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

1. Introduction. This GraphBase module contains the code for two special utility routines, *save_graph* and *restore_graph*, which convert graphs back and forth between the internal representation that is described in GB_GRAPH and a symbolic file format that is described below. Researchers can use these routines to transmit graphs between computers in a machine-independent way, or to use GraphBase graphs with other graph manipulation software that supports the same symbolic format.

All kinds of tricks are possible in the C language, so it is easy to abuse the GraphBase conventions and to create data structures that make sense only on a particular machine. But if users follow the recommended ground rules, *save_graph* will be able to transform their graphs into files that any other GraphBase installation will be able to read with *restore_graph*. The graphs created on remote machines will then be semantically equivalent to the originals.

Restrictions: Strings must contain only standard printable characters, not including \ or " or newline, and must be at most 4095 characters long; the *g→id* string should be at most 154 characters long. All pointers to vertices and arcs must be confined to blocks within the *g→data* area; blocks within *g→aux_data* are not saved or restored. Storage blocks in *g→data* must be "pure"; that is, each block must be entirely devoted either to **Vertex** records, or to **Arc** records, or to characters of strings. The *save_graph* procedure places all **Vertex** records into a single **Vertex** block and all **Arc** records into a single **Arc** block, preserving the relative order of the original records where possible; but it does not preserve the relative order of string data in memory. For example, if *u→name* and *v→name* point to the same memory location in the saved graph, they will point to different memory locations (representing equal strings) in the restored graph. All utility fields must conform to the conventions of the graph's *util_types* string; the `G` option, which leads to graphs within graphs, is not permitted in that string.

```
#define MAX_SV_STRING 4095   /* longest strings supported */
#define MAX_SV_ID 154   /* longest id supported, is less than ID_FIELD_SIZE */
```

```
⟨gb_save.h 1⟩ ≡
  extern long save_graph();
  extern Graph *restore_graph();
```

2. Here is an overview of the C code, `gb_save.c`, for this module:

```
#include "gb_io.h"   /* we use the input/output conventions of GB_IO */
#include "gb_graph.h"   /* and, of course, the data structures of GB_GRAPH */
  ⟨Preprocessor definitions⟩
  ⟨Type declarations 21⟩
  ⟨Private variables 8⟩
  ⟨Private functions 7⟩
  ⟨External functions 4⟩
```

3. External representation of graphs. The internal representation of graphs has been described in GB_GRAPH. We now need to supplement that description by devising an alternative format suitable for human-and-machine-readable files.

The following somewhat contrived example illustrates the simple conventions that we shall follow:

```
* GraphBase graph (util_types IZAZZZZVZZZZSZ,3V,4A)
"somewhat_contrived_example(3.141592653589793238462643383279\
502884197169399375105820974944592307816406286208998628)",1,
3,"pi"
* Vertices
"look",A0,15,A1
"feel",0,-9,A1
"",0,0,0
* Arcs
V0,A2,3,V1
V1,0,5,0
V1,0,-8,1
0,0,0,0
* Checksum 271828
```

The first line specifies the 14 characters of *util_types* and the total number of **Vertex** and **Arc** records; in this case there are 3 vertices and 4 arcs. The next line or lines specify the *id*, n, and m fields of the **Graph** record, together with any utility fields that are not being ignored. In this case, the *id* is a rather long string; a string may be broken into parts by ending the initial parts with a backslash, so that no line of the file has more than 79 characters. The last six characters of *util_types* refer to the utility fields of the **Graph** record, and in this case they are `ZZZZSZ`; so all utility fields are ignored except the second-to-last, *yy*, which is of type string. The *restore_graph* routine will construct a **Graph** record g from this example in which $g\to n = 1$, $g\to m = 3$, and $g\to yy.S =$ `"pi"`.

Notice that the individual field values for a record are separated by commas. If a line ends with a comma, the following line contains additional fields of the same record.

After the **Graph** record fields have been specified, there's a special line '`*␣Vertices`', after which we learn the fields of each vertex in turn. First comes the *name* field, then the *arcs* field, and then any non-ignored utility fields. In this example the *util_types* for **Vertex** records are `IZAZZZ`, so the utility field values are $u.I$ and $w.A$. Let v point to the first **Vertex** record (which incidentally is also pointed to by $g\to vertices$), and let a point to the first **Arc** record. Then in this example we will have $v\to name =$ `"look"`, $v\to arcs = a$, $v\to u.I = 15$, and $v\to w.A = (a + 1)$.

After the **Vertex** records comes a special line '`*␣Arcs`', followed by the fields of each **Arc** record in an entirely analogous way. First comes the *tip* field, then the *next* field, then the *len*, and finally the utility fields (if any). In this example the *util_types* for **Arc** utility fields are `ZV`; hence field a is ignored, and field b is a pointer to a **Vertex**. We will have $a\to tip = v$, $a\to next = (a + 2)$, $a\to len = 3$, and $a\to b.V = (v + 1)$.

