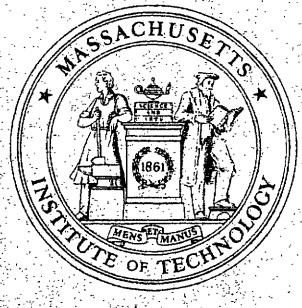
OPERATIONS RESEARCH CENTER

working paper



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

IMPLEMENTING PRIMAL-DUAL NETWORK FLOW ALGORITHMS*

by

H.A. Aashtiani**

and

T.L. Magnanti

OR 055-76 June 1976

* This research has been supported in part by the U.S. Department of Transportation under contract DOT-TSC-1058, Transportation Advanced Research Program (TARP) and by the Office of Naval Research under contract N00014-75-C-0556.
Most of the results of this report were presented at the National ORSA/TIMS meeting, Chicago, May 1975.

**Supported by the Arya-Mehr University of Technology, Iran.

ABSTRACT

We show how data structures similar to those proposed recently for implementing primal simplex based codes for solving network flow problems can be used to implement primal-dual algorithms, particularly the out-of-kilter algorithm. We also study several variants of a basic implementation which incorporate options for labeling, for making cost changes, for sequencing the selection of out-of-kilter arcs, and for implementing *the* primal-dual algorithm. Our investigations indicate that storing and manipulating data efficient leads to substantial reductions in computation time as well as storage requirements.

I. INTRODUCTION

A number of recent investigations of network optimization have illustrated the importance of using efficient data handling procedures to improve algorithmic performance. Gilsinn and Witzgall [19], Golden [27], Johnson [34], Pape [51], and Yen [58] have suggested methods for implementing shortest route algorithms; Kershenbaum [36] and Kershenbaum and Van Slyke [37] have proposed methods for implementing minimum spanning tree algorithms; and Golden, Magnanti, and Nguyen [29] have reported new implementations of vehicle routing algorithms.

In a continuing effort, Glover and Klingman and their colleagues at the University of Texas [9,20,21,22,23,25,26,39,40] have applied ideas first advocated for network applications by Johnson [32] to develop PNET, a major new implementation of the primal simplex method for minimum cost network flow problems (Dantzig [11] and [12]). Using the so-called triple-label list processing structure for describing routed trees within a computer (Scoins [54]), this code has made an order of magnitude improvement upon running times from previous generation minimum cost network flow codes. Motivated by these results and by independent contributions by Graves and McBride [30] and Srinivasan and Thompson [57], several additional codes have been designed [6], [7], [35], [45] which improve upon PNET by incorporating out-of-core implementations, alternative pivoting strategies, and variants to the basic list processing capabilities of PNET.

The success of these new generation codes cast doubt upon the pre-1970's "folklore" that primal-dual approaches, and particularly the out-of-kilter algorithm, for minimum cost network flow problems are superior to primal simplex-based approaches. In fact, Barr, Glover, and Klingman's [4] computational experience indicates that PNET is faster and uses less storage than an improved version of the out-of-kilter algorithm that they have developed.

Hatch [31], in summarizing computational experience for an improved version of primal-dual approaches for transportation and assignment problems, suggests different conclusions. His experience indicates that this new code is more efficient than PNET-1, a modified version of PNET, especially for large scale assignment problems. Unfortunately, he does not discuss any of the details of his implementation.

Several additional researchers [2], [3], [8], [13], [15], [38], [43],

[46] have either proposed other algorithms for minimum cost flow problems or discussed computational experience for this class of problems. For more complete references, see the Bradley [5] and Glover and Klingman [24] surveys and Golden and Magnanti [28] bibliography.

In this paper, we report on new in-core implementations of primal-dual network flow algorithms that use list processing structures similar to, but somewhat more involved than, those used for the newer primal codes. Computational experience suggests that our primal-dual codes are competitive with PNET in terms of running time, especially for assignment problems. In addition, our primal-dual code can be implemented by storing six node length vectors and five and one-half arc length vectors. These storage requirements are less than those for most previous implementations of primal-dual algorithms, but exceed requirements for primal based codes.

We should emphasize that we are reporting computational comparisons between our code and PNET as implemented initially. The primal codes, as refined by several researchers since this implementation of PNET, now run considerably faster. We would expect that a similar level of effort would improve upon the code presented here, while still using the basic constructs that we suggest. We have, in fact, been surprised to find that minor changes in our implementation, as discussed later, have had pronounced effects upon running time. Given this experience, we can envision reductions to the running times of the codes discussed here.

In addition, almost all experimental results published to date for either primal or primal-dual codes concern one time solution of single problems. Other experiments, such as solving network flow problems several times as a subproblem for a decomposition application, might conceivably provide different comparisons between codes.

This paper is organized as follows. The next section briefly reviews the out-of-kilter algorithm, emphasizing its computational requirements. Section III describes data structures used in our implementations, which are discussed in section IV. The final section summarizes our computational results. Throughout the paper, we discuss the out-of-kilter algorithm as a general purpose primal-dual algorithm. Section IV specializes the algorithm to give the primal-dual algorithm.

II. COMPUTATIONAL REQUIREMENTS OF THE OUT-OF-KILTER ALGORITHM

We assume that the minimum cost flow problem is formulated with decision variables x_{ij} as:

$$\begin{aligned} \text{Minimize} \quad & \sum_i \sum_j c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_j x_{ij} - \sum_k x_{ki} = b_i \quad (i=1,\ldots,N) \\ & 0 \leq x_{ij} \leq u_{ij} \quad (\text{all arcs } i\text{-}j). \end{aligned} \tag{2.1}$$

The indices in each summation are restricted to nodes i, j and k corresponding to (directed) arcs i-j and k-i in the given network. We further assume that the data c_{ij}, u_{ij}, and b_i are integers and that the network does not contain multiple arcs.

This formulation encompasses any single commodity (linear) network flow problem including transportation and assignment problems [16]; any network with nonzero lower bounds is converted to this general form by simple changes of variables, and multiple arcs can be modeled as above by inserting an additional node along any multiple arc.

Let π_i for i=1,...,N denote dual variables or shadow prices for the i^{th} equality constraint and for each arc i-j let

$$\bar{c}_{ij} = c_{ij} - \pi_i + \pi_j$$

denote its reduced cost.

The out-of-kilter method starts with, and maintains, values x_{ij} of the primal (flow) variables satisfying the equality constraints in (2.1), and systematically changes values of both the primal variables and dual variables until the following optimality conditions are satisfied:

$$\begin{aligned} & 0 \leq x_{ij} \leq u_{ij} \\ & \bar{c}_{ij} \geq 0 \quad \text{if } x_{ij} = 0 \\ & \bar{c}_{ij} = 0 \quad \text{if } 0 < x_{ij} < u_{ij} \\ & \bar{c}_{ij} \leq 0 \quad \text{if } x_{ij} = u_{ij} \end{aligned} \tag{2.2}$$

for every arc i-j in the network. Following Fulkerson [17], we say that any arc violated these conditions is <u>out-of-kilter</u> arc, and any arc satisfying

these conditions is in-kilter.

Since several authors [16], [17], [33], [42], [48], [56] have carefully discussed the details of the out-of-kilter algorithm, we will only provide a brief summary, emphasizing those aspects of the algorithm that are most essential for understanding implementation.

We first set some notation. We say that node j is reachable from node i if either

$$(1)\quad \bar{c}_{ij} \leq 0 \quad \text{for arc } i\text{-}j \text{ and } x_{ij} < u_{ij}$$

$$(2)\quad \bar{c}_{ij} > 0 \quad \text{for arc } i\text{-}j \text{ and } x_{ij} < 0$$

$$(3)\quad \bar{c}_{ji} \geq 0 \quad \text{for arc } j\text{-}i \text{ and } x_{ji} > 0$$

or

$$(4)\quad \bar{c}_{ji} < 0 \quad \text{for arc } j\text{-}i \text{ and } x_{ji} > u_{ji}.$$

For fixed values of the dual variables, reachability corresponds to possible flow changes (increasing x_{ij} for cases 1 and 2 and decreasing x_{ji} for cases 3 and 4) that either (i) maintain optimality for in-kilter arcs, or (ii) move "toward"optimality for out-of-kilter arcs.

For fixed values of the flow variables x_{ij} and dual variables π_i, let G be a directed graph on nodes $1,2,\ldots,N$ that contains arc i-j if node j is reachable from node i. We will say that node j is reachable from node k via an augmenting path if there is a (directed) path in G connecting node k to node j.

