(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. This is a simple program to find all maximal cliques of a graph, when the graph has at most 64 vertices and is given by its adjacency matrix.

The algorithm, due to Moody and Hollis, is described in Appendix 5 of the book *Mathematical Taxonomy* by Jardine and Sibson (1971). I'm writing this program hastily today because it's a nice example of bitwise manipulation in graph theory, probably suitable for Section 7.1.3 of *The Art of Computer Programming*.

In brief, we have vectors ρ_v and δ_v for each vertex v, where ρ_v is v's row in the adjacency matrix and δ_v is all 1s except for 0 in position v. (Position v in ρ_v is always 1; in other words, we assume that each vertex is adjacent to itself.) One can easily show that a set of vertices q is a clique if and only if q is the bitwise intersection of $(v{\in}q\,?\,\rho_v\!:\delta_v)$ over all vertices v. To find all cliques, we could compute all 2^n such bitwise intersections, discarding duplicates. To find all maximal cliques, we could compute all 2^n such bitwise intersections and discard any clique q that is found to be contained in another. To speed the process up, we can do those discards much more cleverly.

The input graph is specified by an ASCII file in Stanford GraphBase format. The name of that file, say '`foo.gb`', is the one-and-only command-line parameter.

I've instrumented this program to see how many memory references it makes (*mems*) and how many words of workspace it needs (*space*), exclusive of input-output.

(Note: I consider a "clique" to be any complete subgraph of a graph. Many of the older books on graph theory define it to be a *maximal* complete subgraph; but that terminology is now dying out. The earlier definition is less desirable, because for example we'd like a clique of G to be equivalent to an independent set of $\overline{G}$.)

Beware: Memory overflow is not checked. This is not intended to be a robust program; I wrote it only as a experiment.

```
#define size 100000     /* size reserved for the workspace */
#define o mems++
#define oo mems += 2
#include <stdio.h>
#include "gb_graph.h"
#include "gb_save.h"
  unsigned long long rho[64];     /* rows of the adjacency matrix */
  unsigned long long work[size];
  unsigned int mems;
  int space;
  char table[64];
  main(int argc, char *argv[])
  {
    register int i, j, k, l, m, n, p, q, r;
    register Graph *g;
    register Arc *a;
    unsigned long long u, v, w;
    ⟨Input the graph 2⟩;
    ⟨Find the maximal cliques 6⟩;
    ⟨Output the maximal cliques 4⟩;
    printf("(The␣computation␣took␣%u␣mems,␣using␣%d␣words␣of␣workspace.)\n", mems, space);
  }
```

2. ⟨Input the graph 2⟩ ≡

```
if ($argc \neq 2$) {
   fprintf(stderr, "Usage:␣%s␣inputgraph.gb\n", argv[0]);
   exit(−1);
}
$g$ = restore_graph(argv[1]);
if ($\neg g$) {
   fprintf(stderr, "I␣can't␣input␣the␣graph␣%s␣(panic␣code␣%ld)!\n", argv[1], panic_code);
   exit(−2);
}
$n = g\to n$;
if ($n > 64$) {
   fprintf(stderr, "Sorry,␣that␣graph␣has␣%d␣vertices;␣", n);
   fprintf(stderr, "I␣can't␣handle␣more␣than␣64!\n");
   exit(−3);
}
⟨Set up the rho table 3⟩;
```

This code is used in section 1.

3. ⟨Set up the *rho* table 3⟩ ≡

```
for ($j = 0$; $j < n$; $j$++) {
   $w = 1_{LL} \ll j$;
   for ($a = (g\to vertices + j)\to arcs$; $a$; $a = a\to next$) $w \mathrel{|}= 1_{LL} \ll (a\to tip - g\to vertices)$;
   $rho[j] = w$;
}
```

This code is used in section 2.

4. #define *deBruijn* #03f79d71b4ca8b09 /* the least de Bruijn cycle of length 64 */

⟨Output the maximal cliques 4⟩ ≡

```
printf("Graph␣%s␣has␣%d␣maximal␣cliques:\n", g→id, m);
⟨Set up the de Bruijn table 5⟩;
for ($k = 1$; $k \le m$; $k$++) {
   for ($w = work[k]$; $w$; $w = w \oplus v$) {
      $v = w \mathbin{\&} -w$;
      $u = v * deBruijn$;
      $j = table[u \gg 58]$;
      printf("␣%s", (g→vertices + j)→name);
   }
   printf("\n");
}
```

This code is used in section 1.

5. The ruler-function calculation in the previous section isn't part of the inner loop; so I could use a slower, brute-force scheme that simply examines each bit, one bit at a time. But I'm using the de Bruijn method anyway, in order to get experience with it.

⟨Set up the de Bruijn table 5⟩ ≡

```
for ($j = 0, v = 1$; $v$; $j$++, $v \ll= 1$) {
   $u = v * deBruijn$;
   $table[u \gg 58] = j$;
}
```

This code is used in section 4.

6. The algorithm. The *work* area holds all maximal elements of the 2^i bitwise-ands of either ρ_v or δ_v for the first i vertices. There are m of them.

A vertex that's adjacent to all other vertices is just carried along in all entries, so we needn't bother with it.

```
⟨ Find the maximal cliques 6 ⟩ ≡
  $w = 1_{LL} \ll (n-1)$;
  $oo, work[1] = work[size-1] = (w \ll 1) - 1$;      /* $n$ 1s */
  $m = 1, space = 4$;
  for ($i = 0$; $i < n$; $i{+}{+}$)
    if ($oo, rho[i] \neq work[size-1]$) {
      $v = 1_{LL} \ll i$;      /* this is the new vertex we're considering */
      ⟨ Partition the work area, putting entries that contain $v$ last 7 ⟩;
      ⟨ Convert the $v$-containing entries $u$ into $u \mathbin{\&} \rho_v$, $u \mathbin{\&} \delta_v$ 8 ⟩;
    }
```

This code is used in section 1.

