1. Introduction. This is GB_GRAPH, the data-structure module used by all GraphBase routines to allocate memory. The basic data types for graph representation are also defined here.

Many examples of how to use these conventions appear in other GraphBase modules. The best introduction to such examples can probably be found in GB_BASIC, which contains subroutines for generating and transforming various classical graphs.

2. The code below is believed to be system-independent; it should produce equivalent results on all systems, assuming that the standard *calloc* and *free* functions of C are available.

However, a test program helps build confidence that everything does in fact work as it should. To make such a test, simply compile and run `test_graph`. This particular test is fairly rudimentary, but it should be passed before more elaborate routines are tested.

```
⟨ test_graph.c 2 ⟩ ≡
#include "gb_graph.h"    /* all users of GB_GRAPH should do this */
  ⟨ Declarations of test variables 19 ⟩
  int main()
  {
    ⟨ Create a small graph 36 ⟩;
    ⟨ Test some intentional errors 18 ⟩;
    ⟨ Check that the small graph is still there 38 ⟩;
    printf("OK,␣the␣gb_graph␣routines␣seem␣to␣work!\n");
    return 0;
  }
```

3. The C code for GB_GRAPH doesn't have a main routine; it's just a bunch of subroutines waiting to be incorporated into programs at a higher level via the system loading routine. Here is the general outline of `gb_graph.c`:

```
#ifdef SYSV
#include <string.h>
#else
#include <strings.h>
#endif
#include <stdio.h>
#include <stdlib.h>
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 8 ⟩
  ⟨ Private declarations 28 ⟩
  ⟨ External declarations 5 ⟩
  ⟨ External functions 13 ⟩
```

4. The type declarations of GB_GRAPH appear also in the header file `gb_graph.h`. For convenience, that header file also incorporates the standard system headers for input/output and string manipulation.

Some system header files define an unsafe macro called *min*, which will interfere with GraphBase use of a useful identifier. We scotch that.

```
⟨ gb_graph.h 4 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#ifdef SYSV
#include <string.h>
#else
#include <strings.h>
#endif
#undef min
  ⟨ Type declarations 8 ⟩
```

See also sections 6, 7, 15, 17, 22, 25, 33, 41, and 42.

5. GraphBase programs often have a "verbose" option, which needs to be enabled by the setting of an external variable. They also tend to have a variable called *panic_code*, which helps identify unusual errors. We might as well declare those variables here.

```
⟨ External declarations 5 ⟩ ≡
  long verbose = 0;     /* nonzero if "verbose" output is desired */
  long panic_code = 0;     /* set nonzero if graph generator returns null pointer */
```

See also sections 14, 24, and 32.

This code is used in section 3.

6. Every external variable should be declared twice in this CWEB file: once for GB_GRAPH itself (the "real" declaration for storage allocation purposes) and once in `gb_graph.h` (for cross-references by GB_GRAPH users).

```
⟨ gb_graph.h 4 ⟩ +≡
  extern long verbose;     /* nonzero if "verbose" output is desired */
  extern long panic_code;     /* set nonzero if graph generator panics */
```

7. When *panic_code* is assigned a nonzero value, one of the symbolic names defined here is used to help pinpoint the problem. Small values indicate memory limitations; values in the 10s and 20s indicate input/output anomalies; values in the 30s and 40s indicate errors in the parameters to a subroutine. Some panic codes stand for cases the author doesn't think will ever arise, although the program checks for them just to be extra safe. Multiple instances of the same type of error within a single subroutine are distinguished by adding an integer; for example, '*syntax_error* + 1' and '*syntax_error* + 2' identify two different kinds of syntax error, as an aid in trouble-shooting. The *early_data_fault* and *late_data_fault* codes are explained further by the value of *io_errors*.

```
⟨ gb_graph.h 4 ⟩ +≡
#define alloc_fault (−1)     /* a previous memory request failed */
#define no_room 1     /* the current memory request failed */
#define early_data_fault 10     /* error detected at beginning of .dat file */
#define late_data_fault 11     /* error detected at end of .dat file */
#define syntax_error 20     /* error detected while reading .dat file */
#define bad_specs 30     /* parameter out of range or otherwise disallowed */
#define very_bad_specs 40     /* parameter far out of range or otherwise stupid */
#define missing_operand 50     /* graph parameter is Λ */
#define invalid_operand 60     /* graph parameter doesn't obey assumptions */
#define impossible 90     /* "this can't happen" */
```

8. Representation of graphs. The GraphBase programs employ a simple and flexible set of data structures to represent and manipulate graphs in computer memory. Vertices appear in a sequential array of **Vertex** records, and the arcs emanating from each vertex appear in a linked list of **Arc** records. There is also a **Graph** record, to provide information about the graph as a whole.

The structure layouts for **Vertex**, **Arc**, and **Graph** records include a number of utility fields that can be used for any purpose by algorithms that manipulate the graphs. Each utility field is a union type that can be either a pointer of various kinds or a (long) integer.

