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

1. Introduction. This GraphBase module contains the *miles* subroutine, which creates a family of undirected graphs based on highway mileage data between North American cities. Examples of the use of this procedure can be found in the demo programs MILES_SPAN and GB_PLANE.

⟨ gb_miles.h 1 ⟩ ≡
```
extern Graph *miles();
```
See also sections 2, 16, and 21.

2. The subroutine call *miles*(n, *north_weight*, *west_weight*, *pop_weight*, *max_distance*, *max_degree*, *seed*) constructs a graph based on the information in `miles.dat`. Each vertex of the graph corresponds to one of the 128 cities whose name is alphabetically greater than or equal to 'Ravenna, Ohio' in the 1949 edition of Rand McNally & Company's *Standard Highway Mileage Guide*. Edges between vertices are assigned lengths representing distances between cities, in miles. In most cases these mileages come from the Rand McNally Guide, but several dozen entries needed to be changed drastically because they were obviously too large or too small; in such cases an educated guess was made. Furthermore, about 5% of the entries were adjusted slightly in order to ensure that all distances satisfy the "triangle inequality": The graph generated by *miles* has the property that the distance from u to v plus the distance from v to w always exceeds or equals the distance from u to w.

The constructed graph will have $\min(n, 128)$ vertices; the default value $n = 128$ is substituted if $n = 0$. If n is less than 128, the n cities will be selected by assigning a weight to each city and choosing the n with largest weight, using random numbers to break ties in case of equal weights. Weights are computed by the formula

$$\mathit{north\_weight} \cdot \mathit{lat} + \mathit{west\_weight} \cdot \mathit{lon} + \mathit{pop\_weight} \cdot \mathit{pop},$$

where *lat* is latitude north of the equator, *lon* is longitude west of Greenwich, and *pop* is the population in 1980. Both *lat* and *lon* are given in "centidegrees" (hundredths of degrees). For example, San Francisco has $\mathit{lat} = 3778$, $\mathit{lon} = 12242$, and $\mathit{pop} = 678974$; this means that, before the recent earthquake, it was located at 37.78° north latitude and 122.42° west longitude, and that it had 678,974 residents in the 1980 census. The weight parameters must satisfy

$$|\mathit{north\_weight}| \le 100{,}000, \quad |\mathit{west\_weight}| \le 100{,}000, \quad |\mathit{pop\_weight}| \le 100.$$

The constructed graph will be "complete"—that is, it will have edges between every pair of vertices—unless special values are given to the parameters *max_distance* or *max_degree*. If $\mathit{max\_distance} \ne 0$, edges with more than *max_distance* miles will not appear; if $\mathit{max\_degree} \ne 0$, each vertex will be limited to at most *max_degree* of its shortest edges.

Vertices of the graph will appear in order of decreasing weight. The *seed* parameter defines the pseudo-random numbers used wherever a "random" choice between equal-weight vertices or equal-length edges needs to be made.

#define `MAX_N` 128

⟨ gb_miles.h 1 ⟩ +≡
```
#define MAX_N 128   /* maximum and default number of cities */
```

3. Examples: The call *miles*$(100, 0, 0, 1, 0, 0, 0)$ will construct a complete graph on 100 vertices, representing the 100 most populous cities in the database. It turns out that San Diego, with a population of 875,538, is the winning city by this criterion, followed by San Antonio (population 786,023), San Francisco (678,974), and Washington D.C. (638,432).

To get n cities in the western United States and Canada, you can say *miles*$(n, 0, 1, 0, \ldots)$; to get n cities in the Northeast, use a call like *miles*$(n, 1, -1, 0, \ldots)$. A parameter setting like $(50, -500, 0, 1, \ldots)$ produces mostly Southern cities, except for a few large metropolises in the north.

If you ask for *miles*$(n, a, b, c, 0, 1, 0)$, you get an edge between cities if and only if each city is the nearest to the other, among the n cities selected. (The graph is always undirected: There is an arc from u to v if and only if there's an arc of the same length from v to u.)

A random selection of cities can be obtained by calling *miles*$(n, 0, 0, 0, m, d, s)$. Different choices of the seed number s will produce different selections, in a system-independent manner; identical results will be obtained on all computers when identical parameters have been specified. Equivalent experiments on algorithms for graph manipulation can therefore be performed by researchers in different parts of the world. Any value of s between 0 and $2^{31} - 1$ is permissible.

