Important: Before reading ASSIGN_LISA, please read or at least skim the program for GB_LISA.

1. The assignment problem. This demonstration program takes a matrix of numbers constructed by the GB_LISA module and chooses at most one number from each row and column in such a way as to maximize the sum of the numbers chosen. It also reports the number of "mems" (memory references) expended during its computations, so that the algorithm it uses can be compared with alternative procedures.

The matrix has m rows and n columns. If $m \le n$, one number will be chosen in each row; if $m \ge n$, one number will be chosen in each column. The numbers in the matrix are brightness levels (pixel values) in a digitized version of the Mona Lisa.

Of course the author does not pretend that the location of "highlights" in da Vinci's painting, one per row and one per column, has any application to art appreciation. However, this program does seem to have pedagogic value, because the relation between pixel values and shades of gray allows us to visualize the data underlying this special case of the assignment problem; ordinary matrices of numeric data are much harder to perceive. The nonrandom nature of pixels in a work of art might also have similarities to the "organic" properties of data in real-world applications.

This program is optionally able to produce an encapsulated PostScript file from which the solution can be displayed graphically, with halftone shading.

2. As explained in GB_LISA, the subroutine call $lisa(m, n, d, m0, m1, n0, n1, d0, d1, area)$ constructs an $m \times n$ matrix of integers between 0 and d, inclusive, based on the brightness levels in a rectangular region of a digitized Mona Lisa, where $m0$, $m1$, $n0$, and $n1$ define that region. The raw data is obtained as a sum of $(m1 - m0)(n1 - n0)$ pixel values between 0 and 255, then scaled in such a way that sums $\le d0$ are mapped to zero, sums $\ge d1$ are mapped to d, and intermediate sums are mapped linearly to intermediate values. Default values $m1 = 360$, $n1 = 250$, $m = m1 - m0$, $n = n1 - n0$, $d = 255$, and $d1 = 255(m1 - m0)(n1 - n0)$ are substituted if any of the parameters m, n, d, $m1$, $n1$, or $d1$ are zero.

The user can specify the nine parameters $(m, n, d, m0, m1, n0, n1, d0, d1)$ on the command line, at least in a UNIX implementation, thereby obtaining a variety of special effects; the relevant command-line options are `m=`⟨number⟩, `m0=`⟨number⟩, and so on, with no spaces before or after the `=` signs that separate parameter names from parameter values. Additional options are also provided: `-s` (use only Mona Lisa's 16×32 "smile"); `-e` (use only her 20×50 eyes); `-c` (complement black/white); `-p` (print the matrix and solution); `-P` (produce a PostScript file `lisa.eps` for graphic output); `-h` (use a heuristic that applies only when $m = n$); and `-v` or `-V` (print verbose or Very verbose commentary about the algorithm's performance).

Here is the overall layout of this C program:

```
#include "gb_graph.h"   /* the GraphBase data structures */
#include "gb_lisa.h"    /* the lisa routine */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 3 ⟩
  main(argc, argv)
      int argc;   /* the number of command-line arguments */
      char *argv[];   /* an array of strings containing those arguments */
      { ⟨ Local variables 4 ⟩
      ⟨ Scan the command-line options 5 ⟩;
      mtx = lisa(m, n, d, m0, m1, n0, n1, d0, d1, working_storage);
      if (mtx ≡ Λ) {
        fprintf(stderr, "Sorry,␣can't␣create␣the␣matrix!␣(error␣code␣%ld)\n", panic_code);
        return −1;
      }
      printf ("Assignment␣problem␣for␣%s%s\n", lisa_id, ( ∼? ",␣complemented" : "" ) ) ;
      sscanf(lisa_id, "lisa(%lu,%lu,%lu", &m, &n, &d);   /* adjust for defaults */
      if (m ≠ n) heur = 0;
      if (printing) ⟨ Display the input matrix 6 ⟩;
      if (PostScript) ⟨ Output the input matrix in PostScript format 28 ⟩;
      mems = 0;
      ⟨ Solve the assignment problem 24 ⟩;
      if (printing) ⟨ Display the solution 27 ⟩;
      if (PostScript) ⟨ Output the solution in PostScript format 31 ⟩;
      printf("Solved␣in␣%ld␣mems%s.\n", mems, (heur ? "␣with␣square-matrix␣heuristic" : ""));
      return 0;   /* normal exit */
      }
```

3. ⟨ Global variables 3 ⟩ ≡

```
  Area working_storage;   /* where to put the input data and auxiliary arrays */
  long *mtx;   /* input data for the assignment problem */
  long mems;   /* the number of memory references counted while solving the problem */
```

See also section 29.

