }
```

This code is used in section 25.

28. ⟨ Prune the sectors that are used in macro-sectors, and form the lists of SIC sector codes 28 ⟩ ≡

```
for (p = node_index[ADJ_SEC]; p ≥ node_block; p−−) {    /* bottom up */
  if (p→SIC) {    /* original leaf */
    p→SIC_list = gb_virgin_arc();
    p→SIC_list→len = p→SIC;
  } else {
    pl = p + 1; pr = p→rchild;
    if (p→tag ≡ 0) p→tag = pl→tag + pr→tag;
    if (p→tag ≤ 1) ⟨ Replace pl and pr by their union, p 29 ⟩;
  }
}
```

This code is used in section 25.

29. ⟨ Replace *pl* and *pr* by their union, *p* 29 ⟩ ≡

```
{ register Arc *a = pl→SIC_list;
  register long jj = pl→SIC, kk = pr→SIC;
  p→SIC_list = a;
  while (a→next) a = a→next;
  a→next = pr→SIC_list;
  for (k = MAX_N; k; k−−)
    if ((q = node_index[k]) ≠ Λ) {
      if (q ≠ pl ∧ q ≠ pr) q→table[jj] += q→table[kk];
      p→table[k] = pl→table[k] + pr→table[k];
    }
  p→total = pl→total + pr→total;
  p→SIC = jj;
  p→table[jj] += p→table[kk];
  node_index[jj] = p;
  node_index[kk] = Λ;
}
```

This code is used in section 28.

30. If the Users vertex is not omitted, we need to compute each sector's total final demand, which is calculated so that the row sums and column sums of the input/output coefficients come out equal. We've already computed the column sum, *p→total*; we've also computed *p→table*[1] + ⋯ + *p→table*[ADJ_SEC], and put it into *p→table*[MAX_N]. So now we want to replace *p→table*[MAX_N] by *p→total* − *p→table*[MAX_N]. As remarked earlier, this quantity might be negative.

In the special node *p* for the Users vertex, the preliminary processing has made *p→total* = 0; moreover, *p→table*[MAX_N] is the sum of value added, or GNP. We want to switch those fields.

We don't have to set the *tag* fields to 1 in the special nodes, because the remaining parts of the arc-generation algorithm don't look at those fields.

⟨ Make the special nodes invisible if they are omitted, visible otherwise 30 ⟩ ≡

```
if (omit ≡ 2) node_index[ADJ_SEC] = node_index[MAX_N] = Λ;
else if (omit ≡ 1) node_index[MAX_N] = Λ;
else {
  for (k = ADJ_SEC; k; k−−)
    if ((p = node_index[k]) ≠ Λ) p→table[MAX_N] = p→total − p→table[MAX_N];
  p = node_index[MAX_N];    /* the special node */
  p→total = p→table[MAX_N];
  p→table[MAX_N] = 0;
}
```

This code is used in section 25.

31. Index. As usual, we close with an index that shows where the identifiers of *gb_econ* are defined and used.

⟨ Check the parameters and adjust them for defaults 9 ⟩ Used in section 7.
⟨ Choose all sectors 18 ⟩ Used in section 17.
⟨ Compute individual thresholds for each chosen sector 27 ⟩ Used in section 25.
⟨ Compute the $T(l)$ values for subtree p 22 ⟩ Used in section 21.
⟨ Determine the n sectors to use in the graph 17 ⟩ Used in section 7.
⟨ Grow a random subtree with l leaves 21 ⟩ Used in section 17.
⟨ Grow a subtree with l leaves by subdividing largest sectors first 19 ⟩ Used in section 17.
⟨ If the first node on the list is a leaf, delete it and tag it; otherwise replace it by its two children 20 ⟩ Used in section 19.
⟨ Local variables 8, 13 ⟩ Used in section 7.
⟨ Make the special nodes invisible if they are omitted, visible otherwise 30 ⟩ Used in section 25.
⟨ Private variables 12, 26 ⟩ Used in section 7.
⟨ Prune the sectors that are used in macro-sectors, and form the lists of SIC sector codes 28 ⟩ Used in section 25.
⟨ Put the appropriate arcs into the graph 25 ⟩ Used in section 7.
⟨ Read `econ.dat` and note the binary tree structure 14 ⟩ Used in section 7.
⟨ Read and store the output coefficients for sector k 16 ⟩ Used in section 14.
⟨ Read and store the sector names and SIC numbers 15 ⟩ Used in section 14.
⟨ Replace *pl* and *pr* by their union, p 29 ⟩ Used in section 28.
⟨ Set up a graph with n vertices 10 ⟩ Used in section 7.
⟨ Set $p\rightarrow table[2]$, $p\rightarrow table[3]$, ... to convolution of *pl* and *pr* table entries 23 ⟩ Used in section 22.
⟨ Stochastically determine the number of leaves to grow in each of p's children 24 ⟩ Used in section 21.
⟨ Type declarations 11 ⟩ Used in section 7.
⟨ `gb_econ.h` 1, 5 ⟩

May 19, 2018 at 02:29

GB_ECON

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