Check for updates

An Improved Primal Simplex Variant for Pure Processing Networks

MICHAEL D. CHANG and CHOU-HONG J. CHEN
Gonzaga University
and
MICHAEL ENGQUIST
Cleveland Consulting Associates

In processing networks, ordinary network constraints are supplemented by proportional flow restrictions on arcs entering or leaving some nodes. This paper describes a new primal partitioning algorithm for solving pure processing networks using a working basis of variable dimension. In testing against MPSX/370 on a class of randomly generated problems, a FORTRAN implementation of this algorithm was found to be an order-of-magnitude faster. Besides indicating the use of our methods in stand-alone fashion, the computational results also demonstrate the desirability of using these methods as a high-level module in a mathematical programming system.

Categories and Subject Descriptors: G.1.6 [**Numerical Analysis**]: Optimization—*linear programming*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*network problems*; G.4 [**Mathematics of Computing**]: Mathematical Software—*efficiency*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Embedded networks, mathematical programming systems, sparsity

1. INTRODUCTION

Pure processing networks are minimum-cost flow problems in which proportional flow restrictions are permitted on the arcs entering or leaving a node. Applications are widespread, with the proportional flow restrictions governing such things as the size of loan payments in cash flow models and the relation between raw materials and finished products in assembly models. A survey of applications is included in the work of Koene [16]. The proportional flow restrictions can be modeled either as nonnetwork rows or as nonnetwork columns in a linear programming (LP) formulation. Our approach uses nonnetwork columns since they lead to an LP basis with fewer rows. In [16], Koene shows that any LP problem can be readily transformed to a pure processing network problem at the

Authors' addresses: M. D. Chang and C.-H. J. Chen, School of Business Administration, Gonzaga University, Spokane, WA 99258; M. Engquist, Cleveland Consulting Associates, 3415 Greystone Drive, Suite 204, Austin, TX 78731.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0098-3500/89/0300-0064 $01.50

expense of enlarging the problem size. However, the primal partitioning methods of this paper will only be more efficient than standard LP methods when the basic nonnetwork columns form a small fraction of all basic columns. The allowable size of this fraction is yet to be determined.

The success of primal simplex solution procedures for solving pure networks is well known. These procedures depend on special data structures popularized by Glover and Klingman and their co-workers more than a decade ago [7, 8, 13]. A detailed description of network methods is given by Bradley et al. [2]. Standard primal simplex LP codes, however, use data structures for exploiting sparsity in the basis matrix [20, 22]. As shown in [8, 9, 25], specialized network codes have achieved an advantage in solution speed up to two orders of magnitude over standard LP codes.

Since LP problems often have a large network component, ways to exploit this component based on specialized network methods have been sought. This creates a need to reconcile the two data structure types. Two levels of detail for combining these data structures have been used. Glover et al. [12] describe a high-level approach in which a PL/I main program calls both a FORTRAN network code and MPSX/370 [14] modules in a complex solution procedure for a large nonlinear mixed integer program. McBride [18] and Glover and Klingman [10, 11] describe solution methods for embedded network problems in which basis partitioning allows both network and sparse matrix data structures that are tightly integrated in maintaining the basis. The specialized FORTRAN code of [18] ran about five times faster than MINOS [21]. In [11], three large problems, which included both nonnetwork rows and columns were solved using a FORTRAN embedded network code and MPSX/370. The nonnetwork rows make these problems more difficult than the test problems of the present paper, and MPSX/370 solved them about 4 percent faster than the specialized code. These efforts provide a start in delimiting the class of problems that benefit from tight integration of the two data structures.

Tomlin and Welch [25] describe a mathematical programming system written in assembly language that contains two optimizers, one based on network data structures and one based on sparse matrix data structures. Some modifications of network methods were made in order to accommodate the network optimizer into the system. This is a high-level approach since problems that are partly network do not benefit from the specialized network data structures. However, the two optimizers do have common I/O and start routines. For embedded network problems with nonnetwork rows, a two-stage starting procedure is described in which the network portion of the problem is first solved using the network optimizer. This solution is then used to provide a partial basis for the regular LP optimizer. Presumably this approach is superior to using only the regular LP optimizer and its start routine; however, comparative solution times are not provided in [25]. A similar approach was used in [11] where a FORTRAN network optimizer solved Lagrangian relaxations of the original problem iteratively to provide an initial partial basis for MPSX/370. This start procedure resulted in a total solution time that was 21 times smaller than that achieved when the MPSX/370 CRASH start procedure was used.

Chen and Engquist [5] described a primal simplex variant that is the precursor of the algorithm of this paper. One feature of the algorithm of [5] is that all

nonnetwork columns are always basic. This results in a working basis whose dimension is equal to the number of nonnetwork columns. When a FORTRAN implementation of the previous algorithm was tested against MINOS, it was found to be an order-of-magnitude faster on problems with up to 200 nonnetwork columns. The question remained as to whether a similar improvement in efficiency could be achieved against an assembly language code such as MPSX/370. In the present paper, we answer this question in the affirmative. We describe improvements to both the algorithm and implementation of [5]. In particular, the working basis for the current method has a dimension that varies with the number of basic nonnetwork columns. The approach we use involves a fairly tight integration of network and sparse matrix data structures. The design of our processing network code, PROCNET, was influenced by Marsten [17] in that it consists of a library of subroutines that communicate through parameter lists. Like Marsten's XMP code, PROCNET utilizes the LA05 subroutines of Reid [23, 24]. In fact, our extension of the LA05 subroutines may be beneficial to XMP users. This is discussed in Section 4. Although PROCNET is already highly efficient, further speedup is no doubt possible through a tighter integration of network and sparse matrix (LA05) data structures. It would also be of interest in future work to use PROCNET as a module in a high-level approach. For example, in large embedded network problems having both nonnetwork constraints and nonnetwork variables, the nonnetwork constraints could be initially relaxed. PROCNET could then be used to solve the remaining problem and thus provide a partial starting basis for an LP optimizer.