This code is used in section 2.

4. The following local variables are related to the command-line options:

⟨ Local variables 4 ⟩ ≡
```
unsigned long m = 0, n = 0;   /* number of rows and columns desired */
unsigned long d = 0;   /* number of pixel values desired, minus 1 */
unsigned long m0 = 0, m1 = 0;   /* input will be from rows [m0 .. m1) */
unsigned long n0 = 0, n1 = 0;   /* and from columns [n0 .. n1) */
unsigned long d0 = 0, d1 = 0;   /* lower and upper threshold of raw pixel scores */
long ~ = 0;   /* should the input values be complemented? */
long heur = 0;   /* should the square-matrix heuristic be used? */
long printing = 0;   /* should the input matrix and solution be printed? */
long PostScript = 0;   /* should an encapsulated PostScript file be produced? */
```
See also sections 13, 14, and 26.

This code is used in section 2.

5. ⟨ Scan the command-line options 5 ⟩ ≡
```
while (--argc) {
if (sscanf(argv[argc], "m=%lu", &m) ≡ 1) ;
else if (sscanf(argv[argc], "n=%lu", &n) ≡ 1) ;
else if (sscanf(argv[argc], "d=%lu", &d) ≡ 1) ;
else if (sscanf(argv[argc], "m0=%lu", &m0) ≡ 1) ;
else if (sscanf(argv[argc], "m1=%lu", &m1) ≡ 1) ;
else if (sscanf(argv[argc], "n0=%lu", &n0) ≡ 1) ;
else if (sscanf(argv[argc], "n1=%lu", &n1) ≡ 1) ;
else if (sscanf(argv[argc], "d0=%lu", &d0) ≡ 1) ;
else if (sscanf(argv[argc], "d1=%lu", &d1) ≡ 1) ;
else if (strcmp(argv[argc], "-s") ≡ 0) {
  smile;   /* sets m0, m1, n0, n1 */
  d1 = 100000;   /* makes the pixels brighter */
}
else if (strcmp(argv[argc], "-e") ≡ 0) {
  eyes;
  d1 = 200000;
}
else if (strcmp(argv[argc], "-c") ≡ 0) ~= 1;
else
  if (strcmp(argv[argc], "-h") ≡ 0) heur = 1;
  else if (strcmp(argv[argc], "-v") ≡ 0) verbose = 1;
  else if (strcmp(argv[argc], "-V") ≡ 0) verbose = 2;   /* terrifically verbose */
  else if (strcmp(argv[argc], "-p") ≡ 0) printing = 1;
  else if (strcmp(argv[argc], "-P") ≡ 0) PostScript = 1;
  else {
    fprintf(stderr, "Usage:␣%s␣[param=value]␣[-s]␣[-c]␣[-h]␣[-v]␣[-p]␣[-P]\n", argv[0]);
    return −2;
  }
}
```
This code is used in section 2.

6. ⟨ Display the input matrix 6 ⟩ ≡
```
for (k = 0; k < m; k++) { for (l = 0; l < n; l++) printf ("%␣4ld", ~? d − *(mtx + k * n + l) :
      *(mtx + k * n + l) ) ;
  printf("\n"); }
```
This code is used in section 2.

7. We obtain a crude but useful estimate of the computation time by counting mem units, as explained in the MILES_SPAN program.

#define *o* *mems*++
#define *oo* *mems* += 2
#define *ooo* *mems* += 3

8. Algorithmic overview. The assignment problem is the classical problem of weighted bipartite matching: to choose a maximum-weight set of disjoint edges in a bipartite graph. We will consider only the case of complete bipartite graphs, when the weights are specified by an $m \times n$ matrix.

An algorithm is most easily developed if we begin with the assumption that the matrix is square (i.e., that $m = n$), and if we change from maximization to minimization. Then the assignment problem is the task of finding a permutation $\pi[0] \ldots \pi[n-1]$ of $\{0, \ldots, n-1\}$ such that $\sum_{k=0}^{n-1} a_{k\pi[k]}$ is minimized, where $A = (a_{kl})$ is a given matrix of numbers a_{kl} for $0 \le k, l < n$. The algorithm below works for arbitrary real numbers a_{kl}, but we will assume in our implementation that the matrix entries are integers.

