Important: Before reading GB_BASIC, please read or at least skim the program for GB_GRAPH.

1. Introduction. This GraphBase module contains six subroutines that generate standard graphs of various types, together with six routines that combine or transform existing graphs.

Simple examples of the use of these routines can be found in the demonstration programs QUEEN and QUEEN_WRAP.

⟨ gb_basic.h 1 ⟩ ≡

```
extern Graph *board();  /* moves on generalized chessboards */
extern Graph *simplex();  /* generalized triangular configurations */
extern Graph *subsets();  /* patterns of subset intersection */
extern Graph *perms();  /* permutations of a multiset */
extern Graph *parts();  /* partitions of an integer */
extern Graph *binary();  /* binary trees */
extern Graph *complement();  /* the complement of a graph */
extern Graph *gunion();  /* the union of two graphs */
extern Graph *intersection();  /* the intersection of two graphs */
extern Graph *lines();  /* the line graph of a graph */
extern Graph *product();  /* the product of two graphs */
extern Graph *induced();  /* a graph induced from another */
```

See also sections 7, 36, 41, 54, 63, 94, 100, 102, and 104.

2. The C file gb_basic.c has the following overall shape:

```
#include "gb_graph.h"  /* we use the GB_GRAPH data structures */
  ⟨ Preprocessor definitions ⟩
  ⟨ Private variables 3 ⟩
  ⟨ Basic subroutines 8 ⟩
  ⟨ Applications of basic subroutines 101 ⟩
```

3. Several of the programs below allocate arrays that will be freed again before the routine is finished.

⟨ Private variables 3 ⟩ ≡

```
static Area working_storage;
```

See also sections 5, 10, and 51.
This code is used in section 2.

4. If a graph-generating subroutine encounters a problem, it returns Λ (that is, NULL), after putting a code number into the external variable *panic_code*. This code number identifies the type of failure. Otherwise the routine 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_free(working_storage);
          gb_trouble_code = 0;
          return Λ;
        }
```

5. The names of vertices are sometimes formed from the names of other vertices, or from potentially long sequences of numbers. We assemble them in the *buffer* array, which is sufficiently long that the vast majority of applications will be unconstrained by size limitations. The programs assume that BUF_SIZE is rather large, but in cases of doubt they ensure that BUF_SIZE will never be exceeded.

```
#define BUF_SIZE 4096
```

⟨ Private variables 3 ⟩ +≡

```
static char buffer[BUF_SIZE];
```

6. Grids and game boards. The subroutine call *board*(*n1*, *n2*, *n3*, *n4*, *piece*, *wrap*, *directed*) constructs a graph based on the moves of generalized chesspieces on a generalized rectangular board. Each vertex of the graph corresponds to a position on the board. Each arc of the graph corresponds to a move from one position to another.

The first parameters, *n1* through *n4*, specify the size of the board. If, for example, a two-dimensional board with n_1 rows and n_2 columns is desired, you set $n1 = n_1$, $n2 = n_2$, and $n3 = 0$; the resulting graph will have $n_1 n_2$ vertices. If you want a three-dimensional board with n_3 layers, set $n3 = n_3$ and $n_4 = 0$. If you want a 4-D board, put the number of 4th coordinates in *n4*. If you want a d-dimensional board with 2^d positions, set $n1 = 2$ and $n2 = -d$.

In general, the *board* subroutine determines the dimensions by scanning the sequence $(n1, n2, n3, n4, 0) = (n_1, n_2, n_3, n_4, 0)$ from left to right until coming to the first nonpositive parameter n_{k+1}. If $k = 0$ (i.e., if $n1 \le 0$), the default size 8×8 will be used; this is an ordinary chessboard with 8 rows and 8 columns. Otherwise if $n_{k+1} = 0$, the board will have k dimensions $n_1, \ldots, n_k$. Otherwise we must have $n_{k+1} < 0$; in this case, the board will have $d = |n_{k+1}|$ dimensions, chosen as the first d elements of the infinite periodic sequence $(n_1, \ldots, n_k, n_1, \ldots, n_k, n_1, \ldots)$. For example, the specification $(n1, n2, n3, n4) = (2, 3, 5, -7)$ is about as tricky as you can get. It produces a seven-dimensional board with dimensions $(n_1, \ldots, n_7) = (2, 3, 5, 2, 3, 5, 2)$, hence a graph with $2 \cdot 3 \cdot 5 \cdot 2 \cdot 3 \cdot 5 \cdot 2 = 1800$ vertices.

The *piece* parameter specifies the legal moves of a generalized chesspiece. If $piece > 0$, a move from position u to position v is considered legal if and only if the Euclidean distance between points u and v is equal to $\sqrt{piece}$. For example, if $piece = 1$ and if we have a two-dimensional board, the legal moves from (x, y) are to $(x, y \pm 1)$ and $(x \pm 1, y)$; these are the moves of a so-called wazir, the only moves that a king and a rook can both make. If $piece = 2$, the legal moves from (x, y) are to $(x \pm 1, y \pm 1)$; these are the four moves that a king and a bishop can both make. (A piece that can make only these moves was called a "fers" in ancient Muslim chess.) If $piece = 5$, the legal moves are those of a knight, from (x, y) to $(x \pm 1, y \pm 2)$ or to $(x \pm 2, y \pm 1)$. If $piece = 3$, there are no legal moves on a two-dimensional board; but moves from (x, y, z) to $(x \pm 1, y \pm 1, z \pm 1)$ would be legal in three dimensions. If $piece = 0$, it is changed to the default value $piece = 1$.

If the value of *piece* is negative, arbitrary multiples of the basic moves for $|piece|$ are permitted. For example, $piece = -1$ defines the moves of a rook, from (x, y) to $(x \pm a, y)$ or to $(x, y \pm a)$ for all $a > 0$; $piece = -2$ defines the moves of a bishop, from (x, y) to $(x \pm a, y \pm a)$. The literature of "fairy chess" assigns standard names to the following *piece* values: wazir = 1, fers = 2, dabbaba = 4, knight = 5, alfil = 8, camel = 10, zebra = 13, giraffe = 17, fiveleaper = 25, root-50-leaper = 50, etc.; rook = −1, bishop = −2, unicorn = −3, dabbabarider = −4, nightrider = −5, alfilrider = −8, camelrider = −10, etc.

To generate a board with the moves of a king, you can use the *gunion* subroutine below to take the union of boards with $piece = 1$ and $piece = 2$. Similarly, you can get queen moves by taking the union of boards with $piece = -1$ and $piece = -2$.

If $piece > 0$, all arcs of the graph will have length 1. If $piece < 0$, the length of each arc will be the number of multiples of a basic move that produced the arc.

7. If the *wrap* parameter is nonzero, it specifies a subset of coordinates in which values are computed modulo the corresponding size. For example, the coordinates (x, y) for vertices on a two-dimensional board are restricted to the range $0 \le x < n_1$, $0 \le y < n_2$; therefore when $wrap = 0$, a move from (x, y) to $(x + \delta_1, y + \delta_2)$ is legal only if $0 \le x + \delta_1 < n_1$ and $0 \le y + \delta_2 < n_2$. But when $wrap = 1$, the x coordinates are allowed to "wrap around"; the move would then be made to $((x + \delta_1) \bmod n_1, y + \delta_2)$, provided that $0 \le y + \delta_2 < n_2$. Setting $wrap = 1$ effectively makes the board into a cylinder instead of a rectangle. Similarly, the y coordinates are allowed to wrap around when $wrap = 2$. Both x and y coordinates are treated modulo their corresponding sizes when $wrap = 3$; the board is then effectively a torus. In general, coordinates k_1, k_2, … will wrap around when $wrap = 2^{k_1-1} + 2^{k_2-1} + \cdots$. Setting $wrap = -1$ causes all coordinates to be computed modulo their size.

The graph constructed by *board* will be undirected unless $directed \ne 0$. Directed *board* graphs will be acyclic when $wrap = 0$, but they may have cycles when $wrap \ne 0$. Precise rules defining the directed arcs are given below.

Several important special cases are worth noting. To get the complete graph on n vertices, you can say $board(n, 0, 0, 0, -1, 0, 0)$. To get the transitive tournament on n vertices, i.e., the directed graph with arcs from u to v when $u < v$, you can say $board(n, 0, 0, 0, -1, 0, 1)$. To get the empty graph on n vertices, you can say $board(n, 0, 0, 0, 2, 0, 0)$. To get a circuit (undirected) or a cycle (directed) of length n, you can say $board(n, 0, 0, 0, 1, 1, 0)$ and $board(n, 0, 0, 0, 1, 1, 1)$, respectively.

```
⟨gb_basic.h 1⟩ +≡
#define complete(n) board((long)(n),0L,0L,0L,-1L,0L,0L)
#define transitive(n) board((long)(n),0L,0L,0L,-1L,0L,1L)
#define empty(n) board((long)(n),0L,0L,0L,2L,0L,0L)
#define circuit(n) board((long)(n),0L,0L,0L,1L,1L,0L)
#define cycle(n) board((long)(n),0L,0L,0L,1L,1L,1L)
```

8. ⟨Basic subroutines 8⟩ ≡

```
Graph *board(n1, n2, n3, n4, piece, wrap, directed)
    long n1, n2, n3, n4;   /* size of board desired */
    long piece;   /* type of moves desired */
    long wrap;   /* mask for coordinate positions that wrap around */
    long directed;   /* should the graph be directed? */
{ ⟨Vanilla local variables 9⟩
  long n;   /* total number of vertices */
  long p;   /* |piece| */
  long l;   /* length of current arc */

  ⟨Normalize the board-size parameters 11⟩;
  ⟨Set up a graph with n vertices 13⟩;
  ⟨Insert arcs or edges for all legal moves 15⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);   /* alas, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

See also sections 26, 37, 43, 55, 64, 74, 78, 81, 87, 95, and 105.

This code is used in section 2.

9. Most of the subroutines in GB_BASIC use the following local variables.

⟨ Vanilla local variables 9 ⟩ ≡
```
Graph *new_graph;   /* the graph being constructed */
register long i, j, k;   /* all-purpose indices */
register long d;   /* the number of dimensions */
register Vertex *v;   /* the current vertex of interest */
register long s;   /* accumulator */
```
This code is used in sections 8, 26, 37, 43, 55, 64, 74, 78, 81, 87, 95, and 105.

10. Several arrays will facilitate the calculations that *board* needs to make. The number of distinct values in coordinate position k will be $nn[k]$; this coordinate position will wrap around if and only if $wr[k] \neq 0$. The current moves under consideration will be from $(x_1, \ldots, x_d)$ to $(x_1 + \delta_1, \ldots, x_k + \delta_k)$, where δ_k is stored in $del[k]$. An auxiliary array *sig* holds the sums $\sigma_k = \delta_1^2 + \cdots + \delta_{k-1}^2$. Additional arrays *xx* and *yy* hold coordinates of vertices before and after a move is made.

Some of these arrays are also used for other purposes by other programs besides *board*; we will meet those programs later.

We limit the number of dimensions to 91 or less. This is hardly a limitation, since the number of vertices would be astronomical even if the dimensionality were only half this big. But some of our later programs will be able to make good use of 40 or 50 dimensions and perhaps more; the number 91 is an upper limit imposed by the number of standard printable characters (see the convention for vertex names in the *perms* routine).

```
#define MAX_D 91
```

⟨ Private variables 3 ⟩ +≡
```
static long nn[MAX_D + 1];   /* component sizes */
static long wr[MAX_D + 1];   /* does this component wrap around? */
static long del[MAX_D + 1];   /* displacements for the current move */
static long sig[MAX_D + 2];   /* partial sums of squares of displacements */
static long xx[MAX_D + 1], yy[MAX_D + 1];   /* coordinate values */
```

11. ⟨ Normalize the board-size parameters 11 ⟩ ≡
```
if (piece ≡ 0) piece = 1;
if (n1 ≤ 0) { n1 = n2 = 8; n3 = 0; }
nn[1] = n1;
if (n2 ≤ 0) { k = 2; d = −n2; n3 = n4 = 0; }
else {
  nn[2] = n2;
  if (n3 ≤ 0) { k = 3; d = −n3; n4 = 0; }
  else {
    nn[3] = n3;
    if (n4 ≤ 0) { k = 4; d = −n4; }
    else { nn[4] = n4; d = 4; goto done; }
  }
}
if (d ≡ 0) { d = k − 1; goto done; }
⟨ Compute component sizes periodically for d dimensions 12 ⟩;
done:   /* now nn[1] through nn[d] are set up */
```
This code is used in section 8.

12. At this point, $nn[1]$ through $nn[k-1]$ are the component sizes that should be replicated periodically. In unusual cases, the number of dimensions might not be as large as the number of specifications.

⟨ Compute component sizes periodically for d dimensions 12 ⟩ ≡

```
if (d > MAX_D) panic(bad_specs);   /* too many dimensions */
for (j = 1; k <= d; j++, k++) nn[k] = nn[j];
```

This code is used in sections 11 and 27.

13. We want to make the subroutine idiot-proof, so we use floating-point arithmetic to make sure that boards with more than a billion cells have not been specified.

```
#define MAX_NNN 1000000000.0
```

⟨ Set up a graph with n vertices 13 ⟩ ≡

```
{ float nnn;   /* approximate size */
  for (n = 1, nnn = 1.0, j = 1; j <= d; j++) {
    nnn *= (float) nn[j];
    if (nnn > MAX_NNN) panic(very_bad_specs);   /* way too big */
    n *= nn[j];   /* this multiplication cannot cause integer overflow */
  }
  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, "board(%ld,%ld,%ld,%ld,%ld,%ld,%d)", n1, n2, n3, n4, piece, wrap,
      directed ? 1 : 0);
  strcpy(new_graph->util_types, "ZZZIIIZZZZZZZZ");
  ⟨ Give names to the vertices 14 ⟩;
}
```

This code is used in section 8.

14. The symbolic name of a board position like $(3,1)$ will be the string '3.1'. The first three coordinates are also stored as integers, in utility fields $x.I$, $y.I$, and $z.I$, because immediate access to those values will be helpful in certain applications. (The coordinates can, of course, always be recovered in a slower fashion from the vertex name, via *sscanf*.)

The process of assigning coordinate values and names is equivalent to adding unity in a mixed-radix number system. Vertex $(x_1,\ldots,x_d)$ will be in position $x_1n_2\ldots n_d+\cdots+x_{d-1}n_d+x_d$ relative to the first vertex of the new graph; therefore it is also possible to deduce the coordinates of a vertex from its address.

⟨ Give names to the vertices 14 ⟩ ≡

```
{ register char *q;   /* string pointer */
  nn[0] = xx[0] = xx[1] = xx[2] = xx[3] = 0;
  for (k = 4; k <= d; k++) xx[k] = 0;
  for (v = new_graph->vertices; ; v++) {
    q = buffer;
    for (k = 1; k <= d; k++) {
      sprintf(q, ".%ld", xx[k]);
      while (*q) q++;
    }
    v->name = gb_save_string(&buffer[1]);   /* omit buffer[0], which is '.' */
    v->x.I = xx[1]; v->y.I = xx[2]; v->z.I = xx[3];
    for (k = d; xx[k] + 1 == nn[k]; k--) xx[k] = 0;
    if (k == 0) break;   /* a "carry" has occurred all the way to the left */
    xx[k]++;   /* increase coordinate k */
  }
}
```

This code is used in section 13.