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

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

5. The C file `gb_miles.c` has the following overall shape:

```
#include "gb_io.h"      /* we will use the GB_IO routines for input */
#include "gb_flip.h"    /* we will use the GB_FLIP routines for random numbers */
#include "gb_graph.h"   /* we will use the GB_GRAPH data structures */
#include "gb_sort.h"    /* and the linksort routine */
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 9 ⟩
  ⟨ Private variables 10 ⟩
  Graph *miles(n, north_weight, west_weight, pop_weight, max_distance, max_degree, seed)
      unsigned long n;    /* number of vertices desired */
      long north_weight;  /* coefficient of latitude in the weight function */
      long west_weight;   /* coefficient of longitude in the weight function */
      long pop_weight;    /* coefficient of population in the weight function */
      unsigned long max_distance;  /* maximum distance in an edge, if nonzero */
      unsigned long max_degree;    /* maximum number of edges per vertex, if nonzero */
      long seed;   /* random number seed */
  { ⟨ Local variables 6 ⟩
    gb_init_rand(seed);
    ⟨ Check that the parameters are valid 7 ⟩;
    ⟨ Set up a graph with n vertices 8 ⟩;
    ⟨ Read the data file miles.dat and compute city weights 11 ⟩;
    ⟨ Determine the n cities to use in the graph 14 ⟩;
    ⟨ Put the appropriate edges into the graph 17 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);   /* oops, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

6. ⟨ Local variables 6 ⟩ ≡
Graph $*new\_graph$; /* the graph constructed by *miles* */
register long j, k; /* all-purpose indices */

This code is used in section 5.

7. ⟨ Check that the parameters are valid 7 ⟩ ≡
if $(n \equiv 0 \lor n > \texttt{MAX\_N})$ $n = \texttt{MAX\_N}$;
if $(max\_degree \equiv 0 \lor max\_degree \ge n)$ $max\_degree = n - 1$;
if $(north\_weight > 100000 \lor west\_weight > 100000 \lor pop\_weight > 100$
$\lor north\_weight < -100000 \lor west\_weight < -100000 \lor pop\_weight < -100)$ $panic(bad\_specs)$;
/* the magnitude of at least one weight is too big */

This code is used in section 5.

8. ⟨ Set up a graph with n vertices 8 ⟩ ≡
$new\_graph = gb\_new\_graph(n)$;
if $(new\_graph \equiv \Lambda)$ $panic(no\_room)$; /* out of memory before we're even started */
$sprintf(new\_graph\to id$, "miles(%lu,%ld,%ld,%ld,%lu,%lu,%ld)", n, $north\_weight$, $west\_weight$,
pop_weight, $max\_distance$, max_degree, $seed$);
$strcpy(new\_graph\to util\_types$, "ZZIIIIZZZZZZZZ");

This code is used in section 5.

9. Vertices. As we read in the data, we construct a list of nodes, each of which contains a city's name, latitude, longitude, population, and weight. These nodes conform to the specifications stipulated in the GB_SORT module. After the list has been sorted by weight, the top n entries will be the vertices of the new graph.

⟨ Type declarations 9 ⟩ ≡

```
typedef struct node_struct {   /* records to be sorted by gb_linksort */
  long key;   /* the nonnegative sort key (weight plus 2^30) */
  struct node_struct *link;   /* pointer to next record */
  long kk;   /* index of city in the original database */
  long lat, lon, pop;   /* latitude, longitude, population */
  char name[30];   /* "City Name, ST" */
} node;
```

This code is used in section 5.

10. The constants defined here are taken from the specific data in `miles.dat`, because this routine is not intended to be perfectly general.

⟨ Private variables 10 ⟩ ≡

```
static long min_lat = 2672, max_lat = 5042, min_lon = 7180, max_lon = 12312, min_pop = 2521,
    max_pop = 875538;   /* tight bounds on data entries */
static node *node_block;   /* array of nodes holding city info */
static long *distance;   /* array of distances */
```

This code is used in section 5.

11. The data in `miles.dat` appears in 128 groups of lines, one for each city, in reverse alphabetical order. These groups have the general form

```
City Name, ST[lat,lon]pop
d1 d2 d3 d4 d5 d6 ...  (possibly several lines' worth)
```

where `City Name` is the name of the city (possibly including spaces); `ST` is the two-letter state code; `lat` and `lon` are latitude and longitude in hundredths of degrees; `pop` is the population; and the remaining numbers `d1`, `d2`, ... are distances to the previously named cities in reverse order. Each distance is separated from the previous item by either a blank space or a newline character. For example, the line

