ENHANCEMENTS OF SPANNING TREE LABELLING PROCEDURES FOR NETWORK OPTIMIZATION*

RICHARD BARR

Southern Methodist University, Dallas, Texas

FRED GLOVER

University of Colorado, Boulder, Colorado

DARWIN KLINGMAN

University of Texas at Austin, Texas

Abstract

New labelling techniques are provided for accelerating the basis exchange step of specialized linear programming methods for network problems. Computational results are presented which show that these techniques substantially reduce the amount of computation involved in updating operations.

Résumé

Des techniques nouvelles d'étiqueter sont pourvues à accélérer le démarche du base échange des méthodes du simplexe specialisées pour les problèmes du réseau. Les résultats computationeaux sont presentés qui montent que ces techniques ameliorent l'efficience des opérations de revision par un facteur de deux.

1 Introduction

In solving minimum cost flow network problems by specialized simplex methods, an important question is: How can one update the spanning tree basis with the least amount of effort? A partial answer to this question is provided by special list structure techniques such as the API method[6] and the more recent ATI method,[9] which have contributed dramatically to improving the efficiency of network algorithms (see, e.g., references (3, 5, 6, 7, 11)). This paper addresses the issue of which supplemental techniques can be used to implement these list structures (and particularly the ATI method) with greater efficiency.

As shown in reference (6), the major updating calculations of a basis exchange step can be restricted to just one of the two subtrees created by dropping the outgoing arc. Consequently, a natural goal is to identify the

*Received 10 October 1975; revised 3 February 1977 and 31 May 1978.

smaller of these two subtrees by means of a function $t(x)$ that names the number of nodes in the subtree "headed by node x." A clever and rather intricate procedure for doing this was proposed by Srinivasan and Thompson.[13] Unfortunately, this procedure requires sorting the nodes of the subtree by their distances from the root, and then further entails a full subtree update of both the distance values and the $t(x)$ values at each basis exchange step. Because of the substantial amount of work required to update the $t(x)$ list, the advantages of using this list have been largely offset by the computational costs involved in its maintenance, and the potential of the original Srinivasan-Thompson proposal has not been fully realized.

The purpose of this paper is to propose a new type of relabelling scheme that succeeds in updating $t(x)$ without sorting. In fact, this scheme requires even less work than updating the distance values of reference (13). The relabelling is based on "absorbing" $t(x)$ into the updating calculations of the ATI method. Moreover, these calculations are carried out simultaneously with the procedures[9] for updating other changes introduced by the basis exchange step.

To achieve the integration of the ATI calculations and the update of $t(x)$, an index function $f(x)$ is introduced which names the last node in the subtree rooted at x. Additionally, it is shown that $f(x)$ makes it possible to streamline the ATI calculations. Finally, as a bonus, it is shown that $t(x)$ can accommodate all of the relevant functions filled by the distance values, and hence can replace these values. Computational results presented in the last section indicate that the net gains of all these advantages produce a procedure that is approximately twice as fast for implementing the basis exchange operations.

2 Background

2.1 *Problem definition*

The *capacitated minimum cost flow network* or *capacitated transshipment* problem can be stated as follows:

Problem 1

Minimize

$$\sum_{(i,j)\in A} c_{ij}x_{ij}, \tag{1}$$

subject to:

$$\sum_{i\in I_k} x_{ik} - \sum_{j\in O_k} x_{kj} = b_k, k \in N \tag{2}$$

$$0 \leqslant x_{ij} \leqslant u_{ij} \text{ for } (i,j) \in A, \tag{3}$$

where $I_k = \{i \in N: (i, k) \in A\}$ and $O_k = \{j \in N: (k, j) \in A\}$. In standard terminology A is the set of arcs (i, j) of the network $G(N, A)$ and N is the set of nodes. The constant b_k represents the "requirement" at node k, which is frequently referred to as the supply if $b_k < 0$ and as the demand if $b_k > 0$. Associated with each node $k \in N$ is a dual variable π_k called its *node potential*. An arc (i, j) is directed from node i to node j. An arc (i, j) is also said to be out-directed from node i and in-directed to node j. Thus, in particular, I_k is the set of tail nodes of arcs that are in-directed to node k, and O_k is the set of head nodes of arcs that are out-directed from node k.

The flow, cost, and upper bound of arc (i, j) are represented, respectively, by x_{ij}, c_{ij}, and u_{ij}. In the terminology of simplex solution algorithms for networks, the reduced cost of arc $(i, j) \in A$ is $\bar{c}_{ij} = c_{ij} + \pi_i - \pi_j$. The objective is to determine a set of arc flows which satisfies the node requirements and capacity restrictions at minimum total cost.

2.2 *Graphical structure of network bases*

A bounded variable simplex basis for a network flow problem corresponds to a spanning tree with $n - 1$ arcs (where n denotes the cardinality of the set N). An arc is called basic if it is contained among those arcs in the basis tree and is called non-basic otherwise. Each non-basic arc has a flow equal to zero or to its upper bound.

Once the flows on the non-basic arcs have been set, the flows on the basic arcs are uniquely assigned so that equation (2) is satisfied. If equation (3) is also satisfied, this assignment of flows is a basic feasible solution. For each basis, node potentials are assigned values that satisfy complementary slackness; that is, these node potential values are determined so that the reduced cost for each basic arc is zero.

2.3 *Representation of a spanning tree*

The most efficient procedures for solving network flow problems are based on storing the basis as a *rooted tree*. It will be assumed that a basis tree with n nodes and $n - 1$ arcs is known and has been rooted. The root node will be regarded as the "highest" node in the rooted tree, with all other nodes hanging below it. If nodes i and j denote endpoints of a common arc in the rooted tree such that node i is closest to the root, then i is called the *predecessor* of node j and node j is called an *immediate successor* of node i.

