A Scaling Algorithm for Multicommodity Flow Problems

by
Rina R. Schneur
James B. Orlin

WP #3808-95     April 1995
A SCALING ALGORITHM FOR MULTICOMMODITY FLOW PROBLEMS

Rina R. Schneur
PTCG Inc.
Burlington, Massachusetts

James B. Orlin
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract
We present a penalty-based algorithm that solves the multicommodity flow problem as a sequence of a finite number of scaling phases. In the $\varepsilon$-scaling phase the algorithm determines an $\varepsilon$-optimal solution, that is, one in which complementary slackness conditions are satisfied to within $\varepsilon$. We analyze the performance of the algorithm from both the theoretical and practical perspectives.
1. **Introduction**

Multicommodity flow problems arise whenever commodities, vehicles, or messages are to be shipped or transmitted simultaneously from certain origins to certain destinations along arcs of an underlying network. These problems find applications in the study of urban traffic, railway systems, logistics, communication systems, and many other areas as well (see for example Ali et al. [1984], Schneur [1991], Ahuja, Magnanti and Orlin [1993]). The multicommodity flow problem is a generalization of the single commodity network flow problem in which different commodities share a common network, and interact with each other typically through common capacity constraints. In the absence of any interaction among the commodities, the problem can be solved as separate single commodity flow problems. The interaction between the commodities, however, requires one to solve all the single commodity problems concurrently and, therefore, makes multicommodity flow problems much more difficult to solve than the corresponding set of single commodity problems.

Linear multicommodity flow problems can be formulated as linear programs and solved by the simplex algorithm or by interior point methods. The best time bound for linear programming problems is due to Vaidya [1989]. Many applications of multicommodity flow problems, however, lead to linear programs which are too large to solve by a direct application of linear programming software. Researchers have therefore developed specialized adaptations of linear programming algorithms which exploit the special structure and the sparsity inherent in multicommodity network flow problems. Three "classical" approaches to multicommodity flow problems are price-directive decomposition, resource-directive decomposition, and partitioning. These approaches are all based on the simplex method. Assad [1978], Kennington [1978], and Ali, Helgason, Kennington and Lall [1980] review these methods and their computational performance. Recent research has focused mainly on the development of interior point methods, the development of parallel algorithms, and the solution of large-scale problems. Examples of interior point algorithms may be found in Choi and Goldfarb [1989], and Bertsimas and Orlin [1991]. Multicommodity flow problems reveal inherent parallelism. Choi and Goldfarb [1989] as well as Pinar and Zenios [1990] exploited this characteristic in their parallel algorithms. The latter uses linear and quadratic penalty functions.

In this paper we present a scaling-based approximation algorithm. A solution $x$ is called $\varepsilon$-optimal if it is possible to perturb some cost coefficients or capacity constraints by at most $\varepsilon$ units so that the solution $x$ is optimal for the perturbed problem. The scaling approach may be summarized as follows. Let $P(\Delta)$ be a problem which is different by at most $\Delta$ from the original problem $P$. Start by solving $P(\Delta)$ for a sufficiently large $\Delta$, such that $P(\Delta)$ is relatively easy to solve. Then, use the solution to $P(\Delta)$ to solve $P(\Delta/F)$ for some scaling factor $F>1$, and modify $\Delta$ to be $\Delta/F$. Keep iterating until the value of $\Delta$ is sufficiently small. Edmonds and Karp [1972] and Dinic [1973] developed the first scaling algorithms for the minimum cost single commodity network flow problem. Gabow [1985] showed the application of scaling to several problems, including the shortest path, matching and maximum flow problems. Scaling algorithms which use the idea of $\varepsilon$-optimality are theoretically the most efficient algorithms for various network flow problems. Prominent among these are single commodity network flow problems such as minimum cost flow.
(Goldberg and Tarjan [1987], Ahuja, Goldberg, Orlin and Tarjan [1992], and Orlin [1988]) and maximum flow (Ahuja, Orlin and Tarjan [1989]). In this paper we develop a cost scaling algorithm approach to solve multicommodity network flow problems. Other researchers have developed \( \varepsilon \)-optimal algorithms (the traditional notion) for multicommodity flow problems. Such approximation algorithms are described and analyzed in the papers by Grigoriadis and Khachiyan [1991], Klein, Agrawal, Ravi, and Rao [1990], Leighton et. al. [1991], Klein, Plotkin, Stein, and Tardos [1990], Shahrokhi and Matula [1990]. The last two focus on the maximum concurrent flow problem which is a special case of the multicommodity flow problem. In all these cases the focus of the research has been on obtaining good worst case bounds. Our research focuses more on computational performance in practice.

Our algorithm is simple, and yet robust. It solves the multicommodity flow problem as a sequence of penalty problems, each of which is constructed by relaxing the capacity constraints and adding a term for their violation to the objective function. (For detailed description of penalty methods see Fiacco and McCormick [1968].) Each penalty problem is solved to \( \varepsilon \)-optimality using a network-based scaling algorithm. Since the parameters of real-world multicommodity problems, such as cost and capacity, are typically approximate in practice, the algorithm presented here often finds an optimal solution up to the accuracy of the data. The main component of the algorithm consists of moving flow around cycles. Thus, the algorithm focuses on cycles rather than on paths.

The efficiency of the algorithm is a result of using the scaling approach and of exploiting the network structure of the problem. In subsequent sections we present convergence results and prove that the algorithm has some interesting theoretical characteristics. The computational testing provides insight into the behavior of the algorithm and shows that the algorithm is quite efficient and worthy of further consideration. The computational results also reveal that in some cases the theoretical bounds are observed in practice, while in other cases the bounds are much more conservative than the practical performance. The testing shows that the run time of the algorithm is competitive with the run time of recent algorithms for large scale multicommodity flow problems. These algorithms by Barnhart [1993], Barnhart and Sheffi [1993], and Farvolden, Powell and Lustig [1993] are based on dual ascent methods, the primal-dual approach and partitioning method respectively.

The general framework of our algorithm can be used to solve different types of multicommodity flow problems. A variation of it also be used for various network flow problems with side constraints of which the multicommodity flow problem is a special case (see Schneur [1991]). In this paper we focus on the algorithm for the minimum cost multicommodity flow problem. In the next section, we apply a penalty method to the multicommodity flow problem, and formulate it as a penalty problem. We also present in this section a detailed description of the algorithm. In Section 3, we develop the optimality conditions for the penalty problem, and in Section 4, we introduce and prove some of its theoretical properties. Numerical results are presented in Section 5, and in Section 6 we briefly discuss possible extensions of our algorithm to other network flow problems and make some concluding remarks.
2. The Scaling Algorithm for the Penalty Problem

We consider linear multicommodity flow problems on directed networks. For each commodity there is an associated vector of supply/demand. We observe that a multicommodity flow problem on an undirected network with multiple sources and multiple sinks for each commodity can be transformed to a single source-single sink multicommodity flow problem on a directed network. This can be done by adding a super-source and a super-sink node for each commodity. The super-source node is connected to each supply node of that commodity with an outgoing arc from the super-source node whose capacity equals the supply at the node and whose cost is zero. Similar arcs are added from each demand node of that commodity to the super-sink node. The capacity of each such arc equals the demand at the node and its cost is zero. Henceforth, we consider only cases in which there is a single source node and a single sink node for each commodity.

Let \( N \) be a set of \( n \) nodes and \( A \) be a set of \( m \) directed arcs, which together form a network \( G=(N,A) \). There are \( K \) commodities sharing the capacity \( u_{ij} \) of each arc \((i,j)\) in the network. For each commodity \( k \) there is a required flow of \( b_k \) units from its source node \( s(k) \) to its sink node \( t(k) \). The cost of a unit flow of commodity \( k \) on arc \((i,j)\) is \( c_{ij}^k \) and the amount of flow is denoted as \( x_{ij}^k \). Let \( d_i^k \) be the flow balance of commodity \( k \) at node \( i \). Thus, \( d_i^k = b_k \) if \( i=s(k) \); \( d_i^k = -b_k \) if \( i=t(k) \); and \( d_i^k = 0 \) otherwise.

Using these notations, the minimum-cost multicommodity flow problem[MM] may be formulated as follows:

\[
\begin{align*}
\text{Min} & \sum_{(i,j) \in A} \sum_{k=1}^{K} c_{ij}^k x_{ij}^k \\
\text{s.t.} & \sum_{j:(i,j) \in A} x_{ij}^k - \sum_{j:(j,i) \in A} x_{ji}^k = d_i^k \quad \forall i \in N, \forall k=1,\ldots,K \\
& \sum_{k=1}^{K} x_{ij}^k \leq u_{ij} \quad \forall (i,j) \in A \\
& x_{ij}^k \geq 0 \quad \forall (i,j) \in A, \forall k=1,\ldots,K
\end{align*}
\]

The objective function (1) is to minimize the total shipping cost; constraints (2) are referred to as the supply/demand constraints; constraints (3) are called the capacity constraints or the bundle constraints; and constraints (4) are the non-negativity constraints.
In order to formulate the multicommodity flow problem using a penalty function method, we relax the capacity bundle constraints and add a penalty function term of their violation to the objective function.

