Finding Embedded Network Rows in Linear Programs I: Extraction Heuristics

by

Robert E. Bixby[1]

and

Robert Fourer[2]

Technical Report 86-18, August 1986.

[1]Mathematical Sciences Department, Rice University, Houston, Texas 77251.

[2]Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60201.

Finding Embedded Network Rows in Linear Programs I: Extraction Heuristics

Robert E. Bixby
Department of Mathematical Sciences
Rice University, Houston, TX 77251

Robert Fourer
Department of Industrial Engineering and Management Sciences
Northwestern University, Evanston, IL 60201

July 1986

Abstract. An embedded network within a linear program is, roughly speaking, a subset of constraints that represent conservation of flow. In this paper, we examine three broad classes of heuristic techniques—row-scanning deletion, column-scanning deletion, and row-scanning addition—for the extraction of large embedded networks. We describe a variety of implementations, and compare their performance on varied test problems. The success of our tests depends, in part, on several preprocessing steps that scale the constraint matrix and that set aside certain rows and columns. Efficiency of the subsequent network extraction is dependent on the implementation, in predictable ways. Effectiveness is harder to explain; the more sophisticated and expensive implementations seem to be more reliable, but much simpler implementations sometimes find equally large networks. The largest networks are obtained by applying a final augmentation phase, which is studied in the second part of this paper.

This research was supported in part by the following:

For Robert E. Bixby: Air Force Office of Scientific Research grant AFOSR82–0004 to Northwestern University; Exxon Corporation grant to Northwestern University; the Alexander von Humboldt Foundation and the Institut für Ökonometrie und Operations Research der Universität Bonn (while on leave from Rice University); and National Science Foundation grant DCR–8416187 to Rice University.

For Robert Fourer: Exxon Corporation grant to Northwestern University; National Science Foundation grant DMS–8217261 to Northwestern University; AT&T Bell Laboratories (while on leave from Northwestern University).

1. Introduction

A linear program contains an embedded network if, roughly speaking, some subset of its constraints can be interpreted as specifying conservation of flow. In this paper and its successor, we evaluate a variety of heuristic algorithms for finding large embedded networks within linear programs.

Embedded networks are of interest for two reasons. The presence of network constraints usually suggests a useful interpretation of a linear program in terms of some network in the underlying application. The presence of many network constraints also often allows a linear program to be solved faster than other LPs of comparable size, provided that the structure of the network constraints is exploited to advantage. Simplex algorithms for embedded-network linear programs have been the subject of particular investigation (Hartman and Lasdon 1972; Chen and Saigal 1977; Gupta 1978; Glover and Klingman 1981; Ahn 1984; McBride 1985).

Network-finding heuristics are primarily of interest because they may identify large embedded networks cheaply. In principle, a person who is familiar with an LP's formulation should know which constraints are network flows, and should be able to communicate this information to a computer system that solves embedded-network LPs. In practice, however, LPs often have complex structures that tend to obscure their networks. Thus our algorithms sometimes find a reasonably large network where none is immediately obvious in the original application. Even when a substantial embedded network is known in advance, our algorithms may identify it more efficiently and reliably than a human analyst.

The embedded-network problem also offers a case study of the difficulties that are likely to be encountered in devising heuristics for any structural analysis of matrices. The problem is NP-complete and offers enough complications to be interesting; yet it is sufficiently straightforward that we were able to try many combinations of heuristic approaches and implementations.

We have evaluated our algorithms for both *efficiency* and *effectiveness*: for the time they take to find an embedded network, and for the size of the network they find. On test problems from a variety of applications, we have found that efficiency is the more predictable criterion. Effectiveness varies considerably from one implementation to another and from one LP to another; in a sizable minority of cases, seemingly sophisticated algorithms find fewer network constraints than simpler and faster methods. However, simpler algorithms do tend to be less stable, in that minor changes—scanning the rows or columns in a different order, for example—sometimes substantially affect the number of network constraints found.

For most of the LPs we have investigated, our best results depend on a series of heuristic steps: several rounds of preprocessing, initial extraction of an embedded network subset, and then augmentation of the network. The preferred combination of preprocessing steps, detection algorithm and augmentation algorithm is seen to vary from one LP to the next. Thus we recommend that a computer system for network identification be designed to encourage experimentation with different combinations of heuristics. The system that we have built for our tests is set up in this way, and systems for other structural analyses may benefit from a similar design.

The next part of this introduction defines more precisely the embedded networks that we seek. We then briefly describe our data structures for the LP constraint matrix, and the test problems that are used in all our experiments. Finally, we give an outline of the rest of this paper and of its successor.

Definitions

We consider a linear program (LP) of m equations and inequalities in n bounded variables:

$$\sum_{j=1}^{n} a_{ij}x_j \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} b_i, \qquad i = 1, \ldots, m$$

$$l_j \leq x_j \leq u_j, \qquad j = 1, \ldots, n$$

The coefficients of the equations define an $m \times n$ matrix, A.

An LP represents a *pure network flow* problem if each column of A contains at most one $+1$ and one -1, and is otherwise all zeroes. For purposes of this paper, an *embedded network* is defined to be a subset of the rows of A such that each column contains at most one $+1$ and one -1 within that subset, and is otherwise all zeroes within that subset. An embedded-network linear program can be interpreted as imposing conservation of flow at the nodes of a certain network (defined by the network rows) plus other conditions on the flows (as specified by the non-network rows).

We permit two kinds of constraint transformations that may create larger embedded networks. Most important, we consider scaling the rows and columns of A to produce more rows whose nonzero values are all $+1$ or -1; only these "± 1 rows" can possibly be in an embedded network. We also allow for the addition of certain generalized "bounding rows" (or their negatives) to other rows, in a way that allows any bounding row to be included in any embedded network. These transformations leave the solution to the linear program essentially unchanged, though any scaling of the columns will cause a compensating scaling in the values of the variables.

All of our network-finding heuristics are designed to operate upon the ± 1 rows, and to skip over any others. Our task is complicated, however, by the fact that any ± 1 row remains a ± 1 row when it is scaled by -1: every $+1$ becomes a -1, and every -1 becomes a $+1$. The operation of scaling by -1, or *reflection*, can be crucial to whether a row satisfies the network conditions in some column. Thus all of our methods must provide for optional reflection of any rows that they process.

Our goal is to quickly find a large number of ± 1 rows, reflected as necessary, that comprise an embedded network. To this end, our algorithms are heuristics that do not necessarily find the largest possible network. Indeed, since the problem of finding a maximum network is NP-complete (Brown and Wright 1984) we would not expect any algorithm to consistently find the largest possible network in less time than is needed to solve the linear program.

Our algorithms do find a *maximal* embedded network that cannot be enlarged by adding any one non-network row to the network subset. There can be many maximal embedded networks for a given LP, however, and their size can be vary considerably, as our results will show.

Data structures

If A has a sizable embedded network then it must be mostly zeroes. Thus we represent A by a column-list structure (Greenberg 1978) that is widely used in large-scale LP software. The column list uses an integer and a real array of length equal to the number of nonzeroes in A, and an integer array of length $n + 1$:

> `XVALUE`(k) is the kth nonzero value in the column list.
>
> `XINDEX`(k) is the index of the row in which the kth nonzero value lies.
>
> `XPOINT`(j) points to the last component of column $j - 1$ within the column list, with `XPOINT`(1) $= 0$.

The jth column thus has nonzero components `XVALUE`(k) in rows `XINDEX`(k), for $k =$ `XPOINT`(j) $+ 1$, `XPOINT`(j) $+ 2$, ..., `XPOINT`($j + 1$).

To execute most of our algorithms efficiently, we must be able to scan individual rows of A. Thus we also set up a row-list representation in arrays `YVALUE`, `YINDEX`, and `YPOINT`. The data in `YVALUE` is the same as in `XVALUE`, except that the nonzeroes are ordered by row; `YINDEX` holds the corresponding column indices, and `YPOINT`(i) is the position of the last component of row $i - 1$ within the row list.

We speak of the nonzero values in a row or column as its *elements*. A row i and a column j *intersect* if $a_{ij} \neq 0$.

	Rows				*Columns*					
	Free	Eqty	Ineq	Total	Free	Fix	Reg	Bdd	Total	*Nonzeroes*
ENERGY[a]	1	0	2262	2263	0	553	6315	2986	9854	29063
GIFFPINC	1	548	68	617	0	0	834	258	1092	3467
GREENBEA[b]	7	2202	191	2400	4	150	4998	291	5443	32120
PIES	1	480	182	663	0	175	1240	1508	2923	13988
SCAGR25	1	300	171	472	0	0	500	0	500	2029
SCRS8	1	384	106	491	0	0	1169	0	1169	4029
SHIP12L	14	1045	106	1165	0	0	5427	0	5427	21597
SIERRA	1	528	699	1228	0	0	20	2016	2036	9252
STANDATA	1	268	199	468	0	16	955	104	1075	3686

[a] 1590 free rows deleted
[b] 105 free rows deleted

Table 1–1. *Linear programs used in the computational experiments.* Rows are classified as nonconstraining (Free), equality (Eqty) or inequality (Ineq) according to their definition in the `ROWS` section of the MPS file. Columns may be unrestricted (Free), fixed at one value (Fix), regular nonnegative (Reg) or otherwise bounded (Bdd) according to their treatment in the `BOUNDS` section of the MPS file.

Test problems

To compare different methods for finding embedded networks, we started by examining about 30 linear programs from a variety of applications. For the detailed tests reported in this paper, we chose 9 LPs that had the largest embedded networks, in either total rows or proportion of network rows. Table 1–1 gives summary statistics for these problems; Appendix A provides further information and references.

All of our test problems are distributed in the standard MPS format. Two of the test problems have numerous "free" (type `N`) rows that do not constrain the linear program. Although our preprocessing routines set aside such rows, they remain in the data structure, and their elements must be skipped whenever columns are scanned. Thus the presence of these free rows tends to obscure the behavior of the network-finding algorithms; we deleted most of them directly from the MPS file, as indicated in Table 1–1.

Outline

Section 2 describes the major preprocessing steps that precede all network-finding algorithms. A series of reduction steps first identify rows and columns that can be set aside, including the generalized bounding rows mentioned above. Several scaling steps then attempt to increase the number of ± 1 rows.

Sections 3, 4 and 5 present three different approaches to extraction of

embedded networks. The deletion methods of Sections 3 and 4 start with the full set of ± 1 rows, and progressively delete rows until the remaining ones constitute an embedded network; then rows are reinserted, if necessary, to make the network maximal. Section 3's methods examine one row at a time, and decide to delete or reflect the row based on current information about its relationship to other rows. Section 4's methods examine each column once, and delete all but one or two rows intersecting that column. The addition methods of Section 5, in contrast, start with an empty set, and add rows until a maximal network has been built. All of these approaches are based upon established ideas; references are provided at the beginning of each section.

Results of our computational experiments are presented throughout Sections 2–5 as appropriate. A comparison of all network extraction algorithms is then given in Section 6, along with some tentative conclusions. Details of the test problems and procedures are collected in Appendices A and B, respectively.

The successor to this paper investigates several algorithms that augment a known embedded network, by deleting some network rows to allow the addition of others. We report numerous test runs in which one of the network extraction heuristics is followed by one of the augmentation heuristics. For most of the LPs, these runs find the largest networks of which we are aware; often the results come quite close to certain easily computed upper bounds on the size of any network. We conclude with summary statistics on the identified networks and with remarks on alternatives and extensions.

2. Preprocessing

Certain columns of A, such as those that possess only one nonzero value, have no bearing on whether any row subset is an embedded network. Certain rows of A also need not be considered by our algorithms. Rows that contain values other than $+1$, -1 and 0 cannot possibly be in an embedded network; other rows, such as ones that represent some kinds of bounds, can be included in any network subset. We set aside some of these *inessential* rows and columns at several stages in our implementations.

This section describes two collections of preprocessing steps that set aside parts of A. *Reduction* steps look for many simple kinds of inessential rows and columns. *Scaling* steps rescale rows and columns to increase the number of rows in which all elements are $+1$ or -1. We have applied first reduction and then scaling to all of our test problems for all runs. Many of our algorithms subsequently make their own preprocessing passes to identify other kinds of inessential rows and columns, as explained in Sections 3–5.

To keep track of rows and columns that have been set aside as inessential, we maintain arrays `YSTATS` and `XSTATS` of m and n integers, respectively. Typically `YSTATS`(i) or `XSTATS`(j) is 1 for an essential row i or column j; other values indicate rows or columns that have been set aside for various reasons.

When we begin, the sets of indices of the essential rows and columns are just $\mathcal{I}_E = \{1, \ldots, m\}$ and $\mathcal{J}_E = \{1, \ldots, n\}$; we initialize all of `YSTATS` and `XSTATS` to 1. To make the reduction steps efficient, we also set up two arrays to hold the numbers of nonzero elements in the rows and columns: initially `YCOUNT`(i) = `YPOINT`$(i+1)$ − `YPOINT`(i) and `XCOUNT`(j) = `XPOINT`$(j+1)$ − `XPOINT`(j).

Reduction

Any reduction in the numbers of essential rows and columns will permit our network-finding heuristics to operate more efficiently. Some reductions can also help the heuristics to find a larger proportion of embedded network rows, by setting aside columns that have two or more elements, and rows that intersect such columns. Many kinds of reductions are possible, as indicated for example by Brearly, Mitra and Williams (1975) and by Brown and Wright (1981). We have implemented several that are simple and fast, and one that takes advantage of the special properties of certain bounding constraints in embedded networks.

To begin, the following kinds of rows and columns can be set aside as

inessential because they need not appear explicitly in the constraints of the linear program:

Rows that have no elements
Columns that have no elements

Rows specified as free (type `N`)
in the `ROWS` section of the MPS input
Columns that have nonzeroes only in free rows

Columns specified as fixed (type `FX`)
in the `BOUNDS` section of the MPS input
Rows that have nonzeroes only in fixed columns

We detect empty rows and columns by checking for $\texttt{YCOUNT}(i) = 0$ or $\texttt{XCOUNT}(j) = 0$ at initialization. Subsequently, as each free row is removed from $\mathcal{I}$, we scan its elements and decrement their entries in `XCOUNT`; if any $\texttt{XCOUNT}(j)$ falls to zero then all elements of column j must be in free rows. Fixed columns are handled similarly by decrementing `YCOUNT`.

Next, a more complicated reduction can be based on the observation that, if $\texttt{YCOUNT}(i) = 1$ for a row i defined as an equality (type `E`) in the MPS form, then the row only serves to fix a variable. Thus the row and its one intersecting column are inessential. We scan such a row to find the column, and then we scan the column to decrement the `YCOUNT` entries for its elements. If any $\texttt{YCOUNT}(i')$ falls to zero as a result, then row i' meets only fixed columns and is also inessential. The counts of other equality rows may also fall to one, in which case they fix some other columns. To account for all of these possibilities, we use the following inner loop:

Repeat for all $i \in \mathcal{I}_E$:
  If $\texttt{YCOUNT}(i) = 0$ then set $\mathcal{I}_E \leftarrow \mathcal{I}_E \backslash \{i\}$.
  Else if $\texttt{YCOUNT}(i) = 1$ and row i is an equality, then:
    Find the column j intersecting row i.
    Set $\mathcal{I}_E \leftarrow \mathcal{I}_E \backslash \{i\}$, $\mathcal{J}_E \leftarrow \mathcal{J}_E \backslash \{j\}$.
    Set $\texttt{YCOUNT}(l) \leftarrow \texttt{YCOUNT}(l) - 1$ for all $l \in \mathcal{I}_E$ such that $a_{lj} \neq 0$.

This loop is itself repeated until some pass leaves $\mathcal{I}_E$ unchanged.

Finally, if a row intersects at most one column whose count is greater than 1, then it can be regarded as a *generalized bound.* Common instances include simple bounds represented as constraint rows (rather than in the `BOUNDS` section of the MPS form) and rows that are not connected to other eligible rows by any intersecting columns.

Generalized bound rows are inessential because any network can be transformed to include them. In network terms, a generalized bound on an arc is imposed by substituting an appropriate series-parallel construction for that arc. The old arc runs between, say, node i_1 and node i_2; the

substitution creates a new node k, an arc between i_1 and k, and one or more arcs between k and i_2.

In matrix terms, a precise justification can be given as follows. Suppose first that an equality row i has been set aside as a generalized bound, and that a network subset $\mathcal{N}$ has been found within the remaining rows of $\mathcal{I}_E$. Let t be the index of the intersecting column (if any) that has count greater than one. Row i may be made to have only $+1$ and -1 elements, by suitably scaling just the row and its 1-count columns; if column t exists, then a_{it} can be made either $+1$ or -1, by properly setting the sign of the scale on row i. Thus we can consider the following transformations:

If there is no column t: Scale row i as above.

If the network has only a $+1$ (or -1) in column t: Scale row i so that $a_{it} = -1$ (or $+1$).

If the network has a $+1$ and a -1 in column t: Scale row i so that $a_{it} = +1$ (or -1), then subtract row i from the row $s \in \mathcal{N}$ such that $a_{st} = +1$ (or -1).

It is easy to verify that $\mathcal{N} \cup \{i\}$ is an embedded network after any of these operations.

If an inequality row i is set aside as a generalized bound, then it may be converted to an equality by adding a slack variable to the LP. The corresponding slack column has just one element, so i remains a generalized bound row, and the above transformations can then be applied.

To set aside all generalized bounds, we use the following inner loop:

Repeat for all $i \in \mathcal{I}_E$:
  If XCOUNT(j) = 1 for all j such that $a_{ij} \neq 0$,
    except possibly for one t such that $a_{it} \neq 0$:
    If t exists, set XCOUNT(t) $\leftarrow$ XCOUNT(t) $- 1$
    Set $\mathcal{I}_E \leftarrow \mathcal{I}_E \backslash \{i\}$, $\mathcal{J}_E = \mathcal{J}_E \backslash \{j : a_{ij} \neq 0, j \neq t\}$

At least part of every row $i \in \mathcal{I}_E$ must be scanned to check the counts of the intersecting columns. The rows that are generalized bounds must be scanned a second time to remove the 1-count columns from $\mathcal{J}_E$ (by setting XSTATS(j) = 0 for these columns). The decrementing of XCOUNT may create new 1-count columns, and hence further generalized bounds; thus the above loop is itself executed repeatedly until XCOUNT is unchanged at some pass. (After the embedded network is found, the above transformations are applied to the generalized bound rows in the reverse of the order in which they were set aside.)