Let's begin the formal definition of these data structures by declaring the union type **util**. The suffixes $.V$, $.A$, $.G$, and $.S$ on the name of a utility variable mean that the variable is a pointer to a vertex, arc, graph, or string, respectively; the suffix $.I$ means that the variable is an integer. (We use one-character names because such names are easy to type when debugging.)

```
⟨ Type declarations 8 ⟩ ≡
  typedef union {
    struct vertex_struct *V;   /* pointer to Vertex */
    struct arc_struct *A;   /* pointer to Arc */
    struct graph_struct *G;   /* pointer to Graph */
    char *S;   /* pointer to string */
    long I;   /* integer */
  } util;
```

See also sections 9, 10, 12, 20, and 34.

This code is used in sections 3 and 4.

9. Each **Vertex** has two standard fields and six utility fields; hence it occupies 32 bytes on most systems, not counting the memory needed for supplementary string data. The standard fields are

arcs, a pointer to an **Arc**;
name, a pointer to a string of characters.

If v points to a **Vertex** and v→*arcs* is Λ, there are no arcs emanating from v. But if v→*arcs* is non-Λ, it points to an **Arc** record representing an arc from v, and that record has a *next* field that points in the same way to the representations of all other arcs from v.

The utility fields are called u, v, w, x, y, z. Macros can be used to give them syntactic sugar in particular applications. They are typically used to record such things as the in-degree or out-degree, or whether a vertex is 'marked'. Utility fields might also link the vertex to other vertices or arcs in one or more lists.

```
⟨ Type declarations 8 ⟩ +≡
  typedef struct vertex_struct {
    struct arc_struct *arcs;   /* linked list of arcs coming out of this vertex */
    char *name;   /* string identifying this vertex symbolically */
    util u, v, w, x, y, z;   /* multipurpose fields */
  } Vertex;
```

10. Each **Arc** has three standard fields and two utility fields. Thus it occupies 20 bytes on most computer systems. The standard fields are

tip, a pointer to a **Vertex**;
next, a pointer to an **Arc**;
len, a (long) integer.

If a points to an **Arc** in the list of arcs from vertex v, it represents an arc of length $a\to len$ from v to $a\to tip$, and the next arc from v in the list is represented by $a\to next$.

The utility fields are called a and b.

⟨ Type declarations 8 ⟩ +≡

```
typedef struct arc_struct {
  struct vertex_struct *tip;   /* the arc points to this vertex */
  struct arc_struct *next;   /* another arc pointing from the same vertex */
  long len;   /* length of this arc */
  util a, b;   /* multipurpose fields */
} Arc;
```

11. Storage allocation. Memory space must be set aside dynamically for vertices, arcs, and their attributes. The GraphBase routines provided by GB_GRAPH accomplish this task with reasonable ease and efficiency by using the concept of memory "areas." The user should first declare an **Area** variable by saying, for example,

Area *s*;

and if this variable isn't static or otherwise known to be zero, it must be cleared initially by saying '*init_area*(s)'. Then any number of subroutine calls of the form '*gb_alloc*(n, s)' can be given; *gb_alloc* will return a pointer to a block of n consecutive bytes, all cleared to zero. Finally, the user can issue the command

gb_free(s);

this statement will return all memory blocks currently allocated to area s, making them available for future allocation.

The number of bytes n specified to *gb_alloc* must be positive, and it should usually be 1000 or more, since this will reduce the number of system calls. Other routines are provided below to allocate smaller amounts of memory, such as the space needed for a single new **Arc**.

If no memory of the requested size is presently available, *gb_alloc* returns the null pointer Λ. In such cases *gb_alloc* also sets the external variable *gb_trouble_code* to a nonzero value. The user can therefore discover whether any one of an arbitrarily long series of allocation requests has failed by making a single test, '**if** (*gb_trouble_code*)'. The value of *gb_trouble_code* should be cleared to zero by every graph generation subroutine; therefore it need not be initialized to zero.

A special macro *gb_typed_alloc*(n, t, s) makes it convenient to allocate the space for n items of type t in area s.

#define *gb_typed_alloc*(n, t, s) ($t*$)*gb_alloc*((**long**)((n) $*$ **sizeof**(t)), s)

12. The implementation of this scheme is almost ridiculously easy. The value of n is increased by twice the number of bytes in a pointer, and the resulting number is rounded upwards if necessary so that it's a multiple of 256. Then memory is allocated using *calloc*. The extra bytes will contain two pointers, one to the beginning of the block and one to the next block associated with the same area variable.

The **Area** type is defined to be an array of length 1. This makes it possible for users to say just 's' instead of '&s' when using an area variable as a parameter.

⟨ Type declarations 8 ⟩ +≡
#define *init_area*(s) $*s = \Lambda$
  struct area_pointers {
    char $*$*first*; /* address of the beginning of this block */
    struct area_pointers $*$*next*; /* address of area pointers in the previously allocated block */
  };
  typedef struct area_pointers $*$**Area**[1];

13. First we round n up, if necessary, so that it's a multiple of the size of a pointer variable. Then we know we can put **area_pointers** into memory at a position n after any address returned by *calloc*. (This logic should work whenever the number of bytes in a pointer variable is a divisor of 256.)

