1. Intro. Johan de Ruiter presented a beautiful puzzle on 14 March 2018, based on the first 32 digits of π.

It's a special case of the following self-referential problem: Given a directed graph, find all vertex labelings such that each vertex is labeled with the number of distinct labels on its successors.

In Johan's puzzle, some of the labels are given, and we're supposed to find the others. He also presented the digraph in terms of a 10×10 array, with each cell pointing either north, south, east, or west; its successors are the cells in that direction.

I've written this program so that it could be applied to fairly arbitrary digraphs, if I decide to make it more general. The program uses bitmaps in interesting ways, not complicated.

```
#define N (0 ≪ 4)
#define S (1 ≪ 4)
#define E (2 ≪ 4)
#define W (3 ≪ 4)
#define debug 1      /* for optional verbose printing */
#define verts 100      /* vertices in the digraph */
#define maxd 9      /* maximum out-degree in the digraph; must be less than 16 */
#define bitmax (1 ≪ (maxd + 1))
#define infinity (unsigned long long)(−1)
#define o mems++
#define oo mems += 2
#define ooo mems += 3

#include <stdio.h>
#include <stdlib.h>
  long mems;
  char johan[10][10] = {
      {S + 3, W + 1, E + 4, W + 0, S + 1, W + 0, S + 5, S + 0, S + 9, S + 0},
      {E + 0, S + 0, W + 2, S + 6, S + 0, E + 0, S + 0, W + 0, E + 0, S + 5},
      {E + 0, S + 0, E + 0, E + 0, S + 0, S + 0, E + 3, S + 5, W + 8, W + 9},
      {E + 0, E + 0, S + 0, N + 0, S + 0, E + 0, W + 0, S + 0, W + 7, W + 0},
      {E + 9, E + 0, S + 3, S + 0, S + 0, S + 0, W + 0, W + 0, S + 0, W + 0},
      {E + 0, E + 0, E + 0, W + 0, S + 0, E + 0, S + 0, E + 2, S + 0, S + 3},
      {E + 0, E + 8, S + 0, N + 0, S + 0, S + 0, N + 0, W + 0, N + 0, W + 0},
      {N + 4, E + 6, S + 2, N + 6, S + 0, E + 0, S + 0, W + 0, S + 0, N + 0},
      {N + 4, E + 0, E + 0, E + 0, S + 0, W + 0, W + 3, W + 3, W + 0, N + 0},
      {E + 0, E + 8, N + 0, W + 3, N + 0, N + 2, W + 0, W + 7, N + 9, N + 5}};
  int nu[bitmax], gnu[bitmax], un[bitmax];      /* νk, 2^{νk}, and ρ[k] */
  ⟨ Global variables 6 ⟩;
  ⟨ Subroutines 8 ⟩;
  main( )
  {
    register int a, d, g, i, j, k, l, q, t, u, v, x;
    register unsigned long long p;
    ⟨ Compute the nu tables 2 ⟩;
    ⟨ Set up the graph 3 ⟩;
    ⟨ Initialize the bitmaps 4 ⟩;
    ⟨ Initialize the active list 7 ⟩;
    ⟨ Achieve stability 16 ⟩;
    ⟨ Print the solution 17 ⟩;
  }
```

2. ⟨ Compute the *nu* tables 2 ⟩ ≡

```
  for ($o, gnu[0] = 1, k = 0$;  $k < bitmax$;  $k$ += 2)
    $mems$ += 6, $nu[k] = nu[k \gg 1], nu[k+1] = nu[k] + 1, gnu[k] = gnu[k \gg 1], gnu[k+1] = gnu[k] \ll 1$;
  for ($k = 1$;  $k \le maxd$;  $k$++)  $o, un[1 \ll k] = k$;
```

This code is used in section 1.

