1. Intro. This program attempts to complete a partial latin square (also called a "quasigroup with holes") to a complete latin square. It uses fancy methods based on "GAD filtering" to prune the search space, because this problem can be extremely difficult when the squares are large.

GAD filtering ("global all different" filtering) is a way to reduce the domains of variables that are required to be mutually distinct, introduced by J.-C. Régin in 1994. [See the survey by I. P. Gent, I. Miguel, and P. Nightingale in *Artificial Intelligence* **172** (2008), 1973–2000.] The basic idea is to use the well-developed theory of bipartite matching to detect and remove possibilities that can never occur in a matching; this can be done with a beautiful algorithm that finds strong components in an appropriate digraph. I'm writing this program primarily to gain experience with GAD filtering, because the latin square completion problem is essentially a "pure" example of the all-different constraint: If any search problem is improved by GAD filtering, this one surely should be. (Also, I've been fascinated with latin squares ever since my undergraduate days.)

An $n \times n$ latin square is a matrix whose entries lie in an n-element set. Those entries are all required to be different, in every row and in every column. In other words, every row of the matrix is a permutation of the permissible values, and so is every column. A partial latin square is similar, but some of its entries have been left blank. The nonblank values in each row and column are different, and the challenge is to see if we can suitably fill in the blanks. (It's like a sudoku problem, but sudoku has extra constraints.)

Input to this program on *stdin* appears on n lines of n characters each. The characters are either '`.`' (representing a blank), or one of the digits `1` to `9` or `a` to `z` or `A` to `Z`, representing an integer from 1 to n. When n is small, this problem is easily handled by a considerably simpler program called PARTIAL-LATIN-DLX, which sets up suitable input for the exact-cover-solver DLX1-PLATIN. PARTIAL-LATIN-DLX has exactly the same input conventions as this one; but because of its simplicity, it can bog down when n is large. I hope to prove that GAD filtering can often come to the rescue.

```
#define maxn 61    /* 61 is Z in our encoding */
#define qmod ((1 ≪ 14) − 1)    /* one less than a power of 2 that exceeds 3 ∗ maxn ∗ maxn */
#define encode(x) ((x) < 10 ? '0' + (x) : (x) < 36 ? 'a' + (x) − 10 : (x) < 62 ? 'A' + (x) − 36 : '*')
#define decode(c) ((c) ≥ '0' ∧ (c) ≤ '9' ? (c) − '0' : (c) ≥ 'a' ∧ (c) ≤ 'z' ? (c) − 'a' + 10 : (c) ≥
        'A' ∧ (c) ≤ 'Z' ? (c) − 'A' + 36 : −1)
#define bufsize 80
#define O "%"    /* used for percent signs in format strings */
#include <stdio.h>
#include <stdlib.h>
  char buf[bufsize];
  int board[maxn][maxn];    /* a copy of the input */
  int P[maxn][maxn], R[maxn][maxn], C[maxn][maxn];    /* auxiliary matrices */
  ⟨Type definitions 11⟩;
  ⟨Global variables 3⟩;
  ⟨Subroutines 5⟩;
  main(int argc) {    /* give dummy command-line arguments to increase verbosity */
    ⟨Local variables 4⟩;
    ⟨Input the partial latin square 6⟩;
    ⟨Initialize the data structures 15⟩;
    ⟨Solve the problem 72⟩;
    ⟨Say farewell 79⟩;
  }
```

2. This program might produce lots and lots of output if you want to see what it's thinking about. All you have to do is type one or more random arguments on the command line when you call it; this will set *argc* to 1 plus the number of such arguments. (The program never actually looks at those arguments; it merely counts them.)

Here I define macros that control various levels of verbosity.

```
#define showsols (argc > 1)   /* show every solution */
#define shownodes (argc > 2)   /* show search tree nodes */
#define showcauses (argc > 3)   /* show the reasons for backtracking */
#define showlong (argc > 4)   /* make progress reports longer */
#define showmoves (argc > 5)   /* show whenever a tentative assignment is made */
#define showprunes (argc > 6)   /* show whenever an option has been filtered out */
#define showdomains (argc > 7)   /* show domain sizes before branching */
#define showsubproblems (argc > 8)   /* show each matching problem destined for GAD */
#define showmatches (argc > 9)   /* show a match when beginning a GAD filtering step */
#define showT (argc > 10)   /* show what Tarjan's algorithm is doing */
#define showHK (argc > 11)   /* show what the Hopcroft–Karp algorithm is doing */
```

3. This program is instrumented to count "mems," the number of accesses to 64-bit words that aren't in registers on an idealized machine. It's the best way I know to make a fairly decent comparison between solvers on different machines and different operating systems in different years, although of course it's only an ballpark estimate of the true performance because of things like pipelining and caching and branch prediction.

Mems aren't counted for things like printouts or debugging or reading *stdin*, nor for the actual overhead of mem-counting.

```
#define o mems++   /* count one mem */
#define oo mems += 2   /* count two mems */
#define ooo mems += 3   /* count three mems */
#define oooo mems += 4   /* count four mems */
```

⟨ Global variables 3 ⟩ ≡

```
  unsigned long long mems;   /* how many 64-bit memory accesses have we made? */
  unsigned long long thresh = 10000000000;   /* report progress when mems ≥ thresh */
  unsigned long long delta = 10000000000;   /* increase thresh between reports */
  unsigned long long GADstart;   /* mem count when GAD filtering begins */
  unsigned long long GADone;   /* mems used in part one of GAD filtering */
  unsigned long long GADtot;   /* mems used in both parts of GAD filtering */
  unsigned long long GADtries;   /* this many GAD filtering steps */
  unsigned long long GADaborts;   /* this many of them found no matching */
  unsigned long long nodes;   /* this many nodes in the search tree so far */
  unsigned long long count;   /* this many solutions found so far */
  int originaln;
```

See also sections 13, 20, 28, 39, and 74.

This code is used in section 1.

4. Lots of local variables are used here and there, when this program assumes they will be in registers. The main ones are declared here; but others will declared below, in context, when we know their purpose. The C compiler should have great fun optimizing the assignment of these symbolic names to the actual hardware registers that will hold the data at run time.

⟨ Local variables 4 ⟩ ≡

```
  register int a, i, j, k, l, m, p, q, r, s, t, u, v, x, y, z;
```

See also sections 32, 35, 46, and 63.

This code is used in section 1.

5. The first subroutine is one that I hope never gets executed. But here it is, just in case. When it does come into play, it will again prove that "to err is human."

⟨ Subroutines 5 ⟩ ≡

```
void confusion(char *flaw, int why)
{
  fprintf(stderr, "confusion:␣%s(%d)!\n", flaw, why);
}
```

See also sections 19, 25, 26, 29, 30, 73, 76, and 77.

This code is used in section 1.

6. Let's get the boring stuff out of the way first. (This code is copied from PARTIAL-LATIN-DLX.)

⟨ Input the partial latin square 6 ⟩ ≡

```
for (z = m = n = y = 0; ; m++) {
  if (¬fgets(buf, bufsize, stdin)) break;
  if (m ≡ maxn) {
    fprintf(stderr, "Too␣many␣lines␣of␣input!\n"); exit(−1);
  }
  for (p = 0; ; p++) {
    if (buf[p] ≡ '.') {
      z++;
      continue;
    }
    x = decode(buf[p]);
    if (x < 1) break;
    if (x > y) y = x;
    if (p ≡ maxn) {
      fprintf(stderr, "Line␣way␣too␣long:␣%s", buf); exit(−2);
    }
    if (R[m][x − 1]) {
      fprintf(stderr, "Duplicate␣`%c'␣in␣row␣%d!\n", encode(x), m + 1); exit(−3);
    }
    if (C[p][x − 1]) {
      fprintf(stderr, "Duplicate␣`%c'␣in␣column␣%d!\n", encode(x), p + 1); exit(−4);
    }
    board[m][p] = P[m][p] = x, R[m][x − 1] = p + 1, C[p][x − 1] = m + 1;
  }
  if (n ≡ 0) n = p;
  if (n > p) {
    fprintf(stderr, "Line␣has␣fewer␣than␣%d␣characters:␣%s", n, buf); exit(−5);
  }
  if (n < p) {
    fprintf(stderr, "Line␣has␣more␣than␣%d␣characters:␣%s", n, buf); exit(−6);
  }
}
if (m < n) {
  fprintf(stderr, "Fewer␣than␣%d␣lines!\n", n); exit(−7);
}
if (m > n) {
  fprintf(stderr, "more␣than␣%d␣lines!\n", n); exit(−8);
}
if (y > n) {
  fprintf(stderr, "the␣entry␣`%c'␣exceeds␣%d!\n", encode(y), n); exit(−9);
}
fprintf(stderr, "OK,␣I've␣read␣a␣%dx%d␣partial␣latin␣square␣with␣%d␣missing␣entries.\n", n,
    n, z);
originaln = n;
```

This code is used in section 1.

7. A bit of theory. Latin squares enjoy lots of symmetry, some of which is obvious and some of which is less so. One obvious symmetry is between rows and columns: The transpose of a latin square is a latin square. A less obvious symmetry is between rows and values: If we replace the permutation in each row by the inverse permutation, we get another latin square. The same is true for columns, and for partial squares. Thus, for example, the six partial squares

```
314.   32..   2.13   243.   .132   .21.
2..1   1...   41..   .1.3   2.4.   1...
..1.   4.12   3...   1..4   1..4   23.1
..23   .1.3   .34.   3...   ..1.   2.4.
```

are essentially equivalent, obtainable from each other by transposition and/or inversion.

