Important: Before reading ECON_ORDER, please read or at least skim the program for GB_ECON.

1. Near-triangular ordering. This demonstration program takes a matrix of data constructed by the GB_ECON module and permutes the economic sectors so that the first sectors of the ordering tend to be producers of primary materials for other industries, while the last sectors tend to be final-product industries that deliver their output mostly to end users.

More precisely, suppose the rows of the matrix represent the outputs of a sector and the columns represent the inputs. This program attempts to find a permutation of rows and columns that minimizes the sum of the elements below the main diagonal. (If this sum were zero, the matrix would be upper triangular; each supplier of a sector would precede it in the ordering, while each customer of that sector would follow it.)

The general problem of finding a minimizing permutation is NP-complete; it includes, as a very special case, the FEEDBACK ARC SET problem discussed in Karp's classic paper [*Complexity of Computer Computations* (Plenum Press, 1972), 85–103]. But sophisticated "branch and cut" methods have been developed that work well in practice on problems of reasonable size. Here we use a simple heuristic downhill method to find a permutation that is locally optimum, in the sense that the below-diagonal sum does not decrease if any individual sector is moved to another position while preserving the relative order of the other sectors. We start with a random permutation and repeatedly improve it, choosing the improvement that gives the least positive gain at each step. A primary motive for the present implementation was to get further experience with this method of cautious descent, which was proposed by A. M. Gleason in *AMS Proceedings of Symposia in Applied Mathematics* **10** (1958), 175–178. (See the comments following the program below.)

2. As explained in GB_ECON, the subroutine call $econ(n, 2, 0, s)$ constructs a graph whose $n \le 79$ vertices represent sectors of the U.S. economy and whose arcs $u \to v$ are assigned numbers corresponding to the flow of products from sector u to sector v. When $n < 79$, the n sectors are obtained from a basic set of 79 sectors by combining related commodities. If $s = 0$, the combination is done in a way that tends to equalize the row sums, while if $s > 0$, the combination is done by choosing a random subtree of a given 79-leaf tree; the "randomness" is fully determined by the value of s.

This program uses two random number seeds, one for *econ* and one for choosing the random initial permutation. The former is called s and the latter is called t. A further parameter, r, governs the number of repetitions to be made; the machine will try r different starting permutations on the same matrix. When $r > 1$, new solutions are displayed only when they improve on the previous best.

By default, $n = 79$, $r = 1$, and $s = t = 0$. The user can change these default parameters by specifying options on the command line, at least in a UNIX implementation, thereby obtaining a variety of special effects. The relevant command-line options are `-n`⟨number⟩, `-r`⟨number⟩, `-s`⟨number⟩, and/or `-t`⟨number⟩. Additional options `-v` (verbose), `-V` (extreme verbosity), and `-g` (greedy or steepest descent instead of cautious descent) are also provided.

Here is the overall layout of this C program:

```
#include "gb_graph.h"   /* the GraphBase data structures */
#include "gb_flip.h"   /* the random number generator */
#include "gb_econ.h"   /* the econ 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 */
  { unsigned long n = 79;   /* the desired number of sectors */
    long s = 0;   /* random seed for econ */
    long t = 0;   /* random seed for initial permutation */
    unsigned long r = 1;   /* the number of repetitions */
    long greedy = 0;   /* should we use steepest descent? */
    register long j, k;   /* all-purpose indices */
    ⟨ Scan the command-line options 4 ⟩;
    g = econ(n, 2L, 0L, s);
    if (g ≡ Λ) {
      fprintf(stderr, "Sorry,␣can't␣create␣the␣matrix!␣(error␣code␣%ld)\n", panic_code);
      return −1;
    }
    printf("Ordering␣the␣sectors␣of␣%s,␣using␣seed␣%ld:\n", g→id, t);
    printf("␣(%s␣descent␣method)\n", greedy ? "Steepest" : "Cautious");
    ⟨ Put the graph data into matrix form 5 ⟩;
    ⟨ Print an obvious lower bound 6 ⟩;
    gb_init_rand(t);
    while (r−−) ⟨ Find a locally optimum permutation and report the below-diagonal sum 8 ⟩;
    return 0;   /* normal exit */
  }
```

3. Besides the matrix M of input/output coefficients, we will find it convenient to use the matrix Δ, where $\Delta_{jk} = M_{jk} - M_{kj}$.

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

⟨ Global variables 3 ⟩ ≡

```
Graph *g;    /* the graph we will work on */
long mat[79][79];    /* the corresponding matrix */
long del[79][79];    /* skew-symmetric differences */
long best_score = INF;    /* the smallest below-diagonal sum we've seen so far */
```

See also sections 7 and 12.

This code is used in section 2.