One way to approach the assignment problem is to make three simple observations: (a) Adding a constant to any row of the matrix does not change the solution $\pi[0] \ldots \pi[n-1]$. (b) Adding a constant to any column of the matrix does not change the solution. (c) If $a_{kl} \ge 0$ for all k and l, and if $\pi[0] \ldots \pi[n-1]$ is a permutation with the property that $a_{k\pi[k]} = 0$ for all k, then $\pi[0] \ldots \pi[n-1]$ solves the assignment problem.

The remarkable fact is that these three observations actually suffice. In other words, there is always a sequence of constants $(\sigma_0, \ldots, \sigma_{n-1})$ and $(\tau_0, \ldots, \tau_{n-1})$ and a permutation $\pi[0] \ldots \pi[n-1]$ such that

$$
\begin{aligned}
&a_{kl} - \sigma_k + \tau_l \ge 0, && \text{for } 0 \le k < n \text{ and } 0 \le l < n;\\
&a_{k\pi[k]} - \sigma_k + \tau_{\pi[k]} = 0, && \text{for } 0 \le k < n.
\end{aligned}
$$

9. To prove the remarkable fact just stated, we start by reviewing the theory of *unweighted* bipartite matching. Any $m \times n$ matrix $A = (a_{kl})$ defines a bipartite graph on the vertices $(r_0, \ldots, r_{m-1})$ and $(c_0, \ldots, c_{n-1})$ if we say that $r_k \text{ --- } c_l$ whenever $a_{kl} = 0$; in other words, the edges of the bipartite graph are the zeroes of the matrix. Two zeroes of A are called *independent* if they appear in different rows and columns; this means that the corresponding edges have no vertices in common. A set of mutually independent zeroes of the matrix therefore corresponds to a set of mutually disjoint edges, also called a *matching* between rows and columns.

The Hungarian mathematicians Egerváry and König proved [*Matematikai és Fizikai Lapok* **38** (1931), 16–28, 116–119] that the maximum number of independent zeroes in a matrix is equal to the minimum number of rows and/or columns that are needed to "cover" every zero. In other words, if we can find p independent zeroes but not $p + 1$, then there is a way to choose p lines in such a way that every zero of the matrix is included in at least one of the chosen lines, where a "line" is either a row or a column.

Their proof was constructive, and it leads to a useful computer algorithm. Given a set of p independent zeroes of a matrix, let us write $r_k \textbf{ --- } c_l$ or $c_l \textbf{ --- } r_k$ and say that r_k is matched with c_l if a_{kl} is one of these p special zeroes, while we continue to write $r_k \text{ --- } c_l$ or $c_l \text{ --- } r_k$ if a_{kl} is one of the nonspecial zeroes. A given set of p special zeroes defines a choice of p lines in the following way: Column c is chosen if and only if it is reachable by a path of the form

$$r^{(0)} \text{ --- } c^{(1)} \textbf{ --- } r^{(1)} \text{ --- } c^{(2)} \textbf{ --- } \cdots \text{ --- } c^{(q)} \textbf{ --- } r^{(q)}\,, \tag{$*$}$$

where $r^{(0)}$ is unmatched, $q \geq 1$, and $c = c^{(q)}$. Row r is chosen if and only if it is matched with a column that is not chosen. Thus exactly p lines are chosen. We can now prove that the chosen lines cover all the zeroes, unless there is a way to find $p + 1$ independent zeroes.

For if $c \textbf{ --- } r$, either c or r has been chosen. And if $c \text{ --- } r$, one of the following cases must arise. (1) If r and c are both unmatched, we can increase p by matching them to each other. (2) If r is unmatched and $c \textbf{ --- } r'$, then c has been chosen, so the zero has been covered. (3) If r is matched to $c' \neq c$, then either r has been chosen or c' has been chosen. In the latter case, there is a path of the form

$$r^{(0)} \text{ --- } c^{(1)} \textbf{ --- } r^{(1)} \text{ --- } c^{(2)} \textbf{ --- } \cdots \textbf{ --- } r^{(q-1)} \text{ --- } c' \textbf{ --- } r \text{ --- } c\,,$$

where $r^{(0)}$ is unmatched and $q \geq 1$. If c is matched, it has therefore been chosen; otherwise we can increase p by redefining the matching to include

$$r^{(0)} \textbf{ --- } c^{(1)} \text{ --- } r^{(1)} \textbf{ --- } c^{(2)} \text{ --- } \cdots \text{ --- } r^{(q-1)} \textbf{ --- } c' \text{ --- } r \textbf{ --- } c\,.$$