The null pointer Λ is denoted by `0`. Furthermore, a **Vertex** pointer is allowed to have the special value `1`, because of conventions explained in GB_GATES. (This special value appears in the fourth field of the third arc in the example above.) The *restore_graph* procedure does not allow **Vertex** pointers to take on constant values greater than 1, nor does it permit the value '`1`' where an **Arc** pointer ought to be.

There should be exactly as many **Vertex** and **Arc** specifications as indicated after the utility types at the beginning of the file. The final **Arc** should then be followed by a special checksum line, which must contain either a number consistent with the data on all the previous lines or a negative value (which is not checked). All information after the checksum line is ignored.

Users should not edit the files produced by *save_graph*, because an incorrect checksum is liable to ruin everything. However, additional lines beginning with '`*`' may be placed as comments at the very beginning of the file; such lines are immune to checksumming.

4. We can establish these conventions firmly in mind by writing the *restore_graph* routine before we write *save_graph*. The subroutine call *restore_graph*("foo.gb") produces a pointer to the graph defined in file "foo.gb", or a null pointer in case that file is unreadable or incorrect. In the latter case, *panic_code* indicates the problem.

⟨ External functions 4 ⟩ ≡
```
Graph *restore_graph(f)
    char *f;    /* the file name */
{ Graph *g = Λ;    /* the graph being restored */
  register char *p;    /* register for string manipulation */
  long m;    /* the number of Arc records to allocate */
  long n;    /* the number of Vertex records to allocate */
  ⟨ Open the file and parse the first line; goto sorry if there's trouble 5 ⟩;
  ⟨ Create the Graph record g and fill in its fields 6 ⟩;
  ⟨ Fill in the fields of all Vertex records 16 ⟩;
  ⟨ Fill in the fields of all Arc records 17 ⟩;
  ⟨ Check the checksum and close the file 18 ⟩;
  return g;
sorry: gb_raw_close(); gb_recycle(g); return Λ;
}
```
See also section 20.

This code is used in section 2.

5. As mentioned above, users can add comment lines at the beginning of the file, if they put a * at the beginning of every such line. But the line that precedes the data proper must adhere to strict standards.

```
#define panic(c) { panic_code = c; goto sorry; }
```

⟨ Open the file and parse the first line; goto sorry if there's trouble 5 ⟩ ≡
```
gb_raw_open(f);
if (io_errors) panic(early_data_fault);    /* can't open the file */
while (1) {
  gb_string(str_buf, ')');
  if (sscanf(str_buf, "*␣GraphBase␣graph␣(util_types␣%14[ZIVSA],%ldV,%ldA", str_buf + 80, &n,
        &m) ≡ 3 ∧ strlen(str_buf + 80) ≡ 14) break;
  if (str_buf[0] ≠ '*') panic(syntax_error);    /* first line is unreadable */
}
```
This code is used in section 4.

6. The previous code has placed the graph's *util_types* into location *str_buf* +80 and verified that it contains precisely 14 characters, all belonging to the set {Z, I, V, S, A}.

⟨ Create the **Graph** record g and fill in its fields 6 ⟩ ≡

```
g = gb_new_graph(0L);
if (g ≡ Λ) panic(no_room);      /* out of memory before we're even started */
gb_free(g→data);
g→vertices = verts = gb_typed_alloc(n ≡ 0 ? 1 : n, Vertex, g→data);
last_vert = verts + n;
arcs = gb_typed_alloc(m ≡ 0 ? 1 : m, Arc, g→data);
last_arc = arcs + m;
if (gb_trouble_code) panic(no_room + 1);      /* not enough room for vertices and arcs */
strcpy(g→util_types, str_buf + 80);
gb_newline();
if (gb_char() ≠ '"') panic(syntax_error + 1);      /* missing quotes before graph id string */
p = gb_string(g→id, '"');
if (*(p − 2) ≡ '\n' ∧ *(p − 3) ≡ '\\' ∧ p > g→id + 2) {
   gb_newline();
   gb_string(p − 3, '"');
}
if (gb_char() ≠ '"') panic(syntax_error + 2);      /* missing quotes after graph id string */
⟨ Fill in g→n, g→m, and g's utility fields 15 ⟩;
```

This code is used in section 4.

7. The *util_types* and *id* fields are slightly different from other string fields, because we store them directly in the **Graph** record instead of storing a pointer. The other fields to be filled by *restore_graph* can all be done by a macro called *fillin*, which invokes a subroutine called *fill_field*. The first parameter to *fillin* is the address of a field in a record; the second parameter is one of the codes {Z, I, V, S, A}. A global variable *comma_expected* is nonzero when this field is not the first in its record.

The value returned by *fill_field* is nonzero if something goes wrong.