Many other symmetries are also obvious: We can permute the rows, we can permute the columns, we can permute the values. But the latter symmetries aren't especially helpful in the problem we're solving; and it turns out that transposition isn't important either. We'll see, however, that row and column inversion are extremely useful.

8. The latin square completion problem is equivalent to another problem called uniform tripartite triangulation, whose symmetries are a perfect match. A uniform tripartite graph is a three-colorable graph in which exactly half of the neighbors of each vertex are of one color while the other half have the other color. A triangulation of such a graph is a partition of its edges into triangles.

Every $n \times n$ partial latin square defines a tripartite graph on the vertices $\{r_1, \ldots, r_n\}$, $\{c_1, \ldots, c_n\}$, and $\{v_1, \ldots, v_n\}$, if we let

$$\begin{aligned} r_i \mathbin{—} c_j &\iff \text{cell } (i,j) \text{ is blank;} \\ r_i \mathbin{—} v_k &\iff \text{value } k \text{ doesn't appear in row } i; \\ c_j \mathbin{—} v_k &\iff \text{value } k \text{ doesn't appear in column } j. \end{aligned}$$

Furthermore, it's not difficult to verify that this tripartite graph is uniform. One way to see this is to begin with the complete tripartite graph, which corresponds to a completely blank partial square, and then to fill in the entries one by one. Whenever we set cell (i,j) to k, vertices r_i and c_j and v_k each lose two neighbors of opposite colors.

For example, the tripartite graph for the first partial square above has the edges

$$\begin{array}{llllllll} r_1 \mathbin{—} c_4, & r_2 \mathbin{—} c_2, & r_2 \mathbin{—} c_3, & r_3 \mathbin{—} c_1, & r_3 \mathbin{—} c_2, & r_3 \mathbin{—} c_4, & r_4 \mathbin{—} c_1, & r_4 \mathbin{—} c_2; \\ r_1 \mathbin{—} v_2, & r_2 \mathbin{—} v_3, & r_2 \mathbin{—} v_4, & r_3 \mathbin{—} v_2, & r_3 \mathbin{—} v_3, & r_3 \mathbin{—} v_4, & r_4 \mathbin{—} v_1, & r_4 \mathbin{—} v_4; \\ c_1 \mathbin{—} v_1, & c_1 \mathbin{—} v_4, & c_2 \mathbin{—} v_2, & c_2 \mathbin{—} v_3, & c_2 \mathbin{—} v_4, & c_3 \mathbin{—} v_3, & c_4 \mathbin{—} v_2, & c_4 \mathbin{—} v_4; \end{array}$$

and the other five squares have the same graph but with $\{r, c, v\}$ permuted.

9. Notice that the latin square completion problem is precisely the same as the task of triangulating its tripartite graph. And conversely, every uniform tripartite graph on the vertices $\{r_1,\ldots,r_n\}$, $\{c_1,\ldots,c_n\}$, and $\{v_1,\ldots,v_n\}$, corresponds to the problem of completing some $2n\times 2n$ latin square. (That latin square has blanks only in its top left quarter; also, every value $\{n+1,\ldots,2n\}$ occurs in every row and every column.)

[This theory is due to C. J. Colbourn, *Discrete Applied Mathematics* **8** (1984), 25–30, who used it to prove that partial latin square completion is NP complete. Notice that the *complement* of the tripartite graph that corresponds to a partial latin square problem is always triangularizable. Colbourn went up from n to $2n$, because a uniform tripartite graph whose complement isn't triangularizable does not correspond to an $n\times n$ partial latin square. Perhaps a smaller value than $2n$ would be adequate in all cases? I don't know. But n itself is too small.]

One consequence of these observations is that two partial latin squares with the same tripartite graph have exactly the same completion problem. We don't need to know any of the values of the nonblank entries, except for the identities of the missing elements; we don't even have to know n! In this program, the problem is defined solely by the zero-or-nonzero state of the arrays *board*, R, and C, not by the actual contents of those three arrays.

10. The triangularization problem, in turn, is equivalent to $3n$ simultaneous bipartite matching problems.

- The r_i problem: Match $\{j \mid r_i -\!\!- c_j\}$ with $\{k \mid r_i -\!\!- v_k\}$. ("Fill row i.")
- The c_j problem: Match $\{k \mid c_j -\!\!- v_k\}$ with $\{i \mid c_j -\!\!- r_i\}$. ("Fill column j.")
- The v_k problem: Match $\{i \mid v_k -\!\!- r_i\}$ with $\{j \mid v_k -\!\!- c_j\}$. ("Fill in the ks.")

In all three cases, the edges exist precisely when the exact cover problem defined by PARTIAL-LATIN-DLX contains the option '`p`ij `r`ik `c`jk'. So I shall refer to "options" and "edges" and "triples" interchangeably in the program that follows. Every such option is, in fact, essentially a triangle, consisting of three edges—one for the r_i matching, one for the c_j matching, and one for the v_k matching.

In summary: The problem of completing a partial latin square of size $n\times n$ is the problem of triangulating a uniform tripartite graph. The problem of triangulating a uniform tripartite graph with parts of size n is the problem of doing $3n$ simultaneous bipartite matchings. This program relies on GAD filtering, which is based on the rich theory of bipartite matching.

11. Data structures. Like all interesting problems, this one suggests interesting data structures.

At the lowest level, the input data is represented in small structs called tetrads, with four fields each: *up*, *down*, *itm*, and *aux*. Tetrads are modeled after the "nodes" in DLX; indeed, one good way to think of this program is to regard it as an exact cover solver like DLX, which has been extended by introducing GAD filtering to prune unwanted options. The *up* and *down* fields of a tetrad provide doubly linked lists of options, and the *itm* field refers to the head of such a list, just as in DLX.

The *aux* fields aren't presently used in any significant way; they're included primarily so that exactly 16 bytes are allocated, hence *up* and *down* can be fetched and stored simultaneously. But as long as we have them, we might as well put a symbolic name into *aux* for use in debugging.

⟨ Type definitions 11 ⟩ ≡

```
typedef struct {
  int up, down;  /* predecessor and successor in item list */
  int itm;  /* the item whose list contains this tetrad */
  char aux[4];  /* padding, used only for debugging at the moment */
} tetrad;
```

See also sections 12, 27, and 31.

This code is used in section 1.

12. Another way to think of this program is to regard it as solving a constraint satisfaction problem, whose variables have one of three forms: p_{ij}, r_{ik}, or c_{jk}. The domain of each variable is itself a set of variables, namely the "boys" in the matching problem for which this particular variable is a "girl." Thus, variable p_{ij} will have a domain consisting of variables of the form r_{ik}, because of the r_i matchings; variable c_{jk} will have a domain consisting of variables of the form p_{ij}, because of the c_j matchings; variable r_{ik} will have a domain consisting of variables of the form c_{jk}, because of the v_k matchings.

(We could also consider the domains to be options instead of variables/items, because the options have such a strict format.)

Each variable is identified internally by a number from 1 to $3z$, where z is the number of blank positions in the partial input square.

Key information for variable v is stored in its struct, $var[v]$, which has many four-byte fields. One of those fields, *name*, contains the three-character external name, used in printouts. Another field, *pos*, shows v's position in the *vars* array, which contains a permutation of all the variables; variable v is active (that is, not yet assigned a value) if and only if $var[v].pos < active$, where *active* is the number of currently active variables. A third field, *matching*, points to the bipartite matching problem for which v is currently a "girl." A fourth field, *tally*, counts the number of times this variable had no remaining options when forced moves were being propagated.

The other fields of a variable's struct contain data that enters into the GAD filtering algorithm. For example, we'll see below that Hopcroft and Karp's algorithm wants to store information in fields called *mate* and *mark*. Tarjan's algorithm wants to store information in fields called *rank*, *parent*, *arcs*, *link*, and *min*.

An attempt has been made to pair up these four-byte fields so that only one 8-byte memory access is needed to access two of them, as often as possible. (In particular, *bmate* and *gmate*, *mark* and *arcs*, *rank* and *link*, *parent* and *min* want to be buddies.)

⟨ Type definitions 11 ⟩ +≡

```
typedef struct {
  unsigned long long tally;   /* how often has this variable run into trouble? */
  int bmate;   /* a girl's boyfriend when matching */
  int gmate;   /* a boy's girlfriend when matching */
  int pos;   /* position of this variable in vars */
  int matching;   /* the current matching problem in which this var is a girl */
  int mark;   /* state indicator during the Hopcroft–Karp algorithm */
  int arcs;   /* first of a linked list of arcs */
  int rank;   /* serial number of a vertex in Tarjan's algorithm */
  int link;   /* stack pointer in Tarjan's algorithm */
  int parent;   /* predecessor in Tarjan's active tree */
  int min;   /* the magic ingredient of Tarjan's algorithm */
  char name[4];   /* variable's three-character name for printouts */
  int filler;   /* unused field, makes the size a multiple of eight bytes */
} variable;
```

13.

```
#define maxvars (3 * maxn * maxn)   /* upper bound on the number of variables */
```

⟨ Global variables 3 ⟩ +≡

```
tetrad *tet;   /* the tetrads in our data structures */
int vars[maxvars];   /* list of all variables, most active to least active */
int active;   /* this many variables are active */
variable var[maxvars + 1];   /* the variables' homes in our data structures */
```

14. Variable v is a primary item in an exact cover problem. Thus, when v is active, we want to maintain a list of all currently active options that include this item. That list is doubly linked and has a list header, as mentioned above; the header for v is $tet[v]$.