15. Now we come to a slightly tricky part of the routine: the move generator. Let $p = |piece|$. The outer loop of this procedure runs through all solutions of the equation $\delta_1^2 + \cdots + \delta_d^2 = p$, where the δ's are nonnegative integers. Within that loop, we attach signs to the δ's, but we always leave δ_k positive if $\delta_1 = \cdots = \delta_{k-1} = 0$. For every such vector δ, we generate moves from v to $v + \delta$ for every vertex v. When *directed* $= 0$, we use *gb_new_edge* instead of *gb_new_arc*, so that the reverse arc from $v + \delta$ to v is also generated.

```
⟨Insert arcs or edges for all legal moves 15⟩ ≡
  ⟨Initialize the wr, sig, and del tables 16⟩;
  p = piece;
  if (p < 0) p = −p;
  while (1) {
    ⟨Advance to the next nonnegative del vector, or break if done 17⟩;
    while (1) {
      ⟨Generate moves for the current del vector 19⟩;
      ⟨Advance to the next signed del vector, or restore del to nonnegative values and break 18⟩;
    }
  }
```

This code is used in section 8.

16. The C language does not define ≫ unambiguously. If w is negative, the assignment 'w ≫= 1' here should keep w negative. (However, this technicality doesn't matter except in highly unusual cases when there are more than 32 dimensions.)

```
⟨Initialize the wr, sig, and del tables 16⟩ ≡
  { register long w = wrap;
    for (k = 1; k ≤ d; k++, w ≫= 1) {
      wr[k] = w & 1;
      del[k] = sig[k] = 0;
    }
    sig[0] = del[0] = sig[d + 1] = 0;
  }
```

This code is used in section 15.

17. The *sig* array makes it easy to backtrack through all partitions of p into an ordered sum of squares.

```
⟨Advance to the next nonnegative del vector, or break if done 17⟩ ≡
  for (k = d; sig[k] + (del[k] + 1) ∗ (del[k] + 1) > p; k−−) del[k] = 0;
  if (k ≡ 0) break;
  del[k]++;
  sig[k + 1] = sig[k] + del[k] ∗ del[k];
  for (k++; k ≤ d; k++) sig[k + 1] = sig[k];
  if (sig[d + 1] < p) continue;
```

This code is used in section 15.

18.
```
⟨Advance to the next signed del vector, or restore del to nonnegative values and break 18⟩ ≡
  for (k = d; del[k] ≤ 0; k−−) del[k] = −del[k];
  if (sig[k] ≡ 0) break;    /∗ all but del[k] were negative or zero ∗/
  del[k] = −del[k];    /∗ some entry preceding del[k] is positive ∗/
```

This code is used in section 15.

19. We use the mixed-radix addition technique again when generating moves.

```
⟨ Generate moves for the current del vector 19 ⟩ ≡
  for (k = 1; k ≤ d; k++) xx[k] = 0;
  for (v = new_graph→vertices; ; v++) {
    ⟨ Generate moves from v corresponding to del 20 ⟩;
    for (k = d; xx[k] + 1 ≡ nn[k]; k−−) xx[k] = 0;
    if (k ≡ 0) break;     /* a "carry" has occurred all the way to the left */
    xx[k]++;     /* increase coordinate k */
  }
```

This code is used in section 15.

20. The legal moves when *piece* is negative are derived as follows, in the presence of possible wraparound: Starting at $(x_1, \ldots, x_d)$, we move to $(x_1 + \delta_1, \ldots, x_d + \delta_d)$, $(x_1 + 2\delta_1, \ldots, x_d + 2\delta_d)$, $\ldots$, until either coming to a position with a nonwrapped coordinate out of range or coming back to the original point.

A subtle technicality should be noted: When coordinates are wrapped and $piece > 0$, self-loops are possible—for example, in $board(1,0,0,0,1,1,1)$. But self-loops never arise when $piece < 0$.

```
⟨ Generate moves from v corresponding to del 20 ⟩ ≡
  for (k = 1; k ≤ d; k++) yy[k] = xx[k] + del[k];
  for (l = 1; ; l++) {
    ⟨ Correct for wraparound, or goto no_more if off the board 22 ⟩;
    if (piece < 0) ⟨ Go to no_more if yy = xx 21 ⟩;
    ⟨ Record a legal move from xx to yy 23 ⟩;
    if (piece > 0) goto no_more;
    for (k = 1; k ≤ d; k++) yy[k] += del[k];
  }
  no_more:
```

This code is used in section 19.

21.

```
⟨ Go to no_more if yy = xx 21 ⟩ ≡
  {
    for (k = 1; k ≤ d; k++)
      if (yy[k] ≠ xx[k]) goto unequal;
    goto no_more;
  unequal: ;
  }
```

This code is used in section 20.

22.

```
⟨ Correct for wraparound, or goto no_more if off the board 22 ⟩ ≡
  for (k = 1; k ≤ d; k++) {
    if (yy[k] < 0) {
      if (¬wr[k]) goto no_more;
      do yy[k] += nn[k]; while (yy[k] < 0);
    } else if (yy[k] ≥ nn[k]) {
      if (¬wr[k]) goto no_more;
      do yy[k] −= nn[k]; while (yy[k] ≥ nn[k]);
    }
  }
```

This code is used in section 20.

23. ⟨ Record a legal move from xx to yy 23 ⟩ ≡

for $(k = 2, j = yy[1];\ k \le d;\ k{+}{+})\ j = nn[k] * j + yy[k];$
if $(directed)\ gb\_new\_arc(v, new\_graph\rightarrow vertices + j, l);$
else $gb\_new\_edge(v, new\_graph\rightarrow vertices + j, l);$

This code is used in section 20.

24. Generalized triangular boards. The subroutine call *simplex*$(n, n0, n1, n2, n3, n4, directed)$ creates a graph based on generalized triangular or tetrahedral configurations. Such graphs are similar in spirit to the game boards created by *board*, but they pertain to nonrectangular grids like those in Chinese checkers. As with *board* in the case *piece* $= 1$, the vertices represent board positions and the arcs run from board positions to their nearest neighbors. Each arc has length 1.

More formally, the vertices can be defined as sequences of nonnegative integers $(x_0, x_1, \ldots, x_d)$ whose sum is n, where two sequences are considered adjacent if and only if they differ by ±1 in exactly two components—equivalently, if the Euclidean distance between them is $\sqrt{2}$. When $d = 2$, for example, the vertices can be visualized as a triangular array

$$\begin{array}{c} (0,0,3) \\ (0,1,2) \quad (1,0,2) \\ (0,2,1) \quad (1,1,1) \quad (2,0,1) \\ (0,3,0) \quad (1,2,0) \quad (2,1,0) \quad (3,0,0) \end{array}$$

containing $(n + 1)(n + 2)/2$ elements, illustrated here when $n = 3$; each vertex of the array has up to 6 neighbors. When $d = 3$ the vertices form a tetrahedral array, a stack of triangular layers, and they can have as many as 12 neighbors. In general, a vertex in a d-simplicial array will have up to $d(d + 1)$ neighbors.

If the *directed* parameter is nonzero, arcs run only from vertices to neighbors that are lexicographically greater—for example, downward or to the right in the triangular array shown. The directed graph is therefore acyclic, and a vertex of a d-simplicial array has out-degree at most $d(d + 1)/2$.

25. The first parameter, n, specifies the sum of the coordinates $(x_0, x_1, \ldots, x_d)$. The following parameters *n0* through *n4* specify upper bounds on those coordinates, and they also specify the dimensionality d.

If, for example, *n0*, *n1*, and *n2* are positive while *n3* $= 0$, the value of d will be 2 and the coordinates will be constrained to satisfy $0 \le x_0 \le n0$, $0 \le x_1 \le n1$, $0 \le x_2 \le n2$. These upper bounds essentially lop off the corners of the triangular array. We obtain a hexagonal board with $6m$ boundary cells by asking for *simplex*$(3m, 2m, 2m, 2m, 0, 0, 0)$. We obtain the diamond-shaped board used in the game of Hex [Martin Gardner, *The Scientific American Book of Mathematical Puzzles & Diversions* (Simon & Schuster, 1959), Chapter 8] by calling *simplex*$(20, 10, 20, 10, 0, 0, 0)$.

In general, *simplex* determines d and upper bounds $(n_0, n_1, \ldots, n_d)$ in the following way: Let the first nonpositive entry of the sequence $(n0, n1, n2, n3, n4, 0) = (n_0, n_1, n_2, n_3, n_4, 0)$ be n_k. If $k > 0$ and $n_k = 0$, the value of d will be $k - 1$ and the coordinates will be bounded by the given numbers $(n_0, \ldots, n_d)$. If $k > 0$ and $n_k < 0$, the value of d will be $|n_k|$ and the coordinates will be bounded by the first $d + 1$ elements of the infinite periodic sequence $(n_0, \ldots, n_{k-1}, n_0, \ldots, n_{k-1}, n_0, \ldots)$. If $k = 0$ and $n_0 < 0$, the value of d will be $|n_0|$ and the coordinates will be unbounded; equivalently, we may set $n_0 = \cdots = n_d = n$. In this case the number of vertices will be $\binom{n+d}{d}$. Finally, if $k = 0$ and $n_0 = 0$, we have the default case of a triangular array with $3n$ boundary cells, exactly as if $n_0 = -2$.

For example, the specification *n0* $= 3$, *n1* $= -5$ will produce all vertices $(x_0, x_1, \ldots, x_5)$ such that $x_0 + x_1 + \cdots + x_5 = n$ and $0 \le x_j \le 3$. The specification *n0* $= 1$, *n1* $= -d$ will essentially produce all n-element subsets of the $(d + 1)$-element set $\{0, 1, \ldots, d\}$, because we can regard an element k as being present in the set if $x_k = 1$ and absent if $x_k = 0$. In that case two subsets are adjacent if and only if they have exactly $n - 1$ elements in common.

26. ⟨ Basic subroutines 8 ⟩ +≡

```
Graph *simplex(n, n0, n1, n2, n3, n4, directed)
    unsigned long n;    /* the constant sum of all coordinates */
    long n0, n1, n2, n3, n4;    /* constraints on coordinates */
    long directed;    /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  ⟨ Normalize the simplex parameters 27 ⟩;
  ⟨ Create a graph with one vertex for each point 28 ⟩;
  ⟨ Name the points and create the arcs or edges 31 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* darn, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

27. ⟨ Normalize the simplex parameters 27 ⟩ ≡

```
if (n0 ≡ 0) n0 = −2;
if (n0 < 0) { k = 2; nn[0] = n; d = −n0; n1 = n2 = n3 = n4 = 0; }
else {
  if (n0 > n) n0 = n;
  nn[0] = n0;
  if (n1 ≤ 0) { k = 2; d = −n1; n2 = n3 = n4 = 0; }
  else {
    if (n1 > n) n1 = n;
    nn[1] = n1;
    if (n2 ≤ 0) { k = 3; d = −n2; n3 = n4 = 0; }
    else {
      if (n2 > n) n2 = n;
      nn[2] = n2;
      if (n3 ≤ 0) { k = 4; d = −n3; n4 = 0; }
      else {
        if (n3 > n) n3 = n;
        nn[3] = n3;
        if (n4 ≤ 0) { k = 5; d = −n4; }
        else { if (n4 > n) n4 = n;
          nn[4] = n4; d = 4; goto done; }
      }
    }
  }
}
if (d ≡ 0) { d = k − 2; goto done; }
nn[k − 1] = nn[0];
⟨ Compute component sizes periodically for d dimensions 12 ⟩;
done:    /* now nn[0] through nn[d] are set up */
```

This code is used in sections 26, 37, and 44.

28. ⟨ Create a graph with one vertex for each point 28 ⟩ ≡

```
⟨ Determine the number of feasible (x_0, ..., x_d), and allocate the graph 29 ⟩;
sprintf(new_graph→id, "simplex(%lu,%ld,%ld,%ld,%ld,%ld,%d)", n, n0, n1, n2, n3, n4, directed ? 1 : 0);
strcpy(new_graph→util_types, "VVZIIIZZZZZZZZ");    /* hash table will be used */
```

This code is used in section 26.

29. We determine the number of vertices by determining the coefficient of z^n in the power series

$$(1+z+\cdots+z^{n_0})(1+z+\cdots+z^{n_1})\ldots(1+z+\cdots+z^{n_d}).$$

⟨ Determine the number of feasible $(x_0,\ldots,x_d)$, and allocate the graph 29 ⟩ ≡

```
{ long nverts;    /* the number of vertices */
  register long *coef = gb_typed_alloc(n + 1, long, working_storage);
  if (gb_trouble_code) panic(no_room + 1);    /* can't allocate coef array */
  for (k = 0; k ≤ nn[0]; k++) coef[k] = 1;
      /* now coef represents the coefficients of 1 + z + ··· + z^{n_0} */
  for (j = 1; j ≤ d; j++) ⟨ Multiply the power series coefficients by 1 + z + ··· + z^{n_j} 30 ⟩;
  nverts = coef[n];
  gb_free(working_storage);    /* recycle the coef array */
  new_graph = gb_new_graph(nverts);
  if (new_graph ≡ Λ) panic(no_room);    /* out of memory before we're even started */
}
```

This code is used in sections 28 and 38.

30. There's a neat way to multiply by $1+z+\cdots+z^{n_j}$: We multiply first by $1-z^{n_j+1}$, then sum the coefficients.

We want to detect impossibly large specifications without risking integer overflow. It is easy to do this because multiplication is being done via addition.

⟨ Multiply the power series coefficients by $1+z+\cdots+z^{n_j}$ 30 ⟩ ≡