The upper limit on n here is governed by old C conventions in which the first parameter to *calloc* must be less than 2^{16}. Users who need graphs with more than half a million vertices might want to raise this limit on their systems, but they would probably be better off representing large graphs in a more compact way.

Important Note: Programs of the Stanford GraphBase implicitly assume that all memory allocated by *calloc* comes from a single underlying memory array. Pointer values are compared to each other in many places, even when the objects pointed to have been allocated at different times. Strictly speaking, this liberal use of pointer comparisons fails to conform to the restrictions of ANSI Standard C, if the comparison involves a less-than or greater-than relation. Users whose system supports only the strict standard will need to make several dozen changes.

⟨External functions 13⟩ ≡

```
char *gb_alloc(n, s)
    long n;    /* number of consecutive bytes desired */
    Area s;    /* storage area that will contain the new block */
{ long m = sizeof(char *);    /* m is the size of a pointer variable */
  Area t;    /* a temporary pointer */
  char *loc;    /* the block address */
  if (n ≤ 0 ∨ n > #ffff00 − 2 * m) {
    gb_trouble_code |= 2;    /* illegal request */
    return Λ;
  }
  n = ((n + m − 1)/m) * m;    /* round up to multiple of m */
  loc = (char *) calloc((unsigned)((n + 2 * m + 255)/256), 256);
  if (loc) {
    *t = (struct area_pointers *)(loc + n);
    (*t)→first = loc;
    (*t)→next = *s;
    *s = *t;
  } else gb_trouble_code |= 1;
  return loc;
}
```

See also sections 16, 23, 26, 27, 29, 30, 31, 35, 39, 40, 44, 46, 47, and 48.

This code is used in section 3.

14. ⟨External declarations 5⟩ +≡

```
long gb_trouble_code = 0;    /* did gb_alloc return Λ? */
```

15. ⟨`gb_graph.h` 4⟩ +≡

```
extern long gb_trouble_code;    /* anomalies noted by gb_alloc */
```

16. Notice that *gb_free*(s) can be called twice in a row, because the list of blocks is cleared out of the area variable s.

⟨ External functions 13 ⟩ +≡
```
void gb_free(s)
    Area s;
{ Area t;
  while (*s) {
    *t = (*s)→next;
    free((*s)→first);
    *s = *t;
  }
}
```

17. The two external procedures we've defined above should be mentioned in the header file, so let's do that before we forget.

⟨ gb_graph.h 4 ⟩ +≡
```
  extern char *gb_alloc();   /* allocate another block for an area */
#define gb_typed_alloc(n, t, s)   (t*)gb_alloc((long)((n) * sizeof(t)), s)
  extern void gb_free();   /* deallocate all blocks for an area */
```

18. Here we try to allocate 10 million bytes of memory. If we succeed, fine; if not, we verify that the error was properly reported.

(An early draft of this program attempted to allocate memory until all space was exhausted. That tactic provided a more thorough test, but it was a bad idea because it brought certain large systems to their knees; it was terribly unfriendly to other users who were innocently trying to do their own work on the same machine.)

⟨ Test some intentional errors 18 ⟩ ≡
```
if (gb_alloc(0L, s) ≠ Λ ∨ gb_trouble_code ≠ 2) {
  fprintf(stderr, "Allocation error 2 wasn't reported properly!\n"); return −2;
}
for ( ; g→vv.I < 100; g→vv.I++) if (gb_alloc(100000L, s)) {
    g→uu.I++;
    printf(".");
    fflush(stdout);
  }
if (g→uu.I < 100 ∧ gb_trouble_code ≠ 3) {
  fprintf(stderr, "Allocation error 1 wasn't reported properly!\n"); return −1;
}
if (g→uu.I ≡ 0) {
  fprintf(stderr, "I couldn't allocate any memory!\n"); return −3;
}
gb_free(s);   /* we've exhausted memory, let's put some back */
printf("Hey, I allocated %ld00000 bytes successfully. Terrific...\n", g→uu.I);
gb_trouble_code = 0;
```
This code is used in section 2.

19. ⟨ Declarations of test variables 19 ⟩ ≡
```
  Area s;   /* temporary allocations in the test routine */
```
See also section 37.

This code is used in section 2.

20. Growing a graph. Now we're ready to look at the **Graph** type. This is a data structure that can be passed to an algorithm that operates on graphs—to find minimum spanning trees, or strong components, or whatever.

A **Graph** record has seven standard fields and six utility fields. The standard fields are

vertices,	a pointer to an array of **Vertex** records;
n,	the total number of vertices;
m,	the total number of arcs;
id,	a symbolic identification giving parameters of the GraphBase procedure that generated this graph;
util_types,	a symbolic representation of the data types in utility fields;
data,	an **Area** used for **Arc** storage and string storage;
aux_data,	an **Area** used for auxiliary information that some users might want to discard.

The utility fields are called *uu*, *vv*, *ww*, *xx*, *yy*, and *zz*.