4. ⟨ Scan the command-line options 4 ⟩ ≡

```
while (--argc) {
  if (sscanf(argv[argc], "-n%lu", &n) ≡ 1) ;
  else if (sscanf(argv[argc], "-r%lu", &r) ≡ 1) ;
  else if (sscanf(argv[argc], "-s%ld", &s) ≡ 1) ;
  else if (sscanf(argv[argc], "-t%ld", &t) ≡ 1) ;
  else if (strcmp(argv[argc], "-v") ≡ 0) verbose = 1;
  else if (strcmp(argv[argc], "-V") ≡ 0) verbose = 2;
  else if (strcmp(argv[argc], "-g") ≡ 0) greedy = 1;
  else {
    fprintf(stderr, "Usage:␣%s␣[-nN][-rN][-sN][-tN][-g][-v][-V]\n", argv[0]);
    return -2;
  }
}
```

This code is used in section 2.

5. The optimum permutation is a function only of the Δ matrix, because we can subtract any constant from both M_{jk} and M_{kj} without changing the basic problem.

⟨ Put the graph data into matrix form 5 ⟩ ≡

```
{ register Vertex *v;
  register Arc *a;

  n = g→n;
  for (v = g→vertices; v < g→vertices + n; v++)
    for (a = v→arcs; a; a = a→next) mat[v - g→vertices][a→tip - g→vertices] = a→flow;
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++) del[j][k] = mat[j][k] - mat[k][j];
}
```

This code is used in section 2.

6. Nontrivial lower bounds that can be made strong enough to find provably optimum solutions to the ordering problem can be based on linear programming, as shown for example by Grötschel, Jünger, and Reinelt [*Operations Research* **32** (1984), 1195–1220]. The basic idea is to formulate the problem as the task of minimizing $\sum M_{jk}x_{jk}$ for integer variables $x_{jk} \geq 0$, subject to the conditions $x_{jk} + x_{kj} = 1$ and $x_{ik} \leq x_{ij} + x_{jk}$ for all triples (i, j, k) of distinct subscripts; these conditions are necessary and sufficient. Relaxing the integrality constraints gives a lower bound, and we can also add additional inequalities such as $x_{14} + x_{25} + x_{36} + x_{42} + x_{43} + x_{51} + x_{53} + x_{61} + x_{62} \leq 7$. The interesting story of inequalities like this has been surveyed by P. C. Fishburn [*Mathematical Social Sciences* **23** (1992), 67–80].

However, our goal is more modest—we just want to study two of the simplest heuristics. So we will be happy with a trivial bound based only on the constraints $x_{jk} + x_{kj} = 1$.

⟨ Print an obvious lower bound 6 ⟩ ≡

```
{ register long sum = 0;
  for (j = 1; j < n; j++)
    for (k = 0; k < j; k++)
      if (mat[j][k] ≤ mat[k][j]) sum += mat[j][k];
      else sum += mat[k][j];
  printf("(The␣amount␣of␣feed-forward␣must␣be␣at␣least␣%ld.)\n", sum);
}
```

This code is used in section 2.

7. Descent. At each stage in our search, *mapping* will be the current permutation; in other words, the sector in row and column k will be $g\to vertices + mapping[k]$. The current below-diagonal sum will be the value of *score*. We will not actually have to permute anything inside of *mat*.

```
#define sec_name(k) (g→vertices + mapping[k])→name

⟨ Global variables 3 ⟩ +≡
  long mapping[79];   /* current permutation */
  long score;   /* current sum of elements above main diagonal */
  long steps;   /* the number of iterations so far */
```

8. ⟨ Find a locally optimum permutation and report the below-diagonal sum 8 ⟩ ≡

```
  {
    ⟨ Initialize mapping to a random permutation 9 ⟩;
    while (1) {
      ⟨ Figure out the next move to make; break if at local optimum 10 ⟩;
      if (verbose) printf("%8ld␣after␣step␣%ld\n", score, steps);
      else if (steps % 1000 ≡ 0 ∧ steps > 0) {
        putchar('.');
        fflush(stdout);   /* progress report */
      }
      ⟨ Take the next step 13 ⟩;
    }
    printf("\n%s␣is␣%ld,␣found␣after␣%ld␣step%s.\n",
        best_score ≡ INF ? "Local␣minimum␣feed-forward" : "Another␣local␣minimum",
        score, steps, steps ≡ 1 ? "" : "s");
    if (verbose ∨ score < best_score) {
      printf("The␣corresponding␣economic␣order␣is:\n");
      for (k = 0; k < n; k++) printf("␣%s\n", sec_name(k));
      if (score < best_score) best_score = score;
    }
  }
```

This code is used in section 2.

