Important: Before reading MILES_SPAN, please read or at least skim the program for GB_MILES.

1. Minimum spanning trees. A classic paper by R. L. Graham and Pavol Hell about the history of algorithms to find the minimum-length spanning tree of a graph [*Annals of the History of Computing* **7** (1985), 43–57] describes three main approaches to that problem. Algorithm 1, "two nearest fragments," repeatedly adds a shortest edge that joins two hitherto unconnected fragments of the graph; this algorithm was first published by J. B. Kruskal in 1956. Algorithm 2, "nearest neighbor," repeatedly adds a shortest edge that joins a particular fragment to a vertex not in that fragment; this algorithm was first published by V. Jarník in 1930. Algorithm 3, "all nearest fragments," repeatedly adds to each existing fragment the shortest edge that joins it to another fragment; this method, seemingly the most sophisticated in concept, also turns out to be the oldest, being first published by Otakar Borůvka in 1926.

The present program contains simple implementations of all three approaches, in an attempt to make practical comparisons of how they behave on "realistic" data. One of the main goals of this program is to demonstrate a simple way to make machine-independent comparisons of programs written in C, by counting memory references or "mems." In other words, this program is intended to be read, not just performed.

The author believes that mem counting sheds considerable light on the problem of determining the relative efficiency of competing algorithms for practical problems. He hopes other researchers will enjoy rising to the challenge of devising algorithms that find minimum spanning trees in significantly fewer mem units than the algorithms presented here, on problems of the size considered here.

Indeed, mem counting promises to be significant for combinatorial algorithms of all kinds. The standard graphs available in the Stanford GraphBase should make it possible to carry out a large number of machine-independent experiments concerning the practical efficiency of algorithms that have previously been studied only asymptotically.

2. The graphs we will deal with are produced by the *miles* subroutine, found in the GB_MILES module. As explained there, *miles*(*n*, *north_weight*, *west_weight*, *pop_weight*, 0, *max_degree*, *seed*) produces a graph of $n \le 128$ vertices based on the driving distances between North American cities. By default we take $n = 100$, *north_weight* = *west_weight* = *pop_weight* = 0, and *max_degree* = 10; this gives billions of different sparse graphs, when different *seed* values are specified, since a different random number seed generally results in the selection of another one of the $\binom{128}{100}$ possible subgraphs.

The default parameters can be changed by specifying options on the command line, at least in a UNIX implementation, thereby obtaining a variety of special effects. For example, the value of n can be raised or lowered and/or the graph can be made more or less sparse. The user can bias the selection by ranking cities according to their population and/or position, if nonzero values are given to any of the parameters *north_weight*, *west_weight*, or *pop_weight*. Command-line options `-n`⟨number⟩, `-N`⟨number⟩, `-W`⟨number⟩, `-P`⟨number⟩, `-d`⟨number⟩, and `-s`⟨number⟩ are used to specify non-default values of the respective quantities *n*, *north_weight*, *west_weight*, *pop_weight*, *max_degree*, and *seed*.

If the user specifies a `-r` option, for example by saying '`miles_span -r10`', this program will investigate the spanning trees of a series of, say, 10 graphs having consecutive *seed* values. (This option makes sense only if *north_weight* = *west_weight* = *pop_weight* = 0, because *miles* chooses the top n cities by weight. The procedure rarely needs to use random numbers to break ties when the weights are nonzero, because cities rarely have exactly the same weight in that case.)

The special command-line option `-g`⟨filename⟩ overrides all others. It substitutes an external graph previously saved by *save_graph* for the graphs produced by *miles*.

Here is the overall layout of this C program:

```
#include "gb_graph.h"   /* the GraphBase data structures */
#include "gb_save.h"   /* restore_graph */
#include "gb_miles.h"   /* the miles routine */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 3 ⟩
  ⟨ Procedures to be declared early 67 ⟩
  ⟨ Priority queue subroutines 24 ⟩
  ⟨ Subroutines 7 ⟩
  main(argc, argv)
      int argc;   /* the number of command-line arguments */
      char *argv[];   /* an array of strings containing those arguments */
  { unsigned long n = 100;   /* the desired number of vertices */
    unsigned long n_weight = 0;   /* the north_weight parameter */
    unsigned long w_weight = 0;   /* the west_weight parameter */
    unsigned long p_weight = 0;   /* the pop_weight parameter */
    unsigned long d = 10;   /* the max_degree parameter */
    long s = 0;   /* the random number seed */
    unsigned long r = 1;   /* the number of repetitions */
    char *file_name = Λ;   /* external graph to be restored */
    ⟨ Scan the command-line options 4 ⟩;
    while (r--) {
      if (file_name) g = restore_graph(file_name);
      else g = miles(n, n_weight, w_weight, p_weight, 0L, d, s);
      if (g ≡ Λ ∨ g→n ≤ 1) {
        fprintf(stderr, "Sorry,␣can't␣create␣the␣graph!␣(error␣code␣%ld)\n", panic_code);
        return −1;   /* error code 0 means the graph is too small */
      }
      ⟨ Report the number of mems needed to compute a minimum spanning tree of g by various
          algorithms 5 ⟩;
      gb_recycle(g);
```

```
    s++;     /* increase the seed value */
  }
  return 0;     /* normal exit */
}
```

3. ⟨ Global variables 3 ⟩ ≡

```
Graph *g;     /* the graph we will work on */
```

See also sections 6, 10, 13, 19, 23, 31, 37, 57, and 68.

This code is used in section 2.

4. ⟨ Scan the command-line options 4 ⟩ ≡

```
while (--argc) {
  if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) ;
  else if (sscanf(argv[argc], "-N%lu", &n_weight) ≡ 1) ;
  else if (sscanf(argv[argc], "-W%lu", &w_weight) ≡ 1) ;
  else if (sscanf(argv[argc], "-P%lu", &p_weight) ≡ 1) ;
  else if (sscanf(argv[argc], "-d%lu", &d) ≡ 1) ;
  else if (sscanf(argv[argc], "-r%lu", &r) ≡ 1) ;
  else if (sscanf(argv[argc], "-s%ld", &s) ≡ 1) ;
  else if (strcmp(argv[argc], "-v") ≡ 0) verbose = 1;
  else if (strncmp(argv[argc], "-g", 2) ≡ 0) file_name = argv[argc] + 2;
  else {
    fprintf(stderr, "Usage: %s [-nN][-dN][-rN][-sN][-NN][-WN][-PN][-v][-gfoo]\n", argv[0]);
    return -2;
  }
}
if (file_name) r = 1;
```

This code is used in section 2.

5. We will try out four basic algorithms that have received prominent attention in the literature. Graham and Hell's Algorithm 1 is represented by the *krusk* procedure, which uses Kruskal's algorithm after the edges have been sorted by length with a radix sort. Their Algorithm 2 is represented by the *jar_pr* procedure, which incorporates a priority queue structure that we implement in two ways, either as a simple binary heap or as a Fibonacci heap. And their Algorithm 3 is represented by the *cher_tar_kar* procedure, which implements a method similar to Borůvka's that was independently discovered by Cheriton and Tarjan and later simplified and refined by Karp and Tarjan.

```
#define INFINITY (unsigned long) −1    /* value returned when there's no spanning tree */

⟨ Report the number of mems needed to compute a minimum spanning tree of g by various algorithms 5 ⟩ ≡
  printf("The␣graph␣%s␣has␣%ld␣edges,\n", g→id, g→m/2);
  sp_length = krusk(g);
  if (sp_length ≡ INFINITY) printf("␣␣and␣it␣isn't␣connected.\n");
  else printf("␣␣and␣its␣minimum␣spanning␣tree␣has␣length␣%ld.\n", sp_length);
  printf("␣The␣Kruskal/radix-sort␣algorithm␣takes␣%ld␣mems;\n", mems);
  ⟨ Execute jar_pr(g) with binary heaps as the priority queue algorithm 28 ⟩;
  printf("␣the␣Jarnik/Prim/binary-heap␣algorithm␣takes␣%ld␣mems;\n", mems);
  ⟨ Allocate additional space needed by the more complex algorithms; or goto done if there isn't enough
      room 32 ⟩;
  ⟨ Execute jar_pr(g) with Fibonacci heaps as the priority queue algorithm 42 ⟩;
  printf("␣the␣Jarnik/Prim/Fibonacci-heap␣algorithm␣takes␣%ld␣mems;\n", mems);
  if (sp_length ≠ cher_tar_kar(g)) {
    if (gb_trouble_code) printf("␣...oops,␣I've␣run␣out␣of␣memory!\n");
    else printf("␣...oops,␣I've␣got␣a␣bug,␣please␣fix␣fix␣fix\n");
    return −3;
  }
  printf("␣the␣Cheriton/Tarjan/Karp␣algorithm␣takes␣%ld␣mems.\n\n", mems);
done: ;
```

This code is used in section 2.

6. ⟨ Global variables 3 ⟩ +≡

```
  unsigned long sp_length;    /* length of the minimum spanning tree */
```

7. When the *verbose* switch is nonzero, edges found by the various algorithms will call the *report* subroutine.

```
⟨ Subroutines 7 ⟩ ≡
  report(u, v, l)
      Vertex *u, *v;    /* adjacent vertices in the minimum spanning tree */
      long l;    /* the length of the edge between them */
  {
    printf("␣␣%ld␣miles␣between␣%s␣and␣%s␣[%ld␣mems]\n", l, u→name, v→name, mems);
  }
```

See also sections 14, 20, and 55.