	Row deletions					*Column deletions*				
	Empty	Free	Fixed	1-Row	Bound	Empty	Free	Fixed	1-Row	Bound
ENERGY	25	1	2	—	394	—	20	553	—	1542
GIFFPINC	—	1	—	26	—	—	—	—	26	—
GREENBEA	3	7	—	75	53	3	—	150	75	53
PIES	—	1	—	8	8	—	—	175	8	—
SCAGR25	—	1	—	1	129	—	—	—	1	7
SCRS8	—	1	—	25	16	—	—	—	25	9
SHIP12L	109	14	—	204	—	—	—	—	204	—
SIERRA	—	1	—	—	—	—	—	—	—	—
STANDATA	—	1	1	—	39	—	—	16	—	

Rows	
Empty	All-zero
Free	Type FR in MPS input
Fixed	Meets only fixed columns
1-Row	1-count equality
Bound	Generalized bound

Columns	
Empty	All-zero
Free	Meets only free rows
Fixed	Type FX in MPS input
1-Row	Fixed by 1-count row
Bound	1-count column in generalized bound row

Table 2–1. *Inessential rows and columns.*

	Rows		Columns		Nonzeroes		Reduce time
ENERGY	1841	(81%)	7739	(79%)	18488	(64%)	0.61
GIFFPINC	590	(96%)	1066	(98%)	2325	(67%)	0.08
GREENBEA	2262	(94%)	5162	(95%)	30064	(94%)	0.49
PIES	646	(97%)	2740	(94%)	13054	(93%)	0.16
SCAGR25	341	(72%)	492	(98%)	1410	(69%)	0.07
SCRS8	449	(91%)	1135	(97%)	3047	(76%)	0.12
SHIP12L	838	(72%)	5223	(96%)	15558	(72%)	0.26
SIERRA	1227	(100%)	2036	(100%)	7302	(79%)	0.11
STANDATA	427	(91%)	1059	(99%)	3583	(97%)	0.08

Table 2–2. *Test problems after reduction.* The percentages are relative to the comparable values before reduction. The last column gives the time (in seconds) for execution of all reduction routines.

Results of the reductions are shown in Table 2–1. Each of the two loops described above was able to remove some rows for 6 of the 9 test problems; each was executed more than twice for two of the problems.

Statistics for the reduced LPs are summarized in Table 2–2. The percentage of remaining nonzeroes is sometimes much smaller than the percentage of rows times the percentage of columns, due to the elimination of a fairly dense free row that represents the objective function. (Problem SIERRA is a particularly striking example of this phenomenon.)

Scaling

Since we seek embedded pure networks, our next step is to scale the rows and columns of the matrix A so that many rows have only $+1$ and -1 elements. We then set aside the remaining rows, which cannot be in the network.

We do not attempt to update XVALUE or YVALUE when a row or column is scaled. Instead we maintain supplementary real arrays XSCALE and YSCALE such that the current value of a_{ij} is given by $\texttt{YSCALE}(i) * a_{ij} * \texttt{XSCALE}(j)$. Initially $\texttt{YSCALE}(i) = 1.0$ and $\texttt{XSCALE}(j) = 1.0$ for all $i \in \mathcal{I}_E$ and $j \in \mathcal{J}_E$. To scale row i by σ_i (multiplying each of its elements by σ_i) we replace $\texttt{YSCALE}(i)$ by $\sigma_i * \texttt{YSCALE}(i)$; columns are scaled similarly. To account for inaccuracies due to rounding, we use a tolerance, ϵ: a_{ij} is considered to be $+1$ or -1 if the magnitude of $\texttt{YSCALE}(i) * a_{ij} * \texttt{XSCALE}(j)$ lies between $1 - \epsilon$ and $1 + \epsilon$. For the scalings reported below, we took $\epsilon = 10^{-5}$.

To determine whether a ± 1 scaling exists for all rows $i \in \mathcal{I}_E$, we can apply the following myopic algorithm that scans each row and each column once. The sets $\mathcal{I}_S$ and $\mathcal{J}_S$ hold rows and columns whose scales have been determined; the subsets $\mathcal{I}_Q \subset \mathcal{I}_S$ and $\mathcal{J}_Q \subset \mathcal{J}_S$ are queues of rows and columns that have been scaled but that remain to be checked for consistency with previously-assigned column or row scales.

Induced scaling

Initialize $\mathcal{I}_S \leftarrow \emptyset$ and $\mathcal{J}_S \leftarrow \emptyset$.
Repeat until $\mathcal{I}_S = \mathcal{I}_E$ and $\mathcal{J}_S = \mathcal{J}_E$:
  Select any $i \in \mathcal{I}_E \backslash \mathcal{I}_S$ or $j \in \mathcal{J}_E \backslash \mathcal{J}_S$.
    If $i \in \mathcal{I}_E \backslash \mathcal{I}_S$ is selected, initialize $\mathcal{I}_Q \leftarrow \{i\}$, $\mathcal{J}_Q \leftarrow \emptyset$.
    If $j \in \mathcal{J}_E \backslash \mathcal{J}_S$ is selected, initialize $\mathcal{J}_Q \leftarrow \{j\}$, $\mathcal{I}_Q \leftarrow \emptyset$.
  Repeat until $\mathcal{I}_Q = \emptyset$ and $\mathcal{J}_Q = \emptyset$:
    Select any $i \in \mathcal{I}_Q$ or $j \in \mathcal{J}_Q$.
      If $i \in \mathcal{I}_Q$ is selected,
        Repeat for all $j \in \mathcal{J}_E$ such that $a_{ij} \neq 0$:
          If $j \in \mathcal{J}_S$ then
            if $|a_{ij}| \neq 1$ then stop: no scaling exists.
          Otherwise $j \notin \mathcal{J}_S$:
            scale column j so that $|a_{ij}| = 1$, and
            set $\mathcal{J}_S \leftarrow \mathcal{J}_S \cup \{j\}$, $\mathcal{J}_Q \leftarrow \mathcal{J}_Q \cup \{j\}$.
      If $j \in \mathcal{J}_Q$ is selected,
        Repeat for all $i \in \mathcal{I}_E$ such that $a_{ij} \neq 0$:
          If $i \in \mathcal{I}_S$ then
            if $|a_{ij}| \neq 1$ then stop: no scaling exists.
          Otherwise $i \notin \mathcal{I}_S$:
            scale row i so that $|a_{ij}| = 1$, and
            set $\mathcal{I}_S \leftarrow \mathcal{I}_S \cup \{i\}$, $\mathcal{I}_Q \leftarrow \mathcal{I}_Q \cup \{i\}$.

The outermost loop is needed because the submatrix defined by $i \in \mathcal{I}_E$, $j \in \mathcal{J}_E$ may be disconnected. In general, this submatrix has a block-triangular form,

$$\begin{bmatrix} A_E^{(1)} & & & \\ & A_E^{(2)} & & \\ & & \ddots & \\ & & & A_E^{(K)} \end{bmatrix},$$

such that each block $A_E^{(k)}$ is a connected submatrix; the outer loop is executed once for each block. One row or column scale in each block is left arbitrarily at 1.0, after which the rest are forced; the inner loops either determine the forced scales or find a contradiction implying that no complete ± 1 scaling is possible. The selection of $i \in \mathcal{I}_Q$ or $j \in \mathcal{J}_Q$ may be made in any convenient way; our implementation alternates between selecting rows from $\mathcal{I}_Q$ until $\mathcal{I}_Q$ is empty, and columns from $\mathcal{J}_Q$ until $\mathcal{J}_Q$ is empty (in effect, a breadth-first search).

All of our test problems are in fact connected prior to scaling. Only two, GIFFPINC and SIERRA, can be scaled completely; they derive from the same commodity-flow application, as indicated in Appendix A.

A slightly modified induced scaling algorithm may be applied to the seven LPs that cannot be completely scaled. At the two places where the original algorithm finds a contradiction in scales ("stop: no scaling exists") the modified algorithm continues; we have tried two variations, one that rescales the contradicting row or column and one that leaves it unchanged. Either way, however, the performance of the modified algorithm is unsatisfactory. In various runs with different initial choices of $i \in \mathcal{I}_E$ or $j \in \mathcal{J}_E$, we find that it usually *reduces* the number of ± 1 rows.

Thus we are led to apply a more elaborate series of heuristics to try to increase the proportion of ± 1 rows. In general terms, these algorithms proceed as follows:

(1) Set aside empty and single-element columns: Let $\mathcal{J}'_E$ be the set of $j \in \mathcal{J}_E$ such that $a_{ij} \neq 0$ for at least two $i \in \mathcal{I}_E$.

(2) Tentatively scale the rows: Scale each row $i \in \mathcal{I}_E$, if necessary, to maximize the number of columns $j \in J'_E$ such that $|a_{ij}| = 1$.

(3) Tentatively scale the columns: Look for columns $j \in J'_E$ in which all elements have the same magnitude, α_j: either $|a_{ij}| = \alpha_j$ or $a_{ij} = 0$ for every $i \in \mathcal{I}_E$. Scale each of these columns by $1/\alpha_j$.

(4) Improve the scaling: For each $i \in \mathcal{I}_E$, let p_i be the number of columns $j \in \mathcal{J}'_E$ such that $|a_{ij}| \neq 0$ or 1. Let u_k be the number of rows $i \in \mathcal{I}_E$ such that $p_i = k$. Repeat as long as possible:

> Scale a column j so that the vector $u = (u_0, u_1, u_2, \ldots)$ is lexicographically increased.

(5) Extend to a maximal scaling: Repeat for each row i that has not been successfully scaled to have all elements ± 1:

> If possible, adjust the scales so that i becomes a ± 1 row, while all current ± 1 rows remain so.

We describe briefly how these steps are efficiently implemented, and then discuss how well they worked.

Step (1) sets aside columns that obviously can always be scaled to have all elements be $+1$ or -1. It requires only the column counts that are available from the reduction passes.

Step (2) finds the most common magnitude $\sigma_i > 0$ in row i, and scales by $1/\sigma_i$ if $\sigma_i \neq 1$. One or more partial passes through the elements of each row are required. An array of the values p_i for step (4) is also most conveniently initialized by this step.

Step (3) scans the elements of each column until it has encountered two elements of different magnitude. If all elements are found to have the same magnitude $\alpha_j \neq 1$, then the column is scaled by $1/\alpha_j$, and appropriate values of p_i are adjusted accordingly.

In step (4), the vector u need not be maintained explicitly. Instead, define $p_j^{\min}$ to be the smallest p_i for any element in column j:

$$p_j^{\min} = \min\{p_i : i \in \mathcal{I}_E,\ a_{ij} \neq 0\}$$

Then rescaling column j will effect a lexicographic increase in u if and only if either

(i) some row $i \in \mathcal{I}_E$ has $p_i = p_j^{\min}$ but $|a_{ij}| \neq 0$ or 1; or

(ii) there is a magnitude α_j such that $\{i \in \mathcal{I}_E : p_i = p_j^{\min} + 1,\ |a_{ij}| = \alpha_j\}$ has more members than $\{i \in \mathcal{I}_E : p_i = p_j^{\min},\ |a_{ij}| = 1\}$.

Several full or partial passes through column j may be necessary to determine whether either of these conditions holds. If (i) holds then we choose an $\alpha_j \neq 1$ such that $\{i \in \mathcal{I}_E : p_i = p_j^{\min},\ |a_{ij}| = \alpha_j\}$ is as large as possible; otherwise we try to find α_j as defined in (ii). If α_j is found in either case then column j is scaled by $1/\alpha_j$. A final pass of the column sets $p_i = p_i + 1$ for elements that have $|a_{ij}| = 1$, and $p_i = p_i - 1$ for elements that have $|a_{ij}| = \alpha_j$.

After step (4), there is a subset $\mathcal{I}_S = \{i \in \mathcal{I}_E : p_i = 0\}$ of successfully scaled rows. Step (5) must test, for each $l \in \mathcal{I}_E \backslash \mathcal{I}_S$, whether the rows in $\mathcal{I}_S \cup \{l\}$ can be successfully scaled. Whenever the answer is yes, l is added to $\mathcal{I}_S$ (by resetting $p_l = 0$).

To carry out step (5) efficiently, we determine a block-diagonal permutation of the submatrix defined by rows in $\mathcal{I}_S$ and columns in $\mathcal{J}'_E$. Given some $l \in \mathcal{I}_E \backslash \mathcal{I}_S$, we permute row l to conform. If there are K connected blocks in the permutation, the submatrix and row l can be represented as follows:

$$\begin{bmatrix} A_S^{(1)} & 0 & & 0 & 0 \\ 0 & A_S^{(2)} & & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & & A_S^{(K)} & 0 \\ a_l^{(1)} & a_l^{(2)} & \cdots & a_l^{(K)} & a_l^{(K+1)} \end{bmatrix}$$

Existence of the desired scaling has the following characterization:

> The submatrix defined by rows in $\mathcal{I}_S \cup \{l\}$ and columns in $\mathcal{J}'_E$ admits a ± 1 scaling if and only if, for each block $k = 1, \ldots, K$, the elements within the subvector $a_l^{(k)}$ all have the same magnitude.

If all elements within each subvector do have the same magnitude, then the desired scaling is easily constructed. The converse follows from the observation that, for any connected block $A_S^{(k)}$, the ± 1 scaling is unique up to multiplication of all row scales by some constant and division of all column scales by the same constant.

Our implementation of step (5) initially finds the diagonal blocks $A_S^{(k)}$ by a breadth-first search that alternately scans rows in $\mathcal{I}_S$ and columns in $\mathcal{J}'_E$. The column indices of each block are represented as a linked list,

`JBLOCK`	list of indices of all columns in any block
`JSTART`(k)	position within `JBLOCK` of first column of the kth block
`JLINK`(j)	position within `JBLOCK` of the next column that is in the same block as `JBLOCK`(j)

The main loop considers each $l \in \mathcal{I}_E \backslash \mathcal{I}_S$ once. The magnitudes of the elements of row l within each block are checked in a single pass. If all elements have the same magnitude within each block, then l is added to $\mathcal{I}_S$ and the column scales are adjusted appropriately; the block structure is updated by merging all blocks that have columns intersecting row l, together with all columns that intersect row l but that were previously in no block.

	No scaling	Heuristic scaling	Maximal scaling
ENERGY	1050 (57%)	1163 (63%)	1325 (72%)
GIFFPINC	264 (45%)		590 (100%)
GREENBEA	396 (18%)	886 (39%)	963 (43%)
PIES	166 (26%)	232 (36%)	278 (43%)
SCAGR25	172 (50%)	172 (50%)	243 (71%)
SCRS8	35 (8%)	238 (53%)	313 (70%)
SHIP12L	828 (99%)	828 (99%)	830 (99%)
SIERRA	1161 (95%)		1227 (100%)
STANDATA	298 (70%)	341 (80%)	351 (82%)

Table 2–3. *Rows that have only elements of* $+1$, -1 *and* 0. Heuristic scaling includes steps (1)–(4) in the text, and maximal scaling also includes step (5), except for GIFFPINC and SIERRA where the induced scaling algorithm found a complete scaling. The two figures in each column are the absolute number of ± 1 rows and the percentage of essential rows successfully scaled.

	Heuristic scaling		Maximal scaling	
	Added	Time	Added	Time
ENERGY	110	2.01	163	4.27
GREENBEA	490	4.39	76	3.13
PIES	66	1.41	46	0.43
SCAGR25	0	0.11	71	0.18
SCRS8	198	0.32	76	0.21
SHIP12L	0	0.83	6	0.63
STANDATA	43	0.29	10	0.17

Table 2–4. *Effectiveness and efficiency of scaling routines.* Figures on the left are for steps (1)–(4), and on the right are for the subsequent execution of step (5). In each case, the first column indicates the number of additional ± 1 rows created, and the second gives the time for execution.

In the interest of efficiency, current row scales are not maintained; they are easily deduced from the correct column scales. Operations on the linked lists are aided by the maintenance of two auxiliary arrays:

BSIZE(k)	number of columns in the kth block
BNAME(j)	identifying number of the block to which column j belongs, or 0 if it belongs to no block

Merging of blocks is a disjoint set union problem, whose solution we implement in an obvious way. (The more sophisticated and theoretically faster algorithm analyzed by Tarjan (1975) could also have been used.)

Table 2–3 shows the number of ± 1 rows in each test problem before scaling, after steps (1)–(4), and after step (5). Both the scaling heuristics

	Heuristic then Maximal scaling		Maximal scaling alone	
	Added	Time	Added	Time
ENERGY	273	6.28	256	5.15
GREENBEA	566	7.52	494	13.53
PIES	112	1.84	128	0.97
SCAGR25	71	0.29	71	0.20
SCRS8	274	0.53	281	0.52
SHIP12L	6	1.46	6	0.86
STANDATA	53	0.46	53	0.57

Table 2–5. *Comparison of combined heuristic and maximal scaling algorithms with maximal scaling alone.* Figures on the left are for steps (1)–(5), summed from Table 2–4, and on the right are for application of step (5) only. Format is the same as in Table 2–4.

and the maximal scaling algorithm have a substantial effect in most cases. Indeed, for six of the nine test problems, the number of embedded network rows eventually found in the matrix exceeds the number of ± 1 rows that existed before scaling.

We used a straightforward but inefficient implementation of the scaling routines to produce the results in Table 2–3, which were used as a starting point for the heuristics in Sections 3, 4 and 5. Later we implemented much faster versions, for which timings are given in Table 2–4. These versions do not always yield the same number of scaled rows as shown in the previous table, but the differences are slight; after maximal scaling, the greatest changes are 313 to 309 for SCRS8, and 830 to 834 for SHIP12L.

The maximal scaling algorithm can also be applied directly to an unscaled matrix. In the cases of GIFFPINC and SIERRA, this algorithm necessarily finds a complete scaling; but the previously-described induced scaling algorithm took only 0.09 seconds for GIFFPINC and 0.24 for SIERRA, whereas the maximal algorithm took 1.47 and 1.44. Results on other test problems were mixed, as shown in Table 2–5.

3. Row-scanning deletion algorithms

A deletion algorithm starts with a set of matrix rows, and deletes rows one by one until those remaining are a network. A row-scanning algorithm examines one row at a time, and takes an action according to certain properties of the elements in that row.

This section presents our experience with row-scanning deletion algorithms for extracting embedded networks. We begin by describing an algorithm and an implementation that work in two phases; the first phase performs enough deletions to identify a network, and the second restores enough deleted rows to make the network maximal. We subsequently discuss four strategies for making the algorithm faster, which give rise to nine further implementations. Test data pertinent to particular implementations are presented throughout this section; overall comparisons of effectiveness (size of network) and efficiency (run time) are made at the end. Our results show that very efficient implementations need not sacrifice much in effectiveness.

Throughout this section we let $\mathcal{I}_0 \subseteq \{1,\ldots,m\}$ and $\mathcal{J}_0 \subseteq \{1,\ldots,n\}$ be the sets of essential rows and columns that are determined by the reduction and scaling operations of Section 2. Our implementations perform their own preprocessing as appropriate, reducing the essential sets to $\mathcal{I} \subseteq \mathcal{I}_0$ and $\mathcal{J} \subseteq \mathcal{J}_0$. Thus we describe all implementations as applying to rows in $\mathcal{I}$ and columns in $\mathcal{J}$, with the goal being a subset $\mathcal{N} \subset \mathcal{I}$ of network rows. Since these are implementations of deletion algorithms, they initialize $\mathcal{N} \leftarrow \mathcal{I}$, and delete elements from $\mathcal{N}$ until it is a network subset. The full network extracted by the algorithm, taking its own preprocessing into account, is given by $\mathcal{N} \cup (\mathcal{I}_0 \backslash \mathcal{I})$.