The following notational conventions will be used to identify the components of the basis exchange step:

$[p, q]$ = the link between node p and q leaving the basis tree.

$[u, v]$ = the link entering the basis tree where v is the node whose unique path to the root node contains $[p, q]$.

T = the basis tree.

$T(x)$ = the subtree of T that is rooted at node x (hence the subtree that includes x and all its successors under the predecessor ordering).

$p(x)$ = the predecessor of node x where $p(x) = 0$ if node x is the root node.

$s(x)$ = the "thread successor" of x.

Intuitively, function s may be thought of as a thread which passes through each node exactly once in a top to bottom, left to right order starting from the root node.

More precisely the function s satisfies the following inductive characteristics:

(a) Letting 1 denote the root node, the set $\{1, s(1), s^2(1), \ldots s^{n-1}(1)\}$ is precisely the set of nodes of the rooted tree where $s^2(1) = s(s(1))$, $s^3 = s(s^2(1))$, etc. The nodes 1, $s(1)$, ... $s^{k-1}(1)$, will be called the *antecedents* of node $s^k(1)$.

(b) For each node i other than node $s^{n-1}(1)$, $s(i)$ is one of the immediate successors of node i, if i has a successor. Otherwise, $s(i)$ is an immediate successor of the closest predecessor of node i, say node x, such that node x has an immediate successor which is not an antecedent of node i.

(c) $s^n(1) = 1$; that is, the last node of the tree threads back to the root node.

By virtue of the foregoing characterization, the set of nodes of $T(x)$ is $\{x, s^1(x), \ldots, s^k(x)\}$ where k is the largest number such that $p(s^k(x))$ is one of the nodes $x, s^1(x), \ldots, s^{k-1}(x)$. By convention $s^1(x) = s(x)$ and $s^0(x) = x$.

$t(x)$ = the number of nodes in $T(x)$.

$f(x)$ = the "last node," $s^r(x)$, of the nodes in $T(x)$, where $r = t(x) - 1$.

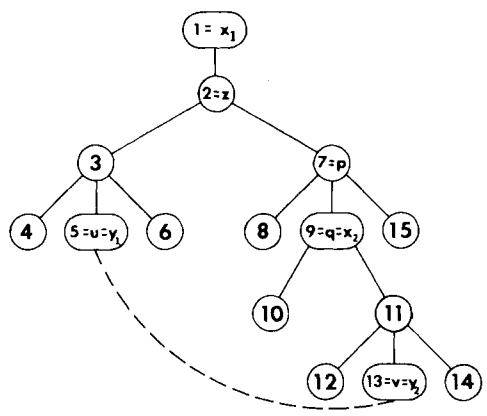
Figure 1 illustrates the above functions as follows. The NODE array specifies the node names. The entries in the arrays p, s, f, and t parallel to a node name specify the values of the functions p, s, f, and t for that node name. Note that the links in Figure 1 indicate the existence of a basis arc, but not the direction of the arc in problem 1.

2.4 *Summary of the network simplex method*

For completeness, the steps of the simplex method specialized to this framework are summarized as follows.

Initialization: Determine Node Potentials

Assume that an initial feasible basis (possibly containing artificial arcs) has been determined and stored as a rooted tree. It is then necessary to determine node potentials π_k for each node k so that the reduced cost



NODE	FUNCTIONS p	s	f	t
1	0	2	15	15
2	1	3	15	14
3	2	4	6	4
4	3	5	4	1
5	3	6	5	1
6	3	7	6	1
7	2	8	15	9
8	7	9	8	1
9	7	10	14	6
10	9	11	10	1
11	9	12	14	4
12	11	13	12	1
13	11	14	13	1
14	11	15	14	1
15	7	1	15	1

FIG. 1. Initial tree and function values.

equals zero for each basic arc (i, j). Because of the redundancy in the defining equations of the network problem, one node potential can be set arbitrarily. Customarily, the node potential of the root is initially set to zero. The remaining node potentials can be determined in a cascading fashion by moving down the tree and computing the potential for node j from the potential previously set for its predecessor, node i, and from the equation

$$c_{ij} + \pi_i - \pi_j = 0,$$

if the basic arc connecting nodes i and j is directed from node i to node j in problem 1, or from

$$c_{ji} + \pi_j - \pi_i = 0,$$

if the basic arc is directed instead from node j to node i in problem 1.

Step 1: Identify the outgoing and incoming arcs.
The fundamental basis exchange step of the simplex method selects an

incoming arc and an outgoing arc from the set of non-basic and basic arcs, respectively.

(a) The incoming arc: A non-basic arc which is profitable (i.e., has zero flow and a negative reduced cost or has saturating (upper bound) flow and a positive reduced cost) is selected to enter the basis. If no such arc exists, the problem is solved. (In this case, the solution is feasible and the current arc flows are optimal if no artificial arcs with positive flow exist: otherwise the problem is infeasible.)

(b) The outgoing arc: The arc to leave the basis is determined by tracing the unique basis path which connects the two nodes of the incoming arc. This *basis equivalent path* can be determined by tracing the predecessors of the two nodes to their initial point of intersection. An attempted change of the flow of the incoming arc in its profitable direction (away from the bound it currently equals) causes a change in the flows of the arcs contained in this basis equivalent path. In order to maintain primal feasibility, the outgoing arc must always be an arc in this path whose flow goes to zero or its upper bound ahead of (or at least as soon as) any others as a result of a flow change in the incoming arc. Such an arc is called a *blocking arc*. (This arc can, of course, be the same as the incoming arc in the capacitated problem.)

