1. Introduction. This is a hastily written implementation of treehull, using treaps to guarantee good average access time.

```
format Graph int    /* gb_graph defines the Graph type and a few others */
format Vertex int
format Arc int
format Area int
#include "gb_graph.h"
#include "gb_miles.h"
#include "gb_rand.h"
⟨Type declarations 3⟩
int n = 128;
⟨Global variables 2⟩
⟨Procedures 9⟩
main()
{
  ⟨Local variables 7⟩
  Graph *g = miles(128, 0, 0, 0, 0, 0, 0);
  mems = ccs = 0;
  ⟨Find convex hull of g 10⟩;
  printf("Total␣of␣%d␣mems␣and␣%d␣calls␣on␣ccw.\n", mems, ccs);
}
```

2. I'm instrumenting this in a simple way.

```
#define o mems++
#define oo mems += 2
#define ooo mems += 3
⟨Global variables 2⟩ ≡
  int mems;    /* memory accesses */
  int ccs;    /* calls on ccw */
```

See also section 5.

This code is used in section 1.

3. Data structures. For now, each vertex is represented by two coordinates stored in the utility fields $x.I$ and $y.I$. I'm also putting a serial number into $z.I$, so that I can check whether different algorithms generate identical hulls.

We use separate nodes for the current convex hull. These nodes have a bunch of fields: $p\to vert$ points to the vertex; $p\to succ$ and $p\to pred$ point to next and previous nodes in a circular list; $p\to left$ and $p\to right$ point to left and right children in a tree that's superimposed on the list; $p\to parent$ is present too, it points to the parent node; $p\to prio$ is the priority if we are implementing the tree as a treap.

The *head* node has the root of the tree in its *right* field, and it represents the special vertex that isn't in the tree.

⟨ Type declarations 3 ⟩ ≡

```
typedef struct node_struct {
  struct vertex_struct *vert;
  struct node_struct *succ, *pred, *left, *right, *parent;
  long prio;
} node;
```

This code is used in section 1.

4. ⟨ Initialize the array of nodes 4 ⟩ ≡

```
head = (node *) gb_alloc((g→n) * sizeof(node), working_storage);
if (head ≡ Λ) return (1);   /* fixthis */
next_node = head;
```

This code is used in section 6.

5. ⟨ Global variables 2 ⟩ +≡

```
node *head;   /* beginning of the hull data structure */
node *next_node;   /* first unused slot in that array */
Area working_storage;
int serial_no = 1;   /* used to disambiguate entries with equal coordinates */
```

6. We assume that the vertices have been given to us in a GraphBase-type graph. The algorithm begins with a trivial hull that contains only the first two vertices.

⟨ Initialize the data structures 6 ⟩ ≡

```
init_area(working_storage);
⟨ Initialize the array of nodes 4 ⟩;
o, u = g→vertices;
v = u + 1;
u→z.I = 0;
v→z.I = 1;
p = ++next_node;
ooo, head→succ = head→pred = head→right = p;
oo, p→succ = p→pred = head;
o, p→parent = head;
oo, p→left = p→right = Λ;
gb_init_rand(110);
o, p→prio = gb_next_rand();
o, head→vert = u;
o, p→vert = v;
next_node++;
if (n < 150) printf("Beginning␣with␣(%s;␣%s)\n", u→name, v→name);
```

This code is used in section 10.

7. We'll probably need a bunch of local variables to do elementary operations on data structures.

⟨ Local variables 7 ⟩ ≡

```
Vertex *u, *v, *vv, *w;
node *p, *pp, *q, *qq, *qqq, *r, *rr, *s, *ss, *tt, **par, **ppar, *prepar, *preppar;
int replaced;    /* will be nonzero if we've just replaced a hull element */
```

This code is used in section 1.