All tetrads following the list headers are grouped into sets of four, one for each option. This gives us extra breathing room, because an option contains only three items (namely p_{ij}, r_{ik}, c_{jk}) and could be packed into just three tetrads. We'll see that it's convenient to know that every option appears in four consecutive tetrads, $tet[a]$, $tet[a+1]$, $tet[a+2]$, $tet[a+3]$, where a is a multiple of 4; the first of these can be used to store information about the option as a whole, while the other three are devoted respectively to p_{ij}, r_{ik}, and c_{jk}.

15. ⟨Initialize the data structures 15⟩ ≡

```
active = mina = totvars = 3 * z;    /* this many variables */
for (p = i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      if (ooo, (¬P[i][j] ∧ ¬R[i][k] ∧ ¬C[j][k])) p++;    /* p options in all */
q = (totvars & −4) + 4 * (p + 1);    /* we'll allocate q tetras */
tet = (tetrad *) malloc(q * sizeof(tetrad));
if (¬tet) {
  fprintf(stderr, "Couldn't␣allocate␣the␣tetrad␣table!\n");
  exit(−66);
}
for (k = 0; k < totvars; k++) oo, vars[k] = k + 1, var[k + 1].pos = k;
for (k = 1; k ≤ totvars; k++) o, tet[k].up = tet[k].down = k;
⟨Name the variables 16⟩;
⟨Create the options 17⟩;
⟨Fix the len fields 18⟩;
```

See also sections 21 and 75.

This code is used in section 1.

16. ⟨Name the variables 16⟩ ≡

```
for (p = i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    if (P[i][j]) P[i][j] = 0;
    else P[i][j] = ++p, sprintf(var[p].name, "p"O"c"O"c", encode(i + 1), encode(j + 1));
  }
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++) {
    if (R[i][k]) R[i][k] = 0;
    else R[i][k] = ++p, sprintf(var[p].name, "r"O"c"O"c", encode(i + 1), encode(k + 1));
  }
for (j = 0; j < n; j++)
  for (k = 0; k < n; k++) {
    if (C[j][k]) C[j][k] = 0;
    else C[j][k] = ++p, sprintf(var[p].name, "c"O"c"O"c", encode(j + 1), encode(k + 1));
  }
```

This code is used in section 15.

17. Each option is given the name 'ijk' for use in printouts and debugging. No mems are charged for storing names, because printouts and debugging are not considered to be part of the problem-solving effort.

⟨ Create the options 17 ⟩ ≡

```
for (q = totvars & −4, i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      if (ooo, (P[i][j] ∧ R[i][k] ∧ C[j][k])) {
        q += 4;
        sprintf(tet[q].aux, ""O"c"O"c"O"c", encode(i + 1), encode(j + 1), encode(k + 1));
        sprintf(tet[q + 1].aux, "p"O"c"O"c", encode(i + 1), encode(j + 1));
        sprintf(tet[q + 2].aux, "r"O"c"O"c", encode(i + 1), encode(k + 1));
        sprintf(tet[q + 3].aux, "c"O"c"O"c", encode(j + 1), encode(k + 1));
        p = P[i][j];
        oo, tet[q + 1].itm = p, r = tet[p].up;
        ooo, tet[p].up = tet[r].down = q + 1, tet[q + 1].up = r;
        p = R[i][k];
        oo, tet[q + 2].itm = p, r = tet[p].up;
        ooo, tet[p].up = tet[r].down = q + 2, tet[q + 2].up = r;
        p = C[j][k];
        oo, tet[q + 3].itm = p, r = tet[p].up;
        ooo, tet[p].up = tet[r].down = q + 3, tet[q + 3].up = r;
      }
for (p = 1; p ≤ totvars; p++) oo, q = tet[p].up, tet[q].down = p;
```

This code is used in section 15.

18. The *itm* field in a list header makes no sense, so we've left it zero so far. But as in DLX, we'll want to know the length of every variable's option list. Thus we use $tet[v].itm$ to keep track of that length. (And when we do so, we'll call that field *len* instead of *itm*.)

#define *len itm*

⟨ Fix the *len* fields 18 ⟩ ≡

```
for (p = 1; p ≤ totvars; p++) {
  for (o, q = tet[p].down, k = 0; q ≠ p; o, q = tet[q].down) k++;
  o, tet[p].len = k;
}
```

This code is used in section 15.

19. A simple routine shows all the options in a given variable's list.

⟨ Subroutines 5 ⟩ +≡

```
void print_options(int v)
{
  register q;
  fprintf(stderr, "options␣for␣"O"s␣("O"sactive,␣length␣"O"d):\n", var[v].name,
      var[v].pos < active ? "" : "in", tet[v].len);
  for (q = tet[v].down; q ≠ v; q = tet[q].down) fprintf(stderr, "␣"O"s", tet[q & −4].aux);
  fprintf(stderr, "\n");
}
```

20. The other major data we need, besides the options, is the set of bipartite matching problems. GAD filtering will refine the original problems into smaller subproblems. These are all kept on a big stack called *mch* (a last-in-first-out list), with the initial problems at the bottom and their refinements at the top.

The "girls" of matching problem m, of size t, appear in $mch[m]$ through $mch[m+t-1]$, and the "boys" appear in $mch[m+t]$ through $mch[m+2t-1]$. The size itself is stored in $mch[m-1]$; and a few other facts about m are kept in $mch[m-2]$, etc.

```
#define msize −1   /* where to find the size of a matching */
#define mparent −2   /* where to find the matching that spawned this one */
#define mstamp −3   /* where to find the trigger for GAD filtering this one */
#define mprev −4   /* the address of the most recent matching */
#define mextra 4   /* this number of special entries begin a matching spec */
#define mchsize 1000000   /* the total size of the mch array */
```

⟨ Global variables 3 ⟩ +≡

```
  int totvars;   /* total number of variables */
  int mch[mchsize];   /* the big stack of matching problems */
  int mchptr = mextra;   /* the current top of this stack */
  int maxmchptr;   /* the largest value assumed by mchptr so far */
```

21. ⟨ Initialize the data structures 15 ⟩ +≡

```
  if (mchsize < 2 ∗ totvars + 4 ∗ n ∗ mextra) {
    fprintf(stderr, "Match␣table␣initial␣overflow␣(mchsize=%d)!\n", mchsize);
    exit(−667);
  }
  ⟨ Create the matching problems of type r_i 22 ⟩;
  ⟨ Create the matching problems of type c_j 23 ⟩;
  ⟨ Create the matching problems of type v_k 24 ⟩;
```

22. ⟨ Create the matching problems of type r_i 22 ⟩ ≡

```
  for (i = 0; i < n; i++) {
    for (p = j = 0; j < n; j++)
      if (o, P[i][j]) oo, mch[mchptr + p++] = P[i][j], var[P[i][j]].matching = mchptr;
    if (p) {
      mch[mchptr + msize] = p;
      for (k = 0; k < n; k++)
        if (o, R[i][k]) o, mch[mchptr + p++] = R[i][k];
      if (p ≠ 2 ∗ mch[mchptr + msize]) confusion("Ri␣girls␣!=␣boys", p);
      if (showsubproblems) print_match_prob(mchptr);
      q = mchptr, mchptr += p + mextra, mch[mchptr + mprev] = q;
      tofilter[tofiltertail++] = q, mch[q + mstamp] = 1;
    }
  }
```

This code is used in section 21.

23. ⟨ Create the matching problems of type c_j 23 ⟩ ≡

```
for (j = 0; j < n; j++) {
  for (p = k = 0; k < n; k++)
    if (o, C[j][k]) oo, mch[mchptr + p++] = C[j][k], var[C[j][k]].matching = mchptr;
  if (p) {
    mch[mchptr + msize] = p;
    for (i = 0; i < n; i++)
      if (o, P[i][j]) o, mch[mchptr + p++] = P[i][j];
    if (p ≠ 2 * mch[mchptr + msize]) confusion("Cj␣girls␣!=␣boys", p);
    if (showsubproblems) print_match_prob(mchptr);
    q = mchptr, mchptr += p + mextra, mch[mchptr + mprev] = q;
    tofilter[tofiltertail++] = q, mch[q + mstamp] = 1;
  }
}
```

This code is used in section 21.

24. ⟨ Create the matching problems of type v_k 24 ⟩ ≡

```
for (k = 0; k < n; k++) {
  for (p = i = 0; i < n; i++)
    if (o, R[i][k]) oo, mch[mchptr + p++] = R[i][k], var[R[i][k]].matching = mchptr;
  if (p) {
    mch[mchptr + msize] = p;
    for (j = 0; j < n; j++)
      if (o, C[j][k]) o, mch[mchptr + p++] = C[j][k];
    if (p ≠ 2 * mch[mchptr + msize]) confusion("Vk␣girls␣!=␣boys", p);
    if (showsubproblems) print_match_prob(mchptr);
    q = mchptr, mchptr += p + mextra, mch[mchptr + mprev] = q;
    tofilter[tofiltertail++] = q, mch[q + mstamp] = 1;
  }
}
```

This code is used in section 21.

25. ⟨ Subroutines 5 ⟩ +≡

```
void print_match_prob(int m)
{
  register int k;
  fprintf(stderr, "Matching␣problem␣"O"d␣(parent␣"O"d,␣size␣"O"d):\n", m, mch[m + mparent],
      mch[m + msize]);
  fprintf(stderr, "girls");
  for (k = 0; k < mch[m + msize]; k++) fprintf(stderr, "␣"O"s", var[mch[m + k]].name);
  fprintf(stderr, "\n");
  fprintf(stderr, "boys");
  for ( ; k < 2 * mch[m + msize]; k++) fprintf(stderr, "␣"O"s", var[mch[m + k]].name);
  fprintf(stderr, "\n");
}
```

26. This program differs from DLX not only because of GAD filtering but also because it considers forced moves to be part of the same node in the search tree. In other words, a new node of the search tree is created only when all active variables have at least two elements in their current domain. By contrast, DLX makes only one choice at each level of search.