10. Now suppose A is a *nonnegative* matrix, of size $n \times n$. Cover the zeroes of A with a minimum number of lines, p, using the algorithm of Egerváry and König. If $p < n$, some elements are still uncovered, so those elements are positive. Suppose the minimum uncovered value is $\delta > 0$. Then we can subtract δ from each unchosen row and add δ to each chosen column. The net effect is to subtract δ from all uncovered elements and to add δ to all doubly covered elements, while leaving all singly covered elements unchanged. This transformation causes a new zero to appear, while preserving p independent zeroes of the previous matrix (since they were each covered only once). If we repeat the Egerváry-König construction with the same p independent zeroes, we find that either p is no longer maximum or at least one more column has been chosen. (The new zero r — c occurs in a row r that was either unmatched or matched to a previously chosen column, because row r was not chosen.) Therefore if we repeat the process, we must eventually be able to increase p until finally $p = n$. This will solve the assignment problem, proving the remarkable claim made earlier.

11. If the given matrix A has m rows and $n > m$ columns, we can extend it artificially until it is square, by setting $a_{kl} = 0$ for all $m \le k < n$ and $0 \le l < n$. The construction above will then apply. But we need not waste time making such an extension, because it suffices to run the algorithm on the original $m \times n$ matrix until m independent zeroes have been found. The reason is that the set of matched vertices always grows monotonically in the Egerváry-König construction: If a column is matched at some stage, it will remain matched from that time on, although it might well change partners. The $n - m$ dummy rows at the bottom of A are always chosen to be part of the covering; so the dummy entries become nonzero only in the columns that are part of some covering. Such columns are part of some matching, so they are part of the final matching. Therefore at most m columns of the dummy entries become nonzero during the procedure. We can always find $n - m$ independent zeroes in the $n - m$ dummy rows of the matrix, so we need not deal with the dummy elements explicitly.

12. It has been convenient to describe the algorithm by saying that we add and subtract constants to and from the columns and rows of A. But all those additions and subtractions can take a lot of time. So we will merely pretend to make the adjustments that the method calls for; we will represent them implicitly by two vectors $(\sigma_0, \ldots, \sigma_{m-1})$ and $(\tau_0, \ldots, \tau_{n-1})$. Then the current value of each matrix entry will be $a_{kl} - \sigma_k + \tau_l$, instead of a_{kl}. The "zeroes" will be positions such that $a_{kl} = \sigma_k - \tau_l$.

Initially we will set $\tau_l = 0$ for $0 \le l < n$ and $\sigma_k = \min\{a_{k0}, \ldots, a_{k(n-1)}\}$ for $0 \le k < m$. If $m = n$ we can also make sure that there's a zero in every column by subtracting $\min\{a_{0l}, \ldots, a_{(n-1)l}\}$ from a_{kl} for all k and l. (This initial adjustment can conveniently be made to the original matrix entries, instead of indirectly via the τ's.) Users can discover if such a transformation is worthwhile by trying the program both with and without the `-h` option.

We have been saying a lot of things and proving a bunch of theorems, without writing any code. Let's get back into programming mode by writing the routine that is called into action when the `-h` option has been specified:

```
#define aa(k,l) *(mtx + k * n + l)   /* a macro to access the matrix elements */
⟨ Subtract column minima in order to start with lots of zeroes 12 ⟩ ≡
  {
    for (l = 0; l < n; l++) {
      o, s = aa(0,l);   /* the o macro counts one mem */
      for (k = 1; k < n; k++)
        if (o, aa(k,l) < s) s = aa(k,l);
      if (s ≠ 0)
        for (k = 0; k < n; k++) oo, aa(k,l) -= s;   /* oo counts two mems */
    }
    if (verbose) printf("␣The␣heuristic␣has␣cost␣%ld␣mems.\n", mems);
  }
```

This code is used in section 24.

13. ⟨ Local variables 4 ⟩ +≡

```
  register long k;   /* the current row of interest */
  register long l;   /* the current column of interest */
  register long j;   /* another interesting column */
  register long s;   /* the current matrix element of interest */
```

14. Algorithmic details. The algorithm sketched above is quite simple, except that we did not discuss how to determine the chosen columns $c^{(q)}$ that are reachable by paths of the stated form $(*)$. It is easy to find all such columns by constructing an unordered forest whose nodes are rows, beginning with all unmatched rows $r^{(0)}$ and adding a row r for which c **—** r when c is adjacent to a row already in the forest.