As a consequence of these conventions, we can visit all arcs of a graph g by using the following program:

```
Vertex *v;
Arc *a;
for (v = g->vertices; v < g->vertices + g->n; v++)
  for (a = v->arcs; a; a = a->next)
    visit(v, a);
```

⟨ Type declarations 8 ⟩ +≡

```
#define ID_FIELD_SIZE 161
  typedef struct graph_struct {
    Vertex *vertices;   /* beginning of the vertex array */
    long n;   /* total number of vertices */
    long m;   /* total number of arcs */
    char id[ID_FIELD_SIZE];   /* GraphBase identification */
    char util_types[15];   /* usage of utility fields */
    Area data;   /* the main data blocks */
    Area aux_data;   /* subsidiary data blocks */
    util uu, vv, ww, xx, yy, zz;   /* multipurpose fields */
  } Graph;
```

21. The *util_types* field should always hold a string of length 14, followed as usual by a null character to terminate that string. The first six characters of *util_types* specify the usage of utility fields u, v, w, x, y, and z in **Vertex** records; the next two characters give the format of the utility fields in **Arc** records; the last six give the format of the utility fields in **Graph** records. Each character should be either `I` (denoting a **long** integer), `S` (denoting a pointer to a string), `V` (denoting a pointer to a **Vertex**), `A` (denoting a pointer to an **Arc**), `G` (denoting a pointer to a **Graph**), or `Z` (denoting an unused field that remains zero). The default for *util_types* is `"ZZZZZZZZZZZZZZ"`, when none of the utility fields is being used.

For example, suppose that a bipartite graph g is using field *g→uu.I* to specify the size of its first part. Suppose further that g has a string in utility field a of each **Arc** and uses utility field w of **Vertex** records to point to an **Arc**. If g leaves all other utility fields untouched, its *util_types* should be `"ZZAZZZSZIZZZZZ"`.

The *util_types* string is presently examined only by the *save_graph* and *restore_graph* routines, which convert GraphBase graphs from internal data structures to symbolic external files and vice versa. Therefore users need not update the *util_types* when they write algorithms to manipulate graphs, unless they are going to use *save_graph* to output a graph in symbolic form, or unless they are using some other GraphBase-related software that might rely on the conventions of *util_types*. (Such software is not part of the "official" Stanford GraphBase, but it might conceivably exist some day.)

22. Some applications of bipartite graphs require all vertices of the first part to appear at the beginning of the *vertices* array. In such cases, utility field $uu.I$ is traditionally given the symbolic name *n_1*, and it is set equal to the size of that first part. The size of the other part is then $g\to n - g\to n\_1$.

#define *n_1* $uu.I$ /* utility field *uu* may denote size of bipartite first part */

⟨ gb_graph.h 4 ⟩ +≡
#define *n_1* $uu.I$
#define *mark_bipartite*$(g, n1)$ $g\to n\_1 = n1, g\to util\_types[8] =$ 'I'

23. A new graph is created by calling *gb_new_graph*(n), which returns a pointer to a **Graph** record for a graph with n vertices and no arcs. This function also initializes several private variables that are used by the *gb_new_arc*, *gb_new_edge*, *gb_virgin_arc*, and *gb_save_string* procedures below.

We actually reserve space for $n + extra\_n$ vertices, although claiming only n, because several graph manipulation algorithms like to add a special vertex or two to the graphs they deal with.

⟨ External functions 13 ⟩ +≡

```
Graph *gb_new_graph(n)
    long n;     /* desired number of vertices */
{
  cur_graph = (Graph *) calloc(1, sizeof(Graph));
  if (cur_graph) {
    cur_graph→vertices = gb_typed_alloc(n + extra_n, Vertex, cur_graph→data);
    if (cur_graph→vertices) {
      Vertex *p;
      cur_graph→n = n;
      for (p = cur_graph→vertices + n + extra_n − 1; p ≥ cur_graph→vertices; p−−)
        p→name = null_string;
      sprintf(cur_graph→id, "gb_new_graph(%ld)", n);
      strcpy(cur_graph→util_types, "ZZZZZZZZZZZZZZ");
    } else {
      free((char *) cur_graph);
      cur_graph = Λ;
    }
  }
  next_arc = bad_arc = Λ;
  next_string = bad_string = Λ;
  gb_trouble_code = 0;
  return cur_graph;
}
```

24. The value of *extra_n* is ordinarily 4, and it should probably always be at least 4.

⟨ External declarations 5 ⟩ +≡

```
long extra_n = 4;     /* the number of shadow vertices allocated by gb_new_graph */
char null_string[1];     /* a null string constant */
```

25. ⟨ gb_graph.h 4 ⟩ +≡

```
extern long extra_n;     /* the number of shadow vertices allocated by gb_new_graph */
extern char null_string[];     /* a null string constant */
extern void make_compound_id();     /* routine to set one id field from another */
extern void make_double_compound_id();     /* ditto, but from two others */
```