We assume here that a utility field takes exactly as much space as a field of any of its constituent types.

```
#define fillin(l,t)  if (fill_field((util *) &(l),t)) goto sorry

⟨ Private functions 7 ⟩ ≡
  static long fill_field(l,t)
      util *l;    /* location of field to be filled in */
      char t;    /* its type code */
  { register char c;    /* character just read */
    if (t ≠ 'Z' ∧ comma_expected) {
      if (gb_char() ≠ ',') return (panic_code = syntax_error − 1);    /* missing comma */
      if (gb_char() ≡ '\n') gb_newline();
      else gb_backup();
    }
    else comma_expected = 1;
    c = gb_char();
    switch (t) {
    case 'I': ⟨ Fill in a numeric field 9 ⟩;
    case 'V': ⟨ Fill in a vertex pointer 10 ⟩;
    case 'S': ⟨ Fill in a string pointer 12 ⟩;
    case 'A': ⟨ Fill in an arc pointer 11 ⟩;
    default: gb_backup(); break;
    }
    return panic_code;
  }
```

See also sections 14, 25, 35, 36, 37, and 39.

This code is used in section 2.

8. Some of the communication between *restore_graph* and *fillin* is best done via global variables.

```
⟨ Private variables 8 ⟩ ≡
  static long comma_expected;    /* should fillin look for a comma? */
  static Vertex *verts;    /* beginning of the block of Vertex records */
  static Vertex *last_vert;    /* end of the block of Vertex records */
  static Arc *arcs;    /* beginning of the block of Arc records */
  static Arc *last_arc;    /* end of the block of Arc records */
```

See also sections 13, 19, 22, and 34.

This code is used in section 2.

9. ⟨ Fill in a numeric field 9 ⟩ ≡

```
  if (c ≡ '-') l→I = −gb_number(10);
  else {
    gb_backup();
    l→I = gb_number(10);
  }
  break;
```

This code is used in section 7.

10. ⟨ Fill in a vertex pointer 10 ⟩ ≡

```
if (c ≡ 'V') {
  l→V = verts + gb_number(10);
  if (l→V ≥ last_vert ∨ l→V < verts) panic_code = syntax_error − 2;   /* vertex address too big */
} else if (c ≡ '0' ∨ c ≡ '1') l→I = c − '0';
else panic_code = syntax_error − 3;   /* vertex numeric address illegal */
break;
```

This code is used in section 7.

11. ⟨ Fill in an arc pointer 11 ⟩ ≡

```
if (c ≡ 'A') {
  l→A = arcs + gb_number(10);
  if (l→A ≥ last_arc ∨ l→A < arcs) panic_code = syntax_error − 4;   /* arc address too big */
} else if (c ≡ '0') l→A = Λ;
else panic_code = syntax_error − 5;   /* arc numeric address illegal */
break;
```

This code is used in section 7.

12. We can restore a string slightly longer than the strings we can save.

⟨ Fill in a string pointer 12 ⟩ ≡

```
if (c ≠ '"') panic_code = syntax_error − 6;   /* missing quotes at beginning of string */
else { register char *p;
  p = gb_string(item_buf, '"');
  while (*(p − 2) ≡ '\n' ∧ *(p − 3) ≡ '\\' ∧ p > item_buf + 2 ∧ p ≤ buffer) {
    gb_newline();
    p = gb_string(p − 3, '"');   /* splice a broken string together */
  }
  if (gb_char() ≠ '"') panic_code = syntax_error − 7;   /* missing quotes at end of string */
  else if (item_buf[0] ≡ '\0') l→S = null_string;
  else l→S = gb_save_string(item_buf);
}
break;
```

This code is used in section 7.

13.
```
#define buffer (&item_buf[MAX_SV_STRING + 3])   /* the last 81 chars of item_buf */
```

⟨ Private variables 8 ⟩ +≡

```
static char item_buf[MAX_SV_STRING + 3 + 81];   /* an item to be output */
```

14. When all fields of a record have been filled in, we call *finish_record* and hope that it returns 0.

⟨ Private functions 7 ⟩ +≡

```
static long finish_record()
{
  if (gb_char() ≠ '\n') return (panic_code = syntax_error − 8);   /* garbage present */
  gb_newline();
  comma_expected = 0;
  return 0;
}
```

15. ⟨ Fill in *g*→*n*, *g*→*m*, and *g*'s utility fields 15 ⟩ ≡

```
panic_code = 0;
comma_expected = 1;
fillin(g->n, 'I');
fillin(g->m, 'I');
fillin(g->uu, g->util_types[8]);
fillin(g->vv, g->util_types[9]);
fillin(g->ww, g->util_types[10]);
fillin(g->xx, g->util_types[11]);
fillin(g->yy, g->util_types[12]);
fillin(g->zz, g->util_types[13]);
if (finish_record()) goto sorry;
```

This code is used in section 6.