Our data structure, which is based on suggestions of Papadimitriou and Steiglitz [*Combinatorial Optimization* (Prentice-Hall, 1982), §11.1], will use several arrays. If row r is matched with column c, we will have *col_mate*$[r] = c$ and *row_mate*$[c] = r$; if row r is unmatched, *col_mate*$[r]$ will be -1, and if column c is unmatched, *row_mate*$[c]$ will be -1. If column c has a mate and is also reachable in a path of the form $(*)$, we will have *parent_row*$[c] = r'$ for some r' in the forest. Otherwise column c is not chosen, and we will have *parent_row*$[c] = -1$. The rows in the current forest will be called *unchosen_row*$[0]$ through *unchosen_row*$[t-1]$, where t is the current total number of nodes.

The amount σ_k subtracted from row k is called *row_dec*$[k]$; the amount τ_l added to column l is called *col_inc*$[l]$. To compute the minimum uncovered element efficiently, we maintain a quantity called *slack*$[l]$, which represents the minimum uncovered element in each column. More precisely, if column l is not chosen, *slack*$[l]$ is the minimum of $a_{kl} - \sigma_k + \tau_l$ for $k \in \{\mathit{unchosen\_row}[0], \ldots, \mathit{unchosen\_row}[q-1]\}$, where $q \le t$ is the number of rows in the forest that we have explored so far. We also remember *slack_row*$[l]$, the number of a row where the stated minimum occurs.

Column l is chosen if and only if *parent_row*$[l] \ge 0$. We will arrange things so that we also have *slack*$[l] = 0$ in every chosen column.

```
⟨ Local variables 4 ⟩ +≡
  long *col_mate;   /* the column matching a given row, or −1 */
  long *row_mate;   /* the row matching a given column, or −1 */
  long *parent_row;   /* ancestor of a given column's mate, or −1 */
  long *unchosen_row;   /* node in the forest */
  long t;   /* total number of nodes in the forest */
  long q;   /* total number of explored nodes in the forest */
  long *row_dec;   /* σ_k, the amount subtracted from a given row */
  long *col_inc;   /* τ_l, the amount added to a given column */
  long *slack;   /* minimum uncovered entry seen in a given column */
  long *slack_row;   /* where the slack in a given column can be found */
  long unmatched;   /* this many rows have yet to be matched */
```

15.
```
⟨ Allocate the intermediate data structures 15 ⟩ ≡
  col_mate = gb_typed_alloc(m, long, working_storage);
  row_mate = gb_typed_alloc(n, long, working_storage);
  parent_row = gb_typed_alloc(n, long, working_storage);
  unchosen_row = gb_typed_alloc(m, long, working_storage);
  row_dec = gb_typed_alloc(m, long, working_storage);
  col_inc = gb_typed_alloc(n, long, working_storage);
  slack = gb_typed_alloc(n, long, working_storage);
  slack_row = gb_typed_alloc(n, long, working_storage);
  if (gb_trouble_code) {
    fprintf(stderr, "Sorry,␣out␣of␣memory!\n");
    return −3;
  }
```

This code is used in section 24.

16. The algorithm operates in stages, where each stage terminates when we are able to increase the number of matched elements.

The first stage is different from the others; it simply goes through the matrix and looks for zeroes, matching as many rows and columns as it can. This stage also initializes table entries that will be useful in later stages.

```
#define INF #7fffffff  /* infinity (or darn near) */
```

⟨ Do the initial stage 16 ⟩ ≡

```
t = 0;  /* the forest starts out empty */
for (l = 0; l < n; l++) {
  o, row_mate[l] = −1;
  o, parent_row[l] = −1;
  o, col_inc[l] = 0;
  o, slack[l] = INF;
}
for (k = 0; k < m; k++) {
  o, s = aa(k, 0);  /* get ready to calculate the minimum entry of row k */
  for (l = 1; l < n; l++)
    if (o, aa(k, l) < s) s = aa(k, l);
  o, row_dec[k] = s;
  for (l = 0; l < n; l++)
    if ((o, s ≡ aa(k, l)) ∧ (o, row_mate[l] < 0)) {
      o, col_mate[k] = l;
      o, row_mate[l] = k;
      if (verbose > 1) printf("␣matching␣col␣%ld==row␣%ld\n", l, k);
      goto row_done;
    }
  o, col_mate[k] = −1;
  if (verbose > 1) printf("␣␣node␣%ld:␣unmatched␣row␣%ld\n", t, k);
  o, unchosen_row[t++] = k;
row_done: ;
}
```

This code is used in section 18.

17. If a subsequent stage has not succeeded in matching every row, we prepare for a new stage by reinitializing the forest as follows.