This code is used in section 2.

8. Strategies and ground rules. Let us say that a *fragment* is any subtree of a minimum spanning tree. All three algorithms we implement make use of a basic principle first stated in full generality by R. C. Prim in 1957: "If a fragment F does not include all the vertices, and if e is a shortest edge joining F to a vertex not in F, then $F \cup e$ is a fragment." To prove Prim's principle, let T be a minimum spanning tree that contains F but not e. Adding e to T creates a circuit containing some edge $e' \neq e$, where e' runs from a vertex in F to a vertex not in F. Deleting e' from $T \cup e$ produces a spanning tree T' of total length no larger than the total length of T. Hence T' is a minimum spanning tree containing $F \cup e$, QED.

9. The graphs produced by *miles* have special properties, and it is fair game to make use of those properties if we can.

First, the length of each edge is a positive integer less than 2^{12}.

Second, the kth vertex v_k of the graph is represented in C programs by the pointer expression *g*→*vertices* + k. If weights have been assigned, these vertices will be in order by weight. For example, if *north_weight* = 1 but *west_weight* = *pop_weight* = 0, vertex v_0 will be the most northerly city and vertex v_{n-1} will be the most southerly.

Third, the edges accessible from a vertex v appear in a linked list starting at *v*→*arcs*. An edge from v to v_j will precede an edge from v to v_k in this list if and only if $j > k$.

Fourth, the vertices have coordinates *v*→*x_coord* and *v*→*y_coord* that are correlated with the length of edges between them: The Euclidean distance between the coordinates of two vertices tends to be small if and only if those vertices are connected by a relatively short edge. (This is only a tendency, not a certainty; for example, some cities around Chesapeake Bay are fairly close together as the crow flies, but not within easy driving range of each other.)

Fifth, the edge lengths satisfy the triangle inequality: Whenever three edges form a cycle, the longest is no longer than the sum of the lengths of the two others. (It can be proved that the triangle inequality is of no use in finding minimum spanning trees; we mention it here only to exhibit yet another way in which the data produced by *miles* is known to be nonrandom.)

Our implementation of Kruskal's algorithm will make use of the first property, and it also uses part of the third to avoid considering an edge more than once. We will not exploit the other properties, but a reader who wants to design algorithms that use fewer mems to find minimum spanning trees of these graphs is free to use any idea that helps.

10. Speaking of mems, here are the simple C instrumentation macros that we use to count memory references. The macros are called *o*, *oo*, *ooo*, and *oooo*; hence Jon Bentley has called this a "little oh analysis." Implementors who want to count mems are supposed to say, e.g., '*oo*,' just before an assignment statement or boolean expression that makes two references to memory. The C preprocessor will convert this to a statement that increases *mems* by 2 as that statement or expression is evaluated.

The semantics of C tell us that the evaluation of an expression like '$a \land (o, a\to len > 10)$' will increment *mems* if and only if the pointer variable a is non-null. Warning: The parentheses are very important in this example, because C's operator $\land$ (i.e., `&&`) has higher precedence than comma.

Values of significant variables, like a in the previous example, can be assumed to be in "registers," and no charge is made for arithmetic computations that involve only registers. But the total number of registers in an implementation must be finite and fixed, independent of the problem size.

C does not allow the *o* macros to appear in declarations, so we cannot take full advantage of C's initialization mechanism when we are counting mems. But it's easy to initialize variables in separate statements after the declarations are done.

#define *o* *mems*++
#define *oo* *mems* += 2
#define *ooo* *mems* += 3
#define *oooo* *mems* += 4

⟨ Global variables 3 ⟩ +≡
long *mems*; /* the number of memory references counted */

11. Examples of these mem-counting conventions appear throughout the program that follows. Some people will undoubtedly ask why the insertion of macros by hand is being recommended here, when it would be possible to develop a fancy system that counts mems automatically. The author believes that it is best to rely on programmers to introduce o and oo, etc., by themselves, for several reasons. (1) The macros can be inserted easily and quickly using a text editor. (2) An implementation need not pay for mems that could be avoided by a suitable optimizing compiler or by making the C program text slightly more complex; thus, authors can use their good judgment to keep programs more readable than if the code were overly hand-optimized. (3) The programmer should be able to see exactly where mems are being charged, as an aid to bottleneck elimination. Occurrences of o and oo make this plain without messing up the program text. (4) An implementation need not be charged for mems that merely provide diagnostic output, or mems that do redundant computations just to double-check the validity of "proven" assertions as a program is being tested.

Computer architecture is converging rapidly these days to the design of machines in which the exact running time of a program depends on complicated interactions between pipelined circuitry and the dynamic properties of cache mapping in a memory hierarchy, not to mention the effects of compilers and operating systems. But a good approximation to running time is usually obtained if we assume that the amount of computation is proportional to the activity of the memory bus between registers and main memory. This approximation is likely to get even better in the future, as RISC computers get faster and faster in comparison to memory devices. Although the mem measure is far from perfect, it appears to be significantly less distorted than any other measurement that can be obtained without considerably more work. An implementation that is designed to use few mems will almost certainly be efficient on today's sequential computers, as well as on the sequential computers we can expect to be built in the foreseeable future. And the converse statement is even more true: An algorithm that runs fast will not consume many mems.

Of course authors are expected to be reasonable and fair when they are competing for minimum-mem prizes. They must be ready to submit their programs to inspection by impartial judges. A good algorithm will not need to abuse the spirit of realistic mem-counting.

Mems can be analyzed theoretically as well as empirically. This means we can attach constants to estimates of running time, instead of always resorting to O notation.

12. Kruskal's algorithm. The first algorithm we shall implement and instrument is the simplest: It considers the edges one by one in order of nondecreasing length, selecting each edge that does not form a cycle with previously selected edges.

We know that the edge lengths are less than 2^{12}, so we can sort them into order with two passes of a 2^6-bucket radix sort. We will arrange to have them appear in the buckets as linked lists of **Arc** records; the two utility fields of an **Arc** will be called *from* and *klink*, respectively.

```
#define from a.V   /* an edge goes from vertex a→from to vertex a→tip */
#define klink b.A  /* the next longer edge after a will be a→klink */
```

```
⟨ Put all the edges into bucket[0] through bucket[63] 12 ⟩ ≡
  o, n = g→n;
  for (l = 0; l < 64; l++) oo, aucket[l] = bucket[l] = Λ;
  for (o, v = g→vertices; v < g→vertices + n; v++)
    for (o, a = v→arcs; a ∧ (o, a→tip > v); o, a = a→next) {
      o, a→from = v;
      o, l = a→len & #3f;   /* length mod 64 */
      oo, a→klink = aucket[l];
      o, aucket[l] = a;
    }
  for (l = 63; l ≥ 0; l−−)
    for (o, a = aucket[l]; a; ) { register long ll;
      register Arc *aa = a;
      o, a = a→klink;
      o, ll = aa→len ≫ 6;   /* length divided by 64 */
      oo, aa→klink = bucket[ll];
      o, bucket[ll] = aa;
    }
```

This code is used in section 14.

13. ⟨ Global variables 3 ⟩ +≡

```
  Arc *aucket[64], *bucket[64];   /* heads of linked lists of arcs */
```

14. Kruskal's algorithm now takes the following form.

```
⟨ Subroutines 7 ⟩ +≡
  unsigned long krusk(g)
      Graph *g;
  { ⟨ Local variables for krusk 15 ⟩
    mems = 0;
    ⟨ Put all the edges into bucket[0] through bucket[63] 12 ⟩;
    if (verbose) printf("␣␣␣[%ld␣mems␣to␣sort␣the␣edges␣into␣buckets]\n", mems);
    ⟨ Put all the vertices into components by themselves 17 ⟩;
    for (l = 0; l < 64; l++)
      for (o, a = bucket[l]; a; o, a = a→klink) {
        o, u = a→from;
        o, v = a→tip;
        ⟨ If u and v are already in the same component, continue 16 ⟩;
        if (verbose) report(a→from, a→tip, a→len);
        o, tot_len += a→len;
        if (−−components ≡ 1) return tot_len;
        ⟨ Merge the components containing u and v 18 ⟩;
      }
    return INFINITY;    /* the graph wasn't connected */
  }
```

15. Lest we forget, we'd better declare all the local variables we've been using.

```
⟨ Local variables for krusk 15 ⟩ ≡
  register Arc *a;    /* current edge of interest */
  register long l;    /* current bucket of interest */
  register Vertex *u, *v, *w;    /* current vertices of interest */
  unsigned long tot_len = 0;    /* total length of edges already chosen */
  long n;    /* the number of vertices */
  long components;
```

This code is used in section 14.

16. The remaining things that *krusk* needs to do are easily recognizable as an application of "equivalence algorithms" or "union/find" data structures. We will use a simple approach whose average running time on random graphs was shown to be linear by Knuth and Schönhage in *Theoretical Computer Science* **6** (1978), 281–315.