8. Here's a routine I used when debugging (in fact I should have written it sooner than I did).

⟨ Verify the integrity of the data structures 8 ⟩ ≡

```
p = head;
count = 0;
do {
  count++;
  p→prio = (p→prio & #ffff0000) + count;
  if (p→succ→pred ≠ p) printf("succ/pred␣failure␣at␣%s!\n", p→vert→name);
  if (p→left ≠ Λ ∧ p→left→parent ≠ p) printf("parent/lchild␣failure␣at␣%s!\n", p→vert→name);
  if (p→right ≠ Λ ∧ p→right→parent ≠ p) printf("parent/rchild␣failure␣at␣%s!\n", p→vert→name);
  p = p→succ;
} while (p ≠ head);
count = 1;
inorder(head→right);
```

This code is used in section 10.

9. ⟨ Procedures 9 ⟩ ≡

```
int count;

inorder(p)
    node *p;
{
  if (p) {
    inorder(p→left);
    if ((p→prio & #ffff) ≠ ++count) {
      printf("tree␣node␣%d␣is␣missing␣at␣%d:␣%s!\n", count, p→prio & #ffff, p→vert→name);
      count = p→prio & #ffff;
    }
    inorder(p→right);
  }
}
```

See also sections 14, 16, and 19.

This code is used in section 1.

10. Hull updating. The main loop of the algorithm updates the data structure incrementally by adding one new vertex at a time. If the new vertex lies outside the current convex hull, we put it into the cycle and possibly delete some vertices that were previously part of the hull.

⟨ Find convex hull of *g* 10 ⟩ ≡
```
⟨Initialize the data structures 6⟩;
for (oo, vv = g→vertices + 2; vv < g→vertices + g→n; vv++) {
  vv→z.I = ++serial_no;
  o, q = head→pred;
  replaced = 0;
  o, u = head→vert;
  if (o, ccw(vv, u, q→vert)) ⟨Do Case 1 12⟩
  else ⟨Do Case 2 17⟩;
  ⟨Verify the integrity of the data structures 8⟩;
}
⟨Print the convex hull 11⟩;
```
This code is used in section 1.

11. Let me do the easy part first, since it's bedtime and I can worry about the rest tomorrow.

⟨ Print the convex hull 11 ⟩ ≡
```
p = head;
printf("The␣convex␣hull␣is:\n");
do {
  printf("␣␣%s\n", p→vert→name);
  p = p→succ;
} while (p ≠ head);
```
This code is used in section 10.

12. In Case 1 we don't need the tree structure since we've already found that the new vertex is outside the hull at the tree root position.

⟨ Do Case 1 12 ⟩ ≡

```
{ qqq = head;
  while (1) {
    o, r = qqq→succ;
    if (r ≡ q) break;    /* can't eliminate any more */
    if (oo, ccw(vv, qqq→vert, r→vert)) break;
    ⟨ Delete or replace qqq from the hull 15 ⟩;
    qqq = r;
  }
  qq = qqq;
  qqq = q;
  while (1) {
    o, r = qqq→pred;
    if (r ≡ qq) break;
    if (oo, ccw(vv, r→vert, qqq→vert)) break;
    ⟨ Delete or replace qqq from the hull 15 ⟩;
    qqq = r;
  }
  q = qqq;
  if (¬replaced) ⟨ Insert vv at the right of the tree 13 ⟩;
  if (n < 150) printf("New␣hull␣sequence␣(%s;␣%s;␣%s)\n", q→vert→name, vv→name, qq→vert→name);
}
```

This code is used in section 10.

13. At this point $q \equiv head\to pred$ is the tree's rightmost node.

⟨ Insert vv at the right of the tree 13 ⟩ ≡

```
{
  tt = next_node++;
  o, tt→vert = vv;
  o, tt→succ = head;
  o, tt→pred = q;
  o, head→pred = tt;
  o, q→succ = tt;
  oo, tt→left = tt→right = Λ;
  o, tt→prio = gb_next_rand();
  if (n < 150) printf("(Inserting␣%s␣at␣right␣of␣tree,␣prio=%d)\n", vv→name, tt→prio);
  if (o, tt→prio < q→prio) rotup(q, &(q→right), tt, tt→prio);
  else {    /* easy case, no rotation necessary */
    o, tt→parent = q;
    o, q→right = tt;
  }
}
```

This code is used in section 12.