⟨ Get ready for another stage 17 ⟩ ≡

```
t = 0;
for (l = 0; l < n; l++) {
  o, parent_row[l] = −1;
  o, slack[l] = INF;
}
for (k = 0; k < m; k++)
  if (o, col_mate[k] < 0) {
    if (verbose > 1) printf("␣␣node␣%ld:␣unmatched␣row␣%ld\n", t, k);
    o, unchosen_row[t++] = k;
  }
```

This code is used in section 18.

18. Here, then, is the algorithm's overall control structure. There are at most m stages, and each stage does $O(mn)$ operations, so the total running time is $O(m^2 n)$.

```
⟨ Do the Hungarian algorithm 18 ⟩ ≡
  ⟨ Do the initial stage 16 ⟩;
  if (t ≡ 0) goto done;
  unmatched = t;
  while (1) {
    if (verbose) printf("␣After␣%ld␣mems␣I've␣matched␣%ld␣rows.\n", mems, m − t);
    q = 0;
    while (1) {
      while (q < t) {
        ⟨ Explore node q of the forest; if the matching can be increased, goto breakthru 19 ⟩;
        q++;
      }
      ⟨ Introduce a new zero into the matrix by modifying row_dec and col_inc; if the matching can be
          increased, goto breakthru 21 ⟩;
    }
  breakthru: ⟨ Update the matching by pairing row k with column l 20 ⟩;
    if (−−unmatched ≡ 0) goto done;
    ⟨ Get ready for another stage 17 ⟩;
  }
done: ⟨ Doublecheck the solution 23 ⟩;
```

This code is used in section 24.

19. ⟨ Explore node q of the forest; if the matching can be increased, **goto** *breakthru* 19 ⟩ ≡

```
  {
    o, k = unchosen_row[q];
    o, s = row_dec[k];
    for (l = 0; l < n; l++)
      if (o, slack[l]) { register long del;
        oo, del = aa(k, l) − s + col_inc[l];
        if (del < slack[l]) {
          if (del ≡ 0) {      /* we found a new zero */
            if (o, row_mate[l] < 0) goto breakthru;
            o, slack[l] = 0;      /* this column will now be chosen */
            o, parent_row[l] = k;
            if (verbose > 1) printf("␣␣node␣%ld:␣row␣%ld==col␣%ld−−row␣%ld\n", t, row_mate[l], l, k);
            oo, unchosen_row[t++] = row_mate[l];
          } else {
            o, slack[l] = del;
            o, slack_row[l] = k;
          }
        }
      }
  }
```

This code is used in section 18.

20. At this point, column l is unmatched, and row k is in the forest. By following parent links in the forest, we can rematch rows and columns so that a previously unmatched row $r^{(0)}$ gets a mate.

⟨ Update the matching by pairing row k with column l 20 ⟩ ≡

```
if (verbose) printf("␣Breakthrough␣at␣node␣%ld␣of␣%ld!\n", q, t);
while (1) {
  o, j = col_mate[k];
  o, col_mate[k] = l;
  o, row_mate[l] = k;
  if (verbose > 1) printf("␣rematching␣col␣%ld==row␣%ld\n", l, k);
  if (j < 0) break;
  o, k = parent_row[j];
  l = j;
}
```

This code is used in section 18.

21. If we get to this point, we have explored the entire forest; none of the unchosen rows has led to a breakthrough. An unchosen column with smallest *slack* will allow us to make further progress.

⟨ Introduce a new zero into the matrix by modifying *row_dec* and *col_inc*; if the matching can be increased, **goto** *breakthru* 21 ⟩ ≡

```
s = INF;
for (l = 0; l < n; l++)
  if (o, slack[l] ∧ slack[l] < s) s = slack[l];
for (q = 0; q < t; q++) ooo, row_dec[unchosen_row[q]] += s;
for (l = 0; l < n; l++)
  if (o, slack[l]) {      /* column l is not chosen */
    o, slack[l] −= s;
    if (slack[l] ≡ 0)
      ⟨ Look at a new zero, and goto breakthru with col_inc up to date if there's a breakthrough 22 ⟩;
  } else oo, col_inc[l] += s;
```

This code is used in section 18.

22. There might be several columns tied for smallest slack. If any of them leads to a breakthrough, we are very happy; but we must finish the loop on *l* before going to *breakthru*, because the *col_inc* variables need to be maintained for the next stage.

Within column *l*, there might be several rows that produce the same slack; we have remembered only one of them, *slack_row*[*l*]. Fortunately, one is sufficient for our purposes. Either we have a breakthrough or we choose column *l*, regardless of which row or rows led us to consider that column.