A last-in-first-out list called the *trail* keeps track of what changes have been made to the database of options; this mechanism allows us to backtrack safely when needed. Some options have been deleted because they've been chosen to be in the final exact cover; others have been deleted because GAD filtering has proved them to be superfluous. The latter are indicated on the trail by adding 1 to their address (which is always a multiple of 4 as explained above).

Another last-in-first-out list, called *forced*, holds the names of options that should be forced at the current search tree node.

Finally, a *first*-in-first-out list called *tofilter* holds the names of matching problems that should be GAD-filtered because their set of edges has gotten smaller.

```
#define pruned 1     /* added to trail address of an option deleted by GAD */

⟨Subroutines 5⟩ +≡
  void print_forced(void)
  {     /* shows the currently forced options */
    register int k;
    for (k = 0; k < forcedptr; k++) fprintf(stderr, "␣"O"s", tet[forced[k]].aux);
    fprintf(stderr, "\n");
  }
  void print_tofilter(void)
  {     /* shows the currently scheduled filterings */
    register int k;
    for (k = tofilterhead; k ≠ tofiltertail; k = (k + 1) & qmod)
      fprintf(stderr, "␣"O"d("O"d)", tofilter[k], mch[tofilter[k] + msize]);
    fprintf(stderr, "\n");
  }
```

27. The path from the root to the currently active node is recorded as a sequence of node structs on the *move* stack.

```
⟨Type definitions 11⟩ +≡
  typedef struct {
    int mchptrstart;     /* mchptr at beginning of this node */
    int trailstart;     /* trailptr at beginning of this node */
    int branchvar;     /* the variable on which we're branching */
    int curchoice;     /* which of its options are we currently pursuing? */
    int choices;     /* how many options does it have? */
    int choiceno;     /* and what's the position of curchoice in that list? */
    unsigned long long nodeid;     /* node number (for printouts only) */
  } node;
```

28. ⟨ Global variables 3 ⟩ +≡

```
int trail[maxvars];   /* deleted options to be restored */
int trailptr;   /* the first unused element of trail */
int forced[maxvars];   /* options that must be chosen at current search node */
int forcedptr;   /* the first unused element of forced */
int tofilter[qmod + 1];   /* matchings that should be GAD filtered */
int tofilterhead, tofiltertail;   /* queue pointers for tofilter */
node move[maxvars];   /* the choices currently being investigated */
int level;   /* depth of the current search tree node */
int maxl;   /* maximum value of level so far */
int mina;   /* minimum value of active so far */
```

29. ⟨ Subroutines 5 ⟩ +≡

```
void print_trail(void)
{
  register int k, l;
  for (k = l = 0; k < trailptr; k++) {
    if (k ≡ move[l].trailstart) {
      fprintf(stderr, "---␣level␣"O"d\n", l);
      l++;
    }
    fprintf(stderr, "␣"O"s"O"s\n", tet[trail[k] & −4].aux, (trail[k] & #3) ? "*" : "");
  }
}
```

30. These data structures have plenty of redundancy, so plenty of things can go wrong. Here's a routine to detect some of the potential anomalies, which we hope to nip in the bud before they cause a major catastrophe.

```
#define sanity_checking 0   /* set this to 1 if you suspect a bug */
⟨Subroutines 5⟩ +≡
  void sanity(void)
  {
    register int k, v, p, l, q;
    for (k = 0; k < totvars; k++) {
      v = vars[k];
      if (var[v].pos ≠ k)
        fprintf(stderr, "wrong␣pos␣field␣in␣variable␣"O"d("O"s)!\n", v, var[v].name);
      if (k < active) {
        if (var[v].matching > move[level].mchptrstart) fprintf(stderr,
              "␣"O"s("O"d)␣has␣matching␣>␣"O"d!\n", var[v].name, v, move[level].mchptrstart);
        for (l = tet[v].len, p = tet[v].down, q = 0; q < l; q++, p = tet[p].down) {
          if (tet[tet[p].up].down ≠ p) fprintf(stderr, "up-down␣off␣at␣"O"d!\n", p);
          if (tet[tet[p].down].up ≠ p) fprintf(stderr, "down-up␣off␣at␣"O"d!\n", p);
          if (p ≡ v) {
            fprintf(stderr, "list␣"O"d("O"s)␣too␣short!\n", v, var[v].name);
            break;
          }
        }
        if (p ≠ v) fprintf(stderr, "list␣"O"d("O"s)␣too␣long!\n", v, var[v].name);
      }
    }
  }
```

31. The graph algorithms within GAD use a simple struct to represent a directed arc.

```
⟨Type definitions 11⟩ +≡
  typedef struct {
    int tip;    /* the vertex pointed to */
    int next;   /* the next arc from the vertex pointed from, or zero */
  } Arc;
```

32. GAD filtering, part one. Recall that every matching is of type r_i or c_j or v_k. For the computer, it means that the girls are respectively the p_{ij} or c_{jk} or r_{ik} items of the options 'p_{ij} r_{ik} c_{jk}' that represent the edges; the boys are respectively the r_{ki} or p_{ij} or c_{jk} items. We access those edges only from the girls' option lists, and the value of *del* tells us where the corresponding boy appears in each edge. (There's also *delp*, which indicates the unused part of that triple.)

⟨Local variables 4⟩ +≡
  register int *b*, *g*, *boy*, *girl*, *n*, *nn*, *del*, *delp*;

33. Here's how we check whether or not matching m is still feasible, given m and the current set of edges.

⟨Apply GAD filtering to matching m; **goto** *abort* if there's trouble 33⟩ ≡
  if (*showmatches*) *fprintf*(*stderr*, "GAD␣filtering␣for␣problem␣"*O*"d\n", m);
  GADstart = *mems*, *GADtries*++;
  o, *mch*[m + *mstamp*] = 0;   /* clear the flag that told us to do this check */
  o, n = *mch*[m + *msize*], *nn* = $n + n$;   /* get the size of this matching problem */
  switch (*oo*, *var*[*mch*[m]].*name*[0]) {   /* what kind of girls do we have here? */
  case 'p': *del* = +1, *delp* = +2; **break**;
  case 'c': *del* = −2, *delp* = −1; **break**;
  case 'r': *del* = +1, *delp* = −1; **break**;
  }
  ⟨Find a matching, or **goto** *abort* 34⟩;
  GADone += *mems* − *GADstart*;
  ⟨Refine this matching problem, if it splits into independent parts 47⟩;
  ⟨Purge any options that belong to different strong components 55⟩;
doneGAD: *GADtot* += *mems* − *GADstart*;

This code is used in section 68.

34. Some of the girls and boys might have become inactive, because of forced moves since this matching problem was set up, In such a case they already have their mates, and they'll be "refined out" as part of GAD filtering.

We begin by taking one pass over all the girls, trying to match up as many as we can. (Please excuse sexist language. I'm too old to make actual passes.)

⟨Find a matching, or **goto** *abort* 34⟩ ≡
  for ($b = n$; $b < nn$; b++) {
    o, *boy* = *mch*[$m + b$];
    if (o, *var*[*boy*].*pos* < *active*) *oo*, *var*[*boy*].*gmate* = *var*[*boy*].*mark* = 0;
      /* every active boy is initially free */
    else o, *var*[*boy*].*mark* = −2;
  }
  for ($f = g = 0$; $g < n$; g++) {
    o, *girl* = *mch*[$m + g$];
    if (o, *var*[*girl*].*pos* ≥ *active*) **continue**;   /* an inactive girl has her mate */
    for (o, a = *tet*[*girl*].*down*; $a \neq$ *girl*; o, a = *tet*[a].*down*) {
      o, *boy* = *tet*[a + *del*].*itm*;
      if (o, ¬*var*[*boy*].*gmate*) **break**;
    }
    if ($a \neq$ *girl*) *oo*, *var*[*girl*].*bmate* = *boy*, *var*[*boy*].*gmate* = *girl*;
    else *ooo*, *var*[*girl*].*bmate* = 0, *var*[*girl*].*parent* = f, *queue*[f++] = *girl*;   /* f girls are free */
  }
  if (f) ⟨Use the Hopcroft–Karp algorithm to complete the matching, or **goto** *abort* 36⟩;

This code is used in section 33.

35. The code here has essentially been transcribed from the program HOPCROFT-KARP, except that I've (shockingly?) deleted most of the comments. Readers are encouraged to study the exposition in that program, because many points of interest are discussed there.

```
⟨Local variables 4⟩ +≡
  register int f, qq, marks, fin_level;
```

36. ⟨Use the Hopcroft–Karp algorithm to complete the matching, or **goto** *abort* 36⟩ ≡

```
  if (showHK) ⟨Print the current matching 37⟩;
  for (r = 1; f; r++) {
    if (showHK) fprintf(stderr, "Beginning␣round␣"O"d...\n", r);
    ⟨Build the dag of shortest augmenting paths (SAPs) 38⟩;
    ⟨If there are no SAPs, goto abort 41⟩;
    ⟨Find a maximal set of disjoint SAPs, and incorporate them into the current matching 43⟩;
    if (showHK) {
      fprintf(stderr, "␣...␣"O"d␣pairs␣now␣matched␣(rank␣"O"d).\n", n − f, fin_level);
      ⟨Print the current matching 37⟩;
    }
  }
```

This code is used in section 34.

37. To report the matches-so-far, we simply show every boy's mate.

```
⟨Print the current matching 37⟩ ≡
  {
    for (p = n; p < nn; p++) {
      girl = var[mch[m + p]].gmate;
      fprintf(stderr, "␣"O"s", girl ? var[girl].name : "???");
    }
    fprintf(stderr, "\n");
  }
```

This code is used in section 36.