```
{
  for (k = n, i = n − nn[j] − 1; i ≥ 0; k−−, i−−) coef[k] −= coef[i];
  s = 1;
  for (k = 1; k ≤ n; k++) {
    s += coef[k];
    if (s > 1000000000) panic(very_bad_specs);    /* way too big */
    coef[k] = s;
  }
}
```

This code is used in section 29.

31. As we generate the vertices, it proves convenient to precompute an array containing the numbers $y_j = n_j + \cdots + n_d$, which represent the largest possible sums $x_j + \cdots + x_d$. We also want to maintain the numbers $\sigma_j = n - (x_0 + \cdots + x_{j-1}) = x_j + \cdots + x_d$. The conditions

$$0 \le x_j \le n_j, \qquad \sigma_j - y_{j+1} \le x_j \le \sigma_j$$

are necessary and sufficient, in the sense that we can find at least one way to complete a partial solution $(x_0, \ldots, x_k)$ to a full solution $(x_0, \ldots, x_d)$ if and only if the conditions hold for all $j \le k$.

There is at least one solution if and only if $n \le y_0$.

We enter the name string into a hash table, using the *hash_in* routine of GB_GRAPH, because there is no simple way to compute the location of a vertex from its coordinates.

$\langle$ Name the points and create the arcs or edges 31 $\rangle \equiv$
  $v = \mathit{new\_graph} \rightarrow \mathit{vertices}$;
  $\mathit{yy}[d+1] = 0$; $\mathit{sig}[0] = n$;
  for $(k = d;\ k \ge 0;\ k{-}{-})\ \mathit{yy}[k] = \mathit{yy}[k+1] + \mathit{nn}[k]$;
  if $(\mathit{yy}[0] \ge n)$ {
    $k = 0$; $\mathit{xx}[0] = (\mathit{yy}[1] \ge n\ ?\ 0 : n - \mathit{yy}[1])$;
    while (1) {
      $\langle$ Complete the partial solution $(x_0, \ldots, x_k)$ 32 $\rangle$;
      $\langle$ Assign a symbolic name for $(x_0, \ldots, x_d)$ to vertex v 34 $\rangle$;
      $\mathit{hash\_in}(v)$;   /* enter $v \rightarrow \mathit{name}$ into the hash table (via utility fields u, v) */
      $\langle$ Create arcs or edges from previous points to v 35 $\rangle$;
      $v{+}{+}$;
      $\langle$ Advance to the next partial solution $(x_0, \ldots, x_k)$, where k is as large as possible; **goto** *last* if there are no more solutions 33 $\rangle$;
    }
  }
last: **if** $(v \ne \mathit{new\_graph} \rightarrow \mathit{vertices} + \mathit{new\_graph} \rightarrow n)$ $\mathit{panic}(\mathit{impossible})$;   /* can't happen */

This code is used in section 26.

32. $\langle$ Complete the partial solution $(x_0, \ldots, x_k)$ 32 $\rangle \equiv$
  for $(s = \mathit{sig}[k] - \mathit{xx}[k], k{+}{+};\ k \le d;\ s \mathrel{-}= \mathit{xx}[k], k{+}{+})$ {
    $\mathit{sig}[k] = s$;
    if $(s \le \mathit{yy}[k+1])\ \mathit{xx}[k] = 0$;
    else $\mathit{xx}[k] = s - \mathit{yy}[k+1]$;
  }
  if $(s \ne 0)$ $\mathit{panic}(\mathit{impossible} + 1)$ /* can't happen */

This code is used in sections 31 and 39.

33. Here we seek the largest k such that x_k can be increased without violating the necessary and sufficient conditions stated earlier.

$\langle$ Advance to the next partial solution $(x_0, \ldots, x_k)$, where k is as large as possible; **goto** *last* if there are no more solutions 33 $\rangle \equiv$
  for $(k = d - 1;\ ;\ k{-}{-})$ {
    if $(\mathit{xx}[k] < \mathit{sig}[k] \land \mathit{xx}[k] < \mathit{nn}[k])$ **break**;
    if $(k \equiv 0)$ **goto** *last*;
  }
  $\mathit{xx}[k]{+}{+}$;

This code is used in sections 31 and 39.

34. As in the *board* routine, we represent the sequence of coordinates $(2,0,1)$ by the string '`2.0.1`'. The string won't exceed `BUF_SIZE`, because the ratio `BUF_SIZE/MAX_D` is plenty big.

The first three coordinate values, (x_0, x_1, x_2), are placed into utility fields x, y, and z, so that they can be accessed immediately if an application needs them.

⟨ Assign a symbolic name for $(x_0, \ldots, x_d)$ to vertex v 34 ⟩ ≡

```
{ register char *p = buffer;    /* string pointer */
  for (k = 0; k <= d; k++) {
    sprintf(p, ".%ld", xx[k]);
    while (*p) p++;
  }
  v->name = gb_save_string(&buffer[1]);    /* omit buffer[0], which is '.' */
  v->x.I = xx[0]; v->y.I = xx[1]; v->z.I = xx[2];
}
```

This code is used in sections 31 and 39.

35. Since we are generating the vertices in lexicographic order of their coordinates, it is easy to identify all adjacent vertices that precede the current setting of $(x_0, x_1, \ldots, x_d)$. We locate them via their symbolic names.

⟨ Create arcs or edges from previous points to v 35 ⟩ ≡

```
for (j = 0; j < d; j++)
  if (xx[j]) { register Vertex *u;    /* previous vertex adjacent to v */
    xx[j]--;
    for (k = j + 1; k <= d; k++)
      if (xx[k] < nn[k]) { register char *p = buffer;    /* string pointer */
        xx[k]++;
        for (i = 0; i <= d; i++) {
          sprintf(p, ".%ld", xx[i]);
          while (*p) p++;
        }
        u = hash_out(&buffer[1]);
        if (u == Λ) panic(impossible + 2);    /* can't happen */
        if (directed) gb_new_arc(u, v, 1L);
        else gb_new_edge(u, v, 1L);
        xx[k]--;
      }
    xx[j]++;
  }
```

This code is used in section 31.

36. Subset graphs. The subroutine call *subsets*$(n, n0, n1, n2, n3, n4, size\_bits, directed)$ creates a graph having the same vertices as *simplex*$(n, n0, n1, n2, n3, n4, directed)$ but with a quite different notion of adjacency. In this we interpret a solution $(x_0, x_1, \ldots, x_d)$ to the conditions $x_0 + x_1 + \cdots + x_d = n$ and $0 \le x_j \le n_j$ not as a position on a game board but as an n-element submultiset of the multiset $\{n_0 \cdot 0, n_1 \cdot 1, \ldots, n_d \cdot d\}$ that has x_j elements equal to j. (If each $n_j = 1$, the multiset is a set; this is an important special case.) Two vertices are adjacent if and only if their intersection has a cardinality that matches one of the bits in *size_bits*, which is an unsigned integer. Each arc has length 1.

For example, suppose $n = 3$ and $(n0, n1, n2, n3) = (2, 2, 2, 0)$. Then the vertices are the 3-element submultisets of $\{0, 0, 1, 1, 2, 2\}$, namely

$$\{0,0,1\}, \quad \{0,0,2\}, \quad \{0,1,2\}, \quad \{0,2,2\}, \quad \{1,1,2\}, \quad \{1,2,2\},$$

which are represented by the respective vectors

$$(2,1,0), \quad (2,0,1), \quad (1,1,1), \quad (1,0,2), \quad (0,2,1), \quad (0,1,2).$$

The intersection of multisets represented by $(x_0, x_1, \ldots, x_d)$ and $(y_0, y_1, \ldots, y_d)$ is

$$\bigl(\min(x_0, y_0), \min(x_1, y_1), \ldots, \min(x_d, y_d)\bigr);$$

each element occurs as often as it occurs in both multisets being intersected. If now $size\_bits = 3$, the multisets will be considered adjacent whenever their intersection contains exactly 0 or 1 elements, because $3 = 2^0 + 2^1$. The vertices adjacent to $\{0, 0, 1\}$, for example, will be $\{0, 2, 2\}$ and $\{1, 2, 2\}$. In this case, every pair of submultisets has a nonempty intersection, so the same graph would be obtained if $size\_bits = 2$.

If *directed* is nonzero, the graph will have directed arcs, from u to v only if $u \le v$. Notice that the graph will have self-loops if and only if the binary representation of *size_bits* contains the term 2^n, in which case there will be a loop from every vertex to itself. (In an undirected graph, such loops are represented by two arcs.)

We define a macro *disjoint_subsets*(n, k) for the case of $\binom{n}{k}$ vertices, adjacent if and only if they represent disjoint k-subsets of an n-set. One important special case is the Petersen graph, whose vertices are the 2-element subsets of $\{0, 1, 2, 3, 4\}$, adjacent when they are disjoint. This graph is remarkable because it contains 10 vertices, each of degree 3, but it has no circuits of length less than 5.

```
⟨ gb_basic.h 1 ⟩ +≡
#define disjoint_subsets(n, k) subsets((long)(k), 1L, (long)(1 − (n)), 0L, 0L, 0L, 1L, 0L)
#define petersen( ) disjoint_subsets(5, 2)
```

37. ⟨ Basic subroutines 8 ⟩ +≡

```
  Graph *subsets(n, n0, n1, n2, n3, n4, size_bits, directed)
      unsigned long n;   /* the number of elements in the multiset */
      long n0, n1, n2, n3, n4;   /* multiplicities of elements */
      unsigned long size_bits;   /* intersection sizes that trigger arcs */
      long directed;   /* should the graph be directed? */
  { ⟨ Vanilla local variables 9 ⟩
    ⟨ Normalize the simplex parameters 27 ⟩;
    ⟨ Create a graph with one vertex for each subset 38 ⟩;
    ⟨ Name the subsets and create the arcs or edges 39 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);   /* rats, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

38. ⟨ Create a graph with one vertex for each subset 38 ⟩ ≡

```
⟨ Determine the number of feasible (x_0, ..., x_d), and allocate the graph 29 ⟩;
sprintf(new_graph->id, "subsets(%lu,%ld,%ld,%ld,%ld,%ld,0x%lx,%d)", n, n0, n1, n2, n3, n4,
      size_bits, directed ? 1 : 0);
strcpy(new_graph->util_types, "ZZZIIIZZZZZZZZ");    /* hash table will not be used */
```

This code is used in section 37.

39. We generate the vertices with exactly the logic used in *simplex*.

⟨ Name the subsets and create the arcs or edges 39 ⟩ ≡

```
v = new_graph->vertices;
yy[d + 1] = 0; sig[0] = n;
for (k = d; k ≥ 0; k--) yy[k] = yy[k + 1] + nn[k];
if (yy[0] ≥ n) {
  k = 0; xx[0] = (yy[1] ≥ n ? 0 : n − yy[1]);
  while (1) {
    ⟨ Complete the partial solution (x_0, ..., x_k) 32 ⟩;
    ⟨ Assign a symbolic name for (x_0, ..., x_d) to vertex v 34 ⟩;
    ⟨ Create arcs or edges from previous subsets to v 40 ⟩;
    v++;
    ⟨ Advance to the next partial solution (x_0, ..., x_k), where k is as large as possible; goto last if there
        are no more solutions 33 ⟩;
  }
}
last: if (v ≠ new_graph->vertices + new_graph->n) panic(impossible);    /* can't happen */
```

This code is used in section 37.

40. The only difference is that we generate the arcs or edges by brute force, examining each pair of vertices to see if they are adjacent or not.

The code here is character-set dependent: It assumes that '`.`' and null have a character code less than '`0`', as in ASCII. It also assumes that characters occupy exactly eight bits.

```
#define UL_BITS 8 * sizeof(unsigned long)    /* the number of bits in size_bits */
```

⟨ Create arcs or edges from previous subsets to v 40 ⟩ ≡

```
{ register Vertex *u;
  for (u = new_graph->vertices; u ≤ v; u++) { register char *p = u->name;
    long ss = 0;    /* the number of elements common to u and v */
    for (j = 0; j ≤ d; j++, p++) {
      for (s = (*p++) − '0'; *p ≥ '0'; p++) s = 10 * s + *p − '0';    /* sscanf(p, "%ld", &s) */
      if (xx[j] < s) ss += xx[j];
      else ss += s;
    }
    if (ss < UL_BITS ∧ (size_bits & (((unsigned long) 1) ≪ ss))) {
      if (directed) gb_new_arc(u, v, 1L);
      else gb_new_edge(u, v, 1L);
    }
  }
}
```

This code is used in section 39.

41. Permutation graphs. The subroutine call *perms*(*n0*, *n1*, *n2*, *n3*, *n4*, *max_inv*, *directed*) creates a graph whose vertices represent the permutations of a multiset that have at most *max_inv* inversions. Two permutations are adjacent in the graph if one is obtained from the other by interchanging two adjacent elements. Each arc has length 1.

For example, the multiset $\{0,0,1,2\}$ has the following twelve permutations:

$$\begin{array}{cccccc} 0012, & 0021, & 0102, & 0120, & 0201, & 0210, \\ 1002, & 1020, & 1200, & 2001, & 2010, & 2100. \end{array}$$

The first of these, 0012, has two neighbors, 0021 and 0102.

The number of inversions is the number of pairs of elements xy such that $x > y$ and x precedes y from left to right, counting multiplicity. For example, 2010 has four inversions, corresponding to $xy \in \{20, 21, 20, 10\}$. It is not difficult to verify that the number of inversions of a permutation is the distance in the graph from that permutation to the lexicographically first permutation.

Parameters *n0* through *n4* specify the composition of the multiset, just as in the *subsets* routine. Roughly speaking, there are *n0* elements equal to 0, *n1* elements equal to 1, and so on. The multiset $\{0,0,1,2,3,3\}$, for example, would be represented by $(n0, n1, n2, n3, n4) = (2,1,1,2,0)$.

Of course, we sometimes want to have multisets with more than five distinct elements; when there are $d+1$ distinct elements, the multiset should have n_k elements equal to k and $n = n_0 + n_1 + \cdots + n_d$ elements in all. Larger values of d can be specified by using $-d$ as a parameter: If $n0 = -d$, each multiplicity n_k is taken to be 1; if $n0 > 0$ and $n1 = -d$, each multiplicity n_k is taken to be equal to $n0$; if $n0 > 0$, $n1 > 0$, and $n2 = -d$, the multiplicities are alternately $(n0, n1, n0, n1, n0, \ldots)$; if $n0 > 0$, $n1 > 0$, $n2 > 0$, and $n3 = -d$, the multiplicities are the first $d+1$ elements of the periodic sequence $(n0, n1, n2, n0, n1, \ldots)$; and if all but $n4$ are positive, while $n4 = -d$, the multiplicities again are periodic.

An example like $(n0, n1, n2, n3, n4) = (1,2,3,4,-8)$ is about as tricky as you can get. It specifies the multiset $\{0,1,1,2,2,2,3,3,3,3,4,5,5,6,6,6,7,7,7,7,8\}$.

If any of the multiplicity parameters is negative or zero, the remaining multiplicities are ignored. For example, if $n2 \le 0$, the subroutine does not look at *n3* or *n4*.

You probably don't want to try *perms*(*n0*, 0, 0, 0, 0, *max_inv*, *directed*) when $n0 > 0$, because a multiset with *n0* identical elements has only one permutation.

The special case when you want all $n!$ permutations of an n-element set can be obtained by calling *all_perms*(n, *directed*).

⟨ `gb_basic.h` 1 ⟩ +≡

#define *all_perms*(n, *directed*) *perms*((**long**)$(1-(n))$, 0_L, 0_L, 0_L, 0_L, 0_L, (**long**)(*directed*))

42. If $max\_inv = 0$, all permutations will be considered, regardless of the number of inversions. In that case the total number of vertices in the graph will be the multinomial coefficient

$$\binom{n}{n_0, n_1, \ldots, n_d}, \qquad n = n_0 + n_1 + \cdots + n_d.$$

The maximum number of inversions in general is the number of inversions of the lexicographically last permutation, namely $\binom{n}{2} - \binom{n_0}{2} - \binom{n_1}{2} - \cdots - \binom{n_d}{2} = \sum_{0 \le j < k \le d} n_j n_k$.

