AN EFFICIENT IMPLEMENTATION OF A NETWORK INTERIOR POINT METHOD*

MAURICIO G.C. RESENDE† AND GERALDO VEIGA‡

Abstract. We describe DLNET, an implementation of the dual affine scaling algorithm for minimum cost capacitated network flow problems. The efficiency of this implementation is the result of three factors: the small number of iterations taken by interior point methods, efficient solution of the linear system that determines the ascent direction using a preconditioned conjugate gradient algorithm and strategies to produce an optimal primal vertex solution. The combination of these ingredients results in a code that can solve minimum cost network flow problems having hundreds of thousands of vertices in a few hours of running time on a workstation. We compare DLNET with network simplex code NETFLO and relaxation code RELAXT-3 on an extensive range of minimum cost network flow problems, including minimum cost circulation, maximum flow and transshipment problems.

Key words. Interior point algorithm, network flows, linear programming, computer implementation, simplex method, network simplex method, conjugate gradient.

1. Introduction. Consider a network with an underlying directed graph $G = (V, E)$, where V is a set of m vertices and E a set of n edges. Let (i, j) denote a directed edge from vertex i to vertex j. For each vertex $i \in V$, let b_i denote the net flow out of vertex i. If $b_i > 0$ vertex i is a source, if $b_i < 0$ vertex i is a sink and, otherwise, vertex i is a transshipment vertex. For each edge $(i, j) \in E$, let c_{ij}, l_{ij} and u_{ij} denote, respectively, the unit flow cost, lower bound and upper bound on flow in edge (i, j). All data are assumed to be integer. A feasible solution of a network flow problem (often referred to as *flow*) is given by the n-dimensional vector x, where component x_{ij} is the flow in edge (i, j), satisfying flow conservation constraints for all vertices and flow lower bound and capacity constraints on all edges.

The minimum cost network flow (MCNF) problem consists of finding a flow of minimum cost, as expressed in the following classical linear programming formulation:

$$\min \sum_{ij \in E} c_{ij} x_{ij} \tag{1.1}$$

subject to:

$$\sum_{jk \in E} x_{jk} - \sum_{kj \in E} x_{kj} = b_j, \quad j \in V \tag{1.2}$$

$$l_{ij} \leq x_{ij} \leq u_{ij}, \quad (i, j) \in E. \tag{1.3}$$

More compactly, the linear program in (1.1-1.3) can be expressed as

$$\min \{c^\top x \mid Ax = b,\ l \leq x \leq u\},$$

*February 1992 – Revised version 1.2 (March 1992). Cite as M.G.C. Resende and G. Veiga, "An efficient implementation of a network interior point method," *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, eds., DIMACS Series on Discrete Mathematics and Theoretical Computer Science, vol. 12, pp. 299-348, 1993.

†AT&T Bell Laboratories, Murray Hill, NJ 07974 USA

‡Department of IEOR, University of California, Berkeley,CA 94720 USA

where A is the incidence matrix of G. If graph G has p components, there are exactly p redundant flow conservation constraints, which are sometimes removed from the problem formulation. We rule out a trivially infeasible problem by assuming

$$\sum_{j \in V^k} b_j = 0, \quad k = 1, \ldots, p, \tag{1.4}$$

where V^k is the set of vertices for the k-th component of G.

Often it is further required that x_{ij} be integer, i.e.

$$l_{ij} \leq x_{ij} \leq u_{ij}, \ x_{ij} \text{ integer}, \ (i,j) \in E. \tag{1.5}$$

In the remainder of this paper we assume, without loss of generality, that $l_{ij} = 0$ for all $(i,j) \in E$ and that $c \neq 0$.

