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

1. Introduction. This GraphBase module contains the *book* subroutine, which creates a family of undirected graphs that are based on classic works of literature. It also contains the *bi_book* subroutine, which creates a related family of bipartite graphs. An example of the use of *book* can be found in the demonstration program BOOK_COMPONENTS.

⟨ `gb_books.h` 1 ⟩ ≡
```
extern Graph *book( );
extern Graph *bi_book( );
```
See also sections 6, 18, and 23.

2. The subroutine call *book*(⟨title⟩, n, x, *first_chapter*, *last_chapter*, *in_weight*, *out_weight*, *seed*) constructs a graph based on the information in ⟨title⟩`.dat`, where ⟨title⟩ is either `"anna"` (for *Anna Karenina*), `"david"` (for *David Copperfield*), `"jean"` (for *Les Misérables*), `"huck"` (for *Huckleberry Finn*), or `"homer"` (for *The Iliad*). Each vertex of the graph corresponds to one of the characters in the selected book. Edges between vertices correspond to encounters between those characters. The length of each edge is 1.

Subsets of the book can be selected by specifying that the edge data should be restricted to chapters between *first_chapter* and *last_chapter*, inclusive. If *first_chapter* = 0, the result is the same as if *first_chapter* = 1. If *last_chapter* = 0 or if *last_chapter* exceeds the total number of chapters in the book, the result is the same as if *last_chapter* were the number of the book's final chapter.

The constructed graph will have $\min(n, N) - x$ vertices, where N is the total number of characters in the selected book. However, if n is zero, n is automatically made equal to the maximum possible value, N. If n is less than N, the $n - x$ characters will be selected by assigning a weight to each character and choosing the n with largest weight, then excluding the largest x of these, using random numbers to break ties in case of equal weights. Weights are computed by the formula

$$\mathit{in\_weight} \cdot \mathit{chapters\_in} + \mathit{out\_weight} \cdot \mathit{chapters\_out},$$

where *chapters_in* is the number of chapters between *first_chapter* and *last_chapter* in which a particular character appears, and *chapters_out* is the number of other chapters in which that character appears. Both *in_weight* and *out_weight* must be at most 1,000,000 in absolute value.

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 needs to be made. As usual with GraphBase routines, different choices of *seed* will in general produce different selections, but in a system-independent manner; identical results will be obtained on all computers when identical parameters have been specified. Any *seed* value between 0 and $2^{31} - 1$ is permissible.

3. Examples: The call *book*(`"anna"`, 0, 0, 0, 0, 0, 0, 0) will construct a graph on 138 vertices that represent all 138 characters of Tolstoy's *Anna Karenina*, as recorded in `anna.dat`. Two vertices will be adjacent if the corresponding characters encounter each other anywhere in the book. The call *book*(`"anna"`, 50, 0, 0, 0, 1, 1, 0) is similar, but it is restricted to the 50 characters that occur most frequently, i.e., in the most chapters. The call *book*(`"anna"`, 50, 0, 10, 120, 1, 1, 0) has the same vertices, but it has edges only for encounters that take place between chapter 10 and chapter 120, inclusive. The call *book*(`"anna"`, 50, 0, 10, 120, 1, 0, 0) is similar, but its vertices are the 50 characters that occur most often in chapters 10 through 120, without regard to how often they occur in the rest of the book. The call *book*(`"anna"`, 50, 0, 10, 120, 0, 0, 0) is also similar, but it chooses 50 characters completely at random (possibly from those that don't occur in the selected chapters at all).

Parameter x, which causes the x vertices of highest weight to be excluded, is usually either 0 or 1. It is provided primarily so that users can set $x = 1$ with respect to *David Copperfield* and *Huckleberry Finn*; those novels are narrated by their principal character, so they have edges between the principal character and almost everybody else. (Characters cannot get into the action of a first-person account unless they encounter the narrator or unless the narrator is quoting some other person's story.) The corresponding graphs tend to have more interesting connectivity properties if we leave the narrator out by setting $x = 1$. For example, there are 87 characters in *David Copperfield*; the call *book*(`"david"`, 0, 1, 0, 0, 1, 1, 0) produces a graph with 86 vertices, one for every character except David Copperfield himself.