26. The *id* field of a graph is sometimes manufactured from the *id* field of another graph. The following routines do this without allowing the string to get too long after repeated copying.

```
⟨External functions 13⟩ +≡
  void make_compound_id(g, s1, gg, s2)    /* sprintf(g→id, "%s%s%s", s1, gg→id, s2) */
      Graph *g;    /* graph whose id is to be set */
      char *s1;    /* string for the beginning of the new id */
      Graph *gg;    /* graph whose id is to be copied */
      char *s2;    /* string for the end of the new id */
  { int avail = ID_FIELD_SIZE − strlen(s1) − strlen(s2);
    char tmp[ID_FIELD_SIZE];

    strcpy(tmp, gg→id);
    if (strlen(tmp) < avail) sprintf(g→id, "%s%s%s", s1, tmp, s2);
    else sprintf(g→id, "%s%.*s...)%s", s1, avail − 5, tmp, s2);
  }
```

27.
```
⟨External functions 13⟩ +≡
  void make_double_compound_id(g, s1, gg, s2, ggg, s3)
        /* sprintf(g→id, "%s%s%s%s%s", s1, gg→id, s2, ggg→id, s3) */
      Graph *g;    /* graph whose id is to be set */
      char *s1;    /* string for the beginning of the new id */
      Graph *gg;    /* first graph whose id is to be copied */
      char *s2;    /* string for the middle of the new id */
      Graph *ggg;    /* second graph whose id is to be copied */
      char *s3;    /* string for the end of the new id */
  { int avail = ID_FIELD_SIZE − strlen(s1) − strlen(s2) − strlen(s3);

    if (strlen(gg→id) + strlen(ggg→id) < avail) sprintf(g→id, "%s%s%s%s%s", s1, gg→id, s2, ggg→id, s3);
    else sprintf(g→id, "%s%.*s...)%s%.*s...)%s", s1, avail/2 − 5, gg→id, s2, (avail − 9)/2, ggg→id, s3);
  }
```

28. But how do the arcs get there? That's where the private variables in *gb_new_graph* come in. If *next_arc* is unequal to *bad_arc*, it points to an unused **Arc** record in a previously allocated block of **Arc** records. Similarly, *next_string* and *bad_string* are addresses used to place strings into a block of memory allocated for that purpose.

```
⟨Private declarations 28⟩ ≡
  static Arc *next_arc;    /* the next Arc available for allocation */
  static Arc *bad_arc;    /* but if next_arc = bad_arc, that Arc isn't there */
  static char *next_string;    /* the next byte available for storing a string */
  static char *bad_string;    /* but if next_string = bad_string, don't byte */
  static Arc dummy_arc[2];    /* an Arc record to point to in an emergency */
  static Graph dummy_graph;    /* a Graph record that's normally unused */
  static Graph *cur_graph = &dummy_graph;    /* the Graph most recently created */
```

This code is used in section 3.

29. All new **Arc** records that are created by the automatic *next_arc*/*bad_arc* scheme originate in a procedure called *gb_virgin_arc*, which returns the address of a new record having type **Arc**.

When a new block of **Arc** records is needed, we create 102 of them at once. This strategy causes exactly 2048 bytes to be allocated on most computer systems—a nice round number. The routine will still work, however, if 102 is replaced by any positive even number. The new block goes into the *data* area of *cur_graph*.

Graph-building programs do not usually call *gb_virgin_arc* directly; they generally invoke one of the higher-level routines *gb_new_arc* or *gb_new_edge* described below.

If memory space has been exhausted, *gb_virgin_arc* will return a pointer to *dummy_arc*, so that the calling procedure can safely refer to fields of the result even though *gb_trouble_code* is nonzero.

```
#define arcs_per_block 102

⟨External functions 13⟩ +≡
  Arc *gb_virgin_arc()
  { register Arc *cur_arc = next_arc;
    if (cur_arc ≡ bad_arc) {
      cur_arc = gb_typed_alloc(arcs_per_block, Arc, cur_graph→data);
      if (cur_arc ≡ Λ) cur_arc = dummy_arc;
      else {
        next_arc = cur_arc + 1;
        bad_arc = cur_arc + arcs_per_block;
      }
    }
    else next_arc++;
    return cur_arc;
  }
```

30. The routine *gb_new_arc*(u, v, len) creates a new arc of length *len* from vertex u to vertex v. The arc becomes part of the graph that was most recently created by *gb_new_graph*—the graph pointed to by the private variable *cur_graph*. This routine assumes that u and v are both vertices in *cur_graph*.

The new arc will be pointed to by u→*arcs*, immediately after *gb_new_arc*(u, v, len) has acted. If there is no room for the new arc, *gb_trouble_code* is set nonzero, but u→*arcs* will point to the non-Λ record *dummy_arc* so that additional information can safely be stored in its utility fields without risking system crashes before *gb_trouble_code* is tested. However, the linking structure of arcs is apt to be fouled up in such cases; programs should make sure that *gb_trouble_code* ≡ 0 before doing any extensive computation on a graph.

```
⟨External functions 13⟩ +≡
  void gb_new_arc(u, v, len)
      Vertex *u, *v;    /* a newly created arc will go from u to v */
      long len;    /* its length */
  { register Arc *cur_arc = gb_virgin_arc();
    cur_arc→tip = v; cur_arc→next = u→arcs; cur_arc→len = len;
    u→arcs = cur_arc;
    cur_graph→m++;
  }
```

31. An undirected graph has "edges" instead of arcs. We represent an edge by two arcs, one going each way.

The fact that *arcs_per_block* is even means that the *gb_new_edge* routine needs to call *gb_virgin_arc* only once instead of twice.