Step 2: Execute the basis exchange.

The outgoing and incoming arcs swap their basic/non-basic statuses to become non-basic and basic, respectively; the basis tree functions, basis flows, and node potentials are then updated and the method returns to Step 1 with a new primal feasible basis.

To illustrate, consider figure 1 and assume that link [5, 13] has been selected to enter the basis. The basis equivalent path for link [5, 13] is the links in the predecessor path of the basis tree from node 5 to node 2 and from node 13 to node 2. Node 2 is referred to as the intersection node.

The purpose of this paper is to show how the functions p, s, f, and t can be efficiently updated and how these functions can be used to find the basis equivalent path and used to update the node potentials.

Before proceeding, it may be helpful to review how the node potentials can be updated from basis to basis. Observe that when $[p, q]$ is deleted from the basis tree, the tree splits into two subtrees. It is only necessary to change the node potentials associated with one of these subtrees (preferably the subtree containing the fewest number of nodes) and these potentials can be updated by simply adding or subtracting δ, the reduced cost of the entering arc, to each node potential in the selected subtree. One adds the value δ if the selected subtree contains the "from" node of the entering arc and subtracts δ otherwise. (For a more complete description see reference (6).)

To illustrate, consider figure 1 and assume that link $[u, v]$ is directed from u to v and let δ_{uv} denote its reduced cost. Updated node potentials can be obtained by subtracting δ_{uv} from each node potential in $T(q)$ (i.e., nodes 9, 10, 11, 12, 13, and 14) or adding δ_{uv} to each node potential in $T - T(q)$ (i.e., nodes 1, 2, 3, 4, 5, 6, 7, 8, and 15).

2.5 *Additional notation*

The basis exchange step may be visualized as consisting of two components:

(1) Dropping link $[p, q]$ to create two independent subtrees: $T(q)$ and $T - T(q)$ (where the latter is the subtree of T that excludes $T(q)$ and all its nodes, and hence which excludes the "connecting link" $[p, q]$);

(2) Adding link $[u, v]$ to create a single new basis tree.

The subtrees $T(q)$ and $T - T(q)$ can be viewed as any two node-disjoint trees which are to be joined by an arc to create a new basis tree. Updating operations will be developed first to make $T(q)$ and $T - T(q)$ into label "independent" trees in preparation for selecting one to be the new "upper tree" (called T_1) rooted at x_1 and the other to be the new "lower tree" (called T_2) rooted at x_2. It may then be assumed that the root of T_1 becomes the root of the new basis tree. Additionally, $[y_1, y_2]$ will be used to denote the link that joins T_1 and T_2 where y_1 is a node of T_1 and y_2 is a node of T_2. Next, operations will be developed to re-root T_2 at y_2 in preparation for attaching T_2 to T_1 via link $[y_1, y_2]$ to create the new basis tree. (There is no requirement that y_1 be distinct from x_1 or that y_2 be distinct from x_2.)

3 Performing the Basis Exchange

3.1 *Finding the basis equivalent path*

In the simplex method an improving non-basic arc is selected to enter the basis if the current basis is not optimal. Once this arc has been selected, it is necessary to locate its basis equivalent path. The arc leaving the basis always lies in the basis equivalent path and corresponds to a blocking arc. The following procedure efficiently locates the basis equivalent path and if the flows, lower bounds, and upper bounds on each basic arc are known, simultaneously determines the arc to leave the basis. Specifically, this loop can be located as follows:

(i) Set $x_u = u$ and $x_v = v$.

(ii) If $t(x_u) < t(x_v)$ go to step v.

(iii) If $t(x_u) > t(x_v)$ go to step vi.

(iv) If $x_u \neq x_v$, go to step v. Otherwise set $z = x_u$ and stop. The loop created by adding $[u, v]$ has been traversed and $x_u = x_v = z$ is the unique node of this loop which lies on the predecessor paths from both u and v

to the root node of the tree. Node z will henceforth be called the intersection node.

(v) Search the predecessor path from x_u to the root of T for a node x such that $t(x) \geqslant t(x_v)$, set $x_u = x$, and go to step (iii).

(vi) Search the predecessor path from x_v to the root of T for a node x such that $t(x) \geqslant t(x_u)$ and set $x_v = x$. If $t(x_u) = t(x_v)$, go to (iv); otherwise go to step (v).

Graphically, the above algorithm implicitly traverses the predecessor path from u to the root of T and the predecessor path from v to the root of T in the most "node efficient" manner to locate the basis equivalent path and the intersection node z. The intersection node possesses the following obvious properties: (1) it will have the same $t(x)$ value on both paths and (2) it will have the same name on both paths. The algorithm avoids checking two nodes for intersection unless condition (1) above holds true. For example, in figure 1 the algorithm will bypass nodes 9 and 7, but it will check node 5 against node 13, then node 3 against node 11, and finally node 2 against node 2. Any nodes above the intersection are ignored.

3.2 *Updating operations*

Using the preceding definitions, rules will be given to find updated values $p^*(x)$, $s^*(x)$, $t^*(x)$, $f^*(x)$ for $p(x)$, $s(x)$, $t(x)$, and $f(x)$, respectively.

For notational ease, we also make use of the "reverse thread" function r, which has the property such that $s(r(x)) = x$. The values of $r(x)$ need not be maintained and may be calculated as needed by using the s function to trace the thread from $p(x)$. For a more efficient method, first let $y = p(x)$. Second, if $s(y) = x$, stop and let $r(x) = y$. Otherwise, let $y = f(s(y))$ and return to the second step.