The out-of-kilter algorithm starts with any given values π_i for the dual variables and any given values x_{ij} for the primal variables satisfying the equality constraints in (2.1). Suppose that k-ℓ is an out-of-kilter arc satisfying condition (1) or (2) above or that ℓ-k is an out-of-kilter arc satisfying condition (3) or (4). In either case, ℓ is reachable from k and arc k-ℓ belongs to G; to simplify our exposition, we will refer to the arc k-ℓ in G as an out-of-kilter arc even though this arc may correspond to arc ℓ-k in the original network. Given arc k-ℓ, the algorithm searches for an augmenting path in G connecting node ℓ to node k to form a cycle C containing arc k-ℓ. The next step of the algorithm depends upon whether or not k is reachable from ℓ for the current values π_i of the dual variables; let R denote the set of nodes that are reachable from node ℓ in G via an

augmenting path. The two contingencies, are:

(i) $k \in R$, that is node k is reachable from node ℓ. By definition of G, each arc i-j in the cycle C corresponds to a "forward" arc i-j or a "backward" arc j-i from the original network. The flows x_{ij} on forward arcs become $x_{ij}+\theta$ and the flows x_{ji} on backwards arcs become $x_{ij}-\theta$ where $\theta = \min\{\theta_{ij}$ for arcs i-j of C$\}$ and

$$\theta_{ij} = \begin{cases} u_{ij}-x_{ij} & \text{if i-j is a forward arc of C and } \bar{c}_{ij} \leq 0 \\ -x_{ij} & \text{if i-j is a forward arc of C and } \bar{c}_{ij} > 0 \\ x_{ji} & \text{if j-i is a backward arc of C and } \bar{c}_{ij} > 0 \\ x_{ji}-u_{ji} & \text{if j-i is a backward arc of C and } \bar{c}_{ji} < 0. \end{cases}$$

(ii) $k \notin R$. The values dual variables π_i for $i \in R$ become $\pi_i+\delta$ for a suitable choice of δ. These alterations do not effect $\bar{c}_{ij}$ if both i and j belong to R or both i and j do not belong to R. If i-j is a <u>boundary arc</u>, that is $i \in R$ and $j \notin R$ or $i \notin R$ and $j \in R$, then $\bar{c}_{ij}$ changes. In the first case it becomes $\bar{c}_{ij}-\delta$ and in the second case it becomes $\bar{c}_{ij}+\delta$. The value for δ is $\delta = \min\{\delta_1, \delta_2\}$ where

$$\delta_1 = \min\{\bar{c}_{ij} : i \in R,\ j \notin R,\ \bar{c}_{ij} > 0 \text{ and } 0 \leq x_{ij} < u_{ij}\}$$

$$\delta_2 = \min\{-\bar{c}_{ij} : i \notin R,\ j \in R,\ \bar{c}_{ij} < 0 \text{ and } 0 < x_{ij} \leq u_{ij}\}.$$

Once these steps have been completed and the values of either the primal or dual variables have been altered, the graph G changes and the steps are repeated. The sources cited previously as references for the out-of-kilter method show that the algorithm terminates after a finite number of steps (see also [14] and [52] for modifications relaxing our assumptions on integral, or rational, data).

To summarize, the basic computational requirements of the algorithm at any step are:

(1) Labeling: showing that $k \in R$ or finding R when $k \notin R$,

(2) Flow Change: computing θ in step (i) and updating flows, or

(3) Cost Change: computing δ in step (ii) and updating $\bar{c}_{ij}$. To compute δ, we must first retrieve from storage $\bar{c}_{ij}$ for all boundary arcs with $i \in R$, $j \notin R$ or $i \notin R$ and $j \in R$. (In our implementation, we do not store values of the dual variables π_i; rather, we store and update the reduced costs $\bar{c}_{ij}$).

In section IV, we consider implementing these computations and discuss several options for the implementation such as the primal-dual algorithm. We first review basic list processing structures for storing and manipulating data.

III. DATA STRUCTURES

Solving large optimization problems requires manipulation of large amounts of data and access of data from large data sets. In most network optimization problems, we can perform these operations efficiently by using list processing structures from computer science. Knuth's [41] impressive reference provides a wealth of material on this subject. Magnanti [44] reviews some of the constructs used in minimum cost flow algorithms and illustrates the data structures with examples. In this section, we briefly indicate the type of data structures used in our codes and discuss how we modify these data structures during the course of the algorithm.

Network Structure

In out-of-kilter algorithms, we must frequently retrieve information about an arc of the network knowing its beginning and ending nodes, usually gathering data (reduced costs, capacities, and/or flow values) for all arcs with the same start node or the same end node. Because searching for a particular arc or set of arcs from among all the arcs in a network can be most time consuming, we require a list structure which would facilitate these operations. Building a latter representation link structure between all arcs which have same start node and another between all arcs which have same end node [19] accelerates the execution of the above tasks.

These link structures can be implemented by two tables - a node table and an arc table. The first table contains two entries (pointers) for each node; one points to the first arc in the arc table which emanates from that node and the other points to the first arc which comes into that node. The second table contains two entries (pointers) for each arc i-j; one points to the next arc in the list which has the same start node i and the other one points to the next arc in the list with the same end node j. A zero instead of a pointer in either case indicates that no arc in the remainder of the arc table has the same start or end node. These two tables link all arcs with same start node, as well as all arcs with same end node.

When the data are sorted in order of their start nodes, as in many applications, the first pointer in the arc storage table is not necessary. But when the data are not sorted, for example when a network is generated

within the context of another procedure or when a new set of arcs are added to the network, it may be more efficient to use such a link, because in most cases, the sorting time is excessive, almost half of the total solution time for an example to be reported later.

If we let N and A denote the number of nodes and arcs in the network for the problem, then the storage requirements for these pointers is 2N+A or 2N+2A words depending upon whether or not the data has been sorted previously. In addition, the arc table includes arc length vectors for arc costs, arc capacities and arc flows. It also contains storage to record the start node i and end node j for each arc i-j.

Therefore, we use either 2N+6A or 2N+7A words of storage for this information. In the next section we show how the arc flows can be stored in a node length vector in place of an arc length vector. (When the arc data is sorted by start node, we do not need to record the start node i of each arc i-j. This information can be recovered, with additional computations, from the data pointing to the first arc with the starting node i.)

Tree Structure

Several data structures have been developed for representing a tree in a computer [41]; the efficiency of each method depends upon the type of application. For this paper, the major tasks are traveling through a tree or a branch of a tree, breaking off a branch from a tree, and connecting a new branch to a tree. Therefore a structure like the triple-label method ([32] and [54]) seems most useful.

In the triple-label method each node has three labels, a predecessor (up), a successor (right-down), and a brother (left). Like a family relation, each node has one father and possibly a number of brothers and sons. The position of a member of family, a node, with respect to other members of the family in the ancestry tree would be shown by three links from that member to three other members.

The first link of any node points to its father (predecessor), the second link points to its oldest son (successor), and finally the third link points to its next younger brother (brother). Exactly one node in the tree, called the "root" of the tree, has no father as specified by a zero for its predecessor.

Since the out-of-kilter algorithm to be considered here works on a forest structure instead of a single tree, more than one node could have

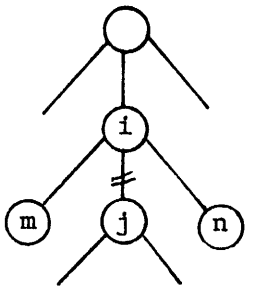
no father,besides the root of the current tree being investigated. We call these nodes "orphan roots", or more simply "orphans"; in our implementations we designate orphan nodes by a negative number for their predecessor labels. Thus a forest consists of a tree and a number of "orphan trees" corresponding to orphan nodes. When performing the out-of-kilter algorithm, the forest is a subset of the graph G with nodes 1,2,...,N and node ℓ is the root of the tree.

The storage requirements for the forest information is 3N. In addition, we require 2 logical node length arrays - one to indicate whether a node is labeled (as defined in the next section) and one to indicate whether an arc i-j in G corresponds to a forward or backward arc from the original network.