4. The subroutine call *bi_book*(⟨title⟩, n, x, *first_chapter*, *last_chapter*, *in_weight*, *out_weight*, *seed*) produces a bipartite graph in which the vertices of the first part are exactly the same as the vertices of the graph returned by *book*, while the vertices of the second part are the selected chapters. For example, *bi_book*(`"anna"`, 50, 0, 10, 120, 1, 1, 0) creates a bipartite graph with 50+111 vertices. There is an edge between each character and the chapters in which that character appears.

5. Chapter numbering needs further explanation. *Anna Karenina* has 239 chapters, which are numbered 1.1 through 8.19 in the work itself but renumbered 1 through 239 as far as the *book* routine is concerned. Thus, setting *first_chapter* = 10 and *last_chapter* = 120 turns out to be equivalent to selecting chapters 1.10 through 4.19 (more precisely, chapter 10 of book 1 through chapter 19 of book 4). *Les Misérables* has an even more involved scheme; its 356 chapters range from 1.1.1 (part 1, book 1, chapter 1) to 5.9.6 (part 5, book 9, chapter 6). After *book* or *bi_book* has created a graph, the external integer variable *chapters* will contain the total number of chapters, and *chap_name* will be an array of strings containing the structured chapter numbers. For example, after *book*(`"jean"`, ...), we will have *chapters* = 356, *chap_name*[1] = `"1.1.1"`, ..., *chap_name*[356] = `"5.9.6"`; *chap_name*[0] will be `""`.

```
#define MAX_CHAPS 360    /* no book will have this many chapters */
```

⟨ External variables 5 ⟩ ≡

```
  long chapters;    /* the total number of chapters in the selected book */
  char *chap_name[MAX_CHAPS] = {""};    /* string names of those chapters */
```

This code is used in section 8.

6. As usual, we put declarations of the external variables into the header file for users to **include**.

⟨ `gb_books.h` 1 ⟩ +≡

```
  extern long chapters;    /* the total number of chapters in the selected book */
  extern char *chap_name[];    /* string names of those chapters */
```

7. If the *book* or *bi_book* 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 *book* 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; }
  format node long    /* the node type is defined below */
```

8. The C file `gb_books.c` has the overall shape shown here. It makes use of an internal subroutine called *bgraph*, which combines the work of *book* and *bi_book*.

```
#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 gb_linksort routine */
⟨ Preprocessor definitions ⟩
⟨ Type declarations 13 ⟩
⟨ Private variables 11 ⟩
⟨ External variables 5 ⟩
static Graph *bgraph(bipartite, title, n, x, first_chapter, last_chapter, in_weight, out_weight, seed)
    long bipartite;    /* should we make the graph bipartite? */
    char *title;    /* identification of the selected book */
    unsigned long n;    /* number of vertices desired before exclusion */
    unsigned long x;    /* number of vertices to exclude */
    unsigned long first_chapter, last_chapter;    /* interval of chapters leading to edges */
    long in_weight;    /* weight coefficient pertaining to chapters in that interval */
    long out_weight;    /* weight coefficient pertaining to chapters not in that interval */
    long seed;    /* random number seed */
{ ⟨ Local variables 9 ⟩
  gb_init_rand(seed);
  ⟨ Check that the parameters are valid 10 ⟩;
  ⟨ Skim the data file, recording the characters and computing their statistics 15 ⟩;
  ⟨ Choose the vertices and put them into an empty graph 27 ⟩;
  ⟨ Read the data file more carefully and fill the graph as instructed 29 ⟩;
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);    /* (expletive deleted) we ran out of memory somewhere back there */
  }
  return new_graph;
}
Graph *book(title, n, x, first_chapter, last_chapter, in_weight, out_weight, seed)
    char *title;
    unsigned long n, x, first_chapter, last_chapter;
    long in_weight, out_weight, seed;
{ return bgraph(0L, title, n, x, first_chapter, last_chapter, in_weight, out_weight, seed); }
Graph *bi_book(title, n, x, first_chapter, last_chapter, in_weight, out_weight, seed)
    char *title;
    unsigned long n, x, first_chapter, last_chapter;
    long in_weight, out_weight, seed;
{ return bgraph(1L, title, n, x, first_chapter, last_chapter, in_weight, out_weight, seed); }
```

9. ⟨ Local variables 9 ⟩ ≡