```
San Francisco, CA[3778,12242]678974
```

specifies the data about San Francisco that was mentioned earlier. From the first few groups

```
Youngstown, OH[4110,8065]115436
Yankton, SD[4288,9739]12011
966
Yakima, WA[4660,12051]49826
1513 2410
Worcester, MA[4227,7180]161799
2964 1520 604
```

we learn that the distance from Worcester, Massachusetts, to Yakima, Washington, is 2964 miles; from Worcester to Youngstown it is 604 miles.

The following two-letter "state codes" are used for Canadian provinces: `BC` = British Columbia, `MB` = Manitoba, `ON` = Ontario, `SK` = Saskatchewan.

⟨ Read the data file `miles.dat` and compute city weights 11 ⟩ ≡

```
node_block = gb_typed_alloc(MAX_N, node, new_graph→aux_data);
distance = gb_typed_alloc(MAX_N * MAX_N, long, new_graph→aux_data);
if (gb_trouble_code) {
  gb_free(new_graph→aux_data);
  panic(no_room + 1);   /* no room to copy the data */
}
if (gb_open("miles.dat") ≠ 0) panic(early_data_fault);
    /* couldn't open "miles.dat" using GraphBase conventions; io_errors tells why */
for (k = MAX_N − 1; k ≥ 0; k−−) ⟨ Read and store data for city k 12 ⟩;
if (gb_close() ≠ 0) panic(late_data_fault);   /* something's wrong with "miles.dat"; see io_errors */
```

This code is used in section 5.

12. The bounds we've imposed on *north_weight*, *west_weight*, and *pop_weight* guarantee that the key value computed here will be between 0 and 2^{31}.

```
⟨ Read and store data for city k 12 ⟩ ≡
  { register node *p;
    p = node_block + k;
    p->kk = k;
    if (k) p->link = p - 1;
    gb_string(p->name, '[');
    if (gb_char() ≠ '[') panic(syntax_error);    /* out of sync in miles.dat */
    p->lat = gb_number(10);
    if (p->lat < min_lat ∨ p->lat > max_lat ∨ gb_char() ≠ ',') panic(syntax_error + 1);
        /* latitude data was clobbered */
    p->lon = gb_number(10);
    if (p->lon < min_lon ∨ p->lon > max_lon ∨ gb_char() ≠ ']') panic(syntax_error + 2);
        /* longitude data was clobbered */
    p->pop = gb_number(10);
    if (p->pop < min_pop ∨ p->pop > max_pop) panic(syntax_error + 3);
        /* population data was clobbered */
    p->key = north_weight * (p->lat - min_lat) + west_weight * (p->lon - min_lon) + pop_weight * (p->pop -
        min_pop) + #40000000;
    ⟨ Read the mileage data for city k 13 ⟩;
    gb_newline();
  }
```

This code is used in section 11.

13. **#define** $d(j,k)$ $*(distance + (\texttt{MAX\_N} * j + k))$

```
⟨ Read the mileage data for city k 13 ⟩ ≡
  {
    for (j = k + 1; j < MAX_N; j++) {
      if (gb_char() ≠ '␣') gb_newline();
      d(j, k) = d(k, j) = gb_number(10);
    }
  }
```

This code is used in section 12.

14. Once all the nodes have been set up, we can use the *gb_linksort* routine to sort them into the desired order. This routine, which is part of the *gb_graph* module, builds 128 lists from which the desired nodes are readily accessed in decreasing order of weight, using random numbers to break ties.

We set the population to zero in every city that isn't chosen. Then that city will be excluded when edges are examined later.

```
⟨ Determine the n cities to use in the graph 14 ⟩ ≡
  { register node *p;    /* the current node being considered */
    register Vertex *v = new_graph->vertices;    /* the first unfilled vertex */
    gb_linksort(node_block + MAX_N - 1);
    for (j = 127; j ≥ 0; j--)
      for (p = (node *) gb_sorted[j]; p; p = p->link) {
        if (v < new_graph->vertices + n) ⟨ Add city p->kk to the graph 15 ⟩
        else p->pop = 0;    /* this city is not being used */
      }
  }
```

This code is used in section 5.

15. Utility fields x and y for each vertex are set to coordinates that can be used in geometric computations; these coordinates are obtained by simple linear transformations of latitude and longitude (not by any kind of sophisticated polyconic projection). We will have

$$0 \le x \le 5132, \qquad 0 \le y \le 3555.$$

Utility field z is set to the city's index number (0 to 127) in the original database. Utility field w is set to the city's population.

The coordinates computed here are compatible with those in the TEX file `cities.texmap`. Users might want to incorporate edited copies of that file into documents that display results obtained with *miles* graphs.