⟨ Look at a new zero, and **goto** *breakthru* with *col_inc* up to date if there's a breakthrough 22 ⟩ ≡

```
{
  o, k = slack_row[l];
  if (verbose > 1)
    printf("␣Decreasing␣uncovered␣elements␣by␣%ld␣produces␣zero␣at␣[%ld,%ld]\n", s, k, l);
  if (o, row_mate[l] < 0) {
    for (j = l + 1; j < n; j++)
      if (o, slack[j] ≡ 0) oo, col_inc[j] += s;
    goto breakthru;
  } else {      /* not a breakthrough, but the forest continues to grow */
    o, parent_row[l] = k;
    if (verbose > 1) printf("␣␣node␣%ld:␣row␣%ld==col␣%ld--row␣%ld\n", t, row_mate[l], l, k);
    oo, unchosen_row[t++] = row_mate[l];
  }
}
```

This code is used in section 21.

23. The code in the present section is redundant, unless cosmic radiation has caused the hardware to malfunction. But there is some reassurance whenever we find that mathematics still appears to be consistent, so the author could not resist writing these few unnecessary lines, which verify that the assignment problem has indeed been solved optimally. (We don't count the mems.)

⟨ Doublecheck the solution 23 ⟩ ≡

```
for (k = 0; k < m; k++)
  for (l = 0; l < n; l++)
    if (aa(k, l) < row_dec[k] − col_inc[l]) {
      fprintf(stderr, "Oops,␣I␣made␣a␣mistake!\n");
      return −6;      /* can't happen */
    }
for (k = 0; k < m; k++) {
  l = col_mate[k];
  if (l < 0 ∨ aa(k, l) ≠ row_dec[k] − col_inc[l]) {
    fprintf(stderr, "Oops,␣I␣blew␣it!\n");
    return −66;      /* can't happen */
  }
}
k = 0;
for (l = 0; l < n; l++)
  if (col_inc[l]) k++;
if (k > m) {
  fprintf(stderr, "Oops,␣I␣adjusted␣too␣many␣columns!\n");
  return −666;      /* can't happen */
}
```

This code is used in section 18.

24. Interfacing. A few nitty-gritty details still need to be handled: Our algorithm is not symmetric between rows and columns, and it works only for $m \le n$; so we will transpose the matrix when $m > n$. Furthermore, our algorithm minimizes, but we actually want it to maximize (except when $\sim$ is nonzero).

Hence, we want to make the following transformations to the data before processing it with the algorithm developed above.

```
⟨ Solve the assignment problem 24 ⟩ ≡
  if (m > n) ⟨ Transpose the matrix 25 ⟩
  else transposed = 0;
  ⟨ Allocate the intermediate data structures 15 ⟩;
  if ( ~≡ 0 )
  for (k = 0; k < m; k++)
    for (l = 0; l < n; l++) aa(k, l) = d − aa(k, l);
  if (heur) ⟨ Subtract column minima in order to start with lots of zeroes 12 ⟩;
  ⟨ Do the Hungarian algorithm 18 ⟩;
```

This code is used in section 2.

25. ⟨ Transpose the matrix 25 ⟩ ≡

```
  {
    if (verbose > 1) printf("Temporarily␣transposing␣rows␣and␣columns...\n");
    tmtx = gb_typed_alloc(m ∗ n, long, working_storage);
    if (tmtx ≡ Λ) {
      fprintf(stderr, "Sorry,␣out␣of␣memory!\n"); return −4;
    }
    for (k = 0; k < m; k++)
      for (l = 0; l < n; l++) ∗(tmtx + l ∗ m + k) = ∗(mtx + k ∗ n + l);
    m = n; n = k;      /∗ k holds the former value of m ∗/
    mtx = tmtx;
    transposed = 1;
  }
```

This code is used in section 24.

26. ⟨ Local variables 4 ⟩ +≡

```
  long ∗tmtx;      /∗ the transpose of mtx ∗/
  long transposed;      /∗ has the data been transposed? ∗/
```

27. ⟨ Display the solution 27 ⟩ ≡

```
  {
    printf("The␣following␣entries␣produce␣an␣optimum␣assignment:\n");
    for (k = 0; k < m; k++) printf("␣[%ld,%ld]\n",
          transposed ? col_mate[k] : k,
          transposed ? k : col_mate[k]);
  }
```

This code is used in section 2.