Let \( e_{ij}(x) = \max \{0, (\sum_{k=1}^{K} x_{ij}^{k} - u_{ij}) \} \) be the amount by which the total flow on arc \((i,j)\) exceeds its capacity. A quadratic penalty function with a penalty parameter \( \rho \) is associated with each violated capacity constraint and a penalty term \( \rho (e_{ij}(x))^2 \) is added to the objective function.

The resulting penalty problem \([\text{PMM}(\rho)]\) is:

\[
\text{Min} \ (f_{\rho}(x) = \sum_{(i,j) \in A} \sum_{k=1}^{K} c_{ij}^{k} x_{ij}^{k} + \sum_{(i,j) \in A} \rho (e_{ij}(x))^2 )
\]

s.t.
\[
\sum_{j: (i,j) \in A} x_{ij}^{k} - \sum_{j: (j,i) \in A} x_{ji}^{k} = d_{i}^{k} \quad \forall i \in N, \forall k=1,...,K \tag{2}
\]
\[
x_{ij}^{k} \geq 0 \quad \forall (i,j) \in A, \forall k=1,...,K \tag{4}
\]

After relaxing the bundle constraints, the remaining constraints in \([\text{PMM}(\rho)]\) decompose into the constraints of \(K\) single commodity flow problems. The objective function, however, is non-separable and nonlinear. Hence, we eliminate the complicating constraints, but introduce nonlinear (convex) and non-separable terms into the objective function.

A sequence of solutions to the penalty problem with an increasing penalty parameter converges to the optimal solution of the original problem as long as the penalty parameter increases without bound (Avriel [1976] and Luenberger [1984]). Penalty methods using the Hessian to solve the penalty problems may fail when the penalty parameter is too large because the Hessian is ill-conditioned (Avriel [1976]). We present a scaling algorithm which utilizes the special network structure of the penalty problem and does not require the computation of the Hessian. The algorithm finds an approximately optimal solution to the penalty problem and successively improves the quality of the solution at each iteration.

Let \( Q^{k} \) denote an undirected cycle of commodity \( k \). Since \( G=(N,A) \) is a directed network but \( Q^{k} \) is an undirected cycle, not all the arcs in \( Q^{k} \) follow the same direction. When we send flow around a cycle we send it in a particular direction. We call the arcs that follow that direction \textit{forward arcs} and the arcs that are opposite to the direction of the flow \textit{backward arcs}. To send \( \delta \) units of flow around \( Q^{k} \) is to increase the flow on each forward arc of \( Q^{k} \) by \( \delta \) units and to decrease the flow on each backward arc of \( Q^{k} \) by \( \delta \) units. We refer to this flow as a \( \delta \)-flow around \( Q^{k} \) and denote it as \( y(\delta, Q^{k}) \). The cost of a \( \delta \)-flow around \( Q^{k} \) is the net change in the objective function \( f_{\rho} \), obtained by sending the flow around the cycle. That is, it is \( f_{\rho}(x + y(\delta, Q^{k})) - f_{\rho}(x) \). If \( f_{\rho}(x + y(\delta, Q^{k})) - f_{\rho}(x) < 0 \), we refer to \( Q^{k} \) as a \textit{negative cost} \( \delta \)-cycle with respect to \( f_{\rho}(x) \), and we refer to \( y(\delta, Q^{k}) \) as a \textit{negative cost} \( \delta \)-flow with respect to \( f_{\rho}(x) \). In order to detect these cycles, we build a residual
network (also called the \(\delta\)-residual network) which is constructed as follows. Let arc \((i,j)^t\) represent an arc from node \(i\) to node \(j\) in the residual network. For each arc \((i,j)\) of the original network we potentially have two oppositely directed arcs in the residual network: the forward arc \((i,j)^t\) whose cost represents increasing the flow on arc \((i,j)\) by \(\delta\) units; the backward arc \((j,i)^t\) whose cost represents decreasing the flow on arc \((i,j)\) by \(\delta\) units; the backward arc \((j,i)^t\) is included in the \(\delta\)-residual network of commodity \(k\) only if we can decrease the flow of commodity \(k\) on arc \((i,j)\) by \(\delta\) units, that is, if \(x^k_{ij} \geq \delta\). Using our notations (\(c\) is the cost vector, \(x\) is the flow and \(e\) is the excess of flow), we can write the cost of each arc on the \(\delta\)-residual network as follows.

The cost of each forward arc \((i,j)\) on the residual network for commodity \(k\), \(\bar{c}^k_{ij}\), is the net change in \(f_p\) obtained by increasing the flow on arc \((i,j)\) by \(\delta\) units. That is:

\[
\bar{c}^k_{ij} = c^k_{ij}\delta + \rho(2\delta e^k_{ij}(x) + \delta^2) \quad \forall (i,j) \in A: e^k_{ij}(x) \geq 0
\]  \(6\)

\[
\bar{c}^k_{ij} = c^k_{ij}\delta + \rho(\delta + \sum_{k \in K} x^k_{ij} - u^k_{ij})^2 \quad \forall (i,j) \in A: -\delta < \sum_{k \in K} x^k_{ij} - u^k_{ij} < 0
\]  \(7\)

and

\[
\bar{c}^k_{ij} = c^k_{ij}\delta \quad \forall (i,j) \in A: \sum_{k \in K} x^k_{ij} - u^k_{ij} \leq -\delta
\]  \(8\)

The cost of each backward arc \((j,i)\) on the residual network for commodity \(k\), \(\bar{c}^k_{ji}\), is the net change in \(f_p\) obtained by decreasing the flow on arc \((i,j)\) by \(\delta\) units. That is:

\[
\bar{c}^k_{ji} = -c^k_{ji}\delta + \rho(-2\delta e^k_{ji}(x) + \delta^2) \quad \forall (j,i) \in A: e^k_{ji}(x) \geq \delta
\]  \(9\)

\[
\bar{c}^k_{ji} = -c^k_{ji}\delta - \rho(e^k_{ji}(x))^2 \quad \forall (j,i) \in A: 0 < e^k_{ji}(x) < \delta
\]  \(10\)

and

\[
\bar{c}^k_{ji} = -c^k_{ji}\delta \quad \forall (j,i) \in A: \sum_{k \in K} x^k_{ij} - u^k_{ij} \leq 0
\]  \(11\)

For example, consider a part of the network as illustrated in Figure 1(a). The numbers in [ ] represent the unit cost for commodity \(k\) on each arc, and the numbers in ( ) represent the excess of flow on each arc (which is a result of the flow of commodity \(k\) as well as the other commodities in
the problem). The calculation of the costs of arc (1,3), when arc (1,3) is either a forward arc on a δ-cycle, or arc (1,3) is a backward arc on the cycle (that is, arc (3,1) of the residual network) are shown in Figure 1(b) for δ=2.0 and ρ=1.0. The costs of the other arcs in the network are calculated in the same way and appear in Figure 1(c). The (clockwise) cycle 1-2-3-1 is a negative cost δ-cycle. Sending 2 units of flow around that cycle means to shift 2 units of flow from path 1-3 to path 1-2-3. That improves the penalty objective function by 2.0 units.

The scaling algorithm for solving the penalty problem consists of repeatedly sending flow around negative cost cycles. At each scaling phase, δ has a fixed value and at each iteration we send δ units of flow around a negative cost δ-cycle of some commodity. When there is no negative cost δ-cycle we decrease the value of δ by a factor of 2. (Dividing δ by a factor other than 2 leads to the same theoretical analysis as the one presented in the next section, and may be a reasonable alternative in practice.) At the end of each phase, we also increase the value of the penalty parameter by some scaling factor less that 2.

Figure 1a: A residual network
Figure 1b: Calculating the cost of the residual arcs

\[ 2 + 16 + 4 = 22 \]
\[ -2 - 16 + 4 = -14 \]

Figure 1c: A negative cost \( \delta \)-cycle

\[ \delta = 2.0 \]
\[ \rho = 1.0 \]

\[ +6 \quad -2 \quad -2 \quad +6 \]
\[ +22 \quad -14 \]

cost of negative \( \delta \)-cycle = -2.0

We say that a flow \( x \) is \((\delta,\rho)\)-optimal for the penalty version of the multicommodity flow problem if \( x \) is a feasible flow for the penalty problem and there is no negative cost \( \delta \)-cycle with
respect to the function $f_\rho$. We present the following Theorem here since it helps to understand the motivation behind our algorithm. We prove it in the next section.