Caveats: This routine, like *gb_new_arc*, should be used only after *gb_new_graph* has caused the private variable *cur_graph* to point to the graph containing the new edge. The routine *gb_new_edge* must not be used together with *gb_new_arc* or *gb_virgin_arc* when building a graph, unless *gb_new_arc* and *gb_virgin_arc* have been called an even number of times before *gb_new_edge* is invoked.

The new edge will be pointed to by $u\to arcs$ and by $v\to arcs$ immediately after *gb_new_edge* has created it, assuming that $u \neq v$. The two arcs appear next to each other in memory; indeed, *gb_new_edge* rigs things so that $v\to arcs$ is $u\to arcs + 1$ when $u < v$.

On many computers it turns out that the first **Arc** record of every such pair of arcs will have an address that is a multiple of 8, and the second **Arc** record will have an address that is not a multiple of 8 (because the first **Arc** will be 20 bytes long, and because *calloc* always returns a multiple of 8). However, it is not safe to assume this when writing portable code. Algorithms for undirected graphs can still make good use of the fact that arcs for edges are paired, without needing any mod 8 assumptions, if all edges have been created and linked into the graph by *gb_new_edge*: The inverse of an arc a from u to v will be arc $a+1$ if and only if $u < v$ or $a\to next = a + 1$; it will be arc $a - 1$ if and only if $u \geq v$ and $a\to next \neq a + 1$. The condition $a\to next = a + 1$ can hold only if $u = v$.

```
#define gb_new_graph gb_nugraph    /* abbreviations for Procrustean linkers */
#define gb_new_arc gb_nuarc
#define gb_new_edge gb_nuedge
```

⟨External functions 13⟩ +≡

```
void gb_new_edge(u, v, len)
    Vertex *u, *v;    /* new arcs will go from u to v and from v to u */
    long len;    /* their length */
{ register Arc *cur_arc = gb_virgin_arc();
  if (cur_arc ≠ dummy_arc) next_arc++;
  if (u < v) {
    cur_arc→tip = v; cur_arc→next = u→arcs;
    (cur_arc + 1)→tip = u; (cur_arc + 1)→next = v→arcs;
    u→arcs = cur_arc;
    v→arcs = cur_arc + 1;
  } else {
    (cur_arc + 1)→tip = v; (cur_arc + 1)→next = u→arcs;
    u→arcs = cur_arc + 1;    /* do this now in case u ≡ v */
    cur_arc→tip = u; cur_arc→next = v→arcs;
    v→arcs = cur_arc;
  }
  cur_arc→len = (cur_arc + 1)→len = len;
  cur_graph→m += 2;
}
```

32. Sometimes (let us hope rarely) we might need to use a dirty trick hinted at in the previous discussion. On most computers, the mate to arc a will be $a - 1$ if and only if *edge_trick* & (**siz_t**) a is nonzero.

⟨External declarations 5⟩ +≡

```
siz_t edge_trick = sizeof(Arc) − (sizeof(Arc) & (sizeof(Arc) − 1));
```

33. ⟨`gb_graph.h` 4⟩ +≡

```
extern siz_t edge_trick;    /* least significant 1 bit in sizeof(Arc) */
```

34. The type **siz_t** just mentioned should be the type returned by C's **sizeof** operation; it's the basic unsigned type for machine addresses in pointers. ANSI standard C calls this type **size_t**, but we cannot safely use **size_t** in all the GraphBase programs because some older C systems mistakenly define **size_t** to be a signed type.

```
format siz_t int
⟨Type declarations 8⟩ +≡
  typedef unsigned long siz_t;   /* basic machine address, as signless integer */
```

35. Vertices generally have a symbolic name, and we need a place to put such names. The *gb_save_string* function is a convenient utility for this purpose: Given a null-terminated string of any length, *gb_save_string* stashes it away in a safe place and returns a pointer to that place. Memory is conserved by combining strings from the current graph into largish blocks of a convenient size.

Note that *gb_save_string* should be used only after *gb_new_graph* has provided suitable initialization, because the private variable *cur_graph* must point to the graph for which storage is currently being allocated, and because the private variables *next_string* and *bad_string* must also have suitable values.

```
#define string_block_size 1016   /* 1024 − 8 is usually efficient */
⟨External functions 13⟩ +≡
  char *gb_save_string(s)
      register char *s;   /* the string to be copied */
  { register char *p = s;
    register long len;   /* length of the string and the following null character */
    while (*p++) ;   /* advance to the end of the string */
    len = p − s;
    p = next_string;
    if (p + len > bad_string) {   /* not enough room in the current block */
      long size = string_block_size;
      if (len > size) size = len;
      p = gb_alloc(size, cur_graph→data);
      if (p ≡ Λ) return null_string;   /* return a pointer to "" if memory ran out */
      bad_string = p + size;
    }
    while (*s) *p++ = *s++;   /* copy the non-null bytes of the string */
    *p++ = '\0';   /* and append a null character */
    next_string = p;
    return p − len;
  }
```

36. The test routine illustrates some of these basic maneuvers.

```
⟨Create a small graph 36⟩ ≡
  g = gb_new_graph(2L);
  if (g ≡ Λ) {
    fprintf(stderr, "Oops, I couldn't even create a trivial graph!\n");
    return −4;
  }
  u = g→vertices; v = u + 1;
  u→name = gb_save_string("vertex 0");
  v→name = gb_save_string("vertex 1");
```

This code is used in section 2.