The vertices of each component (that is, of each connected fragment defined by the edges selected so far) will be linked circularly by *clink* pointers. Each vertex also has a *comp* field that points to a unique vertex representing its component. Each component representative also has a *csize* field that tells how many vertices are in the component.

```
#define clink z.V    /* pointer to another vertex in the same component */
#define comp y.V    /* pointer to component representative */
#define csize x.I    /* size of the component (maintained only for representatives) */
```

```
⟨ If u and v are already in the same component, continue 16 ⟩ ≡
  if (oo, u→comp ≡ v→comp) continue;
```

This code is used in section 14.

17. We don't need to charge any mems for fetching *g→vertices*, because *krusk* has already referred to it.

⟨ Put all the vertices into components by themselves 17 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + n; v++) {
   oo, v→clink = v→comp = v;
   o, v→csize = 1;
}
components = n;
```

This code is used in section 14.

18. The operation of merging two components together requires us to change two *clink* pointers, one *csize* field, and the *comp* fields in each vertex of the smaller component.

Here we charge two mems for the first **if** test, since *u→csize* and *v→csize* are being fetched from memory. Then we charge only one mem when *u→csize* is being updated, since the values being added together have already been fetched. True, the compiler has to be smart to realize that it's safe to add the fetched values *u→csize* + *v→csize* even though u and v might have been swapped in the meantime; but we are assuming that the compiler is extremely clever. (Otherwise we would have to clutter up our program every time we don't trust the compiler. After all, programs that count mems are intended primarily to be read. They aren't intended for production jobs.)

⟨ Merge the components containing u and v 18 ⟩ ≡

```
u = u→comp;    /* u→comp has already been fetched from memory */
v = v→comp;    /* ditto for v→comp */
if (oo, u→csize < v→csize) {
   w = u; u = v; v = w;
}    /* now v's component is smaller than u's (or equally small) */
o, u→csize += v→csize;
o, w = v→clink;
oo, v→clink = u→clink;
o, u→clink = w;
for ( ; ; o, w = w→clink) {
   o, w→comp = u;
   if (w ≡ v) break;
}
```

This code is used in section 14.

19. Jarník and Prim's algorithm. A second approach to minimum spanning trees is also pretty simple, except for one technicality: We want to write it in a sufficiently general manner that different priority queue algorithms can be plugged in. The basic idea is to choose an arbitrary vertex v_0 and connect it to its nearest neighbor v_1, then to connect that fragment to its nearest neighbor v_2, and so on. A priority queue holds all vertices that are adjacent to but not already in the current fragment; the key value stored with each vertex is its distance to the current fragment.

We want the priority queue data structure to support the four operations *init_queue*(*d*), *enqueue*(*v*, *d*), *requeue*(*v*, *d*), and *del_min*( ), described in the GB_DIJK module. Dijkstra's algorithm for shortest paths, described there, is remarkably similar to Jarník and Prim's algorithm for minimum spanning trees; in fact, Dijkstra discovered the latter algorithm independently, at the same time as he came up with his procedure for shortest paths.

As in GB_DIJK, we define pointers to priority queue subroutines so that the queueing mechanism can be varied.

```
#define dist z.I      /* this is the key field for vertices in the priority queue */
#define backlink y.V      /* this vertex is the stated dist away */
⟨ Global variables 3 ⟩ +≡
  void (*init_queue)( );      /* create an empty priority queue */
  void (*enqueue)( );      /* insert a new element in the priority queue */
  void (*requeue)( );      /* decrease the key of an element in the queue */
  Vertex *(*del_min)( );      /* remove an element with smallest key */
```

20. The vertices in this algorithm are initially "unseen"; they become "seen" when they enter the priority queue, and finally "known" when they leave it and enter the current fragment. We will put a special constant in the *backlink* field of known vertices. A vertex will be unseen if and only if its *backlink* is Λ.

```
#define KNOWN (Vertex *) 1      /* special backlink to mark known vertices */
⟨ Subroutines 7 ⟩ +≡
  unsigned long jar_pr(g)
      Graph *g;
  { register Vertex *t;      /* vertex that is just becoming known */
    long fragment_size;      /* number of vertices in the tree so far */
    unsigned long tot_len = 0;      /* sum of edge lengths in the tree so far */
    mems = 0;
    ⟨ Make t = g→vertices the only vertex seen; also make it known 21 ⟩;
    while (fragment_size < g→n) {
      ⟨ Put all unseen vertices adjacent to t into the queue, and update the distances of the other vertices
          adjacent to t 22 ⟩;
      t = (*del_min)( );
      if (t ≡ Λ) return INFINITY;      /* the graph is disconnected */
      if (verbose) report(t→backlink, t, t→dist);
      o, tot_len += t→dist;
      o, t→backlink = KNOWN;
      fragment_size++;
    }
    return tot_len;
  }
```

21. Notice that we don't charge any mems for the subroutine call to *init_queue*, except for mems counted in the subroutine itself. What should we charge in general for subroutine linkage when we are counting mems? The parameters to subroutines generally go into registers, and registers are "free"; also, a compiler can often choose to implement a procedure in line, thereby reducing the overhead to zero. Hence, the recommended method for charging mems with respect to subroutines is: Charge nothing if the subroutine is not recursive; otherwise charge twice the number of things that need to be saved on a runtime stack. (The return address is one of the things that needs to be saved.)

```
⟨ Make t = g→vertices the only vertex seen; also make it known 21 ⟩ ≡
  for (oo, t = g→vertices + g→n − 1; t > g→vertices; t−−) o, t→backlink = Λ;
  o, t→backlink = KNOWN;
  fragment_size = 1;
  (∗init_queue)(0L);    /∗ make the priority queue empty ∗/
```

This code is used in section 20.

22. ⟨ Put all unseen vertices adjacent to t into the queue, and update the distances of the other vertices adjacent to t 22 ⟩ ≡

```
  { register Arc ∗a;    /∗ an arc leading from t ∗/
    for (o, a = t→arcs; a; o, a = a→next) {
      register Vertex ∗v;    /∗ a vertex adjacent to t ∗/
      o, v = a→tip;
      if (o, v→backlink) {    /∗ v has already been seen ∗/
        if (v→backlink > KNOWN) {
          if (oo, a→len < v→dist) {
            o, v→backlink = t;
            (∗requeue)(v, a→len);    /∗ we found a better way to get there ∗/
          }
        }
      } else {    /∗ v hasn't been seen before ∗/
        o, v→backlink = t;
        o, (∗enqueue)(v, a→len);
      }
    }
  }
```

This code is used in section 20.

23. Binary heaps. To complete the *jar_pr* routine, we need to fill in the four priority queue functions. Jarník wrote his original paper before computers were known; Prim and Dijkstra wrote theirs before efficient priority queue algorithms were known. Their original algorithms therefore took $\Theta(n^2)$ steps. Kerschenbaum and Van Slyke pointed out in 1972 that binary heaps could do better. A simplified version of binary heaps (invented by Williams in 1964) is presented here.

A binary heap is an array of n elements, and we need space for it. Fortunately the space is already there; we can use utility field u in each of the vertex records of the graph. Moreover, if *heap_elt*(i) points to vertex v, we will arrange things so that v→*heap_index* $= i$.

#define *heap_elt*(i) $(gv+i)$→$u.V$ /* the ith vertex of the heap; $gv = g$→*vertices* */
#define *heap_index* $v.I$ /* the v utility field says where a vertex is in the heap */

⟨ Global variables 3 ⟩ +≡
Vertex *$*gv$; /* g→*vertices*, the base of the heap array */
long *hsize*; /* the number of elements currently in the heap */

24. To initialize the heap, we need only initialize two "registers" to known values, so we don't have to charge any mems at all. (In a production implementation, this code would appear in-line as part of the spanning tree algorithm.)

Important Note: This routine refers to the global variable g, which is set in *main* (not in *jar_pr*). Suitable changes need to be made if these binary heap routines are used in other programs.

⟨ Priority queue subroutines 24 ⟩ ≡

```
void init_heap(d)    /* makes the heap empty */
     long d;
{
  gv = g->vertices;
  hsize = 0;
}
```

See also sections 25, 26, 27, 30, 33, 34, 38, 45, 50, 51, 52, and 54.

This code is used in section 2.

25. The key invariant property that makes heaps work is

$$heap\_elt(k/2)\to dist \le heap\_elt(k)\to dist, \qquad \text{for } 1 < k \le hsize.$$

(A reader who has not seen heap ordering before should stop at this point and study the beautiful consequences of this innocuously simple set of inequalities.) The enqueueing operation turns out to be quite simple:

```
⟨ Priority queue subroutines 24 ⟩ +≡
  void enq_heap(v, d)
      Vertex *v;   /* vertex that is entering the queue */
      long d;   /* its key (aka dist) */
  { register unsigned long k;   /* position of a "hole" in the heap */
    register unsigned long j;   /* the parent of that position */
    register Vertex *u;   /* heap_elt(j) */
    o, v→dist = d;
    k = ++hsize;
    j = k ≫ 1;   /* k/2 */
    while (j > 0 ∧ (oo, (u = heap_elt(j))→dist > d)) {
      o, heap_elt(k) = u;   /* the hole moves to parent position */
      o, u→heap_index = k;
      k = j;
      j = k ≫ 1;
    }
    o, heap_elt(k) = v;
    o, v→heap_index = k;
  }
```

26. And in fact, the general requeueing operation is almost identical to enqueueing. This operation is popularly called "siftup," because the vertex whose key is being reduced may displace its ancestors higher in the heap. We could have implemented enqueueing by first placing the new element at the end of the heap, then requeueing it; that would have cost at most a couple mems more.