Notice that in the case $d = 1$, we essentially obtain all combinations of $n0 + n1$ elements taken $n1$ at a time. The positions of the 1's correspond to the elements of a subset or sample.

If *directed* is nonzero, the graph will contain only directed arcs from permutations to neighboring permutations that have exactly one more inversion. In this case the graph corresponds to a partial ordering that is a lattice with interesting properties; see the article by Bennett and Birkhoff in *Algebra Universalis* (1994), to appear.

43. The program for *perms* is very similar in structure to the program for *simplex* already considered.

```
⟨ Basic subroutines 8 ⟩ +≡
  Graph *perms(n0, n1, n2, n3, n4, max_inv, directed)
      long n0, n1, n2, n3, n4;      /* composition of the multiset */
      unsigned long max_inv;      /* maximum number of inversions */
      long directed;      /* should the graph be directed? */
  { ⟨ Vanilla local variables 9 ⟩
    register long n;      /* total number of elements in multiset */
    ⟨ Normalize the permutation parameters 44 ⟩;
    ⟨ Create a graph with one vertex for each permutation 46 ⟩;
    ⟨ Name the permutations and create the arcs or edges 48 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);      /* shucks, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

44.
```
⟨ Normalize the permutation parameters 44 ⟩ ≡
  if (n0 ≡ 0) { n0 = 1; n1 = 0; }      /* convert the empty set into {0} */
  else if (n0 < 0) { n1 = n0; n0 = 1; }
  n = BUF_SIZE;      /* this allows us to borrow code from simplex, already written */
  ⟨ Normalize the simplex parameters 27 ⟩;
  ⟨ Determine n and the maximum possible number of inversions 45 ⟩;
```

This code is used in section 43.

45. Here we want to set *max_inv* to the maximum possible number of inversions, if the given value of *max_inv* is zero or if it exceeds that maximum number.

```
⟨ Determine n and the maximum possible number of inversions 45 ⟩ ≡
  { register long ss;      /* max inversions known to be possible */
    for (k = 0, s = ss = 0; k ≤ d; ss += s * nn[k], s += nn[k], k++)
      if (nn[k] ≥ BUF_SIZE) panic(bad_specs);      /* too many elements in the multiset */
    if (s ≥ BUF_SIZE) panic(bad_specs + 1);      /* too many elements in the multiset */
    n = s;
    if (max_inv ≡ 0 ∨ max_inv > ss) max_inv = ss;
  }
```

This code is used in section 44.

46. To determine the number of vertices, we sum the first $max\_inv + 1$ coefficients of a power series in which the coefficient of z^j is the number of permutations having j inversions. It is known [*Sorting and Searching*, exercise 5.1.2–16] that this power series is the "z-multinomial coefficient"

$$\binom{n}{n_0,\ldots,n_d}_z = \frac{n!_z}{n_0!_z\ldots n_d!_z}\,, \qquad \text{where} \qquad m!_z = \prod_{k=1}^{m} \frac{1-z^k}{1-z}\,.$$

⟨ Create a graph with one vertex for each permutation 46 ⟩ ≡

```
{ long nverts;    /* the number of vertices */
  register long *coef = gb_typed_alloc(max_inv + 1, long, working_storage);
  if (gb_trouble_code) panic(no_room + 1);    /* can't allocate coef array */
  coef[0] = 1;
  for (j = 1, s = nn[0]; j ≤ d; s += nn[j], j++)
    ⟨ Multiply the power series coefficients by ∏_{1≤k≤n_j} (1 − z^{s+k})/(1 − z^k) 47 ⟩;
  for (k = 1, nverts = 1; k ≤ max_inv; k++) {
    nverts += coef[k];
    if (nverts > 1000000000) panic(very_bad_specs);    /* way too big */
  }
  gb_free(working_storage);    /* recycle the coef array */
  new_graph = gb_new_graph(nverts);
  if (new_graph ≡ Λ) panic(no_room);    /* out of memory before we're even started */
  sprintf(new_graph→id, "perms(%ld,%ld,%ld,%ld,%ld,%lu,%d)", n0, n1, n2, n3, n4, max_inv,
      directed ? 1 : 0);
  strcpy(new_graph→util_types, "VVZZZZZZZZZZZZ");    /* hash table will be used */
}
```

This code is used in section 43.

47. After multiplication by $(1-z^{k+s})/(1-z^k)$, the coefficients of the power series will be nonnegative, because they are the coefficients of a z-multinomial coefficient.

⟨ Multiply the power series coefficients by $\prod_{1\le k\le n_j}(1-z^{s+k})/(1-z^k)$ 47 ⟩ ≡

```
for (k = 1; k ≤ nn[j]; k++) { register long ii;
  for (i = max_inv, ii = i − k − s; ii ≥ 0; ii−−, i−−) coef[i] −= coef[ii];
  for (i = k, ii = 0; i ≤ max_inv; i++, ii++) {
    coef[i] += coef[ii];
    if (coef[i] > 1000000000) panic(very_bad_specs + 1);    /* way too big */
  }
}
```

This code is used in section 46.

48. As we generate the permutations, we maintain a table $(y_1, \ldots, y_n)$, where y_k is the number of inversions whose first element is the kth element of the multiset. For example, if the multiset is $\{0,0,1,2\}$ and the current permutation is $(2,0,1,0)$, the inversion table is $(y_1, y_2, y_3, y_4) = (0,0,1,3)$. Clearly $0 \le y_k < k$, and $y_k \le y_{k-1}$ when the kth element of the multiset is the same as the $(k-1)$st element. These conditions are necessary and sufficient to define a valid inversion table. We will generate permutations in lexicographic order of their inversion tables.

For convenience, we set up another array z, which holds the initial inversion-free permutation.

```
⟨ Name the permutations and create the arcs or edges 48 ⟩ ≡
  { register long *xtab, *ytab, *ztab;     /* permutations and their inversions */
    long m = 0;     /* current number of inversions */
    ⟨ Initialize xtab, ytab, and ztab 49 ⟩;
    v = new_graph→vertices;
    while (1) {
      ⟨ Assign a symbolic name for (x_1, ..., x_n) to vertex v 52 ⟩;
      ⟨ Create arcs or edges from previous permutations to v 53 ⟩;
      v++;
      ⟨ Advance to the next perm; goto last if there are no more solutions 50 ⟩;
    }
  last: if (v ≠ new_graph→vertices + new_graph→n) panic(impossible);     /* can't happen */
    gb_free(working_storage);
  }
```

This code is used in section 43.

49. ⟨ Initialize *xtab*, *ytab*, and *ztab* 49 ⟩ ≡

```
  xtab = gb_typed_alloc(3 * n + 3, long, working_storage);
  if (gb_trouble_code) {     /* can't allocate xtab */
    gb_recycle(new_graph); panic(no_room + 2); }
  ytab = xtab + (n + 1);
  ztab = ytab + (n + 1);
  for (j = 0, k = 1, s = nn[0]; ; k++) {
    xtab[k] = ztab[k] = j;     /* ytab[k] = 0 */
    if (k ≡ s) {
      if (++j > d) break;
      else s += nn[j];
    }
  }
```

This code is used in section 48.

50. Here is the heart of the permutation logic. We find the largest k such that y_k can legitimately be increased by 1. When we encounter a k for which y_k cannot be increased, we set $y_k = 0$ and adjust the x's accordingly. If no y_k can be increased, we are done.

⟨ Advance to the next perm; **goto** *last* if there are no more solutions 50 ⟩ ≡

```
for (k = n; k; k--) {
  if (m < max_inv ∧ ytab[k] < k - 1)
    if (ytab[k] < ytab[k - 1] ∨ ztab[k] > ztab[k - 1]) goto move;
  if (ytab[k]) {
    for (j = k - ytab[k]; j < k; j++) xtab[j] = xtab[j + 1];
    m -= ytab[k];
    ytab[k] = 0;
    xtab[k] = ztab[k];
  }
}
goto last;
move: j = k - ytab[k];    /* the current location of the kth element, z_k */
xtab[j] = xtab[j - 1]; xtab[j - 1] = ztab[k];
ytab[k]++; m++;
```

This code is used in section 48.

51. A permutation is encoded as a sequence of nonblank characters, using an abbreviated copy of the *imap* code from GB_IO and omitting the characters that need to be quoted within strings. If the number of distinct elements in the multiset is at most 62, only digits and letters will appear in the vertex name.

⟨ Private variables 3 ⟩ +≡

```
static char *short_imap = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw\
    xyz_^~&@,;.:?!%#$+-*/|<=>()[]{}‘’";
```

52. ⟨ Assign a symbolic name for $(x_1, \ldots, x_n)$ to vertex v 52 ⟩ ≡

```
{ register char *p;
  register long *q;
  for (p = &buffer[n - 1], q = &xtab[n]; q > xtab; p--, q--) *p = short_imap[*q];
  v->name = gb_save_string(buffer);
  hash_in(v);    /* enter v->name into the hash table (via utility fields u, v) */
}
```

This code is used in section 48.

53. Since we are generating the vertices in lexicographic order of their inversions, it is easy to identify all adjacent vertices that precede the current setting of $(x_1, \ldots, x_n)$. We locate them via their symbolic names.

⟨ Create arcs or edges from previous permutations to v 53 ⟩ ≡

```
for (j = 1; j < n; j++)
  if (xtab[j] > xtab[j + 1]) { register Vertex *u;    /* previous vertex adjacent to v */
    buffer[j - 1] = short_imap[xtab[j + 1]]; buffer[j] = short_imap[xtab[j]];
    u = hash_out(buffer);
    if (u ≡ Λ) panic(impossible + 2);    /* can't happen */
    if (directed) gb_new_arc(u, v, 1L);
    else gb_new_edge(u, v, 1L);
    buffer[j - 1] = short_imap[xtab[j]]; buffer[j] = short_imap[xtab[j + 1]];
  }
```

This code is used in section 48.

54. Partition graphs. The subroutine call *parts*(*n*, *max_parts*, *max_size*, *directed*) creates a graph whose vertices represent the different ways to partition the integer n into at most *max_parts* parts, where each part is at most *max_size*. Two partitions are adjacent in the graph if one can be obtained from the other by combining two parts. Each arc has length 1.

For example, the partitions of 5 are

$$5, \quad 4+1, \quad 3+2, \quad 3+1+1, \quad 2+2+1, \quad 2+1+1+1, \quad 1+1+1+1+1.$$

Here 5 is adjacent to $4+1$ and to $3+2$; $4+1$ is adjacent also to $3+1+1$ and to $2+2+1$; $3+2$ is adjacent also to $3+1+1$ and to $2+2+1$; etc. If *max_size* is 3, the partitions 5 and $4+1$ would not be included in the graph. If *max_parts* is 3, the partitions $2+1+1+1$ and $1+1+1+1+1$ would not be included.

If *max_parts* or *max_size* are zero, they are reset to be equal to n so that they make no restriction on the partitions.

If *directed* is nonzero, the graph will contain only directed arcs from partitions to their neighbors that have exactly one more part.

The special case when we want to generate all $p(n)$ partitions of the integer n can be obtained by calling *all_parts*(*n*, *directed*).

```
⟨ gb_basic.h 1 ⟩ +≡
#define all_parts(n, directed) parts((long)(n), 0L, 0L, (long)(directed))
```

55. The program for *parts* is very similar in structure to the program for *perms* already considered.

```
⟨ Basic subroutines 8 ⟩ +≡
  Graph *parts(n, max_parts, max_size, directed)
      unsigned long n;    /* the number being partitioned */
      unsigned long max_parts;    /* maximum number of parts */
      unsigned long max_size;    /* maximum size of each part */
      long directed;    /* should the graph be directed? */
  { ⟨ Vanilla local variables 9 ⟩
    if (max_parts ≡ 0 ∨ max_parts > n) max_parts = n;
    if (max_size ≡ 0 ∨ max_size > n) max_size = n;
    if (max_parts > MAX_D) panic(bad_specs);    /* too many parts allowed */
    ⟨ Create a graph with one vertex for each partition 56 ⟩;
    ⟨ Name the partitions and create the arcs or edges 57 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);    /* doggone it, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

56. The number of vertices is the coefficient of z^n in the z-binomial coefficient $\binom{m+p}{m}_z$, where $m =$ *max_parts* and $p =$ *max_size*. This coefficient is calculated as in the *perms* routine.

⟨ Create a graph with one vertex for each partition 56 ⟩ ≡

```
{ long nverts;   /* the number of vertices */
  register long *coef = gb_typed_alloc(n + 1, long, working_storage);
  if (gb_trouble_code) panic(no_room + 1);   /* can't allocate coef array */
  coef[0] = 1;
  for (k = 1; k ≤ max_parts; k++) {
    for (j = n, i = n − k − max_size; i ≥ 0; i−−, j−−) coef[j] −= coef[i];
    for (j = k, i = 0; j ≤ n; i++, j++) {
      coef[j] += coef[i];
      if (coef[j] > 1000000000) panic(very_bad_specs);   /* way too big */
    }
  }
  nverts = coef[n];
  gb_free(working_storage);   /* recycle the coef array */
  new_graph = gb_new_graph(nverts);
  if (new_graph ≡ Λ) panic(no_room);   /* out of memory before we're even started */
  sprintf(new_graph→id, "parts(%lu,%lu,%lu,%d)", n, max_parts, max_size, directed ? 1 : 0);
  strcpy(new_graph→util_types, "VVZZZZZZZZZZZZ");   /* hash table will be used */
}
```

This code is used in section 55.

57. As we generate the partitions, we maintain the numbers $\sigma_j = n - (x_1 + \cdots + x_{j-1}) = x_j + x_{j+1} + \cdots$, somewhat as we did in the *simplex* routine. We set $x_0 =$ *max_size* and $y_j =$ *max_parts* $+ 1 - j$; then when values $(x_1, \ldots, x_{j-1})$ are given, the conditions

$$\sigma_j / y_j \le x_j \le \sigma_j, \qquad x_j \le x_{j-1}$$

characterize the legal values of x_j.

⟨ Name the partitions and create the arcs or edges 57 ⟩ ≡

```
v = new_graph→vertices;
xx[0] = max_size; sig[1] = n;
for (k = max_parts, s = 1; k > 0; k−−, s++) yy[k] = s;
if (max_size * max_parts ≥ n) {
  k = 1; xx[1] = (n − 1)/max_parts + 1;   /* ⌈n/max_parts⌉ */
  while (1) {
    ⟨ Complete the partial solution (x_1, ..., x_k) 58 ⟩;
    ⟨ Assign the name x_1 + ··· + x_d to vertex v 60 ⟩;
    ⟨ Create arcs or edges from v to previous partitions 61 ⟩;
    v++;
    ⟨ Advance to the next partial solution (x_1, ..., x_k), where k is as large as possible; goto last if there
        are no more solutions 59 ⟩;
  }
}
last: if (v ≠ new_graph→vertices + new_graph→n) panic(impossible);   /* can't happen */
```

This code is used in section 55.