2. BACKGROUND ON PROCESSING NETWORKS

The pure processing network problem (problem PPN) is:

$$\text{minimize} \quad c_N x_N + c_P x_P \tag{2.1}$$

$$\text{subject to} \quad A_N x_N + A_P x_P = b \tag{2.2}$$

$$0 \le x_N \le h_N \tag{2.3}$$

$$0 \le x_P \le h_P. \tag{2.4}$$

The $m \times n$ matrix A_N is the node arc incidence matrix for a pure network N, while the $m \times p$ matrix A_P contains the nonnetwork columns. These nonnetwork columns are also called processing columns. Vector b contains the supply values, while c_N and c_P contain unit costs for the vectors of decision variables x_N and x_P. The capacities are the components of the simple upper bound vectors h_N and h_P. It is assumed, without loss of generality, that a slack arc and artificial arcs with Big-M costs have been introduced into the network N so that it is connected and the matrix A_N has rank m. It is also assumed that each row of $[A_N, A_P]$ contains at most one nonzero component from the columns of A_P. The latter assumption is made to simplify the notation, and it does not restrict the application of our methods. Each column of A_P is of the form

$$\begin{array}{ll} \alpha_{vv} & \text{in row } v \\ -\alpha_{vw(z)} & \text{in row } w(z), \quad z = 1, 2, \ldots, t \\ 0 & \text{elsewhere.} \end{array} \tag{2.5}$$

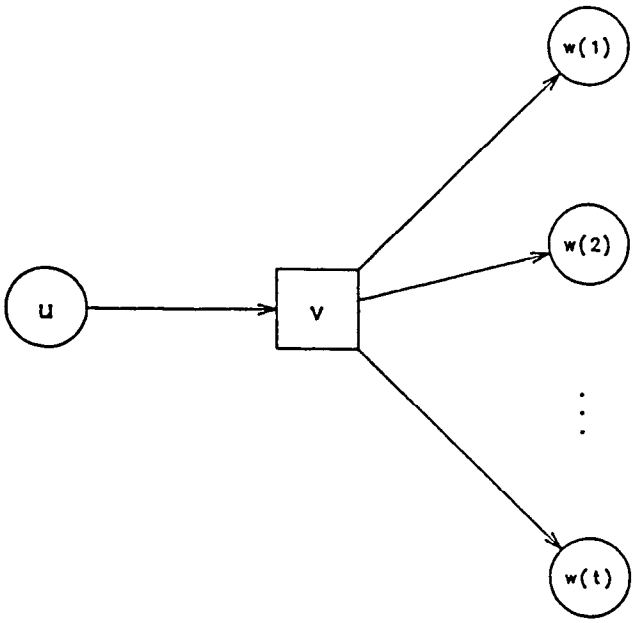

Fig. 1. A splitting node.

Further, $\alpha_{vv} = 1$, $0 < \alpha_{vw} < 1$, and

$$\sum_{z=1}^{t} \alpha_{vw(z)} = 1 \tag{2.6}$$

must hold.

Corresponding to each column of A_P is a column of A_N called an *allocation column*. The following network diagram is associated with this structure.

In Figure 1, the square is termed a splitting node, while the circles are ordinary network nodes. Arc (u, v) is called an allocation arc and its column in A_N is the allocation column. The arcs leaving the square node in Figure 1 are termed processing arcs, and they are represented by a column of the form (2.5). Node v and nodes $w(1), \ldots, w(t)$ in Figure 1 are called processing nodes. Conservation of flow and the conditions on the α_{vw} imply that a flow x entering the splitting node in Figure 1 generates a flow $\alpha_{vw(z)}x$ along processing arc $(v, w(z))$. To be consistent with (2.5) we assume that ordinary network arcs are represented by a column of A_N that contains a 1 in the row corresponding to the tail node of the arc and a -1 in the row corresponding to the head node. In Figure 1, if arc (u, v) corresponds to column r of A_N and the corresponding processing column (2.5) is column s of A_P, then it is assumed that the capacities $[h_N]_r$ and $[h_P]_s$ are equal.

In [16], the definition of pure processing networks includes the structure formed when the direction of the arcs in Figure 1 is reversed. For problems with finite capacities, by complementing flows with respect to these capacities and adjusting supply values appropriately, this structure can be reduced to the one shown in Figure 1. Thus there is no loss of generality in restricting attention to the structure of Figure 1.