Principles

The implementations presented in this section are descendants of the deletion heuristics proposed by Senju and Toyoda (1968) for the knapsack problem. These heuristics were specialized by Brearly, Mitra and Williams (1975) to find large generalized upper bound (or GUB) row subsets (within which each column has at most *one* nonzero element). In their computational tests, Brearly, Mitra and Williams found the Senju-Toyoda methods superior to all of the many others they considered. The same approach was subsequently adapted by Brown and Wright (1981,1984) to the problem of extracting embedded networks, and by Brown, McBride and Wood (1985) to the corresponding problem for generalized networks. Our implementations are based on the Brown-Wright heuristic, which we begin by

describing.

Two rows may be considered to have a *conflict* wherever both have a $+1$ or both have a -1 in the same column. Conflicts prevent the appearance of certain pairs of rows in the embedded network; thus we assign each row a *penalty* equal to the number of conflicts in which it participates. If c_j^+ and c_j^- are the numbers of $+1$'s and -1's in column j, the penalty for row i may be expressed as

$$P_i = \sum_{j \in \mathcal{J}: a_{ij} = +1} (c_j^+ - 1) + \sum_{j \in \mathcal{J}: a_{ij} = -1} (c_j^- - 1)$$

A similar formula gives the penalty for the reflection of row i (the result of scaling it by -1):

$$R_i = \sum_{j \in \mathcal{J}: a_{ij} = +1} c_j^- + \sum_{j \in \mathcal{J}: a_{ij} = -1} c_j^+$$

If $P_i = 0$ for all rows $i \in \mathcal{N}$, then $\mathcal{N}$ defines a network. Otherwise some row in $\mathcal{N}$ must be reflected, or some row must be deleted from $\mathcal{N}$.

Since the goal is to reduce all penalties to zero, it makes sense to reflect row t if $P_t > R_t$, or to delete row t if $R_t \geq P_t > 0$. The algorithm based on these observations can be outlined as follows:

Row-scanning deletion

For $j \in \mathcal{J}$: Compute c_j^+, c_j^- = number of $+1$'s, -1's in column j
For $i \in \mathcal{I}$: Compute $P_i = \sum_{j \in \mathcal{J}: a_{ij} = +1} (c_j^+ - 1) + \sum_{j \in \mathcal{J}: a_{ij} = -1} (c_j^- - 1)$.

Initialize $\mathcal{N} \leftarrow \mathcal{I}$.

Repeat while there exist rows $i \in \mathcal{N}$ such that $P_i > 0$:

  Select any $t \in \mathcal{N}$ such that $P_t > 0$.
  Compute $R_t = \sum_{j \in \mathcal{J}: a_{tj} = +1} c_j^- + \sum_{j \in \mathcal{J}: a_{tj} = -1} c_j^+$

  If $R_t < P_t$ then reflect row t:
    For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow c_j^+ - 1$ and $c_j^- \leftarrow c_j^- + 1$
    For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow c_j^- - 1$ and $c_j^+ \leftarrow c_j^+ + 1$

  Otherwise $R_t \geq P_t$, so delete row t: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{t\}$
    For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow c_j^+ - 1$
    For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow c_j^- - 1$

  For all $i \in \mathcal{N}$: update P_i, if necessary,
    to take account of the new c_j^+ and c_j^- values.

As the algorithm proceeds, c_j^+ and c_j^- are the numbers of $+1$'s and -1's left in column j within the rows of $\mathcal{N}$; P_i is the number of conflicts between row i and other rows of $\mathcal{N}$, and R_i is the number of conflicts between the reflection of row i and other rows of $\mathcal{N}$. Each pass of the main loop either reflects or deletes one row. When the algorithm stops, $P_i = 0$ for all $i \in \mathcal{N}$, so the rows remaining in $\mathcal{N}$ are a network.

To see that the number of passes must be finite, consider the total P of all conflicts within the rows $i \in \mathcal{N}$. Since exactly two rows participate in each conflict,

$$P = \tfrac{1}{2} \sum_{i \in \mathcal{N}} P_i$$

Deleting row t removes all the conflicts in which it participates, and so reduces P by $P_t > 0$. Reflecting row t also removes all the conflicts in which it participates, but adds all the conflicts in which its reflection participates, so that P is reduced by $P_t - R_t > 0$. Either way, the total conflict is reduced at every pass. Eventually it falls to zero and the algorithm halts.

The particular network that is found will depend on the particular method that is used to choose row t in each pass. The original Brown-Wright algorithm chooses t so that P_t is as large as possible:

$$P_t = \max_{i \in \mathcal{N}} P_i$$

Nevertheless, the resulting network can fail to be maximal as defined in Section 1. The algorithm can only guarantee that, when row t is deleted by a pass, it must conflict with one or more other rows in $\mathcal{N}$. At subsequent passes, those others may also be deleted. Thus, following deletion, Brown and Wright (1981) suggest applying a second algorithm that attempts to restore deleted rows:

Reinsertion

Repeat for each t in $\mathcal{I} \backslash \mathcal{N}$:

If $c_j^+ = 0$ for all $a_{tj} = +1$, $j \in \mathcal{J}$; and $c_j^- = 0$ for all $a_{tj} = -1$, $j \in \mathcal{J}$:

Restore row t: $\mathcal{I} \leftarrow \mathcal{I} \cup \{t\}$

For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow 1$

For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow 1$

If $c_j^+ = 0$ for all $a_{tj} = -1$, $j \in J$; and $c_j^- = 0$ for all $a_{tj} = +1$, $j \in \mathcal{J}$:

Reflect and restore row t: $\mathcal{I} \leftarrow \mathcal{I} \cup \{t\}$

For each reflected $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow 1$

For each reflected $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow 1$

There is just one pass for each deleted row. At the end, the network subset must be maximal: every $t \notin \mathcal{N}$ conflicts with some row in $\mathcal{N}$.

We next describe an efficient implementation of the the Brown-Wright algorithms above. The remainder of this section can then compare four ideas for improvement:

- Use a heap to reduce the cost of selecting the largest penalty in the Brown-Wright deletion heuristic.
- Scan only some of the penalties at each pass, even though the largest one may not always be chosen.

- Set aside rows whose penalty is initially zero, or whose penalty falls to zero after some pass.

- Do not update the penalties, except when a row is reflected.

These ideas lead to implementations that are expected to achieve greater speed at little or no loss in effectiveness.

Implementation

Prior to the main deletion loop, two scans of the matrix are needed to initialize arrays. A column-wise scan computes c_j^+ and c_j^- (we actually store $c_j^+ - 1$ and $c_j^+ - 1$) and a row-wise scan computes P_i. Two bookkeeping arrays are also set up, to record the rows in $\mathcal{N}$ and the rows that have been reflected.

In the course of the setup scans, we can cheaply identify certain rows in $\mathcal{I}_0$ and columns in $\mathcal{J}_0$ that become inessential as a result of the scaling described in Section 2. During the column-wise scan we set aside columns that have fewer than two nonzeroes in the ± 1 rows, since there can be no conflicts within these columns. The remaining columns comprise $\mathcal{J}$. During the row scan we then set aside rows that intersect no columns in $\mathcal{J}$, since these rows cannot conflict with any others; the remaining rows comprise $\mathcal{I}$. The columns in $\mathcal{J}_0 \backslash \mathcal{J}$ are temporarily given a special marking in the XSTATS array (Section 1) and the rows in $\mathcal{I}_0 \backslash \mathcal{I}$ are marked specially in the aforementioned bookkeeping array. Table 3–1 shows that significant numbers of inessential columns and rows exist in many of our test problems.

	Columns set aside			*Rows set aside*
	0's	1's	Total	
ENERGY	689	4429	5118 (66%)	121 (9%)
GIFFPINC	0	2	2 (0%)	0 (0%)
GREENBEA	429	2804	3233 (63%)	38 (4%)
PIES	989	1327	2316 (85%)	80 (29%)
SCAGR25	99	202	301 (61%)	0 (0%)
SCRS8	53	797	850 (75%)	16 (5%)
SHIP12L	0	96	96 (2%)	0 (0%)
SIERRA	0	0	0 (0%)	0 (0%)
STANDATA	6	151	157 (15%)	0 (0%)

Table 3–1. *Preprocessing prior to row-scanning deletion.* Columns set aside have no elements or one element within the ± 1 rows; rows set aside intersect only those columns.

We next invoke the deletion algorithm. To streamline the determination of the maximum penalty at each pass, we represent $\mathcal{N}$ as a linked list, stored in an array:

`LINK`(i) index of the row that follows row i

To select t we scan this entire list at each pass, checking the penalty of each row against the largest penalty seen so far.

Using the current values of c_j^+ and c_j^-, computation of R_t requires just a scan of row t. The remaining work lies in updating the counts and penalties. Fortunately, each iteration affects only the penalties of the rows that have conflicts with row t, and possibly with its reflection; penalties of the other rows are unchanged. So if row t is deleted, we can carry out the update as follows:

For each $a_{tj} = +1$, $j \in \mathcal{J}$:
  Update $c_j^+ \leftarrow c_j^+ - 1$
  For each $a_{ij} = +1$, $i \in \mathcal{N}$: update $P_i \leftarrow P_i - 1$
For each $a_{tj} = -1$, $j \in \mathcal{J}$:
  Update $c_j^- \leftarrow c_j^- - 1$
  For each $a_{ij} = -1$, $i \in \mathcal{N}$: update $P_i \leftarrow P_i - 1$

If row t is reflected then some penalties may also increase:

For each $a_{tj} = +1$, $j \in \mathcal{J}$:
  Update $c_j^+ \leftarrow c_j^+ - 1$ and $c_j^- \leftarrow c_j^- + 1$
  For each $a_{ij} = +1$, $i \in \mathcal{N}$: update $P_i \leftarrow P_i - 1$
  For each $a_{ij} = -1$, $i \in \mathcal{N}$: update $P_i \leftarrow P_i + 1$
For each $a_{tj} = -1$, $j \in \mathcal{J}$:
  Update $c_j^- \leftarrow c_j^- - 1$ and $c_j^+ \leftarrow c_j^+ + 1$
  For each $a_{ij} = -1$, $i \in \mathcal{N}$: update $P_i \leftarrow P_i - 1$
  For each $a_{ij} = +1$, $i \in \mathcal{N}$: update $P_i \leftarrow P_i + 1$

In either of these cases, our outer loop is a scan of the elements of row t. For each a_{tj} in the row we update c_j^+ and c_j^-; then we scan the elements of column j, and for each intersecting network row i we update P_i. Thus both the row and column lists of nonzeroes are used in the update.

Since the algorithm is normally applied to a matrix that has a substantial embedded network, many of the rows remain in $\mathcal{N}$ at every pass, and the work of finding the maximum penalty is roughly proportional to m. By comparison, the work of updating the penalties depends only on the number of elements in all columns intersecting row t; this number does not necessarily increase as LPs become larger.

The reinsertion algorithm involves just one fast scan through part or all of each deleted row, plus a second scan if the row is restored. We keep a linked list of deleted rows, sharing space in `LINK`, and attempt to reinsert them in the reverse of their order of deletion.

Selection of rows from a heap

To speed the selection of row t at each pass, we consider partially sorting the initial penalties of the rows into a binary heap (Knuth 1973) with the largest penalty at the top. Formation of the heap is an extra but inexpensive operation, whose cost grows linearly in the number of rows. Two complementary arrays keep track of the heap relationship:

HEAP(l)	row index of the lth heap entry
HPOS(i)	heap position of the ith row: HEAP(HPOS(i)) $= i$

The entries in HEAP have the customary heap order: for each l, the penalty of HEAP(l) is greater than or equal to the penalty of HEAP($2l$) and the penalty of HEAP($2l+1$).

At each pass of the algorithm, t is set to HEAP(1), and the remaining steps are carried out as previously described. Elements of HEAP and HPOS must also be adjusted, however, to maintain the heap ordering as the penalties change. If row t is deleted, the top entry of the heap is removed, and P_t entries have their penalties decreased by one; if t is reflected, the penalty of the top entry is decreased from P_t to R_t, P_t other entries have their penalties decreased by one, and R_t entries have their penalties increased by one.

The heap ordering is maintained by fast operations of "sifting down" (when the top is removed or a penalty is decreased) and "sifting up" (when a penalty is increased). Deletion or reflection of the top element requires only the information in HEAP, but HPOS is needed when the penalties of other elements are changed. Thus HPOS must be updated whenever HEAP is modified.

Although the heap-sifting operations are inherently fast, many may be required, particularly at the earlier passes when the maximum penalty is still large. Hence the success of a heap implementation depends upon the speed of the updates. We used separate (though similar) code in each place where the heap was updated, to take advantage of the special features of particular cases. For example, when a penalty is decreased by one, we expect that the heap order is often left undisturbed. Thus, in such a case, our code does a fast check whether the penalty of HEAP(HPOS(i)) has become less than the penalty of HEAP($2 *$ HPOS(i)) or HEAP($2 *$ HPOS(i) $+ 1$); if not, the sift-down loop can be skipped entirely.

Our implementation decrements P_i once for each conflict between row i and row t; if t is reflected, we also increment P_i once for each conflict between row i and the reflection of row t. Thus P_i may be modified several times, with the result being a net increase, a net decrease, or no change at all. We could keep the number of heap-sifts to a minimum by first determining the total change in each P_i, then sifting HPOS(i) up or down

just once for each i whose penalty shows a change. However, we expect that any gains to be achieved by this arrangement would be balanced by the extra overhead of determining the total change in penalties. Thus we sift HPOS(i) separately for each conflict.

Since the heap contains only rows remaining in $\mathcal{I}$, a linked list of deleted rows can share space in HPOS. The reinsertion of rows is carried out as before.

Partial selection of rows

We were motivated to use a heap because we wanted to select a maximum penalty P_t cheaply. We next consider an alternative approach, in which the selection step is made more efficient by weakening the criterion for row t. In particular, we explore two implementations that use a simple scan of the penalties, but that maximize the penalty at each pass over a suitably chosen subset of the rows in $\mathcal{N}$.

Our implementations employ a linked list of rows in $\mathcal{N}$, as previously described, but with the "last" entry in the list circularly linked back to the "first". At the initial pass, a row $t^{(1)}$ of largest penalty is selected. At each subsequent pass k, however, the list entries are scanned only until some row $t^{(k)}$ is deemed to have a sufficiently large penalty, according to one of the rules described below. We then record the index of the last entry that was scanned, and proceed with the rest of the pass. At pass $k+1$, we start the scan where it left off, with the entry that follows the last one scanned at pass k. Thus, at the beginning of any pass, all rows remaining in $\mathcal{N}$ have been previously scanned about the same number of times.

Our first stopping rule is based on the observation that, in almost all our runs, the number of passes is substantially larger than the size of the maximum penalty. Thus the maximum will often be the same from one pass to the next; it may even increase at some passes, following a reflection. Let $M^{(k-1)}$ be the penalty that $t^{(k-1)}$ has at the $(k-1)$st pass; at the kth pass, we search the linked list as follows:

Partial selection, rule 1

Set $M^{(k)} \leftarrow 0$.
Repeat for rows $i \in \mathcal{N}$:
  If $P_i > M^{(k)}$ then set $t^{(k)} \leftarrow i$, $M^{(k)} \leftarrow P_i$.
  If $M^{(k)} \geq M^{(k-1)}$ then *exit*.

A scan of all rows in the list is forced only at the relatively rare passes where $M^{(k)} < M^{(k-1)}$. If the maximum ever increases then $M^{(k)}$ may be less than the maximum at some passes, but any discrepancy is unlikely to be large or to remain for long.

Our second stopping rule derives from the expectation that all rows having "sufficiently large" penalties will eventually be deleted. At the outset, we specify a threshold $M > 0$; as soon as the scan encounters a row whose penalty meets or exceeds M, it selects that row and stops. When the maximum penalty $M^{(k)}$ finally falls below M, we reset M to $M^{(k)}$. Thus the scan at the kth pass is as follows:

Partial selection, rule 2

Set $M^{(k)} \leftarrow 0$.
Repeat for rows $i \in \mathcal{N}$:
  If $P_i \geq M$ then set $t^{(k)} \leftarrow i$ and *exit*.
  If $P_i > M^{(k)}$ then set $t^{(k)} \leftarrow i$, $M^{(k)} \leftarrow P_i$.
If no $P_i \geq M$ was found, set $M \leftarrow M^{(k)}$.

The reflected penalty $R_{t^{(k)}}$ is computed as before, but now row $t^{(k)}$ is deleted unless $R_{t^{(k)}} < M$ as well as $R_{t^{(k)}} < P_{t^{(k)}}$.

When there are many rows whose penalty is greater than M, rule 2 should require significantly less scanning than rule 1. Once M has been reset to $M^{(k)}$, however, the two rules are much alike. The only difference occurs when $P_{t^{(k)}} > P_{t^{(k-1)}}$; at pass $k+1$, rule 1 will use $M^{(k)} = P_{t^{(k)}}$ as its threshold for exiting the loop, but rule 2 will use the same $M < P_{t^{(k)}}$ that it had at pass k. The threshold for pass 1 may thus temporarily increase, whereas the threshold for pass 2 is nonincreasing.

To determine a reasonable value for M, we have collected the distributions of penalties shown in Table 3–2. We do not consider just the P_i values because, for some of the test problems, certain rows can be reflected to a much lower penalty. Instead we show the minimum of P_i and the reflected penalty, R_i, computed for each row i prior to the first pass. (The value of R_i is not normally computed except for the one selected row during a pass. We ran a specially modified version of our program to generate these numbers.)

For each LP, the upper line in Table 3–2 shows the distribution of $\min(P_i, R_i)$ over all $i \in \mathcal{I}$. The lower line has been determined from the upper one by removing all entries for rows that are deleted by the algorithm (using the rule 1 above); thus the lower line shows the distribution of the penalties that the network rows originally had, before the non-network rows were deleted. In a majority of the problems, all penalties are quite small. In the remaining ones, moreover, virtually all of the rows that started off with $\min(P_i, R_i) > 10$ are eventually deleted. Thus we initialized $M = 10$ in our test of the second rule.

	rows	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*>10*
ENERGY	all	923	230	44	2	3	1	0	0	0	1	0	0
	net	916	188	24	1	1	1	0	0	0	0	0	0
GIFFPINC	all	332	122	88	2	34	0	12	0	0	0	0	0
	net	332	75	44	1	17	0	6	0	0	0	0	0
GREENBEA	all	776	74	24	19	17	0	2	3	0	0	0	10
	net	763	66	17	15	14	0	1	1	0	0	0	0
PIES	all	92	9	33	31	7	18	6	2	0	0	0	0
	net	92	5	12	16	4	3	0	0	0	0	0	0
SCAGR25	all	94	4	42	0	0	0	0	0	0	0	0	0
	net	70	1	0	0	0	0	0	0	0	0	0	0
SCRS8	all	215	56	18	3	1	4	0	0	0	0	0	0
	net	213	47	9	2	0	0	0	0	0	0	0	0
SHIP12L	all	0	0	0	0	0	0	192	362	168	10	2	96
	net	0	0	0	0	0	0	192	362	168	10	1	0
SIERRA	all	46	160	365	50	330	0	0	0	20	60	20	176
	net	46	160	255	0	330	0	0	0	0	0	0	12
STANDATA	all	129	53	105	21	39	1	0	0	0	0	0	3
	net	127	26	87	9	22	0	0	0	0	0	0	0

Table 3–2. *Distribution of* $\min(P_i, R_i)$ *evaluated prior to deletion.* The counts are shown first for all rows, then for rows that remained in the network after deletion (using partial selection of t_k with rule 1).