Variants of the simplex method [8] can be customized to solve the MCNF problem (e.g. [19, 13]) based on two special properties of the graph incidence matrix. Firstly, since a graph incidence matrix is totally unimodular, every primal feasible basis corresponds to an integer flow. Hence, even if integer flow is required, one can relax the integrality constraints in (1.5) and solve the resulting linear program by any simplex variant. Second, in the resulting constraint matrix, there is a one to one correspondence between basic sequences and maximal forests of G, which provides a block triangular ordering for the basic matrix, once redundant constraints are removed. Data structures in implementations of algorithms for solving MCNF problems rely heavily on this property, implying that only integer arithmetic is used and, unlike implementations of the simplex methods for general linear programs, costly factorizations of the basic matrix are unnecessary.

The motivation of this study is that, in practice, the number of iterations taken by interior point algorithms for linear programming appears to grow slowly with problem size. Furthermore, interior point methods do not appear to be affected by degeneracy as much as the simplex method [28]. Most direct comparisons between interior point algorithms and the simplex method (e.g. [2, 21, 23]), conclude that as problem size grows the advantage increasingly tilts toward interior point methods.

To replicate the improved performance observed for the network simplex method over the general simplex method, an interior point implementation dedicated to MCNF problems must use some of the distinguishing properties of the structure of the problem. For example, double precision multiplications are eliminated in operations involving the coefficient matrix and specialized preconditioners based on the network structure can be devised for the conjugate gradient algorithm. Also, the detection of an optimal solution can be based on the integer data.

Several studies compare implementations of interior point algorithms with specialized network codes [3, 4, 24] and conclude that interior point algorithms are not competitive with the specialized network codes. In this paper, we show that a network interior point implementation outperforms specialized network codes on several classes of large MCNF problems. Furthermore, in most problem classes, as the size of the instances grow, so does the difference in solution times.

The dual affine scaling (DAS) algorithm [9] (see also [5, 32]) was among the first interior point methods to be shown to be a competitive alternative to the simplex method [1, 2]. Let A be an $m \times n$ matrix, c, u, and x be n-dimensional vectors and b an m-dimensional vector. The DAS algorithm solves the linear program

$$\min \{c^\top x \mid Ax = b, \ 0 \leq x \leq u\}$$

indirectly by solving its dual

$$\max \{b^\top y - u^\top z \mid A^\top y - z + s = c,\ z \geq 0, s \geq 0\} \tag{1.6}$$

where z and s are an n-dimensional vectors and y is an m-dimensional vector. The algorithm starts with an initial interior solution $\{y^0, z^0, s^0\}$ such that

$$A^\top y^0 - z^0 + s^0 = c,\ z^0 > 0,\ s^0 > 0,$$

and iterates according to

$$\{y^{k+1}, z^{k+1}, s^{k+1}\} = \{y^k, z^k, s^k\} + \alpha\ \{\Delta y, \Delta z, \Delta s\},$$

where the search directions $\Delta y, \Delta z$, and Δs satisfy

$$\begin{aligned} A(Z_k^2 + S_k^2)^{-1} A^\top \Delta y &= b - AZ_k^2(Z_k^2 + S_k^2)^{-1}u, \\ \Delta z &= Z_k^2(Z_k^2 + S_k^2)^{-1}(A^\top \Delta y - S_k^2 u), \\ \Delta s &= \Delta z - A^\top \Delta y, \end{aligned}$$

where

$$Z_k = \text{diag}(z_1^k, \ldots, z_n^k) \text{ and } S_k = \text{diag}(s_1^k, \ldots, s_n^k)$$

and α is such that $z^{k+1} > 0$ and $s^{k+1} > 0$, i.e. $\alpha = \gamma \times \min\{\alpha_z, \alpha_s\}$, where $0 < \gamma < 1$ and

$$\begin{aligned} \alpha_z &= \min\{-z_i^k/(\Delta z)_i \mid (\Delta z)_i < 0,\ i = 1, \ldots, n\} \\ \alpha_s &= \min\{-s_i^k/(\Delta s)_i \mid (\Delta s)_i < 0,\ i = 1, \ldots, n\}. \end{aligned}$$

The dual problem (1.6) has a readily available initial interior point solution:

$$\begin{aligned} y_i^0 &= 0,\ i = 1, \ldots, n \\ s_i^0 &= c_i + \lambda,\ i = 1, \ldots, n \\ z_i^0 &= \lambda,\ i = 1, \ldots, n, \end{aligned}$$

where λ is a scalar such that $\lambda > 0$ and $\lambda > -c_i,\ i = 1, \ldots, n$. In the implementation described in this study (called DLNET), we use $\lambda = 2\ \|c\|_2$.