I *Update $p(x)$, $s(x)$, and $f(x)$ to make the subtree $T(q)$ and $T - T(q)$ independent.*

The reader may find it helpful to perform the following operations using figure 1.

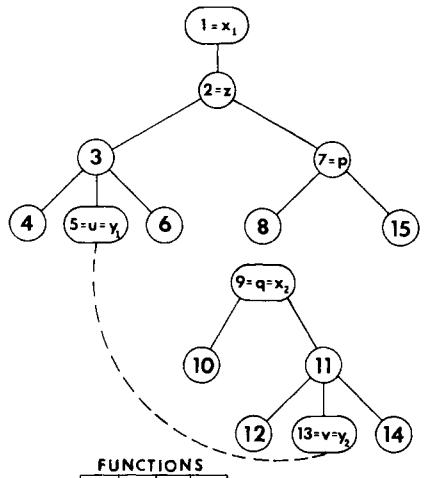
I.1 *Update for $T - T(q)$:*

For $p^*(x)$: No updating of any $p(x)$ is required.

For $s^*(x)$: Set $s^*(r(q)) = s(f(q))$. No other $s(x)$ values are changed.

For $f^*(x)$: Let $x^* = p(s(f(q)))$. If $x^* = 0$, then set x^* equal to the root node index. Set $f^*(x) = r(x)$ (i.e., the node y determined in updating s above) for those nodes x on the path from p to x^* excluding x^* itself if $p(s(f(q))) \neq 0$. (If $x^* = p$, then no updating is done unless x^* is the root node.)

For $t^*(x)$: Set $t^*(x) = t(x) - t(q)$ for those nodes x on the path from p to the root of T. It is important to note that, due to cancellation effects of



NODE	FUNCTIONS p	s	f	t
1	0	2	15	9*
2	1	3	15	8*
3	2	4	6	4
4	3	5	4	1
5	3	6	5	1
6	3	7	6	1
7	2	8	15	3*
8	7	15*	8	1
9	0*	10	14	6
10	9	11	10	1
11	9	12	14	4
12	11	13	12	1
13	11	14	13	1
14	11	9*	14	1
15	7	1	15	1

*indicates values that changed during Step I

FIG. 2. Tree and function values after performing step I.

subsequent calculations, this step can be restricted to the partial backward path from p to the intersection node z found in section 3.1.

Also observe that the updating of the t and f function values can be integrated in the tracing of the path from p to the root of T.

I.2 *Update for* $T(q)$:

For $p^*(x)$: Set $p(q) = 0$

For $s^*(x)$: Set $s^*(f(q)) = q$.

No other updating of any $p(x)$, $s(x)$, $t(x)$, or $f(x)$ values is required for $T(q)$.

Figure 2 illustrates the updated values after performing Step I on figure 1.

II *Decide which of* $T(q)$ *and* $T - T(q)$ *is to be* T_1 *and which is to be* T_2. The rule to use for picking which subtree to re-root depends heavily on when the dual variables are updated. As noted earlier, the updating of

the dual variables involves adding or subtracting a constant from each node potential in one of the subtrees. Further, as will be seen in Step III, updating the thread function involves visiting some of the nodes in the subtree to be re-rooted. Thus these two operations could be integrated.

If the updating of the dual variables is not integrated with the other updating operations, it will be clear from the operations performed in Step III that the subtree to re-root should be selected according to the number of nodes in the predecessor paths from u to the root of $T - T(q)$ and from v to q. In particular, the subtree associated with the path containing the minimum number of nodes should be re-rooted. Further, if the dual variable updating operations are not integrated, the dual variables should be updated in the subtree containing the smallest number of nodes.

Thus the following procedures will be computationally investigated in section 5:

Procedure 1: Let T_2 be the subtree whose predecessor path from the proposed new root to the old root is smallest and separately update the dual variables in the subtree containing the fewer number of nodes.

Procedure 2: Same as Procedure 1 except that if T_2 is to be both re-rooted and its dual variables updated, then integrate the updating of the dual variables with the other updating operations.

One computational disadvantage of the above procedures is that the number of nodes in the path from u to the root of $T - T(q)$ is not known and to calculate it involves traversing the path from the intersection node z to the root of $T - T(q)$ for no purpose other than to calculate this number. Another computational disadvantage simply involves the fact that there are certain computational advantages to always performing all updating operations on one subtree. To partially overcome these difficulties, two compromise procedures are proposed:

Procedure 3: Let T_2 be the subtree containing the smallest number of nodes. Perform all updating operations on T_2, integrating the dual variable updating with the other updating operations.

Procedure 4: Determine T_2 as in Procedure 3. Do not integrate the updating of the dual variables.

III *Make y_2 the new root of T_2 and update $p(x)$, $s(x)$, $t(x)$, and $f(x)$.*

The basic notion behind this step is to reverse the predecessor orientations in the $y_2 - x_2$ path and reset the thread function so as to move all non-$(y_2 - x_2)$-path nodes in the subtree to the left of this path.

For clarity, this and the subsequent step assume that each function (p, f, t, and s) is updated independently. Integration of the operations is described at the end of each step. (Again, the reader may find it helpful to

carry out these operations using an updated figure 2 and Procedure 3 from Section II).

For $p^*(x)$. Reverse the predecessor orientation of the path from y_2 to x_2 as follows:

a.1. If $x_2 = y_2$, stop; do nothing.
a.2. Otherwise set $y = y_2$.
b.1. Recursively set $p^*(p(y)) = y$ and then $y = p(y)$ until and including $p(y) = x_2$.
c.1. Set $p^*(y_2) = 0$.