```
Graph *new_graph;    /* the graph constructed by book or bi_book */
register long j, k;    /* all-purpose indices */
long characters;    /* the total number of characters in the selected book */
register node *p;    /* information about the current character */
```

See also section 21.

This code is used in section 8.

10. **#define** `MAX_CHARS` 600 /* there won't be more characters than this */

⟨ Check that the parameters are valid 10 ⟩ ≡

```
if (n ≡ 0) n = MAX_CHARS;
if (first_chapter ≡ 0) first_chapter = 1;
if (last_chapter ≡ 0) last_chapter = MAX_CHAPS;
if (in_weight > 1000000 ∨ in_weight < −1000000 ∨ out_weight > 1000000 ∨ out_weight < −1000000)
  panic(bad_specs); /* the magnitude of at least one weight is too big */
sprintf(file_name, "%.6s.dat", title);
if (gb_open(file_name) ≠ 0) panic(early_data_fault); /* couldn't open the file; io_errors tells why */
```

This code is used in section 8.

11. ⟨ Private variables 11 ⟩ ≡

```
static char file_name[] = "xxxxxx.dat";
```

See also sections 14 and 24.

This code is used in section 8.

12. Vertices. Each character in a book has been given a two-letter code name for internal use. The code names are explained at the beginning of each data file by a number of lines that look like this:

`XX` ⟨name⟩`,`⟨description⟩

For example, here's one of the lines near the beginning of `"anna.dat"`:

```
AL Alexey Alexandrovitch Karenin, minister of state
```

The ⟨name⟩ does not contain a comma; the ⟨description⟩ might.

A blank line follows the cast of characters.

Internally, we will think of the two-letter code as a radix-36 integer. Thus `AA` will be the number $10\times36+10$, and `ZZ` will be $35\times36+35$. The *gb_number* routine in GB_IO is set up to input radix-36 integers just as it does hexadecimal ones. In *The Iliad*, many of the minor characters have numeric digits in their code names because the total number of characters is too large to permit mnemonic codes for everybody.

```
#define MAX_CODE 1296    /* 36 × 36, the number of two-digit codes in radix 36 */
```

13. In order to choose the vertices, we want to represent each character as a node whose key corresponds to its weight; then the *gb_linksort* routine of GB_SORT will provide the desired rank-ordering. We will find it convenient to use these nodes for all the data processing that *bgraph* has to do.

⟨ Type declarations 13 ⟩ ≡

```
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 code;    /* code number of this character */
  long in;    /* number of occurrences in selected chapters */
  long out;    /* number of occurrences in unselected chapters */
  long chap;    /* seen most recently in this chapter */
  Vertex *vert;    /* vertex corresponding to this character */
} node;
```

This code is used in section 8.

14. Not only do nodes point to codes, we also want codes to point to nodes.

⟨ Private variables 11 ⟩ +≡

```
static node node_block[MAX_CHARS];    /* array of nodes for working storage */
static node *xnode[MAX_CODE];    /* the node, if any, having a given code */
```

15. We will read the data file twice, once quickly (to collect statistics) and once more thoroughly (to record detailed information). Here is the quick version.

⟨ Skim the data file, recording the characters and computing their statistics 15 ⟩ ≡

```
⟨ Read the character codes at the beginning of the data file, and prepare a node for each one 16 ⟩;
⟨ Skim the chapter information, counting the number of chapters in which each character appears 19 ⟩;
if (gb_close( ) ≠ 0) panic(late_data_fault);    /* checksum or other failure in data file; see io_errors */
```

This code is used in section 8.

16. ⟨ Read the character codes at the beginning of the data file, and prepare a node for each one 16 ⟩ ≡

