1. Intro. I'm implementing a pretty algorithm due to Hagauer, Imrich, and Klavžar [*Theoretical Computer Science* **215** (1999), 123–136], which embeds a median graph in a hypercube.

I don't have time to implement the more complex algorithm by which they test a given graph for medianhood. In order to obtain the time bounds they claim, rather elaborate data structures are needed, and from a practical standpoint there is no gain until the graphs are pretty huge. (But from a theoretical standpoint, their asymptotic running time of only $O(n^{3/2}\log n)$ for the more difficult problem is quite startling. They probably wouldn't have discovered the $O(m\log n)$ embedding algorithm described here if they hadn't been stimulated by that much more challenging task.)

For convenience, I assume that the hypercube has dimension at most 32, so that each vertex is represented by a bitstring that fits in a single computer word.

The actual dimension of the hypercube is the number of equivalence classes of edges, where the equivalence relation is generated by saying that edge $w \mathrel{—} x$ is equivalent to edge $y \mathrel{—} z$ whenever $x \mathrel{—} y$ and $z \mathrel{—} w$. (In other words, the "opposite" edges of every 4-cycle $w \mathrel{—} x \mathrel{—} y \mathrel{—} z \mathrel{—} w$ are equivalent.)

Caution: This program may fail disastrously if the input graph isn't a median graph. No attempt is made to accommodate non-median inputs in any graceful way.

```
#define start_vertex g→vertices
#define nmax 100 /* at most this many vertices */
#define cmax 32 /* at most this many edge classes */
#include "gb_graph.h"
#include "gb_save.h"
  ⟨ Preprocessor definitions ⟩
  Graph *g; /* the GraphBase graph we're given */
  ⟨ Global variables 2 ⟩
  ⟨ Subroutines 6 ⟩
  main(int argc, char *argv[])
  {
    register int j, k;
    register Vertex *u, *v, *w;
    register Arc *a, *b, *c;
    if (argc < 2) {
      fprintf(stderr, "Usage: %s foo.gb [verbose]\n", argv[0]);
      exit(-1);
    }
    verbose = argc - 2; /* extra info will be printed if desired */
    g = restore_graph(argv[1]);
    if (¬g) {
      fprintf(stderr, "Sorry, I couldn't input the graph %s!\n", argv[1]);
      exit(-2);
    }
    if (g→n > nmax) {
      fprintf(stderr, "Sorry, the graph has more vertices (%d) than I can handle (%d)!\n",
          g→n, nmax);
      exit(-3);
    }
    ⟨ Prepare the data structures 10 ⟩;
    embed(start_vertex); /* launch the recursion */
    ⟨ Print out the embedding codes 14 ⟩;
  }
```

2. Data structures. One part of the algorithm will behoove us to find the arc from vertex u to vertex v, given u and v. For this random-access task, it's easiest to allocate an $n \times n$ array *adjmat*, with an entry in row u and column v that points to the arc in question.

```
#define the_arc(u,v) adjmat[u - g→vertices][v - g→vertices]
```

⟨ Global variables 2 ⟩ ≡

```
Arc *adjmat[nmax][nmax];    /* adjacency matrix */
```

See also sections 5 and 12.

This code is used in section 1.

3. We want the adjacency lists to be doubly linked, so that it will be easy to remove an arbitrary edge. The doubly linked adjacency lists in this program have a header node $b = v\text{→}arcs$ for every vertex, recognizable by the fact that $b\text{→}tip = \Lambda$. Every arc node a in an adjacency list will have $a\text{→}next\text{→}prev = a\text{→}prev\text{→}next = a$.

```
#define prev a.A    /* utility field a in an Arc node is the inverse of next */
```

⟨ Doubly link all adjacency lists 3 ⟩ ≡

```
for (u = g→vertices; u < g→vertices + g→n; u++) {
  b = gb_virgin_arc(), b→tip = Λ;    /* create the header */
  for (a = u→arcs, u→arcs = b, b→next = a; a; b = a, a = b→next) the_arc(u, a→tip) = a, a→prev = b;
  b→next = u→arcs, u→arcs→prev = b;    /* complete the cycle */
}
```

This code is used in section 10.

4. We also want to compute the distances $d(v)$ from *start_vertex* to each vertex v, representing $d(v)$ in memory as $v\text{→}dist$.

The graph is bipartite, so we know that $d(u) - d(v) = \pm 1$ whenever u — v. We will order the adjacency list of u so that all of its neighbors v with $d(v) < d(u)$ appear before all those with $d(v) > d(u)$. The former v are called "early neighbors" and the latter are "late neighbors."