**Theorem 1:**

A flow $x$ is optimal for PMM($\rho$) if and only if $x$ is $(\delta,\rho)$-optimal for all sufficiently small positive values of $\delta$.

The scaling algorithm for the penalty problem can be viewed as a nonlinear programming algorithm in which the step size at each phase is fixed at a value of $\delta$. We try to find a “direction” (a vector of flow modifications) such that by moving $\delta$ units in this direction the penalty objective function decreases. When we can not find an improving direction with a step size $\delta$, we decrease the step size to $\delta/2$. An alternative approach, in which we first detect the improving direction and then determine the step size, is described in Schneur [1991]. In the remainder of this section we outline the scaling algorithm.

The scaling algorithm for the minimum cost multicommodity flow problem may be described in the general form of a scaling algorithm as follows.

**Given:**

- $c_{ij}^k$ = cost of unit flow on arc $(i,j)$ for commodity $k$
- $u_{ij}^k$ = capacity of arc $(i,j)$
- $b^k$ = amount of supply/demand of commodity $k$.

**Objective:**

Find a minimum cost flow which satisfies the flow requirements for each commodity without violating the capacity constraints.

**SAM.M:** a Scaling Algorithm for Multicommodity Minimum cost flow problems.

```plaintext
begin
determine an initial feasible solution $x$;
choose initial values for $\delta$ and $\rho$;
until $\delta$ and $\rho$ satisfy the termination criteria do
begin
while there is a negative cost $\delta$-cycle with respect to $f_\rho$ do
begin
find a negative cost $\delta$-cycle $Q^k$ for some commodity $k$ with respect to $f_\rho$;
send $\delta$ units of flow around $Q^k$;
end
$\delta$: = $\delta/2$; $\rho$: = $\rho*R$; ($R$ is a constant with $1<R<2$)
end
end
```
We now discuss some practical issues related to the various elements of the algorithm.

**Initial solution:**
An initial solution may be any solution which satisfies the supply/demand constraints. For example, one can satisfy the supply/demand of each commodity by sending $b_k$ units on the shortest path from $s(k)$ to $t(k)$. We henceforth assume that we initialize the solution in this way.

**Parameter setting:**
In our implementation we let the initial value of $\delta$ be the size of the largest demand rounded up to the nearest power of 2, i.e., $\delta_0 = 2^\lceil \log B \rceil$, where $B = \max\{b_k^k: k=1,...,K\}$. Since $\delta$ is halved at each successive scaling phase, within $\lceil \log B \rceil$ additional phases $\delta=1$, a fact needed in the subsequent analysis. The best initial penalty parameter value and its modification rate, $R$, may be empirically determined. Observe that $\delta$ and $\rho$ are modified simultaneously. For theoretical purposes which are discussed later, we need $\delta \cdot \rho$ to decrease by a constant factor in each phase. Since $\delta$ decreases by a factor of 2 at each phase, we require that $R < 2$. We have found $\rho_0=0.3$ and $R \in [1.6,1.7]$ to be good choices for the problems that we have tested.

**Termination rules for the algorithm:**
The termination criteria of the algorithm depend on the values of $\delta$ and $\rho$. (In cases in which the dual solution is obtained, the criteria may also depend on the value of the duality gap.) The final values of $\delta$ and $\rho$ are chosen to ensure that $\delta$ is “sufficiently” small and $\rho$ is “sufficiently” large. In practice, one may bound the penalty parameter value from above in order to help maintain numerical stability or to ensure faster termination of the algorithm. We let $\rho_u$ denote our upper bound on $\rho$.

For each solution to the penalty problem, we also derive a solution to the dual of the penalty problem. The duality gap is the difference between the value of the penalty objective function and the value of this dual solution. We refer to this dual solution and this duality gap as the induced dual solution and the induced duality gap, respectively. We describe these terms in details in Section 4. The induced duality gap may be used in the termination criteria.

Let $\text{GAP}$ be the current value of the induced duality gap, and let $g$ and $\varepsilon$ be small positive values set by the user. Then, the algorithm terminates when one of the following conditions is satisfied:

- $\delta \cdot \rho \leq \varepsilon$ or
- $\text{GAP} \leq g$ and $\rho = \rho_u$.

The values of $\delta \cdot \rho$ and the duality gap indicate how close the solution is to optimality. The lower bounds on those values provide the user the option of terminating with a pre-specified level of solution quality.
When we search for negative cycles we concentrate at one commodity at a time. Thus, one may say that the algorithm uses the concept of coordinate optimization methods. We comment on the selection of commodities and the search for improving cycles in Section 5.

Our scaling algorithm for the penalty problem is similar to the "Discretized Descent Algorithm" described in Section 9 of Rockafellar [1984] for solving convex cost network flow problems. Rockafellar describes an algorithm in which the step size is fixed and flow is sent around cycles. He also suggests solving the problem iteratively with a decreasing sequence of step sizes. If the algorithms are viewed at a high level, the primary differences between Rockafellar's algorithm and ours are that we use this algorithm within the framework of a penalty method in order to solve multicommodity flow problems; we decrease the step size by a fixed factor of two, and we simultaneously increase the penalty parameter by a fixed multiplier at each phase. At a more detailed level, the two approaches differ in the choice of negative cycle detection algorithms and in the analysis of the convergence, errors and running time. We have also performed an extensive computational testing to support our theoretical results and to investigate the behavior of the algorithm in practice.

Algorithm SAM.M assumes that there is a feasible solution to the multicommodity problem in hand. To determine if the problem is indeed feasible, we run a version of the scaling algorithm which is specialized to the feasibility multicommodity flow problem. The algorithm and the way it detects infeasibility are described briefly in Section 6 and in more details in Schneur [1991]. Algorithm SAM.M is illustrated in Figure 2.

![Figure 2: A flow chart of algorithm SAM.M](image-url)
We have developed also an optimal-shift version of the scaling algorithm (Schneur [1991]). The algorithm is motivated by Frank-Wolfe gradient-based algorithm for quadratic programming (Frank and Wolfe [1956]). Among other problems, Frank-Wolfe algorithm is used to solve traffic equilibrium multicommodity flow problems (Sheffi [1976]). The optimal-shift algorithm may be viewed as a network-based scaling version of Frank-Wolfe algorithm. While in algorithm SAM.M we first determine the step size and then look for an improving direction, the optimal-shift scaling algorithm is more similar to conventional nonlinear algorithms. First the direction of improvement is determined. Then, the optimal step size along this direction is calculated. A direction of improvement is found by detecting a negative cost cycle in the residual network of each commodity. The costs of the residual network of commodity k represent the derivative of the penalty objective function with respect to the flow of that commodity on each arc. This is a different residual network than the one used in algorithm SAM.M. Once a negative cycle is detected, we shift the optimal amount of flow around it, i.e., the amount which results in the maximum improvement of the penalty objective function. The optimal amount is the amount which drives the cycle derivative cost to zero. The optimal amount to shift on a cycle can be calculated using a binary search technique.

3. Optimality Conditions

The optimality conditions for the penalty problem are derived from the Kuhn-Tucker optimality conditions for nonlinear programming. Since the objective function is convex and the constraints are linear, the Kuhn-Tucker conditions are sufficient for optimality (see, for example, Avriel [1976]).

For a general nonlinear programming problem \( \{ \min f(x): q(x)=0, x \geq 0 \} \), the Kuhn-Tucker optimality conditions for a feasible solution \( x \) are (\( \Delta \) denotes derivative):
\[
x(\Delta_x f) + \mu \Delta_x q \geq 0 \quad \text{and} \quad \Delta_x f + \mu \Delta_x q \geq 0
\]
for some vector \( \mu \).

Applied to a feasible solution for the penalty problem \([\text{PMM}(\rho)]\), the optimality conditions for each arc \((i,j)\) and commodity \(k\) are:
\[
x^k_{ij}(c^k_{ij} + 2 \rho e^k_{ij}(x) + \mu_i - \mu_j) = 0 \quad \text{and} \quad c^k_{ij} + 2 \rho e^k_{ij}(x) + \mu_i - \mu_j \geq 0
\]
for some values of \( \mu_i \) and \( \mu_j \).

Let \( \theta^k_{ij} \) be the reduced cost of arc \((i,j)\) and commodity \(k\) which is defined as
\[
\theta^k_{ij} = c^k_{ij} + 2 \rho e^k_{ij}(x) + \mu_i - \mu_j
\]
(14)

The optimality conditions for a feasible solution \( x \) may now be written as
\[
x^k_{ij} > 0 \implies \theta^k_{ij} = 0
\]
(15)
and
The optimality conditions indicate that in the optimal solution, flow is assigned only to arcs whose reduced costs equal 0.