(It is important to note that $p(x)$ and $p^*(x)$ must be kept distinct from each other; i.e., it is not legitimate to replace $p(x)$ by $p^*(x)$ in the computation.)

For $s^*(x)$: If $x_2 = y_2$ no updating of $s(x)$ is required. Otherwise execute the following steps:

a.1. *Set* $x = y_2$, $w = f(x)$, and $z = s(w)$.
b.1. Let $y = r(x)$.
b.2. If $p(z) = p(x)$, set $s^*(y) = z$, $s^*(w) = p(x)$, $w = f(p(x))$, and $z = s(w)$; otherwise, set $s^*(w) = p(x)$ and $w = y$.
b.3. Set $x = p(x)$.
b.4. If $x \neq x_2$, go to b.1.
c.1. Set $s^*(w) = y_2$ and stop. In the updating of f, this last value of w (i.e., the node whose new thread is the new root y_2) plays a primary role. (Note that w is either equal to $f(x_2)$ or the last y.)

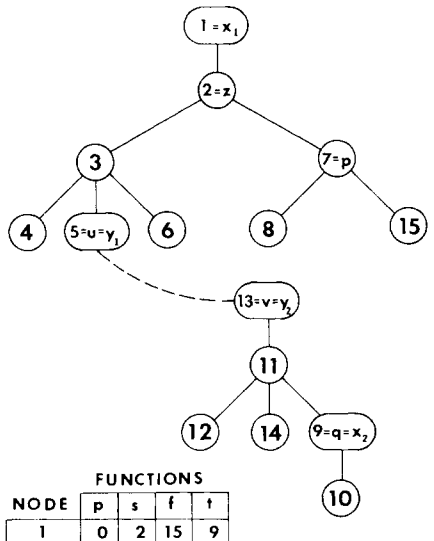
For $f^*(x)$: If $x_2 = y_2$, no updating is required. Otherwise, update as follows:

c.1. Let $\bar{x}_2$ be the unique node on the predecessor path from y_2 to x_2 such that $p(\bar{x}_2) = x_2$.
c.2. If $f(x_2) \neq f(\bar{x}_2)$, then $f^*(x_2) = f(x_2)$ (i.e., $f(x_2)$ does not change). Otherwise, $f^*(x_2) = w$, where $w = r(\bar{x}_2)$. (As noted above, w will be identified automatically at the conclusion of updating s.)
c.3. Set $f^*(x) = f^*(x_2)$ for all x on the predecessor path from y_2 to x_2.

For $t^*(x)$:

a.1. Set $t^*(y_2) = t(x_2)$. (But if $T(q) = T_1$ and the restricted update of $t(x)$ was carried out in Step I.1, set $t^*(y_2) = t(x_2) - t(q)$.)
b.1. For each x on the predecessor path from y_2 to x_2 (excluding $x = x_2$), set $t^*(p(x)) = t^*(y_2) - t(x)$. (Again note that $t(x)$ and $t^*(x)$ must be kept distinct and it is assumed that $p(x)$ has not been updated.)

Figure 3 indicates the updated function values obtained by applying Step III to figure 2. It appears that in order to integrate updating and optimize computational operations, Step III should be implemented as follows. Start by executing those operations labelled with an "a." Next, traverse the path from y_2 to x_2 and, for each x in sequence on this path, use the "b" operations to simultaneously update the functions $p(x)$, $s(x)$,



NODE	FUNCTIONS p	s	f	t
1	0	2	15	9
2	1	3	15	8
3	2	4	6	4
4	3	5	4	1
5	3	6	5	1
6	3	7	6	1
7	2	8	15	3
8	7	15	8	1
9	11*	10	10*	2*
10	9	13*	10	1
11	13*	12	10*	5*
12	11	14*	12	1
13	0*	11*	10*	6*
14	11	9*	14	1
15	7	1	15	1

*indicates values that were set during Step III

FIG. 3. Tree and function values after performing step III.

and $t(x)$, the basic flow values associated with arcs on this path, plus coordinate the updating of the node potential values with the updating of $s(x)$. Next, the remaining operations labelled with a "c" are performed and the new predecessor path from x_2 to y_2 should be traversed to update $f(x)$.

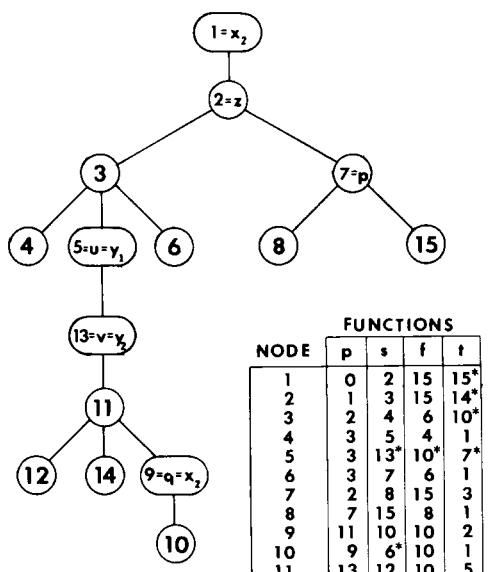
IV *Attach T_2 to T_1 by adding arc $[y_1, y_2]$ to create the new basis tree (where T_2 is now rooted at y_2 as a result of Step III).*

As before, in the following $p(x)$, $s(x)$, $f(x)$, and $t(x)$ refer to the value of the functions p, s, f, and t after the execution of Step III above and the rules for updating each function assume that no other functions have been updated since Step III.