In order to exploit the network structure of PPN, it is necessary to see how this structure carries over to a basis. The first observation to be made is that the slack arc column must be present in any basis matrix, otherwise the rows of the matrix would sum to zero. Next we note that when the processing columns and the slack column are removed from a basis matrix containing r processing columns, the remaining columns correspond to the arcs in $r + 1$ trees. This follows by a simple counting argument. These $r + 1$ trees are called *basis trees.* The basis tree that is incident to the problem's slack arc, when taken together with that arc, forms the basis quasitree. For the remaining r trees, root nodes are chosen arbitrarily and the resulting r rooted trees are called the rooted basis trees. We let a basis matrix B containing r processing columns be partitioned as

$$B = [B_1, B_2] \tag{2.7}$$

where B_1 contains network columns and B_2 contains the processing columns.

If the last r rows of B correspond to roots of rooted basis trees, then

$$B_1 = \begin{bmatrix} T \\ D \end{bmatrix} \quad \text{and} \quad B_2 = \begin{bmatrix} C \\ F \end{bmatrix} \tag{2.8}$$

results where T and C have $m - r$ rows and D and F have r rows. The working basis corresponding to basis B is defined to be the matrix Q where

$$Q = F - DT^{-1}C. \tag{2.9}$$

It can be shown that Q is nonsingular (see, e.g., [5, 15]).

The matrix T in (2.8) corresponds to a collection of quasitrees. By judicious use of matrices T and Q, updated entering columns and dual variables for the primal simplex algorithm can be computed without the need for maintaining a factorization of the entire basis matrix B. If

$$\tilde{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

is the entering column and it is partitioned to be compatible with (2.8), then a straightforward calculation shows that the updated entering column

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

is obtained by solving

$$Qy_2 = a_2 - DT^{-1}a_1 \tag{2.10}$$

$$y_1 = T^{-1}a_1 - T^{-1}Cy_2. \tag{2.11}$$

Similarly we partition the basic costs $c_B = [c_1, c_2]$ and the dual variables $\pi = [\pi_1, \pi_2]$ so that they are compatible with (2.8). The dual variables are computed by solving

$$\pi_2 Q = c_2 - c_1 T^{-1}C \tag{2.12}$$

$$\pi_1 = c_1 T^{-1} - \pi_2 DT^{-1}. \tag{2.13}$$

Suppose that column j of B_2 is the processing column with splitting node v. P_{ij} is defined to be the set of processing nodes in rooted basis tree i that correspond to nonzero values in column j of B_2. The following theorem was proved in [5].

THEOREM. *For a basis B, the elements q_{ij} of the working basis Q satisfy*

$$q_{ij} = \sum_{w \in P_{ij}} \alpha_{vw} \tag{2.14}$$

where the sum in (2.14) *is defined to be zero in case P_{ij} is empty.*

If the entering column corresponds to an arc with both end nodes in a common basis tree, it follows from (2.10) and (2.11) that the only flow changes occur on the cycle in the tree formed by the entering arc. Furthermore, the theorem implies that no working basis update is required in this case. This type of pivot is termed a *pure network pivot* while all other pivots are termed *processing network pivots.*

3. PRIMAL SIMPLEX VARIANT

The fundamental observation in the development of the primal simplex variant that we use relates to Figure 1. The flow on the allocation arc (u, v) must equal the value of the variable for the associated processing column. Thus, if the processing column is to be the leaving variable, the allocation arc can leave the basis instead. Note that the allocation arc must be basic in this situation, since otherwise the pivot would lead to the impossible situation in which both the allocation arc and the processing column are nonbasic.

Before stating the simplex algorithm, we outline the situations that are to be considered in updating the basis trees and the working basis Q during the basis exchange step. Before the basis exchange is executed, we assume that τ_0 is the basis quasitree and τ_i, $i = 1, 2, \ldots, r$ are the rooted basis trees. Those basis trees that have been changed during the exchange step will be designated by means of an asterisk. If a change to τ_i, $i \neq 0$, results in a change to one of the sets P_{ij} in (2.14), then row i of Q must be updated.

Several cases occur in the basis exchange step of the simplex algorithm. However, the variant we describe allows us to restrict attention to the two following cases.

(i) The entering column is a processing column and the leaving column is a network column (arc). If the leaving arc is in τ_i, then row i of Q will be updated (unless $i = 0$) and an additional row and column will be adjoined to Q.

(ii) Both the entering and leaving columns are network columns (arcs). If the entering arc is incident to τ_i and τ_j, then these two trees are joined to form τ_i^*. If the leaving arc is contained in τ_k, then τ_k splits into two trees upon its removal. One of these becomes τ_j^* and the other becomes τ_k^*. If i, j, and k are nonzero and distinct, then three rows of Q will be updated. Otherwise special cases occur in which at most two rows of Q are updated. One of these special cases is the pure network pivot for which no updating of rows of Q is necessary.

We remark that cases in which a processing column leaves the basis are not considered here, since the allocation arc can be chosen to leave instead.

The basis trees can be visualized as hanging downward from their roots. The node incident to the slack arc in the basis quasitree is taken as the root there. If two basis trees τ_i and τ_j are joined by an entering arc, the resulting τ_i^* will retain the root of τ_i while τ_j will hang below τ_i in τ_i^*. Also when a leaving arc is deleted from a basis tree τ_k, an upper tree τ_{k1} that contains the root of τ_k and a lower tree τ_{k2} are formed.