Selection of zero-penalty rows

If $P_i = 0$ at any pass of the deletion algorithm, then row i may be placed unconditionally in the network. Subsequent passes need not update or scan P_i, although the values of c_j^+ and c_j^- must still take row i into account. The cost per pass may thus be somewhat reduced.

To see that the algorithm remains valid when zero-penalty rows are treated in this way, consider the following modified statement:

Row-scanning deletion, zero-penalty option

For $j \in \mathcal{J}$: Compute c_j^+, c_j^- =number of +1s, −1s in column j

For $i \in \mathcal{I}$: Compute $P_i = \sum_{j \in \mathcal{J}: a_{ij}=+1}(c_j^+ - 1) + \sum_{j \in \mathcal{J}: a_{ij}=-1}(c_j^- - 1)$.

Initialize $\mathcal{N} \leftarrow \mathcal{I}, \mathcal{N}^* \leftarrow \emptyset$.
For $i \in \mathcal{N}$ such that $P_i = 0$:
  Optionally set $\mathcal{N} \leftarrow \mathcal{N} \backslash \{i\}$, $\mathcal{N}^* \leftarrow \mathcal{N}^* \cup \{i\}$.

Repeat while there exist rows $i \in \mathcal{N}$ such that $P_i > 0$:

  Select any $t \in \mathcal{N}$ such that $P_t > 0$.
  Compute $R_t = \sum_{j \in \mathcal{J}: a_{tj}=+1} c_j^- + \sum_{j \in \mathcal{J}: a_{tj}=-1} c_j^+$

  If $R_t < P_t$ then reflect row t:
    For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow c_j^+ - 1$ and $c_j^- \leftarrow c_j^- + 1$
    For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow c_j^- - 1$ and $c_j^+ \leftarrow c_j^+ + 1$

  Otherwise $R_t \geq P_t$, so delete row t: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{t\}$
    For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow c_j^+ - 1$
    For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow c_j^- - 1$

  For all $i \in \mathcal{N}$: update P_i, if necessary,
    to take account of the new c_j^+ and c_j^- values.

  For any i such that $P_i = 0$: Optionally set $\mathcal{N} \leftarrow \mathcal{N} \backslash i$, $\mathcal{N}^* \leftarrow \mathcal{N}^* \cup \{i\}$.

Since the values of c_j^+ and c_j^- are not decremented when a row is moved from $\mathcal{N}$ to $\mathcal{N}^*$, the penalty P_t (or R_t) must represent the number of conflicts between row t (or its reflection) and other rows in $\mathcal{N} \cup \mathcal{N}^*$. Thus the total of all conflicts in $\mathcal{N} \cup \mathcal{N}^*$ decreases at every pass, by the argument given previously, and the algorithm must still eventually stop.

When the algorithm does stop, $P_i = 0$ for any i that may remain in $\mathcal{N}$; hence no row in $\mathcal{N}$ can conflict with any other row in $\mathcal{N}$ or $\mathcal{N}^*$. Moreover, no row in $\mathcal{N}^*$ can conflict with any other row in $\mathcal{N}^*$, because these rows have penalties of zero when they enter $\mathcal{N}^*$ and are never subsequently reflected. Thus there can be no conflicts at all within $\mathcal{N} \cup \mathcal{N}^*$, and it must represent a network when the algorithm terminates. (If $t_k \in \mathcal{N}$ is reflected at some pass, then it may temporarily come into conflict with some rows in $\mathcal{N}^*$. The above argument implies, however, that t_k will have to be reflected again or deleted at a later pass, in such a way that no conflicts remain when the algorithm stops.)

For our simplest implementations of this zero-penalty option, row i is placed in $\mathcal{N}^*$ if and only if it has $P_i = 0$ initially (that is, before the first pass of the algorithm). Other rows are put in the heap or linked list, and the main loop of the algorithm proceeds as before. This approach is equivalent to treating $\{i \in \mathcal{I} : P_i = 0\}$ as a set of "prespecified network rows" in the manner described by Brown and Wright (1984).

Checking for the initially zero penalties is done in the same row scan that computes the penalties, at little extra cost; if $P_i = 0$ then row i is set aside, in much the same way as the inessential rows described previously.

	Rows		*Nonzeroes*	
	All	$P_i = 0$	All	$P_i = 0$
ENERGY	1204	743 (62%)	5407	4144 (77%)
GIFFPINC	590	332 (56%)	2323	1125 (48%)
GREENBEA	925	706 (76%)	4194	2444 (58%)
PIES	198	92 (46%)	952	576 (61%)
SCAGR25	243	51 (21%)	476	52 (11%)
SCRS8	297	201 (68%)	617	394 (64%)
SHIP12L	830	0 (0%)	15374	0 (0%)
SIERRA	1227	30 (2%)	7302	100 (1%)
STANDATA	351	75 (21%)	2086	436 (21%)

Table 3–3. *Initially zero penalties.* The left-hand columns give the total of rows in $\mathcal{I}$, and the number of rows in $\mathcal{I}$ such that $P_i = 0$ initially (followed by the latter as a percentage of the former). The right-hand columns provide analogous information for the numbers of nonzeroes in these rows, within all columns $j \in \mathcal{J}$.

For our tests, the numbers of initially zero penalties are shown in Table 3–3. The proportion of zero penalties ranges from none to three-quarters of the rows in $\mathcal{I}$; it exceeds 40% in five of the nine LPs.

A comparison of Tables 3–2 and 3–3 shows that some of the positive-penalty rows would have an initial penalty of zero if they were reflected. We cannot simply reflect all these rows and add them to $\mathcal{N}^*$, however. Each reflection changes c_j^- and c_j^+, with possible effects upon the penalties of many other rows (and their reflections).

We also consider a more elaborate version of the linked-list implementation, in which rows may be moved from $\mathcal{N}$ to $\mathcal{N}^*$ at the end of any pass. If a row is reflected to have zero penalty, then it is moved to $\mathcal{N}^*$ immediately. Other rows are moved when they are found to have zero penalty in the course of scanning the linked list; although an extra cost is incurred to check for $P_i = 0$ and to remove elements from the list, extra savings are expected because the list becomes shorter. Since eventually all $P_i = 0$, this implementation always stops with $\mathcal{N} = \emptyset$ and the network in $\mathcal{N}^*$. (A heap-based analogue is not so attractive, because the heap updates do not present as convenient an opportunity to search for zero penalties.)

Table 3–4 shows the success of our various linked-list implementations in reducing the numbers of rows examined. Each pass examines some number of penalties in determining t; the total of these numbers, over all passes, is given in the "scans" column. The "per row" column can be interpreted as the average number of passes at which each row in $\mathcal{I}$ was examined, and the "per pass" column as the average percentage of passes at which each row in $\mathcal{I}$ was examined. All of our expectations concerning these quantities are borne out by the data. The difference between full and partial selection

Max P_i scan	Take $P_i = 0$	ENERGY scans	per row	per pass	GIFFPINC scans	per row	per pass	GREENBEA scans	per row	per pass
all	no	142202	118	97.6%	68155	116	89.5%	85932	93	97.8%
part	no	30751	26	21.1%	4624	8	6.1%	37844	41	43.1%
≤ 10	no	17515	15	11.1%	4624	8	6.1%	14189	15	15.3%
part	init	11673	25	20.9%	1876	7	5.6%	8660	40	41.6%
part	yes	7594	16	13.6%	1531	6	4.6%	5688	26	27.3%
≤ 10	init	6610	14	10.9%	1876	7	5.6%	3132	14	13.6%
≤ 10	yes	3063	7	5.1%	1531	6	4.6%	1684	8	7.2%

Max P_i scan	Take $P_i = 0$	PIES scans	per row	per pass	SCAGR25 scans	per row	per pass	SCRS8 scans	per row	per pass
all	no	11401	58	82.3%	30164	124	88.0%	10777	36	95.5%
part	no	1820	9	13.1%	1918	8	5.6%	2247	8	19.9%
≤ 10	no	1820	9	13.1%	1918	8	5.6%	2247	8	19.9%
part	init	900	8	12.1%	1477	8	5.5%	692	7	19.0%
part	yes	870	8	11.7%	1225	6	3.9%	444	5	12.2%
≤ 10	init	900	8	12.1%	1477	8	5.5%	692	7	19.0%
≤ 10	yes	870	8	11.7%	1225	6	3.9%	444	5	12.2%

Max P_i scan	Take $P_i = 0$	SHIP12L scans	per row	per pass	SIERRA scans	per row	per pass	STANDATA scans	per row	per pass
all	no	76501	92	94.1%	467507	381	81.6%	39594	113	88.1%
part	no	6125	7	7.5%	16761	14	2.9%	3342	10	7.4%
≤ 10	no	1562	2	1.9%	12895	11	2.2%	2381	7	5.3%
part	init	6125	7	7.5%	16341	14	2.9%	2588	9	7.3%
part	yes	4083	5	5.0%	13507	11	2.4%	1812	7	5.1%
≤ 10	init	1562	2	1.9%	12535	10	2.2%	1848	7	5.3%
≤ 10	yes	839	1	1.0%	10035	8	1.8%	1186	4	3.4%

Table 3–4. *Scanning the linked list of penalties.* The first data column gives the number penalties examined in all scans of the list; the second data column is the first divided by the number of rows initially in the linked list; and the third data column is the second divided by the number of passes that the algorithm made.

For each LP, the first column (Max P_i scan) distinguishes implementations by how they carried out the scan: sequentially through the entire list (all), or partially by rule 1 (part) or rule 2 (≤ 10). The second column (Take $P_i = 0$) distinguishes implementations by their use of the zero-penalty option: not at all (no), before the first pass only (init), or at every pass (yes).

is dramatic in every case. Special handling of $P_i \geq 10$ and $P_i = 0$ further reduces the number of scans for LPs that have many large-penalty or zero-penalty rows.

Selection without updating

Another way to reduce the cost of a pass, proposed in Ahn (1984), is to avoid updating the penalties entirely. Instead, rows are considered for deletion or reflection in decreasing order of their original penalties, prior to the first pass. This ordering remains unchanged throughout the algorithm, except to accommodate reflected rows.

The following steps serve to carry out an algorithm along these lines:

Row-scanning deletion, no updating

For $j \in \mathcal{J}$: Compute c_j^+, c_j^- = number of $+1$'s, -1's in column j

For $i \in \mathcal{I}$: Compute $P_i^{(0)} = \sum_{j \in \mathcal{J}: a_{ij}=+1}(c_j^+ - 1) + \sum_{j \in \mathcal{J}: a_{ij}=-1}(c_j^- - 1)$.

Set $\mathcal{N}^* \leftarrow \{i \in \mathcal{I} : P_i^{(0)} = 0\}$, $\mathcal{N} \leftarrow I \backslash \mathcal{N}^*$.

Let $P = \frac{1}{2}\sum_{i \in \mathcal{N}} P_i^{(0)}$.

Repeat while $P > 0$:

- Select any $P_t^{(0)} = \max_{i \in \mathcal{N}} P_i^{(0)}$.
- Compute $P_t = \sum_{j \in \mathcal{J}: a_{tj}=+1}(c_j^+ - 1) + \sum_{j \in \mathcal{J}: a_{tj}=-1}(c_j^- - 1)$
- Compute $R_t = \sum_{j \in \mathcal{J}: a_{tj}=+1} c_j^- + \sum_{j \in \mathcal{J}: a_{tj}=-1} c_j^+$
- If $R_t < P_t$ and $R_t < P_t^{(0)}$, then reflect row t:
  - For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow c_j^+ - 1$ and $c_j^- \leftarrow c_j^- + 1$
  - For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow c_j^- - 1$ and $c_j^+ \leftarrow c_j^+ + 1$
  - Set $P \leftarrow P - P_t + R_t$.
  - If $R_t > 0$ then reset $P_t^{(0)} \leftarrow R_t$.
  - Otherwise $R_t = 0$, so set $\mathcal{N} \leftarrow \mathcal{N} \backslash \{t\}$, $\mathcal{N}^* \leftarrow \mathcal{N}^* \cup \{t\}$.
- Otherwise $R_t \geq P_t$ or $P_t > R_t \geq P_t^{(0)}$, so delete row t: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{t\}$
  - For each $a_{tj} = +1$, $j \in \mathcal{J}$: set $c_j^+ \leftarrow c_j^+ - 1$
  - For each $a_{tj} = -1$, $j \in \mathcal{J}$: set $c_j^- \leftarrow c_j^- - 1$
  - Set $P \leftarrow P - P_t$.

The algorithm must maintain P, the total conflict count, so that it knows when to stop; it cannot check directly whether the individual penalties are zero, because it does not update them.

To determine the row t, we maintain a collection of linked lists, one for each possible penalty. This arrangement, essentially a bucket sort of the penalties, requires two arrays:

`PTOP`(p)	first row that has penalty equal to p
`PLINK`(i)	next row that has the same penalty as row i

If no rows have penalty equal to p then PTOP(p) = 0; if i is the last row in its list then PLINK(i) = 0. Rows whose penalty is zero need never be entered in the list. The value $M = \max_{i \in \mathcal{N}} P_i^{(0)}$ can never increase and is easily maintained; t is set to PTOP(M) at most passes, with some extra work if PTOP(M) = 0. If a row is reflected to a positive penalty, then the linked list is easily adjusted by setting PLINK(t) $\leftarrow$ PTOP(R_t) and PTOP(R_t) $\leftarrow$ t.

Comparisons

Table 3–5 reports the reflections, deletions and insertions performed by each of our implementations on each test problem. The number of deletions minus the number of the insertions is the number of non-network rows, shown in the column labeled "Out". Larger values in this column thus imply smaller networks. We prefer to count non-network rows, rather than network rows, because the number of non-network rows is a clearer measure of success when the network is reasonably large. As an example, in a subset of 750 eligible rows, an embedded network of 650 would seem to be almost as big as a network of 700; but in fact 100 non-network rows must be deleted in the one case, whereas only 50 are deleted in the other.

On the whole, the test results are as expected. Reflections play a large part in the operation of the algorithm, but the role of reinsertions is more varied. The simple partial scan yields as large an embedded network as the full scan, with the minor exception of two extra rows deleted from STANDATA. The various modified partial scans generally yield the same number of non-network rows, or just a few more.

The no-updating option, by contrast, deletes more rows than the partial scan in all tests but one. Some of the differences are quite large: 21 for GREENBEA and SCAGR25, 17 for SIERRA, 39 for STANDATA. However, except in the case of SIERRA, reinsertions eventually make up for most of these deletions.

The behavior of the heap implementations is mixed. In most of the tests, the full-scan implementation and the two heap implementations find networks that differ in size by three rows or less. Larger differences occur only for GIFFPINC and SIERRA. (It might seem that the heap version—without the zero-penalty option—should find the same embedded network as the full-scan version, since both preprocess in the same way and both choose a row of largest penalty. Both do not necessarily choose the same row t, however, when more than one row achieves the largest penalty at the same pass.)

Table 3–6 summarizes timings for the test problems. There is little variation from one implementation to the next in the cost of setting up the penalty arrays, reinserting deleted rows, or cleaning up at the end (mostly

Max P_i scan	Take $P_i = 0$	ENERGY Refl	Del	Ins	*Out*	GIFFPINC Refl	Del	Ins	*Out*	GREENBEA Refl	Del	Ins	*Out*
all	no	48	73	0	73	14	115	6	109	47	48	0	48
heap	no	48	73	1	72	12	117	6	111	47	48	0	48
heap	init	48	73	0	73	6	123	2	121	48	47	0	47
part	no	48	73	0	73	14	115	6	109	47	48	0	48
part	init	48	73	0	73	14	115	6	109	47	48	0	48
part	yes	48	73	0	73	14	115	6	109	47	48	0	48
≤ 10	no	52	79	6	73	14	115	6	109	52	48	0	48
≤ 10	init	52	79	6	73	14	115	6	109	54	51	0	51
≤ 10	yes	52	78	6	72	14	115	6	109	52	55	0	55
no upd	yes	43	75	0	75	9	120	11	109	43	69	15	54

Max P_i scan	Take $P_i = 0$	PIES Refl	Del	Ins	*Out*	SCAGR25 Refl	Del	Ins	*Out*	SCRS8 Refl	Del	Ins	*Out*
all	no	4	66	0	66	70	71	0	71	12	26	0	26
heap	no	2	68	3	65	69	72	0	72	11	27	0	27
heap	init	2	68	4	64	69	71	1	70	13	26	2	24
part	no	4	66	0	66	70	71	0	71	12	26	0	26
part	init	4	66	0	66	70	71	0	71	12	26	0	26
part	yes	4	66	0	66	70	92	0	92	12	26	0	26
≤ 10	no	4	66	0	66	70	71	0	71	12	26	0	26
≤ 10	init	4	66	0	66	70	71	0	71	12	26	0	26
≤ 10	yes	4	66	0	66	70	92	0	92	12	26	0	26
no upd	yes	0	70	0	70	71	92	20	72	8	34	4	30

Max P_i scan	Take $P_i = 0$	SHIP12L Refl	Del	Ins	*Out*	SIERRA Refl	Del	Ins	*Out*	STANDATA Refl	Del	Ins	*Out*
all	no	1	97	0	97	43	424	0	424	50	78	2	76
heap	no	1	97	0	97	37	431	0	431	53	73	0	73
heap	init	1	97	0	97	49	419	0	419	53	73	0	73
part	no	1	97	0	97	43	424	0	424	48	80	2	78
part	init	1	97	0	97	43	424	0	424	48	80	2	78
part	yes	1	97	0	97	43	424	0	424	48	80	2	78
≤ 10	no	1	97	0	97	40	436	0	436	47	80	2	78
≤ 10	init	1	97	0	97	40	436	0	436	47	80	2	78
≤ 10	yes	1	97	0	97	40	436	0	436	47	80	2	78
no upd	yes	1	97	0	97	40	441	0	441	24	119	30	89

Table 3–5. *Effectiveness of row-scanning deletion heuristics.* For each implementation, the data columns are the number of rows reflected (Refl) and deleted (Del), the number subsequently reinserted (Ins), and the number of non-network rows following the reinsertion (Out).

The first two columns are the same as in Table 3–4, with the addition of entries for the heap implementations (heap) and the no-updating implementation (no upd).