For $p^*(x)$: Set $p^*(y_2) = y_1$.

For $s^*(x)$: Set $s^*(f(y_2)) = s(y_1)$ and $s^*(y_1) = y_2$.

For $f^*(x)$: Set $\bar{x} = p(s(y_1))$. If $\bar{x} = 0$ set $\bar{x} = x_1$. Then for those nodes x



NODE	FUNCTIONS p	s	f	t
1	0	2	15	15*
2	1	3	15	14*
3	2	4	6	10*
4	3	5	4	1
5	3	13*	10*	7*
6	3	7	6	1
7	2	8	15	3
8	7	15	8	1
9	11	10	10	2
10	9	6*	10	1
11	13	12	10	5
12	11	14	12	1
13	5*	11	10	6
14	11	9	14	1
15	7	1	15	1

* indicates values that changed during Step IV

FIG. 4. Tree and function values after performing step IV.

on the backward path from y_1 to $\bar{x}$, excluding $\bar{x}$ itself if $p(s(y_1)) \neq 0$, set $f^*(x) = f(y_2)$.

For $t^*(x)$: Set $t^*(x) = t(x) + t(y_2)$ for all x on the path from y_1 to x_1. (But if $T(q) = T_2$, and the restricted update of $t(x)$ was applied in Step I.1, then the current step should be restricted to those x on the path from y_1 to z, excluding z itself, for the "intersection" node z as identified in Section 3.1.) Note that the t and f functions can be updated simultaneously in tracing the path from y_1 to x_1.

Figure 4 illustrates the final function values derived from Step (iv). The proposed procedure for updating $t(x)$ clearly requires less effort than updating the distance function of reference (13), which involves an addition for every node of the subtree $T(q)$, and in the case of $T(q) = T_1$, requires an addition for every node of T. The fact that $t(x)$ can replace the distance function is a direct consequence of the observation that $t(x)$ can be used in essentially the same manner as the distance function to facilitate operations of identifying the loop created by adding arc (u, v)

to the basis tree (i.e., finding the basis equivalence path of (u, v)) as shown in Section 3.1.

4 Initialization

It is left to characterize the procedure for establishing the initial values of $t(x)$ and $f(x)$. This occurs simultaneously with the initial determination of the $s(x)$ values as follows.

Let x_0 denote the root of the tree. Consider the step in which $s^{k+1}(x_0) = s(s^k(x_0))$ is identified ($k \geqslant 0$). If $s^k(x_0)$ is the predecessor of $s^{k+1}(x_0)$ (via the predecessor indexing), do nothing. Otherwise, for all nodes $s^i(x_0)$ on the predecessor path from $s^k(x_0)$ to the predecessor of $s^{k+1}(x_0)$, excluding the predecessor of $s^{k+1}(x_0)$ itself, set $t(s^i(x_0)) = k + 1 - i$, and set $f(s^i(x_0)) = s^k(x_0)$.

When the last node $s^{n-1}(x_0)$ of the network is determined, set $t(s^i(x_0)) = n - i$ and set $f(s^i(x_0)) = s^{n-1}(x_0)$ for all $s^i(x_0)$ on the predecessor path from $s^{n-1}(x_0)$ to x_0.

To easily keep track of the index i for each node $s^i(x_0)$ that is to be considered on a given step, it is convenient to keep a list that consists precisely of the indexes i of the nodes $s^i(x_0)$ to x_0. Specifically, to begin with the list contains the single index 0 (for $s^0(x_0)$). When $s^{k+1}(x_0)$ is created, the number $k + 1$ is added to the end of the list. When a predecessor path from $s^k(x_0)$ is traced, consisting of r nodes (say) $s^i(x_0)$ whose values $t(s^i(x_0))$ and $f(s^i(x_0))$ are to be set, the indexes of these r nodes will be exactly the corresponding last r numbers on the list. By removing these numbers from the list just before adding the number $k + 1$, the desired structure of the list is maintained.

5 Computational Analysis

5.1. *Highlights of the development of the ARC-II computer code*

To evaluate the foregoing procedures, henceforth referred to as the Extended Threaded Index (XTI) Method, we developed a new in-core computer code entitled ARC-II for solving capacitated transshipment problems. ARC-II is written in a "manilla" FORTRAN with several subroutines to perform the various updating operations, for the following reasons: (1) this modular approach simplifies testing of different updating procedures, (2) minimal recoding is required to fit different machine and computer conventions, and (3) unbiased comparisons can be made with codes which have not been "customized" to a particular machine or compiler. One disadvantage of this approach, of course, is that the reported times are conservative, since programs which have

been "tuned" to a particular operating environment execute substantially faster. However, our purpose in developing ARC-II was to obtain unbiased comparisons between the XTI approach and other procedures available in the literature. To this end, the same starting and pivoting procedures as described in references (8, 9) for transshipment problems are used.

After initially developing ARC-II, preliminary testing was conducted on the recoding rules described in part II of Section 3. Our initial testing indicated that procedure 3 was never a good rule and that procedure 4 marginally dominated procedures 1 and 2. Thus, the times on the ARC-II code reported subsequently in this section reflect the use of procedure 4.

Another supplementary feature tested was maintaining the "reverse thread function" described in Section 3.2. Using this function in conjunction with the other functions previously discussed, one can eliminate all searching involved in the basis exchange operation; i.e., updating the thread function is simply a matter of resetting known pointers. The disadvantages of using the reverse thread directly include increased memory requirements and the need to maintain an additional set of function values. Our tests using the reverse thread function indicate that solution times are reduced by approximately 5%. In our opinion, this reduction is not sufficient to warrant the additional memory space, and therefore, ARC-II does not maintain such a function.