9. ⟨ Initialize *mapping* to a random permutation 9 ⟩ ≡

```
  steps = score = 0;
  for (k = 0; k < n; k++) {
    j = gb_unif_rand(k + 1);
    mapping[k] = mapping[j];
    mapping[j] = k;
  }
  for (j = 1; j < n; j++)
    for (k = 0; k < j; k++) score += mat[mapping[j]][mapping[k]];
  if (verbose > 1) {
    printf("\nInitial␣permutation:\n");
    for (k = 0; k < n; k++) printf("␣%s\n", sec_name(k));
  }
```

This code is used in section 8.

10. If we move, say, *mapping*[5] to *mapping*[3] and shift the previous entries *mapping*[3] and *mapping*[4] right one, the score decreases by

$$del[mapping[5]][mapping[3]] + del[mapping[5]][mapping[4]]\,.$$

Similarly, if we move *mapping*[5] to *mapping*[7] and shift the previous entries *mapping*[6] and *mapping*[7] left one, the score decreases by

$$del[mapping[6]][mapping[5]] + del[mapping[7]][mapping[5]]\,.$$

The number of possible moves is $(n-1)^2$. Our job is to find the one that makes the score decrease, but by as little as possible (or, if $greedy \neq 0$, to make the score decrease as much as possible).

```
⟨ Figure out the next move to make; break if at local optimum 10 ⟩ ≡
  best_d = greedy ? 0 : INF;
  best_k = −1;
  for (k = 0; k < n; k++) { register long d = 0;
    for (j = k − 1; j ≥ 0; j−−) {
      d += del[mapping[k]][mapping[j]];
      ⟨ Record the move from k to j, if d is better than best_d 11 ⟩;
    }
    d = 0;
    for (j = k + 1; j < n; j++) {
      d += del[mapping[j]][mapping[k]];
      ⟨ Record the move from k to j, if d is better than best_d 11 ⟩;
    }
  }
  if (best_k < 0) break;
```

This code is used in section 8.

11.
```
⟨ Record the move from k to j, if d is better than best_d 11 ⟩ ≡
  if (d > 0 ∧ (greedy ? d > best_d : d < best_d)) {
    best_k = k;
    best_j = j;
    best_d = d;
  }
```

This code is used in section 10.

12.
```
⟨ Global variables 3 ⟩ +≡
  long best_d;   /* best improvement seen so far on this step */
  long best_k, best_j;   /* moving best_k to best_j improves by best_d */
```

13. ⟨ Take the next step 13 ⟩ ≡

```
if (verbose > 1)
   printf("Now␣move␣%s␣to␣the␣%s,␣past\n", sec_name(best_k), best_j < best_k ? "left" : "right");
j = best_k;
k = mapping[j];
do {
   if (best_j < best_k) mapping[j] = mapping[j − 1], j−−;
   else mapping[j] = mapping[j + 1], j++;
   if (verbose > 1) printf("␣␣␣␣%s␣(%ld)\n", sec_name(j),
         best_j < best_k ? del[mapping[j + 1]][k] : del[k][mapping[j − 1]]);
} while (j ≠ best_j);
mapping[j] = k;
score −= best_d;
steps++;
```

This code is used in section 8.

14. How well does cautious descent work? In this application, it is definitely too cautious. For example, after lots of computation with the default settings, it comes up with a pretty good value (457342), but only after taking 39,418 steps! Then (if $r > 1$) it tries again and stops with 461584 after 47,634 steps. The greedy algorithm with the same starting permutations obtains the local minimum 457408 after only 93 steps, then 460411 after 83 steps. The greedy algorithm tends to find solutions that are a bit inferior, but it is so much faster that it allows us to run many more experiments. After 20 trials with the default settings, it finds a permutation with only 456315 below the diagonal, and after about 250 more it reduces this upper bound to 456295. (Gerhard Reinelt has proved, via branch-and-cut, that 456295 is in fact optimum.)

The method of stratified greed, which is illustrated in the FOOTBALL module, should do better than the ordinary greedy algorithm; and interesting results can be expected when stratified greed is compared also to other methods like simulated annealing and genetic breeding. Comparisons should be made by seeing which method can come up with the best upper bound after calculating for a given number of mems (see MILES_SPAN). The upper bound obtained in any run is a random variable, so several independent trials of each method should be made.

Question: Suppose we divide the vertices into two subsets and prescribe a fixed permutation on each subset. Is it NP-complete to find the optimum way to merge these two permutations—i.e., to find a permutation, extending the given ones, that has the smallest below-diagonal sum?

15. Index. We close with a list that shows where the identifiers of this program are defined and used.

May 19, 2018 at 02:29

ECON_ORDER

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