		Delete			
	Set up	high	low	Reinsert	Clean up
ENERGY	0.64	0.94	0.08	0.05	0.04
GIFFPINC	0.09	0.40	0.03	0.01	0.01
GREENBEA	0.55	0.77	0.12	0.04	0.03
PIES	0.21	0.10	0.01	0.00	0.01
SCAGR25	0.04	0.20	0.02	0.01	0.00
SCRS8	0.07	0.08	0.01	0.00	0.01
SHIP12L	0.57	1.08	0.14	0.03	0.02
SIERRA	0.26	2.80	0.14	0.04	0.01
STANDATA	0.08	0.27	0.04	0.01	0.01

Table 3–6. *Efficiency of row-scanning deletion heuristics.* The columns give CPU seconds required for the initial computation and arrangement of penalties (Set up), the deletion algorithm (Delete), the reinsertion algorithm (Insert), and housekeeping following the algorithms (Clean up). Times for the deletion algorithm are the highest recorded, using full scan, and the lowest recorded, using the no-update option.

scanning the bookkeeping and status arrays). Thus the table gives average times for these activities. On the other hand, the cost of the deletion algorithm varies widely, with the full-scan version being the most expensive and the no-update version the least expensive in every case. Timings for these two versions are also given in the summary table.

Table 3–7 gives detailed timings for deletion in all of the implementations. Times for five of the problems are uniformly small, and two patterns emerge among the four largest. ENERGY and GREENBEA have a majority of zero-penalty rows, but their maximum penalties at each pass are relatively difficult to find (as indicated by the data in Table 3–4). Thus the heap takes less time than the straightforward partial-scan, but takes about the same time as a partial scan that avoids zero-penalty rows. (The heap implementation cannot derive a comparable benefit from avoiding zero-penalty rows; even if three-quarters of the rows have penalties of zero, their removal reduces the depth of the heap by just two.) SHIP12L and SIERRA have few zero-penalty rows, and their maximum penalties at each pass are relatively easy to find. For these problems the heap takes substantially more time than partial-scan, but little is gained by specially treating the penalties of zero. Truncation of penalties reduced the times for all four of these problems, though to widely varying degrees.

Because all implementations expend comparable effort on activities other than deletion, the ratios between the deletion times of different implementations are much larger than the ratios between the the total times. Table 3–6 shows that most of the time outside deletion is spent on setup; in some cases, setting up the arrays for the deletion and reinsertion algorithms is more work than carrying out the algorithms.

Max P_i scan	Take $P_i = 0$	ENERGY Delete	ENERGY Total	ENERGY *Out*	GIFFPINC Delete	GIFFPINC Total	GIFFPINC *Out*	GREENBEA Delete	GREENBEA Total	GREENBEA *Out*
all	no	0.94	1.67	73	0.40	0.49	109	0.77	1.39	48
heap	no	0.21	0.95	72	0.07	0.18	111	0.36	1.00	48
heap	init	0.20	0.93	73	0.06	0.17	121	0.36	0.96	47
part	no	0.35	1.07	73	0.07	0.17	109	0.52	1.15	48
part	init	0.22	0.94	73	0.05	0.15	109	0.35	0.96	48
part	yes	0.21	0.92	73	0.06	0.17	109	0.35	0.96	48
≤ 10	no	0.27	1.01	73	0.07	0.18	109	0.34	0.95	48
≤ 10	init	0.21	0.94	73	0.06	0.16	109	0.29	0.91	51
≤ 10	yes	0.19	0.90	72	0.05	0.16	109	0.29	0.94	55
no upd	yes	0.08	0.83	75	0.03	0.13	109	0.12	0.76	54

Max P_i scan	Take $P_i = 0$	PIES Delete	PIES Total	PIES *Out*	SCAGR25 Delete	SCAGR25 Total	SCAGR25 *Out*	SCRS8 Delete	SCRS8 Total	SCRS8 *Out*
all	no	0.10	0.33	66	0.20	0.25	71	0.08	0.16	26
heap	no	0.05	0.27	65	0.06	0.10	72	0.02	0.10	27
heap	init	0.04	0.26	64	0.07	0.11	70	0.02	0.11	24
part	no	0.05	0.28	66	0.05	0.11	71	0.03	0.11	26
part	init	0.05	0.27	66	0.04	0.08	71	0.02	0.11	26
part	yes	0.05	0.25	66	0.05	0.09	92	0.02	0.10	26
≤ 10	no	0.05	0.27	66	0.05	0.10	71	0.03	0.10	26
≤ 10	init	0.04	0.27	66	0.04	0.09	71	0.01	0.10	26
≤ 10	yes	0.05	0.28	66	0.05	0.09	92	0.01	0.10	26
no upd	yes	0.01	0.24	70	0.02	0.07	72	0.01	0.09	30

Max P_i scan	Take $P_i = 0$	SHIP12L Delete	SHIP12L Total	SHIP12L *Out*	SIERRA Delete	SIERRA Total	SIERRA *Out*	STANDATA Delete	STANDATA Total	STANDATA *Out*
all	no	1.08	1.70	97	2.80	3.11	424	0.27	0.37	76
heap	no	0.87	1.49	97	0.54	0.86	431	0.09	0.18	73
heap	init	0.86	1.49	97	0.54	0.85	419	0.09	0.18	73
part	no	0.71	1.34	97	0.44	0.74	424	0.08	0.18	78
part	init	0.70	1.32	97	0.44	0.73	424	0.07	0.17	78
part	yes	0.68	1.29	97	0.42	0.72	424	0.07	0.16	78
≤ 10	no	0.39	1.01	97	0.40	0.70	436	0.07	0.16	78
≤ 10	init	0.40	1.01	97	0.40	0.69	436	0.06	0.14	78
≤ 10	yes	0.40	1.02	97	0.40	0.70	436	0.06	0.16	78
no upd	yes	0.14	0.75	97	0.14	0.44	441	0.04	0.13	89

Table 3–7. *Efficiency and effectiveness of row-scanning deletion.* For each implementation, the tables show CPU seconds required for the deletion passes alone (Delete), CPU seconds required for all phases of the algorithm (Total), and the number of non-network rows following deletion and reinsertion (Out). The first two columns are the same as in Table 3–5.

4. Column-scanning deletion algorithms

A column-scanning algorithm examines one matrix column at a time, and takes an action according to certain properties of the elements in that column. Thus deletion algorithms based on column scanning operate quite differently from the previously-described algorithms based on row scanning.

This section summarizes our experience with column-scanning deletion algorithms for extracting embedded networks. We describe fifteen implementations that fall into two major classes, according to whether they maintain information about the contents of specific rows. Within each class the implementations differ in their ordering of the columns to be scanned, and in their priorities for choosing rows to delete.

This section is organized much like the preceding one. The principles of the algorithm and the fundamentals of its implementation are described first. Then particular implementations are distinguished, and their efficiency and effectiveness are compared.

We also adopt the set notation of the preceding section. Subsets $\mathcal{I}_0$ and $\mathcal{J}_0$ of essential rows and columns are assumed to be available from previous reductions and scalings. Some of our implementations further reduce $\mathcal{J}_0$ to $\mathcal{J}$ by their own preprocessing, but in this section all take $\mathcal{I} = \mathcal{I}_0$. The deletion algorithms are applied to $\mathcal{I}$ and $\mathcal{J}$; they start with $\mathcal{N} \leftarrow \mathcal{I}$, and reduce $\mathcal{N}$ until it is a network subset.

Principles

We consider a family of deletion algorithms first studied in the dissertation of Ahn (1984). The underlying idea is simple: for any column of the matrix, only two of the intersecting rows may be in the embedded network. Any other intersecting rows will have to be deleted. As a consequence, the set of essential rows may be reduced to a network subset $\mathcal{N}$ by an algorithm that scans each essential column exactly once:

Initialize $\mathcal{N} \leftarrow \mathcal{I}$.
Repeat for each $j \in \mathcal{J}$:
  Let $\mathcal{S}_j = \{i \in \mathcal{N} : a_{ij} \neq 0\}$.
  If $|\mathcal{S}_j| \geq 2$ then:
    If possible, choose $p, q \in \mathcal{S}_j$ with $a_{pj} = +1$, $a_{qj} = -1$,
      and delete all other rows: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{i \in \mathcal{S}_j : i \neq p, i \neq q\}$.
    Otherwise, choose some $p \in \mathcal{S}_j$,
      and delete all other rows: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{i \in \mathcal{S}_j : i \neq p\}$.

Clearly the scan of column j leaves no conflicts within that column (in the sense of Section 3) and creates no further conflicts in other columns. Thus,

after all columns have been scanned, the set $\mathcal{N}$ must represent a network.

This algorithm is easily extended to allow for the reflection of rows. Reflection is important when, for example, a column j has two or more $+1$'s but no -1's within the rows of $\mathcal{N}$. As stated above, the algorithm deletes all but one row p in $\mathcal{S}_j$; however, if some row $q \in \mathcal{S}_j$ can be reflected, then a_{qj} becomes -1 and two rows p, $q \in \mathcal{S}_j$ may remain in the network. The same can be said when a column has two or more -1's and no $+1$'s. Even when a column does have elements of both signs, the possibility of reflection increases the number of different pairs of rows that are eligible to be selected.

Unfortunately, rows cannot be reflected indiscriminately as a column is scanned. A reflection may create conflicts in other columns, as a result of which $\mathcal{N}$ may fail to be a network when the algorithm stops. To get around this problem, we distinguish "new" and "old" rows in the set $\mathcal{N}$. Initially, all rows are new; but whenever a column is scanned and *two* intersecting rows remain undeleted, both become old (if they are not old already). The full algorithm is stated as follows:

Column-scanning deletion

Initialize $\mathcal{N} \leftarrow \mathcal{I}$, and $L_i \leftarrow$ `new` for all $i \in \mathcal{N}$.

Repeat for each $j \in \mathcal{J}$:

Let $\mathcal{S}_j = \{i \in \mathcal{N} : a_{ij} \neq 0\}$.

If $|\mathcal{S}_j| \geq 2$ then

If possible, choose p, $q \in \mathcal{S}_j$ such that
$L_p =$ `new`, or $L_p = L_q =$ `old` and $a_{pj} = +1$, $a_{qj} = -1$:

If $a_{pj} = a_{qj}$, reflect row p.

Delete any other rows: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{i \in \mathcal{S}_j : i \neq p, i \neq q\}$.
Reset $L_p \leftarrow$ `old`, $L_q \leftarrow$ `old`.

Otherwise, choose some $p \in \mathcal{S}_j$,
and delete all other rows: $\mathcal{N} \leftarrow \mathcal{N} \backslash \{i \in \mathcal{S}_j : i \neq p\}$.

The labeling of old and new rows guarantees that, at any pass, a new row cannot contain an element in a previously-scanned column, unless it contains the only element in such a column. Thus the reflection of a new row cannot create a conflict in any previously-scanned column. After column j has been scanned, there can be no conflicts within column j or within columns previously scanned; so after all columns are scanned, $\mathcal{N}$ must represent a network.

Following the deletion algorithm, rows are restored if necessary to produce a maximal network subset:

Reinsertion

For each $j \in \mathcal{J}$:
  Let $c_j^+ = 0$ if and only if no $a_{ij} = +1$ for any $i \in \mathcal{N}$.
  Let $c_j^- = 0$ if and only if no $a_{ij} = -1$ for any $i \in \mathcal{N}$.

Repeat for each $t \in \mathcal{I} \backslash \mathcal{N}$:
  If $c_j^+ = 0$ for all $a_{tj} = +1$, $j \in \mathcal{J}$; and $c_j^- = 0$ for all $a_{tj} = -1$, $j \in \mathcal{J}$:
    Restore row t: $\mathcal{N} \leftarrow \mathcal{N} \cup \{t\}$
    For each $a_{tj} = +1$, $j \in \mathcal{J}$: make $c_j^+ \neq 0$
    For each $a_{tj} = -1$, $j \in \mathcal{J}$: make $c_j^- \neq 0$
  If $c_j^+ = 0$ for all $a_{tj} = -1$, $j \in \mathcal{J}$; and $c_j^- = 0$ for all $a_{tj} = +1$, $j \in \mathcal{J}$:
    Reflect and restore row t: $\mathcal{N} \leftarrow \mathcal{N} \cup \{t\}$
    For each reflected $a_{tj} = +1$, $j \in \mathcal{J}$: make $c_j^+ \neq 0$
    For each reflected $a_{tj} = -1$, $j \in \mathcal{J}$: make $c_j^- \neq 0$

This algorithm is identical to the reinsertion described in Section 3, except for some details in the handling of c_j^+ and c_j^-. In Section 3 we could assume that c_j^+ and c_j^- were available, because they were updated at every pass of row-scanning deletion. Here we define these quantities explicitly.

Implementation of the deletion algorithm

We employ the same two bookkeeping arrays as in our implementations of row-scanning deletion, one to record the rows in $\mathcal{N}$ and one to record the rows that have been reflected. The initialization and preprocessing scans are also similar, but they vary according to the row-deletion criterion and the column ordering; we discuss them later in this section.

The only potentially expensive part of a pass in column-scanning deletion is the search for the rows p and q. Since we have tried several criteria for choosing these rows, our implementations vary in details. Nevertheless, all are based on the same fundamental idea.

The algorithm will "take" at most two rows in the set $\mathcal{S}_j$. Thus we do not want to waste the time or space that would be necessary to store $\mathcal{S}_j$ explicitly (as, say, an array of row indices). Instead, we examine each row just once and decide immediately whether to take or delete it, based on what has been seen previously. If the row is taken, then we may have to delete another row that was previously taken.

As an example, consider how a column could be scanned if new rows are preferred to old rows. Row i is a new row if $L_i =$ `new`; call it an "old + row" if $L_i =$ `old` and $a_{ij} > 0$, or an "old − row" if $L_i =$ `old` and $a_{ij} < 0$. As the scan of column j proceeds, the algorithm may be regarded as inhabiting one of the seven "states" listed below, depending on the number and kind of

rows that have already been taken. Upon encountering a_{ij}, the algorithm's actions depend on what state it is in and on what kind of row it has found:

(a) No rows taken yet:
If i is a new row, take it and go to state (b)
If i is an old + row, take it and go to state (c)
If i is an old − row, take it and go to state (d)

(b) One new row already taken:
If i is a new row, take it and go to state (g)
If i is an old row, take it and go to state (f)

(c) One old + row already taken:
If i is a new row, take it and go to state (f)
If i is an old + row, delete it
If i is an old − row, take it and go to state (e)

(d) One old − row already taken:
If i is a new row, take it and go to state (f)
If i is an old + row, take it and go to state (e)
If i is an old − row, delete it

(e) Two old rows already taken:
If i is a new row, take it, delete one old row and go to state (f)
If i is an old row, delete it

(f) One old and one new row already taken:
If i is a new row, take it, delete the old row and go to state (g)
If i is an old row, delete it

(g) Two new rows already taken:
Delete any row

The algorithm processes column j by starting in state (a). It examines the elements a_{ij}, $j \in \mathcal{N}$, in any convenient order, and proceeds to other states depending on what it finds.

After all elements have been scanned, the final state determines whether any rows must be reflected or changed from new to old. States (a)–(e) require no reflections or changes. When the final state is (f), the new row must be relabeled as old, and if the new row and the old row have elements of the same sign in column j then the new row must be reflected. When the final state is (g), both new rows must be relabeled as old, and if they have elements of the same sign in column j then one of them must be reflected.

We implement each state as a separate block of statements in our program. Thus, in processing a_{ij}, the algorithm only needs to check whether i is a new row, an old + row or an old − row; all relevant information about previously-scanned elements is conveyed implicitly. The transition to another state is accomplished, if necessary, by a transfer of control to another statement block.

Implementation of the reinsertion algorithm

The reinsertion following column-scanning deletion is implemented much like the reinsertion following row-scanning deletion. However, before the main loop of the algorithm can begin, we must set up two arrays to hold c_j^+ and c_j^- for all $j \in \mathcal{J}$:

XPOS(j)	equals zero if and only if the network has no positive element in column j
XNEG(j)	equals zero if and only if the network has no negative element in column j

We could create these arrays by again scanning the elements of all columns $j \in \mathcal{J}$, or by scanning the elements of all rows $i \in \mathcal{N}$. However, we prefer to avoid another possibly costly matrix scan. Instead, we use a more complicated implementation that tentatively sets XPOS(j) and XNEG(j) immediately after column j is scanned by the deletion algorithm, and that makes certain corrections later if necessary.

Suppose that the deletion algorithm has just finished with column j. All but one or two intersecting rows have been deleted, possibly one row has been reflected, and possibly one or two new rows have been relabeled as old. We tentatively set XPOS in one of three ways:

(i) If there is now an old + row i in column j, set XPOS(j) = i.

(ii) If there is still a new + row i in column j, set XPOS(j) = $-i$.

(iii) Otherwise set XPOS(j) = 0.

The setting of XNEG is similar, but with "− row" replacing "+ row".

Two kinds of corrections may be needed. In case (i) or (ii), row i may be deleted by the scan of some column subsequent to j. In case (ii), it also can happen that row i is reflected as the result of a later scan. (Row i cannot be already reflected when case (ii) occurs, since any row reflected by the algorithm becomes an old row.) We fix up XPOS and XNEG accordingly by running the following loop after all columns have been scanned, but before any rows are reinserted:

For all $j \in \mathcal{J}$:
  If XPOS(j) = $\pm i$ but row i has been deleted, set XPOS(j) ← 0.
  If XPOS(j) = $-i$ then:
    If row i has been reflected, set XPOS(j) ← 0 and XNEG(j) ← i;
    Otherwise set XPOS(j) ← i.
  If XNEG(j) = $\pm i$ but row i has been deleted, set XNEG(j) ← 0.
  If XNEG(j) = $-i$ then:
    If row i has been reflected, set XNEG(j) ← 0 and XPOS(j) ← i;
    Otherwise set XNEG(j) ← i.

These operations require a scan of XPOS and XNEG, but not of the matrix elements. We can also skip the second half of the loop if XNEG(j) is reset to i in the first half. No special cost is incurred to determine whether row i has been deleted or reflected, because this information is directly available in the aforementioned bookkeeping arrays.

Column orderings

The column-scanning deletion algorithm can choose to scan the columns in any order. The simplest and cheapest implementation examines the columns $j \in \mathcal{J}$ in the order assigned to them by the data structure: first the column whose data begins at XPOINT(1) within XINDEX and XVALUE (if $1 \in \mathcal{J}$), then the column whose data begins at XPOINT(2) (if $2 \in \mathcal{J}$), and so forth. In our implementations, this *natural* order is identical to the ordering of the columns in the MPS-form input. If the scanning order has negligible effect upon the performance of the algorithm, then the natural order should be preferred.

To determine what effect the column ordering does have, we also tried the reverse of the natural order—which is equally simple and cheap—and a random ordering. For our LP test problems, the natural order was far from random; typically, the columns were arranged by time period or by type of activity.

Finally, we considered orderings based on the numbers of nonzero elements in the columns. It can be argued that columns of lowest count should be scanned first, because they require the fewest deletions; or it can be argued that columns of highest count should be scanned first, because they contain the most information about rows that will have to be deleted. We tried scanning in both increasing and decreasing order of column count.

Since column counts in sparse linear programs tend to be small, we expect many ties at relatively few values. Thus, prior to the deletion, we do a bucket sort of the columns into a collection of linked lists represented by two arrays:

CTOP(c)	first column that has count equal to c
CLINK(j)	next column that has the same count as column j (0 if none)

Each pass can then cheaply determine the next column to be scanned. Ties are broken arbitrarily by the ordering of the columns within each linked list; in our implementation, the ordering is the reverse of the natural order.