16. The rest is easy.

⟨ Fill in the fields of all **Vertex** records 16 ⟩ ≡

```
{ register Vertex *v;
  gb_string(str_buf, '\n');
  if (strcmp(str_buf, "* Vertices") ≠ 0)
    panic(syntax_error + 3);    /* introductory line for vertices is missing */
  gb_newline();
  for (v = verts; v < last_vert; v++) {
    fillin(v->name, 'S');
    fillin(v->arcs, 'A');
    fillin(v->u, g->util_types[0]);
    fillin(v->v, g->util_types[1]);
    fillin(v->w, g->util_types[2]);
    fillin(v->x, g->util_types[3]);
    fillin(v->y, g->util_types[4]);
    fillin(v->z, g->util_types[5]);
    if (finish_record()) goto sorry;
  }
}
```

This code is used in section 4.

17. ⟨ Fill in the fields of all **Arc** records 17 ⟩ ≡

```
{ register Arc *a;
  gb_string(str_buf, '\n');
  if (strcmp(str_buf, "* Arcs") ≠ 0) panic(syntax_error + 4);
      /* introductory line for arcs is missing */
  gb_newline();
  for (a = arcs; a < last_arc; a++) {
    fillin(a->tip, 'V');
    fillin(a->next, 'A');
    fillin(a->len, 'I');
    fillin(a->a, g->util_types[6]);
    fillin(a->b, g->util_types[7]);
    if (finish_record()) goto sorry;
  }
}
```

This code is used in section 4.

18. ⟨ Check the checksum and close the file 18 ⟩ ≡

```
{ long s;
  gb_string(str_buf, '\n');
  if (sscanf(str_buf, "* Checksum %ld", &s) ≠ 1) panic(syntax_error + 5);
      /* checksum line is missing */
  if (gb_raw_close() ≠ s ∧ s ≥ 0) panic(late_data_fault);   /* checksum does not match */
}
```

This code is used in section 4.

19. Saving a graph. Now that we know how to restore a graph, once it has been saved, we are ready to write the *save_graph* routine.

Users say *save_graph*(g, `"foo.gb"`); our job is to create a file `"foo.gb"` from which the subroutine call *restore_graph*(`"foo.gb"`) will be able to reconstruct a graph equivalent to g, assuming that g meets the restrictions stated earlier. If nothing goes wrong, *save_graph* should return the value zero. Otherwise it should return an encoded trouble report.

We will set things up so that *save_graph* produces a syntactically correct file `"foo.gb"` in almost every case, with explicit error indications written at the end of the file whenever certain aspects of the given graph have had to be changed. The value -1 will be returned if $g \equiv \Lambda$; the value -2 will be returned if $g \neq \Lambda$ but the file `"foo.gb"` could not be opened for output; the value -3 will be returned if memory is exhausted. In other cases a file `"foo.gb"` will be created.

Here is a list of things that might go wrong, and the corresponding corrective actions to be taken in each case, assuming that *save_graph* does create a file:

```
#define bad_type_code #1        /* illegal character, is changed to 'Z' */
#define string_too_long #2      /* extralong string, is truncated */
#define addr_not_in_data_area #4  /* address out of range, is changed to Λ */
#define addr_in_mixed_block #8  /* address not in pure block, is Λified */
#define bad_string_char #10     /* illegal string character, is changed to '?' */
#define ignored_data #20        /* nonzero value in 'Z' format, is not output */
```

⟨ Private variables 8 ⟩ +≡

```
  static long anomalies;     /* problems accumulated by save_graph */
  static FILE *save_file;    /* the file being written */
```

20. ⟨ External functions 4 ⟩ +≡

```
  long save_graph(g, f)
      Graph *g;     /* graph to be saved */
      char *f;      /* name of the file to be created */
  { ⟨ Local variables for save_graph 24 ⟩
    if (g ≡ Λ ∨ g→vertices ≡ Λ) return −1;     /* where is g? */
    anomalies = 0;
    ⟨ Figure out the extent of g's internal records 27 ⟩;
    save_file = fopen(f, "w");
    if (¬save_file) return −2;     /* oops, the operating system won't cooperate */
    ⟨ Translate g into external format 30 ⟩;
    ⟨ Make notes at the end of the file about any changes that were necessary 46 ⟩;
    fclose(save_file);
    gb_free(working_storage);
    return anomalies;
  }
```

21. The main difficulty faced by *save_graph* is the problem of translating vertex and arc pointers into symbolic form. A graph's vertices usually appear in a single block, *g→vertices*, but its arcs usually appear in separate blocks that were created whenever the *gb_new_arc* routine needed more space. Other blocks, created by *gb_save_string*, are usually also present in the *g→data* area. We need to classify the various data blocks. We also want to be able to handle graphs that have been created with homegrown methods of memory allocation, because GraphBase structures need not conform to the conventions of *gb_new_arc* and *gb_save_string*.