5.2 *Computational comparisons*

To determine the efficiency of the XTI procedures, ARC-II was compared with three out-of-kilter codes referred to hereinafter as SHARE,[4] SUPERK,[1] and BSRL (developed by T. Bray and C. Witzgall while at Boeing Scientific Laboratories). Additionally, one dual simplex based code,[5] called DNET, one negative cycle code,[2] called BENN, and two primal simplex based codes, called PNET and PNET-I, were tested for comparative purposes. PNET[8] uses the API list structure[6] and PNET-I[9] uses the ATI list structure.[9]

All of the above mentioned codes are in-core codes; i.e., the program and all of the problem data simultaneously reside in fast-access memory. All are coded in FORTRAN and none of them (including ARC-II) has been tuned (optimized) for a particular compiler. All of the problems were solved on the CDC 6600 at the University of Texas Computation Center using the RUN compiler. The computer jobs were executed during periods when the machine load was approximately the same, and all solution times are exclusive of input and output; i.e., the total time spent solving the problem was recorded by calling a CPU clock upon starting to solve the problem and again when the solution was obtained.

Since the test problems of reference (11) are currently used worldwide

TABLE 1

SOLUTION TIMES IN SECONDS ON A CDC 6600

Problems	ARC-II	PNET	PNET-I	DNET	BENN	SUPERK	SHARE	BSRL
1	.78	1.30	1.07	12.85	20.25	5.68	17.76	30.25
2	.89	1.49	1.25	13.56	24.36	6.47	21.34	21.59
3	1.01	1.94	1.64	21.44	34.56	6.87	26.16	31.47
4	.95	1.64	1.27	17.96	31.45	6.57	25.13	36.47
5	1.25	1.88	1.63	23.34	52.10	6.77	30.97	47.73
6	2.11	3.55	2.86	46.10	61.00	11.05	46.40	46.64
7	2.23	4.06	3.37	74.88	DNR*	12.86	65.92	113.12
8	2.99	4.72	4.10	97.92	DNR	13.69	81.00	175.10
9	2.99	4.80	4.15	101.65	DNR	13.40	81.21	186.99
10	4.02	5.88	5.27	95.96	DNR	14.13	84.24	184.75
11	1.92	3.52	2.31	19.87	17.44	6.44	19.93	30.39
12	2.36	4.87	3.71	26.58	20.31	6.47	21.17	22.08
13	3.13	5.52	3.47	27.98	24.92	7.25	25.81	20.02
14	2.96	6.02	3.44	30.15	27.40	6.95	24.95	23.11
15	3.12	6.50	4.79	31.57	DNR	7.56	27.05	21.08
16	1.38	2.40	2.15	14.77	11.77	5.27	21.51	15.05
17	1.87	3.11	2.60	DNR	20.10	8.36	32.40	64.64
18	1.26	1.92	1.70	DNR	11.31	5.13	20.06	18.31
19	1.72	2.60	2.40	DNR	20.62	8.49	31.75	61.07
20	1.28	2.67	2.47	DNR	10.38	4.69	18.11	25.72
21	1.83	2.76	2.46	DNR	20.35	7.96	32.60	61.39
22	1.26	2.22	2.01	DNR	9.97	4.60	17.91	24.84
23	1.67	3.00	2.74	DNR	19.81	7.91	32.66	67.96
24	1.52	3.12	2.91	DNR	11.71	5.59	25.27	21.57
25	1.83	4.17	3.96	DNR	18.27	8.37	33.19	48.40
26	1.08	4.45	4.05	DNR	11.38	5.51	25.05	19.34
27	1.62	4.42	4.21	DNR	16.37	7.50	30.45	41.98
28	4.40	6.35	5.37	DNR	DNR	13.91	53.87	83.98
29	4.87	7.39	6.25	DNR	DNR	14.51	52.55	117.83
30	4.88	9.08	7.90	DNR	DNR	16.00	61.33	152.21
31	5.68	9.59	7.58	DNR	DNR	17.05	61.33	135.73
32	7.42	15.70	11.73	DNR	DNR	22.88	78.63	553.93
33	7.82	20.20	15.95	DNR	DNR	25.89	101.92	210.14
34	8.21	17.10	13.76	DNR	DNR	25.42	92.25	248.16
35	8.81	19.39	15.87	DNR	DNR	29.96	DNR	DNR

*DNR–did not run.

for comparison purposes, they were also used in our comparison. This comparison included several different types of problems (e.g., assignment, transportation, and minimum cost flow network problems), both capacitated and uncapacitated, and with varying node and arc requirements. The problem specifications of these 35 problems as required on the input cards to the network generator are given in reference (11). Problems 1–5 are 100 $\times$ 100 transportation problems; problems 6–10 are 150 $\times$ 150 transportation problems. Problems 11–15 are 200 $\times$ 200 assignment problems. Problems 16–27 are 400 node capacitated transshipment prob-

TABLE 2

CODE SPECIFICATIONS

Developer	Name	Type	Number of Arrays
1 Barr, Glover, Klingman	ARC-II	Primal Simplex Network	7N + 3A
2 Barr, Glover, Klingman	SUPERK	Out-of-Kilter	4N + 9A
3 Bennington	BENN	Non-Simplex	6N + 11A
4 Bray and Witzgall	BSRL	Out-of-Kilter	6N + 8A
5 Clasen	SHARE	Out-of-Kilter	6N + 7A
6 Glover, Karney, Klingman	PNET	Primal Simplex Network	7N + 3A
7 Glover, Karney, Klingman	DNET	Dual Simplex Network	9N + 3A
8 Glover, Karney, Klingman	PNET-I	Primal Simplex Network	6N + 3A
9 General Motors	GM	Out-of-Kilter	3N + 6A

N–Node Length
A–Arc Length

lems; problems 28–35 are uncapacitated 1000 and 1500 node transshipment problems. Table 1 reports the solution times for each of these problems for each of the codes.