38. ⟨Build the dag of shortest augmenting paths (SAPs) 38⟩ ≡

```
  fin_level = −1, k = 0;    /* k entries have been compiled into tip and next */
  for (marks = l = i = 0, q = f; ; l++) {
    for (qq = q; i < qq; i++) {
      o, girl = queue[i];
      if (var[girl].pos ≥ active) confusion("inactive␣girl␣in␣SAP", girl);
      for (o, a = tet[girl].down; a ≠ girl; o, a = tet[a].down) {
        oo, boy = tet[a + del].itm, p = var[boy].mark;
        if (p ≡ 0) ⟨Enter boy into the dag 40⟩
        else if (p ≤ l) continue;
        if (showHK) fprintf(stderr, "␣"O"s->"O"s=>"O"s\n", var[boy].name, var[girl].name,
              var[girl].bmate ? var[var[girl].bmate].name : "bot");
        ooo, arc[++k].tip = girl, arc[k].next = var[boy].arcs, var[boy].arcs = k;
      }
    }
    if (q ≡ qq) break;    /* stop if nothing new on the queue for the next level */
  }
```

This code is used in section 36.

39. ⟨ Global variables 3 ⟩ +≡
```
int queue[maxn];    /* girls seen during the breadth-first search */
int marked[maxn];    /* which boys have been marked */
int dlink;    /* head of the list of free boys in the dag */
Arc arc[maxn + maxn];    /* suitable partners and links */
int lboy[maxn];    /* the boys being explored during the SAP demolition */
```

40. ⟨ Enter *boy* into the dag 40 ⟩ ≡
```
{
  if (fin_level ≥ 0 ∧ var[boy].gmate) continue;
  else if (fin_level < 0 ∧ (o, ¬var[boy].gmate)) fin_level = l, dlink = 0, q = qq;
  oo, var[boy].mark = l + 1, marked[marks++] = boy, var[boy].arcs = 0;
  if (o, var[boy].gmate) o, queue[q++] = var[boy].gmate;
  else {
    if (showHK) fprintf(stderr, "␣top->"O"s\n", var[boy].name);
    o, arc[++k].tip = boy, arc[k].next = dlink, dlink = k;
  }
}
```
This code is used in section 38.

41. We have no SAPs if and only no free boys were found.

⟨ If there are no SAPs, **goto** *abort* 41 ⟩ ≡
```
if (fin_level < 0) {
  if (showcauses) fprintf(stderr, "␣problem␣"O"d␣has␣no␣matching\n", m);
  GADone += mems − GADstart;
  GADtot += mems − GADstart;
  GADaborts++;
  goto abort;
}
```
This code is used in section 36.

42. ⟨ Reset all marks to zero 42 ⟩ ≡
```
while (marks) oo, var[marked[−−marks]].mark = 0;
```
This code is used in section 43.

43. ⟨ Find a maximal set of disjoint SAPs, and incorporate them into the current matching 43 ⟩ ≡

```
while (dlink) {
  o, boy = arc[dlink].tip, dlink = arc[dlink].next;
  l = fin_level;
enter_level: o, lboy[l] = boy;
advance: if (o, var[boy].arcs) {
    o, girl = arc[var[boy].arcs].tip, var[boy].arcs = arc[var[boy].arcs].next;
    o, b = var[girl].bmate;
    if (¬b) ⟨ Augment the current matching and continue 44 ⟩;
    if (o, var[b].mark < 0) goto advance;
    boy = b, l−−;
    goto enter_level;
  }
  if (++l > fin_level) continue;
  o, boy = lboy[l];
  goto advance;
}
⟨ Reset all marks to zero 42 ⟩;
```

This code is used in section 36.

44. At this point $girl = g_0$ and $boy = lboy[0] = b_0$ in an augmenting path. The other boys are $lboy[1]$, $lboy[2]$, etc.

⟨ Augment the current matching and **continue** 44 ⟩ ≡

```
{
  if (l) confusion("free␣girl", l);   /* free girls should occur only at level 0 */
  ⟨ Remove g from the list of free girls 45 ⟩;
  while (1) {
    if (showHK)
      fprintf(stderr, ""O"s␣"O"s-"O"s", l ? "," : "␣match", var[boy].name, var[girl].name);
    o, var[boy].mark = −1;
    ooo, j = var[boy].gmate, var[boy].gmate = girl, var[girl].bmate = boy;
    if (j ≡ 0) break;   /* boy was free */
    o, girl = j, boy = lboy[++l];
  }
  if (showHK) fprintf(stderr, "\n");
  continue;
}
```

This code is used in section 43.

45. ⟨ Remove g from the list of free girls 45 ⟩ ≡

```
f−−;   /* f is the number of free girls */
o, j = var[girl].parent;   /* where is girl in queue? */
ooo, i = queue[f], queue[j] = i, var[i].parent = j;   /* OK to clobber queue[f] */
```

This code is used in section 44.

46. GAD filtering, part two. Once a witness to a perfect matching is known, we can set up a directed acyclic graph whose strong components tell us whether or not we can reduce the remaining problem.

GAD filtering applies in general to cases where boys outnumber girls. The dag that's constructed is tripartite in such a case, and it's also somewhat complicated. But we're dealing with the simple case when boys and girls are equinumerous; so our dag is defined entirely on the set of boys. Boy b' has an arc to boy $b \neq b'$ if and only if b' is adjacent to a girl mated to b.

If that dag isn't strongly connected, we make progress! The boys in each of its strong components, and their mates, form smaller matching problems whose solutions can be found independently, without losing any solutions to the overall matching problem we began with. "Cross edges" between different strong components can therefore be deleted. (Technically speaking, the strong components correspond to minimal Hall sets, also known as elementary bigraphs.)

And we're in luck, because of Robert E. Tarjan's beautiful linear-time algorithm to find strong components. The code here follows closely the tried and true implementation of his algorithm that can be found in the program ROGET-COMPONENTS (part of The Stanford GraphBase).

Again I've (shockingly?) deleted most of the comments, and readers are encouraged to read the original exposition.

⟨Local variables 4⟩ +≡
  register int *stack*, *pboy*, *newn*;

47. ⟨Refine this matching problem, if it splits into independent parts 47⟩ ≡
  ⟨Make all vertices unseen and all arcs untagged 49⟩;
  ⟨Build the digraph for the current matching 48⟩;
  $r = stack = 0$;
  for $(b = n;\ b < nn;\ b{+}{+})$ {
    $o, v = mch[m + b]$;
    if $(o, \neg var[v].rank)$ {   /* vertex/boy v is still unseen */
      ⟨Perform a depth-first search with v as the root, finding the strong components of all unseen vertices reachable from v 50⟩;
    }
  }

This code is used in section 33.

48. ⟨Build the digraph for the current matching 48⟩ ≡
  for $(k = 0, g = 0;\ g < n;\ g{+}{+})$ {
    $o, girl = mch[m + g]$;
    if $(o, var[girl].pos \geq active)$ **continue**;
    $o, boy = var[girl].bmate$;
    for $(o, a = tet[girl].down;\ a \neq girl;\ o, a = tet[a].down)$ {
      $o, pboy = tet[a + del].itm$;
      if $(pboy \neq boy)$ $ooo, arc[{+}{+}k].tip = boy, arc[k].next = var[pboy].arcs, var[pboy].arcs = k$;
    }
  }

This code is used in section 47.

49. ⟨Make all vertices unseen and all arcs untagged 49⟩ ≡
  for $(b = n;\ b < nn;\ b{+}{+})$ {
    $o, boy = mch[m + b]$;
    $oo, var[boy].rank = var[boy].arcs = 0$;
  }

This code is used in section 47.

50. ⟨ Perform a depth-first search with v as the root, finding the strong components of all unseen vertices reachable from v 50 ⟩ ≡

```
{
  o, var[v].parent = 0;
  ⟨ Make vertex v active 51 ⟩;
  do ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and
      calling it v 52 ⟩ while (v);
}
```

This code is used in section 47.

51. ⟨ Make vertex v active 51 ⟩ ≡

```
oo, var[v].rank = ++r, var[v].link = stack, stack = v;
o, var[v].min = v;
```

This code is used in sections 50 and 52.

52. ⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 52 ⟩ ≡

```
{
  o, a = var[v].arcs;    /* v's first remaining untagged arc, if any */
  if (showT) fprintf(stderr, "␣Tarjan␣sees␣"O"s(rank␣"O"d)->"O"s\n", var[v].name, var[v].rank,
        a ? var[arc[a].tip].name : "/\\");
  if (a) {
    oo, u = arc[a].tip, var[v].arcs = arc[a].next;    /* tag the arc from v to u */
    if (o, var[u].rank) {    /* we've seen u already */
      if (oo, var[u].rank < var[var[v].min].rank) o, var[v].min = u;
          /* non-tree arc, just update var[v].min */
    } else {    /* u is presently unseen */
      o, var[u].parent = v;    /* the arc from v to u is a new tree arc */
      v = u;    /* u will now be the current vertex */
      ⟨ Make vertex v active 51 ⟩;
    }
  } else {    /* all arcs from v are tagged, so v matures */
    o, u = var[v].parent;    /* prepare to backtrack in the tree of active vertices */
    if (var[v].min ≡ v) ⟨ Remove v and all its successors on the active stack from the tree, and mark
          them as a strong component of the graph 53 ⟩
    else    /* the arc from u to v has just matured, making var[v].min visible from u */
     if (ooo, var[var[v].min].rank < var[var[u].min].rank) o, var[u].min = var[v].min;
    v = u;    /* the former parent of v is the new current vertex v */
  }
}
```

This code is used in section 50.