**Lemma 1:**

Let \( x \) be a \((\delta, \rho)\)-optimal solution to the penalty problem. Then, there exists a vector \( \mu \) of simplex multipliers, such that the following conditions hold:

\[
x^k_{y} = 0 \implies \theta^k_{y} \geq 0.
\]  

(16)

We first claim that there is no negative cost cycle within the \( \delta \)-residual network for commodity \( k \). To see this claim, let \( y \) denote a unit flow around some cycle \( Q \) in the \( \delta \)-residual network for commodity \( k \), and consider \([f_p(x + \delta y) - f_p(x)]\) along that cycle. The contribution from each forward arc \((i,j)^f\) of the \( \delta \)-residual network of commodity \( k \):

\[
h^k_{ij} = c^k_{ij} + 2\rho e_{ij}(x) + \delta \rho
\]

and for each backward arc \((j,i)^f\) of the \( \delta \)-residual network of commodity \( k \):

\[
h^k_{ji} = -c^k_{ij} - 2\rho e_{ij}(x) + \delta \rho.
\]

(19)

(20)

Let \( \mu^k_i \) denote the negative of the length of the shortest path from the source node of commodity \( k \) to node \( i \) with respect to the arc costs \( h^k_{ij} \). We assume without loss of generality that the original network is strongly connected, and hence the residual network for each commodity is strongly connected. Given that and the fact that there is no negative cost cycle, \( \mu^k_i \) is well-defined.

By the optimality conditions for the shortest path problem (see, for example, Ahuja, Magnanti and Orlin [1993]) it follows that for each arc \((i,j)^f\)

\[
h^k_{ij} + \mu^k_i - \mu^k_j \geq 0
\]

(21)

and thus

\[
c^k_{ij} + 2\rho e_{ij}(x) + \delta \rho + \mu^k_i - \mu^k_j \geq 0.
\]

(22)

Using the definition in (14), it follows from (22) that \( \theta^k_{y} \geq -\delta \rho \) for each arc \((i,j)^f\) in the \( \delta \)-residual network for commodity \( k \). When \( x^k_{y} \leq \delta \) arc \((i,j)^f\) is the only residual arc between \( i \) and \( j \), and thus the condition in (18) is proved. Moreover, if \( x^k_{y} \geq \delta \), then both \((i,j)^f\) and \((j,i)^f\) are in the \( \delta \)-residual
network, and thus $\theta^k_{i,j} = -\theta^k_{j,i} \geq -\delta \rho$. It follows that for arcs where $x^k_{i,j} \geq \delta$, $-\delta \rho \leq \theta^k_{i,j} \leq \delta \rho$ and thus the condition is (17) is proved as well.

We have shown in Lemma 1 that at the end of each $\delta \rho$-phase, when there are no negative cost $\delta$-cycles, we can bound the violation of the optimality condition by $\delta \rho$. That is, the algorithm finds an $\varepsilon$-optimal solution to the penalty problem with $\varepsilon = \delta \rho$. When the flow values are multiples of $\delta$ (in the algorithm, for example, this occurs when the demand of each commodity is integral and $\delta = 1/2^n$ for some integer $n$) the $\delta$-residual network is the same as the residual network. In Lemma 1 we can replace the conditions in (17)-(18) by

$$x^k_{i,j} > 0 \implies -\delta \rho \leq \theta^k_{i,j} \leq \delta \rho \quad \text{(23)}$$

and

$$x^k_{i,j} = 0 \implies -\delta \rho \leq \theta^k_{i,j}. \quad \text{(24)}$$

**Corollary 1:**

Suppose that $x$ is a $(\delta,\rho)$-optimal solution and $x^k_{i,j}$ is a multiple of $\delta$ for all $i, j, k$. Then there exists a marginal cost vector $p$ with $|p^k_{i,j}| \leq \delta \rho$ for all $i, j, k$ such that $x$ is an optimal flow for the penalty problem with the cost vector $c$ replace by $c+p$.

**Proof:**

The Kuhn-Tucker conditions are satisfied by $x$ if $c$ in (23)-(24) is replaced by $c+p$, where $p$ is defined as follows. For arc $(i,j)$ and commodity $k$ for which $x^k_{i,j} > 0$ as in (23), $p^k_{i,j} = \theta^k_{i,j}$. For arc $(i,j)$ and commodity $k$ for which $x^k_{i,j} = 0$ as in (24), $p^k_{i,j} = -\min\{0, \theta^k_{i,j}\}$. In each case, $|p^k_{i,j}| \leq \delta \rho$, and the Kuhn-Tucker conditions are satisfied for vector $\mu$.

We are now ready to prove Theorem 1. Recall the Theorem as stated in the previous section.

**Theorem 1:**

A flow $x$ is optimal for the penalty problem $\text{PMM(}\rho\text{)}$ if and only if $x$ is $(\delta,\rho)$-optimal for all sufficiently small positive values of $\delta$.

**Proof:**

By the definition of a $(\delta,\rho)$-optimal solution, if $x$ is not $(\delta,\rho)$-optimal for some positive value of $\delta$, then $x$ is infeasible or there is a cost-improving $\delta$-cycle. In either case, $x$ is not optimal for the penalty problem, and thus the **only if** part of the theorem is true. Suppose conversely that $x$ is $(\delta,\rho)$-optimal for all positive values of $\delta$. Let the vectors $h^k, \mu^k$ and $\theta^k$ be defined as in the proof of Lemma 1, but with the value $\delta=0$. The same proof shows that the optimality conditions are satisfied by $\theta$ and $\mu$, and with $\delta=0$ it shows that $x$ is optimal. Thus, the **if** part of the theorem is true as well.

\[\square\]
4. The Lagrangian Dual and Performance Analysis

In this section, we provide worst case performance for algorithm SAM.M. In Section 4.1 we focus on deviation from optimality. In Section 4.2 we focus on worst case time complexity. To simplify the analysis, we assume that the supply/demand values are integral for all the commodities. We start with $\delta = 2^{\lceil \log B \rceil}$ and for the sake of the analysis, we assume that the algorithms runs at least until $\delta \leq 1$. The following lemma shows that under these assumptions all the flows are multiples of $\delta$ in the phases where $\delta \leq 1$, and therefore the optimality conditions in (23)-(24) are satisfied.

**Lemma 2:**

a) If a flow $x_{ij}^k$ is a multiple of $\delta$ at the beginning of a $\delta$-scaling phase, then it is a multiple of $\delta'$ during any subsequent $\delta'$-scaling phase of the algorithm.

b) The flows during the algorithm are integral multiples of $\delta$ on all for which the initial flow is 0.

c) When $\delta \leq 1$, all the arc flows are integral multiples of $\delta$.

**Proof:**

a) Suppose that $x_{ij}^k$ is the flow at the beginning of a $\delta$-scaling phase, and it is an integral multiple of $\delta$. Let $y_{ij}^k$ be the flow after one additional phase. If flow is sent on $(i,j)^k$, then $\delta$ units of flow are sent and $y_{ij}^k$ is also an integral multiple of $\delta$. If $\delta$ is halved, then $y_{ij}^k$ is also an integral multiple of the revised $\delta$. Thus once the flow on $(i,j)^k$ is an integral multiple of $\delta$, it continues to be an integral multiple of $\delta$ at all subsequent phases.

b) Part (a) shows that once a flow is an integral multiple of $\delta$, it remains an integral multiple of $\delta$ for the rest of algorithm. Therefore all arcs $(i,j)^k$ whose initial flow is 0 have flows that are integral multiple of $\delta$.

c) Since we start the algorithm with $\delta = 2^{\lceil \log B \rceil}$ and halve the value of $\delta$ at each phase, after $\lceil \log B \rceil$ phases we have $\delta = 1$. Since the supply/demand values are integers, the same argument as in part (a) shows that all the arc flows are integral at each phase. So when $\delta = 1$, all flows are integral multiple of $\delta$. Using the results in (a), we conclude that at any subsequent phase when $\delta \leq 1$, all the flows are multiples of $\delta$.