28. Encapsulated PostScript. A special output file called `lisa.eps` is written if the user has selected the `-P` option. This file contains a sequence of PostScript commands that can be used to generate an illustration within many kinds of documents. For example, if TEX is being used with the `dvips` output driver from Radical Eye Software and with the associated `epsf.tex` macros, one can say

`\epsfxsize=10cm \epsfbox{lisa.eps}`

within a TEX document and the illustration will be typeset in a box that is 10 centimeters wide.

The conventions of PostScript allow the illustration to be scaled to any size. Best results are probably obtained if each pixel is at least one millimeter wide (about 1/25 inch) when printed.

The illustration is formed by first "painting" the input data as a rectangle of pixels, with up to 256 shades of gray. Then the solution pixels are framed in black, with a white trim just inside the black edges to help make the frame visible in already-dark places. The frames are created by painting over the original image; the center of each solution pixel retains its original color.

Encapsulated PostScript files have a simple format that is recognized by many software packages and printing devices. We use a subset of PostScript that should be easy to convert to other languages if necessary.

```
⟨ Output the input matrix in PostScript format 28 ⟩ ≡
  {
    eps_file = fopen("lisa.eps", "w");
    if (¬eps_file) {
      fprintf(stderr, "Sorry,␣I␣can't␣open␣the␣file␣`lisa.eps'!\n");
      PostScript = 0;
    } else {
      fprintf(eps_file, "%%!PS-Adobe-3.0␣EPSF-3.0\n");    /* 1.0 and 2.0 also OK */
      fprintf(eps_file, "%%%%BoundingBox:␣-1␣-1␣%ld␣%ld\n", n + 1, m + 1);
      fprintf(eps_file, "/buffer␣%ld␣string␣def\n", n);
      fprintf(eps_file, "%ld␣%ld␣8␣[%ld␣0␣0␣-%ld␣0␣%ld]\n", n, m, n, m, m);
      fprintf(eps_file, "{currentfile␣buffer␣readhexstring␣pop}␣bind\n");
      fprintf(eps_file, "gsave␣%ld␣%ld␣scale␣image\n", n, m);
      for (k = 0; k < m; k++) ⟨ Output row k as a hexadecimal string 30 ⟩;
      fprintf(eps_file, "grestore\n");
    }
  }
```

This code is used in section 2.

29. ⟨ Global variables 3 ⟩ +≡

```
  FILE *eps_file;    /* file for encapsulated PostScript output */
```

30. This program need not produce machine-independent output, so we can safely use floating-point arithmetic here. At most 64 characters (32 pixel-bytes) are output on each line.

```
⟨ Output row k as a hexadecimal string 30 ⟩ ≡
  { register float conv = 255.0/(float) d;
  register long x; for (l = 0; l < n; l++) { x = (long)( conv * (float)( ~? d − aa(k, l) : aa(k, l) ) ) ;
  fprintf(eps_file, "%02lx", x > 255 ? 255L : x);
  if ((l & #1f) ≡ #1f) fprintf(eps_file, "\n");
  }
  if (n & #1f) fprintf(eps_file, "\n");
  }
```

This code is used in section 28.

31. ⟨ Output the solution in PostScript format 31 ⟩ ≡

```
{
  fprintf(eps_file, "/bx␣{moveto␣0␣1␣rlineto␣1␣0␣rlineto␣0␣-1␣rlineto␣closepath\n");
  fprintf(eps_file, "␣gsave␣.3␣setlinewidth␣1␣setgray␣clip␣stroke");
  fprintf(eps_file, "␣grestore␣stroke}␣bind␣def\n");
  fprintf(eps_file, "␣.1␣setlinewidth\n");
  for (k = 0; k < m; k++) fprintf(eps_file, "␣%ld␣%ld␣bx\n",
        transposed ? k : col_mate[k],
        transposed ? n - 1 - col_mate[k] : m - 1 - k);
  fclose(eps_file);
}
```

This code is used in section 2.

32. Index. As usual, we close with a list of identifier definitions and uses.

October 28, 2005 at 09:05

ASSIGN_LISA

© 1993 Stanford University

This file may be freely copied and distributed, provided that no changes whatsoever are made. All users are asked to help keep the Stanford GraphBase files consistent and "uncorrupted," identical everywhere in the world. Changes are permissible only if the modified file is given a new name, different from the names of existing files in the Stanford GraphBase, and only if the modified file is clearly identified as not being part of that GraphBase. (The CWEB system has a "change file" facility by which users can easily make minor alterations without modifying the master source files in any way. Everybody is supposed to use change files instead of changing the files.) The author has tried his best to produce correct and useful programs, in order to help promote computer science research, but no warranty of any kind should be assumed.

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.