```
for (k = 0; k < MAX_CODE; k++) xnode[k] = Λ;
{ register long c;    /* current code entering the system */
  p = node_block;    /* current node entering the system */
  while ((c = gb_number(36)) ≠ 0) {    /* note that 00 is not a legal code */
    if (c ≥ MAX_CODE ∨ gb_char( ) ≠ '␣') panic(syntax_error);    /* unreadable line in data file */
    if (p ≥ &node_block[MAX_CHARS]) panic(syntax_error + 1);    /* data has too many characters */
    p→link = (p ≡ node_block ? Λ : p − 1);
    p→code = c;
    xnode[c] = p;
    p→in = p→out = p→chap = 0;
    p→vert = Λ;
    p++;
    gb_newline( );
  }
  characters = p − node_block;
  gb_newline( );    /* bypass the blank line that terminates the character data */
}
```

This code is used in section 15.

17. Later we will read through this part of the file again, extracting additional information if it turns out to be relevant. The ⟨description⟩ string is provided to users in a *desc* field, in case anybody cares to look at it. The *in* and *out* statistics are also made available in utility fields called *in_count* and *out_count*. The code value is placed in the *short_code* field.

```
#define desc  z.S    /* utility field z points to the ⟨description⟩ string */
#define in_count  y.I    /* utility field y counts appearances in selected chapters */
#define out_count  x.I    /* utility field x counts appearances in other chapters */
#define short_code  u.I    /* utility field u contains a radix-36 number */
```

⟨ Read the data about characters again, noting vertex names and the associated descriptions 17 ⟩ ≡

```
{ register long c;    /* current code entering the system a second time */
  while ((c = gb_number(36)) ≠ 0) { register Vertex *v = xnode[c]→vert;
    if (v) {
      if (gb_char( ) ≠ '␣') panic(impossible);    /* can't happen */
      gb_string(str_buf, ',');    /* scan the ⟨name⟩ part */
      v→name = gb_save_string(str_buf);
      if (gb_char( ) ≠ ',') panic(syntax_error + 2);    /* missing comma after ⟨name⟩ */
      if (gb_char( ) ≠ '␣') panic(syntax_error + 3);    /* missing space after comma */
      gb_string(str_buf, '\n');    /* scan the ⟨description⟩ part */
      v→desc = gb_save_string(str_buf);
      v→in_count = xnode[c]→in;
      v→out_count = xnode[c]→out;
      v→short_code = c;
    }
    gb_newline( );
  }
  gb_newline( );    /* bypass the blank line that terminates the character data */
}
```

This code is used in section 29.

18. ⟨ gb_books.h 1 ⟩ +≡
#define *desc* *z.S* /* utility field definitions for the header file */
#define *in_count* *y.I*
#define *out_count* *x.I*
#define *short_code* *u.I*

19. Edges. The second part of the data file has a line for each chapter, containing "cliques of encounters." For example, the line

```
3.22:AA,BB,CC,DD;CC,DD,EE;AA,FF
```

means that, in chapter 22 of book 3, there were encounters between the pairs

`AA-BB`, `AA-CC`, `AA-DD`, `BB-CC`, `BB-DD`, `CC-DD`, `CC-EE`, `DD-EE`, and `AA-FF`.

(The encounter `CC-DD` is specified twice, once in the clique `AA,BB,CC,DD` and once in `CC,DD,EE`; this does not imply anything about the actual number of encounters between `CC` and `DD` in the chapter.)

A clique might involve one character only, when that character is featured in sort of a soliloquy.

A chapter might contain no references to characters at all. In such a case the '`:`' following the chapter number is omitted.

There might be more encounters than will fit on a single line. In such cases, continuation lines begin with '`&:`'. This convention turns out to be needed only in `homer.dat`; chapters in *The Iliad* are substantially more complex than the chapters in other GraphBase books.

On our first pass over the data, we simply want to compute statistics about who appears in what chapters, so we ignore the distinction between commas and semicolons.

⟨ Skim the chapter information, counting the number of chapters in which each character appears 19 ⟩ ≡