```
⟨Priority queue subroutines 24⟩ +≡
  void req_heap(v, d)
      Vertex *v;   /* vertex whose key is being reduced */
      long d;   /* its new dist */
  { register unsigned long k;   /* position of a "hole" in the heap */
    register unsigned long j;   /* the parent of that position */
    register Vertex *u;   /* heap_elt(j) */
    o, v→dist = d;
    o, k = v→heap_index;   /* now heap_elt(k) = v */
    j = k ≫ 1;   /* k/2 */
    if (j > 0 ∧ (oo, (u = heap_elt(j))→dist > d)) {   /* change is needed */
      do {
        o, heap_elt(k) = u;   /* the hole moves to parent position */
        o, u→heap_index = k;
        k = j;
        j = k ≫ 1;   /* k/2 */
      } while (j > 0 ∧ (oo, (u = heap_elt(j))→dist > d));
      o, heap_elt(k) = v;
      o, v→heap_index = k;
    }
  }
```

27. Finally, the procedure for removing the vertex with smallest key is only a bit more difficult. The vertex to be removed is always *heap_elt*(1). After we delete it, we "sift down" *heap_elt*(*hsize*), until the basic heap inequalities hold once again.

At a crucial point in this process, we have $j\to dist < u\to dist$. We cannot then have $j = hsize + 1$, because the previous steps have made $(hsize + 1)\to dist = u\to dist = d$.

⟨ Priority queue subroutines 24 ⟩ +≡

```
Vertex *del_heap()
{ Vertex *v;   /* vertex to return */
  register Vertex *u;   /* vertex being sifted down */
  register unsigned long k;   /* hole in the heap */
  register unsigned long j;   /* child of that hole */
  register long d;   /* u→dist, the vertex of the vertex being sifted */
  if (hsize ≡ 0) return Λ;
  o, v = heap_elt(1);
  o, u = heap_elt(hsize--);
  o, d = u→dist;
  k = 1;
  j = 2;
  while (j ≤ hsize) {
    if (oooo, heap_elt(j)→dist > heap_elt(j + 1)→dist) j++;
    if (heap_elt(j)→dist ≥ d) break;
    o, heap_elt(k) = heap_elt(j);   /* NB: we cannot have j > hsize, see above */
    o, heap_elt(k)→heap_index = k;
    k = j;   /* the hole moves to child position */
    j = k ≪ 1;   /* 2k */
  }
  o, heap_elt(k) = u;
  o, u→heap_index = k;
  return v;
}
```

28. OK, here's how we plug binary heaps into Jarník/Prim.

⟨ Execute *jar_pr*(*g*) with binary heaps as the priority queue algorithm 28 ⟩ ≡

```
init_queue = init_heap;
enqueue = enq_heap;
requeue = req_heap;
del_min = del_heap;
if (sp_length ≠ jar_pr(g)) {
  printf("␣...oops,␣I've␣got␣a␣bug,␣please␣fix␣fix␣fix\n");
  return −4;
}
```

This code is used in section 5.

29. Fibonacci heaps. The running time of Jarník/Prim with binary heaps, when the algorithm is applied to a connected graph with n vertices and m edges, is $O(m \log n)$, because the total number of operations is $O(m+n) = O(m)$ and each heap operation takes at most $O(\log n)$ time.

Fibonacci heaps were invented by Fredman and Tarjan in 1984, in order to do better than this. The Jarník/Prim algorithm does $O(n)$ enqueueing operations, $O(n)$ delete-min operations, and $O(m)$ requeueing operations; so Fredman and Tarjan designed a data structure that would support requeueing in "constant amortized time." In other words, Fibonacci heaps allow us to do m requeueing operations with a total cost of $O(m)$, even though some of the individual requeueings might take longer. The resulting asymptotic running time is then $O(m + n \log n)$. (This turns out to be optimum within a constant factor, when the same technique is applied to Dijkstra's algorithm for shortest paths. But for minimum spanning trees the Fibonacci method is not always optimum; for example, if $m \approx n\sqrt{\log n}$, the algorithm of Cheriton and Tarjan has slightly better asymptotic behavior, $O(m \log\log n)$.)

Fibonacci heaps are more complex than binary heaps, so we can expect that overhead costs will make them non-competitive unless m and n are quite large. Furthermore, it is not clear that the running time with simple binary heaps will behave as $m \log n$ on realistic data, because $O(m \log n)$ is a worst-case estimate based on rather pessimistic assumptions. (For example, requeueing might rarely require many iterations of the siftup loop.) But it will be instructive to implement Fibonacci heaps as best we can, just to see how good they look in actual practice.

Let us say that the *rank* of a node in a forest is the number of children it has. A Fibonacci heap is an unordered forest of trees in which the key of each node is less than or equal to the key of each child of that node, and in which the following further condition, called property F, also holds: The ranks $\{r_1, r_2, \ldots, r_k\}$ of the children of every node of rank k, when put into nondecreasing order $r_1 \le r_2 \le \cdots \le r_k$, satisfy $r_j \ge j-2$ for all j.

As a consequence of property F, we can prove by induction that every node of rank k has at least F_{k+2} descendants (including itself). Therefore, for example, we cannot have a node of rank ≥ 30 unless the total size of the forest is at least $F_{32} = 2{,}178{,}309$. We cannot have a node of rank ≥ 46 unless the total size of the forest exceeds 2^{32}.

30. We will represent a Fibonacci heap with a rather elaborate data structure, in order to guarantee the efficiency of all the necessary operations. Each node will have four pointers: *parent*, the node's parent (or Λ if the node is a root); *child*, one of the node's children (or undefined if the node has no children); *lsib* and *rsib*, the node's left and right siblings. The children of each node, and the roots of the forest, are doubly linked by *lsib* and *rsib* in circular lists; the nodes in these lists can appear in any convenient order, and the *child* pointer can point to any child.

Besides the four pointers, there is a *rank* field, which tells how many children exist, and a *tag* field, which is either 0 or 1.

Suppose a node has children of ranks $\{r_1, r_2, \ldots, r_k\}$, where $r_1 \le r_2 \le \cdots \le r_k$. We know that $r_j \ge j-2$ for all j; we say that the node has l *critical* children if there are l cases of equality, where $r_j = j-2$. Our implementation will guarantee that any node with l critical children will have at least l tagged children of the corresponding ranks. For example, suppose a node has seven children, of respective ranks $\{1,1,1,2,4,4,6\}$. Then it has three critical children, because $r_3 = 1$, $r_4 = 2$, and $r_6 = 4$. In our implementation, at least one of the children of rank 1 will have $tag = 1$, and so will the child of rank 2; so will one of the children of rank 4.

There is an external pointer called *F_heap*, which indicates a node whose key is smallest. (If the heap is empty, *F_heap* is Λ.)

```
⟨ Priority queue subroutines 24 ⟩ +≡
  void init_F_heap(d)
      long d;
  { F_heap = Λ; }
```

31. ⟨ Global variables 3 ⟩ +≡

```
  Vertex *F_heap;    /* pointer to the ring of root nodes */
```

32. We can save a bit of space and time by combining the *rank* and *tag* fields into a single *rank_tag* field, which contains $rank * 2 + tag$.

Vertices in GraphBase graphs have six utility fields. That's just enough for *parent*, *child*, *lsib*, *rsib*, *rank_tag*, and the key field *dist*. But unfortunately we also need the *backlink* field, so we are over the limit. That's not really so bad, however; we can set up another array of n records, and point to it. The extra running time needed for indirect pointing does not have to be charged to mems, because a production system involving Fibonacci heaps would simply redefine **Vertex** records to have seven utility fields instead of six. In this way we can simulate the behavior of larger records without changing the basic GraphBase conventions.

We will want an **Arc** record for each vertex in our next algorithm, so we might as well allocate storage for it now even though Fibonacci heaps need only two of the five fields.

```
#define newarc u.A   /* v->newarc points to an Arc record associated with v */
#define parent newarc->tip
#define child newarc->a.V
#define lsib v.V
#define rsib w.V
#define rank_tag x.I
```

⟨ Allocate additional space needed by the more complex algorithms; or **goto** *done* if there isn't enough room 32 ⟩ ≡

```
{ register Arc *aa;
  register Vertex *uu;
  aa = gb_typed_alloc(g->n, Arc, g->aux_data);
  if (aa == Λ) {
    printf(" and there isn't enough space to try the other methods.\n\n");
    goto done;
  }
  for (uu = g->vertices; uu < g->vertices + g->n; uu++, aa++) uu->newarc = aa;
}
```

This code is used in section 5.

33. The *potential energy* of a Fibonacci heap, as we are representing it, is defined to be the number of trees in the forest plus twice the total number of tagged children. When we operate on a heap, we will store potential energy to be used up later; then it will be possible to do the later operations with only a small incremental cost to the running time. (Potential energy is just a way to prove that the amortized cost is small; it does not appear explicitly in our implementation. It simply explains why the number of mems we compute will always be $O(m + n \log n)$.)

Enqueueing is easy: We simply insert the new element as a new tree in the forest. This costs a constant amount of time, including the cost of one new unit of potential energy for the new tree.

We can assume that *F_heap→dist* appears in a register, so we need not charge a mem to fetch it.