The bulk of the work in the DAS algorithm is related to building and updating the matrix $AD_kA^\top$ and solving the system of linear equations

$$AD_kA^\top \Delta y = b - AZ_k^2 D_k u, \tag{1.7}$$

where $D_k = (Z_k^2 + S_k^2)^{-1}$. This system determines the ascent direction at each iteration of the algorithm. Whereas for a large class of linear programs system (1.7) can be handled efficiently by direct factorization methods, this is not the case for MCNF problems. In [25] a direct method and an iterative approach based on the conjugate gradient method are compared on randomly generated assignment problems. That study illustrates, for MCNF problems, the gains observed with an iterative approach over a direct factorization method. In a companion paper [26] the relative performance of the interior point approach using a preconditioned conjugate gradient algorithm to the network simplex code NETFLO [19] and the relaxation algorithm code RELAX [6] was shown to improve with the size of the instance. However, the study left unanswered whether an interior point implementation could outperform a network simplex or relaxation method implementation on MCNF problems.

```
procedure pcg(A, D_k, b̄, ε_cg, Δy)
1     Δy_0 := 0;
2     r_0 := b̄;
3     z_0 := M^{-1} r_0;
4     p_0 := z_0;
5     i := 0;
6     do stopping criterion not satisfied →
7           q_i := A D_k A^⊤ p_i;
8           α_i := z_i^⊤ r_i / p_i^⊤ q_i;
9           Δy_{i+1} := Δy_i + α_i p_i;
10          r_{i+1} := r_i − α_i q_i;
11          z_{i+1} := M^{-1} r_{i+1};
12          β_i := z_{i+1}^⊤ r_{i+1} / z_i^⊤ r_i;
13          p_{i+1} := z_{i+1} + β_i p_i;
14          i := i + 1
15    od;
16    Δy := Δy_i
end pcg;
```

FIG. 2.1. *The preconditioned conjugate gradient algorithm*

This paper builds on [26] where we use a conjugate gradient with diagonal and spanning tree preconditioners. Whereas the old implementation was limited to handling uncapacitated bipartite MCNF problems, DLNET can solve capacitated MCNF problems as formulated in (1.1–1.5). In addition, we implement two new stopping criteria and a more stable preconditioned conjugate gradient procedure. The paper is organized as follows. In Section 2 we describe a generic preconditioned conjugate gradient algorithm used in DLNET. The preconditioners applied to the conjugate gradient algorithm defined in Section 2 are described in Section 3. In Section 4 we describe the stopping strategies implemented in DLNET. Computational results on a wide range of MCNF problems are given in Section 5. Concluding remarks are made in Section 6.

2. Computing the Ascent Direction. The computational efficiency of DLNET relies heavily on a preconditioned conjugate gradient algorithm to solve the direction finding system at each iteration. We differ slightly from the preconditioned conjugate gradient algorithm described in [2]. Here, the preconditioned conjugate gradient algorithm is used to solve

$$M^{-1}(AD_kA^\top)\Delta y = M^{-1}\bar{b} \tag{2.1}$$

where M is a positive definite matrix and $\bar{b} = b - AZ_k^2D_k^{-1}u$. The objective is to make the preconditioned matrix

$$M^{-1}(AD_kA^\top)$$

less ill-conditioned than $AD_kA^\top$, improving the convergence of the conjugate gradient algorithm.