37. ⟨ Declarations of test variables 19 ⟩ +≡

```
Graph *g;
Vertex *u, *v;
```

38. If the "edge trick" fails, the standard GraphBase routines are unaffected except for the demonstration program MILES_SPAN. (And that program uses *edge_trick* only when printing verbose comments.)

⟨ Check that the small graph is still there 38 ⟩ ≡

```
if (strncmp(u→name, v→name, 7)) {
  fprintf(stderr, "Something␣is␣fouled␣up␣in␣the␣string␣storage␣machinery!\n");
  return −5;
}
gb_new_edge(v, u, −1L);
gb_new_edge(u, u, 1L);
gb_new_arc(v, u, −1L);
if ((edge_trick & (siz_t)(u→arcs)) ∨ (edge_trick & (siz_t)(u→arcs→next→next)) ∨ ¬(edge_trick &
      (siz_t)(v→arcs→next))) printf("Warning:␣The␣\"edge␣trick\"␣failed!\n");
if (v→name[7] + g→n ≠ v→arcs→next→tip→name[7] + g→m − 2) {      /* '1' + 2 ≠ '0' + 5 − 2 */
  fprintf(stderr, "Sorry,␣the␣graph␣data␣structures␣aren't␣working␣yet.\n");
  return −6;
}
```

This code is used in section 2.

39. Some applications might need to add arcs to several graphs at a time, violating the assumptions stated above about *cur_graph* and the other private variables. The *switch_to_graph* function gets around that restriction, by using the utility slots *ww*, *xx*, *yy*, and *zz* of **Graph** records to save and restore the private variables.

Just say *switch_to_graph*(g) in order to make *cur_graph* be g and to restore the other private variables that are needed by *gb_new_arc*, *gb_virgin_arc*, *gb_new_edge*, and *gb_save_string*. Restriction: The graph g being switched to must have previously been switched from; that is, it must have been *cur_graph* when *switch_to_graph* was called previously. Otherwise its private allocation variables will not have been saved. To meet this restriction, you should say *switch_to_graph*(Λ) just before calling *gb_new_graph*, if you intend to switch back to the current graph later.

(The swap-in-swap-out nature of these conventions may seem inelegant, but convenience and efficiency are more important than elegance when most applications do not need the ability to switch between graphs.)

⟨ External functions 13 ⟩ +≡

```
void switch_to_graph(g)
    Graph *g;
{
  cur_graph→ww.A = next_arc; cur_graph→xx.A = bad_arc;
  cur_graph→yy.S = next_string; cur_graph→zz.S = bad_string;
  cur_graph = (g ? g : &dummy_graph);
  next_arc = cur_graph→ww.A; bad_arc = cur_graph→xx.A;
  next_string = cur_graph→yy.S; bad_string = cur_graph→zz.S;
  cur_graph→ww.A = Λ;
  cur_graph→xx.A = Λ;
  cur_graph→yy.S = Λ;
  cur_graph→zz.S = Λ;
}
```

40. Finally, here's a routine that obliterates an entire graph when it is no longer needed:

```
⟨External functions 13⟩ +≡
  void gb_recycle(g)
        Graph *g;
  {
    if (g) {
      gb_free(g→data);
      gb_free(g→aux_data);
      free((char *) g);      /* the user must not refer to g again */
    }
  }
```

41. ⟨gb_graph.h 4⟩ +≡

```
#define gb_new_graph gb_nugraph      /* abbreviations for external linkage */
#define gb_new_arc gb_nuarc
#define gb_new_edge gb_nuedge
  extern Graph *gb_new_graph();      /* create a new graph structure */
  extern void gb_new_arc();      /* append an arc to the current graph */
  extern Arc *gb_virgin_arc();      /* allocate a new Arc record */
  extern void gb_new_edge();      /* append an edge (two arcs) to the current graph */
  extern char *gb_save_string();      /* store a string in the current graph */
  extern void switch_to_graph();      /* save allocation variables, swap in others */
  extern void gb_recycle();      /* delete a graph structure */
```

42. Searching for vertices. We sometimes want to be able to find a vertex, given its name, and it is nice to do this in a standard way. The following simple subroutines can be used:

hash_in(v) puts the name of vertex v into the hash table;

hash_out(s) finds a vertex named s, if present in the hash table;

hash_setup(g) prepares a hash table for all vertices of graph g;

hash_lookup(s, g) looks up the name s in the hash table of g.