```
for (k = 1; k < MAX_CHAPS ∧ ¬gb_eof( ); k++) {
  gb_string(str_buf, ':');      /* read past the chapter number */
  if (str_buf[0] ≡ '&') k--;      /* continuation of previous chapter */
  while (gb_char( ) ≠ '\n') { register long c = gb_number(36);
    if (c ≥ MAX_CODE) panic(syntax_error + 4);      /* missing punctuation between characters */
    p = xnode[c];
    if (p ≡ Λ) panic(syntax_error + 5);      /* unknown character */
    if (p→chap ≠ k) {
      p→chap = k;
      if (k ≥ first_chapter ∧ k ≤ last_chapter) p→in++;
      else p→out++;
    }
  }
  gb_newline( );
}
if (k ≡ MAX_CHAPS) panic(syntax_error + 6);      /* too many chapters */
chapters = k − 1;
```

This code is used in section 15.

20. Our second pass over the data is very similar to the first, if we are simply computing a bipartite graph. In that case we add an edge to the graph between each selected chapter and each selected character in that chapter. Local variable *chap_base* will point to a vertex such that *chap_base* + k is the vertex corresponding to chapter k.

The *in_count* of a chapter vertex is the degree of that vertex, i.e., the number of selected characters that appear in the corresponding chapter. The *out_count* is the number of characters that appear in the chapter but were omitted from the graph. Thus the *in_count* and *out_count* for chapters are analogous to the *in_count* and *out_count* for characters.

```
⟨ Read the chapter information a second time and create the appropriate bipartite edges 20 ⟩ ≡
  {
    for (p = node_block; p < node_block + characters; p++) p→chap = 0;
    for (k = 1; ¬gb_eof( ); k++) {
      gb_string(str_buf, ':');      /* read the chapter number */
      if (str_buf[0] ≡ '&') k−−;
      else {
        if (str_buf[strlen(str_buf) − 1] ≡ '\n') str_buf[strlen(str_buf) − 1] = '\0';
        chap_name[k] = gb_save_string(str_buf);
      }
      if (k ≥ first_chapter ∧ k ≤ last_chapter) { register Vertex *u = chap_base + k;
        if (str_buf[0] ≠ '&') {
          u→name = chap_name[k];
          u→desc = null_string;
          u→in_count = u→out_count = 0;
        }
        while (gb_char( ) ≠ '\n') { register long c = gb_number(36);
          p = xnode[c];
          if (p→chap ≠ k) { register Vertex *v = p→vert;
            p→chap = k;
            if (v) { gb_new_edge(v, u, 1_L);
              u→in_count++;
            } else u→out_count++;
          }
        }
      }
      gb_newline( );
    }
  }
```

This code is used in section 29.

21. ⟨ Local variables 9 ⟩ +≡

```
  Vertex *chap_base;      /* the bipartite vertex for chapter k is chap_base + k */
```

22. The second pass has to work a little harder when we are recording encounters from cliques, but the logic isn't difficult really. We insert a reference to the first chapter that generated each edge, in utility field *chap_no* of the corresponding **Arc** record.