14. The link from parent to child hasn't been set when the priorities indicate necessary rotation.

⟨ Procedures 9 ⟩ +≡

```
rotup(p, pp, q, qp)
    node *p;    /* parent of inserted node */
    node **pp;    /* link field in parent */
    node *q;    /* inserted node */
    long qp;    /* its priority */
{ node *pr, **ppr;    /* grandparent */
  node *qq;    /* child who is reparented */
  while (1) {
    o, pr = p→parent;
    if (o, pr→right ≡ p) ppr = &(pr→right);
    else ppr = &(pr→left);
    if (pp ≡ &(p→right)) {    /* we should rotate left */
      if (n < 150) printf("...(rotating␣left)\n");
      o, qq = q→left;
      o, q→left = p;
      o, p→parent = q;
      o, p→right = qq;
      if (qq ≠ Λ) o, qq→parent = p;
    }
    else {    /* we should rotate right */
      if (n < 150) printf("...(rotating␣right)\n");
      o, qq = q→right;
      o, q→right = p;
      o, p→parent = q;
      o, p→left = qq;
      if (qq ≠ Λ) o, qq→parent = p;
    }
    if (o, qp ≥ pr→prio) break;
    p = pr;
    pp = ppr;
  }
  o, q→parent = pr;
  o, *ppr = q;
}
```

15. Nodes don't need to be recycled.

⟨ Delete or replace *qqq* from the hull 15 ⟩ ≡

```
if (replaced) {
  o, pp = qqq→pred;
  o, tt = qqq→succ;
  o, pp→succ = tt;
  o, tt→pred = pp;
  o, prepar = qqq→parent;
  if (o, prepar→right ≡ qqq) par = &(prepar→right);
  else par = &(prepar→left);
  o, pp = qqq→left;
  if (o, (ss = qqq→right) ≡ Λ) {
    if (n < 150) printf("(Deleting␣%s␣from␣tree,␣case␣1)\n", qqq→vert→name);
    o, *par = pp;
    if (pp ≠ Λ) o, pp→parent = prepar;
  }
  else if (pp ≡ Λ) {
    if (n < 150) printf("(Deleting␣%s␣from␣tree,␣case␣2)\n", qqq→vert→name);
    o, *par = ss;
    o, ss→parent = prepar;
  }
  else {
    if (n < 150) printf("(Deleting␣%s␣from␣tree,␣hard␣case)\n", qqq→vert→name);
    oo, deldown(prepar, par, pp, ss, pp→prio, ss→prio);
  }
}
else {
  o, qqq→vert = vv;
  replaced = 1;
}
```

This code is used in sections 12 and 17.

16. ⟨ Procedures 9 ⟩ +≡

```
deldown(p, pp, ql, qr, qlp, qrp)
    node *p;   /* parent of deleted node */
    node **pp;   /* link field in that parent */
    node *ql, *qr;   /* children of deleted node */
    int qlp, qrp;   /* their priorities */
{ node *qq;   /* grandchild of deleted node */
  if (qlp < qrp) {
    if (n < 150) printf("...(moving left child up)\n");
    o, ql->parent = p;
    o, *pp = ql;
    o, qq = ql->right;
    if (qq ≠ Λ) o, deldown(ql, &(ql->right), qq, qr, qq->prio, qrp);   /* tail recursion */
    else {
      o, ql->right = qr;
      o, qr->parent = ql;
    }
  }
  else {
    if (n < 150) printf("...(moving right child up)\n");
    o, qr->parent = p;
    o, *pp = qr;
    o, qq = qr->left;
    if (qq ≠ Λ) o, deldown(qr, &(qr->left), ql, qq, qlp, qq->prio);   /* tail recursion */
    else {
      o, qr->left = ql;
      o, ql->parent = qr;
    }
  }
}
```