Updating Forests

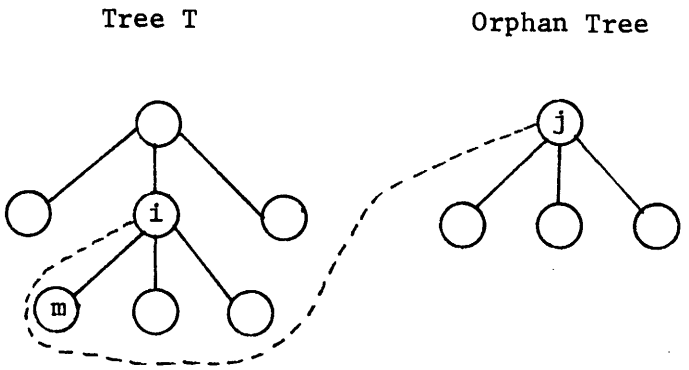
In our implementation of the out-of-kilter algorithm, we use a forest structure which must be updated at times either by adding or by deleting an arc i-j from the forest. Figures 3.1 and 3.2 indicate the modifications required to the data structures when making these changes. Note that as shown, we only add arc i-j when node j is an orphan root.

Often in the course of the algorithm we will use these operations together by adding "the subtree routed at node j to node i". In this case, we first delete the arc p-j from the forest, where p is the current predecessor of node j so that node j becomes an orphan root. Then we add arc i-j to the forest as illustrated in Figure 3.2.



	After Deletion		
As Pictured	If j=n was the successor of node i	If j=m was the youngest son of node i	If j was the only son of node i
Broth (n) = m	Suc (i) = m	Broth (n) = 0	Suc (i) = 0
Pred (j) = -1	Pred (j) = -1	Pred (j) = -1	Pred (j) = -1
Broth (j) = 0	Broth (j) = 0		

Figure 3.1 Deleting Arc i-j From the Forest



	After Addition
As Pictured	If node i had no successors
Broth (m) = j	Suc (i) = j
Pred (j) = i	Pred (j) = i
Broth (j) = 0	Broth (j) = 0

Figure 3.2 Adding Arc i-j to the Forest

IV. IMPLEMENTATION AND OPTIONS

In this section, we describe implementations of the out-of-kilter algorithm that use the data structures just discussed. We also consider several options to the basic implementation, such as a primal-dual algorithm, that effect storage requirements and/or execution time. We will adopt the notation used in section II for describing the out-of-kilter algorithm. In particular, arc k-ℓ is the arc in G that we are trying to place in-kilter at the current iteration of the algorithm.

Labeling and Forests

Implementations of the out-of-kilter method perform the first requirement at each step, determining whether or not node k is reachable from node ℓ via an augmenting path in the graph G, by constructing a tree rooted at node ℓ. The computations start with an initial tree T rooted at node ℓ, possibly consisting solely of node ℓ. If at any point in the construction no node $j \notin T$ is reachable from a node $i \in T$, then T cannot be enlarged and the set R of nodes reachable from ℓ via an augmenting path coincides with the nodes in T. Otherwise, some node $j \notin T$ is reachable from a node $i \in T$. After finding such a pair of nodes, the algorithm adds node j and arc i-j to the tree. This process is repeated until either node k has been added to the tree or the tree cannot be enlarged any further.

We adopt standard notation and say that any node contained in T is labeled and any labeled node i is scanned if we know that no node $j \notin T$ is reachable from node i.

There are several possible options for implementing this general tree growing procedure. The most straightforward implementation would initiate the procedure from scratch with $T = \{\ell\}$ after each flow change, or whenever we select a new out-of-kilter arc k-ℓ from G. This type of implementation ignores a great deal of information generated previously by the algorithm. Suppose, for example, that we have just added node k to the tree T shown on the left in Figure 4.1. The change in flow along the path from node ℓ to node k might remove one or more of the arcs along this path from G; no other arc in G will be altered, though. Suppose, in this instance, that the flow change causes only arc 25-3 to drop from G and that node 3 is reachable from node 6. Then if we add node 3 to the tree indi-

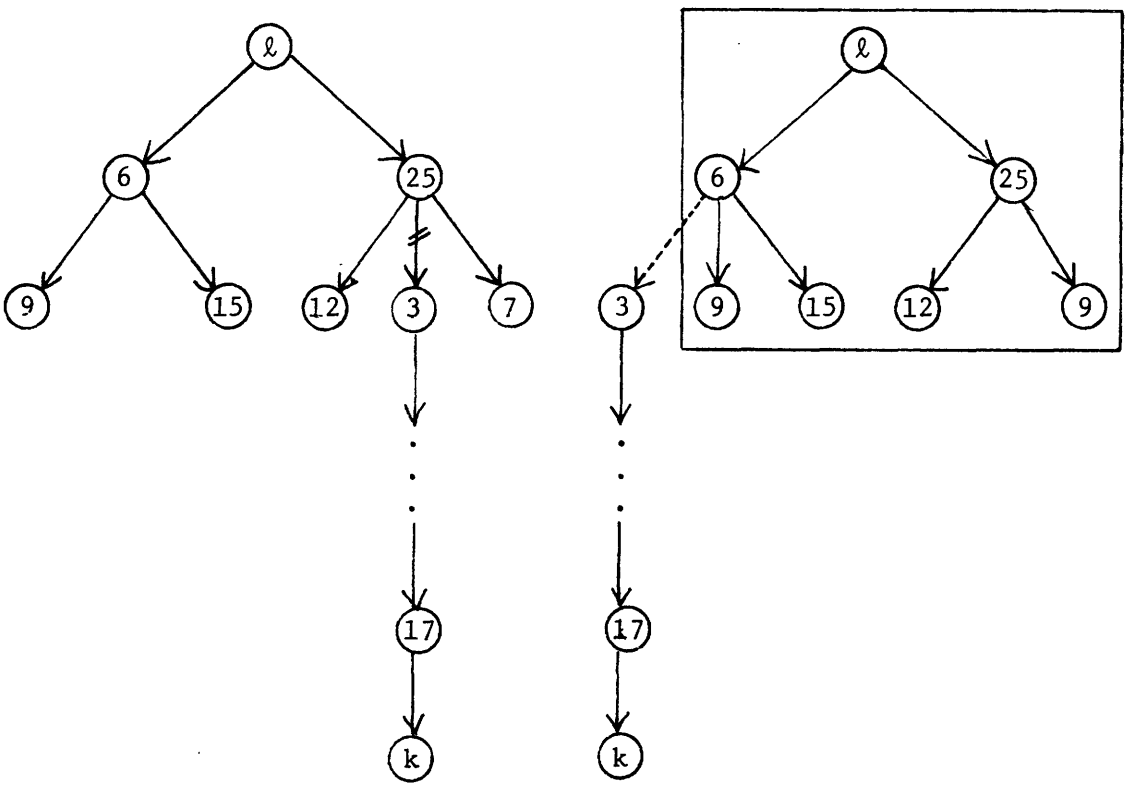


Figure 4.1 Saving Information After Flow Changes

cated within the box in Figure 4.1 and, at the same time, add the entire path P from node 3 to node k, then we find a new path from node ℓ to node k easily. By recalling that whenever we reach node 3, we may also reach node k, we have eliminated adding the arcs along P to the tree one arc at a time.

We implement this idea by storing and manipulating a forest in G which will include the tree being constructed as one component as well as components, such as the path P, which contain other reachability data. We assume that every node in G is contained in one component of the forest; some components may consist of a single node.

We update the forest structure at three points within the algorithm:

(1) Each time we select a new out-of-kilter arc k-ℓ, we must make node ℓ the root of the current tree, if it is not already the root. This task can be accomplished easily by (i) setting the predecessor of the current root to -1 so that it becomes an orphan root, (ii) breaking-off node ℓ from its predecessor p (i.e. deleting arc p-ℓ from the forest), if node ℓ is not already an orphan root, and (iii) setting the

predecessor of node ℓ to 0 to designate that it is the root of the tree. At this point the tree may contain only node ℓ; however, it usually contains more nodes and possibly even node k.

(2) Whenever we want to add a new arc i-j in G to the tree, when i is in T, we add the subtree in the current forest rooted at j to node i.

(3) After each flow change some of the arcs in the cycle C may leave G. For each such arc i-j we simply delete the arc i-j from the forest so that node j becomes a new orphan root.

These modifications to the labeling procedure permit us to retain labeling information from step to step. As we have seen, the data manipulations for making the required changes in the forest structure in each case are simple to implement using the list processing structures described in the last section.