The preconditioned conjugate gradient algorithm is presented in the pseudo-code in Figure 2.1. The computationally intensive steps in the preconditioned conjugate gradient algorithm are lines 3, 7 and 11 of the pseudo-code. Those lines correspond to a matrix-vector multiplication (7) and solving systems of linear equations (3 and 11). Line 3 is computed once and lines 7 and 11 are computed once every conjugate

gradient iteration. The matrix-vector multiplications carried out are of the form $AD_kA^\top\ p_i$ are carried out without forming $AD_kA^\top$ explicitly. It is more efficient to carry out the above matrix-vector multiplication by decomposing it into three sparse matrix-vector multiplications. Let

$$\zeta' = A^\top p_i \quad \text{and} \quad \zeta'' = D_k\zeta'.$$

Then

$$(A\ (D_k\ (A^\top p_i))) = A\zeta''.$$

Note that the matrix-vector multiplications are $\mathcal{O}(n)$, involving n additions, $2n$ subtractions and n floating point multiplications. Note further that this computation can be carried out in parallel. In this paper, however, we limit ourselves to serial implementations. See [26] for numerical results of a parallel implementation of the matrix-vector multiplication in the conjugate gradient algorithm.

The preconditioned residual is computed in lines 3 and 11 and amounts to solving the system of linear equations

$$Mz_{i+1} = r_{i+1}, \tag{2.2}$$

where M is a positive definite matrix such that the system can be easily solved. Such preconditioners are the subject of Section 3.

It was pointed out in [2] that the DAS algorithm is particularly well suited to use approximate solutions of the ascent direction linear system. To determine when the direction Δy_i produced by the conjugate gradient algorithm is satisfactory, we use the suggestion made in [16] and compute the angle θ between $(AD_kA^\top)\Delta y_i$ and $\bar{b}$ and stop the conjugate gradient procedure when $|1-\cos\theta| < \epsilon_{cos}$, where ϵ_{cos} is some small tolerance. The computation of

$$\cos\theta = \frac{|\bar{b}^\top(AD_kA^\top)\Delta y_i|}{\|\bar{b}\| \cdot \|(AD_kA^\top)\Delta y_i\|}$$

has the complexity of one conjugate gradient iteration and therefore is not carried out at each conjugate gradient iteration. The cosine is computed every l_{cos} conjugate gradient iterations.

3. Preconditioners. A useful preconditioner for the conjugate gradient algorithm must be such that the system of linear equations (2.2) is easy to solve and at the same time reduces the number of conjugate gradient iterations. A diagonal matrix constitutes the most straightforward and perhaps the most common preconditioner used in conjunction with the conjugate gradient algorithm [12]. They are simple to compute, taking $\mathcal{O}(n)$ double precision multiplications and can be very effective [26, 25, 34]. The diagonal preconditioner used in DLNET is $M = \text{diag}\ (AD_kA^\top)$ and can be computed in $\mathcal{O}(n)$ double precision additions and multiplications. The preconditioned residue systems of lines 3 and 11 of the conjugate gradient pseudo-code can each be solved in $\mathcal{O}(m)$ double precision divisions.

Karmarkar and Ramakrishnan [17] and Vaidya [31] have suggested using a maximum weighted spanning tree preconditioner for network flow problems. Since, in our presentation, the graph G is not necessarily connected, we identify a maximal forest using as weights the diagonal elements of the current scaling matrix,

$$w = D_ke, \tag{3.1}$$

where e is a unit n-vector. The maximal forest is computed by approximately ordering the edges with a bucket sort and applying Kruskal's algorithm [27].

At the k-th iteration of the DAS algorithm, let $\mathcal{S}_k$ be the submatrix of A with columns corresponding to edges in the maximal forest, $t_1, \ldots, t_q$. The preconditioner can be written as

$$M = \mathcal{S}_k \mathcal{D}_k \mathcal{S}_k^\top,$$

where

$$\mathcal{D}_k = \text{diag}(1/z_{t_1}^2 + 1/s_{t_1}^2, \ldots, 1/z_{t_q}^2 + 1/s_{t_q}^2).$$

For simplicity of notation, we include in S_k the linear dependent rows corresponding to the redundant flow conservation constraints. At each conjugate gradient iteration, the preconditioned residue system

$$(\mathcal{S}_k \mathcal{D}_k \mathcal{S}_k^\top) z_{i+1} = r_{i+1} \tag{3.2}$$

is solved with the variables corresponding to redundant constraints set to zero. As with the diagonal preconditioner, (3.2) can be solved in $\mathcal{O}(m)$ time, as the system coefficient matrix can be ordered into a block triangular form.