4.1 The Lagrangian dual and the duality gap

Let $\sigma_{ij}$ be the slack variable of the bundle capacity constraint of arc $(i,j)$. The penalty problem (2),(4),(5) may be rewritten as:

\[ \text{[PMM}(\rho)\text{]} \]

\[
\begin{align*}
\text{Min} & \quad \left( \sum_{(i,j) \in A} \sum_{k=1}^{K} c_{ij}^k x_{ij}^k + \sum_{(i,j) \in A} \rho (e_{ij})^2 \right) \\
\text{s.t.} & \quad \sum_{j \in (i,j) \in A} x_{ij}^k - \sum_{j \in (j,i) \in A} x_{ji}^k = d_i^k, \quad \forall i \in N, \forall k=1,\ldots,K
\end{align*}
\]
\[ \sum_{i=1}^{k} x_{ij}^k + \sigma_{ij} - e_{ij} = u_{ij} \quad \forall (i,j) \in A \quad (25) \]

\[ e_{ij} \geq 0, \sigma_{ij}^k \geq 0 \quad \forall (i,j) \in A \quad (26) \]

\[ x_{ij}^k \geq 0 \quad \forall (i,j) \in A, \forall k=1,...,K \quad (4) \]

Constraints (25) are derived from the definitions of \( e_{ij} \) and \( \sigma_{ij} \). Let \( \lambda_{ij} \) be the dual variables of constraints (25), and let \( H(c, \rho, x, \lambda) \) be defined as follows:

\[ H(c, \rho, x, \lambda) = \sum_{(i,j) \in A} \sum_{k=1}^{\lambda} c_{ij}^k x_{ij}^k + \sum_{(i,j) \in A} \rho (e_{ij})^2 + \sum_{(i,j) \in A} \lambda_{ij} \left( \sum_{k=1}^{\lambda} x_{ij}^k + \sigma_{ij} - e_{ij} - u_{ij} \right) \quad (27) \]

We now relax constraints (25) and obtain the Lagrangian relaxation:

**[LPMM(\rho, \lambda)]**

\[ v(c, \rho, \lambda) = \text{Min} \{ H(c, \rho, x, \lambda) \} \quad (28) \]

s.t.

\[ \sum_{j \neq (i,j) \in A} x_{ij}^k - \sum_{j \neq (i,j) \in A} x_{ji}^k = d_i^k \quad \forall i \in N, \forall k=1,...,K \quad (2) \]

\[ e_{ij} \geq 0, \sigma_{ij}^k \geq 0 \quad \forall (i,j) \in A \quad (26) \]

\[ x_{ij}^k \geq 0 \quad \forall (i,j) \in A, \forall k=1,...,K \quad (4) \]

The Lagrangian dual problem can be formulated as:

**[LDPMM(\rho)]**

\[ \text{Max}_{\lambda} \left[ v(c, \rho, \lambda) \right] \quad (29) \]

The problem \( \text{LPMM}(\rho, \lambda) \) is separable with respect to the variables \( \sigma, e \) and \( x \). For each fixed value of \( \lambda \), the solution to \( \text{[LPMM}(\rho, \lambda)] \) is therefore determined as follows:

* \( \sigma_{ij} \) is chosen to minimize \( \sum_{(i,j) \in A} \sigma_{ij} \lambda_{ij} \) subject to \( \sigma_{ij} \geq 0 \). Thus, \( \sigma_{ij} \) is set to 0 if \( \lambda_{ij} \geq 0 \), and it can be arbitrarily large if \( \lambda_{ij} < 0 \). Consequently, if \( \lambda_{ij} < 0 \) for some arc \((i,j)\), then \( v(c, \rho, \lambda) = -\infty \);

* \( e_{ij} \) is chosen to minimize \( \sum_{(i,j) \in A} (\rho (e_{ij})^2 - \lambda_{ij} e_{ij}) \) subject to \( e_{ij} \geq 0 \). Thus, \( e_{ij} \) is chosen such that \( e_{ij}(2\rho e_{ij} - \lambda_{ij}) = 0 \). That is, either \( e_{ij} = 0 \), or \( e_{ij} = \lambda_{ij}/2\rho \);
\( x^*_k \) is chosen to minimize \( \sum_{i=1}^{k} \sum_{j \in A} (c_i^k + \lambda_{ij}) \) subject to (2),(4). Thus, \( x^*_k \) is the flow resulting from sending the demand of each commodity along the shortest path from its source to its sink, with respect to the costs \( (c_i^k + \lambda_{ij}) \).

Note that for each fixed value of \( \lambda \), the time required to determine the value of the Lagrangian objective function is (at most) the time required to solve K shortest path problems.

Let \( x \) be a solution to the penalty problem PMM(\( \rho \)) and let \( e \) be the vector of excess generated by \( x \). Associated with the primal solution \( (x,e) \) is a dual solution \( (\lambda^0, e^0, x^0, \sigma^0) \), determined as follows:

* \( \lambda^0 = 2pe \);
* \( e^0 = e \);
* \( \sigma^0 \) is the slack variable of the capacity bundle constraint;
* \( x^0 \) is the flow which results from sending each commodity along the shortest path from its source to its sink with respect to the costs \( (c + 2pe) \).

We have selected \( \lambda^0 \) to be a vector such that the optimal choice of \( e^0 \) is \( e^0 = \lambda^0/2\rho = e \). Also \( x^0 \) is the flow which results from sending each commodity along the shortest path with respect to the costs \( (c + \lambda^0) \). In addition, the slack variables may be positive only for arcs with a zero excess, and therefore \( \lambda^0 \sigma^0 = 2pe \sigma^0 = 0 \). Thus, \( (e^0, x^0, \sigma^0) = (e, x^0, \sigma^0) \) solves the Lagrangian problem [LPMM(\( \rho, \lambda^0 \))] . We refer to this associated dual solution as the induced dual solution.

The value of the induced dual solution \( v(c, \rho, \lambda^0) \) can be calculated as:

\[
\nu(c, \rho, \lambda^0) = \sum_{i,j \in A} \left( \sum_{k=1}^{k} c_i^k + 2\rho e_{ij} \right) x^*_k - 2\rho e_{ij} u_{ij} \quad (30)
\]

The value of the primal solution \( z(c, x, \rho) \) is:

\[
z(c, x, \rho) = \sum_{i,j \in A} \left( \sum_{k=1}^{k} c_i^k x^*_k + \rho (e_{ij})^2 \right) \quad (31)
\]

Thus, we can calculate the induced duality gap \( \text{GAP}(c, \rho) \) as:

\[
\text{GAP}(c, \rho) = z(c, x, \rho) - \nu(c, \rho, \lambda^0) \quad (32)
\]

The Lagrangian dual defined here is only one way to define the dual problem. In addition there are various methods to solve that problem (e.g. subgradient method). Here we present a very simple and fast method for computing a dual solution. The method appears to be very good in practice (see the results of Section 5), and is useful in proving bounds. We next give some bounds on the size of the induced duality gap. This allows us to use the value of the induced duality gap as a possible termination criterion and for performance analysis.
Theorem 2:  
If \( x = x^* \) is an optimal solution to the penalty problem \( PMM(\rho) \), then the induced duality gap is zero.

Proof:  
Let \( x \) be a feasible solution to \( PMM(\rho) \) and let \( (\lambda^0, e^0, x^0, \sigma^0) \) be the induced dual solution. In general \( x^0 \neq x \) (\( x \) is any feasible solution to the penalty problem). However, when \( x \) is the optimal solution \( x^* \), the optimality conditions (23)-(24) hold with \( \delta = 0 \). For each commodity the optimality conditions become the same as the conditions for the shortest path problem with costs \( c + 2\rho e \). Thus in this case \( x^0 = x^* \). By substituting \( \sum_{i=1}^{j} x_{ij}^k - u_{ij} \) by \( e_{ij} \) in (30) we get \( v(c, \rho, \lambda^*) = z(c, x^*, \rho) \), and so the duality gap is zero.

In the next Theorem we develop an upper bound on the value of the induced duality gap for any \((\delta, \rho)\)-optimal solution.

Theorem 3:  
Suppose that \( y \) is a \((\delta, \rho)\)-optimal solution determined by the algorithm when \( \delta \leq 1 \). Then the value of the induced duality gap is at most \( 2\delta \rho nD \), where \( D = \sum_{k=1}^{n} b^k \) is the sum of all the demands.

Proof:  
Let \( y \) be a \((\delta, \rho)\)-optimal solution to the penalty problem for some value of \( \delta \) which is at most 1, and let \( e^* \) and \( \sigma^* \) denote the excess and slack vectors associated with \( y \), respectively. Let \( \lambda_{ij}^* = 2\rho e_{ij}^* \). Let \( (\lambda^*, w, e^*, \sigma^*) \) denote the induced dual solution. We now want to evaluate \( H(c, \rho, y, \lambda^*) - H(c, \rho, w, \lambda^*) \).