Labeling Order

The solution time for out-of-kilter algorithm is very sensitive to how, and in which order, the algorithm "travels" through a tree visiting and labeling the nodes when constructing T. We distinguish two major procedures concerning the order for traveling through the tree.

In the first procedure we travel by level, "first-labeled first-scanned" [9], adding to the tree all possible arcs in G from any node when it is visited. In the second procedure, we travel to the right, "last-labeled first-scanned", as soon as possible, in an attempt to find an augementing path from ℓ to k quickly by following each branch until we reach a node where the tree cannot be extended further. Experiments have shown that although the lengths of the augmenting paths for the first procedure are shorter than those produced by the second procedure, yielding in fact the shortest paths from ℓ to k [9], the first procedure is much slower than the second procedure.

The following sequence of instructions are followed for the traveling and labeling procedure when traveling to the right. By the current node, we mean the node from which the method is now scanning to add any possible augmenting arcs.

Step 1 - Move downward and to the right as much as possible from the current node by moving from each node encountered to its successor, until a node j is found with no successor in the current tree T. Go to Step 2.

Step 2 - If there is no arc i-j in G from the current node i to a node j not in the current tree T, then go to Step 3. Otherwise, select any such arc i-j in G and add the subtree rooted at node j to the current tree below node i. If k lies in the subtree, then terminate; we have found an augmenting path from node ℓ to node k. Otherwise, designate node j as the current node and go to Step 1.

Step 3 - No unlabeled node is reachable from the current node; that is, the current node is scanned. If the current node has a brother, then continue with Step 1 with its brother designated as the current node;otherwise go to Step 4.

Step 4 - If the current node has a father then continue Step 1 with the father designated as the current node; otherwise there is no augmenting path from node ℓ to node k with respect to the current node potentials and the node potentials must be changed.

In contrast, when labeling by level we start at the root, at "level" n=0, in the tree, and continue to travel through the tree by level (n=1,2,...). For each level n, we move from the right to the left in the tree through the labeled nodes which are in the same level (that is, have the same number of arcs in the tree in the path joining them and the root); we scan each node i as we encounter it, adding all unlabeled nodes (and the subtree rooted at these nodes) reachable from node i to the tree.

Options for Traveling and Labeling

There are several options which we may use when searching for an augmenting path. First, whenever an arc i-j from G is added to the tree, to recognize whether node k appears in the subtree rooted at node j it is necessary to move from node k toward its predecessors. We can avoid repeating this task several times by using one node-length logical array

to keep track of nodes in the path between node k and the orphan root corresponding to the orphan tree which contains node k. Then when we add node j to the tree, we merely check the logical array to see if node k belongs to the orphan tree with j as an orphan root. The logical array is recomputed whenever we start the labeling process with an out-of-kilter arc ℓ-k; that is, after we make a flow change or select a new out-of-kilter arc.

Secondly, the method can work on several out-of-kilter arcs simultaneously instead of a single arc at each iteration. We may simultaneously consider all out of kilter arcs j-ℓ in the graph G for all j, and search for augmenting paths that start at node ℓ and end at one of the nodes j. This option can be implemented using the same logical array used when only a single node j=k is considered, as above, by keeping track of all nodes which are in the paths between each such node j and its corresponding orphan root. This implementation needs more work per iteration, but finds augmenting paths in fewer iterations.

Implementing the Flow Changes

After finding an augmenting path from node ℓ to node k, we have used a two-pass procedure to change the flows. In the first pass the maximum flow change, θ, is calculated (see the algorithm description in Section II); in the second pass, we make the flow changes. Any arc in the augmenting path reaching the maximum flow change breaks off from the tree T and its successor node becomes an orphan root. Each pass is performed easily by moving backward from node k toward node ℓ using the predecessor relation.

An alternative procedure is to maintain the maximum flow change possible from the root ℓ to every node in T as the tree is being constructed. We can then eliminate the first pass in the flow change segment of the algorithm.

Implementing the Cost Changes

Like flow changes, the cost changes can be implemented by a two-pass procedure. In the first pass we calculate the maximum allowable cost change, δ, and in the second pass we perform the cost changes. For each pass we may use the same traveling procedure used when labeling, starting at the root of the tree, to access to all boundary arcs between the

labeled nodes and unlabeled nodes in the network.

There are optional implementations which may be used for this step. During the first pass, we could save a list of all boundary arcs in an array for use in the second pass and avoid using the traveling procedure again. The maximum dimension of this array is usually less than the half of the number of the arcs. This implementation has had a great effect on the solution time.

During the second pass of the cost changes, we could also construct a list of all labeled nodes i corresponding to boundary arcs i-j that are added to the graph G after the cost change. If arc k-ℓ still remains out-of-kilter after the cost change, then instead of starting the labeling procedure at the root of the tree again, we continue the procedure with the nodes in the list (since no arc will be added to any node of the tree not in the list, these nodes will already have been scanned). A "first-in first-out" queue structure is suitable for choosing the nodes from this list.

We can imbed this queue structure within the labeling procedure discussed previously; when starting with a new out-of-kilter arc, we empty the queue and put ℓ, the root of the tree, in the queue. At each step we select a node from the top of the queue as the current node and apply the traveling procedure starting from this current node; if the traveling procedure ends without any augmenting path, we then select the next node from the queue to use as the current node and initiate the traveling procedure again. When the queue is empty, we have completed labeling without finding an augmenting path, and we next envoke the cost change segment of the algorithm.

The above option greatly improves the solution time without using much extra storage, since the maximum length of the queue is usually very short (about one tenth of the number of nodes in our experimentations).

There is another option which might be used in place of the first pass of the cost change routine. Since the exact set of boundary arcs is hard to identify during the course of the labeling routine, we could choose to store a larger set - all arcs i-j with the property that node $j \notin T$ is not reachable from node i when scanning node i. This set contains all of the boundary arcs and possibly other arcs i-j since node j might be labeled from some other node of T during the labeling process, even though it couldn't be labeled from node i. Our computational experience

showed that this option was not effective.

Order of Choosing the Out-of-Kilter Arcs

There are several options concerning the manner and order for choosing the out-of-kilter arcs to bring in-kilter, especially when artificial arcs have been added to place the problem into a circulation form. This well known network device [16] permits the algorithm to start with any values x_{ij} for the arc flows. Recall that the artificial variable technique expands the network to satisfy the conservation equation $f(i) = b_i$ for $i=1,\ldots,N$, where

$$f(i) = \sum_j x_{ij} - \sum_r x_{ri}$$

is the net flow out of node i, by adding a "super source" and "super sink". An artificial arc with initial flow f(i) and capacity b_i connects the super source to any "source node" i with $b_i > 0$ and $f(i) \neq b_i$. An artificial arc with initial flow $-f(i)$ and capacity $(-b_i)$ connects any "sink node" i with $b_i < 0$ and $f(i) \neq b_i$ to the super sink. In addition, one arc with capacity $\Sigma\{b_i | b_i > 0 \text{ and } f(i) \neq b_i\}$ and initial flow $\Sigma\{f(i) | b_i > 0 \text{ and } f(i) \neq b_i\}$ connects the super sink to the super source.

Imposing large negative costs on all of these arcs ensures, if possible, that the artificial arc joining node i from the original network has flow equal to b_i in the solution to the expanded problem. In this case, every artificial arc is out-of-kilter initially. Alternately, if we assume that the initial flow in every artificial arc does not exceed the arc capacity (as for example when starting with zero flows so that each $f(i) = 0$), then the procedure can start with only a subset of these arcs out-of-kilter:

(i) only the arc connecting the super sink and super source,

(ii) only the arcs joining the super source with the source nodes, or

(iii) only the arcs joining the sink nodes with the super sink.

In each case, we impose large negative costs on the arcs specified in (i), (ii), or (iii) and set to zero the costs in all other artificial arcs; when the arcs specified in (i), (ii), or (iii) are in-kilter for each case, then all of the artificial arcs are in-kilter and satisfy $f(i) = b_i$.

The results presented in the next section show that the second option is more efficient; the reason might be that the tree spans more nodes for

the other options and becomes more costly to work with.

Initial Solutions

The out-of-kilter algorithm can start with a zero feasible solution or with a nonzero feasible solution, provided by an initialization procedure such as the Modified Row Minimum Rule [21,22,57]. Many recent codes start with a nonzero feasible solution claiming that such initializations lead to more efficient implementations, but the results in the next section run counter to this claim, at least for our implementation of out-of-kilter algorithm.