53. ⟨ Remove v and all its successors on the active stack from the tree, and mark them as a strong component of the graph 53 ⟩ ≡

```
{
  t = stack;
  o, stack = var[v].link;
  for (newn = 0, p = t; ; o, p = var[p].link) {
    o, var[p].rank = maxn + mchptr;    /* "infinity" */
    newn++;
    if (p ≡ v) break;
  }
  if (newn ≡ n) goto doneGAD;    /* sorry, there's no refinement yet */
  if (newn > 1 ∨ (o, var[v].pos < active)) {
    ⟨ Create a new matching subproblem for this strong component 54 ⟩;
    if (newn ≡ 1) {
      o, girl = var[v].gmate;
      for (o, a = tet[girl].down; a ≠ girl; o, a = tet[a].down)
        if (o, tet[a + del].itm ≡ v) break;
      if (a ≡ girl) confusion("lost␣option", girl);
      opt = a & −4;
      if (o, ¬tet[opt].itm) oo, tet[opt].itm = 1, forced[forcedptr++] = opt;
    }
  }
}
```

This code is used in section 52.

54. ⟨ Create a new matching subproblem for this strong component 54 ⟩ ≡

```
if (mchptr + mextra + newn + newn ≥ mchsize) {
  fprintf(stderr, "Match␣table␣overflow␣(mchsize="O"d)!\n", mchsize);
  exit(−666);
}
oo, mch[mchptr + mstamp] = 0, mch[mchptr + mparent] = m, mch[mchptr + msize] = newn;
for (k = mchptr; ; o, k++, t = var[t].link) {
  o, mch[k + newn] = t;
  ooo, girl = var[t].gmate, mch[k] = girl, var[girl].matching = mchptr;
  if (t ≡ v) break;
}
if (showsubproblems) print_match_prob(mchptr);
o, k = mchptr, mchptr += mextra + newn + newn, mch[mchptr + mprev] = k;
if (mchptr > maxmchptr) maxmchptr = mchptr;
```

This code is used in section 53.

55. Confession: I inserted a trick in this code, by adding *mchptr* to *maxn* when resetting the ranks of boys a new matching problem. I hope the reader will agree that it's a good trick.

⟨ Purge any options that belong to different strong components 55 ⟩ ≡
  for $(g = 0;\ g < n;\ g{+}{+})$ {
    $o, girl = mch[m + g]$;
    if $(o, var[girl].pos \geq active)$ **continue**;
    for $(o, a = tet[girl].down;\ a \neq girl;\ o, a = tet[a].down)$ {
      $o, boy = tet[a + del].itm$;
      if $(oo, maxn + var[girl].matching \neq var[boy].rank)$ {   /* different subproblems */
        $opt = a \mathbin{\&} -4$; ⟨ Delete the superfluous option *opt* 58 ⟩;
        $oo, t = var[tet[a + del].itm].matching$;
        if $(o, \neg mch[t + mstamp])$
          $oo, mch[t + mstamp] = 1, tofilter[tofiltertail] = t, tofiltertail = (tofiltertail + 1) \mathbin{\&} qmod$;
        $oo, t = var[tet[a + delp].itm].matching$;
        if $(o, \neg mch[t + mstamp])$
          $oo, mch[t + mstamp] = 1, tofilter[tofiltertail] = t, tofiltertail = (tofiltertail + 1) \mathbin{\&} qmod$;
      }
    }
  }

This code is used in section 33.

56. Hiding and unhiding. Now it's time to implement the basic operations by which options are deleted and later undeleted. The philosophy of "dancing links" operates here, because we are deleting from doubly linked lists.

To hide a tetrad, we simply delete it from the list that it's in. To hide an option, we hide all three of its tetrads. Unhiding does this in reverse.

Actually it's not quite as simple as it may sound, because deleting from a variable's list changes the length of that list. Therefore we schedule GAD filtering for that variable's matching.

Furthermore, the new length might be 1, in which case we schedule a forced move.

The new length might even be 0. In that case we set *foundzero* $= v$; but we don't abort immediately, because it's difficult to "partially undo" a complex sequence of updates. Later, when we reach a quiet time, *foundzero* will tell us to abort, after which all changes will be properly undone.

⟨ Hide the tetrad t 56 ⟩ ≡
```
oo, p = tet[t].up, q = tet[t].down, r = tet[t].itm;
oo, tet[p].down = q, tet[q].up = p;
oo, l = tet[r].len − 1, tet[r].len = l;
o, s = var[r].matching;
if (o, ¬mch[s + mstamp])
  oo, mch[s + mstamp] = 1, tofilter[tofiltertail] = s, tofiltertail = (tofiltertail + 1) & qmod;
if (l ≤ 1) {
  if (l ≡ 0) oo, var[r].tally++, zerofound = r;
  else {    /* prepare to force r */
    o, p = tet[r].down & −4;
    if (o, ¬tet[p].itm) oo, tet[p].itm = 1, forced[forcedptr++] = p;
  }
}
```
This code is used in sections 58 and 60.

57. ⟨ Unhide the tetrad t 57 ⟩ ≡
```
oo, p = tet[t].up, q = tet[t].down, r = tet[t].itm;
oo, tet[p].down = tet[q].up = t;
oo, l = tet[r].len + 1, tet[r].len = l;
```
This code is used in sections 59 and 61.

58. ⟨ Delete the superfluous option *opt* 58 ⟩ ≡
```
{
  if (showprunes) fprintf(stderr, "␣pruning␣"O"s\n", tet[opt].aux);
  o, trail[trailptr++] = opt + pruned;
  o, tet[opt].up = 1;    /* mark a deleted option */
  zerofound = 0;
  t = opt + 1; ⟨ Hide the tetrad t 56 ⟩;
  t = opt + 2; ⟨ Hide the tetrad t 56 ⟩;
  t = opt + 3; ⟨ Hide the tetrad t 56 ⟩;
  if (zerofound) {
    if (showcauses) fprintf(stderr, "␣no␣options␣for␣"O"s\n", var[zerofound].name);
    goto abort;
  }
}
```
This code is used in section 55.

59. ⟨ Undelete the superfluous option *opt* 59 ⟩ ≡

```
{
  t = opt + 3; ⟨ Unhide the tetrad t 57 ⟩;
  t = opt + 2; ⟨ Unhide the tetrad t 57 ⟩;
  t = opt + 1; ⟨ Unhide the tetrad t 57 ⟩;
  o, tet[opt].up = 0;
}
```

This code is used in section 70.

60. Now we implement the fundamental mechanism that contributes an option to the final exact cover, causing three variables to become inactive (thus "frozen" until we backtrack later).

The main point of interest is that we keep the three option lists intact, so that we can undo this operation later. But we hide everything else in sight.

⟨ Force the option *opt* 60 ⟩ ≡

```
{
  if (showmoves) fprintf(stderr, " forcing "O"s\n", tet[opt].aux);
  if (o, tet[opt].up) {
    if (showcauses) fprintf(stderr, " option "O"s was deleted\n", tet[opt].aux);
    goto abort;
  }
  zerofound = 0;
  o, trail[trailptr++] = opt;
  ooo, pij = tet[opt + 1].itm, rik = tet[opt + 2].itm, cjk = tet[opt + 3].itm;
  o, m = var[pij].matching;
  if (¬mch[m + mstamp])
    oo, mch[m + mstamp] = 1, tofilter[tofiltertail] = m, tofiltertail = (tofiltertail + 1) & qmod;
  o, m = var[rik].matching;
  if (¬mch[m + mstamp])
    oo, mch[m + mstamp] = 1, tofilter[tofiltertail] = m, tofiltertail = (tofiltertail + 1) & qmod;
  o, m = var[cjk].matching;
  if (¬mch[m + mstamp])
    oo, mch[m + mstamp] = 1, tofilter[tofiltertail] = m, tofiltertail = (tofiltertail + 1) & qmod;
  ⟨ Make pij, rik, cjk inactive 62 ⟩;
  for (o, a = tet[pij].down; a ≠ pij; o, a = tet[a].down)
    if (a ≠ opt + 1) {
      t = a + 1; ⟨ Hide the tetrad t 56 ⟩;
      t = a + 2; ⟨ Hide the tetrad t 56 ⟩;
    }
  for (o, a = tet[rik].down; a ≠ rik; o, a = tet[a].down)
    if (a ≠ opt + 2) {
      t = a + 1; ⟨ Hide the tetrad t 56 ⟩;
      t = a − 1; ⟨ Hide the tetrad t 56 ⟩;
    }
  for (o, a = tet[cjk].down; a ≠ cjk; o, a = tet[a].down)
    if (a ≠ opt + 3) {
      t = a − 2; ⟨ Hide the tetrad t 56 ⟩;
      t = a − 1; ⟨ Hide the tetrad t 56 ⟩;
    }
  if (zerofound) {
    if (showcauses) fprintf(stderr, " no options for "O"s\n", var[zerofound].name);
    goto abort;
  }
}
```

This code is used in sections 67 and 68.

61. ⟨ Unforce the option *opt* 61 ⟩ ≡

```
{
  ooo, pij = tet[opt + 1].itm, rik = tet[opt + 2].itm, cjk = tet[opt + 3].itm;
  for (o, a = tet[cjk].up; a ≠ cjk; o, a = tet[a].up)
    if (a ≠ opt + 3) {
      t = a − 2; ⟨ Unhide the tetrad t 57 ⟩;
      t = a − 1; ⟨ Unhide the tetrad t 57 ⟩;
    }
  for (o, a = tet[rik].up; a ≠ rik; o, a = tet[a].up)
    if (a ≠ opt + 2) {
      t = a − 1; ⟨ Unhide the tetrad t 57 ⟩;
      t = a + 1; ⟨ Unhide the tetrad t 57 ⟩;
    }
  for (o, a = tet[pij].up; a ≠ pij; o, a = tet[a].up)
    if (a ≠ opt + 1) {
      t = a + 2; ⟨ Unhide the tetrad t 57 ⟩;
      t = a + 1; ⟨ Unhide the tetrad t 57 ⟩;
    }
  active += 3;      /* hooray for the sparse-set technique */
}
```

This code is used in section 70.