A simple data structure based on **block_rep** records will facilitate our task. Each **block_rep** will be set up to contain the information we need to know about a particular block of data accessible from *g→data*. Such blocks are classified into four categories, identified by the *cat* field in a **block_rep**:

```
#define unk 0   /* cat value for blocks of unknown nature */
#define ark 1   /* cat value for blocks assumed to hold Arc records */
#define vrt 2   /* cat value for blocks assumed to hold Vertex records */
#define mxt 3   /* cat value for blocks being used for more than one purpose */
```

⟨ Type declarations 21 ⟩ ≡

```
typedef struct {
  char *start_addr;   /* starting address of a data block */
  char *end_addr;   /* ending address of a data block */
  long offset;   /* index number of first record in the block, if known */
  long cat;   /* cat code for the block */
  long expl;   /* have we finished exploring this block? */
} block_rep;
```

This code is used in section 2.

22. The **block_rep** records don't need to be linked together in any fancy way, because there usually aren't very many of them. We will simply create an array, organized in decreasing order of *start_addr* and *end_addr*, with a dummy record standing as a sentinel at the end.

A system-dependent change might be necessary in the following code, if pointer values can be longer than 32 bits, or if comparisons between pointers are undefined.

⟨ Private variables 8 ⟩ +≡

```
static block_rep *blocks;   /* beginning of table of block representatives */
static Area working_storage;
```

23. Initially we set the *end_addr* field to the location following a block's data area. Later we will change it as explained below.

The code in this section uses the fact that all bits of storage blocks are zero until set nonzero. In particular, the *cat* field of each **block_rep** will initially be *unk*, and the *expl* will be zero; the *start_addr* and *end_addr* of the sentinel record will be zero.

⟨ Initialize the *blocks* array 23 ⟩ ≡

```
{ Area t;    /* variable that runs through g→data */
  for (*t = *(g→data), block_count = 0; *t; *t = (*t)→next) block_count++;
  blocks = gb_typed_alloc(block_count + 1, block_rep, working_storage);
  if (blocks ≡ Λ) return −3;    /* out of memory */
  for (*t = *(g→data), block_count = 0; *t; *t = (*t)→next, block_count++) {
    cur_block = blocks + block_count;
    while (cur_block > blocks ∧ (cur_block − 1)→start_addr < (*t)→first) {
      cur_block→start_addr = (cur_block − 1)→start_addr;
      cur_block→end_addr = (cur_block − 1)→end_addr;
      cur_block−−;
    }
    cur_block→start_addr = (*t)→first;
    cur_block→end_addr = (char *) *t;
  }
}
```

This code is used in section 27.

24. ⟨ Local variables for *save_graph* 24 ⟩ ≡

```
register block_rep *cur_block;    /* the current block of interest */
long block_count;    /* how many blocks have we processed? */
```

See also section 31.

This code is used in section 20.

25. The *save_graph* routine makes two passes over the graph. The goal of the first pass is reconnaissance: We try to see where everything is, and we prune off parts that don't conform to the restrictions. When we get to the second pass, our task will then be almost trivial. We will be able to march through the known territory and spew out a copy of what we encounter. (Items that are "pruned" are not actually removed from g itself, only from the portion of g that is saved.)

The first pass is essentially a sequence of calls of the *lookup* macro, which looks at one field of one record and notes whether the existence of this field extends the known boundaries of the graph. The *lookup* macro is a shorthand notation for calling the *classify* subroutine. We make the same assumption about field sizes as the *fill_field* routine did above.

```
#define lookup(l, t) classify((util *) &(l), t)     /* explore field l of type t */
⟨Private functions 7⟩ +≡
  static void classify(l, t)
      util *l;     /* location of field to be classified */
      char t;     /* its type code, from the set {Z, I, V, S, A} */
  { register block_rep *cur_block;
    register char *loc;
    register long tcat;     /* category corresponding to t */
    register long tsize;     /* record size corresponding to t */
    switch (t) {
    default: return;
    case 'V':
      if (l→I ≡ 1) return;
      tcat = vrt;
      tsize = sizeof(Vertex);
      break;
    case 'A': tcat = ark;
      tsize = sizeof(Arc);
      break;
    }
    if (l→I ≡ 0) return;
    ⟨Classify a pointer variable 26⟩;
  }
```

26. Here we know that l points to a **Vertex** or to an **Arc**, according as *tcat* is *vrt* or *ark*. We need to check that this doesn't violate any assumptions about all such pointers lying in pure blocks within the *g→data* area.

```
⟨Classify a pointer variable 26⟩ ≡
  loc = (char *) l→V;
  for (cur_block = blocks; cur_block→start_addr > loc; cur_block++) ;
  if (loc < cur_block→end_addr) {
    if ((loc − cur_block→start_addr) % tsize ≠ 0 ∨ loc + tsize > cur_block→end_addr) cur_block→cat = mxt;
    if (cur_block→cat ≡ unk) cur_block→cat = tcat;
    else if (cur_block→cat ≠ tcat) cur_block→cat = mxt;
  }
```

This code is used in section 25.

27. We go through the list of blocks repeatedly until we reach a stable situation in which every *vrt* or *ark* block has been explored.