The Primal Dual Algorithm

The primal-dual algorithm for the minimum cost flow problem starts with, and maintains, arc flows x_{ij} satisfying the optimality conditions (2.2), and terminates when the conservation (equality) equations of (2.1) are satisfied. We can use the out-of-kilter algorithm to implement this algorithm, starting with any given values (possibly zero) for the dual variables π_i, or equivalently, starting with any given values $\bar{c}_{ij}$ for the reduced costs of the arcs. (Shapiro [55] has discussed relationships between the primal-dual and out-of-kilter algorithms).

We initiate the algorithm by setting the flow on any arc i-j to $x_{ij} = 0$ if $\bar{c}_{ij} \geq 0$ and to $x_{ij} = u_{ij}$ if $\bar{c}_{ij} < 0$, therefore satisfying conditions (2.2). We next introduce a super source and super sink, as discussed previously, to place the problem into circulation form. The out-of-kilter algorithm will then continue to satisfy the optimality conditions (2.2) for every arc in the original network.

It is possible to implement this algorithm by allowing only the arc k-ℓ and the arcs in the forest to have flow not assigned to one of its flow bounds. The initial solution with the artificial arcs as the only arcs in the forest satisfies this property. To maintain the property throughout the algorithm, we must be sure that whenever an arc i-j is deleted from the forest, its flow is $x_{ij} = 0$ or $x_{ij} = u_{ij}$.

To ensure this condition requires modifications to the algorithm, though. When adding node j to the tree during the labeling process, we first delete arc i-j from the forest, where i is the predecessor of node j. If $0 < x_{ij} < u_{ij}$ and $\bar{c}_{ij} = 0$, then we simply reverse the predecessor relation between nodes i and j so that i becomes a son of node j. This

reversal is possible, in this case, because i and j are reachable from each other.

This technique applies to any arc from the original network, since these arcs start, and thus remain, in-kilter satisfying one of the three conditions $x_{ij} = 0$, $x_{ij} = u_{ij}$ or $0 < x_{ij} < u_{ij}$ and $\bar{c}_{ij} = 0$. The method might not be applicable if the arc i-j to drop from the forest is an artificial arc though, since, in this case, the arc might not satisfy any of these three conditions. In this instance, we modify the out-of-kilter algorithm by reversing the predecessor relation between nodes i and j, even though node j is not reachable from node i according to our previously established conventions. The algorithm will retain its convergence properties though, since no in-kilter arc ever becomes out-of-kilter and, in Fulkerson's terminology ([16] or [18]), after each flow change the "kilter-number" of arc k-ℓ either improves by an integer or remains unaltered and one more node is added to the current tree. Consequently, after a finite number of iterations the solution is optimal with all arcs in-kilter, i.e. all kilter-numbers are zero. This modified version of the out-of-kilter algorithm behaves somewhat differently than the original version, since the kilter-number of some of the artificial arcs might increase during the execution of the algorithm.

V. EVALUATING THE NEW CODE

This section summarizes computational experience with the new code, called KILTER, compares the code with other available codes, and indicates how the implementation options discussed in the previous section affect the total computation time. The section illustrates possible tradeoffs between computation time and storage.

Comparing KILTER with Other Codes

Klingman, Napier, and Stutz [40] have benchmarked the out-of-kilter codes SHARE [10,50], Boeing, SUPERK [4], and the primal network code PNET on 40 different types and sizes of problems created by NETGEN, a computer program for generating minimum cost flow problems. We have chosen a subset of these 40 problems for our computational experiments, selecting different types and sizes of problems.

Table 5.1 specifies the developer, code name, approach, and the storage requirement for several recent computer codes for solving network flow problems. Table 5.2 shows the input specifications of test problems that we have used; included are problems for transporation models, assignment models, capacitated network models, and an uncapacitated network model.

Table 5.3 presents the computational times (excluding input and output) for solving problems, as created by the NETGEN generator, with the test specifications. The solution times for the SHARE, Boeing, SUPERK, PNET and I/O PNET codes were obtained (in [35] for I/O PNET and in [40] for the other codes) on a CDC 6600 computer using the FORTRAN Run compiler, while the times for the KILTER code were obtained on an IBM 370/168 computer using the FORTRAN G compiler (which is similar to the RUN compiler).

A noteworthy feature of the computational results is that the newer generation codes I/O PNET, PNET, SUPERK and KILTER are decidedly superior to the other codes. Roughly, I/O PNET, PNET, SUPERK and KILTER are at least four times, and in many cases 8-10 times, faster than the other codes. Also, PNET is roughly twice as fast as the efficient out-of-kilter code SUPERK. I/O PNET is within 10% of SUPERK with respect to total solution time (central processing time and accessing time to the out-of-core file) and is consistently faster with respect to central processing time (cp).

Comparison of KILTER with SUPERK and PNET is difficult because of limited computation experience. But, from the results of Table 5.3 it seems

Table 5.1: Code Specifications

	Developer	Name	Approach	Number of Arrays
1.	Aashtiani and Magnanti	KILTER	Out-of-kilter	6N + 5(1/2)A*
2.	Barr, Glover, and Klingman	SUPERK	Out-of-kilter	4N + 9A
3.	Bennington	BENN	Non-simplex	6N + 11A
4.	Boeing	Boeing	Out-of-kilter	6N + 8A
5.	Clasen	SHARE	Out-of-kilter	6N + 7A
6.	Glover, Karney, and Klingman	PNET	Primal network	9N + 3A
7.	Glover, Karney, and Klingman	DNET	Dual network	9N + 3A
8.	Glover, Karney, Klingman, and Stutz	PNET-I	Primal network	8N + 3A
9.	General Motors	GM	Out-of-kilter	3N + 6A
10.	Karney and Klingman	PNET I/O	Primal network (In-core out-of core)	7N + $3B_A$
11.	Texas Water Development Board	TWB	Out-of-kilter	4N + N^2 + 7A

N : Node-length array

A : Arc-length array

B_A : Buffer array

* One node length array could be reduced in size to about (1/10)N.

Table 5.2: Problem Specifications

Prob. No.†	Number of Nodes	Number of Sources	Number of Sinks	Number of Arcs	Total Supply	Cost Range		Percent of High Cost Arcs	Percent of Arcs Capacitated	Upper Bound Range		Random No. Seed
						Min	Max			Min	Max	
1	200	100	100	1300	100,000	1	100	0	0	0	0	13502460
2	300	150	150	4500	150,000	1	100	0	0	0	0	13502460
3	300	150	150	5155	150,000	1	100	0	0	0	0	13502460
4	400	200	200	1500	200	1	100	0	0	0	0	13502460
5	400	200	200	3000	200	1	100	0	0	0	0	13502460
6	400	200	200	4500	200	1	100	0	0	0	0	13502460
7	400	8	60	2443	400,000	1	100	20.	30.	16,000	30,000	13502460
8	400	4	12	2676	400,000	1	100	30.	80.	16,000	30,000	13502460
9	1000	50	50	4800	1,000,000	1	100	0	0	0	0	13502460

† These problems correspond to those numbered 1,7,8,11, 13, 15, 17, 25, and 31 in reference [40].

1-3 are transportation problems

4-6 are assignment problems

7-8 are capacitated network problems

9 is an uncapacitated network problem

Table 5.3: Solution Times (in seconds)

Problem Number	KILTER†	PNET I/O TM	PNET I/O cp	PNET	SUPERK	SHARE	Boeing	Objective Value Function	Objective Value Function For KILTER
1	2.85	3.71	1.75	1.30	5.68	17.76	30.25	2,153,303	2,153,303
2	10.48	11.44	5.20	4.06	12.86	65.92	113.12	2,046,034	2,046,034
3	12.97	15.71	7.18	4.72	13.09	81.00	175.10	2,155,354	2,155,354
4	2.34	5.70	3.71	3.52	6.44	19.93	30.39	4,563	4,563
5	3.89	14.89	8.58	5.52	7.25	25.81	20.02	3,070	3,070
6	5.29	16.45	8.60	6.50	7.56	27.05	21.08	2,721	2,721
7	5.81	8.42	3.56	3.23	8.47	32.40	54.10	44,683,151	44,682,512
8	6.23	11.60	5.49	3.26	8.14	31.34	56.85	56,967,178	56,506,992
9	10.88	20.46	13.45	9.59	17.05	61.33	135.73	82,561,499	82,560,095

† Using an IBM 370/168 FORTRAN G Compiler. Other times report from references [35] and [40] on a CDC 660 using the FORTRAN RUN Compiler.

cp denotes central processing time

TM denotes central processing time and accessing time to the out-of-core file

1-3 are transportation problems

4-6 are assignment problems

7-8 are capacitated network problems

9 is an uncapacitated network problem

KILTER is faster than SUPERK and PNET is twice as fast as KILTER for transportation problems. For assignment problems, KILTER is one of the fastest codes, operating about 50% faster than SUPERK and 30% faster than PNET. For uncapacitated and capacitated network probelsm PNET is almost always faster than KILTER. Observe that KILTER runs much faster than SUPERK for problem 9 with a high ratio of nodes to arcs.