```
⟨ Priority queue subroutines 24 ⟩ +≡
  void enq_F_heap(v, d)
      Vertex *v;    /* vertex that is entering the queue */
      long d;    /* its key (aka dist) */
  {
    o, v→dist = d;
    o, v→parent = Λ;
    o, v→rank_tag = 0;    /* v→child need not be set */
    if (F_heap ≡ Λ) {
      oo, F_heap = v→lsib = v→rsib = v;
    } else { register Vertex *u;
      o, u = F_heap→lsib;
      o, v→lsib = u;
      o, v→rsib = F_heap;
      oo, F_heap→lsib = u→rsib = v;
      if (F_heap→dist > d) F_heap = v;
    }
  }
```

34. Requeueing is of medium difficulty. If the key is being decreased in a root node, or if the decrease doesn't make the key less than the key of its parent, no links need to change (except possibly *F_heap* itself). Otherwise we detach the node and its descendants from its present family and put this former subtree into the forest as a new tree. (One unit of potential energy must be stored with it.)

The rank of the former parent, p, decreases by 1. If p is a root, we're done. Otherwise if p was not tagged, we tag it (and pay for two additional units of energy). Property F still holds, because an untagged node can always admit a decrease in rank. If p was tagged, however, we detach p and its remaining descendants, making it another new tree of the forest, with p no longer tagged. Removing the tag releases enough stored energy to pay for the extra work of moving p. Then we must decrease the rank of p's parent, and so on, until finally we get to a root or to an untagged node. The total net cost is at most three units of energy plus the cost of relinking the original node, so it is $O(1)$.

We needn't clear the tag fields of root nodes, because we never look at them.

```
⟨Priority queue subroutines 24⟩ +≡
  void req_F_heap(v, d)
      Vertex *v;    /* vertex whose key is being reduced */
      long d;    /* its new dist */
  { register Vertex *p, *pp;    /* parent and grandparent of v */
    register Vertex *u, *w;    /* other vertices being modified */
    register long r;    /* twice the rank plus the tag */
    o, v→dist = d;
    o, p = v→parent;
    if (p ≡ Λ) {
      if (F_heap→dist > d) F_heap = v;
    } else if (o, p→dist > d)
      while (1) {
        o, r = p→rank_tag;
        if (r ≥ 4)    /* v is not an only child */
          ⟨Remove v from its family 35⟩;
        ⟨Insert v into the forest 36⟩;
        o, pp = p→parent;
        if (pp ≡ Λ) {    /* the parent of v is a root */
          o, p→rank_tag = r − 2; break;
        }
        if ((r & 1) ≡ 0) {    /* the parent of v is untagged */
          o, p→rank_tag = r − 1; break;    /* now it's tagged */
        } else o, p→rank_tag = r − 2;    /* tagged parent will become a root */
        v = p; p = pp;
      }
  }
```

35. ⟨Remove v from its family 35⟩ ≡

```
  {
    o, u = v→lsib;
    o, w = v→rsib;
    o, u→rsib = w;
    o, w→lsib = u;
    if (o, p→child ≡ v) o, p→child = w;
  }
```

This code is used in section 34.

36. ⟨ Insert v into the forest 36 ⟩ ≡

```
o, v->parent = Λ;
o, u = F_heap->lsib;
o, v->lsib = u;
o, v->rsib = F_heap;
oo, F_heap->lsib = u->rsib = v;
if (F_heap->dist > d) F_heap = v;    /* this can happen only with the original v */
```

This code is used in section 34.

37. The *del_min* operation is even more interesting; this, in fact, is where most of the action lies. We know that *F_heap* points to the vertex v we will be deleting. That's nice, but we need to figure out the new value of *F_heap*. So we have to look at all the children of v and at all the root nodes in the forest. We have stored up enough potential energy to do that, but we can reclaim the potential only if we rebuild the Fibonacci heap so that the rebuilt version contains relatively few trees.

The solution is to make sure that the new heap has at most one root of each rank. Whenever we have two tree roots of equal rank, we can make one the child of the other, thus reducing the number of trees by 1. (The new child does not violate Property F, nor is it critical, so we can mark it untagged.) The largest rank is always $O(\log n)$, if there are n nodes altogether, and we can afford to pay $\log n$ units of time for the work that isn't reclaimed from potential energy.

An array of pointers to roots of known rank is used to help control this part of the process.

⟨ Global variables 3 ⟩ +≡

```
Vertex *new_roots[46];    /* big enough for queues of size 2^32 */
```

38. ⟨ Priority queue subroutines 24 ⟩ +≡

```
Vertex *del_F_heap()
{ Vertex *final_v = F_heap;    /* the node to return */
  register Vertex *t, *u, *v, *w;    /* registers for manipulation of links */
  register long h = −1;    /* the highest rank present in new_roots */
  register long r;    /* rank of current tree */
  if (F_heap) {
    if (o, F_heap->rank_tag < 2) o, v = F_heap->rsib;
    else {
      o, w = F_heap->child;
      o, v = w->rsib;
      oo, w->rsib = F_heap->rsib;    /* link children of deleted node into the list */
      for (w = v; w ≠ F_heap->rsib; o, w = w->rsib) o, w->parent = Λ;
    }
    while (v ≠ F_heap) {
      o, w = v->rsib;
      ⟨ Put the tree rooted at v into the new_roots forest 39 ⟩;
      v = w;
    }
    ⟨ Rebuild F_heap from new_roots 41 ⟩;
  }
  return final_v;
}
```

39. The work we do in this step is paid for by the unit of potential energy being freed as v leaves the old forest, except for the work of increasing h; we charge the latter to the $O(\log n)$ cost of building *new_roots*.

⟨Put the tree rooted at v into the *new_roots* forest 39⟩ ≡

```
o, r = v->rank_tag >> 1;
while (1) {
  if (h < r) {
    do {
      h++;
      o, new_roots[h] = (h == r ? v : Λ);
    } while (h < r);
    break;
  }
  if (o, new_roots[r] == Λ) {
    o, new_roots[r] = v;
    break;
  }
  u = new_roots[r];
  o, new_roots[r] = Λ;
  if (oo, u->dist < v->dist) {
    o, v->rank_tag = r << 1;    /* v is not critical and needn't be tagged */
    t = u; u = v; v = t;
  }
  ⟨Make u a child of v 40⟩;
  r++;
}
o, v->rank_tag = r << 1;    /* every root in new_roots is untagged */
```

This code is used in section 38.

40. When we get to this step, u and v both have rank r, and $u\to dist \ge v\to dist$; u is untagged.

⟨Make u a child of v 40⟩ ≡

```
if (r == 0) {
  o, v->child = u;
  oo, u->lsib = u->rsib = u;
} else {
  o, t = v->child;
  oo, u->rsib = t->rsib;
  o, u->lsib = t;
  oo, u->rsib->lsib = t->rsib = u;
}
u->parent = v;
```

This code is used in section 39.

41. And now we can breathe easy, because the last step is trivial.

⟨ Rebuild *F_heap* from *new_roots* 41 ⟩ ≡

```
if (h < 0) F_heap = Λ;
else { long d;      /* smallest key value seen so far */
  o, u = v = new_roots[h];      /* u and v will point to beginning and end of list, respectively */
  o, d = u→dist;
  F_heap = u;
  for (h−−; h ≥ 0; h−−)
    if (o, new_roots[h]) {
      w = new_roots[h];
      o, w→lsib = v;
      o, v→rsib = w;
      if (o, w→dist < d) {
        F_heap = w;
        d = w→dist;
      }
      v = w;
    }
  o, v→rsib = u;
  o, u→lsib = v;
}
```

This code is used in section 38.

42. ⟨ Execute *jar_pr*(*g*) with Fibonacci heaps as the priority queue algorithm 42 ⟩ ≡