A starting PPN basis can be obtained by first setting the variables x_P in (2.1)–(2.4) to upper or lower bounds according to some heuristic procedure. These variables are set nonbasic in the PPN basis. Those x_P variables at upper bounds induce supplies in network N in addition to those represented by b in (2.2). The resulting network problem is solved to optimality to give the initial collection of basis trees which, in this case, consists of a single quasitree. Of course, this quasitree may contain artificial arcs with positive flow. An extension of this starting procedure has been implemented as described in Section 4.

We introduce the vector λ to represent certain quantities which may be thought of as pseudo node potentials.

$$\lambda = c_1 T^{-1}. \tag{3.1}$$

It will be useful to extend λ by defining $\lambda_j = 0$ for root nodes j of rooted basis trees. For convenience, this extension will also be denoted as λ.

3.1 Primal Simplex Algorithm for PPN

(0) Obtain an initial basis. Set up the initial basis trees and working basis. Compute initial dual variables and basic solution.

(1) Price nonbasic processing columns and arcs. If an entering processing column is found, go to Step 3. If an entering arc is found, go to Step 2. If no entering processing column or arc exists, check for basic artificial arcs with positive flow. If basic artificials with positive flow exist, stop with an infeasible problem; otherwise, stop with an optimal solution.

(2) If both end nodes of entering arc e are not in a common basis tree τ, go to Step 3. Otherwise, restrict the ratio test and flow update to the arcs on the cycle formed in τ by e. Update λ on the tree hanging below e after the leaving arc is removed. Go to Step 6.

(3) Compute y_1 and y_2 using (2.10) and (2.11).

(4) Perform the ratio test restricted to arcs. Update basic solution values.

(5) Update basis trees and working basis (basis exchange step). If an arc is entering the basis go to (ii).

  (i) A processing column enters the basis, and the leaving arc is in τ_k. The leaving arc is removed to form an upper tree τ_{k1} and a lower tree τ_{k2}. Tree τ_{k2} becomes τ_{r+1}, λ is updated on τ_{r+1}, and row $r+1$ of Q is created using (2.14). Column $r+1$ of Q corresponding to the entering column is also created using (2.14). Tree τ_{k1} becomes τ_k^* and row k of Q is updated (unless $k = 0$). Go to Step 6.

  (ii) An arc e enters the basis. (The details follow for this step when e is incident to τ_i and τ_j, the leaving arc is in τ_k, and i, j, k are nonzero and distinct. The remaining cases involve at most two rows of Q and the details are omitted.) First, τ_j hangs below τ_i via arc e to form τ_i^* and λ is updated on τ_j. Row i of Q is updated to form Q^*. Next the leaving arc is removed from τ_k to form an upper tree τ_{k1} and a lower tree τ_{k2}. The lower tree becomes τ_j^* and λ is updated on τ_j^*. Row j of Q^* is updated to form Q^{**}. Finally τ_{k1} becomes τ_k^* and row k of Q^{**} is updated to form Q^{***}.

(6) Update π_2 using (2.12). Compute π_1 using

$$\pi_1 = \lambda - \pi_2 D T^{-1}, \tag{3.2}$$

where λ has been previously updated. Go to Step 1.

In the above algorithm, processing columns are allowed to enter the basis, but not to leave. However, when the basis is reinverted, eligible processing columns are removed from the basis via a series of degenerate pivots. More details on this procedure are given in Section 4.

Updating of λ on a tree that is rehung is done by adding a certain constant to these λ values in the same way as node potentials are updated in the pure network case.

4. IMPLEMENTATION

A FORTRAN implementation, PROCNET, of the primal simplex algorithm of Section 3 was created. This version of PROCNET extends and enhances a previous version, which is described in [5]. Problem data storage in PROCNET is accomplished by means of arrays for arc costs, capacities, and head nodes. Also, arrays containing the nonzero values in processing columns and the positions of these values are used. The costs of processing columns, components of c_P in (2.1), are assumed to be zero, since such costs can be placed on the allocation arcs instead.

The basis trees are incorporated into a single, larger tree following [10, 11]. This tree is referred to as the master basis tree and its root is called the master root. The roots of all basis trees are connected to the master root by artificial arcs, and the slack arc of the basis quasitree is disregarded since it plays no role in the implementation. For maintaining the master basis tree, the predecessor, depth, thread, and reverse thread functions [2, 13, 15] are employed.

Since the updates to Q in Step 5 of the primal simplex algorithm involve more changes to rows than to columns, we have elected to maintain Q by applying column replacement techniques to its transpose Q^T.

Three LA05 subroutines were written in FORTRAN by Reid, and they are described in [23, 24]. These subroutines implement a sparse variant of the Bartels-Golub algorithm [1]. In order to utilize these subroutines for maintaining the working basis for PROCNET, we needed to extend them by providing a means for adjoining additional rows and columns. The two subroutines we created for this purpose are described in this section and we note that they may be useful in situations other than maintaining a working basis. For example, Marsten's XMP linear programming code [17] uses the original LA05 subroutines for maintaining the LP basis. Our additional subroutines can be used to provide a restart capability for XMP when one or more rows are adjoined to an LP problem.

We proceed with an explanation of the functions of the three original LA05 subroutines as applied to the $r \times r$ matrix Q^T in PROCNET.

The LA05A subroutine produces a factorization

$$Q^T = LU. \tag{4.1}$$

The lower triangular factor L is maintained as a sequence of eta vectors. The upper triangular factor U is stored as a sparse matrix with the nonzeros in the

rows held in packed form, while only the positions for nonzeros in columns are kept. Additional information that is maintained includes the pivot order and the number of nonzeros per row or column.