58. ⟨ Complete the partial solution $(x_1, \ldots, x_k)$ 58 ⟩ ≡

```
for (s = sig[k] − xx[k], k++; s; k++) {
   sig[k] = s;
   xx[k] = (s − 1)/yy[k] + 1;
   s −= xx[k];
}
d = k − 1;     /* the smallest part is x_d */
```

This code is used in section 57.

59. Here we seek the largest k such that x_k can be increased without violating the necessary and sufficient conditions stated earlier.

⟨ Advance to the next partial solution $(x_1, \ldots, x_k)$, where k is as large as possible; **goto** *last* if there are no more solutions 59 ⟩ ≡

```
if (d ≡ 1) goto last;
for (k = d − 1; ; k−−) {
   if (xx[k] < sig[k] ∧ xx[k] < xx[k − 1]) break;
   if (k ≡ 1) goto last;
}
xx[k]++;
```

This code is used in section 57.

60. ⟨ Assign the name $x_1 + \cdots + x_d$ to vertex v 60 ⟩ ≡

```
{ register char *p = buffer;     /* string pointer */
   for (k = 1; k ≤ d; k++) {
      sprintf(p, "+%ld", xx[k]);
      while (*p) p++;
   }
   v→name = gb_save_string(&buffer[1]);     /* omit buffer[0], which is '+' */
   hash_in(v);     /* enter v→name into the hash table (via utility fields u, v) */
}
```

This code is used in section 57.

61. Since we are generating the partitions in lexicographic order of their parts, it is reasonably easy to identify all adjacent vertices that precede the current setting of $(x_1, \ldots, x_d)$, by splitting x_j into two parts when $x_j \neq x_{j+1}$. We locate previous partitions via their symbolic names.

⟨ Create arcs or edges from v to previous partitions 61 ⟩ ≡

```
if (d < max_parts) {
   xx[d + 1] = 0;
   for (j = 1; j ≤ d; j++) {
      if (xx[j] ≠ xx[j + 1]) { long a, b;
         for (b = xx[j]/2, a = xx[j] − b; b; a++, b−−) ⟨ Generate a subpartition (n_1, ..., n_{d+1}) by splitting
               x_j into a + b, and make that subpartition adjacent to v 62 ⟩;
      }
      nn[j] = xx[j];
   }
}
```

This code is used in section 57.

62. The values of $(x_1,\ldots,x_{j-1})$ have already been copied into $(n_1,\ldots,n_{j-1})$. Our job is to copy the smaller parts $(x_{j+1},\ldots,x_d)$ while inserting a and b in their proper places, knowing that $a \ge b$.

⟨ Generate a subpartition $(n_1,\ldots,n_{d+1})$ by splitting x_j into $a+b$, and make that subpartition adjacent to v 62 ⟩ ≡

```
{ register Vertex *u;     /* previous vertex adjacent to v */
  register char *p = buffer;
  for (k = j + 1; xx[k] > a; k++) nn[k - 1] = xx[k];
  nn[k - 1] = a;
  for ( ; xx[k] > b; k++) nn[k] = xx[k];
  nn[k] = b;
  for ( ; k <= d; k++) nn[k + 1] = xx[k];
  for (k = 1; k <= d + 1; k++) {
    sprintf(p, "+%ld", nn[k]);
    while (*p) p++;
  }
  u = hash_out(&buffer[1]);
  if (u == Λ) panic(impossible + 2);     /* can't happen */
  if (directed) gb_new_arc(v, u, 1L);
  else gb_new_edge(v, u, 1L);
}
```

This code is used in section 61.

63. Binary tree graphs. The subroutine call *binary*(n, *max_height*, *directed*) creates a graph whose vertices represent the binary trees with n internal nodes and with all leaves at a distance that is at most *max_height* from the root. Two binary trees are adjacent in the graph if one can be obtained from the other by a single application of the associative law for binary operations, i.e., by replacing some subtree of the form $(\alpha\cdot\beta)\cdot\gamma$ by the subtree $\alpha\cdot(\beta\cdot\gamma)$. (This transformation on binary trees is often called a "rotation.") If the *directed* parameter is nonzero, the directed arcs go from a tree containing $(\alpha\cdot\beta)\cdot\gamma$ to a tree containing $\alpha\cdot(\beta\cdot\gamma)$ in its place; otherwise the graph is undirected. Each arc has length 1.

For example, the binary trees with three internal nodes form a circuit of length 5. They are

$$(a\cdot b)\cdot(c\cdot d),\quad a\cdot(b\cdot(c\cdot d)),\quad a\cdot((b\cdot c)\cdot d),\quad (a\cdot(b\cdot c))\cdot d,\quad ((a\cdot b)\cdot c)\cdot d,$$

if we use infix notation and name the leaves (a,b,c,d) from left to right. Here each tree is related to its two neighbors by associativity. The first and last trees are also related in the same way.

If *max_height* $=0$, it is changed to n, which means there is no restriction on the height of a leaf. In this case the graph will have exactly $\binom{2n+1}{n}/(2n+1)$ vertices; furthermore, each vertex will have exactly $n-1$ neighbors, because a rotation will be possible just above every internal node except the root. The graph in this case can also be interpreted geometrically: The vertices are in one-to-one correspondence with the triangulations of a regular $(n+2)$-gon; two triangulations are adjacent if and only if one is obtained from the other by replacing the pair of adjacent triangles ABC, DCB by the pair ADC, BDA.

The partial ordering corresponding to the directed graph on $\binom{2n+1}{n}/(2n+1)$ vertices created by *all_trees*$(n,1)$ is a lattice with interesting properties. See Huang and Tamari, *Journal of Combinatorial Theory* **A13** (1972), 7–13.

⟨ `gb_basic.h` 1 ⟩ +≡
#define *all_trees*(n, *directed*) *binary*((**long**)(n), $0_{\rm L}$, (**long**)(*directed*))

64. The program for *binary* is very similar in structure to the program for *parts* already considered. But the details are more exciting.

⟨ Basic subroutines 8 ⟩ +≡

```
Graph *binary(n, max_height, directed)
    unsigned long n;    /* the number of internal nodes */
    unsigned long max_height;    /* maximum height of a leaf */
    long directed;    /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  if (2 * n + 2 > BUF_SIZE) panic(bad_specs);    /* n is too huge for us */
  if (max_height ≡ 0 ∨ max_height > n) max_height = n;
  if (max_height > 30) panic(very_bad_specs);    /* more than a billion vertices */
  ⟨ Create a graph with one vertex for each binary tree 65 ⟩;
  ⟨ Name the trees and create the arcs or edges 67 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* uff da, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

65. The number of vertices is the coefficient of z^n in the power series G_h, where $h = max\_height$ and G_h satisfies the recurrence

$$G_0 = 1, \qquad G_{h+1} = 1 + zG_h^2.$$

The coefficients of G_5 are ≤ 55308, but the coefficients of G_6 are much larger; they exceed one billion when $28 \le n \le 49$, and they exceed one million when $17 \le n \le 56$. In order to avoid overflow during this calculation, we use a special method when $h \ge 6$ and $n \ge 20$: In such cases, graphs of reasonable size arise only if $n \ge 2^h - 7$, and we look at the coefficient of $z^{-(2^h-1-n)}$ in $R_h = G_h/z^{2^h-1}$, which is a power series in z^{-1} defined by the recurrence

$$R_0 = 1, \qquad R_{h+1} = R_h^2 + z^{1-2^{h+1}}.$$

⟨ Create a graph with one vertex for each binary tree 65 ⟩ ≡

```
{ long nverts;      /* the number of vertices */
  if (n ≥ 20 ∧ max_height ≥ 6) ⟨ Compute nverts using the R series 66 ⟩
  else {
    nn[0] = nn[1] = 1;
    for (k = 2; k ≤ n; k++) nn[k] = 0;
    for (j = 2; j ≤ max_height; j++)
      for (k = n − 1; k; k−−) {
        for (s = 0, i = k; i ≥ 0; i−−) s += nn[i] ∗ nn[k − i];      /* overflow impossible */
        nn[k + 1] = s;
      }
    nverts = nn[n];
  }
  new_graph = gb_new_graph(nverts);
  if (new_graph ≡ Λ) panic(no_room);      /* out of memory before we're even started */
  sprintf(new_graph→id, "binary(%lu,%lu,%d)", n, max_height, directed ? 1 : 0);
  strcpy(new_graph→util_types, "VVZZZZZZZZZZZZ");      /* hash table will be used */
}
```

This code is used in section 64.

66. The smallest nontrivial graph that is unilaterally disallowed by this procedure on the grounds of size limitations occurs when *max_height* $= 6$ and $n = 20$; it has 14,162,220 vertices.

⟨ Compute *nverts* using the R series 66 ⟩ ≡

```
{ register float ss;
  d = (1L << max_height) − 1 − n;
  if (d > 8) panic(bad_specs + 1);    /* too many vertices */
  if (d < 0) nverts = 0;
  else {
    nn[0] = nn[1] = 1;
    for (k = 2; k ≤ d; k++) nn[k] = 0;
    for (j = 2; j ≤ max_height; j++) {
      for (k = d; k; k−−) {
        for (ss = 0.0, i = k; i ≥ 0; i−−) ss += ((float) nn[i]) ∗ ((float) nn[k − i]);
        if (ss > MAX_NNN) panic(very_bad_specs + 1);    /* way too big */
        for (s = 0, i = k; i ≥ 0; i−−) s += nn[i] ∗ nn[k − i];    /* overflow impossible */
        nn[k] = s;
      }
      i = (1L << j) − 1;
      if (i ≤ d) nn[i]++;    /* add z^{1−2^j} */
    }
    nverts = nn[d];
  }
}
```

This code is used in section 65.

67. We generate the trees in lexicographic order of their Polish prefix notation, encoded in binary notation as $x_0x_1\ldots x_{2n}$, using '1' for an internal node and '0' for a leaf. For example, the five trees when $n=3$ are

$$1010100,\quad 1011000,\quad 1100100,\quad 1101000,\quad 1110000,$$

in lexicographic order. The algorithm for lexicographic generation maintains three auxiliary arrays l_j, y_j, and σ_j, where

$$\sigma_j \;=\; n-j+\sum_{i=0}^{j-1}x_i \;=\; -1+\sum_{i=j}^{2n}(1-x_i)$$

is one less than the number of 0's (leaves) in $(x_j,\ldots,x_{2n})$. The values of l_j and y_j are harder to describe formally; l_j is 2^{h-l} when $h=\mathit{max\_height}$ and when x_j represents a node at level l of the tree, based on the values of $(x_0,\ldots,x_{j-1})$. The value of y_j is a binary encoding of tree levels in which an internal node has not yet received a right child; y_j is also the maximum number of future leaves that can be produced by previously specified internal nodes without exceeding the maximum height. The number of 1-bits in y_j is the minimum number of future leaves, based on previous specifications.

Therefore if $\sigma_j>y_j$, x_j is forced to be 1. If $l_j=1$, x_j is forced to be 0. If the number of 1-bits of y_j is equal to σ_j, x_j is forced to be 0. Otherwise x_j can be either 0 or 1, and it will be possible to complete the partial solution $x_0\ldots x_j$ to a full Polish prefix code $x_0\ldots x_{2n}$.

For example, here are the arrays for one of the binary trees that is generated when $n=h=3$:

$$\begin{array}{rcccccccc}
j & = & 0 & 1 & 2 & 3 & 4 & 5 & 6\\
l_j & = & 8 & 4 & 2 & 2 & 1 & 1 & 4\\
y_j & = & 0 & 4 & 6 & 4 & 5 & 4 & 0\\
\sigma_j & = & 3 & 3 & 3 & 2 & 2 & 1 & 0\\
x_j & = & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}$$

If $x_j=1$ and $j<2n$, we have $l_{j+1}=l_j/2$, $y_{j+1}=y_j+l_{j+1}$, and $\sigma_{j+1}=\sigma_j$. If $x_j=0$ and $j<2n$, we have $l_{j+1}=2^t$, $y_{j+1}=y_j-2^t$, and $\sigma_{j+1}=\sigma_j-1$, where 2^t is the least power of 2 in the binary representation of y_j. It is not difficult to prove by induction that $\sigma_j<y_j+l_j$, assuming that $n<2^h$.

```
⟨ Name the trees and create the arcs or edges 67 ⟩ ≡
  { register long *xtab, *ytab, *ltab, *stab;
    ⟨ Initialize xtab, ytab, ltab, and stab; also set d = 2n 68 ⟩;
    v = new_graph→vertices;
    if (ltab[0] > n) {
      k = 0; xtab[0] = n ? 1 : 0;
      while (1) {
        ⟨ Complete the partial tree x0 ... xk 69 ⟩;
        ⟨ Assign a Polish prefix code name to vertex v 71 ⟩;
        ⟨ Create arcs or edges from v to previous trees 72 ⟩;
        v++;
        ⟨ Advance to the next partial tree x0 ... xk, where k is as large as possible; goto last if there are
            no more solutions 70 ⟩;
      }
    }
  }
last: if (v ≠ new_graph→vertices + new_graph→n) panic(impossible);     /* can't happen */
  gb_free(working_storage);
```

This code is used in section 64.

68. ⟨ Initialize *xtab*, *ytab*, *ltab*, and *stab*; also set $d = 2n$ 68 ⟩ ≡

```
xtab = gb_typed_alloc(8 * n + 4, long, working_storage);
if (gb_trouble_code) {      /* no room for xtab */
  gb_recycle(new_graph); panic(no_room + 2); }
d = n + n;
ytab = xtab + (d + 1);
ltab = ytab + (d + 1);
stab = ltab + (d + 1);
ltab[0] = 1L << max_height;
stab[0] = n;      /* ytab[0] = 0 */
```

This code is used in section 67.

69. ⟨ Complete the partial tree $x_0 \ldots x_k$ 69 ⟩ ≡

```
for (j = k + 1; j <= d; j++) {
  if (xtab[j - 1]) {
    ltab[j] = ltab[j - 1] >> 1;
    ytab[j] = ytab[j - 1] + ltab[j];
    stab[j] = stab[j - 1];
  } else {
    ytab[j] = ytab[j - 1] & (ytab[j - 1] - 1);      /* remove least significant 1-bit */
    ltab[j] = ytab[j - 1] - ytab[j];
    stab[j] = stab[j - 1] - 1;
  }
  if (stab[j] <= ytab[j]) xtab[j] = 0;
  else xtab[j] = 1;      /* this is the lexicographically smallest completion */
}
```

This code is used in section 67.

70. As in previous routines, we seek the largest k such that x_k can be increased without violating the necessary and sufficient conditions stated earlier.

⟨ Advance to the next partial tree $x_0 \ldots x_k$, where k is as large as possible; **goto** *last* if there are no more solutions 70 ⟩ ≡