⟨ Figure out the extent of *g*'s internal records 27 ⟩ ≡

```
{ long activity;
  ⟨ Initialize the blocks array 23 ⟩;
  lookup(g→vertices, 'V');
  lookup(g→uu, g→util_types[8]);
  lookup(g→vv, g→util_types[9]);
  lookup(g→ww, g→util_types[10]);
  lookup(g→xx, g→util_types[11]);
  lookup(g→yy, g→util_types[12]);
  lookup(g→zz, g→util_types[13]);
  do { activity = 0;
    for (cur_block = blocks; cur_block→end_addr; cur_block++) {
      if (cur_block→cat ≡ vrt ∧ ¬cur_block→expl) ⟨ Explore a block of supposed vertex records 28 ⟩
      else if (cur_block→cat ≡ ark ∧ ¬cur_block→expl) ⟨ Explore a block of supposed arc records 29 ⟩
      else continue;
      cur_block→expl = activity = 1;
    }
  } while (activity);
}
```

This code is used in section 20.

28. While we are exploring a block, the *lookup* routine might classify a previously explored block (or even the current block) as *mxt*. Therefore some data we assumed would be accessible will actually be removed from the graph; contradictions that arose might no longer exist. But we plunge ahead anyway, because we aren't going to try especially hard to "save" portions of graphs that violate our ground rules.

⟨ Explore a block of supposed vertex records 28 ⟩ ≡

```
{ register Vertex *v;
  for (v = (Vertex *) cur_block→start_addr;
        (char *)(v + 1) ≤ cur_block→end_addr ∧ cur_block→cat ≡ vrt; v++) {
    lookup(v→arcs, 'A');
    lookup(v→u, g→util_types[0]);
    lookup(v→v, g→util_types[1]);
    lookup(v→w, g→util_types[2]);
    lookup(v→x, g→util_types[3]);
    lookup(v→y, g→util_types[4]);
    lookup(v→z, g→util_types[5]);
  }
}
```

This code is used in section 27.

29. ⟨ Explore a block of supposed arc records 29 ⟩ ≡

```
{ register Arc *a;
  for (a = (Arc *) cur_block→start_addr;
        (char *)(a + 1) ≤ cur_block→end_addr ∧ cur_block→cat ≡ ark; a++) {
    lookup(a→tip, 'V');
    lookup(a→next, 'A');
    lookup(a→a, g→util_types[6]);
    lookup(a→b, g→util_types[7]);
  }
}
```

This code is used in section 27.

30. OK, the first pass is complete. And the second pass is routine:

⟨ Translate g into external format 30 ⟩ ≡

```
⟨ Orient the blocks table for translation 32 ⟩;
⟨ Initialize the output buffer mechanism and output the first line 38 ⟩;
⟨ Translate the Graph record 41 ⟩;
⟨ Translate the Vertex records 42 ⟩;
⟨ Translate the Arc records 44 ⟩;
⟨ Output the checksum line 45 ⟩;
```

This code is used in section 20.

31. During this pass we decrease the *end_addr* field of a **block_rep**, so that it points to the first byte of the final record in a *vrt* or *ark* block.

The variables m and n are set to the number of arc records and vertex records, respectively.

⟨ Local variables for *save_graph* 24 ⟩ +≡

```
long m;   /* total number of Arc records to be translated */
long n;   /* total number of Vertex records to be translated */
register long s;   /* accumulator register for arithmetic calculations */
```

32. One tricky point needs to be observed, in the unusual case that there are two or more blocks of **Vertex** records: The base block *g→vertices* must come first in the final ordering. (This is the only exception to the rule that **Vertex** and **Arc** records each retain their relative order with respect to less-than and greater-than.)

⟨ Orient the *blocks* table for translation 32 ⟩ ≡

```
m = 0; ⟨ Set n to the size of the block that starts with g→vertices 33 ⟩;
for (cur_block = blocks + block_count − 1; cur_block ≥ blocks; cur_block −−) {
  if (cur_block→cat ≡ vrt) {
    s = (cur_block→end_addr − cur_block→start_addr)/sizeof(Vertex);
    cur_block→end_addr = cur_block→start_addr + ((s − 1) * sizeof(Vertex));
    if (cur_block→start_addr ≠ (char *) g→vertices) {
      cur_block→offset = n; n += s;
    }   /* otherwise cur_block→offset remains zero */
  } else if (cur_block→cat ≡ ark) {
    s = (cur_block→end_addr − cur_block→start_addr)/sizeof(Arc);
    cur_block→end_addr = cur_block→start_addr + ((s − 1) * sizeof(Arc));
    cur_block→offset = m;
    m += s;
  }
}
```

This code is used in section 30.

33. ⟨ Set n to the size of the block that starts with *g→vertices* 33 ⟩ ≡

```
n = 0;
for (cur_block = blocks + block_count − 1; cur_block ≥ blocks; cur_block −−)
  if (cur_block→start_addr ≡ (char *) g→vertices) {
    n = (cur_block→end_addr − cur_block→start_addr)/sizeof(Vertex);
    break;
  }
```

This code is used in section 32.