3. The arcs from vertex v begin at $arcs[v]$, as in the Stanford GraphBase. The reverse arcs that run *to* vertex v begin at $scra[v]$.

```
#define inx(i, j)  (10 * (i) + (j))
#define newarc(ii, jj)
          mems += 8, next[++arcptr] = arcs[inx(i, j)], tip[arcptr] = inx(ii, jj), arcs[inx(i, j)] = arcptr,
              next[++arcptr] = scra[inx(ii, jj)], tip[arcptr] = inx(i, j), scra[inx(ii, jj)] = arcptr, d++
```

⟨ Set up the graph 3 ⟩ ≡

```
  for ($i = 0$;  $i < 10$;  $i$++)
    for ($j = 0$;  $j < 10$;  $j$++) {
      $v = inx(i, j)$;
      sprintf(name[v], "%02d", v);
      $known[v] = (johan[i][j]$ & #f ? $johan[i][j]$ & #f : $-1$);
      $d = 0$;
      switch ($johan[i][j] \gg 4$) {
      case $N \gg 4$:
        for ($k = 0$;  $k < i$;  $k$++)  newarc(k, j);  break;
      case $S \gg 4$:
        for ($k = 9$;  $k > i$;  $k$--)  newarc(k, j);  break;
      case $E \gg 4$:
        for ($k = 9$;  $k > j$;  $k$--)  newarc(i, k);  break;
      case $W \gg 4$:
        for ($k = 0$;  $k < j$;  $k$++)  newarc(i, k);  break;
      }
      if ($d > maxd$) {
        fprintf(stderr, "The␣outdegree␣of␣%s␣should␣be␣at␣most␣%d,␣not␣%d!\n", name[v], maxd, d);
        exit(−1);
      }
      $o, deg[v] = d$;
      if ($d \le 1$)  $known[v] = d$;      /* we can consider this label prespecified */
    }
```

This code is used in section 1.

4. The set of possible labels for vertex v is kept in $bits[v]$, a ($maxd$+1)-bit number. It's either a single bit (if v's label was prespecified) or $1 + 2 + \cdots + 2^d$ (if v has degree d and wasn't given a label).

⟨ Initialize the bitmaps 4 ⟩ ≡

```
  for ($i = 0$;  $i < 10$;  $i$++)
    for ($j = 0$;  $j < 10$;  $j$++) {
      $o, v = inx(i, j), l = johan[i][j]$ & #f;
      if ($known[v] \ge 0$)  $o, bits[v] = 1 \ll known[v]$;
      else  $oo, bits[v] = (deg[v]$ ? $(1 \ll (deg[v] + 1)) - 2 : 1)$;
    }
```

This code is used in section 1.

5. Stability. This program relies on an interesting notion of "stability." Suppose the successors of v are $w_1, \ldots, w_d$, and consider the set of all 1-bit codes $(x_1, \ldots, x_d)$ such that $x_j \subseteq \mathit{bits}[w_j]$ and $y = \mathit{gnu}[x_1 \mid \cdots \mid x_d] \subseteq \mathit{bits}[v]$.

We will use simple backtracking to compute a_j, the bitwise OR of all such x_j, as well as a_0, the bitwise OR of all such y.

If $a_j = \mathit{bits}[w_j]$ for all j and $a_0 = \mathit{bits}[v]$, we say that vertex v is *stable*. Otherwise we have reduced the number of possibilities, so we've made progress.

When every vertex is stable, we hope that every bitmap has size 1.

Otherwise the problem will have to broken into cases. I'll cross that bridge only if I need to.

(I might as well note here that stability is a fairly weak condition. For example, v will be stable if $\mathit{bits}[v] = 2^{d+1} - 1$ and $\mathit{bits}[w_1] = \cdots = \mathit{bits}[w_d]$, even though many possibilities might remain for the labels of w_1 through w_d. Yet I am optimistic, as well as curious, as I write this code.)

6. All vertices are initially "active." The idea of our main algorithm is very simple: We shall choose an active vertex v, test it for stability, and make it inactive (at least temporarily). Then, if that stability test has changed $\mathit{bits}[u]$, for any $u \in \{v, w_1, \ldots, w_d\}$, we activate u and all of its predecessors, because those vertices may now be unstable. This downhill process continues until complete stability is achieved.