Routines *hash_in* and *hash_out* apply to the current graph being created, while *hash_setup* and *hash_lookup* apply to arbitrary graphs.

Important: Utility fields u and v of each vertex are reserved for use by the search routine when hashing is active. You can crash the system if you try to fool around with these values yourself, or if you use any subroutines that change those fields. The first two characters in the current graph's *util_types* field should be `VV` if the hash table information is to be saved by GB_SAVE.

Warning: Users of this hash scheme must preserve the number of vertices $g\to n$ in the current graph g. If $g\to n$ is changed, the hash table will be worthless, unless *hash_setup* is used to rehash everything.

```
⟨ gb_graph.h 4 ⟩ +≡
  extern void hash_in();     /* input a name to the hash table of current graph */
  extern Vertex *hash_out();     /* find a name in hash table of current graph */
  extern void hash_setup();     /* create a hash table for a given graph */
  extern Vertex *hash_lookup();     /* find a name in a given graph */
```

43. The lookup scheme is quite simple. We compute a more-or-less random value h based on the vertex name, where $0 \le h < n$, assuming that the graph has n vertices. There is a list of all vertices whose hash address is h, starting at $(g\to vertices + h)\to hash\_head$ and linked together in the *hash_link* fields, where *hash_head* and *hash_link* are utility fields $u.V$ and $v.V$.

```
#define hash_link u.V
#define hash_head v.V
```

44. ⟨ External functions 13 ⟩ +≡

```
  void hash_in(v)
      Vertex *v;
  { register char *t = v→name;
    register Vertex *u;
    ⟨ Find vertex u, whose location is the hash code for string t 45 ⟩;
    v→hash_link = u→hash_head;
    u→hash_head = v;
  }
```

45. The hash code for a string $c_1c_2\ldots c_l$ of length l is a nonlinear function of the characters; this function appears to produce reasonably random results between 0 and the number of vertices in the current graph. Simpler approaches were noticeably poorer in the author's tests.

Caution: This hash coding scheme is system-dependent, because it uses the system's character codes. If you create a graph on a machine with ASCII code and save it with GB_SAVE, and if you subsequently ship the resulting text file to some friend whose machine does not use ASCII code, your friend will have to rebuild the hash structure with *hash_setup* before being able to use *hash_lookup* successfully.

#define `HASH_MULT` 314159 /* random multiplier */
#define `HASH_PRIME` 516595003 /* the 27182818th prime; it's less than 2^{29} */

⟨ Find vertex u, whose location is the hash code for string t 45 ⟩ ≡
{ **register long** h;
  for $(h = 0;\ *t;\ t{+}{+})$ {
    $h \mathrel{+}= (h \oplus (h \gg 1)) + \texttt{HASH\_MULT} * (\textbf{unsigned char})\ *t;$
    while $(h \ge \texttt{HASH\_PRIME})\ h \mathrel{-}= \texttt{HASH\_PRIME};$
  }
  $u = cur\_graph\to vertices + (h \% cur\_graph\to n);$
}

This code is used in sections 44 and 46.

46. If the hash function were truly random, the average number of string comparisons made would be less than $(e^2 + 7)/8 \approx 1.80$ on a successful search, and less than $(e^2 + 1)/4 \approx 2.10$ on an unsuccessful search [*Sorting and Searching*, Section 6.4, Eqs. (15) and (16)].

⟨ External functions 13 ⟩ +≡
Vertex $*hash\_out(s)$
    char $*s$;
{ **register char** $*t = s$;
  register Vertex $*u$;
  ⟨ Find vertex u, whose location is the hash code for string t 45 ⟩;
  for $(u = u\to hash\_head;\ u;\ u = u\to hash\_link)$
    if $(strcmp(s, u\to name) \equiv 0)$ **return** u;
  return Λ; /* not found */
}

47. ⟨ External functions 13 ⟩ +≡
void $hash\_setup(g)$
    Graph $*g$;
{ **Graph** $*save\_cur\_graph$;
  if $(g \wedge g\to n > 0)$ { **register Vertex** $*v$;
    $save\_cur\_graph = cur\_graph;$
    $cur\_graph = g;$
    for $(v = g\to vertices;\ v < g\to vertices + g\to n;\ v{+}{+})\ v\to hash\_head = \Lambda;$
    for $(v = g\to vertices;\ v < g\to vertices + g\to n;\ v{+}{+})\ hash\_in(v);$
    $g\to util\_types[0] = g\to util\_types[1] =$ `'V'`; /* indicate usage of *hash_head* and *hash_link* */
    $cur\_graph = save\_cur\_graph;$
  }
}

48. ⟨ External functions 13 ⟩ +≡

```
Vertex *hash_lookup(s, g)
      char *s;
      Graph *g;
  { Graph *save_cur_graph;
    if (g ∧ g->n > 0) { register Vertex *v;
      save_cur_graph = cur_graph;
      cur_graph = g;
      v = hash_out(s);
      cur_graph = save_cur_graph;
      return v;
    }
    else return Λ;
  }
```

49. Index. Here is a list that shows where the identifiers of this program are defined and used.

February 25, 2003 at 09:55

GB_GRAPH

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