7. To visualize the current situation, suppose bit v is the leftmost. Then we want to rearrange *work*[1] thru *work*[m] so that they have the form

$$\begin{matrix} 0\,\alpha_1 \\ \vdots \\ 0\,\alpha_k \\ 1\,\beta_1 \\ \vdots \\ 1\,\beta_{m-k} \end{matrix}$$

This loop is like the partitioning step of radix-exchange. There always is at least one element u with $u \mathbin{\&} v \neq 0$. For convenience (and speed) we keep $work[0] = 0$.

```
⟨ Partition the work area, putting entries that contain $v$ last 7 ⟩ ≡
  $j = 1, k = m$;
  while (1) {
    while (($o, work[j] \mathbin{\&} v) \equiv 0$) $j{+}{+}$;
    while (($o, work[k] \mathbin{\&} v) \neq 0$) $k{-}{-}$;
    if ($j > k$) break;
    $oo, u = work[j], work[j] = work[k], work[k] = u$;
    $j{+}{+}, k{-}{-}$;
  }
```

This code is used in section 6.

8. Now comes the interesting part. At this point $j = k + 1$.

Entries 1 thru k of the workspace should simply be carried over to the next round. For if $u = 0\,\alpha_i$ in the notation above, we have $u \,\&\, \rho_i \subseteq u \,\&\, \delta_i = u$; and u (which is currently maximal) won't be contained in any other entries.

Thus we focus our attention on the remaining entries in the current workspace, namely $\mathit{work}[j]$ thru $\mathit{work}[m]$, which need to be "split."

Let u be the current entry; we want to split it into $u' = u\&\rho_v$ and $u'' = u\&\delta_v$. The first one, $u' = (1\beta)\&\rho_v$, must be checked against all other first entries before it is accepted for the new round. (It can't be contained in a second entry, because it has 1 in position v.) If it is contained in a first entry already generated, we drop it. But if it contains one of those entries, we might need to drop more than one of them.

The second entry, $u'' = (1\beta) \,\&\, \delta_v = 0\beta$, is fairly easy to deal with. It can be contained in some first entry $(1\beta') \,\&\, \rho_v$ only if $\beta \subseteq \beta'$; hence $\beta = \beta'$ and $1\beta \subseteq \rho_v$. If $1\beta \not\subseteq \rho_v$, we accept 0β if and only if it isn't contained in any held-over entry 0α.

In the following steps, positions p thru $\mathit{size} - 2$ of the workspace hold the first entries u' that have been tentatively accepted. Positions $k + 1$ thru l hold the second entries u' that have definitely been accepted.

Notice that the resulting algorithm avoids linked memory, so it ought to be cache-friendly.

```
⟨ Convert the $v$-containing entries $u$ into $u \,\&\, \rho_v$, $u \,\&\, \delta_v$ 8 ⟩ ≡
  for ($l = k, p = \mathit{size} - 1$; $j \le m$; $j$++) {
    $o, u = \mathit{work}[j], q = \mathit{size} - 2$;
    $w = u \,\&\, \mathit{rho}[i]$;    /* $w = u'$; we've already fetched $\mathit{rho}[i]$ from memory */
    if ($u \ne w$) {
      for ( ; $q \ge p$; $q$−−) {
        if (($o, w \,\&\, \mathit{work}[q]) \equiv w$) goto second_entry;
        if (($w \,\&\, \mathit{work}[q]) \equiv \mathit{work}[q]$) goto absorb;
      }
      $o, \mathit{work}[--p] = w$;    /* accept $u'$, tentatively */
      if ($\mathit{space} < m + 2 + \mathit{size} - p$) $\mathit{space} = m + 2 + \mathit{size} - p$;
      goto second_entry;
    }
  absorb: ⟨ Handle the cases where $w$ may absorb previous first entries 9 ⟩;
    if ($u \equiv w$) continue;
  second_entry: $w = u \,\&\, \sim v$;    /* $w = u''$ */
    for ($q = 1$; $q \le k$; $q$++)
      if (($o, w \,\&\, \mathit{work}[q]) \equiv w$) goto done_with_u;
    $o, \mathit{work}[++l] = w$;    /* accept $u''$ */
  done_with_u: continue;
  }
  for ($m = l$; $p < \mathit{size} - 1$; $p$++) $oo, \mathit{work}[++m] = \mathit{work}[p]$;
```

This code is used in section 6.

9. Finally, we need another loop analogous to radix-exchange partitioning. But this time we will move all entries contained in w down, and all entries not contained in w up. (In fact we don't really move anything down, because those entries will be discarded.)

When we come here with $u \neq w$, the first test made is redundant. (I mean, we know that $w \mathbin{\&} work[q] = work[q]$, hence $w \mid work[q] = w$.) I could avoid that by reordering the loop, and recopying part of it to be down only when $u = w$; but I decided not to bother with such a tricky optimization.

⟨Handle the cases where w may absorb previous first entries 9⟩ ≡

```
  $o, r = p, work[p-1] = 0;$
  **while** (1) {
    **while** $(o, (w \mid work[q]) \neq w)$  $q--;$
    **while** $(o, (w \mid work[r]) \equiv w)$  $r{+}{+};$
    **if** $(q < r)$ **break**;
    $oo, work[q] = work[r], work[r] = 0, q--, r{+}{+};$
  }
  $o, work[q] = w, p = q;$
```

This code is used in section 8.

10. Index.

MAXCLIQUES