The list of active vertices is doubly linked, with links in *llink* and *rlink*, and with *active* as the header.

For each vertex we maintain $\mathit{size}[v]$, the product of the cardinalities of its successor bitmaps $\mathit{bits}[w_j]$, so that we can repeatedly choose an active vertex of minimum size.

```
#define active verts
⟨ Global variables 6 ⟩ ≡
  int llink[verts + 1], rlink[verts + 1];
  int deg[verts], arcs[verts], scra[verts], bits[verts], isactive[verts], known[verts];
  unsigned long long size[verts];
  char name[verts][8];      /* each vertex name is assumed to be at most seven characters */
  int tip[2 * verts * verts], next[2 * verts * verts];
  int arcptr = 0;      /* this many entries of tip and next are in use */
```

See also sections 15 and 18.

This code is used in section 1.

7.
```
⟨ Initialize the active list 7 ⟩ ≡
  for (v = 0; v < verts; v++) {
    oo, llink[v] = (v ? v − 1 : active), rlink[v] = v + 1;
    for (o, p = 1, a = arcs[v]; a; o, a = next[a]) ooo, p *= nu[bits[tip[a]]];
    oo, isactive[v] = 1, size[v] = p;
  }
  oo, llink[active] = active − 1, rlink[active] = 0;
```

This code is used in section 1.

8. When I'm debugging, I'll probably want to print status information.

```
⟨Subroutines 8⟩ ≡
  void printvert(int v, FILE *stream)
  {
    register int b, d;
    fprintf(stream, "%s(", name[v]);
    for (b = bits[v], d = 0; (1 << d) ≤ b; d++)
      if ((1 << d) & b) fprintf(stream, "%x", d);
    fprintf(stream, ")");
  }
```

See also sections 9 and 12.

This code is used in section 1.

9. ⟨Subroutines 8⟩ +≡

```
  void printact(void)
  {
    register int v;
    for (v = rlink[active]; v ≠ active; v = rlink[v]) {
      if (llink[v] ≠ active) fprintf(stderr, "␣");
      printvert(v, stderr);
    }
    fprintf(stderr, "\n");
  }
```

10. The stability test. Here's the fun routine that motivated me to write this program.

The total number of solutions $(x_1, \ldots, x_d)$ to v's stability problem is at most *size*[v]. But of course we hope to cut this number way down. The nicest part of the following code is its calculation of *goal* bits, to rule out impossible partial solutions.

Once again I follow Algorithm 7.2.2B.

```
⟨Backtrack through $v$'s successor labels 10⟩ ≡
$b1$: $mems$ += 4, $w[0] = v$, $wb[0] = bits[v]$, $wbp[0] = 0$;
  **for** ($o, a = arcs[v], d = 0$; $a$; $o, a = next[a], d$++)
    $mems$ += 5, $w[d+1] = tip[a]$, $wb[d+1] = bits[tip[a]]$, $wbp[d+1] = 0$;
  **for** ($o, k = d, g = bits[v]$; $k$; $k$−−) $o, goal[k] = g, g = (g \mid (g \gg 1)) \,\&\, ((1 \ll k) - 1)$;
  $l = 1$;
$b2$: **if** ($l > d$) ⟨Visit a solution and **goto** $b5$ 11⟩;
  $o, x = wb[l] \,\&\, -wb[l]$;     /* the lowest bit */
$b3$: $oo, s[l] = s[l-1] \mid x$;
  **if** ($oo, gnu[s[l]] \,\&\, goal[l]$) {
    $o, move[l$++$] = x$;
    **goto** $b2$;
  }
$b4$: **for** ($x$ ≪= 1; $o, x \le wb[l]$; $x$ ≪= 1)
    **if** ($x \,\&\, wb[l]$) **goto** $b3$;
$b5$: **if** (−−$l$) {
    $o, x = move[l]$;
    **goto** $b4$;
  }
  ⟨Activate vertices whose bitmaps have changed, and their predecessors 13⟩;
```

This code is used in section 16.