Both CTOP and CLINK are set up in one scan of the elements $j \in \mathcal{J}_0$. In accumulating the counts, we disregard inessential rows $i \notin \mathcal{I}_0$. We also mark as inessential any columns that have fewer than two elements within

the rows in $\mathcal{I}_0$; such columns cannot affect the embedded network, and by setting them aside we avoid having to scan them again during the deletion algorithm.

To assess the effects of different column orderings, we have made special runs (separate from the timing runs) that keep detailed tallies of the algorithm's passage through successive states. Table 4–1 shows some results for the example in which new rows are preferred. The different column orderings yield significantly different data in most instances, suggesting that the orderings really do make a difference to the algorithm's behavior. Similar differences were observed in the data collected from our other implementations.

Row-deletion criteria

If the scan of a column finds three or more elements, or two elements of the same sign in old rows, then the algorithm must decide which rows to delete. Like the ordering of the columns, the criterion for the deletion of rows may be a strong influence on performance.

If the algorithm is implemented straightforwardly from the above description, then it keeps no information about the rows except whether they are old or new. Thus the deletion criterion may take new rows in preference to old ones, as suggested above; or, it may take old rows in preference to new, using nearly the same states but somewhat different rules. Since neither option seems obviously preferable, we have considered both. Within each column, our implementations scan the elements according to their order in `XINDEX` and `XVALUE`; in effect, this natural ordering of the elements serves as a tie-breaker when two or more elements would be equally preferred.

As an alternative, an implementation may maintain additional row information for use by a deletion criterion. We have adopted a suggestion from Ahn (1984) that the desirability of a row be measured by the number of elements remaining in that row, within the columns not yet scanned. Rows with fewer elements remaining are to preferred, because they will have fewer chances to be deleted; a row with no elements remaining, in particular, is guaranteed to remain in $\mathcal{N}$. Where more than one row has the same number of remaining elements, the tie can be broken first by favoring either new or old rows, and then if necessary by the natural ordering.

Our implementation uses an extra array, `YELEFT`, to hold the number of elements remaining. An initial column-wise scan of the elements sets aside all columns that have fewer than two elements, as explained above, and initializes `YELEFT`(i) to be the number of elements in row i within the remaining columns. Thus we do not penalize a row for the one-element columns that it intersects.

ENERGY	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	1506	872	166	2185	445	30	16	2
Decr cnt	1524	800	167	2089	426	10	3	50
$1 \ldots n$	1522	788	197	2229	318	17	19	12
$n \ldots 1$	1518	801	178	2115	445	9	16	32

GIFFPINC	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	492	396	97	891	153	32	1	0
Decr cnt	642	244	139	849	188	0	0	68
$1 \ldots n$	467	482	54	914	218	0	0	0
$n \ldots 1$	642	244	139	849	188	0	0	68

GREENBEA	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	949	626	122	1116	617	14	13	13
Decr cnt	647	567	126	826	562	0	4	34
$1 \ldots n$	779	767	72	955	715	6	11	17
$n \ldots 1$	782	580	126	970	569	8	16	23

PIES	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	309	65	50	347	85	12	17	34
Decr cnt	119	58	41	153	73	0	4	58
$1 \ldots n$	170	56	44	194	86	0	0	66
$n \ldots 1$	309	62	53	347	85	13	17	36

SCAGR25	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	26	87	78	28	163	28	42	1
Decr cnt	25	1	96	49	96	0	0	71
$1 \ldots n$	25	91	52	117	51	0	1	48
$n \ldots 1$	13	93	75	26	155	24	57	1

Table 4–1. *Effects of different column orderings, with new rows taken in preference to old rows.*

Columns scanned: number of times a column scan ended in certain states (as defined in the text): two old rows taken (e), an old and a new row taken (f), or two new rows taken (g).

Elements scanned: number of times, summed over all scans, that the algorithm processed an element when it had already taken one old row (c or d), one new row (b), two old rows (e), an old and a new row (f), or two new rows (g).

SCRS8	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	27	209	44	50	239	8	21	11
Decr cnt	22	199	43	44	229	0	16	23
$1 \ldots n$	16	220	37	260	28	2	17	1
$n \ldots 1$	26	209	44	49	241	3	17	18

SHIP12L	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	4142	612	97	4838	97	633	0	25
Decr cnt	4070	612	96	4766	95	625	1	26
$1 \ldots n$	1426	648	84	2770	95	13	1	14
$n \ldots 1$	4125	612	97	4821	97	633	0	25

SIERRA	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	750	690	251	1705	351	190	289	42
Decr cnt	834	540	305	1712	256	306	217	61
$1 \ldots n$	740	700	251	1730	321	180	309	32
$n \ldots 1$	834	540	305	1712	256	306	217	61

STANDATA	*Columns scanned*			*Elements scanned*				
	2 Old	2 Mix	2 New	1 Old	1 New	2 Old	2 Mix	2 New
Incr cnt	610	203	74	753	137	126	17	9
Decr cnt	251	240	45	452	90	15	41	25
$1 \ldots n$	264	181	81	493	83	17	19	7
$n \ldots 1$	460	246	34	655	118	114	15	3

Table 4–1. *(continued).*

The scan of a column is implemented by use of states (a) to (d) above, plus the following:

(e) Two rows already taken; smallest YELEFT is attained in a new row

(f) Two rows already taken; smallest YELEFT is attained in an old + row

(g) Two rows already taken; smallest YELEFT is attained in an old − row

Appropriate transition rules for these states are readily determined; they depend on YELEFT(i) for each intersecting row $i \in \mathcal{N}$, as well as on whether i is a new row, an old + row, or an old − row. Thus the logic of the scan is more complicated. Some extra cost is incurred in storing the values of YELEFT for the one or two rows that have already been taken, and in comparing YELEFT values for different rows. When the scan of a column is completed, YELEFT must also be decremented by one for any taken row, at a slight extra cost.

When either column counts are used for the column ordering, or row

counts are used in the deletion criterion, we compute them by an initial column-wise scan of matrix elements. When both column counts and row counts are used, they can be computed in the same scan. This initial scan also serves as a preprocessing step, by setting aside columns in $\mathcal{J}_0$ that have fewer than two elements within the rows in $\mathcal{I}_0$. Then we let $\mathcal{J}$ be the set of remaining columns, and $\mathcal{I} = \mathcal{I}_0$.

When neither row counts nor column counts are used, no initial matrix scan is necessary. Thus we take $\mathcal{I} = \mathcal{I}_0$ and $\mathcal{J} = \mathcal{J}_0$; relative to the other implementations, we avoid the cost of preprocessing and of cleanup (to reset status vectors for columns that were set aside). The deletion algorithm may become more costly as a result, however, because it has to scan all rows in $\mathcal{J}$, including those that have less than two elements within rows in $\mathcal{I}$. The preprocessing for reinsertion may also have more work to do, because the set $\mathcal{J}$ is larger.

Comparisons

Table 4–2 lists the reflections, deletions and insertions performed by the different implementations, in the format of Table 3–5. At first glance, this data seems chaotic. Although there is significant variation between implementations for every test problem, there is no obvious trend.

A somewhat clearer picture is provided by Table 4–3, which shows the largest and smallest networks achieved with and without the use of row counts in the deletion criterion. For some test problems, notably GIFFPINC and SCAGR25, the row counts seem to have little value. On the whole, however, implementations that use row counts tend to be more reliable, in the sense that there is less variation between their best and worst results. They never exhibit egregiously bad behavior, as is sometimes seen in the implementations of criteria that do not use row counts. (More than half the rows of SHIP12L and SIERRA are deleted in some cases.) The variation would be even greater were it not for reinsertions, which naturally tend to be more numerous when more rows have been deleted.

A preference for new or old rows makes a predictably small difference when row counts are the primary deletion criterion. (Thus tests of a few cases were skipped, as can be seen in Table 4–2.) When row counts are not used, on the other hand, then the preference for new or old rows is much more significant. No clear trend emerges, however; even in tests on the same LP, a preference for new rows may give better results with some column orderings but worse results with others.

The order of column scanning also sometimes accounts for large but unpredictable differences in performance. Use of column counts is not clearly preferable, and scanning by increasing count is not clearly better or worse than scanning by decreasing count. Random scanning gives

Row count	Col order	Row type	ENERGY Refl	Del	Ins	*Out*	GIFFPINC Refl	Del	Ins	*Out*	GREENBEA Refl	Del	Ins	*Out*
yes	incr	new	149	77	1	76	198	87	0	87	128	50	1	49
yes	incr	old	142	76	1	75	170	87	0	87	125	50	1	49
yes	decr	new	216	87	1	86	112	112	5	107	76	53	2	51
yes	decr	old	196	87	1	86	112	112	5	107	74	51	1	50
yes	$1 \ldots n$	new	400	89	3	86	0	129	20	109	77	56	4	52
yes	$n \ldots 1$	new	146	76	1	75	112	112	5	107	94	54	1	53
yes	rand	new	218	116	5	111	49	123	7	116	73	68	9	59
no	incr	new	107	134	13	121	202	92	0	92	192	76	6	70
no	incr	old	140	173	5	168	171	92	8	84	121	71	0	71
no	decr	new	79	87	12	75	244	80	3	77	146	86	5	81
no	decr	old	134	172	1	171	244	80	3	77	105	54	1	53
no	$1 \ldots n$	new	98	88	5	83	0	129	20	109	103	86	4	82
no	$1 \ldots n$	old	285	184	1	183	0	129	20	109	77	95	3	92
no	$n \ldots 1$	new	99	120	9	111	244	80	3	77	159	98	6	92
no	$n \ldots 1$	old	142	174	2	172	244	80	3	77	107	77	1	76

Row count	Col order	Row type	PIES Refl	Del	Ins	*Out*	SCAGR25 Refl	Del	Ins	*Out*	SCRS8 Refl	Del	Ins	*Out*
yes	incr	new	9	61	0	61	72	70	0	70	25	30	2	28
yes	incr	old	8	62	0	62	72	70	0	70	25	30	2	28
yes	decr	new	15	62	0	62	25	71	0	71	27	24	0	24
yes	decr	old	8	62	0	62	25	71	0	71	15	26	0	26
yes	$1 \ldots n$	new	14	65	2	63	25	71	0	71	49	22	0	22
yes	$n \ldots 1$	new	10	61	0	61	71	70	0	70	28	26	0	26
yes	rand	new	10	63	1	62	45	71	0	71	21	28	1	27
no	incr	new	13	71	7	64	71	71	0	71	24	49	10	39
no	incr	old	16	62	0	62	73	71	1	70	35	29	2	27
no	decr	new	20	70	0	70	49	94	23	71	18	48	7	41
no	decr	old	24	62	0	62	26	71	0	71	21	29	0	29
no	$1 \ldots n$	new	34	76	0	76	93	49	0	49	35	35	11	24
no	$1 \ldots n$	old	24	62	0	62	91	49	0	49	62	29	0	29
no	$n \ldots 1$	new	8	74	11	63	71	82	11	71	26	49	9	40
no	$n \ldots 1$	old	12	62	0	62	73	71	1	70	34	25	0	25

Table 4–2. *Effectiveness of column-scanning deletion.* Implementations are classified according to whether they use row counts as a deletion criterion, how they order the columns for scanning, and whether they give preference to new or old rows. The column orderings are increasing or decreasing by number of nonzeroes (incr, decr), natural order forwards or backwards ($1 \ldots n$, $n \ldots 1$) and random (rand).

Row count	Col order	Row type	SHIP12L Refl	Del	Ins	*Out*	SIERRA Refl	Del	Ins	*Out*	STANDATA Refl	Del	Ins	*Out*
yes	incr	new	636	97	0	97	206	348	5	343	83	72	3	69
yes	incr	old	636	97	0	97	176	352	4	348	83	72	3	69
yes	decr	new	637	106	0	106	191	365	11	354	77	59	0	59
yes	decr	old	637	106	0	106	161	360	11	349	63	60	0	60
yes	$1 \ldots n$	new	13	144	38	106	191	368	4	364	76	83	0	83
yes	$n \ldots 1$	new	636	97	0	97	191	365	11	354	71	56	0	56
yes	rand	new	614	125	0	125	283	457	9	448	73	87	5	82
no	incr	new	613	742	636	106	346	886	205	681	73	155	15	140
no	incr	old	612	733	636	97	16	695	4	691	76	95	12	83
no	decr	new	613	735	638	97	229	873	174	699	56	87	4	83
no	decr	old	613	742	636	106	0	691	0	691	38	110	0	110
no	$1 \ldots n$	new	13	735	14	721	341	881	190	691	192	93	24	69
no	$1 \ldots n$	old	13	742	12	730	16	698	7	691	106	65	0	65
no	$n \ldots 1$	new	613	742	636	106	229	873	174	699	64	165	19	146
no	$n \ldots 1$	old	612	733	636	97	0	691	0	691	35	83	2	81

Table 4–2. *(continued).*

		Using row counts Out	Versions	*No row counts* Out	Versions
ENERGY	best	75	incr/old, $n \ldots 1$/new	75	decr/new
	worst	86	decr/old, $1 \ldots n$/new	183	$1 \ldots n$/old
GIFFPINC	best	87	incr/new, incr/old	77	decr/new, $n \ldots 1$/old
	worst	109	$1 \ldots n$/new	109	$1 \ldots n$/new, $1 \ldots n$/old
GREENBEA	best	49	incr/new, incr/old	53	decr/old
	worst	53	$n \ldots 1$/new	92	$1 \ldots n$/old, $n \ldots 1$/new
PIES	best	61	incr/new, $n \ldots 1$/new	62	incr/old, $1 \ldots n$/old
	worst	63	$1 \ldots n$/new	76	$1 \ldots n$/new
SCAGR25	best	70	incr/new, $n \ldots 1$/new	49	$1 \ldots n$/new, $1 \ldots n$/old
	worst	71	decr/new, $1 \ldots n$/new	71	incr/new, $n \ldots 1$/new
SCRS8	best	22	$1 \ldots n$/new	24	$1 \ldots n$/new
	worst	28	incr/new, incr/old	41	decr/new
SHIP12L	best	97	incr/new, $n \ldots 1$/new	97	decr/new, $n \ldots 1$/old
	worst	106	decr/new, $1 \ldots n$/new	730	$1 \ldots n$/old
SIERRA	best	343	incr/new	681	incr/new
	worst	364	$1 \ldots n$/new	699	decr/new, $n \ldots 1$/new
STANDATA	best	56	$n \ldots 1$/new	65	$1 \ldots n$/old
	worst	83	$1 \ldots n$/new	146	$n \ldots 1$/new

Table 4–3. *Sample of best and worst results extracted from Table 4–2.* Where more than two runs were tied for best or worst, only two representatives are shown.

generally mediocre results in the one implementation we tried (and we have no reason to believe it would perform differently in other implementations).

Timings of the test runs showed that only two factors made a consistently substantial difference: whether row counts were used in the deletion criterion, and whether column counts were used for the scan order. The results for all four combinations of these factors, averaged over all relevant runs, are shown in Table 4–4.

Just as in row-scanning deletion, setting up was a major part of the cost. The extra setup before reinsertion was relatively, but not negligibly, inexpensive. Use of row counts in the deletion criterion led to only a small extra cost in setup and execution.

The largest differences involve the runs that use neither row nor column counts. These implementations avoid setup and cleanup work at the cost of additional work elsewhere, as explained previously. Generally their total cost was lower; the main exception was for PIES, which has the highest proportion of zero-count and one-count columns and of rows that intersect only such columns.

ENERGY		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.33	0.36	0.08	0.04	0.03	0.83
yes	no	0.28	0.36	0.09	0.04	0.04	0.81
no	yes	0.30	0.32	0.08	0.04	0.04	0.78
no	no		0.59	0.14	0.04		0.77

GIFFPINC		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.05	0.09	0.02	0.01	0.00	0.17
yes	no	0.04	0.09	0.02	0.01	0.00	0.15
no	yes	0.04	0.09	0.02	0.01	0.00	0.15
no	no		0.08	0.01	0.01		0.10

GREENBEA		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.28	0.39	0.06	0.03	0.02	0.78
yes	no	0.25	0.37	0.05	0.03	0.02	0.72
no	yes	0.26	0.35	0.06	0.03	0.02	0.72
no	no		0.54	0.08	0.04		0.65

PIES		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.12	0.07	0.01	0.01	0.01	0.21
yes	no	0.11	0.08	0.01	0.01	0.01	0.21
no	yes	0.11	0.06	0.01	0.01	0.01	0.20
no	no		0.24	0.04	0.01		0.28

SCAGR25		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.02	0.03	0.00	0.00	0.00	0.05
yes	no	0.02	0.03	0.01	0.00	0.01	0.07
no	yes	0.02	0.02	0.01	0.01	0.00	0.06
no	no		0.04	0.01	0.00		0.05

Table 4–4. *Efficiency of column-scanning deletion.* The implementations have been collected into four groups—according to whether they used row counts in the deletion criterion, and whether they used column counts to determine the scanning order—and the timings (in seconds) are averaged over these groups. (Timings for individual steps do not always add exactly to the totals, because all averages have been rounded to two places. Blank entries indicate steps that require zero time because they are skipped in the "no/no" implementation.)

SCRS8		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.04	0.04	0.01	0.01	0.01	0.09
yes	no	0.04	0.04	0.01	0.01	0.01	0.10
no	yes	0.04	0.03	0.01	0.01	0.00	0.08
no	no		0.07	0.01	0.01		0.09

SHIP12L		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.27	0.51	0.08	0.04	0.02	0.92
yes	no	0.22	0.46	0.07	0.06	0.02	0.83
no	yes	0.23	0.48	0.09	0.13	0.02	0.96
no	no		0.46	0.08	0.11		0.65

SIERRA		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.10	0.24	0.03	0.03	0.01	0.40
yes	no	0.09	0.21	0.03	0.03	0.01	0.37
no	yes	0.09	0.20	0.03	0.05	0.01	0.37
no	no		0.19	0.03	0.05		0.28

STANDATA		*Deletion*		*Reinsertion*			
Row counts	Column counts	Setup	Delete	Setup	Insert	Concl	*Total*
yes	yes	0.04	0.09	0.01	0.01	0.00	0.15
yes	no	0.03	0.07	0.02	0.00	0.00	0.13
no	yes	0.04	0.08	0.01	0.01	0.00	0.14
no	no		0.07	0.01	0.01		0.10

Table 4–4. *(continued).*

5. Row-scanning addition algorithms

An addition algorithm is the opposite of a deletion algorithm. It starts with an empty subset, which is trivially a network, and tries to add rows so that the subset remains a network. When no more rows can be added, a maximal embedded network has been found.

Addition algorithms need to scan each essential row only once. We view them as unsophisticated methods that are most attractive for their speed and simplicity. This section first presents a straightforward addition algorithm in general terms, then describes four implementations that add the rows in different orders. Comparisons of the results appear at the end.

As in previous sections, we assume that sets $\mathcal{I}_0$ and $\mathcal{J}_0$ of essential rows are available following the reduction and scaling described in Section 2. After possibly some further preprocessing, our implementations operate upon subsets $\mathcal{I} \subseteq \mathcal{I}_0$ and $\mathcal{J} \subseteq \mathcal{J}_0$. All start with a network subset $\mathcal{N} = \emptyset$, and add rows to $\mathcal{N}$ as they proceed.