```
for (k = d - 1; ; k--) {
  if (k <= 0) goto last;      /* this happens only when n <= 1 */
  if (xtab[k]) break;      /* find rightmost 1 */
}
for (k--; ; k--) {
  if (xtab[k] == 0 && ltab[k] > 1) break;
  if (k == 0) goto last;
}
xtab[k]++;
```

This code is used in section 67.

71. In the *name* field, we encode internal nodes of the binary tree by '`.`' and leaves by '`x`'. Thus the five trees shown above in binary code will be named

`.x.x.xx`, `.x..xxx`, `..xx.xx`, `..x.xxx`, `...xxxx`,

respectively.

⟨ Assign a Polish prefix code name to vertex v 71 ⟩ ≡

```
{ register char *p = buffer;    /* string pointer */
  for (k = 0; k ≤ d; k++, p++) *p = (xtab[k] ? '.' : 'x');
  v→name = gb_save_string(buffer);
  hash_in(v);    /* enter v→name into the hash table (via utility fields u, v) */
}
```

This code is used in section 67.

72. Since we are generating the trees in lexicographic order of their Polish prefix notation, it is relatively easy to find all pairs of trees that are adjacent via one application of the associative law: We simply replace a substring of the form `..`$\alpha\beta$ by `.`α`.`β, when α and β are Polish prefix strings. The result comes earlier in lexicographic order, so it will be an existing vertex unless it violates the *max_height* restriction.

⟨ Create arcs or edges from v to previous trees 72 ⟩ ≡

```
for (j = 0; j < d; j++)
  if (xtab[j] ≡ 1 ∧ xtab[j + 1] ≡ 1) {
    for (i = j + 1, s = 0; s ≥ 0; s += (xtab[i + 1] ≪ 1) − 1, i++) xtab[i] = xtab[i + 1];
    xtab[i] = 1;
    { register char *p = buffer;    /* string pointer */
      register Vertex *u;
      for (k = 0; k ≤ d; k++, p++) *p = (xtab[k] ? '.' : 'x');
      u = hash_out(buffer);
      if (u) {
        if (directed) gb_new_arc(v, u, 1L);
        else gb_new_edge(v, u, 1L);
      }
    }
    for (i−−; i > j; i−−) xtab[i + 1] = xtab[i];    /* restore xtab */
    xtab[i + 1] = 1;
  }
```

This code is used in section 67.

73. Complementing and copying. We have seen how to create a wide variety of basic graphs with the *board*, *simplex*, *subsets*, *perms*, *parts*, and *binary* procedures. The remaining routines of GB_BASIC are somewhat different. They transform existing graphs into new ones, thereby presenting us with an almost mind-boggling array of further possibilities.

The first of these transformations is perhaps the simplest. It complements a given graph, making vertices adjacent if and only if they were previously nonadjacent. More precisely, the subroutine call *complement*(g, *copy*, *self*, *directed*) returns a graph with the same vertices as g, but with complemented arcs. If $self \neq 0$, the new graph will have a self-loop from a vertex v to itself when the original graph did not; if $self = 0$, the new graph will have no self-loops. If $directed \neq 0$, the new graph will have an arc from u to v when the original graph did not; if $directed = 0$, the new graph will be undirected, and it will have an edge between u and v when the original graph did not. In the latter case, the original graph should also be undirected (that is, its arcs should come in pairs, as described in the *gb_new_edge* routine of GB_GRAPH).

If $copy \neq 0$, a double complement will actually be done. This means that the new graph will essentially be a copy of the old, except that duplicate arcs (and possibly self-loops) will be removed. Regardless of the value of *copy*, information that might have been present in the utility fields will not be copied, and arc lengths will all be set to 1.

One possibly useful feature of the graphs returned by *complement* is worth noting. The vertices adjacent to v, namely the list

$$v\to arcs\to tip,\quad v\to arcs\to next\to tip,\quad v\to arcs\to next\to next\to tip,\quad \ldots,$$

will be in strictly decreasing order (except in the case of an undirected self-loop, when v itself will appear twice in succession).

74. ⟨ Basic subroutines 8 ⟩ +≡

```
Graph *complement(g, copy, self, directed)
    Graph *g;      /* graph to be complemented */
    long copy;     /* should we double-complement? */
    long self;     /* should we produce self-loops? */
    long directed; /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  register long n;
  register Vertex *u;
  register siz_t delta;   /* difference in memory addresses */
  if (g ≡ Λ) panic(missing_operand);   /* where's g? */
  ⟨ Set up a graph with the vertices of g 75 ⟩;
  sprintf(buffer, ",%d,%d,%d)", copy ? 1 : 0, self ? 1 : 0, directed ? 1 : 0);
  make_compound_id(new_graph, "complement(", g, buffer);
  ⟨ Insert complementary arcs or edges 76 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);   /* worse luck, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

75. In several of the following routines, it is efficient to circumvent C's normal rules for pointer arithmetic, and to use the fact that the vertices of a graph being copied are a constant distance away in memory from the vertices of its clone.

#define *vert_offset*(*v*, *delta*) ((**Vertex** ∗)(((**siz_t**) *v*) + *delta*))

⟨Set up a graph with the vertices of *g* 75⟩ ≡

```
n = g→n;
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);    /* out of memory before we're even started */
delta = ((siz_t)(new_graph→vertices)) − ((siz_t)(g→vertices));
for (u = new_graph→vertices, v = g→vertices; v < g→vertices + n; u++, v++)
  u→name = gb_save_string(v→name);
```

This code is used in sections 74, 78, and 81.

76. A temporary utility field in the new graph is used to remember which vertices are adjacent to a given vertex in the old one. We stamp the *tmp* field of *v* with a pointer to *u* when there's an arc from *u* to *v*.

#define *tmp* *u.V* /∗ utility field *u* for temporary use as a vertex pointer ∗/

⟨Insert complementary arcs or edges 76⟩ ≡

```
for (v = g→vertices; v < g→vertices + n; v++) { register Vertex *vv;
  u = vert_offset(v, delta);    /* vertex in new_graph corresponding to v in g */
  { register Arc *a;
    for (a = v→arcs; a; a = a→next) vert_offset(a→tip, delta)→tmp = u;
  }
  if (directed) {
    for (vv = new_graph→vertices; vv < new_graph→vertices + n; vv++)
      if ((vv→tmp ≡ u ∧ copy) ∨ (vv→tmp ≠ u ∧ ¬copy))
        if (vv ≠ u ∨ self) gb_new_arc(u, vv, 1L);
  } else {
    for (vv = (self ? u : u + 1); vv < new_graph→vertices + n; vv++)
      if ((vv→tmp ≡ u ∧ copy) ∨ (vv→tmp ≠ u ∧ ¬copy)) gb_new_edge(u, vv, 1L);
  }
}
for (v = new_graph→vertices; v < new_graph→vertices + n; v++) v→tmp = Λ;
```

This code is used in section 74.

77. Graph union and intersection. Another simple way to get new graphs from old ones is to take the union or intersection of their sets of arcs. The subroutine call *gunion*(*g*, *gg*, *multi*, *directed*) produces a graph with the vertices and arcs of *g* together with the arcs of another graph *gg*. The subroutine call *intersection*(*g*, *gg*, *multi*, *directed*) produces a graph with the vertices of *g* but with only the arcs that appear in both *g* and *gg*. In both cases we assume that *gg* has the same vertices as *g*, in the sense that vertices in the same relative position from the beginning of the vertex array are considered identical. If the actual number of vertices in *gg* exceeds the number in *g*, the extra vertices and all arcs touching them in *gg* are suppressed.

The input graphs are assumed to be undirected, unless the *directed* parameter is nonzero. Peculiar results might occur if you mix directed and undirected graphs, but the subroutines will not "crash" when they are asked to produce undirected output from directed input.

If *multi* is nonzero, the new graph may have multiple edges: Suppose there are k_1 arcs from u to v in g and k_2 such arcs in gg. Then there will be $k_1 + k_2$ in the union and $\min(k_1, k_2)$ in the intersection when $multi \neq 0$, but at most one in the union or intersection when $multi = 0$.

The lengths of arcs are copied to the union graph when $multi \neq 0$; the minimum length of multiple arcs is retained in the union when $multi = 0$.

The lengths of arcs in the intersection graph are a bit trickier. If multiple arcs occur in g, their minimum length, l, is computed. Then we compute the maximum of l and the lengths of corresponding arcs in gg. If $multi = 0$, only the minimum of those maxima will survive.

78. ⟨ Basic subroutines 8 ⟩ +≡

```
Graph *gunion(g, gg, multi, directed)
    Graph *g, *gg;    /* graphs to be united */
    long multi;    /* should we reproduce multiple arcs? */
    long directed;    /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  register long n;
  register Vertex *u;
  register siz_t delta, ddelta;    /* differences in memory addresses */
  if (g ≡ Λ ∨ gg ≡ Λ) panic(missing_operand);    /* where are g and gg? */
  ⟨ Set up a graph with the vertices of g 75 ⟩;
  sprintf(buffer, ",%d,%d)", multi ? 1 : 0, directed ? 1 : 0);
  make_double_compound_id(new_graph, "gunion(", g, ",", gg, buffer);
  ddelta = ((siz_t)(new_graph→vertices)) − ((siz_t)(gg→vertices));
  ⟨ Insert arcs or edges present in either g or gg 79 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* uh oh, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

79. ⟨ Insert arcs or edges present in either g or gg 79 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + n; v++) {
  register Arc *a;
  register Vertex *vv = vert_offset(v, delta);   /* vertex in new_graph corresponding to v in g */
  register Vertex *vvv = vert_offset(vv, −ddelta);   /* vertex in gg corresponding to v in g */
  for (a = v→arcs; a; a = a→next) {
    u = vert_offset(a→tip, delta);
    ⟨ Insert a union arc or edge from vv to u, if appropriate 80 ⟩;
  }
  if (vvv < gg→vertices + gg→n)
    for (a = vvv→arcs; a; a = a→next) {
      u = vert_offset(a→tip, ddelta);
      if (u < new_graph→vertices + n) ⟨ Insert a union arc or edge from vv to u, if appropriate 80 ⟩;
    }
}
for (v = new_graph→vertices; v < new_graph→vertices + n; v++) v→tmp = Λ, v→tlen = Λ;
```

This code is used in section 78.

80. We use the *tmp* trick of *complement* to remember which arcs have already been recorded from u, and we extend it so that we can maintain minimum lengths. Namely, *uu→tmp* will equal u if and only if we have already seen an arc from u to uu; and if so, *uu→tlen* will be one such arc. In the undirected case, *uu→tlen* will point to the first arc of an edge pair that touches u.

The only thing slightly nontrivial here is the way we keep undirected edges grouped in pairs. We generate a new edge from vv to u only if $vv \le u$, and if equality holds we advance a so that we don't see the self-loop in both directions. Similar logic will be repeated in many of the programs below.

```
#define tlen z.A   /* utility field z regarded as a pointer to an arc */
```

⟨ Insert a union arc or edge from vv to u, if appropriate 80 ⟩ ≡

```
{ register Arc *b;
  if (directed) {
    if (multi ∨ u→tmp ≠ vv) gb_new_arc(vv, u, a→len);
    else {
      b = u→tlen;
      if (a→len < b→len) b→len = a→len;
    }
    u→tmp = vv;   /* remember that we've seen this */
    u→tlen = vv→arcs;
  } else if (u ≥ vv) {
    if (multi ∨ u→tmp ≠ vv) gb_new_edge(vv, u, a→len);
    else {
      b = u→tlen;
      if (a→len < b→len) b→len = (b + 1)→len = a→len;
    }
    u→tmp = vv;
    u→tlen = vv→arcs;
    if (u ≡ vv ∧ a→next ≡ a + 1) a++;   /* bypass second half of self-loop */
  }
}
```

This code is used in section 79.

81. ⟨ Basic subroutines 8 ⟩ +≡

```
Graph *intersection(g, gg, multi, directed)
    Graph *g, *gg;    /* graphs to be intersected */
    long multi;    /* should we reproduce multiple arcs? */
    long directed;    /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  register long n;
  register Vertex *u;
  register siz_t delta, ddelta;    /* differences in memory addresses */
  if (g ≡ Λ ∨ gg ≡ Λ) panic(missing_operand);    /* where are g and gg? */
  ⟨ Set up a graph with the vertices of g 75 ⟩;
  sprintf(buffer, ",%d,%d)", multi ? 1 : 0, directed ? 1 : 0);
  make_double_compound_id(new_graph, "intersection(", g, ",", gg, buffer);
  ddelta = ((siz_t)(new_graph→vertices)) − ((siz_t)(gg→vertices));
  ⟨ Insert arcs or edges present in both g and gg 82 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* whoops, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

82. Two more temporary utility fields are needed here.

```
#define mult v.I    /* utility field v, counts multiplicity of arcs */
#define minlen w.I    /* utility field w, records the smallest length */
```

```
⟨ Insert arcs or edges present in both g and gg 82 ⟩ ≡
  for (v = g→vertices; v < g→vertices + n; v++) { register Arc *a;
    register Vertex *vv = vert_offset(v, delta);    /* vertex in new_graph corresponding to v in g */
    register Vertex *vvv = vert_offset(vv, −ddelta);    /* vertex in gg corresponding to v in g */
    if (vvv ≥ gg→vertices + gg→n) continue;
    ⟨ Take note of all arcs from v 85 ⟩;
    for (a = vvv→arcs; a; a = a→next) {
      u = vert_offset(a→tip, ddelta);
      if (u ≥ new_graph→vertices + n) continue;
      if (u→tmp ≡ vv) { long l = u→minlen;
        if (a→len > l) l = a→len;    /* maximum */
        if (u→mult < 0) ⟨ Update minimum of multiple maxima 84 ⟩
        else ⟨ Generate a new arc or edge for the intersection, and reduce the multiplicity 83 ⟩;
      }
    }
  }
  ⟨ Clear out the temporary utility fields 86 ⟩;
```

This code is used in section 81.

83. ⟨ Generate a new arc or edge for the intersection, and reduce the multiplicity 83 ⟩ ≡

```
{
  if (directed) gb_new_arc(vv, u, l);
  else {
    if (vv ≤ u) gb_new_edge(vv, u, l);
    if (vv ≡ u ∧ a→next ≡ a + 1) a++;    /* skip second half of self-loop */
  }
  if (¬multi) {
    u→tlen = vv→arcs;
    u→mult = −1;
  } else if (u→mult ≡ 0) u→tmp = Λ;
  else u→mult−−;
}
```

This code is used in section 82.

84. We get here if and only if *multi* = 0 and *gg* has more than one arc from *vv* to *u* and *g* has at least one arc from *vv* to *u*.

⟨ Update minimum of multiple maxima 84 ⟩ ≡

```
{ register Arc *b = u→tlen;    /* previous arc or edge from vv to u */
  if (l < b→len) {
    b→len = l;
    if (¬directed) (b + 1)→len = l;
  }
}
```

This code is used in section 82.

85. ⟨ Take note of all arcs from v 85 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  u = vert_offset(a→tip, delta);
  if (u→tmp ≡ vv) {
    u→mult++;
    if (a→len < u→minlen) u→minlen = a→len;
  } else u→tmp = vv, u→mult = 0, u→minlen = a→len;
  if (u ≡ vv ∧ ¬directed ∧ a→next ≡ a + 1) a++;    /* skip second half of self-loop */
}
```

This code is used in section 82.