34. We will store material to be output in the *buffer* array, so that we can compute the correct checksum.

⟨ Private variables 8 ⟩ +≡

```
static char *buf_ptr;    /* the first unfilled position in buffer */
static long magic;    /* the checksum */
```

35. ⟨ Private functions 7 ⟩ +≡

```
static void flushout()    /* output the buffer to save_file */
{
  *buf_ptr++ = '\n';
  *buf_ptr = '\0';
  magic = new_checksum(buffer, magic);
  fputs(buffer, save_file);
  buf_ptr = buffer;
}
```

36. If a supposed string pointer is zero, we output the null string. (This case arises when a string field has not been initialized, for example in vertices and arcs that have been allocated but not used.)

⟨ Private functions 7 ⟩ +≡

```
static void prepare_string(s)
    char *s;    /* string that is moved to item_buf */
{ register char *p, *q;
  item_buf[0] = '"';
  p = &item_buf[1];
  if (s ≡ 0) goto sready;
  for (q = s; *q ∧ p ≤ &item_buf[MAX_SV_STRING]; q++, p++)
    if (*q ≡ '"' ∨ *q ≡ '\n' ∨ *q ≡ '\\' ∨ imap_ord(*q) ≡ unexpected_char) {
      anomalies |= bad_string_char;
      *p = '?';
    } else *p = *q;
  if (*q) anomalies |= string_too_long;
sready: *p = '"';
  *(p + 1) = '\0';
}
```

37. The main idea of this part of the program is to format an item into *item_buf*, then move it to *buffer*, making sure that there is always room for a comma.

```
#define append_comma *buf_ptr++ = ','
⟨Private functions 7⟩ +≡
  static void move_item()
  { register long l = strlen(item_buf);
    if (buf_ptr + l > &buffer[78]) {
      if (l ≤ 78) flushout();
      else { register char *p = item_buf;
        if (buf_ptr > &buffer[77]) flushout();      /* no room for initial " */
        do {
          for ( ; buf_ptr < &buffer[78]; buf_ptr++, p++, l--) *buf_ptr = *p;
          *buf_ptr++ = '\\';
          flushout();
        } while (l > 78);
        strcpy(buffer, p);
        buf_ptr = &buffer[l];
        return;
      }
    }
    strcpy(buf_ptr, item_buf);
    buf_ptr += l;
  }
```

38. ⟨Initialize the output buffer mechanism and output the first line 38⟩ ≡

```
  buf_ptr = buffer;
  magic = 0;
  fputs("*␣GraphBase␣graph␣(util_types␣", save_file);
  { register char *p;
    for (p = g→util_types; p < g→util_types + 14; p++)
      if (*p ≡ 'Z' ∨ *p ≡ 'I' ∨ *p ≡ 'V' ∨ *p ≡ 'S' ∨ *p ≡ 'A') fputc(*p, save_file);
      else fputc('Z', save_file);
  }
  fprintf(save_file, ",%ldV,%ldA)\n", n, m);
```

This code is used in section 30.

39. A macro called *trans*, which is sort of an inverse to *fillin*, takes care of the main work in the second pass.

```
#define trans(l,t) translate_field((util *) &(l),t)

⟨ Private functions 7 ⟩ +≡
  static void translate_field(l,t)
      util *l;   /* address of field to be output in symbolic form */
      char t;    /* type of formatting desired */
  { register block_rep *cur_block;
    register char *loc;
    register long tcat;   /* category corresponding to t */
    register long tsize;   /* record size corresponding to t */
    if (comma_expected) append_comma;
    else comma_expected = 1;
    switch (t) {
    default: anomalies |= bad_type_code;   /* fall through to case Z */
    case 'Z': buf_ptr--;   /* forget spurious comma */
      if (l->I) anomalies |= ignored_data;
      return;
    case 'I': numeric: sprintf(item_buf,"%ld",l->I); goto ready;
    case 'S': prepare_string(l->S); goto ready;
    case 'V':
      if (l->I ≡ 1) goto numeric;
      tcat = vrt; tsize = sizeof(Vertex); break;
    case 'A': tcat = ark; tsize = sizeof(Arc); break;
    }
    ⟨ Translate a pointer variable 40 ⟩;
  ready: move_item();
  }
```

40. ⟨ Translate a pointer variable 40 ⟩ ≡

```
  loc = (char *) l->V;
  item_buf[0] = '0'; item_buf[1] = '\0';   /* Λ will be the default */
  if (loc ≡ Λ) goto ready;
  for (cur_block = blocks; cur_block->start_addr > loc; cur_block++) ;
  if (loc > cur_block->end_addr) {
    anomalies |= addr_not_in_data_area;
    goto ready;
  }
  if (cur_block->cat ≠ tcat ∨ (loc − cur_block->start_addr) % tsize ≠ 0) {
    anomalies |= addr_in_mixed_block;
    goto ready;
  }
  sprintf(item_buf,"%c%ld",t,cur_block->offset + ((loc − cur_block->start_addr)/tsize));
```

This code is used in section 39.