The spanning tree preconditioner has been previously used in [26, 15, 14].

In practice, the diagonal preconditioner is effective during the initial iterations of the DAS algorithm. As the DAS iterations progress, the spanning tree preconditioner is more effective as it becomes a better approximation of matrix $AD_kA^\top$. In the DLNET implementation, we begin with the diagonal preconditioner and monitor the number of iterations required by the conjugate gradient algorithm. When the conjugate gradient takes more than $\beta\sqrt{m}$ iterations, where $\beta > 0$, DLNET switches to the spanning tree preconditioner. We also set upper and lower limits to the number of DAS iterations using a diagonal preconditioned conjugate gradient.

4. Stopping with an Optimal Flow. The simplex method restricts the sequence of solutions it generates to vertices of the linear programming polytope. Since the constraint matrix is totally unimodular, and assuming integrality of the data, when a simplex variant is applied to a MCNF problem, the optimal solution is integer. On the other hand, the DAS algorithm generates a sequence of dual interior solutions. For general linear programs, a tentative primal solution is computed based on each dual iterate [29]. These sequences converge, respectively, to the relative interiors of the primal and dual optimal faces [30]. Unless the primal optimal solution is unique, the primal solution returned by DAS is not guaranteed to be integer. Furthermore, we wish to use MCNF-specific properties to stop the algorithm earlier than its theoretical convergence. We discuss below the stopping strategies implemented in DLNET.

4.1. Stopping with Basic Solution. As explained in Section 3, computing the spanning tree preconditioner involves identifying a basic sequence for the MCNF problem. Under a dual nondegeneracy assumption, as DAS converges, this basic sequence corresponds to an optimal one. At the end of each iteration of DAS, the maximal forest used to build the preconditioner can be used to compute a tentative primal optimal solution in $\mathcal{O}(m)$ operations. Under dual degeneracy this technique can still be useful if only a small number of degeneracies is present in the optimal dual face. Also, a linear program can be made dual nondegenerate by applying the classical perturbation scheme of [7] to the cost vector. See [26] for details on how this has been implemented in DLNET.

Let $\mathcal{T} = \{t_1, \ldots, t_q\}$ denote the set of edge indices in the maximal forest used to compute the preconditioner. To obtain a tentative primal basic solution, we first set flow of edges not in the forest to either its upper or lower bound. For all $i \in E \setminus \mathcal{T}$:

$$x_i^* = \begin{cases} 0 & \text{if } s_i^k > z_i^k \\ u_i & \text{otherwise,} \end{cases}$$

where s^k and z^k are the current iterates of the dual slack vectors as defined in (1.6). The remaining basic edges have flows that satisfy the linear system

$$A_{\mathcal{T}} x_{\mathcal{T}}^* = b - \sum_{i \in \Omega} u_i A_i, \tag{4.1}$$

where $\Omega = \{i \in E \setminus \mathcal{T} | s_i^k > z_i^k\}$. Linear system (4.1) can be solved in $\mathcal{O}(m)$ time. If $u_{\mathcal{T}} \geq x_{\mathcal{T}}^* \geq 0$ then the primal solution is feasible.