The LA05B subroutine solves sets of equations

$$Q^T\bar{x} = \bar{b} \tag{4.2}$$

and

$$Q\bar{x} = \bar{b} \tag{4.3}$$

where $\bar{x}$ and $\bar{b}$ are r-dimensional vectors. This solution is carried out with the use of (4.1).

The third subroutine, LA05C, revises the factorization (4.1) when one of the columns of Q^T is changed.

In order to accommodate additional rows and columns for Q^T in (4.1), we embed this matrix in a larger matrix $\bar{Q}^T$ where

$$\bar{Q}^T = \begin{bmatrix} I & 0 \\ 0 & Q^T \end{bmatrix} \tag{4.4}$$

and I is an identity matrix of dimension s. When an additional row d of Q^T of length $r + 1$ is created, it is embedded in a row of length $s + r$ containing $s - 1$ initial zeros as

$$[0, \ldots, 0, d] \tag{4.5}$$

and the row of (4.5) is inserted as row s of $\bar{Q}^T$ in (4.4). Likewise, an additional column of Q^T is supplied with $s - 1$ initial zeros and inserted as column s in (4.4).

A new subroutine LA05SS was created that takes the original factorization of Q^T from (4.1) as provided by LA05A and adjusts the information for storing L and U to produce the factorization

$$\bar{Q}^T = \bar{L}\bar{U} \tag{4.6}$$

where

$$\bar{L} = \begin{bmatrix} I & 0 \\ 0 & L \end{bmatrix} \tag{4.7}$$

and

$$\bar{U} = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix} \tag{4.8}$$

Essentially what is done is to change the pointers for rows and columns of U and to insert the nonzeros (ones) for the identity matrix. The rows acted on by the eta vectors of L must also be changed appropriately.

A new subroutine LA05TT carries out the task of inserting a new row (4.5) when this is required. We note that changes are needed for the data maintaining $\bar{U}$ but eta vectors corresponding to $\bar{L}$ remain correct. For insertion of a new column to $\bar{Q}^T$, LA05C is used.

PROCNET obtains an initial basis for PPN by means of a heuristic based on [3, 6]. In order to apply the heuristic, the arc data for each processing arc are

generated. The resulting pure network with proportional flow restrictions relaxed is solved first. Next the flow values of the relaxed solution on the allocation arcs are used to create a new pure network problem with nonzero lower bounds on the processing arcs. If the flow value on the allocation arc (u, v) of Figure 1 is x, then the lower bound on arc $(v, w(z))$ is set to $0.75\alpha_{vw(z)}x$. This second network problem is solved and the flows on allocation arcs are saved. If such flows are at the original problem bounds, the corresponding processing columns are made nonbasic while the allocation arc is basic. If an allocation arc has a flow between the original problem bounds, a parallel allocation arc is created with a capacity equal to this flow value. The parallel arc is given the same cost as the original allocation arc, while the original allocation arc is replaced by a modified allocation arc whose capacity equals the original capacity less the flow value. In the initial PPN basis, the modified allocation arc is nonbasic at zero, the parallel arc is nonbasic at capacity and the corresponding processing column is basic. The initial working basis Q is an $r \times r$ identity matrix where r is the number of parallel arcs created. The initial flows through allocation arcs and their parallel arcs induce supply values on the remainder of the network. This remaining network problem is solved to optimality and the resulting optimal tree becomes the initial PPN basis quasitree.

We note that an improvement to the way allocation arcs and their parallel arcs are handled has resulted in about a 10 percent decrease in the number of PPN iterations over the previous version of PROCNET [5]. This improvement is accomplished by reinstating the original allocation arc and eliminating the modified allocation arc and its parallel arc once one of the latter arcs enters the basis. In the previous version of the code both the modified allocation arc and its parallel arc were maintained for all PPN pivots, and this restricted the amount of flow change that could be achieved on a single pivot involving these arcs.

PROCNET uses two conditions to trigger a basis reinversion. The first of these conditions is a total of 40 column and row/column updates of Q^T. The other condition involves the dimension s of I in (4.8). After s processing columns have entered the basis, the next entering processing column causes a basis reinversion during which a new identity I is created. PROCNET currently uses a value of 10 for s. At the time of basis reinversion, PROCNET searches for basic processing columns having corresponding allocation arcs (see Figure 1), which are nonbasic. Before this basis reinversion takes place, a series of degenerate pivots is executed in which such processing columns are made nonbasic while their allocation arcs become basic.

Pricing for PROCNET is handled by means of two candidate lists, L1 and L2. L1 is used for pure network pivots while processing network pivots arising from either entering processing columns or arcs are placed on L2. In order to identify arcs that correspond to pure network pivots, basis trees are numbered. A node length array, treenum, assigns to each node of a given tree the number of that tree. If treenum values at end nodes of a pivot eligible arc agree, the arc is placed on L1. Otherwise, it is placed on L2. PROCNET repeats Step 2 of the primal simplex algorithm for all eligible arcs from L1 before updating π_2 in Step 6. The length of L1 was set at 50 and the length of L2 was set at 30. After all pivots from L1 have been made, up to 15 of the best pivots from L2 are made following the logic of [19].