Principles

Addition heuristics for the embedded GUB problem were studied by Brearly, Mitra and Williams (1975), who found that they often rivaled deletion algorithms in effectiveness. Several row-scanning addition heuristics are developed for embedded generalized networks by Brown, McBride and Wood (1985).

Our row-scanning addition algorithm is much the same as the reinsertion algorithm in Sections 3 and 4, except that it starts with an empty network subset instead of a subset determined by some preceding algorithm. Its steps are as follows:

Row-scanning addition

Initialize $\mathcal{N} \leftarrow \emptyset$, and $c_j^+ \leftarrow c_j^- \leftarrow 0$ for all $j \in \mathcal{J}$.

Repeat for each $i \in \mathcal{I}$:

  If $c_j^+ = 0$ for all $a_{ij} = +1$, $j \in \mathcal{J}$; and $c_j^- = 0$ for all $a_{ij} = -1$, $j \in \mathcal{J}$:

    Add row i: $\mathcal{N} \leftarrow \mathcal{N} \cup \{i\}$

    For each $a_{ij} = +1$, $j \in \mathcal{J}$: make $c_j^+ \neq 0$

    For each $a_{ij} = -1$, $j \in \mathcal{J}$: make $c_j^- \neq 0$

  If $c_j^+ = 0$ for all $a_{ij} = -1$, $j \in \mathcal{J}$; and $c_j^- = 0$ for all $a_{ij} = +1$, $j \in \mathcal{J}$:

    Reflect and add row i: $\mathcal{N} \leftarrow \mathcal{N} \cup \{i\}$

    For each reflected $a_{ij} = +1$, $j \in \mathcal{J}$: make $c_j^+ \neq 0$

    For each reflected $a_{ij} = -1$, $j \in \mathcal{J}$: make $c_j^- \neq 0$

Each row in $\mathcal{I}$ need be considered for addition only once. When all rows have been considered, $\mathcal{N}$ must contain a maximal network subset.

Implementation

The above algorithm may try to add the rows in any order. Our two simplest and fastest implementations take the rows in their natural order within the data structure, and in the reverse of their natural order. No setup is required, except for initialization to zero of the two arrays that hold c_j^+ and c_j^-; two other bookkeeping arrays that indicate network status—whether a row is in $\mathcal{N}$, and whether it has been reflected—are set by the algorithm as it examines each row. The subsets of eligible rows are just $\mathcal{J} = \mathcal{J}_0$ and $\mathcal{I} = \mathcal{I}_0$.

We also consider two implementations that use row counts to determine an ordering. An initial column-wise pass sets aside all $j \in \mathcal{J}_0$ that have fewer than two elements within the rows of $\mathcal{I}_0$, since these columns are irrelevant to whether any subset is a network; the remaining columns comprise $\mathcal{J}$. The same pass counts the number of intersections between each row $i \in \mathcal{I}_0$ and the columns of $\mathcal{J}$. Rows that have counts of zero can then be set aside; since they intersect only one-element columns within $\mathcal{J}_0$, they can appear in any network. The remaining rows comprise $\mathcal{I}$, and are bucket sorted by count into series of linked lists:

RTOP(r)	first row that has count equal to r
RLINK(i)	next row that has the same count as row i (0 if none)

Using these lists, we can easily scan the rows in order of either increasing or decreasing count. (Ties are broken in the reverse of the natural order, because it is convenient to set up the linked lists in that order.)

Because $\mathcal{N} = \emptyset$ initially, the first row scanned can always be added either reflected or unreflected. A subsequently scanned row can also be added either way provided that it intersects no columns in $\mathcal{J}$ that intersect rows in $\mathcal{N}$. In these "free" cases we arbitrarily add the row unreflected. If instead we added all such rows reflected, then the algorithm would find the same network except with signs of all rows in $\mathcal{N}$ reversed.

Comparisons

Table 5–1 summarizes the algorithm's behavior. The row orderings are seen to make a considerable difference in almost every case. Increasing count seems generally preferable to decreasing, and one or the other of the natural orderings is often equally good or better. Perhaps the decreasing-count implementation tends to put several large-count rows in the network

at the outset, consequently blocking many small-count rows later on.

Table 5–2 summarizes the timings. Since one pass through the matrix elements is required both for setting up the row counts and for adding the network rows, the setup took about the same time as the addition algorithm. Thus the runs that did not use row counts were about twice as fast.

Row count	Row order	ENERGY Unrf	Refl	Free	*Out*	GIFFPINC Unrf	Refl	Free	*Out*	GREENBEA Unrf	Refl	Free	*Out*
yes	incr	260	90	768	86	215	38	233	104	146	34	686	59
yes	decr	632	144	194	234	198	204	92	96	630	81	82	132
no	$1 \ldots n$	491	266	374	194	297	151	51	91	156	49	685	73
no	$n \ldots 1$	392	67	771	95	264	81	152	93	560	135	168	100

Row count	Row order	PIES Unrf	Refl	Free	*Out*	SCAGR25 Unrf	Refl	Free	*Out*	SCRS8 Unrf	Refl	Free	*Out*
yes	incr	71	2	54	71	72	25	75	71	212	15	41	29
yes	decr	104	12	13	69	72	48	51	72	206	15	31	45
no	$1 \ldots n$	82	8	118	70	51	91	52	49	214	35	35	29
no	$n \ldots 1$	106	3	99	70	72	47	52	72	211	24	47	31

Row count	Row order	SHIP12L Unrf	Refl	Free	*Out*	SIERRA Unrf	Refl	Free	*Out*	STANDATA Unrf	Refl	Free	*Out*
yes	incr	0	87	637	106	60	30	646	491	32	53	191	75
yes	decr	720	1	12	97	550	40	133	504	133	68	39	111
no	$1 \ldots n$	0	637	87	106	20	396	321	490	20	168	53	110
no	$n \ldots 1$	708	1	24	97	390	0	146	691	4	48	203	96

Table 5–1. *Effectiveness of row-scanning addition heuristics.* Separate counts are given for rows that had to be added unreflected (Unrf), rows that had to be added reflected (Refl), and rows that could have been added either reflected or unreflected (Free). The last figure (Out) represents total rows not included in the network.

Row counts	ENERGY Setup	Add	Concl	*Total*	GIFFPINC Setup	Add	Concl	*Total*
yes	0.32	0.36	0.04	0.72	0.05	0.06	0.01	0.12
no	0.08	0.28		0.36	0.01	0.05		0.06

Row counts	GREENBEA Setup	Add	Concl	*Total*	PIES Setup	Add	Concl	*Total*
yes	0.28	0.24	0.03	0.55	0.11	0.04	0.01	0.16
no	0.05	0.22		0.26	0.02	0.06		0.08

Row counts	SCAGR25 Setup	Add	Concl	*Total*	SCRS8 Setup	Add	Concl	*Total*
yes	0.02	0.02	0.01	0.04	0.05	0.03	0.01	0.08
no	0.00	0.02		0.02	0.01	0.04		0.05

Row counts	SHIP12L Setup	Add	Concl	*Total*	SIERRA Setup	Add	Concl	*Total*
yes	0.25	0.26	0.03	0.54	0.11	0.11	0.01	0.23
no	0.04	0.24		0.28	0.02	0.11		0.13

Row counts	STANDATA Setup	Add	Concl	*Total*
yes	0.05	0.04	0.01	0.08
no	0.01	0.04		0.05

Table 5–2. *Efficiency of row-scanning addition heuristics.* For each LP, times (in seconds) are averaged over all implementations that use row counts (yes) and all that do not use rows counts (no). (Times for setup, addition and conclusion may not sum exactly to the total, due to the effects of rounding. Blank entries indicate that a conclusion step is unnecessary when row counts are not used.)

6. Summary and conclusions

Our experimental results are summarized in Figure 6–1. For each of the nine LPs, the behavior of each implementation is plotted as a point in two dimensions, with times along the horizontal axis and numbers of non-network rows on the vertical axis. Thus an implementation dominates all others that appear above and to the right. If there were a consistent tradeoff between efficiency and effectiveness, the points would lie on a curve stretching roughly from the upper left to the lower right; but this is clearly seldom the case.

As expected, addition algorithms that do not sort the rows are the fastest. Addition algorithms that do sort the rows are generally next-fastest, though they sometimes overlap the faster column-scanning deletion algorithms. In the cases of SCAGR25 and SHIP12L, one of the cheapest addition runs also found the largest network; but in most cases, the largest network was found by some implementation of a deletion heuristic.

There is a curious relationship between the performance of column-scanning and row-scanning deletion. In every case, the most successful column-scanning runs either dominate all the row-scanning runs, or are nearly as good as the best row-scanning runs; row scanning never dominates column scanning. Yet the worst column-scanning runs are often far inferior in the size of the network that is detected. This behavior suggests that there is a fundamental difference in the nature of the two approaches.

Our row-scanning heuristics use strong criteria to guide their actions. The row penalty, in particular, can be regarded as a summary of information about all rows that conflict with a given row (or its reflection). Each pass must choose a row that has maximum (or almost maximum) penalty, and must delete or reflect the chosen row according to the relative size of its penalty and reflected penalty. As a result of these restrictions, our implementations of row-scanning heuristics are about equally effective; their variation is mainly in efficiency.

Our column-scanning heuristics use weaker criteria. Each pass chooses a column from some simple ordering that is not directly related to network properties. Only the rows intersecting the chosen column are then considered for deletion or reflection. In deciding which intersecting rows to delete, some of our implementations use another weak criterion; others use a moderately strong criterion that is not specifically associated with network properties (like the row penalties) but that does take account of the nonzeroes left in all the unscanned columns. By using these simple criteria, our column-scanning implementations all manage to be fairly efficient, but they vary widely in effectiveness. The pattern can be seen most clearly in

the data for ENERGY and GREENBEA, where the row-scanning points tend to lie along a horizontal line and column-scanning points along a vertical line.

Given a new linear program in which an embedded network is sought, what can we recommend? It seems that the largest network is most likely to be found by some inexpensive addition or column-scanning deletion algorithm. However, different implementations of these algorithms give the best results for different problems. To use these methods effectively, therefore, it is necessary to make several runs using a variety of implementations. If only one run can be made, then a more expensive row-scanning deletion algorithm will more reliably give a large network.

How could we search for even larger networks? One possibility is to try an even broader collection of implementations. For example, in the column-scanning algorithms we could process the elements of each column in the reverse of their natural order; in the row-scanning algorithms we could examine the row penalties in a different order, so that ties between rows of equal penalty would be broken differently. No doubt some larger networks could be found in this way. More runs would be necessary, however, and each further run would have a lower chance of success.

As an alternative, we can consider augmenting known networks. We are motivated by the knowledge that all of the embedded networks found by our implementations, whatever their size, are maximal. To move from a smaller maximal network to a larger one, certain rows of the smaller one must be deleted, permitting certain rows of the larger to be added. This suggests the possibility of heuristics that, given an existing maximal network, attempt to enlarge it by systematically searching for simple exchanges of network and non-network rows.

The second part of this paper describes implementations of three exchange heuristics. Our experiments show that significantly larger networks can often be found, at reasonable cost, by running first a deletion or addition algorithm and then an exchange algorithm. In most cases, the sizes of the resulting networks nearly equal certain easily-computed upper bounds, so that we can be sure of having found a nearly maximum embedded network.

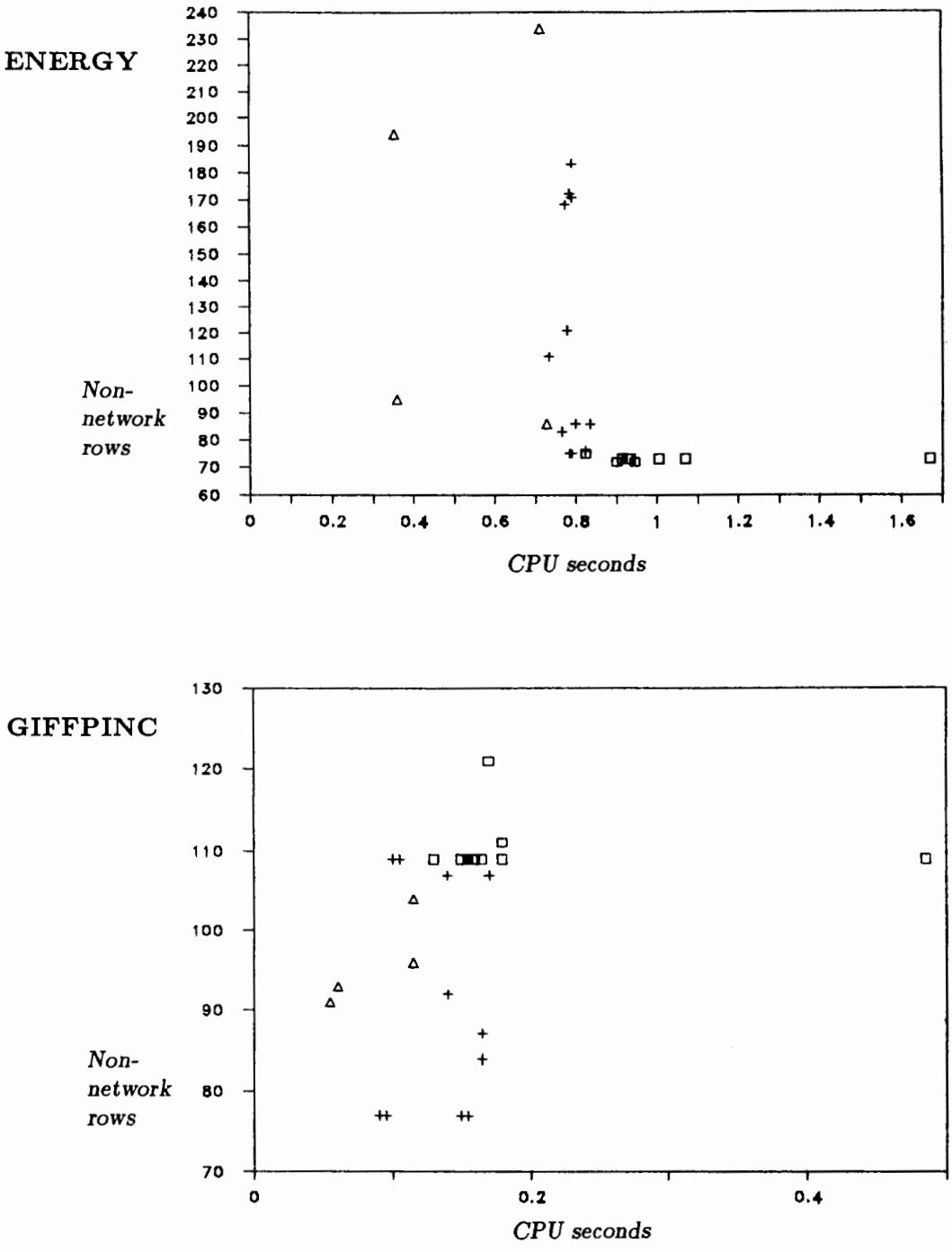


Figure 6–1. *Summary of embedded networks found by the extraction heuristics.* The vertical axis is the number of rows not in the network. The horizontal axis is the total execution time, including setup, preprocessing and cleanup where appropriate.