Let \( \theta \) and \( \mu \) be vectors chosen as in the proof of Lemma 1. Thus \( \theta_{ij} = e_{ij}^k + \lambda_{ij}^k + \mu_i^k - \mu_j^k \) for all arcs \((i, j)^k\). Moreover, by Lemma 1,

\[
\begin{align*}
\theta_{ij}^k &> 0 \quad \Rightarrow \quad -\delta \rho \leq \theta_{ij}^k \leq \delta \rho \\
\theta_{ij}^k &< 0 \quad \Rightarrow \quad \theta_{ij}^k \leq -\delta \rho \\
\theta_{ij}^k &< 0 \quad \Rightarrow \quad \theta_{ij}^k \leq \delta \rho
\end{align*}
\]

and

\[
\begin{align*}
\theta_{ij}^k &< 0 \quad \Rightarrow \quad \theta_{ij}^k \leq \delta \rho \\
\theta_{ij}^k &< 0 \quad \Rightarrow \quad \theta_{ij}^k \leq \delta \rho
\end{align*}
\]

The effect of including the dual variables term \( \sum_{(i, j) \in A} \sum_{k=1}^{n} y_{ij}^k (\mu_i^k - \mu_j^k) \) in the objective function is only a constant which is equal to \( b^k \mu_{t(k)}^k \) for each commodity, where \( t(k) \) is the sink node of commodity \( k \). This is because the flow of each commodity can be decomposed into paths from \( s(k) \) to \( t(k) \) (\( s(k) \) is the source node of commodity \( k \)), \( b^k \mu_{s(k)}^k = 0 \), and the dual variable terms at all other nodes cancel out. Thus the value \( H(c, \rho, y, \lambda^*) \) is equal to

\[
\sum_{(i, j) \in A} \sum_{k=1}^{n} \theta_{ij}^k y_{ij}^k + \sum_{(i, j) \in A} \rho (e_{ij}^*)^2 + \sum_{(i, j) \in A} \lambda_{ij}^k (\sigma_{ij}^* - e_{ij}^* - u_{ij}) - \sum_{k=1}^{n} b^k \mu_{t(k)}^k
\]

and therefore
\[ H(c, \rho, y, \lambda^*) - H(c, \rho, w, \lambda^*) = \sum_{(i,j) \in A} \sum_{k=1}^{n} \theta_{ij}^k (y_{ij}^k - w_{ij}^k) \]

\[ \leq \sum_{(i,j) \in A} \sum_{k=1}^{n} [\delta \rho (y_{ij}^k) - \delta \rho (w_{ij}^k)] \]

\[ < \sum_{k=1}^{n} 2\delta \rho n b^k \leq 2\delta \rho n D \]

The second to last inequality follows from the fact that for each commodity \( k \), the flow in \( y \) (and also in \( w \)) may be written as the sum of at most \( b^k \) units of flow on paths, each of which has fewer than \( n \) arcs. (in fact, one may restrict attention to the case in which the flow is \( b^k \) units along a single path.)

4.2 Worst Case Computational Analysis

In order to derive a worst case bound, we limit the negative cycle search in the algorithm to \( \delta \)-cycles whose mean cost is at most \(-\delta^2 \rho \). The mean cost of each cycle is greater than \(-\delta^2 \rho \) if and only if adding \( \delta^2 \rho \) units to the cost of each arc results in no negative cost cycle. Since we always send \( \delta \) units of flow around a cycle, one way to avoid detecting cycles whose mean cost is greater than \(-\delta^2 \rho \) (and thus limit the search to cycles whose mean cost is at most \(-\delta^2 \rho \)) is to add \( \delta \rho \) to the unit cost of each arc. When we replace the unit cost vector \( c \) by \( c + \delta \rho \) the bound on the optimality condition violation in Lemma 1 becomes \( 2\delta \rho \). Also, in Corollary 1 the \((\delta, \rho)\)-optimal solution is optimal for costs \( c + \rho \) where \(|\rho| \leq 2\delta \rho \). As a result, the bound on the induced duality gap in Theorem 3 becomes \( 4\delta \rho n D \).

**Lemma 3:**

Suppose that we limit the search in the algorithm to \( \delta \)-cycles whose mean is at most \(-\delta^2 \rho \). Then the improvement in the objective function after each iteration is at least \( 2\delta^2 \rho \).

**Proof:**

When \( \delta \) units are sent around a cycle \( Q \) whose mean cost is at most \(-\delta^2 \rho \), the improvement in the objective function which equals the total cost of that cycle is at least \( \delta^2 \rho |Q| \), where \(|Q| \) is the number of arcs in the cycle. Since there are at least two arcs in each cycle, the improvement after each \( \delta \)-shift is at least \( 2\delta^2 \rho \).

**Theorem 4:**

When we limit the search in the algorithm to \( \delta \)-cycles whose mean is at most \(-\delta^2 \rho \), the number of negative cost cycles found in each phase is at most \( 2nD/\delta \).

**Proof:**

From the results of Theorem 3 modified to the case when the mean cost of each cycle is at most \(-\delta^2 \rho \), we get that the bound on the induced duality gap is at most \( 4\delta \rho n D \). Since the improvement per sending flow around a negative \( \delta \)-cycle is at least \( 2\delta^2 \rho \), the number of negative cost cycles detected in each phase is bounded by \( 4\delta \rho n D / 2\delta^2 \rho = 2nD/\delta \).
Let $\xi$ be a lower bound on the value of $\delta$. Then the bound on the total number of negative cycles throughout the algorithm is $2nD/\xi = O(nD/\xi)$.

Schneur [1991] presents a more detailed analysis of the bounds on the number of algorithmic operations. Under some (not very limiting) assumptions, we can show that the number of operations performed by algorithm SAM.M is $O(n^2mK^2/\xi^2)$. Note, however, that this bound and the bound of Theorem 4 are not as good as the ones given by Klein, Plotkin, Stein, and Tardos [1990].

5. Computational Results

We have performed extensive computational tests and have analyzed algorithm SAM.M on a variety of problem instances. These tests have included the investigation of different implementation ideas and the testing of the sensitivity of the algorithm to various data parameters, such as the number of arcs, the number of commodities, and the congestion in the network. We have also compared the practical behavior of the algorithm, as observed in the computational tests, with the theoretical characteristics and bounds we have derived. These results are presented in Schneur[1991].

In this paper, we focus on results which highlight certain features of the algorithm, and provide some insight regarding its bottleneck operations and practical convergence. We also report and discuss running time results.

The test problem instances has been acquired from three sources:

(A) The published literature on multicommodity flow problems (Assad [1976]). These instances have medium size directed networks (up to 100 nodes and 200 arcs), with a relatively small number of commodities (up to 35).

(B) An industrial application which models a communication problem. The number of commodities in this problem is relatively large (around 600).

(C) Randomly generated problem instances. These instances were generated by RAM_GEN, a random multicommodity flow problem generator developed by us. We generated networks of various sizes, different number of commodities and different levels of congestion.

Each generated network has an underlying grid form. The length $L$ (the number of arcs in each horizontal line) and the height $H$ (the number of arcs in each vertical line) are user-defined. Each network is derived from an undirected network, that is, there are two oppositely directed arcs between each pair of connected nodes. Thus, $n = (L+1)*(H+1)$ and $m = 2*[ L^*(H+1) + H^*(L+1) ]$. To increase the number of arcs in some networks, the generator connects each node $i$ with two randomly selected nodes from the column on the right-hand side of node $i$. An example with $L=5$ and $H=3$ is illustrated in Figure 3. In this case $n$ remains the same as in the pure grid network, and $m = 2*[ L^*(H+1) + H^*(L+1) + 2*L^*(H+1)]$.
The cost on each arc is randomly chosen from a uniform distribution between 0.0 and a user-defined parameter \( c_{\text{max}} \).

The user sets the number of commodities, denoted as \( K \). Then, the generator randomly selects \( K \) source-sink pairs. The source and sink nodes can be unrestricted, or else limited only to the boundary nodes of the grid.

The demand for each commodity is randomly chosen from a uniform distribution between 0.0 and a user-defined parameter \( B_{\text{max}} \).

The capacity on each arc is randomly selected from a uniform distribution between bounds which depend on the size of the network, the number of commodities and the demand values.

![Network generated by RAM_GEN](image)

**Figure 3:** A network generated by RAM_GEN

A detailed description of the problem instances from all three sources is given in Schneur[1991].

5.1 **Bottleneck Operations and Negative Cycles Detection**

The bottleneck operation of the algorithm is the detection of negative cost \( \delta \)-cycles. We present here an average distribution of the time the algorithm spends on each type of operation it performs.

The algorithm consists of the following components:
- Initializing the variables and setting the parameters
- Calculating the costs of the residual arcs at the beginning of each phase
- Searching for a negative cost cycle until one is detected (successful iterations)
- Searching for a negative cost cycle and concluding that no such cycle exists (an unsuccessful iteration which occurs once at the end of each phase)
- Modifying the flow, the excess and cost of each arc at each iteration, when δ units are sent around a negative cost cycle.

![Pie chart showing distribution of running time](image)

**Figure 4: Distribution of the running time (average values)**

The distribution of the running time among the various types of operations in the algorithm is illustrated in Figure 4. The proportion of the running time for each part is the average based on the running time for instances from all three data sources. It appears that the time for calculating the cost at each phase is negligible. In addition, only 2% of the time is spent on modifying the flows excesses and costs on the arcs. Most of the time (about 96%) is spent on searching and detecting negative cost cycles. About 5% of this time is spent on unsuccessful iterations.