41. ⟨ Translate the **Graph** record 41 ⟩ ≡

```
prepare_string(g→id);
if (strlen(g→id) > MAX_SV_ID) {
  strcpy(item_buf + MAX_SV_ID + 1, "\"");
  anomalies |= string_too_long;
}
move_item();
comma_expected = 1;
trans(g→n, 'I');
trans(g→m, 'I');
trans(g→uu, g→util_types[8]);
trans(g→vv, g→util_types[9]);
trans(g→ww, g→util_types[10]);
trans(g→xx, g→util_types[11]);
trans(g→yy, g→util_types[12]);
trans(g→zz, g→util_types[13]);
flushout();
```

This code is used in section 30.

42. ⟨ Translate the **Vertex** records 42 ⟩ ≡

```
{ register Vertex *v;
  fputs("*␣Vertices\n", save_file);
  for (cur_block = blocks + block_count − 1; cur_block ≥ blocks; cur_block−−)
    if (cur_block→cat ≡ vrt ∧ cur_block→offset ≡ 0) ⟨ Translate all Vertex records in cur_block 43 ⟩;
  for (cur_block = blocks + block_count − 1; cur_block ≥ blocks; cur_block−−)
    if (cur_block→cat ≡ vrt ∧ cur_block→offset ≠ 0) ⟨ Translate all Vertex records in cur_block 43 ⟩;
}
```

This code is used in section 30.

43. ⟨ Translate all **Vertex** records in *cur_block* 43 ⟩ ≡

```
for (v = (Vertex *) cur_block→start_addr; v ≤ (Vertex *) cur_block→end_addr; v++) {
  comma_expected = 0;
  trans(v→name, 'S');
  trans(v→arcs, 'A');
  trans(v→u, g→util_types[0]);
  trans(v→v, g→util_types[1]);
  trans(v→w, g→util_types[2]);
  trans(v→x, g→util_types[3]);
  trans(v→y, g→util_types[4]);
  trans(v→z, g→util_types[5]);
  flushout();
}
```

This code is used in section 42.

44. ⟨ Translate the **Arc** records 44 ⟩ ≡

```
{ register Arc *a;
  fputs("*␣Arcs\n", save_file);
  for (cur_block = blocks + block_count − 1; cur_block ≥ blocks; cur_block −−)
    if (cur_block→cat ≡ ark)
      for (a = (Arc *) cur_block→start_addr; a ≤ (Arc *) cur_block→end_addr; a++) {
        comma_expected = 0;
        trans(a→tip, 'V');
        trans(a→next, 'A');
        trans(a→len, 'I');
        trans(a→a, g→util_types[6]);
        trans(a→b, g→util_types[7]);
        flushout();
      }
}
```

This code is used in section 30.

45. ⟨ Output the checksum line 45 ⟩ ≡

```
fprintf(save_file, "*␣Checksum␣%ld\n", magic);
```

This code is used in section 30.

46. ⟨ Make notes at the end of the file about any changes that were necessary 46 ⟩ ≡

```
if (anomalies) {
  fputs(">␣WARNING:␣I␣had␣trouble␣making␣this␣file␣from␣the␣given␣graph!\n", save_file);
  if (anomalies & bad_type_code)
    fputs(">>␣The␣original␣util_types␣had␣to␣be␣corrected.\n", save_file);
  if (anomalies & ignored_data)
    fputs(">>␣Some␣data␣suppressed␣by␣Z␣format␣was␣actually␣nonzero.\n", save_file);
  if (anomalies & string_too_long)
    fputs(">>␣At␣least␣one␣long␣string␣had␣to␣be␣truncated.\n", save_file);
  if (anomalies & bad_string_char)
    fputs(">>␣At␣least␣one␣string␣character␣had␣to␣be␣changed␣to␣'?'.\n", save_file);
  if (anomalies & addr_not_in_data_area)
    fputs(">>␣At␣least␣one␣pointer␣led␣out␣of␣the␣data␣area.\n", save_file);
  if (anomalies & addr_in_mixed_block)
    fputs(">>␣At␣least␣one␣data␣block␣had␣an␣illegal␣mixture␣of␣records.\n", save_file);
  if (anomalies & (addr_not_in_data_area + addr_in_mixed_block))
    fputs(">>␣␣(Pointers␣to␣improper␣data␣have␣been␣changed␣to␣0.)\n", save_file);
  fputs(">␣You␣should␣be␣able␣to␣read␣this␣file␣with␣restore_graph,\n", save_file);
  fputs(">␣but␣the␣graph␣you␣get␣won't␣be␣exactly␣like␣the␣original.\n", save_file);
}
```

This code is used in section 20.

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

February 25, 2003 at 09:59

GB_SAVE

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