(I originally called them "down-neighbors" and "up-neighbors". But that terminology was confusing, because an "up-neighbor" actually appears *lower* in the breadth-first search tree rooted at *start_vertex*, when we follow the typical computer-science convention of putting the root at the top.)

```
#define dist z.I    /* distance from start_vertex to v is stored in utility field z */
#define link y.V    /* utility field y holds a pointer */
```

⟨ Compute all distances from *start_vertex* 4 ⟩ ≡

```
for (u = g→vertices; u < g→vertices + g→n; u++) u→dist = −1, u→link = Λ;
start_vertex→dist = 0;
for (u = v = start_vertex; u; u = u→link) {    /* ye olde breadth-first search */
  for (b = u→arcs, a = b→next; a→tip; a = a→next)
    if (a→tip→dist < 0) a→tip→dist = u→dist + 1, v→link = a→tip, v = a→tip;
    else if (a→tip→dist < u→dist) {    /* a→tip is a early neighbor */
      c = a→prev;
      if (c ≠ b) {
        c→next = a→next, c→next→prev = c;    /* move a to the beginning */
        a→next = b→next, a→next→prev = a;
        b→next = a, a→prev = b;
        a = c;
      }
    }
}
```

This code is used in section 10.

5. We use a typical union-find algorithm to handle the equivalence classes: Each edge is part of a circular list containing all edges that are currently in its class. Each edge also points to the "leader" of the class, and the leader knows the class size. When two classes merge, the leader of the smaller class resigns her post and defers to the leader of the larger class.

Classes eventually acquire a serial number from 1 to c, where c is the final number of classes. This serial number is acquired only when we are sure that the class won't be merged with another whose serial number was already assigned.

Utility fields in **Arc** nodes, called *elink*, *leader*, *size*, and *serial*, handle these aspects of the data structure. Since each edge u — v appears in memory as two consecutive **Arc** nodes, one for $u->v$ and another for $v->u$, we store *leader* and *size* in the smaller of these nodes, *elink* and *serial* in the larger.

```
#define edge_trick (sizeof(Arc) & -sizeof(Arc))  /* improve the old edge_trick */
#define lower(a) (edge_trick & (unsigned long)(a) ? (a) - 1 : (a))
#define upper(a) (edge_trick & (unsigned long)(a) ? (a) : (a) + 1)
#define other(a) (edge_trick & (unsigned long)(a) ? (a) - 1 : (a) + 1)
#define elink(a) upper(a)->b.A  /* pointer to an equivalent edge */
#define leader(a) lower(a)->b.A  /* pointer to the class representative */
#define size(a) lower(a)->len  /* size of this equivalence class */
#define serial(a) upper(a)->len  /* serial number of this equivalence class */
```

⟨ Global variables 2 ⟩ +≡

```
int classes;  /* the number of equivalence classes with assigned numbers */
```

6. Here's a routine that might help when debugging.

⟨ Subroutines 6 ⟩ ≡

```
void print_edge(Arc *e)
{
  register Arc *f;
  if (¬e->tip) printf("(header)");
  else {
    f = leader(e);
    printf("Edge %s--%s", e->tip->name, other(e)->tip->name);
    if (serial(f)) printf(", class %d", serial(f));
    else printf(" (0x%x)", (unsigned long) f);
    printf(", size %d\n", size(f));
  }
}
```

See also sections 7, 8, and 11.

This code is used in section 1.

7. The serial number is stored only with the leader.

⟨ Subroutines 6 ⟩ +≡

```
void serialize(Arc *e)  /* assign a serial number */
{
  e = leader(e);
  if (serial(e) ≡ 0) {
    serial(e) = ++classes;
    if (classes > cmax) {
      fprintf(stderr, "Overflow: more than %d classes!\n", cmax);
      exit(-5);
    }
  }
}
```

8. ⟨ Subroutines 6 ⟩ +≡

```
void unionize(Arc *e, Arc *f)      /* merge two classes */
{
  register Arc *g;
  e = leader(e), f = leader(f);
  if (e ≠ f) {
    if (serial(e) ∧ serial(f) ∧ serial(e) ≠ serial(f)) {
      fprintf(stderr, "I␣goofed␣(merging␣two␣serialize␣classes)!\n");
      exit(−69);
    }
    if (size(e) > size(f)) g = e, e = f, f = g;      /* make e the smaller class */
    leader(e) = f, size(f) += size(e);
    if (serial(e)) serial(f) = serial(e);
    for (g = elink(e); g ≠ e; g = elink(g)) leader(g) = f;
    g = elink(e), elink(e) = elink(f), elink(f) = g;
  }
}
```

9. ⟨ Make each edge its own anonymous equivalence class 9 ⟩ ≡

```
for (u = g→vertices; u < g→vertices + g→n; u++)
  for (a = u→arcs→next; a→tip; a = a→next)
    if (a ≡ lower(a)) elink(a) = leader(a) = a, size(a) = 1, serial(a) = 0;
```

This code is used in section 10.