86. ⟨ Clear out the temporary utility fields 86 ⟩ ≡

```
for (v = new_graph→vertices; v < new_graph→vertices + n; v++) {
  v→tmp = Λ;
  v→tlen = Λ;
  v→mult = 0;
  v→minlen = 0;
}
```

This code is used in section 82.

87. Line graphs. The next operation in GB_BASIC's repertoire constructs the so-called line graph of a given graph g. The subroutine that does this is invoked by calling '$lines(g, directed)$'.

If $directed = 0$, the line graph has one vertex for each edge of g; two vertices are adjacent if and only if the corresponding edges have a common vertex.

If $directed \neq 0$, the line graph has one vertex for each arc of g; there is an arc from vertex u to vertex v if and only if the arc corresponding to u ends at the vertex that begins the arc corresponding to v.

All arcs of the line graph will have length 1.

Utility fields $u.V$ and $v.V$ of each vertex in the line graph will point to the vertices of g that define the corresponding arc or edge, and $w.A$ will point to the arc from $u.V$ to $v.V$ in g. In the undirected case we will have $u.V \le v.V$.

```
⟨ Basic subroutines 8 ⟩ +≡
  Graph *lines(g, directed)
      Graph *g;      /* graph whose lines will become vertices */
      long directed;      /* should the graph be directed? */
  { ⟨ Vanilla local variables 9 ⟩
    register long m;      /* the number of lines */
    register Vertex *u;
    if (g ≡ Λ) panic(missing_operand);      /* where is g? */
    ⟨ Set up a graph whose vertices are the lines of g 89 ⟩;
    if (directed) ⟨ Insert arcs of a directed line graph 92 ⟩
    else ⟨ Insert edges of an undirected line graph 93 ⟩;
    ⟨ Restore g to its pristine original condition 88 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);      /* (sigh) we ran out of memory somewhere back there */
    }
    return new_graph;
  near_panic: ⟨ Recover from potential disaster due to bad data 90 ⟩;
  }
```

88. We want to add a data structure to g so that the line graph can be built efficiently. But we also want to preserve g so that it exhibits no traces of occupation when *lines* has finished its work. To do this, we will move utility field $v\rightarrow z$ temporarily into a utility field $u\rightarrow z$ of the line graph, where u is the first vertex having $u\rightarrow u.V \equiv v$, whenever such a u exists. Then we'll set $v\rightarrow map = u$. We will then be able to find u when v is given, and we'll be able to cover our tracks later.

In the undirected case, further structure is needed. We will temporarily change the *tip* field in the second arc of each edge pair so that it points to the line-graph vertex that points to the first arc of the pair.

The *util_types* field of the graph does not indicate the fact that utility fields $u.V$, $v.V$, and $w.A$ of each vertex will be set, because those utility fields are pointers from the new graph to the original graph. The *save_graph* procedure does not deal with pointers between graphs.

```
#define map z.V      /* the z field treated as a vertex pointer */

⟨ Restore g to its pristine original condition 88 ⟩ ≡
  for (u = new_graph→vertices, v = Λ; u < new_graph→vertices + m; u++) {
    if (u→u.V ≠ v) {
      v = u→u.V;      /* original vertex of g */
      v→map = u→map;      /* restore original value of v→z */
      u→map = Λ;
    }
    if (¬directed) ((u→w.A) + 1)→tip = v;
  }
```

This code is used in sections 87 and 90.

89. Special care must be taken to avoid chaos when the user is trying to construct the undirected line graph of a directed graph. Otherwise we might trash the memory, or we might leave the original graph in a garbled state with pointers leading into supposedly free space.

⟨ Set up a graph whose vertices are the lines of g 89 ⟩ ≡

```
m = (directed ? g→m : (g→m)/2);
new_graph = gb_new_graph(m);
if (new_graph ≡ Λ) panic(no_room);     /* out of memory before we're even started */
make_compound_id(new_graph, "lines(", g, directed ? ",1)" : ",0)");
u = new_graph→vertices;
for (v = g→vertices + g→n − 1; v ≥ g→vertices; v−−) { register Arc *a;
  register long mapped = 0;     /* has v→map been set? */
  for (a = v→arcs; a; a = a→next) { register Vertex *vv = a→tip;
    if (¬directed) {
      if (vv < v) continue;
      if (vv ≥ g→vertices + g→n) goto near_panic;     /* original graph not undirected */
    }
    ⟨ Make u a vertex representing the arc a from v to vv 91 ⟩;
    if (¬mapped) {
      u→map = v→map;     /* z.V = map incorporates all bits of utility field z, whatever its type */
      v→map = u;
      mapped = 1;
    }
    u++;
  }
}
if (u ≠ new_graph→vertices + m) goto near_panic;
```

This code is used in section 87.

90. ⟨ Recover from potential disaster due to bad data 90 ⟩ ≡

```
m = u − new_graph→vertices;
⟨ Restore g to its pristine original condition 88 ⟩;
gb_recycle(new_graph);
panic(invalid_operand);     /* g did not obey the conventions for an undirected graph */
```

This code is used in section 87.

91. The vertex names in the line graph are pairs of original vertex names, separated by '`--`' when undirected, '`->`' when directed. If either of the original names is horrendously long, the villainous Procrustes chops it off arbitrarily so that it fills at most half of the name buffer.

⟨ Make u a vertex representing the arc a from v to vv 91 ⟩ ≡

```
u→u.V = v;
u→v.V = vv;
u→w.A = a;
if (¬directed) {
  if (u ≥ new_graph→vertices + m ∨ (a + 1)→tip ≠ v) goto near_panic;
  if (v ≡ vv ∧ a→next ≡ a + 1) a++;     /* skip second half of self-loop */
  else (a + 1)→tip = u;
}
sprintf(buffer, "%.*s-%c%.*s", (BUF_SIZE − 3)/2, v→name,
    directed ? '>' : '-', BUF_SIZE/2 − 1, vv→name);
u→name = gb_save_string(buffer);
```

This code is used in section 89.

92. ⟨ Insert arcs of a directed line graph 92 ⟩ ≡

```
for (u = new_graph→vertices; u < new_graph→vertices + m; u++) {
  v = u→v.V;
  if (v→arcs) {      /* v→map has been set up */
    v = v→map;
    do {
      gb_new_arc(u, v, 1L);
      v++;
    } while (v→u.V ≡ u→v.V);
  }
}
```

This code is used in section 87.

93. An undirected line graph will contain no self-loops. It contains multiple edges only if the original graph did; in that case, there are two edges joining a line to each of its parallel mates, because each mate hits both of its endpoints.

The details of this section deserve careful study. We use the fact that the first vertices of the lines occur in nonincreasing order.

⟨ Insert edges of an undirected line graph 93 ⟩ ≡

```
for (u = new_graph→vertices; u < new_graph→vertices + m; u++) { register Vertex *vv;
  register Arc *a; register long mapped = 0;
  v = u→u.V;      /* we look first for prior lines that touch the first vertex */
  for (vv = v→map; vv < u; vv++) gb_new_edge(u, vv, 1L);
  v = u→v.V;      /* then we look for prior lines that touch the other one */
  for (a = v→arcs; a; a = a→next) {
    vv = a→tip;
    if (vv < u ∧ vv ≥ new_graph→vertices) gb_new_edge(u, vv, 1L);
    else if (vv ≥ v ∧ vv < g→vertices + g→n) mapped = 1;
  }
  if (mapped ∧ v > u→u.V)
    for (vv = v→map; vv→u.V ≡ v; vv++) gb_new_edge(u, vv, 1L);
}
```

This code is used in section 87.

94. Graph products. Three ways have traditionally been used to define the product of two graphs. In all three cases the vertices of the product graph are ordered pairs (v, v'), where v and v' are vertices of the original graphs; the difference occurs in the definition of arcs. Suppose g has m arcs and n vertices, while g' has m' arcs and n' vertices. The *cartesian product* of g and g' has $mn' + m'n$ arcs, namely from (u, u') to (v, u') whenever there's an arc from u to v in g, and from (u, u') to (u, v') whenever there's an arc from u' to v' in g'. The *direct product* has mm' arcs, namely from (u, u') to (v, v') in the same circumstances. The *strong product* has both the arcs of the cartesian product and the direct product.

Notice that an undirected graph with m edges has $2m$ arcs. Thus the number of edges in the direct product of two undirected graphs is twice the product of the number of edges in the individual graphs. A self-loop in g will combine with an edge in g' to make two parallel edges in the direct product.

The subroutine call *product*(*g*, *gg*, *type*, *directed*) produces the product graph of one of these three types, where $type = 0$ for cartesian product, $type = 1$ for direct product, and $type = 2$ for strong product. The length of an arc in the cartesian product is copied from the length of the original arc that it replicates; the length of an arc in the direct product is the minimum of the two arc lengths that induce it. If $directed = 0$, the product graph will be an undirected graph with edges consisting of consecutive arc pairs according to the standard GraphBase conventions, and the input graphs should adhere to the same conventions.

```
⟨gb_basic.h 1⟩ +≡
#define cartesian 0
#define direct 1
#define strong 2
```

95. ⟨Basic subroutines 8⟩ +≡

```
Graph *product(g, gg, type, directed)
    Graph *g, *gg;   /* graphs to be multiplied */
    long type;   /* cartesian, direct, or strong */
    long directed;   /* should the graph be directed? */
{ ⟨Vanilla local variables 9⟩
  register Vertex *u, *vv;
  register long n;   /* the number of vertices in the product graph */
  if (g ≡ Λ ∨ gg ≡ Λ) panic(missing_operand);   /* where are g and gg? */
  ⟨Set up a graph with ordered pairs of vertices 96⟩;
  if ((type & 1) ≡ 0) ⟨Insert arcs or edges for cartesian product 97⟩;
  if (type) ⟨Insert arcs or edges for direct product 99⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);   /* @¿*#!, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

96. We must be constantly on guard against running out of memory, especially when multiplying information.

The vertex names in the product are pairs of original vertex names separated by commas. Thus, for example, if you cross an *econ* graph with a *roget* graph, you can get vertices like `"Financial␣services,\␣mediocrity"`.

⟨ Set up a graph with ordered pairs of vertices 96 ⟩ ≡

```
{ float test_product = ((float)(g→n)) ∗ ((float)(gg→n));
  if (test_product > MAX_NNN) panic(very_bad_specs);   /∗ way too many vertices ∗/
}
n = (g→n) ∗ (gg→n);
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);   /∗ out of memory before we're even started ∗/
for (u = new_graph→vertices, v = g→vertices, vv = gg→vertices;
        u < new_graph→vertices + n; u++) {
  sprintf(buffer, "%.*s,%.*s", BUF_SIZE/2 − 1, v→name, (BUF_SIZE − 1)/2, vv→name);
  u→name = gb_save_string(buffer);
  if (++vv ≡ gg→vertices + gg→n) vv = gg→vertices, v++;   /∗ "carry" ∗/
}
sprintf(buffer, ",%d,%d)", (type ? 2 : 0) − (int)(type & 1), directed ? 1 : 0);
make_double_compound_id(new_graph, "product(", g, ",", gg, buffer);
```

This code is used in section 95.

97. ⟨ Insert arcs or edges for cartesian product 97 ⟩ ≡

```
{ register Vertex ∗uu, ∗uuu;
  register Arc ∗a;
  register siz_t delta;   /∗ difference in memory addresses ∗/
  delta = ((siz_t)(new_graph→vertices)) − ((siz_t)(gg→vertices));
  for (u = gg→vertices; u < gg→vertices + gg→n; u++)
    for (a = u→arcs; a; a = a→next) {
      v = a→tip;
      if (¬directed) {
        if (u > v) continue;
        if (u ≡ v ∧ a→next ≡ a + 1) a++;   /∗ skip second half of self-loop ∗/
      }
      for (uu = vert_offset(u, delta), vv = vert_offset(v, delta);
              uu < new_graph→vertices + n; uu += gg→n, vv += gg→n)
        if (directed) gb_new_arc(uu, vv, a→len);
        else gb_new_edge(uu, vv, a→len);
    }
  ⟨ Insert arcs or edges for first component of cartesian product 98 ⟩;
}
```

This code is used in section 95.

98. ⟨Insert arcs or edges for first component of cartesian product 98⟩ ≡

```
for (u = g->vertices, uu = new_graph->vertices; uu < new_graph->vertices + n; u++, uu += gg->n)
  for (a = u->arcs; a; a = a->next) {
    v = a->tip;
    if (!directed) {
      if (u > v) continue;
      if (u == v && a->next == a + 1) a++;      /* skip second half of self-loop */
    }
    vv = new_graph->vertices + ((gg->n) * (v - g->vertices));
    for (uuu = uu; uuu < uu + gg->n; uuu++, vv++)
      if (directed) gb_new_arc(uuu, vv, a->len);
      else gb_new_edge(uuu, vv, a->len);
  }
```

This code is used in section 97.

99. ⟨Insert arcs or edges for direct product 99⟩ ≡