Pure network pivots are accomplished following the same procedure used in a pure network algorithm. For processing network pivots, y_2 in (2.10) is computed using LA05B. If i denotes a processing column such that $[y_2]_i \neq 0$, then PROCNET flags basis trees containing processing nodes corresponding to column i. In computing y_1 by means of the reverse thread in (2.11), only trees that are flagged are traced. Since processing columns do not leave the basis until a reinversion occurs, only y_1 values are used in the ratio test.

All parameter settings for PROCNET mentioned in this section remained fixed at these values for the testing described in Section 5.

5. COMPUTATIONAL RESULTS

In this study, test problems were solved by MPSX/370 and PROCNET. Testing was done on the IBM 3081-D at the University of Texas. As previously noted, MPSX/370 is an assembly language program while PROCNET is written in FORTRAN. PROCNET was compiled using the FORTVS compiler with optimization level 3, and it maintains all real values using double precision. The execution times for both codes are reported in central processor seconds. These times do not include input or output with the exception that one line of output (iteration log) is produced by MPSX/370 each iteration. This small amount of output has a negligible effect on the overall comparison of the two codes.

The CRASH and PRIMAL modules of MPSX/370 were employed in solving the test problems. To be comparable with PROCNET, the reduced cost tolerance (XTOLDJ) of MPSX/370 was set to 10^{-5}. It was necessary to set the feasibility tolerance (XTOLV) to 10^{-4} since the default value of 10^{-5} kept MPSX/370 from reaching feasibility on the test problems. All other parameters for MPSX/370 were set to default values.

Parameters used by PROCNET, in addition to those previously discussed, are provided next. The reduced cost and feasibility tolerances were both set to 10^{-5}. Big-M costs on artificial arcs were set to 99999, while pivots with minimum ratio less than 10^{-10} were treated as degenerate. In LA05A and LA05C, pivot elements less than 0.1 times the largest element in the pivot row were excluded. Default values were used for other LA05 parameters.

The class of allocation/processing (AP) network problems was used for testing. These problems have a dual block angular form where subproblems corresponding to diagonal blocks are transportation problems and coupling columns are processing columns. This class of test problems was also used in [4, 5].

Problem data, with the exception of total supply, was randomly generated. All constraints were formulated as equalities. As the problems are generated, a feasible flow is created. The capacity of each arc with finite capacity is set to a parameter μ times the feasible flow generated for that arc. Total supply was set at 10,000 for all problems. Two cost ranges, A and B, were used. Cost range A has costs on allocation arcs in the range 100 to 150 and other arc costs in the range 1 to 100. Cost range B has allocation arc costs in the range 1 to 100 with other arc costs in the range -100 to -1.

Because a machine-dependent (CDC) random number generator was employed in the previous studies [4, 5], we were not able to include the problems from those studies here. The test problems solved are similar to those of [5]; however,

Table I. AP Problem Data

Problem	Finite capacity arcs	μ	Rows (m)	Columns ($n+p$)	Processing columns (p)	Nonzeros per proc. column	Cost range
1	allocation arcs	1.1	901	3,010	10	16	A
2	allocation arcs	2.0	901	3,010	10	16	A
3	allocation arcs	∞	901	3,010	10	16	A
4	all arcs	1.1–2.0	901	3,010	10	16	A
5	allocation arcs	1.1	2,001	5,050	50	6	A
6	allocation arcs	2.0	2,001	5,050	50	6	A
7	allocation arcs	∞	2,001	5,050	50	6	A
8	all arcs	1.1–2.0	2,001	5,050	50	6	A
9	allocation arcs	1.1	1,501	4,900	100	4	A
10	allocation arcs	2.0	1,501	4,900	100	4	A
11	allocation arcs	∞	1,501	4,900	100	4	A
12	all arcs	1.1–2.0	1,501	4,900	100	4	A
13	allocation arcs	1.1	2,251	5,550	150	4	A
14	allocation arcs	2.0	2,251	5,550	150	4	A
15	allocation arcs	∞	2,251	5,550	150	4	A
16	all arcs	1.1–2.0	2,251	5,550	150	4	A
17	allocation arcs	1.1	1,951	5,000	200	4	A
18	allocation arcs	2.0	1,951	5,000	200	4	A
19	allocation arcs	∞	1,951	5,000	200	4	A
20	all arcs	1.1–2.0	1,951	5,000	200	4	A
21	allocation arcs	1.1	1,801	4,750	250	4	A
22	allocation arcs	2.0	1,801	4,750	250	4	A
23	allocation arcs	∞	1,801	4,750	250	4	A
24	all arcs	1.1–2.0	1,801	4,750	250	4	A
25	allocation arcs	1.1	2,001	5,050	50	6	B
26	allocation arcs	2.0	2,001	5,050	50	6	B
27	allocation arcs	∞	2,001	5,050	50	6	B
28	all arcs	1.1–2.0	2,001	5,050	50	6	B

problems with more processing columns and problems with variable μ values are included here.

Test problem data are given in Table I. Problems in the groups 1–4, 5–8, ..., 25–28, have the same network topology. Also, groups 5–8 and 25–28 differ only in their cost ranges. For problems 4, 8, ..., 28, the value of μ was randomly selected from the interval [1.1, 2.0] for each arc. Computational results for these problems are reported in Table II. The count of iterations for both codes begins with the first pivot after the start (CRASH) procedure. The number of basic processing columns at optimality is obtained by PROCNET. This number provides an estimate of the dimension of the matrix Q^T near optimality. The ratio of total solution time for MPSX/370 to that of PROCNET is 11.46. Average time per iteration for MPSX/370 is 0.0638 sec/iteration while for PROCNET it is 0.0283 sec/iteration. The ratio of these average times per iteration is about 2.25. The larger ratio for total solution time can be attributed to the superiority of the PROCNET start procedure and pivot strategy over that of MPSX/370 on the problems tested.