Optimality can be verified producing a dual feasible solution (y^*, s^*, z^*) that is either complementary or that implies a duality gap less than 1. We build the tentative optimal dual solution by first identifying the dual face complementary to x^*, represented by

$$\mathcal{F} = \{i \in \mathcal{T} \mid 0 < x_i^* < u_i\},$$

the set of edges with zero dual slacks. To ensure a complementary primal dual pair, we project orthogonally the current dual interior vector y^k onto the support affine space of the dual face,

$$\min_{y \in \Re^m} \{\|y^* - y^k\| \mid A_{\mathcal{F}}^\top y^* = c_{\mathcal{F}}\}. \tag{4.2}$$

A simliar scheme that uses orthogonal projection to attempt to identify the optimal face has been independently investigated in Kalinski and Ye [15] and Mehrotra and Ye [22]. Ye [33] has analyzed that procedure to prove finite convergence of interior point algorithms for linear programming.

Matrix $A_{\mathcal{F}}^\top$ can be reordered into a block triangular form, with each block corresponding to a component of graph $\tilde{G} = (V, \mathcal{F})$. Since $\tilde{G}$ is a forest, the affine space is the sum of orthogonal one-dimensional subspaces. By computing the orthogonal projections onto each individual subspace independently, the procedure can be completed in $\mathcal{O}(m)$ time.

Assume $\tilde{G}$ has p components, with $T_1, \ldots, T_p$ as the sets of edges in each component tree. After reordering, we have

$$A_{\mathcal{F}} = \begin{bmatrix} A_{T_1} & & \\ & \ddots & \\ & & A_{T_p} \end{bmatrix}.$$

For $i = 1, \ldots, p$, V_i and m_i are, respectively, the set and the number of vertices spanned by edges in T_i, A_{T_i} is an $(m_i + 1) \times m_i$ matrix and each subspace

$$\Psi_i = \{y_{V_i} \in \Re^{m_i+1} \mid A_{T_i}^\top y_{V_i} = c_{T_i}\}$$

has dimension one. Then, for all $y_{V_i} \in \Psi_i$, we have

$$y_{V_i} = y_{V_i}^0 + \alpha_i y_{V_i}^h, \tag{4.3}$$

where $y^0_{V_i}$ is a given solution in Ψ_i and $y^h_{V_i}$ is a solution of the homogeneous system $A^\top_{T_i} y_{V_i} = 0$. Since A_{T_i} is the incidence matrix of a tree, the unit vector is a homogeneous solution. The given solution $y^0_{V_i}$ can be computed by selecting $v \in V_i$, setting $y^0_v = 0$ and solving the triangular system resulting from removing from matrix A_{T_i} the row corresponding to vertex v,

$$\tilde{A}^\top_{T_i} y_{V_i \setminus \{v\}} = c_{T_i},$$

where $\tilde{A}_{T_i}$ is the triangular matrix. With the representation in (4.3), the orthogonal projection of y_{V_i} onto subspace Ψ_i is

$$y^*_{V_i} = y^0_{V_i} + \frac{e^\top_{V_i}(y_{V_i} - y^0_{V_i})}{m_i} e_{V_i}$$

where e is the unit vector.

The orthogonal projection as indicated in (4.2) is obtained by combining the projections onto each subspace,

$$y^* = (y^*_{V_1}, \ldots, y^*_{V_q}).$$

We build a feasible dual solution by computing the slacks as

$$z^*_i = \begin{cases} -\delta_i & \text{if } \delta_i < 0 \\ 0 & \text{otherwise} \end{cases} \qquad s^*_i = \begin{cases} 0 & \text{if } \delta_i < 0 \\ \delta_i & \text{otherwise,} \end{cases}$$

where $\delta_i = c_i - A^\top y^*$.

The primal and dual solutions, x^* and (y^*, s^*, z^*), are optimal if complementary slackness is satisfied, i.e. if for all $i \in E \setminus \mathcal{T}$ either $s^*_i > 0$ and $x^*_i = 0$ or $z^*_i > 0$ and $x^*_i = u_i$. Otherwise, the primal solution, x^*, is still optimal if the duality gap is less than 1, i.e. if $c^\top x^* - b^\top y^* + u^\top z^* < 1$.