```
init_queue = init_F_heap;
enqueue = enq_F_heap;
requeue = req_F_heap;
del_min = del_F_heap;
if (sp_length ≠ jar_pr(g)) {
  printf("␣...oops,␣I've␣got␣a␣bug,␣please␣fix␣fix␣fix\n");
  return −5;
}
```

This code is used in section 5.

43. Binomial queues. Jean Vuillemin's "binomial queue" structures [*CACM* **21** (1978), 309–314] provide yet another appealing way to maintain priority queues. A binomial queue is a forest of trees with keys ordered as in Fibonacci heaps, satisfying two conditions that are considerably stronger than the Fibonacci heap property: Each node of rank k has children of respective ranks $\{0, 1, \ldots, k-1\}$; and each root of the forest has a different rank. It follows that each node of rank k has exactly 2^k descendants (including itself), and that a binomial queue of n elements has exactly as many trees as the number n has 1's in binary notation.

We could plug binomial queues into the Jarník/Prim algorithm, but they don't offer advantages over the heap methods already considered because they don't support the requeueing operation as nicely. Binomial queues do, however, permit efficient merging—the operation of combining two priority queues into one—and they achieve this without as much space overhead as Fibonacci heaps. In fact, we can implement binomial queues with only two pointers per node, namely a pointer to the largest child and another to the next sibling. This means we have just enough space in the utility fields of GraphBase **Arc** records to link the arcs that extend out of a spanning tree fragment. The algorithm of Cheriton, Tarjan, and Karp, which we will consider soon, maintains priority queues of arcs, not vertices; and it requires the operation of merging, not requeueing. Therefore binomial queues are well suited to it, and we will prepare ourselves for that algorithm by implementing basic binomial queue procedures.

Incidentally, if you wonder why Vuillemin called his structure a binomial queue, it's because the trees of 2^k elements have many pleasant combinatorial properties, among which is the fact that the number of elements on level l is the binomial coefficient $\binom{k}{l}$. The backtrack tree for subsets of a k-set has the same structure. A picture of a binomial-queue tree with $k = 5$, drawn by Jill C. Knuth, appears as the frontispiece of *The Art of Computer Programming*, facing page 1 of Volume 1.

#define *qchild* *a.A* /* pointer to the arc for largest child of an arc */
#define *qsib* *b.A* /* pointer to next larger sibling, or from largest to smallest */

44. A special header node is used at the head of a binomial queue, to represent the queue itself. The *qsib* field of this node points to the smallest root node in the forest. ("Smallest" means smallest in rank, not in key value.) The header also contains a *qcount* field, which takes the place of *qchild*; the *qcount* is the total number of nodes, so its binary representation characterizes the sizes of the trees accessible from *qsib*.

For example, suppose a queue with header node h contains five elements $\{a, b, c, d, e\}$ whose keys happen to be ordered alphabetically. The first tree might be the single node c; the other tree might be rooted at a, with children e and b. Then we have

$$\begin{array}{ll} h\to qcount = 5, & h\to qsib = c; \\ c\to qsib = a; & \\ a\to qchild = b; & \\ b\to qchild = d, & b\to qsib = e; \\ e\to qsib = b. & \end{array}$$

The other fields $c\to qchild$, $a\to qsib$, $e\to qchild$, $d\to qsib$, and $d\to qchild$ are undefined. We can save time by not loading or storing the undefined fields, which make up about 3/8 of the structure.

An empty binomial queue would have $h\to qcount = 0$ and $h\to qsib$ undefined.

Like Fibonacci heaps, binomial queues store potential energy: The number of energy units present is simply the number of trees in the forest.

#define *qcount* *a.I* /* this field takes the place of *qchild* in header nodes */

45. Most of the operations we want to do with binomial queues rely on the following basic subroutine, which merges a forest of m nodes starting at q with a forest of mm nodes starting at qq, putting a pointer to the resulting forest of $m + mm$ nodes into $h\to qsib$. The amortized running time is $O(\log m)$, independent of mm.

The *len* field, not *dist*, is the key field for this queue, because our nodes in this case are arcs instead of vertices.

```
⟨Priority queue subroutines 24⟩ +≡
  qunite(m, q, mm, qq, h)
      register long m, mm;    /* number of nodes in the forests */
      register Arc *q, *qq;    /* binomial trees in the forests, linked by qsib */
      Arc *h;    /* h→qsib will get the result */
  { register Arc *p;    /* tail of the list built so far */
    register long k = 1;    /* size of trees currently being processed */
    p = h;
    while (m) {
      if ((m & k) ≡ 0) {
        if (mm & k) {    /* qq goes into the merged list */
          o, p→qsib = qq; p = qq; mm −= k;
          if (mm) o, qq = qq→qsib;
        }
      } else if ((mm & k) ≡ 0) {    /* q goes into the merged list */
        o, p→qsib = q; p = q; m −= k;
        if (m) o, q = q→qsib;
      } else ⟨Combine q and qq into a "carry" tree, and continue merging until the carry no longer
            propagates 46⟩;
      k ≪= 1;
    }
    if (mm) o, p→qsib = qq;
  }
```

46. As we have seen in Fibonacci heaps, two heap-ordered trees can be combined by simply attaching one as a new child of the other. This operation preserves binomial trees. (In fact, if we use Fibonacci heaps without ever doing a requeue operation, the forests that appear after every *del_min* are binomial queues.) The number of trees decreases by 1, so we have a unit of potential energy to pay for this computation.

⟨ Combine q and qq into a "carry" tree, and continue merging until the carry no longer propagates 46 ⟩ ≡

```
{ register Arc *c;   /* the "carry," a tree of size 2k */
  register long key;   /* c→len */
  register Arc *r, *rr;   /* remainders of the input lists */
  m −= k; if (m) o, r = q→qsib;
  mm −= k; if (mm) o, rr = qq→qsib;
  ⟨ Set c to the combination of q and qq 47 ⟩;
  k ≪= 1; q = r; qq = rr;
  while ((m | mm) & k) {
    if ((m & k) ≡ 0) ⟨ Merge qq into c and advance qq 49 ⟩
    else {
      ⟨ Merge q into c and advance q 48 ⟩;
      if (mm & k) {
        o, p→qsib = qq; p = qq; mm −= k;
        if (mm) o, qq = qq→qsib;
      }
    }
    k ≪= 1;
  }
  o, p→qsib = c; p = c;
}
```

This code is used in section 45.

47. ⟨ Set c to the combination of q and qq 47 ⟩ ≡

```
if (oo, q→len < qq→len) {
  c = q, key = q→len;
  q = qq;
} else c = qq, key = qq→len;
if (k ≡ 1) o, c→qchild = q;
else {
  o, qq = c→qchild;
  o, c→qchild = q;
  if (k ≡ 2) o, q→qsib = qq;
  else oo, q→qsib = qq→qsib;
  o, qq→qsib = q;
}
```

This code is used in section 46.

48. At this point, $k > 1$.

⟨ Merge q into c and advance q 48 ⟩ ≡

```
{
  m −= k; if (m) o, r = q→qsib;
  if (o, q→len < key) {
    rr = c; c = q; key = q→len; q = rr;
  }
  o, rr = c→qchild;
  o, c→qchild = q;
  if (k ≡ 2) o, q→qsib = rr;
  else oo, q→qsib = rr→qsib;
  o, rr→qsib = q;
  q = r;
}
```

This code is used in section 46.

49. ⟨ Merge qq into c and advance qq 49 ⟩ ≡

```
{
  mm −= k; if (mm) o, rr = qq→qsib;
  if (o, qq→len < key) {
    r = c; c = qq; key = qq→len; qq = r;
  }
  o, r = c→qchild;
  o, c→qchild = qq;
  if (k ≡ 2) o, qq→qsib = r;
  else oo, qq→qsib = r→qsib;
  o, r→qsib = qq;
  qq = rr;
}
```

This code is used in section 46.

50. OK, now the hard work is done and we can reap the benefits of the basic *qunite* routine. One easy application enqueues a new arc in $O(1)$ amortized time.

⟨ Priority queue subroutines 24 ⟩ +≡

```
qenque(h, a)
    Arc *h;    /* header of a binomial queue */
    Arc *a;    /* new element for that queue */
{ long m;
  o, m = h→qcount;
  o, h→qcount = m + 1;
  if (m ≡ 0) o, h→qsib = a;
  else o, qunite(1L, a, m, h→qsib, h);
}
```

51. Here, similarly, is a routine that merges one binomial queue into another. The amortized running time is proportional to the logarithm of the number of nodes in the smaller queue.

⟨ Priority queue subroutines 24 ⟩ +≡

```
qmerge(h, hh)
    Arc *h;   /* header of binomial queue that will receive the result */
    Arc *hh;   /* header of binomial queue that will be absorbed */
{ long m, mm;
  o, mm = hh→qcount;
  if (mm) {
    o, m = h→qcount;
    o, h→qcount = m + mm;
    if (m ≥ mm) oo, qunite(mm, hh→qsib, m, h→qsib, h);
    else if (m ≡ 0) oo, h→qsib = hh→qsib;
    else oo, qunite(m, h→qsib, mm, hh→qsib, h);
  }
}
```

52. The other important operation is, of course, deletion of a node with the smallest key. The amortized running time is proportional to the logarithm of the queue size.

⟨ Priority queue subroutines 24 ⟩ +≡

```
Arc *qdel_min(h)
    Arc *h;   /* header of binomial queue */
{ register Arc *p, *pp;   /* current node and its predecessor */
  register Arc *q, *qq;   /* current minimum node and its predecessor */
  register long key;   /* q→len, the smallest key known so far */
  long m;   /* number of nodes in the queue */
  long k;   /* number of nodes in tree q */
  register long mm;   /* number of nodes not yet considered */
  o, m = h→qcount;
  if (m ≡ 0) return Λ;
  o, h→qcount = m − 1;
  ⟨ Find and remove a tree whose root q has the smallest key 53 ⟩;
  if (k > 2) {
    if (k + k ≤ m) oo, qunite(k − 1, q→qchild→qsib, m − k, h→qsib, h);
    else oo, qunite(m − k, h→qsib, k − 1, q→qchild→qsib, h);
  } else if (k ≡ 2) o, qunite(1L, q→qchild, m − k, h→qsib, h);
  return q;
}
```

53. If the tree with smallest key is the largest in the forest, we don't have to change any links to remove it, because our binomial queue algorithms never look at the last *qsib* pointer.

We use a well-known binary number trick: $m \mathbin{\&} (m-1)$ is the same as m, except that the least significant 1 bit is deleted.

⟨ Find and remove a tree whose root q has the smallest key 53 ⟩ ≡