```
{ Vertex *uu; Arc *a;
  siz_t delta0 = ((siz_t)(new_graph->vertices)) - ((siz_t)(gg->vertices));
  siz_t del = (gg->n) * sizeof(Vertex);
  register siz_t delta, ddelta;

  for (uu = g->vertices, delta = delta0; uu < g->vertices + g->n; uu++, delta += del)
    for (a = uu->arcs; a; a = a->next) {
      vv = a->tip;
      if (!directed) {
        if (uu > vv) continue;
        if (uu == vv && a->next == a + 1) a++;      /* skip second half of self-loop */
      }
      ddelta = delta0 + del * (vv - g->vertices);
      for (u = gg->vertices; u < gg->vertices + gg->n; u++) { register Arc *aa;
        for (aa = u->arcs; aa; aa = aa->next) { long length = a->len;
          if (length > aa->len) length = aa->len;
          v = aa->tip;
          if (directed) gb_new_arc(vert_offset(u, delta), vert_offset(v, ddelta), length);
          else gb_new_edge(vert_offset(u, delta), vert_offset(v, ddelta), length);
        }
      }
    }
}
```

This code is used in section 95.

100. Induced graphs. Another important way to transform a graph is to remove, identify, or split some of its vertices. All of these operations are performed by the *induced* routine, which users can invoke by calling '*induced*(g, *description*, *self*, *multi*, *directed*)'.

Each vertex v of g should first be assigned an "induction code" in its field $v\to ind$, which is actually utility field z. The induction code is 0 if v is to be eliminated; it is 1 if v is to be retained; it is $k > 1$ if v is to be split into k nonadjacent vertices having the same neighbors as v did; and it is $k < 0$ if v is to be identified with all other vertices having the same value of k.

For example, suppose g is a circuit with vertices $\{0, 1, \ldots, 9\}$, where j is adjacent to k if and only if $k = (j \pm 1) \bmod 10$. If we set

$$\begin{array}{l} 0\to ind = 0, \quad 1\to ind = 5\to ind = 9\to ind = -1, \quad 2\to ind = 3\to ind = -2, \\ 4\to ind = 6\to ind = 8\to ind = 1, \quad \text{and } 7\to ind = 3, \end{array}$$

the induced graph will have vertices $\{-1, -2, 4, 6, 7, 7', 7'', 8\}$. The vertices adjacent to 6, say, will be -1 (formerly 5), 7, $7'$, and $7''$. The vertices adjacent to -1 will be those formerly adjacent to 1, 5, or 9, namely -2 (formerly 2), 4, 6, and 8. The vertices adjacent to -2 will be those formerly adjacent to 2 or 3, namely -1 (formerly 1), -2 (formerly 3), -2 (formerly 2), and 4. Duplicate edges will be discarded if $multi \equiv 0$, and self-loops will be discarded if $self \equiv 0$.

The total number of vertices in the induced graph will be the sum of the positive *ind* fields plus the absolute value of the most negative *ind* field. This rule implies, for example, that if at least one vertex has $ind = -5$, the induced graph will always have a vertex -4, even though no *ind* field has been set to -4.

The *description* parameter is a string that will appear as part of the name of the induced graph; if $description = 0$, this string will be empty. In the latter case, users are encouraged to assign a suitable name to the *id* field of the induced graph themselves, characterizing the method by which the *ind* codes were set.

If the *directed* parameter is zero, the input graph will be assumed to be undirected, and the output graph will be undirected.

When $multi = 0$, the length of an arc that represents multiple arcs will be the minimum of the multiple arc lengths.

#define *ind* $z.I$

⟨ gb_basic.h 1 ⟩ +≡
#define *ind* $z.I$ /* utility field z when used to induce a graph */

101. Here's a simple example: To get a complete bipartite graph with parts of sizes *n1* and *n2*, we can start with a trivial two-point graph and split its vertices into *n1* and *n2* parts.

```
⟨ Applications of basic subroutines 101 ⟩ ≡
  Graph *bi_complete(n1, n2, directed)
      unsigned long n1;   /* size of first part */
      unsigned long n2;   /* size of second part */
      long directed;   /* should all arcs go from first part to second? */
  { Graph *new_graph = board(2L, 0L, 0L, 0L, 1L, 0L, directed);
    if (new_graph) {
      new_graph→vertices→ind = n1;
      (new_graph→vertices + 1)→ind = n2;
      new_graph = induced(new_graph, Λ, 0L, 0L, directed);
      if (new_graph) {
        sprintf(new_graph→id, "bi_complete(%lu,%lu,%d)",
            n1, n2, directed ? 1 : 0);
        mark_bipartite(new_graph, n1);
      }
    }
    return new_graph;
  }
```

See also section 103.

This code is used in section 2.

102. The *induced* routine also provides a special feature not mentioned above: If the *ind* field of any vertex *v* is `IND_GRAPH` or greater (where `IND_GRAPH` is a large constant, much larger than the number of vertices that would fit in computer memory), then utility field *v→subst* should point to a graph. A copy of the vertices of that graph will then be substituted for *v* in the induced graph.

This feature extends the ordinary case when $v\to ind > 0$, which essentially substitutes an empty graph for v.

If substitution is being used to replace all of g's vertices by disjoint copies of some other graph g', the induced graph will be somewhat similar to a product graph. But it will not be the same as any of the three types of output produced by *product*, because the relation between g and g' is not symmetrical. Assuming that no self-loops are present, and that graphs (g, g') have respectively (m, m') arcs and (n, n') vertices, the result of substituting g' for all vertices of g has $m'n + mn'^2$ arcs.

```
#define IND_GRAPH 1000000000   /* when ind is a billion or more, */
#define subst y.G   /* we'll look at the subst field */

⟨ gb_basic.h 1 ⟩ +≡
#define IND_GRAPH 1000000000
#define subst y.G
```

103. For example, we can use the `IND_GRAPH` feature to create a "wheel" of n vertices arranged cyclically, all connected to one or more center points. In the directed case, the arcs will run from the center(s) to a cycle; in the undirected case, the edges will join the center(s) to a circuit.

⟨ Applications of basic subroutines 101 ⟩ +≡

```
Graph *wheel(n, n1, directed)
    unsigned long n;    /* size of the rim */
    unsigned long n1;    /* number of center points */
    long directed;    /* should all arcs go from center to rim and around? */
{ Graph *new_graph = board(2L, 0L, 0L, 0L, 1L, 0L, directed);    /* trivial 2-vertex graph */
  if (new_graph) {
    new_graph→vertices→ind = n1;
    (new_graph→vertices + 1)→ind = IND_GRAPH;
    (new_graph→vertices + 1)→subst = board(n, 0L, 0L, 0L, 1L, 1L, directed);    /* cycle or circuit */
    new_graph = induced(new_graph, Λ, 0L, 0L, directed);
    if (new_graph) {
      sprintf(new_graph→id, "wheel(%lu,%lu,%d)",
          n, n1, directed ? 1 : 0);
    }
  }
  return new_graph;
}
```

104. ⟨ `gb_basic.h` 1 ⟩ +≡

```
extern Graph *bi_complete();
extern Graph *wheel();    /* standard applications of induced */
```

105. ⟨ Basic subroutines 8 ⟩ +≡

```
Graph *induced(g, description, self, multi, directed)
    Graph *g;    /* graph marked for induction in its ind fields */
    char *description;    /* string to be mentioned in new_graph→id */
    long self;    /* should self-loops be permitted? */
    long multi;    /* should multiple arcs be permitted? */
    long directed;    /* should the graph be directed? */
{ ⟨ Vanilla local variables 9 ⟩
  register Vertex *u;
  register long n = 0;    /* total number of vertices in induced graph */
  register long nn = 0;    /* number of negative vertices in induced graph */
  if (g ≡ Λ) panic(missing_operand);    /* where is g? */
  ⟨ Set up a graph with the induced vertices 106 ⟩;
  ⟨ Insert arcs or edges for induced vertices 110 ⟩;
  ⟨ Restore g to its original state 109 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* aargh, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

106. ⟨ Set up a graph with the induced vertices 106 ⟩ ≡

```
⟨ Determine n and nn 107 ⟩;
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);    /* out of memory before we're even started */
⟨ Assign names to the new vertices, and create a map from g to new_graph 108 ⟩;
sprintf(buffer, ",%s,%d,%d,%d",
    description ? description : null_string,
    self ? 1 : 0, multi ? 1 : 0, directed ? 1 : 0);
make_compound_id(new_graph, "induced(", g, buffer);
```

This code is used in section 105.

107. ⟨ Determine n and nn 107 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + g→n; v++)
  if (v→ind > 0) {
    if (n > IND_GRAPH) panic(very_bad_specs);    /* way too big */
    if (v→ind ≥ IND_GRAPH) {
      if (v→subst ≡ Λ) panic(missing_operand + 1);    /* substitute graph is missing */
      n += v→subst→n;
    } else n += v→ind;
  } else if (v→ind < −nn) nn = −(v→ind);
if (n > IND_GRAPH ∨ nn > IND_GRAPH) panic(very_bad_specs + 1);    /* gigantic */
n += nn;
```

This code is used in section 106.

108. The negative vertices get the negative number as their name. Split vertices get names with an optional prime appended, if the *ind* field is 2; otherwise split vertex names are obtained by appending a colon and an index number between 0 and $ind - 1$. The name of a vertex within a graph *v→subst* is composed of the name of v followed by a colon and the name within that graph.

We store the original *ind* field in the *mult* field of the first corresponding vertex in the new graph, and change *ind* to point to that vertex. This convention makes it easy to determine the location of each vertex's clone or clones. Of course, if the original *ind* field is zero, we leave it zero (Λ), because it has no corresponding vertex in the new graph.

⟨ Assign names to the new vertices, and create a map from g to *new_graph* 108 ⟩ ≡

```
for (k = 1, u = new_graph→vertices; k ≤ nn; k++, u++) {
  u→mult = −k;
  sprintf(buffer, "%ld", −k);
  u→name = gb_save_string(buffer);
}
for (v = g→vertices; v < g→vertices + g→n; v++)
  if ((k = v→ind) < 0) v→map = (new_graph→vertices) − (k + 1);
  else if (k > 0) {
    u→mult = k;
    v→map = u;
    if (k ≤ 2) {
      u→name = gb_save_string(v→name);
      u++;
      if (k ≡ 2) {
        sprintf(buffer, "%s'", v→name);
        u→name = gb_save_string(buffer);
        u++;
      }
    } else if (k ≥ IND_GRAPH) ⟨ Make names and arcs for a substituted graph 114 ⟩
    else
      for (j = 0; j < k; j++, u++) {
        sprintf(buffer, "%.*s:%ld", BUF_SIZE − 12, v→name, j);
        u→name = gb_save_string(buffer);
      }
  }
```

This code is used in section 106.

109. ⟨ Restore g to its original state 109 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + g→n; v++)
  if (v→map) v→ind = v→map→mult;
for (v = new_graph→vertices; v < new_graph→vertices + n; v++) v→u.I = v→v.I = v→z.I = 0;
    /* clear tmp, mult, tlen */
```

This code is used in section 105.

110. The heart of the procedure to construct an induced graph is, of course, the part where we map the arcs of g into arcs of *new_graph*.

Notice that if v has a self-loop in the original graph and if v is being split into several vertices, it will produce arcs between different clones of itself, but it will not produce self-loops unless $self \neq 0$. In an undirected graph, a loop from a vertex to itself will not produce multiple edges among its clones, even if $multi \neq 0$.

More precisely, if v has k clones u through $u+k-1$, an original directed arc from v to v will generate all k^2 possible arcs between them, except that the k self-loops will be eliminated when $self \equiv 0$. An original undirected edge from v to v will generate $\binom{k}{2}$ edges between distinct clones, together with k undirected self-loops if $self \neq 0$.

```
⟨ Insert arcs or edges for induced vertices 110 ⟩ ≡
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    u = v→map;
    if (u) { register Arc *a; register Vertex *uu, *vv;
      k = u→mult;
      if (k < 0) k = 1;      /* k is the number of clones of v */
      else if (k ≥ IND_GRAPH) k = v→subst→n;
      for ( ; k; k−−, u++) {
        if (¬multi) ⟨ Take note of existing edges that touch u 111 ⟩;
        for (a = v→arcs; a; a = a→next) {
          vv = a→tip;
          uu = vv→map;
          if (uu ≡ Λ) continue;
          j = uu→mult;
          if (j < 0) j = 1;      /* j is the number of clones of vv */
          else if (j ≥ IND_GRAPH) j = vv→subst→n;
          if (¬directed) {
            if (vv < v) continue;
            if (vv ≡ v) {
              if (a→next ≡ a + 1) a++;      /* skip second half of self-loop */
              j = k, uu = u;      /* also skip duplicate edges generated by self-loop */
            }
          }
          ⟨ Insert arcs or edges from vertex u to vertices uu through uu + j − 1 112 ⟩;
        }
      }
    }
  }
```

This code is used in section 105.

111. Again we use the *tmp* and *tlen* trick of *gunion* to handle multiple arcs. (This trick explains why the code in the previous section tries to generate as many arcs as possible from a single vertex u, before changing u.)

```
⟨ Take note of existing edges that touch u 111 ⟩ ≡
  for (a = u→arcs; a; a = a→next) {
    a→tip→tmp = u;
    if (directed ∨ a→tip > u ∨ a→next ≡ a + 1) a→tip→tlen = a;
    else a→tip→tlen = a + 1;
  }
```

This code is used in section 110.

112. ⟨Insert arcs or edges from vertex u to vertices uu through $uu + j - 1$ 112⟩ ≡

```
for ( ; j; j--, uu++) {
  if (u ≡ uu ∧ ¬self) continue;
  if (uu→tmp ≡ u ∧ ¬multi) ⟨Update the minimum arc length from u to uu, then continue 113⟩;
  if (directed) gb_new_arc(u, uu, a→len);
  else gb_new_edge(u, uu, a→len);
  uu→tmp = u;
  uu→tlen = ((directed ∨ u ≤ uu) ? u→arcs : uu→arcs);
}
```

This code is used in section 110.

113. ⟨Update the minimum arc length from u to uu, then **continue** 113⟩ ≡

```
{ register Arc *b = uu→tlen;   /* existing arc or edge from u to uu */
  if (a→len < b→len) {
    b→len = a→len;   /* remember the minimum length */
    if (¬directed) (b + 1)→len = a→len;
  }
  continue;
}
```

This code is used in sections 112 and 114.

114. We have now accumulated enough experience to finish off the one remaining piece of program with ease.

⟨Make names and arcs for a substituted graph 114⟩ ≡

```
{ register Graph *gg = v→subst;
  register Vertex *vv = gg→vertices;
  register Arc *a;
  siz_t delta = ((siz_t) u) − ((siz_t) vv);
  for (j = 0; j < v→subst→n; j++, u++, vv++) {
    sprintf(buffer, "%.*s:%.*s", BUF_SIZE/2 − 1, v→name, (BUF_SIZE − 1)/2, vv→name);
    u→name = gb_save_string(buffer);
    for (a = vv→arcs; a; a = a→next) { register Vertex *vvv = a→tip;
      Vertex *uu = vert_offset(vvv, delta);
      if (vvv ≡ vv ∧ ¬self) continue;
      if (uu→tmp ≡ u ∧ ¬multi) ⟨Update the minimum arc length from u to uu, then continue 113⟩;
      if (¬directed) {
        if (vvv < vv) continue;
        if (vvv ≡ vv ∧ a→next ≡ a + 1) a++;   /* skip second half of self-loop */
        gb_new_edge(u, uu, a→len);
      } else gb_new_arc(u, uu, a→len);
      uu→tmp = u;
      uu→tlen = ((directed ∨ u ≤ uu) ? u→arcs : uu→arcs);
    }
  }
}
```

This code is used in section 108.

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

October 28, 2003 at 10:23

GB_BASIC

© 1993 Stanford University

This file may be freely copied and distributed, provided that no changes whatsoever are made. All users are asked to help keep the Stanford GraphBase files consistent and "uncorrupted," identical everywhere in the world. Changes are permissible only if the modified file is given a new name, different from the names of existing files in the Stanford GraphBase, and only if the modified file is clearly identified as not being part of that GraphBase. (The CWEB system has a "change file" facility by which users can easily make minor alterations without modifying the master source files in any way. Everybody is supposed to use change files instead of changing the files.) The author has tried his best to produce correct and useful programs, in order to help promote computer science research, but no warranty of any kind should be assumed.

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