These results highlight the importance of having an efficient procedure for determining negative cost cycles. To detect negative cost cycles, we used a modification of the Label Correcting algorithm with a search on the predecessor tree which is simple and yet efficient (see Ahuja, Magnanti and Orlin [1993] for a description of this algorithm). In our implementation we found that checking for a negative cycle after few updates of the labels, rather than after each update, improves the total time for detecting a cycle. In addition, in order to decrease the number of searches for negative cycles, we save a collection C of cycles along which flow has been sent, but are likely to become negative again. We keep in C those cycles whose cost is smaller than some threshold value. After sending flow around a cycle, we update the cost of each cycle in C. Some of
these cycles may become negative, and if so are used in subsequent iterations. As long as there is a negative cycle in \( C \) we do not search for another negative cost cycle. The collection \( C \) uses a data structure that enables a fast update of the cycles' cost. Storing these cycles and updating their cost is time consuming, but the saving in overall cycles detection time more than compensates for this computational expense. In fact, in our computational testing about 50% of all negative cycles used for shifting flow were from the collection \( C \), while managing the collection added only about 3% to the running time cycles. Moreover, we limit our search only to "sufficiently negative" cycles in order to eliminate the detection of negative cost cycles with very small absolute cost, and improve the theoretical convergence of the algorithm (as shown in Section 4). Since a cycle corresponds to a particular commodity, we need to choose a commodity at each iteration. The order in which commodities are considered does not influence the theoretical worst case performance of the algorithm. It does affect, however, the performance of the algorithm in practice. Empirical results show that scanning the commodities in a cyclic order and detecting one cycle for each commodity provides a faster convergence than searching for all negative cycles for one commodity and then moving to the next one (Schneur [1991]).

5.2 Practical convergence

In any approximation algorithm, there are various ways of evaluating the quality of the solution. In general, one would want an algorithm based on a scaling and penalty function method to have the following properties:

1. As the penalty parameter increases and the scaling parameter decreases, the flow cost
   \[ \sum_{i = 1}^{\infty} \sum_{j \in A} (c_{ij} x_{ij}) \] associated with the algorithm's solution converges to the cost of the optimal solution to the original problem.

2. As the penalty parameter increases and the scaling parameter decreases, the penalty cost (excess cost) converges to zero, and thus the total cost (flow cost + excess cost) converges to the cost of the optimal solution to the original problem.

3. As the penalty parameter increases, the value of the maximum excess \( \max_{i,j} e_{ij}(x) \) converges to zero.

In our computational testing, we found the above properties to be typical to the behavior of our algorithm. In Figures 5 and 6, we illustrate a typical convergence, which corresponds to a problem instance from set (A). The results were similar for the problems in set (B) and set (C). The excess cost is the difference between the total cost and the flow cost in Figure 5.
Figure 5: The typical cost at the end of each $\delta \rho$-phase

Figure 6: The maximum excess at the end of each $\delta \rho$-phase

Theoretical analysis of penalty methods shows that the total cost and the flow cost both increase when $\rho$ increases. In our scaling algorithm, however, we modify $\delta$ and $\rho$ simultaneously such that $\delta \rho$ decreases. Since the solution at the end of each phase is $(\delta, \rho)$-optimal, we typically
have at each phase a solution that is closer to the optimal one for the corresponding penalty problem. In our tests the total cost usually decreases between phases until it finally converges to the optimal solution. Note that the excess cost is a product of an excess term and a penalty parameter. When the penalty parameter increases, we expect the excess to decrease, and we see from Figure 6 that this is indeed the case. This decrease, however, is not always sufficient to lead to a decrease in the penalty cost (which is the excess term multiplied by the penalty parameter). The flow cost may go up or down. An increased penalty parameter typically leads to an increase in the flow cost, whereas a decreased scaling parameter $\delta$ typically leads to a decrease in the flow cost. We observe that the flow cost and total cost both converge to the optimal solution value at the end of the algorithm and thus, the excess cost converges to zero.

A typical curve of the total cost at each iteration during the first four phases of the algorithm is illustrated in Figure 7. During each $\delta\rho$-phase of the algorithm, the total cost decreases at each iteration because flow is sent around negative cost cycles. When a phase terminates we increase the penalty parameter; therefore, the total cost which corresponds to the new penalty parameter increases. The total cost decreases again during the next $\delta\rho$-phase. Thus, the total cost at the end of a $\delta\rho$-phase is not necessarily smaller than the total cost at the end of the previous phase. Nevertheless, as illustrated in Figures 7, the total cost typically decreases between phases.

![Figure 7: The total cost at each iteration](image)

The scaling algorithm is a primal algorithm which concentrates on solving the primal problem, while the dual heuristic is somewhat naive. Therefore, the value of the primal solution for the penalty problem (after a number of phases) may be much closer to the optimal solution than is the value of the induced dual solution. The primal and dual costs are plotted in Figure 8. The
difference between the two graphs is the value of the duality gap, which gives us an upper bound on the distance of the solution from the optimal value. In general, we expect the duality gap to decrease between phases (this is also suggested by Theorem 3). From Figure 8 we see that this is typically the case.

![Figure 8: The duality gap at the end of each $\delta \rho$-phase](image)

5.3 Running time

We report here some numerical results for representative problems from the three different data sources. The results correspond to $\varepsilon$-optimal solutions, where the values of $\delta \rho$ and $\delta$ are bounded by user-defined values $\varepsilon$ and $\xi$, respectively. In addition to the size of the problem, the values of $\varepsilon$ and $\xi$, and the value of the optimal solution when available, Table 1 contains the following results:

* % $\delta$-saturated arcs. The percentage of saturated arcs at the $\varepsilon$-optimal solution usually reflects the congestion level of the problem (in term of demand-to-capacity ratio). For our purposes a saturated arc is an arc for which the flow is within $\delta$ units from the capacity (we call it a $\delta$-saturated arc).

* Costs. Once we know the optimal solution, we can compare it to the flow cost (FC) of the $\varepsilon$-optimal solution. Note that FC may be smaller than the optimal solution (by a small amount) since we have relaxed the capacity constraints. The excess cost (EC) which is the value of the penalty term, and its proportion of the total cost (TC=FC+EC), provide us information about the deviation from feasibility.
* **Maximum excess.** The maximum value of the excess \( e_{\text{max}} \) and the maximum excess-to-capacity ratio \( \left(\frac{e}{u}\right)_{\text{max}} \) represent the maximum violation of the capacity constraints.

* **# of searches.** The number of times we search for a negative cost cycle is highly correlated to the number of operations required to find the reported solution. This is because the search for negative cycles is the primary time-consuming operation in the algorithm (Figure 8).

* **# of iterations.** This is the number of flow shifts around negative cost cycles.

* **# of phases.** The number of \( \delta p \)-phases performed.

* **Running time.** The computational testing was performed in three environments:

1. A UNIX-based VAX Station 3100, model 30. The algorithm was coded using the 'C' compiler of the *Athena* project at the Massachusetts Institute of Technology. Running times were measured by the 'time()' function of the utilities library 'sys/times.h'.

2. A DOS-based PC IBM compatible, AT 386/33 Mhz. The algorithm was coded using Borland's 'Turbo C'. Running times were measured by the 'time()' function in the utility library 'time.h'.

3. An IBM RISC 6000, model 530. The algorithm was coded in 'C'. Running times were measured by the 'time()' function in the utility library 'time.h'.

Due to memory (RAM) limitations, we could not test the problems from data set B and some problems in data set C on the AT 386.
<table>
<thead>
<tr>
<th>Problem</th>
<th>A1.6</th>
<th>A3.7</th>
<th>B0</th>
<th>C25</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>47</td>
<td>85</td>
<td>49</td>
<td>150</td>
</tr>
<tr>
<td>m</td>
<td>98</td>
<td>204</td>
<td>260</td>
<td>1110</td>
</tr>
<tr>
<td>K</td>
<td>15</td>
<td>18</td>
<td>585</td>
<td>40</td>
</tr>
<tr>
<td>Opt.Sol.</td>
<td>497.00</td>
<td>2155.0</td>
<td>1182054</td>
<td>---</td>
</tr>
<tr>
<td>$\xi$</td>
<td>0.01</td>
<td>0.01</td>
<td>0.1</td>
<td>0.01</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.18</td>
<td>0.18</td>
<td>2.7</td>
<td>0.35</td>
</tr>
<tr>
<td>$%\delta$-sat.</td>
<td>21</td>
<td>11</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>FC</td>
<td>496.95</td>
<td>2153.7</td>
<td>1181770</td>
<td>2840.8</td>
</tr>
<tr>
<td>EC</td>
<td>0.02</td>
<td>0.98</td>
<td>252</td>
<td>1.7</td>
</tr>
<tr>
<td>TC</td>
<td>496.97</td>
<td>2154.7</td>
<td>1182023</td>
<td>2842.5</td>
</tr>
<tr>
<td>EC/TC</td>
<td>0.004</td>
<td>0.045</td>
<td>0.02</td>
<td>0.059</td>
</tr>
<tr>
<td>$\varepsilon_{\text{max}}$</td>
<td>0.017</td>
<td>0.096</td>
<td>2.22</td>
<td>0.077</td>
</tr>
<tr>
<td>$(\varepsilon/u)_{\text{max}}$</td>
<td>0.003</td>
<td>0.008</td>
<td>0.007</td>
<td>0.009</td>
</tr>
<tr>
<td># search.</td>
<td>57</td>
<td>1107</td>
<td>4010</td>
<td>4720</td>
</tr>
<tr>
<td># iter.</td>
<td>33</td>
<td>842</td>
<td>1086</td>
<td>2849</td>
</tr>
<tr>
<td>#δp-phases</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>$T(\text{sec})^{*}$</td>
<td>1/1/&lt;1</td>
<td>15/8/4</td>
<td>45/-/16</td>
<td>210/-/37</td>
</tr>
</tbody>
</table>