10. A few more data structures will be introduced after we get inside the main algorithm itself, but we have now discussed all of the necessary preprocessing.

⟨ Prepare the data structures 10 ⟩ ≡

```
⟨ Doubly link all adjacency lists 3 ⟩;
⟨ Compute all distances from start_vertex 4 ⟩;
⟨ Make each edge its own anonymous equivalence class 9 ⟩;
```

This code is used in section 1.

11. The main algorithm. The heart of this program is a recursive procedure called *embed*(r). Vertex r is the "root" of all vertices currently reachable from it; in fact, r lies on a shortest path from every such vertex to the original *start_vertex*, in the original graph. Edges have gradually been pruned away, cutting r off from those closer to the start.

Procedure *embed*(r) runs through all of r's neighbors s, making additional cuts so that each s will play the role of root in its own (smaller and smaller) subgraph.

```
⟨ Subroutines 6 ⟩ +≡
  void embed(Vertex *r)
  {
    register Vertex *s, *u, *v, *w, *vv;
    register Arc *aa, *a, *b;
    if (verbose) printf("Beginning to embed subgraph %s:\n", r→name);
    for (aa = r→arcs→next; (s = aa→tip) ≠ Λ; aa = aa→next) {
      ⟨ Record r and s for postprocessing 13 ⟩;
      ⟨ Find and delete all edges equivalent to r — s 15 ⟩;
      embed(s);
      serialize(aa);
      if (verbose) printf("␣Edge␣%s␣--␣%s␣is␣in␣class␣%d\n", r→name, s→name, serial(leader(aa)));
    }
    if (verbose) printf("Done␣with␣%s.\n", r→name);
  }
```

12. The edges r — s reflected in successive calls of *embed* form a spanning tree of the original graph. In other words, every vertex s (except for *start_vertex*) is chosen exactly once, by its unique parent r. The final embedding code for s will be the code for r plus a bit in the position corresponding to the serial number of the edge r — s.

Serial numbers are assigned bottom-up to spanning tree edges, but our definition of embedding codes is top-down. So we record the edges in two tables *rtab* and *stab*; the actual codes will be assigned after all embedding has been completed.

```
#define code y.I  /* we no longer need the link field when code is used */
⟨ Global variables 2 ⟩ +≡
  Vertex *rtab[nmax], *stab[nmax];  /* edges of the spanning tree */
  int tabptr;  /* number of spanning edges recorded so far */
```

13. ⟨ Record r and s for postprocessing 13 ⟩ ≡

```
rtab[tabptr] = r, stab[tabptr] = s, tabptr++;
```

This code is used in section 11.

14. ⟨ Print out the embedding codes 14 ⟩ ≡

```
printf("The␣codewords␣are:\n␣%s␣=␣00000000\n", start_vertex→name);
start_vertex→code = #0;
for (k = 0; k < tabptr; k++) {
  u = rtab[k], v = stab[k], a = the_arc(u, v);
  j = 1 ≪ (serial(leader(a)) − 1);
  if (u→code & j) {
    fprintf(stderr, "I␣goofed␣(class␣%d␣used␣twice)!\n", serial(leader(a)));
    exit(−8);
  }
  v→code = u→code | j;
  printf("␣%s␣=␣%08x\n", v→name, v→code);
}
```

This code is used in section 1.

15. Now we get to real meat. Vertices of the current graph can be partitioned into *left vertices*, which are closer to r than to s, and *right vertices*, which are closer to s than to r. Each right vertex v has a *rank*, which is the shortest distance from v to a left vertex. Similarly, each left vertex u has rank $1-d$, where d is the shortest distance from u to a right vertex. Thus, u has rank zero if it is adjacent to a right vertex, otherwise its rank is negative. We will be particularly interested in vertices of rank 0, 1, or 2. Clearly r has rank 0 and s has rank 1.

A median graph has the property that its vertices of rank 1 form a convex set. In other words, all shortest paths between two such vertices lie entirely within the set. Furthermore every vertex v of rank 1 is adjacent to exactly one vertex u of rank 0, which we will represent by $v\mapsto mate$.