17. ⟨ Do Case 2 17 ⟩ ≡

```
{ o, qq = head→right;
  while (1) {
    if (qq ≡ q ∨ (o, ccw(u, vv, qq→vert))) {
      o, r = qq→left;
      if (r ≡ Λ) {
        preppar = qq;
        o, ppar = &(qq→left);
        break;
      }
    }
    else {
      o, r = qq→right;
      if (r ≡ Λ) {
        preppar = qq;
        o, ppar = &(qq→right);
        o, qq = qq→succ;
        break;
      }
    }
    qq = r;
  }
  if (o, (r = qq→pred) ≡ head ∨ (oo, ccw(vv, qq→vert, r→vert))) {
    if (r ≠ head) {
      while (1) {
        qqq = r;
        o, r = qqq→pred;
        if (r ≡ head) break;
        if (oo, ccw(vv, r→vert, qqq→vert)) break;
        ⟨ Delete or replace qqq from the hull 15 ⟩;
      }
      r = qqq;
    }
    qqq = qq;
    while (1) {
      if (qqq ≡ q) break;
      oo, rr = qqq→succ;
      if (oo, ccw(vv, qqq→vert, rr→vert)) break;
      ⟨ Delete or replace qqq from the hull 15 ⟩;
      qqq = rr;
    }
    if (¬replaced) ⟨ Insert vv in tree, linked by ppar 18 ⟩;
    if (n < 150)
      printf("New␣hull␣sequence␣(%s;␣%s;␣%s)\n", r→vert→name, vv→name, qqq→vert→name);
  }
}
```

This code is used in section 10.

18. ⟨Insert *vv* in tree, linked by *ppar* 18⟩ ≡

```
{
  tt = next_node++;
  o, tt→vert = vv;
  o, tt→succ = qq;
  o, tt→pred = r;
  o, qq→pred = tt;
  o, r→succ = tt;
  oo, tt→left = tt→right = Λ;
  o, tt→prio = gb_next_rand();
  if (n < 150) printf("(Inserting␣%s␣at␣bottom␣of␣tree,␣prio=%d)\n", vv→name, tt→prio);
  if (o, tt→prio < preppar→prio) rotup(preppar, ppar, tt, tt→prio);
  else {     /* easy case, no rotation needed */
    o, tt→parent = preppar;
    o, *ppar = tt;
  }
}
```

This code is used in section 17.

19. Determinants. I need code for the primitive function *ccw*. Floating-point arithmetic suffices for my purposes.

We want to evaluate the determinant

$$ccw(u,v,w)=\begin{vmatrix}u(x) & u(y) & 1\\ v(x) & v(y) & 1\\ w(x) & w(y) & 1\end{vmatrix}=\begin{vmatrix}u(x)-w(x) & u(y)-w(y)\\ v(x)-w(x) & v(y)-w(y)\end{vmatrix}.$$

```
⟨Procedures 9⟩ +≡
  int ccw(u, v, w)
      Vertex *u, *v, *w;
  { register double wx = (double) w→x.I, wy = (double) w→y.I;
    register double det = ((double) u→x.I − wx) * ((double) v→y.I − wy) − ((double)
        u→y.I − wy) * ((double) v→x.I − wx);
    Vertex *uu = u, *vv = v, *ww = w, *t;
    if (det ≡ 0) {
      det = 1;
      if (u→x.I > v→x.I ∨ (u→x.I ≡ v→x.I ∧ (u→y.I > v→y.I ∨ (u→y.I ≡ v→y.I ∧ u→z.I > v→z.I)))) {
        t = u; u = v; v = t; det = −det;
      }
      if (v→x.I > w→x.I ∨ (v→x.I ≡ w→x.I ∧ (v→y.I > w→y.I ∨ (v→y.I ≡ w→y.I ∧ v→z.I > w→z.I)))) {
        t = v; v = w; w = t; det = −det;
      }
      if (u→x.I > v→x.I ∨ (u→x.I ≡ v→x.I ∧ (u→y.I > v→y.I ∨ (u→y.I ≡ v→y.I ∧ u→z.I < v→z.I)))) {
        det = −det;
      }
    }
    if (n < 150)
      printf("cc(%s;␣%s;␣%s)␣is␣%s\n", uu→name, vv→name, ww→name, det > 0 ? "true" : "false");
    ccs++;
    return (det > 0);
  }
```