62. This step sets the mates so that GAD filtering will know how to deal with these newly inactive variables.

⟨ Make *pij*, *rik*, *cjk* inactive 62 ⟩ ≡

```
  o, var[pij].bmate = rik, var[pij].gmate = cjk;
  o, p = var[pij].pos;
  if (p ≥ active) confusion("inactive␣pij", pij);
  o, v = vars[−− active];
  oo, vars[active] = pij, var[pij].pos = active;
  o, vars[p] = v, var[v].pos = p;
  o, var[rik].bmate = cjk, var[rik].gmate = pij;
  o, p = var[rik].pos;
  if (p ≥ active) confusion("inactive␣rik", rik);
  o, v = vars[−− active];
  o, vars[active] = rik, var[rik].pos = active;
  o, vars[p] = v, var[v].pos = p;
  o, var[cjk].bmate = pij, var[cjk].gmate = rik;
  o, p = var[cjk].pos;
  if (p ≥ active) confusion("inactive␣cjk", cjk);
  o, v = vars[−− active];
  o, vars[active] = cjk, var[cjk].pos = active;
  o, vars[p] = v, var[v].pos = p;
```

This code is used in section 60.

63. ⟨ Local variables 4 ⟩ +≡

```
  int bvar, opt, pij, rik, cjk, vv, zerofound, maxtally;
```

64. The search tree. As stated above, the backtracking in this program traverses an implicit search tree whose structure is somewhat different from that of DLX, because "forced moves" are incorporated into the tree node in which they were forced. Filtering operations are also included in each node. (Thus the structure conforms more to some of the CSP-solving programs I've been reading.)

The basic idea is to keep going until forcing and filtering give no further information. Then we choose a variable on which to branch. If that variable has t possible values, we implicitly branch into t subtrees, one at a time. Each of those subtrees begins with a forced move to set one of those t values; then we let things play out until again becoming quiescent (and branching again), or until we actually find a solution (oh happy day), or until a contradiction arises. In the latter case, the program says '**goto** *abort*'; we carefully undo all the steps since the beginning of this subnode, then move to the next of the t alternatives. Eventually we'll have tried all t of the possibilities; it will be time to abort again, until we've explored the entire tree.

65. To launch this process, essentially at the root node, we check to see if any forced moves or contradictions were present in the original problem. (It's easy to construct partial latin squares that obviously have no completion.) That gives us the opportunity to reach our first stable state and we'll be ready to make the first branch.

```
⟨Prime the pump at the root node 65⟩ ≡
  o, move[0].mchptrstart = mchptr;
  for (v = 1; v ≤ totvars; v++)
    if (o, tet[v].len ≤ 1) {
      if (¬tet[v].len) {
        if (showcauses) fprintf(stderr, "␣"O"s␣already␣has␣no␣options!\n", var[v].name);
        goto abort;
      }
      o, t = tet[v].down & −4;      /* schedule a forced move, but don't do it yet */
      if (o, ¬tet[t].itm) oo, tet[t].itm = 1, forced[forcedptr++] = t;
    }
```

This code is used in section 72.

66. When we are ready to branch, we use the MRV heuristic ("minimum remaining values"), by finding an active variable with the smallest domain. This domain should have at least two elements, because of our forcing strategy. And fortunately it also seems to have at *most* two elements, in most of the problems that I'm particularly anxious to solve.

Of course I do check to see that no forced moves have been overlooked. Bugs lurk everywhere and I must constantly be on the lookout for flaws in my reasoning.

```
⟨Choose the variable for branching 66⟩ ≡
  if (showdomains) fprintf(stderr, "Branching␣at␣level␣"O"d:", level);
  for (t = totvars, k = 0; k < active; k++) {
    o, v = vars[k];
    if (showdomains) fprintf(stderr, "␣"O"s("O"d)", var[v].name, tet[v].len);
    if (o, tet[v].len ≤ t) {
      if (tet[v].len ≤ 1) confusion("missed␣force", v);
      if (tet[v].len < t) oo, bvar = v, t = tet[v].len, maxtally = var[v].tally;
      else if (o, var[v].tally > maxtally) o, bvar = v, t = tet[v].len, maxtally = var[v].tally;
    }
  }
  if (showdomains) fprintf(stderr, "\n");
```

This code is used in section 67.

67. Here now is the main loop, which is the context within which most of this program operates.

⟨ Main loop 67 ⟩ ≡

```
choose: level++;
  if (level > maxl) maxl = level;
  ⟨ Choose the variable for branching 66 ⟩;
  o, move[level].mchptrstart = mchptr, move[level].trailstart = trailptr;
  o, move[level].branchvar = bvar, move[level].choices = t;
  o, move[level].curchoice = tet[bvar].down, move[level].choiceno = 1;
enternode: move[level].nodeid = ++nodes;
  if (sanity_checking) sanity();
  if (shownodes) {
    v = move[level].branchvar;
    u = tet[move[level].curchoice + (var[v].name[0] ≡ 'c' ? -2 : +1)].itm;
    fprintf(stderr, "L"O"d:␣"O"s="O"s␣("O"d␣of␣"O"d),␣node␣"O"lld,␣"O"lld␣mems\n", level,
        var[v].name, var[u].name, move[level].choiceno, move[level].choices, move[level].nodeid, mems);
  }
  if (mems ≥ thresh) {
    thresh += delta;
    if (showlong) print_state();
    else print_progress();
  }
  o, opt = move[level].curchoice & -4;
  ⟨ Force the option opt 60 ⟩;
mainplayer: ⟨ Carry out all scheduled forcings and filterings until none remain 68 ⟩;
  if (active < mina) mina = active;
  if (active) goto choose;
  count++;
  if (showsols) ⟨ Print a solution 78 ⟩;
abort: if (level) {
    ⟨ Cancel all scheduled forcing and filtering 69 ⟩;
    ⟨ Unrefine all refinements made at this level 71 ⟩;
    ⟨ Roll back the trail to the beginning of this level 70 ⟩;
    if (o, move[level].choiceno < move[level].choices) {
      oo, move[level].curchoice = tet[move[level].curchoice].down;
      o, move[level].choiceno++;
      goto enternode;
    }
    level--;
    if (showcauses) fprintf(stderr, "done␣with␣branches␣from␣node␣"O"lld\n", move[level].nodeid);
    goto abort;
  }
```

This code is used in section 72.

68. A queue is used for matchings to be filtered, because we want to maximize the time between initial scheduling and actual filtering. (Filtering does more when fewer options remain.) On the other hand, there's no reason to delay a forcing, so we use a stack for that.

⟨ Carry out all scheduled forcings and filterings until none remain 68 ⟩ ≡
while (1) {
  while (*forcedptr*) {
    o, *opt* = *forced*[−−*forcedptr*];
    o, *tet*[*opt*].*itm* = 0;   /∗ this option is no longer on the *forced* stack ∗/
    ⟨ Force the option *opt* 60 ⟩;
  }
  if (*tofilterhead* ≡ *tofiltertail*) **break**;
  o, *m* = *tofilter*[*tofilterhead*], *tofilterhead* = (*tofilterhead* + 1) & *qmod*;
  o, *mch*[*m* + *mstamp*] = 0;   /∗ this matching is no longer in the *tofilter* queue ∗/
  ⟨ Apply GAD filtering to matching *m*; **goto** *abort* if there's trouble 33 ⟩;
}

This code is used in section 67.

69. ⟨ Cancel all scheduled forcing and filtering 69 ⟩ ≡
while (*forcedptr*) {
  o, *opt* = *forced*[−−*forcedptr*];
  o, *tet*[*opt*].*itm* = 0;
}
while (*tofilterhead* ≠ *tofiltertail*) {
  o, *m* = *tofilter*[*tofilterhead*], *tofilterhead* = (*tofilterhead* + 1) & *qmod*;
  o, *mch*[*m* + *mstamp*] = 0;
}

This code is used in section 67.

70. ⟨ Roll back the trail to the beginning of this level 70 ⟩ ≡
o;   /∗ fetch *move*[*level*].*trailstart* and *move*[*level*].*mchptrstart* ∗/
while (*trailptr* ≠ *move*[*level*].*trailstart*) {
  o, *opt* = *trail*[−−*trailptr*] & −4;
  if (*trail*[*trailptr*] & #3) ⟨ Undelete the superfluous option *opt* 59 ⟩
  else ⟨ Unforce the option *opt* 61 ⟩;
}

This code is used in section 67.

71. ⟨ Unrefine all refinements made at this level 71 ⟩ ≡
while (*mchptr* > *move*[*level*].*mchptrstart*) {
  oo, *m* = *mch*[*mchptr* + *mprev*], *n* = *mch*[*m* + *msize*], *p* = *mch*[*m* + *mparent*];
  for (*k* = 0; *k* < *n*; *k*++) *oo*, *var*[*mch*[*m* + *k*]].*matching* = *p*;
  mchptr = *m*;
}
if (*mchptr* ≠ *move*[*level*].*mchptrstart*) *confusion*(`"mchptrstart"`, *mchptr* − *move*[*level*].*mchptrstart*);

This code is used in section 67.