Whenever $v \mathrel{—} v'$ is an edge between vertices of rank 1, there's a corresponding edge $u \mathrel{—} u'$ between their mates. Therefore every path $v = v_0 \mathrel{—} v_1 \mathrel{—} \cdots \mathrel{—} v_k = s$ involving vertices of rank 1 corresponds to a path $u = u_0 \mathrel{—} u_1 \mathrel{—} \cdots \mathrel{—} u_k = r$ involving vertices of rank 0. The edges $u_j \mathrel{—} v_j$ are equivalent to $r \mathrel{—} s$, by our definition of equivalence, because of the 4-cycles $u_j \mathrel{—} u_{j\pm1} \mathrel{—} v_{j\pm1} \mathrel{—} v_j \mathrel{—} u_j$; so they are among the edges we will be removing from the graph in this part of the algorithm. The codeword for v_j will be the same as the codeword for u_j, except for a 1 in the component that corresponds to the serial number of class $r \mathrel{—} s$.

Hagauer, Imrich, and Klavžar noticed that a simple breath-first search will suffice to identify all vertices of ranks 0, 1, and 2, because of the special structure of median graphs. For example, if v has rank 1, its "early neighbors" must have rank 0 or 1, by convexity. And its "late neighbors" must have rank 1 or 2.

To implement their method, I'll form a sequential list of all vertices that have rank 1, starting at s and following the *link* fields. I'll set $v\mapsto mark = s$ whenever the rank of v is 1 or 2, and in that case I'll also assign the appropriate value to $v\mapsto rank$. (This marking technique ensures that previous *rank* values needn't be cleared.)

Breadth-first search in the following program is implemented as a queue that runs from v to vv, via *link* fields.

```
#define mate x.V    /* the rank-0 mate of a rank-1 vertex */
#define mark w.V    /* a specific vertex or a special Boolean code */
#define rank v.I    /* 1 or 2 */
⟨ Find and delete all edges equivalent to r — s 15 ⟩ ≡
  s↦mark = s, s↦rank = 1, s↦link = Λ;
  for (v = vv = s; v; v = v↦link) {
    ⟨ Find the mate of v and delete the corresponding edge 16 ⟩;
    for (a = v↦arcs↦next; a↦tip; a = a↦next) {
      u = a↦tip;
      if (u↦dist > v↦dist) ⟨ Classify u as rank 1 or rank 2 17 ⟩
      else ⟨ Note an equivalence if u has rank 1 18 ⟩;
    }
  }
```

This code is used in section 11.

16. In this step, vertex v has rank 1, so it should have exactly one early neighbor u that does not have rank 1; this vertex will be $v\rightarrow mate$.

Since early neighbors precede late neighbors in an adjacency list, we'll find u quickly. When we do, we want to delete the edge $v \mathrel{---} u$ and make it equivalent to edge $r \mathrel{---} s$.

```
⟨ Find the mate of v and delete the corresponding edge 16 ⟩ ≡
  for (a = v→arcs→next; ; a = a→next) {
    u = a→tip;
    if (¬(u→mark ≡ s ∧ u→rank ≡ 1)) break;
  }
  v→mate = u;
  a→next→prev = a→prev, a→prev→next = a→next;      /* delete a */
  b = other(a);      /* other(a) = the_arc(u, v) */
  b→next→prev = b→prev, b→prev→next = b→next;      /* delete a's inverse */
  unionize(a, aa);
```

This code is used in section 15.

17. In this step, u is a late neighbor of v, and v has rank 1. Vertex u itself has at least one early neighbor, namely v. If u has rank 1, we know that it will also have a (unique) early neighbor of rank 0.

Suppose w is an early neighbor and $w \neq v$. Then w has distance 2 from v, and the median x of $\{s, v, w\}$ will have rank 1 by convexity. Several cases arise: If w has rank 0, then $x = v$; otherwise we will have classified w properly when processing x. It follows that vertex u has rank 2 if and only if vertex w is known to have rank 2.

```
⟨ Classify u as rank 1 or rank 2 17 ⟩ ≡
  {
    if (u→mark ≡ s ∧ u→rank ≡ 1) continue;      /* rank 1 is correct */
    u→mark = s, u→rank = 2;      /* tentatively assign rank 2 */
    b = u→arcs→next, w = b→tip;
    if (w ≡ v) {
      w = b→next→tip;
      if (w ≡ Λ ∨ w→dist > u→dist) continue;      /* rank 2 is correct */
    }
    if (¬(w→mark ≡ s ∧ w→rank ≡ 2)) u→rank = 1, u→link = 0, vv→link = u, vv = u;      /* enqueue u */
  }
```

This code is used in section 15.

18. In this step, u is an early neighbor of v, and v has rank 1. If u also has rank 1, the edge $v \mathrel{---} u$ is equivalent to the edge from $v\rightarrow mate$ to $u\rightarrow mate$, so we need to record that fact.

```
⟨ Note an equivalence if u has rank 1 18 ⟩ ≡
  if (u→mark ≡ s ∧ u→rank ≡ 1) unionize(a, the_arc(v→mate, u→mate));
```

This code is used in section 15.

19. Index.

EMBED