

1. Introduction. This is a hastily written implementation of the daghull algorithm.

```

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"
int n = 128;
⟨Global variables 2⟩
⟨Procedures 13⟩
main()
{
    ⟨Local variables 7⟩
    Graph *g = miles(128, 0, 0, 0, 0, 0, 0);
    mems = ccs = 0;
    ⟨Find convex hull of g 8⟩;
    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
⟨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.

A vertex v in the convex hull also has a successor $v\text{-succ}$ and predecessor $v\text{-pred}$, stored in utility fields u and v . There's also a pointer to an "instruction," $v\text{-inst}$, for which I'm using an arc record although I need only two fields; that one goes in utility field w .

An instruction has two parts. Its tip is a vertex to be used in counterclockwise testing. Its $next$ is another instruction to be followed; or, if Λ , there's no next instruction; or, if a pointer to the smallest possible instruction, we will do something special to update the current convex hull.

```
#define succ u.V
#define pred v.V
#define inst w.A
```

4. \langle Initialize the array of instructions 4 $\rangle \equiv$
`first_inst = (Arc *) gb_alloc((4 * g-n - 2) * sizeof(Arc), working_storage);`
`if (first_inst \equiv Λ) return (1); /* fixthis */`
`next_inst = first_inst;`

This code is used in section 6.

5. \langle Global variables 2 $\rangle + \equiv$
`Arc *first_inst; /* beginning of the array of instructions */`
`Arc *next_inst; /* first unused slot in that array */`
`Vertex *rover; /* one of the vertices in the convex hull */`
`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.

\langle Initialize the data structures 6 $\rangle \equiv$
`init_area(working_storage);`
 \langle Initialize the array of instructions 4 \rangle ;
`o, u = g-vertices;`
`v = u + 1;`
`u-z.I = 0;`
`v-z.I = 1;`
`oo, u-succ = u-pred = v;`
`oo, v-succ = v-pred = u;`
`oo, first_inst-tip = u; first_inst-next = first_inst;`
`oo, (++next_inst)-tip = v; next_inst-next = first_inst;`
`o, u-inst = first_inst;`
`o, v-inst = next_inst++;`
`rover = u;`
`if (n < 150) printf("Beginning with (%s,%s)\n", u-name, v-name);`

This code is used in section 8.

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

\langle Local variables 7 $\rangle \equiv$
`Vertex *u, *v, *vv, *w;`
`Arc *p, *q, *r, *s;`

This code is used in section 1.

8. 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 8 ⟩ ≡
  ⟨ Initialize the data structures 6 ⟩;
  for (oo, vv = g-vertices + 2; vv < g-vertices + g-n; vv++) {
    vv→z.I = ++serial_no;
    ⟨ Follow the instructions; continue if vv is inside the current hull 10 ⟩;
    ⟨ Update the convex hull, knowing that vv lies outside the consecutive hull vertices u and v 11 ⟩;
  }
  ⟨ Print the convex hull 9 ⟩;

```

This code is used in section 1.

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

```

⟨ Print the convex hull 9 ⟩ ≡
  u = rover;
  printf("The convex hull is:\n");
  do {
    printf("%s\n", u-name);
    u = u-succ;
  } while (u ≠ rover);

```

This code is used in section 8.

```

10. ⟨ Follow the instructions; continue if vv is inside the current hull 10 ⟩ ≡
  p = first_inst;
  do {
    if (oo, ccw(p-tip, vv, (p + 1)-tip)) p++;
    q = p;
    o, p = p-next;
  } while (p > first_inst);
  if (p ≡  $\Lambda$ ) continue;
  o, v = q-tip;
  o, u = v-pred;

```

This code is used in section 8.

11. \langle Update the convex hull, knowing that vv lies outside the consecutive hull vertices u and v 11 $\rangle \equiv$

```

o, q-next = next_inst;
while (1) {
  o, w = u-pred;
  if (w  $\equiv$  v) break;
  if (ccw(vv, w, u)) break;
  o, u-inst-next = next_inst;
  if (rover  $\equiv$  u) rover = w;
  u = w;
}
while (1) {
  o, w = v-succ;
  if (w  $\equiv$  u) break;
  if (ccw(w, vv, v)) break;
  if (rover  $\equiv$  v) rover = w;
  v = w;
  o, v-inst-next = next_inst;
}
oo, u-succ = v-pred = vv;
oo, vv-pred = u; vv-succ = v;
 $\langle$  Compile two new instructions, for  $(u, vv)$  and  $(vv, v)$  12  $\rangle$ ;

```

This code is used in section 8.

12. \langle Compile two new instructions, for (u, vv) and (vv, v) 12 $\rangle \equiv$

```

o, next_inst-tip = vv;
o, next_inst-next = first_inst;
o, vv-inst = next_inst;
next_inst++;
o, next_inst-tip = u;
o, next_inst-next = next_inst + 1;
next_inst++;
o, next_inst-tip = v;
o, next_inst-next = first_inst;
o, v-inst = next_inst;
next_inst++;
o, next_inst-tip = vv;
o, next_inst-next =  $\Lambda$ ;
next_inst++;
if (n < 150) printf("New_hull_sequence_(%s;_%s;_%s)\n", u-name, vv-name, v-name);

```

This code is used in section 11.

13. 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 13⟩ ≡

```

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);
  }

```

This code is used in section 1.

Arc: 4, 5, 7.

Area: 5.

ccs: 1, 2, 13.

ccw: 2, 10, 11, 13.

det: 13.

first_inst: 4, 5, 6, 10, 12.

g: 1.

gb_alloc: 4.

gb_graph: 1.

Graph: 1.

init_area: 6.

inst: 3, 6, 11, 12.

main: 1.

mems: 1, 2.

miles: 1.

n: 1.

name: 6, 9, 12, 13.

next: 3, 6, 10, 11, 12.

next_inst: 4, 5, 6, 11, 12.

o: 2.

oo: 2, 6, 8, 10, 11.

p: 7.

pred: 3, 6, 10, 11.

printf: 1, 6, 9, 12, 13.

q: 7.

r: 7.

rover: 5, 6, 9, 11.

s: 7.

serial_no: 5, 8.

succ: 3, 6, 9, 11.

t: 13.

tip: 3, 6, 10, 12.

u: 7, 13.

uu: 13.

v: 7, 13.

Vertex: 5, 7, 13.

vertices: 6, 8.

vv: 7, 8, 10, 11, 12, 13.

w: 7, 13.

working_storage: 4, 5, 6.

ww: 13.

wx: 13.

wy: 13.

- ⟨ Compile two new instructions, for (u, vv) and (vv, v) 12 ⟩ Used in section 11.
- ⟨ Find convex hull of g 8 ⟩ Used in section 1.
- ⟨ Follow the instructions; **continue** if vv is inside the current hull 10 ⟩ Used in section 8.
- ⟨ Global variables 2, 5 ⟩ Used in section 1.
- ⟨ Initialize the array of instructions 4 ⟩ Used in section 6.
- ⟨ Initialize the data structures 6 ⟩ Used in section 8.
- ⟨ Local variables 7 ⟩ Used in section 1.
- ⟨ Print the convex hull 9 ⟩ Used in section 8.
- ⟨ Procedures 13 ⟩ Used in section 1.
- ⟨ Update the convex hull, knowing that vv lies outside the consecutive hull vertices u and v 11 ⟩ Used in section 8.