```
mm = m & (m − 1);
o, q = h→qsib;
k = m − mm;
if (mm) {      /* there's more than one tree */
  p = q; qq = h;
  o, key = q→len;
  do { long t = mm & (mm − 1);
    pp = p; o, p = p→qsib;
    if (o, p→len ≤ key) {
      q = p; qq = pp; k = mm − t; key = p→len;
    }
    mm = t;
  } while (mm);
  if (k + k ≤ m) oo, qq→qsib = q→qsib;      /* remove the tree rooted at q */
}
```

This code is used in section 52.

54. To complete our implementation, here is an algorithm that traverses a binomial queue, "visiting" each node exactly once, destroying the queue as it goes. The total number of mems required is about $1.75m$.

⟨ Priority queue subroutines 24 ⟩ +≡

```
qtraverse(h, visit)
    Arc *h;      /* head of binomial queue to be unraveled */
    void (*visit)();      /* procedure to be invoked on each node */
{ register long m;      /* the number of nodes remaining */
  register Arc *p, *q, *r;      /* current position and neighboring positions */
  o, m = h→qcount;
  p = h;
  while (m) {
    o, p = p→qsib;
    (*visit)(p);
    if (m & 1) m−−;
    else {
      o, q = p→qchild;
      if (m & 2) (*visit)(q);
      else {
        o, r = q→qsib;
        if (m & (m − 1)) oo, q→qsib = p→qsib;
        (*visit)(r);
        p = r;
      }
      m −= 2;
    }
  }
}
```

55. Cheriton, Tarjan, and Karp's algorithm. The final algorithm we shall consider takes yet another approach to spanning tree minimization. It operates in two distinct stages: Stage 1 creates small fragments of the minimum tree, working locally with the edges that lead out of each fragment instead of dealing with the full set of edges at once as in Kruskal's method. As soon as the number of component fragments has been reduced from n to $\lfloor\sqrt{n}\,\rfloor$, stage 2 begins. Stage 2 runs through the remaining edges and builds a $\lfloor\sqrt{n}\,\rfloor \times \lfloor\sqrt{n}\,\rfloor$ matrix, which represents the problem of finding a minimum spanning tree on the remaining $\lfloor\sqrt{n}\,\rfloor$ components. A simple $O(\sqrt{n}\,)^2 = O(n)$ algorithm then completes the job.

The philosophy underlying stage 1 is that an edge leading out of a vertex in a small component is likely to lead to a vertex in another component, rather than in the same one. Thus each delete-min operation tends to be productive. Karp and Tarjan proved [*Journal of Algorithms* **1** (1980), 374–393] that the average running time on a random graph with n vertices and m edges will be $O(m)$.

The philosophy underlying stage 2 is that the problem on an initially sparse graph eventually reduces to a problem on a smaller but dense graph that is best solved by a different method.

```
⟨Subroutines 7⟩ +≡
  unsigned long cher_tar_kar(g)
      Graph *g;
  { ⟨Local variables for cher_tar_kar 56⟩
    mems = 0;
    ⟨Do stage 1 of cher_tar_kar 58⟩;
    if (verbose) printf("␣␣␣␣[Stage␣1␣has␣used␣%ld␣mems]\n", mems);
    ⟨Do stage 2 of cher_tar_kar 64⟩;
    return tot_len;
  }
```

56. We say that a fragment is *large* if it contains $\lfloor\sqrt{n+1}+\frac{1}{2}\rfloor$ or more vertices. As soon as a fragment becomes large, stage 1 stops trying to extend it. There cannot be more than $\lfloor\sqrt{n}\,\rfloor$ large fragments, because $(\lfloor\sqrt{n}\,\rfloor+1)\lfloor\sqrt{n+1}+\frac{1}{2}\rfloor > n$. The other fragments are called *small*.

Stage 1 keeps a list of all the small fragments. Initially this list contains n fragments consisting of one vertex each. The algorithm repeatedly looks at the first fragment on its list, and finds the smallest edge leading to another fragment. These two fragments are removed from the list and combined. The resulting fragment is put at the end of the list if it is still small, or put onto another list if it is large.

```
⟨Local variables for cher_tar_kar 56⟩ ≡
  register Vertex *s, *t;    /* beginning and end of the small list */
  Vertex *large_list;    /* beginning of the list of large fragments */
  long frags;    /* current number of fragments, large and small */
  unsigned long tot_len = 0;    /* total length of all edges in fragments */
  register Vertex *u, *v;    /* registers for list manipulation */
  register Arc *a;    /* and another */
  register long j, k;    /* index registers for stage 2 */
```

See also section 61.

This code is used in section 55.

57. We need to make *lo_sqrt* global so that the *note_edge* procedure below can access it.

```
⟨Global variables 3⟩ +≡
  long lo_sqrt, hi_sqrt;    /* ⌊√n⌋ and ⌊√(n+1) + 1/2⌋ */
```

58. There is a nonobvious way to compute $\lfloor\sqrt{n+1}+\frac{1}{2}\rfloor$ and $\lfloor\sqrt{n}\,\rfloor$. Since $\sqrt{n}$ is small and arithmetic is mem-free, the author couldn't resist writing the **for** loop shown here. Of course, different ground rules for counting mems would be appropriate if this sort of computing were a critical factor in the running time.

```
⟨ Do stage 1 of cher_tar_kar 58 ⟩ ≡
  o, frags = g→n;
  for (hi_sqrt = 1; hi_sqrt * (hi_sqrt + 1) ≤ frags; hi_sqrt++) ;
  if (hi_sqrt * hi_sqrt ≤ frags) lo_sqrt = hi_sqrt;
  else lo_sqrt = hi_sqrt − 1;
  large_list = Λ;
  ⟨ Create the small list 59 ⟩;
  while (frags > lo_sqrt) {
    ⟨ Combine the first fragment on the small list with its nearest neighbor 60 ⟩;
    frags−−;
  }
```

This code is used in section 55.

59. To represent fragments, we will use several utility fields already defined above. The *lsib* and *rsib* pointers are used between fragments in the small list, which is doubly linked; *s* points to the first small fragment, *s→rsib* to the next, ..., *t→lsib* to the second-from-last, and *t* to the last. The pointer fields *s→lsib* and *t→rsib* are undefined. The *large_list* is singly linked via *rsib* pointers, terminating with Λ.

The *csize* field of each fragment tells how many vertices it contains.

The *comp* field of each vertex is Λ if this vertex represents a fragment (i.e., if this vertex is in the small list or *large_list*); otherwise it points to another vertex that is closer to the fragment representative.

Finally, the *pq* pointer of each fragment points to the header node of its priority queue, which is a binomial queue containing all unlooked-at arcs that originate from vertices in the fragment. This pointer is identical to the *newarc* pointer already set up. In a production implementation, we wouldn't need *pq* as a separate field; it would be part of a vertex record. So we do not pay any mems for referring to it.

```
#define pq newarc

⟨ Create the small list 59 ⟩ ≡
  o, s = g→vertices;
  for (v = s; v < s + frags; v++) {
    if (v > s) {
      o, v→lsib = v − 1; o, (v − 1)→rsib = v;
    }
    o, v→comp = Λ;
    o, v→csize = 1;
    o, v→pq→qcount = 0;    /* the binomial queue is initially empty */
    for (o, a = v→arcs; a; o, a = a→next) qenque(v→pq, a);
  }
  t = v − 1;
```

This code is used in section 58.

60. ⟨ Combine the first fragment on the small list with its nearest neighbor 60 ⟩ ≡

```
v = s;
o, s = s→rsib;   /* remove v from small list */
do {
  a = qdel_min(v→pq);
  if (a ≡ Λ) return INFINITY;   /* the graph isn't connected */
  o, u = a→tip;
  while (o, u→comp) u = u→comp;   /* find the fragment pointed to */
} while (u ≡ v);   /* repeat until a new fragment is found */
if (verbose) ⟨ Report the new edge verbosely 63 ⟩;
o, tot_len += a→len;
o, v→comp = u;
qmerge(u→pq, v→pq);
o, old_size = u→csize;
o, new_size = old_size + v→csize;
o, u→csize = new_size;
⟨ Move u to the proper list position 62 ⟩;
```

This code is used in section 58.

61. ⟨ Local variables for *cher_tar_kar* 56 ⟩ +≡

```
long old_size, new_size;   /* size of fragment u, before and after */
```

62. Here is a fussy part of the program. We have just merged the small fragment v into another fragment u. If u was already large, there's nothing to do (except to check if the small list has just become empty). Otherwise we need to move u to the end of the small list, or we need to put it onto the large list. All these cases are special, if we want to avoid unnecessary memory references; so let's hope we get them right.

⟨ Move u to the proper list position 62 ⟩ ≡