The efficacy of the PROCNET start procedure and pivot strategy is strongly dependent on μ as shown in Table III. The ratio of total MPSX/370 solution

Table II. Computational Results for AP Problems

Problem	PROCNET start time	PROCNET total time	PROCNET iterations	MPSX/370 CRASH time	MPSX/370 total time	MPSX/370 iterations	Basic proc. columns at optimality
1	2.2	5.4	292	7.2	61.2	2,107	10
2	1.8	15.6	1,259	7.2	128.4	3,225	10
3	1.4	18.8	1,537	7.8	116.4	2,755	10
4	2.3	21.3	2,004	6.6	205.8	5,722	10
5	5.5	23.2	604	21.0	557.4	8,063	24
6	3.9	37.3	1,059	20.4	928.6	12,144	47
7	3.5	48.2	1,531	21.6	765.0	9,929	50
8	4.4	59.9	2,038	13.2	1,234.8	16,827	47
9	3.4	9.4	306	16.8	103.2	2,727	77
10	2.9	21.8	1,041	16.8	151.2	3,341	77
11	2.5	24.3	1,179	16.8	141.6	3,008	99
12	4.4	32.1	1,800	11.4	114.6	6,230	98
13	4.8	14.7	324	24.6	166.8	3,322	150
14	4.3	31.7	916	27.0	239.4	3,847	150
15	3.6	32.3	984	26.4	211.2	3,350	150
16	5.6	43.1	1,706	17.4	297.6	5,658	150
17	9.6	37.1	816	24.0	608.4	8,760	71
18	6.4	59.1	1,248	23.4	786.6	10,953	182
19	5.0	68.0	1,565	23.4	656.4	8,850	196
20	8.3	83.7	1,829	13.2	1,239.0	16,885	187
21	9.6	37.3	818	24.6	457.8	8,078	71
22	6.3	71.3	1,562	25.8	735.0	11,054	212
23	4.8	92.0	2,110	29.4	645.0	9,163	238
24	9.2	120.8	2,606	14.4	1,240.8	17,794	227
25	5.3	20.9	550	21.0	561.0	8,480	15
26	2.8	78.4	2,636	21.0	798.0	10,229	40
27	2.5	101.0	3,562	20.4	715.8	9,231	48
28	3.8	93.3	3,498	13.8	1,057.8	14,097	46
TOTAL	130.1	1,302.0	41,380	516.6	14,925.0	225,829	

Table III. Performance Values versus μ

Problems	μ	PROCNET % pure network pivots	PROCNET avg. pivot time	MPSX/370 avg. pivot time	MPSX/370 to avg. pivot time ratio	PROCNET total time ratio
1, 5, . . . , 25	1.1	17.3	0.0290	0.0572	1.97	17.00
2, 6, . . . , 26	2.0	20.3	0.0295	0.0662	2.24	11.95
3, 7, . . . , 27	∞	21.3	0.0290	0.0671	2.31	8.45
4, 8, . . . , 28	1.1–2.0	28.6	0.0269	0.0637	2.37	11.87

time to that of PROCNET decreases from 17.00 for $\mu = 1.1$ down to 8.45 for $\mu = \infty$. These results show that PROCNET is especially effective on tightly capacitated problems. Average pivot times for the processing network code are remarkably stable as μ varies. Apparently, the tendency to a smaller number of basic processing columns at optimality when $\mu = 1.1$ is offset by a smaller percentage of pure network pivots. On the other hand, it seems that the larger percentage of pure network pivots for PROCNET on problems 4, 8, . . . , 28 results in a somewhat smaller average pivot time.

Table IV. Performance Values versus Number of Processing Columns

Problems	Process columns (p)	PROCNET % pure network pivots	PROCNET avg. pivot time	MPSX/370 avg. pivot time	MPSX/370 to avg. pivot time ratio	PROCNET total time ratio
1–4	10	42.5	0.0105	0.0350	3.33	8.38
5–8, 25–28	50	21.6	0.0278	0.0727	2.62	14.32
9–12	100	36.3	0.0172	0.0293	1.70	5.83
13–16	150	31.1	0.0263	0.0507	1.93	7.51
17–20	200	10.2	0.0401	0.0706	1.76	13.27
21–24	250	12.0	0.0411	0.0648	1.58	9.58

In Table IV, variation in code performance with the number of processing columns p is given. Except for $p = 50$, average pivot times for PROCNET increase with p while the percentage of pure network pivots is roughly decreasing. More nonzeros per processing column is perhaps the major factor that makes the problems with $p = 50$ difficult for MPSX/370. We note that for problems 5–8, the MPSX/370 to PROCNET total time ratio is 20.7. The problems with $p = 100$ and $p = 150$ are easier than problems with larger p values for both codes, although MPSX/370 to PROCNET total time ratios are down. A possible explanation is that the transportation subproblems have a somewhat different structure. This consists of fewer origins, more destinations, and more arcs per origin than those with larger p values. Also, when p equals 100, the number of problem rows is lower. From the results of Table IV, we conclude that PROCNET remains very efficient for problems with up to 250 processing columns.