4.2. Stopping with Maximum Flow Solution. Determining the magnitude in the classical perturbation technique poses a major obstacle. The theoretical acceptable perturbation can be too small to resolve all dual degeneracies in a reasonable number of DAS iterations. A larger perturbation, on the other hand, can change the combinatorial structure of the optimal dual face. An alternative stopping procedure consists of identifying the optimal dual face with the dual interior solution and computing an optimal primal solution by solving a maximum flow problem on a restricted network. Compared to solving spanning tree based linear systems, the maximum flow problem displays a high theoretical complexity. However, the low practical complexity of new maximum flow algorithms make this procedure an attractive option. Furthermore, the stopping test is not performed at every iteration of the DAS algorithm.

As described in the previous section, the dual iterates generated by the DAS algorithm converge to the relative interior of the optimal dual face [30]. In practice, the algorithm identifies the set of active dual constraints defining a tentative optimal dual face

$$\mathcal{F} = \{i \in E \mid s^k_i - z^k_i < \epsilon \text{ or } \gamma \leq s^k_i / z^k_i \leq 1/\gamma\},$$

where $\epsilon > 0$ and $\gamma > 0$ are small tolerances. Unless the DAS algorithm is close to convergence, there is no guarantee that $\mathcal{F}$ actually defines a dual face, as the set $\{y \in \Re^m \mid A^\top_{\mathcal{F}} y = c_{\mathcal{F}}\}$ can be empty.

Let $\tilde{G} = (V, \mathcal{F})$ be the original graph G with the $\mathcal{F}$ as its set of edges. A maximal forest $\mathcal{T}$ of $\tilde{G}$ defines a dual face. If $\mathcal{F}$ also defines a dual face, edges in $\mathcal{F} \setminus \mathcal{T}$

correspond to redundant hyperplanes, and the face is unique. When this is not the case, we select the maximal forest according the ordering used for the preconditioner computation. The tentative dual optimal solution is computed by projecting the current dual interior vector y^k onto the support affine space of the dual face defined by $\mathcal{T}$,

$$\min_{y^* \in \Re^m} \{\|y^* - y^k\| \mid A_{\mathcal{T}}^\top y^* = c_{\mathcal{T}}\}.$$

This operation is identical to the orthogonal projection described earlier in this section. The dual slack for the tentative dual optimal solution are computed as

$$z_i^* = \begin{cases} -\delta_i & \text{if } \delta_i < 0 \\ 0 & \text{otherwise} \end{cases} \qquad s_i^* = \begin{cases} 0 & \text{if } \delta_i < 0 \\ \delta_i & \text{otherwise,} \end{cases}$$

where $\delta_i = c_i - A^\top y^*$.

Based on the projected dual solution y^*, the tentative optimal face is redefined as

$$\tilde{\mathcal{F}} = \{i \in E \mid |c_i - A_{\cdot i}^\top y^*| < \epsilon\}.$$

A primal solution complementary to the tentative optimal solution has the non-active edges set to lower or upper bounds. For all $i \in E \setminus \tilde{\mathcal{F}}$,

$$x_i^* = \begin{cases} 0 & \text{if } c_i - A_{\cdot i}^\top y^* > 0 \\ u_i & \text{otherwise.} \end{cases}$$

Flow in the active edges satisfies

$$A_{\tilde{\mathcal{F}}} x_{\tilde{\mathcal{F}}}^* = \tilde{b} = b - \sum_{i \in \Omega} u_i A_i, \tag{4.4}$$

$$0 \le x_i^* \le u_i, \quad i \in E \setminus \tilde{\mathcal{F}}, \tag{4.5}$$

where $\Omega = \{i \in E \setminus \tilde{\mathcal{F}} \mid c_i - A_{\cdot i}^\top y^* < 0\}$.