```
if (old_size ≥ hi_sqrt) {   /* u was large */
  if (t ≡ v) s = Λ;   /* small list just became empty */
} else if (new_size < hi_sqrt) {   /* u was and still is small */
  if (u ≡ t) goto fin;   /* u is already where we want it */
  if (u ≡ s) o, s = u→rsib;   /* remove u from front */
  else {
    ooo, u→rsib→lsib = u→lsib;   /* detach u from middle */
    o, u→lsib→rsib = u→rsib;   /* do you follow the mem-counting here? */
  }
  o, t→rsib = u;   /* insert u at the end */
  o, u→lsib = t;
  t = u;
} else {   /* u has just become large */
  if (u ≡ t) {
    if (u ≡ s) goto fin;   /* well, keep it small, we're done anyway */
    o, t = u→lsib;   /* remove u from end */
  } else if (u ≡ s) o, s = u→rsib;   /* remove u from front */
  else {
    ooo, u→rsib→lsib = u→lsib;   /* detach u from middle */
    o, u→lsib→rsib = u→rsib;
  }
  o, u→rsib = large_list; large_list = u;   /* make u large */
}
fin: ;
```

This code is used in section 60.

63. We don't have room in our binomial queues to keep track of both endpoints of the arcs. But the arcs occur in pairs, and by looking at the address of a we can tell whether the matching arc is $a+1$ or $a-1$. (See the explanation in GB_GRAPH.)

⟨ Report the new edge verbosely 63 ⟩ ≡
report((*edge_trick* & (**siz_t**) a ? $a-1$: $a+1$)→*tip*, a→*tip*, a→*len*);

This code is used in sections 60 and 70.

64. Cheriton, Tarjan, and Karp's algorithm (continued). And now for the second part of the algorithm. Here we need to find room for a $\lfloor\sqrt{n}\,\rfloor \times \lfloor\sqrt{n}\,\rfloor$ matrix of edge lengths; we will use random access into the z utility fields of vertex records, since these haven't been used for anything yet by *cher_tar_kar*. We can also use the v utility fields to record the arcs that are the source of the best lengths, since this was the *lsib* field (no longer needed). The program doesn't count mems for updating that field, since it considers its goal to be simply the calculation of minimum spanning tree length; the actual edges of the minimum spanning tree are computed only for *verbose* mode. (We want to see how competitive *cher_tar_kar* is when we streamline it as much as possible.)

In stage 2, the vertices will be assigned integer index numbers between 0 and $\lfloor\sqrt{n}\,\rfloor - 1$. We'll put this into the *csize* field, which is no longer needed, and call it *findex*.

```
#define findex csize
#define matx(j, k) (gv + ((j) * lo_sqrt + (k)))→z.I   /* distance between fragments j and k */
#define matx_arc(j, k) (gv + ((j) * lo_sqrt + (k)))→v.A   /* arc corresponding to matx(j, k) */
#define INF 30000   /* upper bound on all edge lengths */

⟨Do stage 2 of cher_tar_kar 64⟩ ≡
  gv = g→vertices;   /* the global variable gv helps access auxiliary memory */
  ⟨Map all vertices to their index numbers 65⟩;
  ⟨Create the reduced matrix by running through all remaining edges 66⟩;
  ⟨Execute Prim's algorithm on the reduced matrix 69⟩;
```

This code is used in section 55.

65. The vertex-mapping algorithm is $O(n)$ because each non-null *comp* link is examined at most three times. We set the *comp* field to null as an indication that *findex* has been set.

```
⟨Map all vertices to their index numbers 65⟩ ≡
  if (s ≡ Λ) s = large_list;
  else t→rsib = large_list;
  for (k = 0, v = s; v; o, v = v→rsib, k++) o, v→findex = k;
  for (v = g→vertices; v < g→vertices + g→n; v++)
    if (o, v→comp) {
      for (t = v→comp; o, t→comp; t = t→comp) ;
      o, k = t→findex;
      for (t = v; o, u = t→comp; t = u) {
        o, t→comp = Λ;
        o, t→findex = k;
      }
    }
```

This code is used in section 64.

66. ⟨Create the reduced matrix by running through all remaining edges 66⟩ ≡

```
  for (j = 0; j < lo_sqrt; j++)
    for (k = 0; k < lo_sqrt; k++) o, matx(j, k) = INF;
  for (kk = 0; s; o, s = s→rsib, kk++) qtraverse(s→pq, note_edge);
```

This code is used in section 64.

67. The *note_edge* procedure "visits" every edge in the binomial queues traversed by *qtraverse* in the preceding code. Global variable *kk*, which would be a global register in a production version, is the index of the fragment from which this arc emanates.

⟨ Procedures to be declared early 67 ⟩ ≡
```
void note_edge(a)
     Arc *a;
{ register long k;
  o, k = a→tip→findex;
  if (k ≡ kk) return;
  if (oo, a→len < matx(kk, k)) {
    o, matx(kk, k) = a→len;
    o, matx(k, kk) = a→len;
    matx_arc(kk, k) = matx_arc(k, kk) = a;
  }
}
```
This code is used in section 2.

68. As we work on the final subproblem of size $\lfloor\sqrt{n}\rfloor \times \lfloor\sqrt{n}\rfloor$, we'll have a short vector that tells us the distance to each fragment that hasn't yet been joined up with fragment 0. The vector has -1 in positions that already have been joined up. In a production version, we could keep this in row 0 of *matx*.

⟨ Global variables 3 ⟩ +≡
```
long kk;   /* current fragment */
long distance[100];   /* distances to at most ⌊√n⌋ unhit fragments */
Arc *dist_arc[100];   /* the corresponding arcs, for verbose mode */
```

69. The last step, as suggested by Prim, repeatedly updates the distance table against each row of the matrix as it is encountered. This is the algorithm of choice to find the minimum spanning tree of a complete graph.

⟨ Execute Prim's algorithm on the reduced matrix 69 ⟩ ≡
```
{ long d;   /* shortest entry seen so far in distance vector */
  o, distance[0] = −1;
  d = INF;
  for (k = 1; k < lo_sqrt; k++) {
    o, distance[k] = matx(0, k);
    dist_arc[k] = matx_arc(0, k);
    if (distance[k] < d) d = distance[k], j = k;
  }
  while (frags > 1) ⟨ Connect fragment 0 with fragment j, since j is the column achieving the smallest
        distance, d; also compute j and d for the next round 70 ⟩;
}
```
This code is used in section 64.

70. ⟨ Connect fragment 0 with fragment j, since j is the column achieving the smallest distance, d; also compute j and d for the next round 70 ⟩ ≡

```
{
  if (d ≡ INF) return INFINITY;    /* the graph isn't connected */
  o, distance[j] = −1;    /* fragment j now will join up with fragment 0 */
  tot_len += d;
  if (verbose) {
    a = dist_arc[j];
    ⟨ Report the new edge verbosely 63 ⟩;
  }
  frags−−;
  d = INF;
  for (k = 1; k < lo_sqrt; k++)
    if (o, distance[k] ≥ 0) {
      if (o, matx(j, k) < distance[k]) {
        o, distance[k] = matx(j, k);
        dist_arc[k] = matx_arc(j, k);
      }
      if (distance[k] < d) d = distance[k], kk = k;
    }
  j = kk;
}
```

This code is used in section 69.

71. Conclusions. The winning algorithm, of the four methods considered here, on problems of the size considered here, with respect to mem counting, is clearly Jarník/Prim with binary heaps. Second is Kruskal with radix sorting, on sparse graphs, but the Fibonacci heap method beats it on dense graphs. Procedure *cher_tar_kar* never comes close, although every step it takes seems to be reasonably sensible and efficient, and although the implementation above gives it the benefit of every doubt when counting its mems. It apparently loses because it more or less gives up a factor of 2 by dealing with each edge twice; the other methods put very little effort into discarding an arc whose mate has already been processed.

But it is important to realize that mem counting is not the whole story. Further tests were made on a Sun SPARCstation 2, in order to measure the true running times when all the complications of pipelining, caching, and compiler optimization are taken into account. These runs showed that Kruskal's algorithm was actually best, at least on the particular system tested:

optimization level	`-g`	`-O2`	`-O3`	mems
Kruskal/radix	132	111	111	8379
Jarník/Prim/binary	307	226	212	7972
Jarník/Prim/Fibonacci	432	350	333	11519
Cheriton/Tarjan/Karp	686	509	492	17090

(Times are shown in seconds per 100,000 runs with the default graph *miles*$(100,0,0,0,0,10,0)$. Optimization level `-O4` gave the same results as `-O3`. Optimization does not change the mem count.) Thus the Kruskal procedure used only about 160 nanoseconds per mem, without optimization, and about 130 with; the others used about 380 to 400 ns/mem without optimization, 270 to 300 with. The mem measure gave consistent readings for the three "sophisticated" data structures, but the "naïve" Kruskal method blended better with hardware. The complete graph *miles*$(100,0,0,0,0,99,0)$, obtained by specifying option `-d100`, gave somewhat different statistics:

optimization level	`-g`	`-O2`	`-O3`	mems
Kruskal/radix	1846	1787	1810	63795
Jarník/Prim/binary	2246	1958	1845	50594
Jarník/Prim/Fibonacci	2675	2377	2248	58544
Cheriton/Tarjan/Karp	8881	6964	6909	165749

Now the identical machine instructions took significantly longer per mem—presumably because of cache misses, although the frequency of conditional jump instructions might also be a factor. Careful analyses of these phenomena should be instructive. Future computers are expected to be more nearly limited by memory speed; therefore the running time per mem is likely to become more uniform between methods, although cache performance will probably always be a factor.

The *krusk* procedure might go even faster if it were given a streamlined union/find algorithm. Or would such "streamlining" negate some of its present efficiency?

72. Index. We close with a list that shows where the identifiers of this program are defined and used. A special index term, 'discussion of *mems*', indicates sections where there are nontrivial comments about instrumenting a C program in the manner being recommended here.

April 17, 2003 at 13:44

MILES_SPAN

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