```
#define x_coord x.I
#define y_coord y.I
#define index_no z.I
#define people w.I
```

⟨ Add city p→kk to the graph 15 ⟩ ≡

```
{
  v→x_coord = max_lon − p→lon;   /* x coordinate is complement of longitude */
  v→y_coord = p→lat − min_lat;
  v→y_coord += (v→y_coord) ≫ 1;   /* y coordinate is 1.5 times latitude */
  v→index_no = p→kk;
  v→people = p→pop;
  v→name = gb_save_string(p→name);
  v++;
}
```

This code is used in section 14.

16. ⟨ `gb_miles.h` 1 ⟩ +≡

```
#define x_coord x.I   /* utility field definitions for the header file */
#define y_coord y.I
#define index_no z.I
#define people w.I
```

17. Arcs. We make the distance negative in the matrix entry for an arc that is not to be included. Nothing needs to be done in this regard unless the user has specified a maximum degree or a maximum edge length.

⟨ Put the appropriate edges into the graph 17 ⟩ ≡
```
if (max_distance > 0 ∨ max_degree > 0) ⟨ Prune unwanted edges by negating their distances 18 ⟩;
{ register Vertex *u, *v;
  for (u = new_graph→vertices; u < new_graph→vertices + n; u++) {
    j = u→index_no;
    for (v = u + 1; v < new_graph→vertices + n; v++) {
      k = v→index_no;
      if (d(j,k) > 0 ∧ d(k,j) > 0) gb_new_edge(u, v, d(j,k));
    }
  }
}
```
This code is used in section 5.

18. ⟨ Prune unwanted edges by negating their distances 18 ⟩ ≡
```
{ register node *p;
  if (max_degree ≡ 0) max_degree = MAX_N;
  if (max_distance ≡ 0) max_distance = 30000;
  for (p = node_block; p < node_block + MAX_N; p++)
    if (p→pop) {    /* this city not deleted */
      k = p→kk;
      ⟨ Blank out all undesired edges from city k 19 ⟩;
    }
}
```
This code is used in section 17.

19. Here we reuse the key fields of the nodes, storing complementary distances there instead of weights. We also let the sorting routine change the link fields. The other fields, however—especially *pop*—remain unchanged. Yes, the author knows this is a wee bit tricky, but why not?

⟨ Blank out all undesired edges from city k 19 ⟩ ≡
```
{ register node *q;
  register node *s = Λ;    /* list of nodes containing edges from city k */
  for (q = node_block; q < node_block + MAX_N; q++)
    if (q→pop ∧ q ≠ p) {    /* another city not deleted */
      j = d(k, q→kk);    /* distance from p to q */
      if (j > max_distance) d(k, q→kk) = −j;
      else {
        q→key = max_distance − j;
        q→link = s;
        s = q;
      }
    }
  gb_linksort(s);    /* now all the surviving edges from p are in the list gb_sorted[0] */
  j = 0;    /* j counts how many edges have been accepted */
  for (q = (node *) gb_sorted[0]; q; q = q→link)
    if (++j > max_degree) d(k, q→kk) = −d(k, q→kk);
}
```
This code is used in section 18.

20. Random access to the distance matrix is provided to users via the external function *miles_distance*. Caution: This function can be used only with the graph most recently made by *miles*, and only when the graph's *aux_data* has not been recycled, and only when the z utility fields have not been used for another purpose.

The result might be negative when an edge has been suppressed. Moreover, we can in fact have $miles\_distance(u,v) < 0$ when $miles\_distance(v,u) > 0$, if the distance in question was suppressed by the *max_degree* constraint on u but not on v.

```
long miles_distance(u, v)
    Vertex *u, *v;
{
  return d(u→index_no, v→index_no);
}
```

21. ⟨ gb_miles.h 1 ⟩ +≡

```
extern long miles_distance();
```

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

April 17, 2003 at 13:49

GB_MILES

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