6. CONCLUSIONS

The new version of our pure processing network code, PROCNET, derives part of its efficiency from a working basis of variable dimension. An extension of the LA05 subroutines has been described, and this extension is used in PROCNET to maintain the working basis. PROCNET substantially outperforms MPSX/370 in both time per pivot and total solution time on problems with up to 250 processing columns. The faster time per pivot of PROCNET indicates that it will be useful as part of a mathematical programming system, while the faster overall solution time shows that it can be used to advantage in stand-alone fashion. Since PROCNET is an all-FORTRAN code, it is highly portable as well.

REFERENCES

1. BARTELS, R., AND GOLUB, G. The simplex method of linear programming using LU decomposition. *Commun. ACM 12*, 5 (May 1969), 266–268.
2. BRADLEY, G., BROWN, G., AND GRAVES, G. Design and implementation of large scale primal transshipment algorithms. *Manage. Sci. 24*, 1 (Sept. 1977), 1–34.
3. CHARNES, A., COOPER, W., DIVINE, D., HINKEL, W., KONING, J., AND LOVEGREN, V. A seashore rotation goal programming model for Navy use. Res. Rep. CCS 429, Center for Cybernetic Studies, Univ. of Texas, Austin, 1982.
4. CHEN, C.-H., AND ENGQUIST, M. Computational comparison of two solution procedures for allocation processing networks. *Math. Program. Stud. 26* (1986), 218–220.
5. CHEN, C.-H., AND ENGQUIST, M. A primal simplex approach to pure processing networks. *Manage. Sci. 32*, 12 (Dec. 1986), 1582–1598.

6. GLOVER, F., GLOVER, R., AND MARTINSON, F. A netform system for resource planning in the U.S. Bureau of Land Management. *J. Oper. Res. Soc. 35*, 7 (July 1984), 605–616.
7. GLOVER, F., KARNEY, D., AND KLINGMAN, D. Implementation and computational comparisons of primal, dual, and primal-dual computer codes for minimum cost network flow problems. *Networks 4*, 3 (1974), 191–212.
8. GLOVER, F., KARNEY, D., KLINGMAN, D., AND NAPIER, A. A computational study on start procedures, basis change criteria, and solution algorithms for transportation problems. *Manage. Sci. 20*, 5 (Jan. 1974), 793–813.
9. GLOVER, F., AND KLINGMAN, D. Capsule view of future developments of large-scale network and network-related problems. Res. Rep. CCS 238, Center for Cybernetic Studies, Univ. of Texas, Austin, 1975.
10. GLOVER, F., AND KLINGMAN, D. The simplex SON algorithm for LP/embedded network problems. *Math. Program.. Stud. 15* (1981), 148–176.
11. GLOVER, F., AND KLINGMAN, D. Basis change characterizations for the simplex SON algorithm for LP/embedded networks. *Math. Program. Stud. 24* (1985), 141–157.
12. GLOVER, F., KLINGMAN, D., PHILLIPS, N., AND ROSS, G. Integrating modeling, algorithm design, and computational implementation to solve a large-scale nonlinear mixed integer programming problem. *Ann. Oper. Res. 5* (1985/6), 395–411.
13. GLOVER, F., KLINGMAN, D., AND STUTZ, J. Augmented threaded index method for network optimization. *INFOR 12*, 3 (Oct. 1974), 293–298.
14. *IBM Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual, 4th Edition.* IBM Corp., Technical Publications Dept., White Plains, N.Y., 1979.
15. KENNINGTON, J., AND HELGASON, R. *Algorithms for Network Programming*, John Wiley, New York, 1980.
16. KOENE, J. Minimal cost flow in processing networks: A primal approach. Ph.D. dissertation, Eindhoven Univ. of Technology, Eindhoven, The Netherlands, 1982.
17. MARSTEN, R. The design of the XMP linear programming library. *ACM Trans. Math. Softw. 7*, 4 (Dec. 1981), 481–497.
18. MCBRIDE, R. Solving embedded generalized network problems. *European J. Oper. Res. 21*, 1 (July 1985), 82–92.
19. MULVEY, J. Pivot strategies for primal-simplex network codes. *J. ACM 25*, 2 (Apr. 1978), 266–270.
20. MURTAGH, B. *Advanced Linear Programming: Computation and Practice.* McGraw-Hill, New York, 1981.
21. MURTAGH, B., AND SAUNDERS, M. Large scale linearly constrained optimization. *Math. Program. 14*, 1 (Jan. 1978), 41–72.
22. ORCHARD-HAYS, W. *Advanced Linear Programming Computing Techniques.* McGraw-Hill, New York, 1968.
23. REID, J. FORTRAN subroutines for handling sparse linear programming bases. Rep. AERE–R8269, Computer Science and Systems Div., AERE Harwell, Oxfordshire, England, 1976.
24. REID, J. A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases. *Math. Program 24*, 1 (Sept. 1982), 55–69.
25. TOMLIN, J., AND WELCH, J. Integration of a primal simplex network algorithm with a large-scale mathematical programming system. *ACM Trans. Math. Softw. 11*, 1 (Mar. 1985), 1–11.

Received November 1986; revised May 1988; accepted September 1988