The results in table 1 clearly indicate the superiority of ARC-II over all other codes tested. Furthermore, the data indicate, rather startlingly, that ARC-II is approximately twice as fast as one of the (previously) fastest codes in the literature, PNET-I.

It is also particularly noteworthy that the solution times for the ARC-II code are based on using the simple pivot strategies of reference (8), rather than the more sophisticated candidate list strategies whose superiority has been documented by Mulvey.[12] We have used the simpler pivot strategies to differentiate more clearly the contribution of the new labelling procedures. (There has indeed been some confusion introduced into the literature by a number of recent studies whose comparisons have not been based on fundamental methodological differences in labelling procedures, but simply on differences in pivot strategies, and not clearly identified as such.) Thus, the times in table 1, while extremely fast, should not be construed as the best attainable with the ARC-II code. Preliminary tests with candidate list strategies, not yet refined to achieve the most effective trade-offs with the new labelling procedures, have, in fact, resulted in times that are roughly half of those in table 1.

5.3 *Memory requirements of the codes*

Table 2 indicates the number of node and arc length arrays required in

each of the codes tested for solving capacitated problems. It should be noted that the program memory requirements of all of the codes tested were quite close (within 1000 words) excluding the array requirements. Thus the important factor in comparing codes is the number of node and arc length arrays. Also, it should be noted that the primal and dual simplex codes require one less arc length array if the problem is uncapacitated. This is not true of the out-of-kilter codes.

Since any meaningful network problem has to have more arcs than nodes, it is clear by table 2 that the primal and dual simplex codes have a distinct advantage (in terms of memory requirements) over all of the other codes. Further, this advantage greatly increases as the number of arcs increases and if the problem is uncapacitated. For example, for a problem which has 10 times as many arcs as nodes, ARC-II, PNET, PNET-I, or DNET require only about one-half the memory space of the best of the other codes.

Acknowledgments

A number of particularly apt comments by a referee have improved the readability of this paper and are sincerely acknowledged. Also, the editorial assistance of Dr John Hultz, Systems Analyst of Analysis, Research, and Computation, Inc., and Mr David Karney, Systems Analyst at the Center for Cybernetic Studies, the University of Texas at Austin, are appreciated.

The authors also wish to acknowledge the cooperation of the staff of the University of Texas Computation Center, and Southern Methodist University Computation Center.

This research was partly supported by ONR Contracts N00014-75-C-0616, N00014-75-C-0569, and N00014-78-0222 with the Center for Cybernetic Studies, University of Texas.

References

(1) R.S. Barr, F. Glover, and D. Klingman, "An improved version of the Out-of-Kilter Method and a comparative study of computer codes," Mathematical Programming, vol. 7, no. 1, 1974, 60–87.

(2) G.E. Bennington, "An efficient minimal cost flow algorithm," OR Report 75, North Carolina State University, Raleigh, North Carolina, June 1972.

(3) G. Bradley, G. Brown, and G. Graves, "A comparison of storage structure for primal network codes," presented at ORSA/TIMS conference, Chicago, April 1975.

(4) R.J. Clasen, "The numerical solution of network problems using the Out-of-Kilter algorithm," RAND Corporation Memorandum RM-5456-PR, Santa Monica, California, March 1968.

(5) F. Glover, D. Karney, and D. Klingman, "Double-pricing dual and feasible start algorithms for the capacitated transportation (distribution) problem," CCS Re-

search Report 105, Center for Cybernetic Studies, University of Texas, Austin, Texas 78712.

(6) F. Glover, D. Karney, and D. Klingman, "The augmented precedessor index method for locating stepping stone paths and assigning dual prices in distribution problems," Transportation Sci., vol. 6, 1972, 171–180.

(7) F. Glover, D. Karney, D. Klingman, and A. Napier, "A computational study on start procedures, basis change criteria, and solution algorithms for transportation problems," Management Sci. vol. 20, no. 5, 1974, 793–813.

(8) F. Glover, D. Karney, and D. Klingman, "Implementation and computational study on start procedures and basis change criteria for a primal network code," Networks, vol. 20, 1974, 191–212.

(9) F. Glover, D. Klingman, and J. Stutz, "Augmented Threaded Index Method," INFOR, vol. 12, no. 3, 1974, 293–298.

(10) E. Johnson, "Networks and basic solutions," Operations Research, vol. 14, no. 4, 1966, 619–623.

(11) D. Klingman, A. Napier, and J. Stutz, "NETGEN–a program for generating large scale (un)capacitated assignment, transportation, and minimum cost flow network problems," Management Sci., vol. 20, no. 5, 1974, 813–819.

(12) J. Mulvey, "Column weighting factors and other enhancements to the Augmented Threaded Index Method for network optimization," Joint ORSA/TIMS Conference, San Juan, Puerto Rico, 1974.

(13) V. Srinivasan, and G.L. Thompson, "Accelerated algorithms for labeling and relabeling of trees with application for distribution problems," J. Assoc. Comput. Machinery, vol. 19, no. 4, 1972, 712–726.