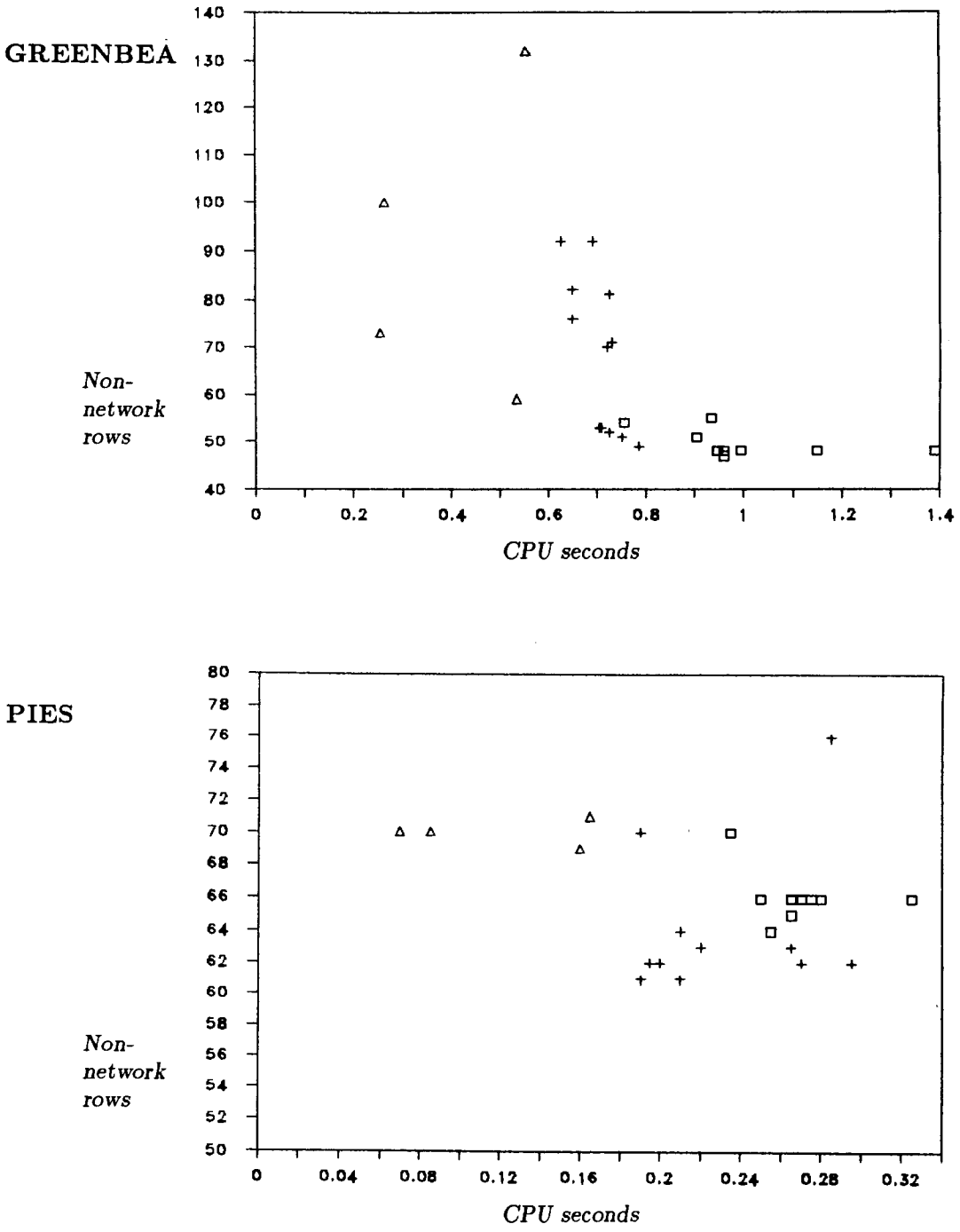


Figure 6–1. *Summary of embedded networks (continued).*

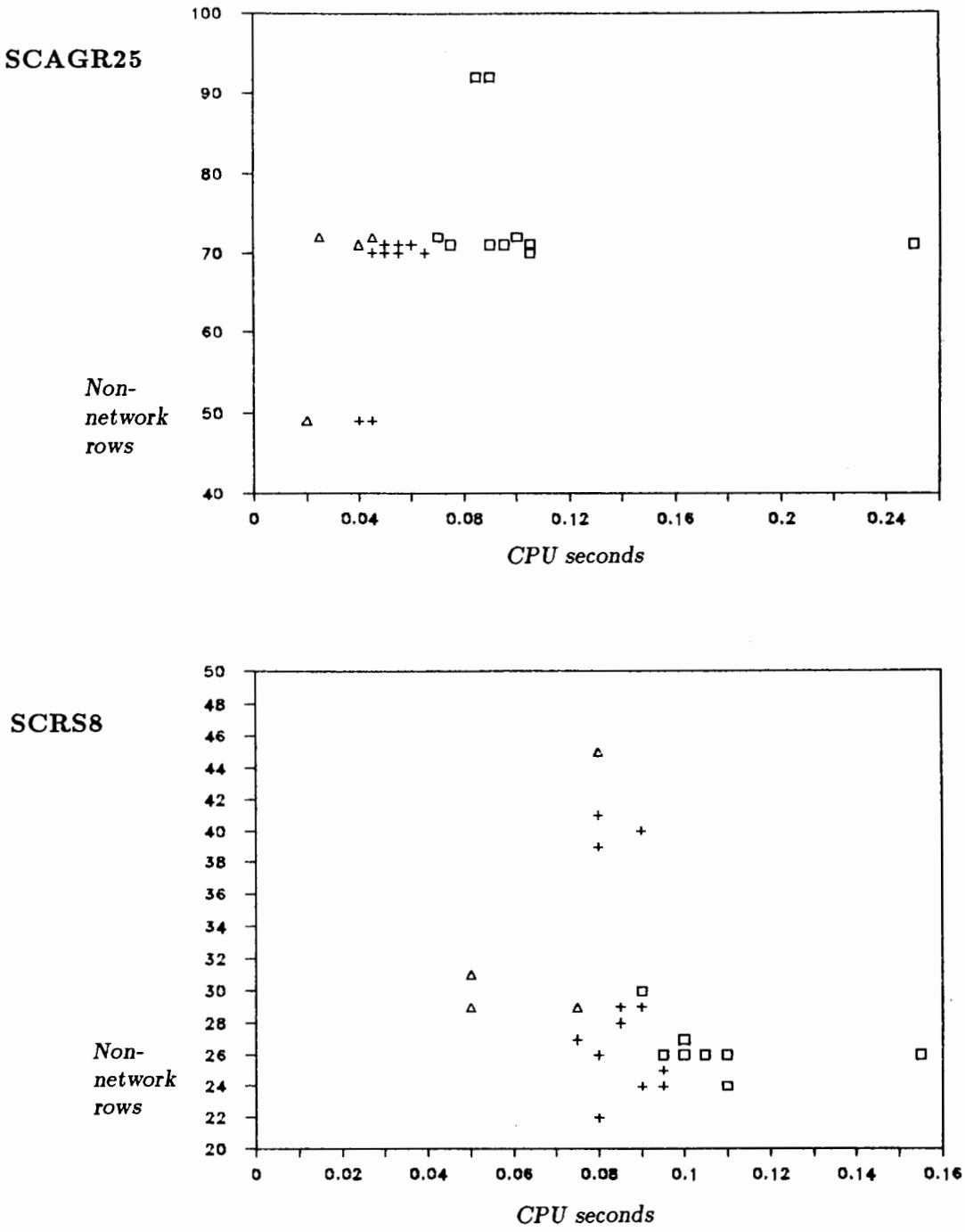


Figure 6–1. *Summary of embedded networks (continued).*

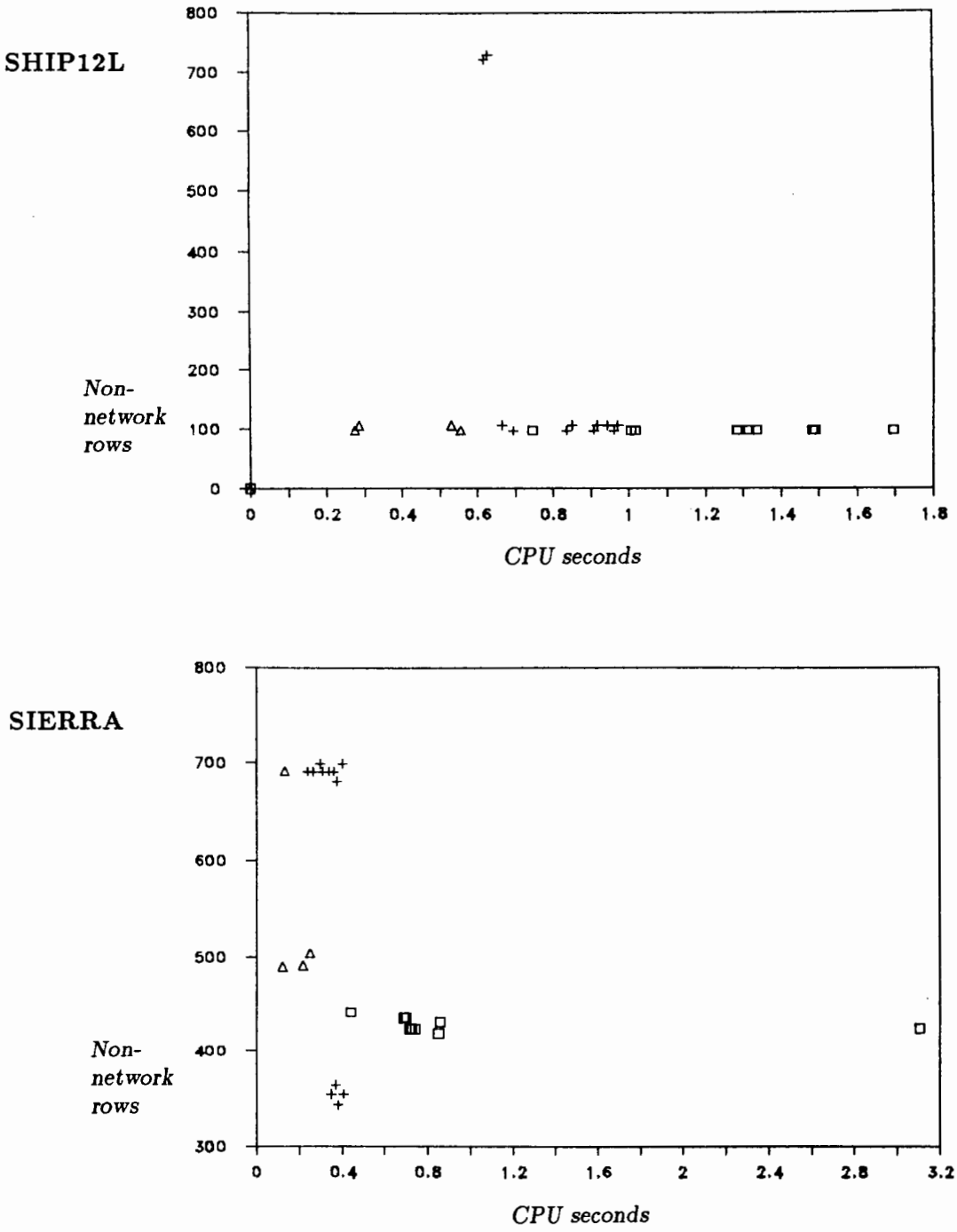
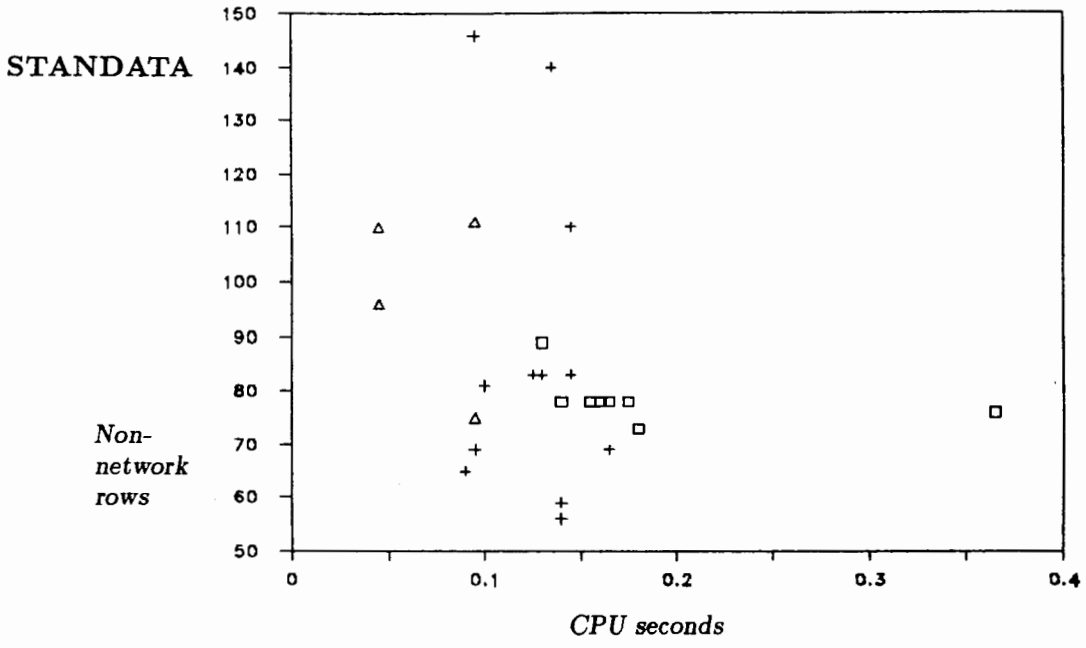


Figure 6–1. *Summary of embedded networks (continued).*



□ **Row-scanning deletion**
\+ **Column-scanning deletion**
△ **Row-scanning addition**

Figure 6–1. *Summary of embedded networks (continued).*

Appendix A. Test problems

The linear programs for our computational tests were taken from a variety of applications. All were supplied to us in the standard MPS form (Murtagh and Saunders 1983) and are currently available by electronic mail through NETLIB (Dongarra and Grosse 1985; Gay 1985).

ENERGY is one of many planning models developed at the Energy Information Administration, U.S. Department of Energy; it is identified as `80BAU8B` in the `NAME` line of the MPS file. It was supplied to us by William Kurator.

GIFFPINC and SIERRA were developed by Helgason, Kennington and Wong (1981) for national forest management. The underlying model is minimum-cost multicommodity flow, but GIFFPINC incorporates only a single commodity; SIERRA has two commodities.

GREENBEA and STANDATA are slightly modified versions of two production and blending models, originally used by an oil refining company. Numerous generalized upper bounds, specified in the original file for STANDATA, have been converted to explicit constraints.

PIES is a model of energy supply and distribution for the United States. It is a component of the Project Independence Evaluation System Integrating Model for energy policy studies (Hogan 1975).

SCAGR25 is a 25-period planning model for the expansion of a large dairy farm, developed by Swart, Smith and Holderby (1975).

SCRS8 is derived from a 16-period model of the United States' options for a transition from oil and gas to synthetic fuels. It was constructed by Ho (1977) based on a model by Manne (1975).

SHIP12L is a 12-commodity distribution model developed by Fourer for a producer of packaged baked goods. The non-network constraints derive from restrictions on production and transshipment.

Appendix B. Test procedures

Implementation. All of our programs are written in Fortran. They should run, with minor modifications, on any computer that has a Fortran 77 compiler.

We employed a separate preprocessing program to initially interpret the MPS file for each LP used in our tests. The preprocessor's output is an ASCII representation of the LP matrix, arranged so that it is easily read into the data structures described by Section 1. The row and column names

from the MPS form are written to a second file. We used this arrangement so that the MPS files would have to be read only once at the beginning of our tests; the preprocessor could be combined with the rest of our system for more routine use.

The main program is invoked by typing, say,

```
select scagr25
```

The model name following `select` identifies all files to be read or written; in this case they would begin with `scagr25`, and would end with a suffix that distinguishes their type.

The system is controlled interactively by making selections from a series of menus. For example, the following choices are presented immediately after `select scagr25` has been typed:

```
Input format:  [Matrix]  LP
```

The first (and default) option is to read a binary representation of the constraint matrix data structure (discussed further below); the second option reads the ASCII representation from the MPS form preprocessor. If the `Matrix` option is chosen, the next menu is

```
Restore statuses and scales?  [Yes]  No
```

which offers the option of reading a binary representation of status, scale and count vectors (as also discussed below). If the `LP` option is chosen, the next menu is instead

```
Reduce matrix:  [All]  Partial  Free-rows  None  Restore
```

The default option is to make all reductions described in Section 2; the next three provide for fewer or no reductions; and the last reads the aforementioned status, scale and count vectors. A one-line description of each option can be obtained by responding to any prompt with a question mark.

After the appropriate files have been read and reductions have been made, the system returns repeatedly to the following menu of major options:

```
SELECT: [Count]  Scale  Find  Write  Quit
```

Each option (except `Quit`) leads to one or more further levels of menus for a particular purpose. Naturally, `Scale` performs the scalings described in Section 2, and `Find` offers network-finding heuristics from Sections 3–5. `Count` provides various summary statistics.

The `Write` option can send a binary representation of the LP data structure to a file, and can create an accompanying file of status, scale and count vectors. These files can be read under option `LP` at the beginning of

subsequent runs, so that setup, reduction and scaling need not be repeated. Such a feature is most useful for testing different versions of the heuristics.

`Write` also offers the option of writing a new MPS file that represents the LP after reduction and scaling, and that flags all embedded network rows. This file can serve as input to systems that are designed to solve embedded-network linear programs.

Timings. All tests reported in this paper were run on a VAX 11/785 with floating-point hardware, under version 3 of the VMS operating system. Programs were written in Fortran 77 and were compiled under default options, including optimization.

Test runs were made at times of relatively light system load, with the operating system's working-set options specified as follows:

```
/Limit= 300   /Quota= 1024   /Extent= 1024   /Noadjust
```

Most timings were sensitive to these settings; in particular, the reported times would have been lower, but less reliable, if we had used `/Adjust` rather than `/Noadjust`.

The VAX processing times, obtained by calls to the VMS routine `LIB$STAT_TIMER`, are in multiples of .01 second. Typical variation of times from run to run was less than 5%. To reduce the variation somewhat, all timed runs of the network-finding heuristics were made twice; the reported times are averages of the results.

Every application of a network-finding heuristic was followed by a call to a utility routine that counted the network rows and columns. This routine also checked that each column contained at most one $+1$ and one -1 element within the network rows; thus we could be confident that the reported sizes of the network subsets were not overstated due to program bugs. Timings do not include this or other data-gathering routines. Most times do include a very small amount of data collection, such as would be involved in counting the number of passes in an algorithm. Where data-gathering within an algorithm was substantial, as in counting different states for column-scanning deletion or counting operations on the linked list for row-scanning deletion, we made both a timing run that did not collect data and a data-collection run that was not timed.

References

AHN, Kyu Ho, "Algorithms for Identification of Embedded Networks and Specialized Simplex Methods for Embedded-Network Linear Programs," Ph.D. dissertation, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL (1984).

BREARLY, A.L., G. Mitra and H.P. Williams, "Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm," *Mathematical Programming* 8 (1975) 54–83.

BROWN, Gerald G., Richard D. McBride and R. Kevin Wood, "Extracting Embedded Generalized Networks from Linear Programming Problems," *Mathematical Programming* 32 (1985) 11–31.

BROWN, Gerald G. and William G. Wright, "Automatic Identification of Embedded Structure in Large-Scale Optimization Models," in Harvey J. Greenberg and John S. Maybee (Eds.), *Computer-Assisted Analysis and Model Simplification*, Academic Press, New York (1981) 369–388.

BROWN, Gerald G. and William G. Wright, "Automatic Identification of Embedded Network Rows in Large-Scale Optimization Models," *Mathematical Programming* 29 (1984) 41–56.

CHEN, S. and R. Saigal, "A Primal Algorithm for Solving a Capacitated Network Flow Problem with Additional Linear Constraints," *Networks* 7 (1977) 59–79.

DONGARRA, Jack J. and Eric Grosse, "Distribution of Mathematical Software Via Electronic Mail," Technical Memorandum No. 48, Argonne National Laboratory, Argonne, IL (1985).

GAY, David M., "Electronic Mail Distribution of Linear Programming Test Problems," *Committee on Algorithms Newsletter* 13 (1985) 10–12. (Also Numerical Analysis Manuscript 86–0, AT&T Bell Laboratories, Murray Hill, NJ).

GLOVER, Fred and Darwin Klingman, "The Simplex SON Algorithm for LP/Embedded Network Problems," *Mathematical Programming Study* 15 (1981) 148–176.

GREENBERG, Harvey J., "A Tutorial on Matricial Packing," in Harvey J. Greenberg (Ed.), *Design and Implementation of Optimization Software*, Sijthoff and Noordhoff, Alphen aan den Rijn, The Netherlands (1978) 109–142.

GUPTA, Reeta, "Solving the Generalized Transportation Problem with Constraints," *Zeitschrift für Angewandte Mathematik und Mechanik* 58 (1978) 451–458.

HARTMAN, J.K. and L.S. Lasdon, "A Generalized Upper Bounding Algorithm for Multicommodity Network Flow Problems," *Networks* 1 (1972) 333–354.

HELGASON, R., J. Kennington and P. Wong, "An Application of Network Programming for National Forest Planning," Technical Report OR 81006, Department of Operations Research, Southern Methodist University, Dallas, TX (1981).

Ho, J.K., "Nested Decomposition of a Dynamic Energy Model," *Management Science* 23 (1977) 1022–1026.

HOGAN, William W., "Energy Policy Models for Project Independence," *Computers and Operations Research* 2 (1975) 251–271.

KNUTH, Donald E., *The Art of Computer Programming, volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA (1973).

MANNE, Alan S., "U.S. Options for a Transition for Oil and Gas to Synthetic Fuels," Discussion Paper 26D, Public Policy Program, Kennedy School of Government, Harvard University, Cambridge, MA (1975).

MCBRIDE, Richard D., "Solving Embedded Generalized Network Problems," *European Journal of Operational Research* 21 (1985) 82–92.

MURTAGH, Bruce A. and Michael A. Saunders, "MINOS 5.0 User's Guide," Technical Report SOL 83–20, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, CA (1983).

SENJU, Shizuo and Yoshiaki Toyoda, "An Approach to Linear Programming with 0-1 Variables," *Management Science* 15 (1968) B-196–B-207.

SWART, William, Calvin Smith and Thomas Holderby, "Expansion Planning for a Large Dairy Farm," in Harvey M. Salkin and Jahar Saha (Eds.), *Studies in Linear Programming*, American Elsevier, New York (1975) 163–182.

TARJAN, Robert Endre, "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the Association for Computing Machinery* 22 (1975) 215–225.