Notice that for problems 7, 8, and 9, the optimal objective function values obtained by the KILTER code on the IBM 370/168 computer are slightly different than those obtained with the other codes on the CDC 6600 computer. Apparently the NETGEN generator is not completely independent of the computer system as suggested by the code's developers. This experience agrees with results of other researchers.

Of the codes reported, KILTER is probably the fastest out-of-kilter code; it might be the fastest of these codes for assignment problems. From the viewpoint of both solution time and storage requirements, the performance of the KILTER code lies somewhere between that of SUPERK and PNET [†].

Hatch [31] has reported computational experience (on a CDC 6600 computer with a FORTRAN RUN complier) for Decision System Associates' implementation of the out-of-kilter algorithm. The solution times reported for this code are consistently less than those of PNET for several transportation and assignment problems - in particular for problems numbered 6 through 15 from reference [40] - and are less than those reported in Table 5.3 for KILTER for the assignment problems 4-6. We are unsure of the details of his implementation such as whether the code has been tailored for transportation and assignment problems, what list processing structures have been used, and the code's storage requirements. It is possible that the techniques used in his implementation can be coupled with the techniques used in KILTER to improve further upon out-of-kilter codes.

Again, though, we caution about making definitive comparisons between

---

[†] Since we have completed the work reported here, Ron Denbo and John Mulvey ("On the Analysis and Comparison of Mathematical Programming Algorithms and Software", Tech. Report HBS 76-19, Graduate School of Business, Harvard University) have compared KILTER with LPNET, a very fast primal simplex based code. Although the tests were conducted on a PDP10 computer and KILTER was designed for an IBM 370/168, the computational results do indicate that the most recent primal codes may be two to three times faster than KILTER when solving problems from scratch.

the various codes based upon this data. The limited experiments and the variations in both programmers and computer system environments make comparisons very difficult. Uniform testing procedures are required. Also, all of the experimental results that we have conducted concern a one time solution of a single problem. Other experiments, such as resolving a problem with slightly modified data as when performing sensitivity analysis or when solving a problem as a subproblem for a decomposition application, might conceivably provide different comparisons between the codes. Nevertheless, our computational results do indicate that the list processing structures that we have used do improve implementation of the out-of-kilter algorithm.

Testing Options

This section presents experimental results concerning some of the implementing options discussed in the previous section. Although the experiments are not sufficient for reaching definitive conclusions, they do indicate how sensitive solution time is with respect to minor changes in the implementation.

Table 5.4 lists nine of the options that we have tested. Most of the problems solved in this section were created by a simple network generator (NGC) developed for the purpose of this study. Aashtiani [1] describes the details of the generator. Table 5.8 compares the NGC and NETGEN generators for a limited set of problems. NETGEN seems to generate transportation problems, and NGC seems to generate other problem types, which are more difficult to solve.

We conducted our experiments in three phases. In the first two phases we used the multics system at M.I.T. (a timesharing system available on a Honeywell computer) which is about 3.5 times slower than the IBM 370/168 computer for this class of applications. We have not reported all of our computational experience for testing the various options. Rather, we present results for the benchmark problems introduced earlier in this section which are representative of our experience in general. Finally, we have not tested each option on all of the benchmark problems.

Table 5.5 represents the type of results that we obtained in phase 1 for testing options KILTER 1,2,3, and 6. Keeping track of nodes in the

Table 5.4: Option Specifications

KILTER 0 - Original code storing and manipulating a forest structure, using zero values for initial flows, and first selecting out-of-kilter arcs joining the super source and the source nodes to place in-kilter.

KILTER 1 - Keeping track of nodes in the path between node k and its orphan root, in KILTER 0.

KILTER 2 - Traveling by level, in KILTER 0.

KILTER 3 - Saving the list of the boundary arcs in the first pass of the cost changes for the second pass; also, using a queue structure to store nodes for starting the traveling procedure, in KILTER 0.

KILTER 4 - Saving a list of arcs during labeling which include all boundary arcs, in KILTER 3.

KILTER 5 - Working on a single out-of-kilter arc between the super sink and super source, in KILTER 3.

KILTER 6 - Working on out-of-kilter arcs between the sinks and the super sink in KILTER 3.

KILTER 7 - Using arc data ordered by start node, in KILTER 3.

KILTER 8 - Using a non-zero initial feasible solution, in KILTER 3.

KILTER 9 - Primal-Dual, in KILTER 3.

path between node k and its corresponding orphan root as in KILTER 1 has only a small effect on the solution time (about 5%), but requires one more logical node-length array for storage. The labeling time when traveling by level as in KILTER 2 is almost twice the labeling time for KILTER 0. Though the number of flow changes are less, the time required to find each augmeting path is more; as we mentioned before, the procedure seems to be building and manipulating larger augmenting trees at considerable computational cost.

Comparing the total time taken to make cost changes in KILTER 3 (see also table 5.6) with the time taken by KILTER 0 indicates that saving the list of the boundary arcs through the first pass of the cost changes causes a small improvement in the solution time. Using the queue structure for keeping the nodes to start the traveling procedure, however, reduces total labeling time by approximately 50%. This great reduction in the solution time justifies the additional storage required (one-tenth of node-length vector) by this option.

The phase 2 and 3 experiments, as summarized in Tables 5.6 - 5.8, use an improved version of the NGC generator and the basic KILTER 0 code. The code KILTER 3 was improved even further when passing from phase 2 to phase 3.

As Tables 5.6 and 5.7 indicate, storing a set of arcs, that include all of the boundary arcs, during the labeling process for use in the cost stage segment of the code as in KILTER 4 increases solution time since more work is needed for labeling.

Table 5.7 also shows that working only on a single out-of-kilter arc between the super sink and super source node as in KILTER 5, is almost twice as slow as KILTER 3. Although the number of cost changes is very small for this option, the time for each cost change iteration is considerably longer. It is possible that the tree becomes more expanded in KILTER 5 since most of the artificial arcs between the super source and source nodes are in the tree. The solution time increases if we use the artificial arcs between the sinks and the super sink node as the only out-of-kilter arcs as in KILTER 6. The solution time then increases by a factor of 4 (see Tables 5.5 and 5.7).

Using arc data sorted in order of their start nodes is advantageous from the viewpoint of both solution time and saving one arc-length array of storage. Table 5.7 shows the improvement in the solution time for this option as implemented in KILTER 7. Using this option to solve a problem posed with unsorted data is not beneficial, however, because the sorting time is almost half the total solution time.

In the KILTER 8 option, we started with a non-zero initial feasible solution. Although there is a reduction in the number of flow changes, the solution time increases by a factor of 2 (see Table 5.7). Most of the other recent codes recommend starting with a non-zero initial feasible solution such as the Modified Row Minimum Rule.

Table 5.8 reports some results for comparing KILTER 9, an implementation of the primal-dual algorithm, with KILTER 3. Using KILTER 9 reduces solution time by roughly 20%; in particular, the code greatly improves upon running times for transportation problems. Also, as we mentioned in the last section, this option has the advantage of requiring less storage, replacing one arc-length array by one node-length array and one logical arc-length array.

To observe how the KILTER code performs with network problems arising in applications, we experimented with five capacitated network problems described in Table 5.9. The first two small problems are from Bell Laboratories and problems 3,4, and 5 are three energy problems. The number of flow and cost changes for problems 4 and 5 are very high when contrasted with comparably sized problems created by the network generator (particularly problem 9 in Table 5.2). Although these problems are not very large, they are hard to solve because of the high values for the total supplies and the large ranges for the cost values. Other researchers [24,45] have also noted that these problem characteristics lead to more difficult problems.