```
#define chap_no a.I  /* utility field a holds a chapter number */
⟨ Read the chapter information a second time and create the appropriate edges for encounters 22 ⟩ ≡
  for (k = 1; ¬gb_eof( ); k++) { char *s;
    s = gb_string(str_buf, ':');  /* read the chapter number */
    if (str_buf[0] ≡ '&') k−−;
    else { if (*(s − 2) ≡ '\n') *(s − 2) = '\0';
      chap_name[k] = gb_save_string(str_buf);
    }
    if (k ≥ first_chapter ∧ k ≤ last_chapter) { register long c = gb_char( );
      while (c ≠ '\n') { register Vertex **pp = clique_table;
        register Vertex **qq, **rr;  /* pointers within the clique table */
        do { c = gb_number(36);  /* set c to code for next character of clique */
          if (xnode[c]→vert)  /* is that character a selected vertex? */
            *pp++ = xnode[c]→vert;  /* if so, that vertex joins the current clique */
          c = gb_char( );
        } while (c ≡ ',');  /* repeat until end of the clique */
        for (qq = clique_table; qq + 1 < pp; qq++)
          for (rr = qq + 1; rr < pp; rr++)
            ⟨ Make the vertices *qq and *rr adjacent, if they aren't already 25 ⟩;
      }
    }
    gb_newline( );
  }
```

This code is used in section 29.

23.
```
⟨ gb_books.h 1 ⟩ +≡
#define chap_no a.I  /* utility field definition in the header file */
```

24.
```
⟨ Private variables 11 ⟩ +≡
  static Vertex *clique_table[30];  /* pointers to vertices in the current clique */
```

25.
```
⟨ Make the vertices *qq and *rr adjacent, if they aren't already 25 ⟩ ≡
  { register Vertex *u = *qq, *v = *rr;
    register Arc *a;
    for (a = u→arcs; a; a = a→next)
      if (a→tip ≡ v) goto found;
    gb_new_edge(u, v, 1L);  /* not found, so they weren't already adjacent */
    if (u < v) a = u→arcs;
    else a = v→arcs;  /* the new edge consists of arcs a and a + 1 */
    a→chap_no = (a + 1)→chap_no = k;
  found: ;
  }
```

This code is used in section 22.

26. Administration. The program is now complete except for a few missing organizational details. I will add these after lunch.

27. OK, I'm back; what needs to be done? The main thing is to create the graph itself.

```
⟨ Choose the vertices and put them into an empty graph 27 ⟩ ≡
  if (n > characters) n = characters;
  if (x > n) x = n;
  if (last_chapter > chapters) last_chapter = chapters;
  if (first_chapter > last_chapter) first_chapter = last_chapter + 1;
  new_graph = gb_new_graph(n − x + (bipartite ? last_chapter − first_chapter + 1 : 0));
  if (new_graph ≡ Λ) panic(no_room);      /* out of memory already */
  strcpy(new_graph→util_types, "IZZIISIZZZZZZZ");      /* declare the types of utility fields */
  sprintf(new_graph→id, "%sbook(\"%s\",%lu,%lu,%lu,%lu,%ld,%ld,%ld)", bipartite ? "bi_" : "", title,
      n, x, first_chapter, last_chapter, in_weight, out_weight, seed);
  if (bipartite) {
    mark_bipartite(new_graph, n − x);
    chap_base = new_graph→vertices + (new_graph→n_1 − first_chapter);
  }
  ⟨ Compute the weights and assign vertices to chosen nodes 28 ⟩;
```

This code is used in section 8.

28.

```
⟨ Compute the weights and assign vertices to chosen nodes 28 ⟩ ≡
  for (p = node_block; p < node_block + characters; p++)
    p→key = in_weight * (p→in) + out_weight * (p→out) + #40000000;
  gb_linksort(node_block + characters − 1);
  k = n;      /* we will look at this many nodes */
  { register Vertex *v = new_graph→vertices;      /* the next vertex to define */
    for (j = 127; j ≥ 0; j−−)
      for (p = (node *) gb_sorted[j]; p; p = p→link) {
        if (x > 0) x−−;      /* ignore this node */
        else p→vert = v++;      /* choose this node */
        if (−−k ≡ 0) goto done;
      }
  }
done: ;
```

This code is used in section 27.

29. Once the graph is there, we're ready to fill it in.

```
⟨ Read the data file more carefully and fill the graph as instructed 29 ⟩ ≡
  if (gb_open(file_name) ≠ 0) panic(impossible + 1);
      /* this can't happen, because we were successful before */
  ⟨ Read the data about characters again, noting vertex names and the associated descriptions 17 ⟩;
  if (bipartite) ⟨ Read the chapter information a second time and create the appropriate bipartite edges 20 ⟩
  else ⟨ Read the chapter information a second time and create the appropriate edges for encounters 22 ⟩;
  if (gb_close( ) ≠ 0) panic(impossible + 2);      /* again, can hardly happen the second time around */
```

This code is used in section 8.

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

August 4, 2025 at 09:09

GB_BOOKS

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