72. ⟨ Solve the problem 72 ⟩ ≡
⟨ Prime the pump at the root node 65 ⟩;
goto *mainplayer*;
⟨ Main loop 67 ⟩;

This code is used in section 1.

73. Learning from previous runs. The tally counts have turned out to be tremendously helpful. But they have no effect whatsoever on the first dozen or so levels of the tree, except after the algorithm has been run using bad choices for awhile.

So I'm experimenting with the idea of running for awhile, then saving the tallies-so-far and restarting.

The following subroutine stores the current tallies, for a problem with z variables, in a file whose name is '`plgad`z`.tally`'.

```
#define tallyfiletemplate "plgad"O"d.tally"

⟨ Subroutines 5 ⟩ +≡
  void save_tallies(int z)
  {
    register int v;
    sprintf(tallyfilename, tallyfiletemplate, z);
    tallyfile = fopen(tallyfilename, "w");
    if (¬tallyfile) {
      fprintf(stderr, "I can't open file `"O"s' for writing!\n", tallyfilename);
    } else {
      for (v = 1; v ≤ z; v++) fprintf(tallyfile, ""O"20lld "O"s\n", var[v].tally, var[v].name);
      fclose(tallyfile);
      fprintf(stderr, "Tallies saved in file `"O"s'.\n", tallyfilename);
    }
  }
```

74. ⟨ Global variables 3 ⟩ +≡

```
  FILE *tallyfile;
  char tallyfilename[32];
```

75. We check at the beginning whether a tally file is available.

```
⟨ Initialize the data structures 15 ⟩ +≡
  sprintf(tallyfilename, tallyfiletemplate, totvars);
  tallyfile = fopen(tallyfilename, "r");
  if (tallyfile) {
    for (v = 1; v ≤ totvars; v++) {
      if (¬fgets(buf, bufsize, tallyfile)) break;
      if (var[v].name[0] ≠ buf[21] ∨ var[v].name[1] ≠ buf[22] ∨ var[v].name[2] ≠ buf[23]) break;
      sscanf(buf, ""O"20lld", &var[v].tally);
    }
    if (v ≤ totvars)
      for (v−−; v ≥ 1; v−−) var[v].tally = 0;      /* oops, wrong file */
    else fprintf(stderr, "(tallies initialized from file `"O"s')\n", tallyfilename);
  }
```

76. Miscellaneous loose ends. In a long run, it's nice to know how much of the search tree has been explored. The computer's best guess, based on the assumption that the tree-so-far is typical of the tree-as-a-whole, is computed by the following routine copied from DLX1.

⟨ Subroutines 5 ⟩ +≡

```
void print_progress(void)
{
  register int l, k, d, c, p;
  register double f, fd;
  fprintf(stderr, " after "O"lld mems: "O"lld sols,", mems, count);
  for (f = 0.0, fd = 1.0, l = 1; l < level; l++) {
    k = move[l].choiceno, d = move[l].choices;
    fd *= d, f += (k - 1)/fd;    /* choice at level l is k of d */
    fprintf(stderr, " "O"c"O"c", encode(k), encode(d));
  }
  fprintf(stderr, " "O".5f\n", f + 0.5/fd);
}
```

77. A longer progress report shows the entire *move* stack.

⟨ Subroutines 5 ⟩ +≡

```
void print_state(void)
{
  register int l, v;
  fprintf(stderr, "Current state (level "O"d):\n", level);
  for (l = 1; l ≤ level; l++) {
    switch (move[l].curchoice & #3) {
    case 1: case 2: v = tet[move[l].curchoice + 1].itm; break;
    case 3: v = tet[move[l].curchoice - 2].itm; break;
    }
    fprintf(stderr, " "O"s="O"s ("O"d of "O"d), node "O"lld\n", var[move[l].branchvar].name,
        var[v].name, move[l].choiceno, move[l].choices, move[l].nodeid);
  }
  fprintf(stderr, " "O"lld solution"O"s, "O"lld mems, maxl "O"d, mina "O"d so far.\n",
      count, count ≡ 1 ? "" : "s", mems, maxl, mina);
}
```

78. ⟨ Print a solution 78 ⟩ ≡

```
{
  printf("Solution␣#"O"lld:\n", count);
  for (t = 0; t < trailptr; t++)
    if ((trail[t] & #3) ≡ 0) {
      opt = trail[t];
      i = decode(tet[opt].aux[0]);
      j = decode(tet[opt].aux[1]);
      k = decode(tet[opt].aux[2]);
      board[i − 1][j − 1] = k;
    }
  for (i = 0; i < originaln; i++) {
    for (j = 0; j < originaln; j++) printf(""O"c", encode(board[i][j]));
    printf("\n");
  }
  print_state();
}
```

This code is used in section 67.

79. And all's well that ends well. (Unless there was a bug.)

⟨ Say farewell 79 ⟩ ≡

```
  save_tallies(totvars);
  fprintf(stderr, "Altogether␣"O"llu␣solution"O"s,␣"O"llu␣mems,␣"O"llu␣nodes.\n", count,
      count ≡ 1 ? "" : "s", mems, nodes);
  fprintf(stderr, "(GAD␣time␣"O"llu+"O"llu,␣"O"llu/"O"llu␣aborted;", GADone,
      GADtot − GADone, GADaborts, GADtries);
  fprintf(stderr, "␣maxl="O"d,␣mina="O"d,␣maxmchptr="O"d)\n", maxl, mina, maxmchptr);
```

This code is used in section 1.

80. Index.

⟨ Apply GAD filtering to matching m; **goto** *abort* if there's trouble 33 ⟩ Used in section 68.
⟨ Augment the current matching and **continue** 44 ⟩ Used in section 43.
⟨ Build the dag of shortest augmenting paths (SAPs) 38 ⟩ Used in section 36.
⟨ Build the digraph for the current matching 48 ⟩ Used in section 47.
⟨ Cancel all scheduled forcing and filtering 69 ⟩ Used in section 67.
⟨ Carry out all scheduled forcings and filterings until none remain 68 ⟩ Used in section 67.
⟨ Choose the variable for branching 66 ⟩ Used in section 67.
⟨ Create a new matching subproblem for this strong component 54 ⟩ Used in section 53.
⟨ Create the matching problems of type c_j 23 ⟩ Used in section 21.
⟨ Create the matching problems of type r_i 22 ⟩ Used in section 21.
⟨ Create the matching problems of type v_k 24 ⟩ Used in section 21.
⟨ Create the options 17 ⟩ Used in section 15.
⟨ Delete the superfluous option *opt* 58 ⟩ Used in section 55.
⟨ Enter *boy* into the dag 40 ⟩ Used in section 38.
⟨ Explore one step from the current vertex v, possibly moving to another current vertex and calling it v 52 ⟩ Used in section 50.
⟨ Find a matching, or **goto** *abort* 34 ⟩ Used in section 33.
⟨ Find a maximal set of disjoint SAPs, and incorporate them into the current matching 43 ⟩ Used in section 36.
⟨ Fix the *len* fields 18 ⟩ Used in section 15.
⟨ Force the option *opt* 60 ⟩ Used in sections 67 and 68.
⟨ Global variables 3, 13, 20, 28, 39, 74 ⟩ Used in section 1.
⟨ Hide the tetrad t 56 ⟩ Used in sections 58 and 60.
⟨ If there are no SAPs, **goto** *abort* 41 ⟩ Used in section 36.
⟨ Initialize the data structures 15, 21, 75 ⟩ Used in section 1.
⟨ Input the partial latin square 6 ⟩ Used in section 1.
⟨ Local variables 4, 32, 35, 46, 63 ⟩ Used in section 1.
⟨ Main loop 67 ⟩ Used in section 72.
⟨ Make all vertices unseen and all arcs untagged 49 ⟩ Used in section 47.
⟨ Make vertex v active 51 ⟩ Used in sections 50 and 52.
⟨ Make *pij*, *rik*, *cjk* inactive 62 ⟩ Used in section 60.
⟨ Name the variables 16 ⟩ Used in section 15.
⟨ Perform a depth-first search with v as the root, finding the strong components of all unseen vertices reachable from v 50 ⟩ Used in section 47.
⟨ Prime the pump at the root node 65 ⟩ Used in section 72.
⟨ Print a solution 78 ⟩ Used in section 67.
⟨ Print the current matching 37 ⟩ Used in section 36.
⟨ Purge any options that belong to different strong components 55 ⟩ Used in section 33.
⟨ Refine this matching problem, if it splits into independent parts 47 ⟩ Used in section 33.
⟨ Remove g from the list of free girls 45 ⟩ Used in section 44.
⟨ Remove v and all its successors on the active stack from the tree, and mark them as a strong component of the graph 53 ⟩ Used in section 52.
⟨ Reset all marks to zero 42 ⟩ Used in section 43.
⟨ Roll back the trail to the beginning of this level 70 ⟩ Used in section 67.
⟨ Say farewell 79 ⟩ Used in section 1.
⟨ Solve the problem 72 ⟩ Used in section 1.
⟨ Subroutines 5, 19, 25, 26, 29, 30, 73, 76, 77 ⟩ Used in section 1.
⟨ Type definitions 11, 12, 27, 31 ⟩ Used in section 1.
⟨ Undelete the superfluous option *opt* 59 ⟩ Used in section 70.
⟨ Unforce the option *opt* 61 ⟩ Used in section 70.
⟨ Unhide the tetrad t 57 ⟩ Used in sections 59 and 61.
⟨ Unrefine all refinements made at this level 71 ⟩ Used in section 67.
⟨ Use the Hopcroft–Karp algorithm to complete the matching, or **goto** *abort* 36 ⟩ Used in section 34.

PARTIAL-LATIN-GAD