TABLE 5.5: Solution Time (Sec.) on Multics for Problem 1

Feature	Total Solution Time	Labeling Time	Flow Changes Time	Cost Changes Time	No. of Flow Changes	No. of Cost Changes
KILTER 0	17.92	9.32	2.26	6.24	401	344
KILTER 1	16.99	9.01	2.04	5.95	401	344
KILTER 2	25.21	17.28	1.89	6.05	387	343
KILTER 3	13.37	5.16	2.09	6.11	403	348
KILTER 4	47.06				336	230

TABLE 5.6: Solution Times (sec.) On Multics for Selected Problems

Features	KILTER 0				KILTER 3				KILTER 4				KILTER 5				KILTER 5				
Prob. No.	TST	LT	No.FC / FCT	No.CC / CCT	TST	LT	No.FC / FCT	No.CC / CCT	TST	LT	No.FC / FCT	No.CC / CCT	TST	LT	No.FC / FCT	No.CC / CCT	TST	LT	No.FC / FCT	No.CC / CCT	Obj. Value Function
1	10.02	5.42	355 / 1.7	276 / 2.9	9.94	5.35	355 / 1.70	276 / 2.89	14.75	7.52	355 / 3.30	276 / 3.93					54.79		301	218	1,214,978
4	11.82	5.52	197 / 1.20	512 / 5.10	8.35	2.35	197 / 1.20	512 / 4.80	8.87	2.52	197 / 1.70	512 / 4.65									6,276
5					10.04	3.00	197 / 1.20	434 / 5.84	11.28	3.38	197 / 1.80	434 / 6.10									3,147
7	34.61	18.86	133 / 0.72	296 / 15.03	24.99	9.54	133 / 0.70	296 / 14.55	25.19	10.73	133 / 3.40	296 / 10.96	24.66	11.69	114 / 0.58	173 / 14.39					41,965,423
8	26.81	13.72	53 / 0.28	256 / 12.53	17.32	4.51	53 / 0.28	255 / 12.53													44,614,329
9					54.55	20.54	224 / 1.66	877 / 32.36													82,685,944

TST = Total Solution Time

LT = Labeling Time

No.FC / FCT = Number of Flow Changes / Flow Changes Time

No.CC / CCT = Number of Cost Changes / Cost Changes Time

TABLE 5.7: Solution Times (sec.) On IBM 370/168 For Selected Problems

fea-ture	KILTER 3			KILTER 4			KILTER 5			KILTER 7				KILTER 8			
Prob. No.	Total Solution Time	No. of Flow Changes	No. of Cost Changes	Total Solution Time	No. of Flow Changes	No. of Cost Changes	Total Solution Time	No. of Flow Changes	No. of Cost Changes	Total Solution Time	Sort-ing Time	No. of Flow Changes	No. of Cost Changes	Total Solution Time	No. of Flow Changes	No. of Cost Changes	Objective Value Function
1	2.61	325	261	3.17	325	261	5.08	289	76	2.44	1.1	325	261	5.83	198	247	1,385,334
4	2.81	196	538							2.80	1.94	196	538				Infeasible
5	5.12	199	481											9.25	44	504	3,094
6	6.10	200	432	6.95	200	432*	10.98	197	59	6.00	2.57	200	432				2,089

TABLE 5.8: Solution Times (sec.) On IBM 370/168 For NGC and NETGEN

NGC GENERATOR								NETGEN GENERATOR			
Feature	KILTER 3			KILTER 9				KILTER 9			
Problem Number	Solution Time	No. of Flow Changes	No. of Cost Changes	Solution Time	No. of Flow Changes	No. of Cost Changes	Objective Value Function	Solution Time	No. of Flow Changes	No. of Cost Changes	Objective Value Function
1	2.61	325	261	2.10	319	261	1,385,334	2.85	364	305	2,153,303
2	12.45	558	345	9.44	546	345	907,550	10.48	827	367	2,046,034
3	15.40	616	348	10.91	577	348	836,847	12.97	917	359	2,155,354
4	2.80	196	538	2.71	228	573	Infeasible	2.34	200	508	4,563
5	5.12	199	481	4.66	231	517	3,094	3.89	200	429	3,070
6	6.1	200	283	5.91	232	468	2,089	5.29	200	383	2,721
7	7.60	120	283	6.60	119	283	30,891,536	5.81	103	290	44,682,512
8	3.48	52	213	3.12	52	213	29,829,216	6.23	74	298	56,506,992
9	18.40	192	864	16.07	186	864	82,910,560	10.88	160	861	82,560,096

TABLE 5.9: Solution Times (sec.) on IBM 370/168 for Some Real Problems

Prob. No.	No. of Nodes	No. of Arcs	Total Supply	Cost Range		No. of Flow Changes	No. of Cost Changes	Solution Time	Objective Value
				Min.	Max.				
1	35	95	602,875	-3423	+6072	37	59	.09	6,304,116,740
2	63	209	1,041,815	-3065	+119682	62	79	.20	6,567,632,900
3	114	588	354,622,200	.000000	.999999	105	98	1.47	42,492,976
4	586	3675	2,000,798,000	.000000	.999999	520	711	60.23	207,624,864
5	709	4837	2,000,498,000	.000000	.999999	594	874	82.62	207,788,048

VI. CONCLUSIONS

Using data structures to recall labeling information from previous iterations, KILTER runs faster, and uses less storage, than the efficient out-of-kilter code SUPERK. In terms of running time, the KILTER code is competitive with the primal network code PNET as implemented in the early 1970's, particularly for assignment problems. KILTER uses more arc length arrays for storage than primal based codes, but uses less node length array storage.

Of the options tested, the primal-dual algorithm equipped with a storage facility to keep track of nodes from which labeling can be extended after a cost change gives the best running times. In addition, the primal-dual algorithm uses less storage than other implementations of the out-of-kilter method.

Labeling options and out-of-kilter selection rules have had a pronounced effect upon solution time. A "last-labeled first-scanned" labeling option has outperformed a "first-labeled first-scanned" option. Moreover, when adding artificial arcs, first selecting those out-of-kilter arcs to bring in-kilter that join the super source to the source nodes has performed better than other arc selection options.

Our experimentation indicates that the out-of-kilter algorithm is very sensitive to minor changes in implementation. Certain minor improvements in the code are possible, such as imbeding the labeling option of KILTER 1 within KILTER 9 to reduce running time. In light of this sensitivity to implementation, further experimentation may lead to greater improvements. Our current feeling, though, is that of the codes designed for general minimum cost network flow problems, recent primal simplex based codes may outperform primal-dual approaches when solving a problem from scratch. We are not sure about comparisons between these types of algorithms for other types of applications, though, or for codes specially tailored for certain classes of problems such as the assignment and transportation problems discussed by Hatch.

Acknowledgements

We are grateful to S. Chen and R. Saigal of Bell Laboratories and to R. Marsten of M.I.T. for supplying us with applications for our computational experiments.

REFERENCES

1. Aashtiani, H., "Solving Large Scale Network Optimization Problems By the Out-of-Kilter Method", M.S. Thesis, Operations Research Center, M.I.T., February 1976.

2. Balas, E. and P. L. Hammer, "On the Transportation Problem - Parts I and II", Cahiers du Centre d'Etudes de Recherche Operationelle, 4(2), 1962, pp. 98-116 and 4(3), 1962, pp. 131-160.

3. Balinsky, M. L. and R. E. Gomory, "A Primal Method for the Assignment and Transportation Problems", Management Science, 10, 1964, pp. 578-593.

4. Barr, R. S., F. Glover, and D. Klingman, "An Improved Version of the Out-of-Kilter Method and a Comparative Study of Computer Codes", Math. Prog., 7(1), 1974, pp. 60-87.

5. Bradley, G. H., "Survey of Deterministic Networks", AIIE Transactions, 7(3), 1975, pp. 222-234.

6. Bradley, G. H., G. G. Brown, and G. W. Graves, "A Comparison of Some Storage Structures for Network Codes", ORSA Bulletin, 23(1), 1975, B-115.

7. Bradley, G. H., G. G. Brown, and G. W. Graves, "Tailoring Primal Network Codes to Classes of Problems with Common Structure", ORSA Bulletin, 23(2), 1975, B-386.

8. Busacker, R. G. and P. J. Gowens, "A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns", ORO Tech. Report 15, Opers. Res. Office, Johns Hopkins Univ., 1961.