VAX Station 3100 / AT 386 / RISC 6000

Table 1: Results for representative problems

Additional interesting issues, such as how the results are affected by the parameters of the problem, are presented in Schneur [1991].
In order to evaluate the performance of the algorithm in terms of running time, we compared it to other algorithms for the same set of problem instances. To the best of our knowledge, there is no single set of problem instances which has been widely used for testing other multicommodity flow algorithms. A quite popular (and available) set of problems was that used initially by Assad (our set A). The algorithms for multicommodity flow problems with the theoretical performance are based on more general algorithms for linear programming. Among the "classical" decomposition and partitioning algorithms, Assad found price-directive decomposition (Dantzig-Wolfe) to have the best performance in practice, at least for his data set (Assad [1976]). A primal-dual network algorithm [PDN] for large-scale multicommodity flow problems has been developed by Barnhart [1988] and has been tested using Assad's test problems. A primal partitioning algorithm based on price directive decomposition [PPLP] (Farvolden and Powell [1991] and Farvolden, Powell and Lustig [1993]), aiming to solve large-scale problems in the Less-than-Truck-Load industry, has been tested on Assad's problems as well. This algorithm has also outperformed general purpose LP algorithms (MINOS and OB1). We compare our algorithm (as restricted to Assad's problem instances) with the corresponding results for PDN, PPLP, and a Dantzig-Wolfe (DW) implementation by Barnhart [1988]. These comparisons provide some sense for the relative computational performance of the algorithm. They are, however, limited in scope and do not provide a complete picture.

The algorithms DW and PDN have been programmed in VAX FORTRAN 4.5 and have been tested on a VAX Station II. PPLP has been programmed in FORTRAN 77 and tested on a Micro VAX Israel (Farvolden and Powell [1991]). The VAX was the common computational environment used by all algorithms. We need, however, to interpret our results based on the performance of the different VAX computational environments. According to performance comparison tables provided by Digital, the CPU performance of the VAX Station 3100 is 2.8 times higher than the CPU performance of the Micro VAX II, and the performance of the VAX Station II is at least as high as the one for the Micro VAX II. Therefore, the results for algorithm SAM.M on the VAX Station 3100 in Table 2 and Figure 9 are multiplied by 2.8. In addition, we provide the running time of algorithm SAM.M on the PC AT 386/33 Mhz. A special implementation of the algorithm on the PC AT 386 has led to further improvement in the running time of the algorithm. The only means of comparison between the PC or RISC 6000 and the other computers are MIPS (Million Instructions Per Second), which is the most common means of comparison in the computer industry. The PC AT 386/33 Mhz has about the same value of MIPS as the Micro VAX II (around 5 MIPS), and the RISC is about 7 times faster (34.5 MIPS). We feel, however, that the comparison with the VAX Station 3100 is a more accurate one, and therefore the other results in Table 2 are reported only for future comparisons.

This running time evaluation is approximate, and it is used only to provide a sense of the performance of our scaling algorithm relative to other recent algorithms and more traditional ones. There may be other factors which influence the running time, such as the amount of available RAM, the compiler and the programming language. In general, FORTRAN compilers may be better than 'C' compilers, but we do not have information about the specific compilers. The compiler optimizer has not been used for SAM.M, DW or PDN but may have been used for PPLP.
In addition, the other algorithms have found an optimal solution in the cases reported here, while the solutions found by algorithm SAM.M violate the capacity constraints by up to 0.2%-1.0%. While such violations may be acceptable in most applications, it may give our algorithm an unfair advantage.

The running time for the different algorithms are reported in Table 2 and illustrated in Figure 9a,b. All the problems in set A1 have 47 nodes and 98 arcs, and all the problems in set A3 have 85 nodes and 204 arcs. The total running time for all the problems is 83 seconds for algorithm DW, 85 seconds for algorithm PDN, 42 seconds for algorithm PPLP, 39 for the VAX implementation of algorithm SAM.M (after it is multiplied by 2.8), and 10 seconds for the PC implementation of algorithm SAM.M. The running time on the RISC 6000 was less than 1 second for all the instances.

<table>
<thead>
<tr>
<th>P</th>
<th>DW</th>
<th>PDN</th>
<th>PPLP</th>
<th>SAM.M 3100</th>
<th>SAM.M AT 3100*2.8</th>
<th>SAM.M AT 386</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1.1</td>
<td>6.3</td>
<td>5</td>
<td>2.4</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
</tr>
<tr>
<td>A1.2</td>
<td>7.4</td>
<td>5</td>
<td>2.0</td>
<td>1</td>
<td>2.8</td>
<td>0.5</td>
</tr>
<tr>
<td>A1.3</td>
<td>7.1</td>
<td>4</td>
<td>2.8</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
</tr>
<tr>
<td>A1.4</td>
<td>7.8</td>
<td>5</td>
<td>2.7</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
</tr>
<tr>
<td>A1.5</td>
<td>7.6</td>
<td>5</td>
<td>3.8</td>
<td>1</td>
<td>2.8</td>
<td>0.5</td>
</tr>
<tr>
<td>A1.6</td>
<td>9.6</td>
<td>4</td>
<td>3.5</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
</tr>
<tr>
<td>A3.2</td>
<td>14.2</td>
<td>32</td>
<td>11.7</td>
<td>5</td>
<td>14.0</td>
<td>2</td>
</tr>
<tr>
<td>A3.5</td>
<td>9.6</td>
<td>5</td>
<td>1.3</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
</tr>
<tr>
<td>A3.6</td>
<td>13.4</td>
<td>20</td>
<td>12.0</td>
<td>2</td>
<td>5.6</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Comparison of running time (sec)
Figure 9(a): Comparison of running time for set A1

Figure 9(b): Comparison of running time for set A3
6. **Extensions and Summary**

We have presented in this paper a simple and yet robust algorithm for multicommodity flow problems. The algorithm consists of solving a sequence of penalty problems by repeatedly sending flow around negative cost cycles. Two parameter control the solution process: the penalty parameter $\rho$ which specifies the cost of a unit penalty function for violating the capacity constraints, and the scaling parameter $\delta$ which govern the amount of flow sent around cycles. These parameters also control the maximum deviation from optimality at the end of each phase of the algorithm.

We have analyzed our algorithm from both theoretical and practical perspectives. The computational results seem not only to support the theoretical properties we have derived, but also to demonstrate that our algorithm has merit for solving multicommodity flow problems of various types and sizes.

This paper focuses on the scaling algorithm for minimum cost multicommodity flow problems. The general scheme, however, can be used to solve other types of multicommodity flow problems as well as network flow problem with side constraints of which the multicommodity problem is a special case. For example, the *feasibility multicommodity flow problem* is similar to the minimum cost multicommodity problem, except that all the unit flow costs are zero. As a result the penalty terms are the only components of the objective functions and the penalty parameter has no influence on the solution of the penalty problem and can be arbitrarily set to 1. Thus, only one penalty problem needs to be solved with a decreasing scaling parameter. The feasibility scaling algorithm may also be used as a subroutine in an algorithm which solves the *maximum concurrent flow problem*. This problem is a special case of multicommodity flow problems in which we need to send commodities along a capacitated network. In order to be "fair" to all commodities, the same ratio of each commodity's demand should be sent, and the objective is to maximize this ratio (Shahrokhi and Matula [1990]). Observe that for each fixed ratio in the maximum concurrent flow problem we can solve a feasibility problem to determine if this ratio is feasible. Thus, we can apply a binary search to determine the maximum feasible ratio, where a feasibility problem is solved at each iteration by algorithm. A detailed description and analysis of all these algorithms may be found in Schneuer[1991].

**Acknowledgment**

Many thanks to Charu Aggarwal and Hitendra Wadhwa for all their useful comments, and to Prof. Cindy Barnhart for providing us with data sets. Part of this work has been done under contract #ONR:N00014-94-1-0099.
References