11. ⟨Visit a solution and **goto** $b5$ 11⟩ ≡

```
  {
    **if** ($debug$) $printsol(d)$;
    **for** ($k = 1$; $k < l$; $k$++) $oo, wbp[k]$ |= $move[k]$;
    $ooo, wbp[0]$ |= $gnu[s[l-1]]$;
    **goto** $b5$;
  }
```

This code is used in section 10.

12. ⟨Subroutines 8⟩ +≡

```
  void printsol(int d)
  {
    register int k;
    fprintf(stderr, "␣%s->", name[w[0]]);
    for (k = 1; k ≤ d; k++) fprintf(stderr, "%d", un[move[k]]);
    fprintf(stderr, "\n");
  }
```

13. If there were no solutions, we've been given an impossible problem.

⟨ Activate vertices whose bitmaps have changed, and their predecessors 13 ⟩ ≡
```
for ($k = 0$; $k \le d$; $k$++)
  if ($oo, wbp[k] \ne wb[k]$) {
    if ($wbp[k] \equiv 0$) {
      fprintf(stderr, "Contradiction␣reached␣while␣testing␣stability␣of␣%s!\n", name[w[0]]);
      exit(−666);
    }
    $o, u = w[k]$;
    $oo, bits[u] = wbp[k]$;
    if (debug) {
      fprintf(stderr, "␣␣now␣");
      printvert(u, stderr);
      fprintf(stderr, "\n");
    }
    ⟨ Activate u 14 ⟩;
    for ($o, a = scra[u]$; $a$; $o, a = next[a]$) {
      $o, u = tip[a]$;
      $ooo, size[u] = (size[u]/nu[wb[k]]) * nu[wbp[k]]$;
      ⟨ Activate u 14 ⟩;
    }
  }
```
This code is used in section 10.

14. ⟨ Activate u 14 ⟩ ≡
```
if ($o, \neg isactive[u]$) {
  $mems$ += $5, t = llink[active], rlink[t] = llink[active] = u, llink[u] = t, rlink[u] = active$;
  $o, isactive[u] = roundno + 1$;
}
```
This code is used in section 13.

15. ⟨ Global variables 6 ⟩ +≡
```
int $w[maxd + 1]$, $wb[maxd + 1]$, $wbp[maxd + 1]$, $goal[maxd + 1]$, $move[maxd + 1]$, $s[maxd + 1]$;
```

16. The main loop. Hurray: We're ready to put everything together.

Besides our desire to choose an active item of minimum size, we want to keep cycling through the array. So we choose each item at most once per "round." The value of *isactive*[*v*] tells in which round *v* will become activated.

```
⟨ Achieve stability 16 ⟩ ≡
  while (o, rlink[active] ≠ active) {
    for (o, u = rlink[active], q = roundno + 2;  u ≠ active;  o, u = rlink[u])
      if (o, isactive[u] < q)  o, q = isactive[u], p = size[u], v = u;
      else if (isactive[u] ≡ q ∧ (o, size[u] < p))  p = size[u], v = u;
    o, isactive[v] = 0, roundno = q;
    mems += 4, u = llink[v], t = rlink[v], rlink[u] = t, llink[t] = u;
    tests++;
    if (debug) {
      fprintf(stderr, "%d:␣", tests);
      printvert(v, stderr);
      fprintf(stderr, "␣->␣");
      for (a = arcs[v];  a;  a = next[a])  printvert(tip[a], stderr);
      fprintf(stderr, "\n");
    }
    ⟨ Backtrack through v's successor labels 10 ⟩;
  }
```

This code is used in section 1.

17. ⟨ Print the solution 17 ⟩ ≡

```
  fprintf(stderr, "Stability␣achieved␣after␣%d␣tests,␣%d␣rounds,␣%ld␣mems.\n", tests, roundno,
      mems);
  for (v = 0;  v < verts;  v++) {
    if (v)  printf("␣");
    printvert(v, stdout);
  }
  printf("\n");
```

This code is used in section 1.

18. ⟨ Global variables 6 ⟩ +≡

```
  int tests,  roundno;
```

19. Index.

BACK-PI-DAY