9. Charnes, A., F. Glover, D. Karney, D. Klingman, and J. Stutz, "Past, Present, and Future of Large Scale Transportation and Transshipment Computer Coes", Computers and Operations Research 2(2), 1975, pp. 71-81.

10. Clasen, R. J., "The Numerical Solution of Network Problems Using the Out-of-Kilter Algorithm", RAND Corporations Memorandum, RM-5456-PR, Santa Monica, California, March 1968.

11. Dantzig, G. B., "Application of the Simplex Method to a Transportation Problem", in Activity Analysis of Production and Allocation, T.C.Koopmans (ed.,), Wiley and Sons, New York (1951), pp. 395-373.

12. Dantzig, G. B., Linear Programming and Extensions, Princeton University Press, 1963.

13. Dennis, J. B., "A High-Speed Computer Technique for the Transportation Problem", JACM, 8, 1958, pp. 132-153.

14. Edmonds, J. and R. M. Karp, "Theoretical Improvement in Algorithmic Efficiency for Network Flow Problems", JACM, 19(2), 1972, pp. 248-264.

15. Flood, M. M., "A Transporation Algorithm and Code", Naval Research Logistics Quarterly, 8, 1961, pp. 257-275.

16. Ford, L. R., Jr. and D. Fulkerson, Flow in Networks, Princeton University Press, 1962.

17. Fulkerson, D. R., "Flow Networks and Combinatorial Operations Research", American Mathematical Monthly, 73(2), 1966, pp. 115-138.

18. Fulkerson, D. R., "An Out-of-Kilter Method for Solving Minimal-Cost Flow Problems", J. Soc. Indust. Appl. Math., 9, 1961, pp. 18-27.

19. Gilsinn, J. and C. Witzgall, "A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees", NBS Technical Notes 772, 1973.

20. Glover, F., D. Karney, and D. Klingman, "The Augmented Predecessor Index Method for Locating Stepping Stones Paths and Assigning Dual Prices in Distribution Problems", Transportation Science, 6(1), 1972, pp. 171-180.

21. Glover, F., D. Karney, and D. Klingman, "Implementation and Computational Comparisons of Primal, Dual, and Primal-Dual Computer Codes for Minimum Cost Network Flow Problems", Research Report CS 136, Center for Cybernetic Studies, University of Texas, BEB-512, Austin, 1973.

22. Glover, F., D. Karney, D. Klingman, and A. Napier, "A Computation Study on Start Procedures, Basis Change Criteria and Solution Algorithms for Transportation Problems", Management Science, 20(5), 1974, pp. 793-819.

23. Glover, F. and D. Klingman, "Locating Stepping Stone Paths in Distribution Problems Via the Predecessor Index Method", Transportation Science, 6(2), 1970.

24. Glover, F. and D. Klingman, "Capsule View of Future Research on Large Scale Network and Network-Related Problems", Workshop on Integer Programming, Bonn, September 8-12, 1975.

25. Glover, F. and D. Klingman, "Double-Pricing Dual and Feasible Start Algorithms for the Capacitated Transportation (Distribution) Problems", University of Texas, Austin, 1970.

26. Glover, F., D. Klingman, and J. Stutz, "Augmenting Threaded Index Method for Network Optimization", INFOR, 12, 1974, pp. 293-298.

27. Golden, B., "Shortest Path Algorithms: A Comparison", Working Paper OR 044-75, Opers. Res. Center, M.I.T., Oct. 1975 (to appear in Opns. Res.).

28. Golden, B. and T. Magnanti, "Deterministic Network Optimization: A Bibliography", Working Paper OR 054-76, Opers. Res. Center, M.I.T., June 1975.

29. Golden, B., T. Magnanti, H. Nguyen, "Implementing Vehicle Routing Algorithms", Technical Report TR-115, Operations Research Center, M.I.T., September 1975.

30. Graves, G. W. and R. D. McBride, "The Factorization Approach to Large Scale Linear Programming", Working Paper No. 208, Western Management Sci. Inst., U.C.L.A., August 1973 (to appear in Math. Prog.).

31. Hatch, R. S., "Bench Marks Comparing Transportation Codes Based on Primal Simplex and Primal-Dual Algorithms", Opns. Res., 23(6), 1975, pp. 1167-1172.

32. Johnson, E. L., "Networks and Basic Solutions", Opns. Res., 14, 1966, pp. 89-95.

33. Johnson, E. L., "Network Flow", in S.E.Elmaghraby and J.J.Moder (eds.), Operations Research Handbook (to appear).

34. Johnson, E. L., "On Shortest Paths and Sorting", Proc. of 1972 ACM Conference, Boston, August 1972, pp. 510-517.

35. Karney, D. and D. Klingman, "Implementation and Computational Study on an In-Core Out-of-Core Primal Network Code", University of Texas, August, 1973.

36. Kershenbaum, A., "Computing Capacitated Minimal Spanning Trees Efficiently", Networks, 4(4), 1974, pp. 299-310.

37. Kershenbaum, A. and R. M. Van Slyke, "Computing Minimum Spanning Trees Efficienctly", Proceedings of the 25th Annual Conference of ACM, Boston, 1972, pp. 518-527.

38. Klein, M., "A Primal Method for Minimum Cost Fows with Applications to the Assignment and Transportation Problems", Management Science, 14(3), 1967, pp. 205-220.

39. Klingman, D., A. Napier, and G. Ross, "A Computational Study of the Effects of Problem Dimensions on Solution Time for Transportation Problem", JACM (to appear).

40. Klingman, D., A. Napier, and J. Stutz, "NETGEN - A Program for Generating Large Scale (un)capacitated Assignment, Transportation and Minimum Cost Flow Network Problems", Management Science, 20(5), 1974, pp. 814-822.

41. Knuth, D., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1975.

42. Lawler, E., Combinatorial Optimization: Networks and Matroids, Holt, Rinehart, and Winston (to appear).

43. Lee, S., "An Experimental Study of the Transportation Algorithms", M.S. Thesis, Graduate School of Business, University of California at Los Angeles, 1958.

44. Magnanti, T., "Optimization for Sparse Systems", in Sparse Matrix Computations (J. R. Bunch and D. J. Rose, eds.), Academic Press, New York, 1976, pp. 147-176.

45. Mulvey, J. M., "Special Structures in Network Models and Associated Applications", Ph.D. Thesis in Management, University of California at Los Angeles, 1975.

46. Mueller-Merbach Heiner, "An Improved Starting Algorithm for the Ford-Fulkerson Approach to the Transportation Problems", Management Sicence, 13(1), September 1966, pp. 97-104.

47. Napier, A., "A Computer Generator for Uncapacitated Networks", Unpublished report.

48. "OPHELIE II: Mathematical Programming System", Control Data Corporation, Minneapolis, Minnesota, 1970.

49. "OPHELIE/LP: Linear Programming Subsystem of OPHELIE II", Control Data Corporation, Minneapolis, Minnesota, 1970.

50. "Out-of-Kilter Network Routine", SHARE Distribution 3536, SHARE Distribution Agency, Hawthorne, New York, 1967.

51. Pape, U., "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem", Math. Prog., 7, 1974, pp. 212-222.

52. Pla, J.-M., "An Out-of-Kilter Algorithm for Solving Minimum Cost Potential Problems", Math. Prog., 1, 1971, pp.275-590.

53. Potts, R. and R. Oliver, Flows in Transportation Networks, Academic Press, New York, 1972.

54. Scions, H. I., "The Compact Representation of a Routed Tree and the Transportation Problem", International Sym. on Math. Prog., London, 1964.

55. Shapiro, J. F., "A Note on the Primal-Dual and Out-of-Kilter Algorithms for Network Optimization", Working Paper OR 040-75, Opers. Res. Center, M.I.T., March 1975 (to appear in Networks).

56. Simmonard, Michel, Linear Programming (William S. Jewell, trans.), Prentice-Hall, 1966.

57. Srinivasan, V. and G. L. Thompson, "Benefit-Cost Analysis of Coding Techniques for the Primal Transportation Algorithm", JACM, 20, 1973, pp. 194-213.

58. Yen, J., "Finding the Lengths of All Shortest Paths in N-node Non-negative Distance Complete Networks Using $\frac{1}{2}N^3$ Additions and N^3 Comparisons", JACM, 19(3), 1972, pp. 423-424.