Because of the nature of most maximum flow algorithms (they generate integer flows), an integer solution to the system above can be obtained by solving a maximum flow problem on the restricted network $\tilde{G} = (\tilde{V}, \tilde{E})$, where

$$\tilde{V} = \{\sigma, \theta, V\}$$

and

$$\tilde{E} = \{\Sigma, \Theta, \tilde{\mathcal{F}}\}.$$

The additional edges are such that

$$\Sigma = \{(\sigma, i) \mid i \in V,\ \tilde{b}_i > 0\}$$

with capacity $\tilde{b}_i$, and

$$\Theta = \{(i, \theta) \mid i \in V,\ \tilde{b}_i < 0\}$$

with capacity $-\tilde{b}_i$.

Let $\mathcal{M}_{\sigma,\theta}$ be the maximum flow from σ to θ in $\tilde{G}$, with x_i^* as the flow on each edge $i \in \tilde{\mathcal{F}}$. If

$$\mathcal{M}_{\sigma,\theta} = \sum_{\{i \in V \mid \tilde{b}_i > 0\}} \tilde{b}_i,$$

then $x_{\tilde{\mathcal{F}}}^*$ is a feasible flow for the restricted network problem (4.4-4.5). Furthermore, x^* and (y^*, s^*, z^*) are optimal primal-dual complementary solutions for the original problem.

5. Computational Investigation. The aim of this investigation is to compare network implementations of interior point, simplex, and relaxation algorithms on a wide range of MCNF problems, including minimum cost circulation, maximum flow, and transshipment. We are motivated by empirical evidence indicating that for general linear programming the relative speedup of interior point algorithms to the simplex method increases with problem size. We seek to establish whether a similar phenomenon occurs with network flow problems, for which specialized network simplex codes have been shown to outperform general-purpose simplex codes by several orders of magnitude. We compare our interior point code with the network simplex code NETFLO [19]. We also compare our code with RELAXT-3 [6] an implementation of the relaxation algorithm, reported to outperform NETFLO on several classes of MCNF problems.

In [26] we demonstrated that if the conjugate gradient algorithm is implemented in parallel on an eight processor Alliant FX/80, an interior point code could solve large assignment problems faster than serial versions of the network simplex code NETFLO as well the relaxation method code RELAX (an early version of RELAXT-3). In this investigation, we wish to determine if a serial interior point implementation can accomplish the same. For this investigation we have developed DLNET, a new network implementation of the DAS algorithm that was briefly described in the first sections of this paper.

Finally, we wish to demonstrate the feasibility of using an interior point code to solve MCNF problems having hundreds of thousands of vertices in reasonable running time.

All instances used in the computational investigation are generated with problem generators distributed for The First DIMACS International Algorithm Implementation Challenge [10]. The problems are grouped into five categories: minimum cost circulation, transshipment on a mesh, minimum cost-maximum flow on a grid, maximum flow on a random layered graph, and problems generated with the standard NETGEN [20] generator. All instances and/or generators can be obtained via anonymous `ftp` from `dimacs.rutgers.edu`.

5.1. Computing Environment. The computational experiment is conducted on a Silicon Graphics IRIS workstation, model 4D/240, with four 25 MHz IP7 processors, a MIPS R2010A/R3010 FPU, a MIPS R2000A/R3000 CPU, 64 Kbytes instruction cache, 64 Kbytes data cache, and 256 Mbytes of main memory. The swapping device is configured to allow processes over 1 Gbytes in size to run. The operating system is IRIX System V Release 3.3.2.

DLNET is written mostly in FORTRAN, with only memory management and input/output routines written in C, `yacc` and `lex`. The experiments were done with version 1.4b (30 Jan 92) of DLNET. Version 1.4b of the code contains 5337 lines of FORTRAN (3434 of which are comments), 5623 lines of C, 515 lines of `yacc` and 116 lines of `lex`. NETFLO and RELAXT-3 are FORTRAN codes. NETFLO has 1072 lines of code (290 of which are comments) while RELAXT-3 has 2559 lines (653 of which are comment lines). We modified NETFLO and RELAXT-3 to avoid integer overflow when computing the optimal objective function value. These computations were originally done in integer arithmetic and are carried out here in double precision floating point arithmetic. They are computed once, upon termination of the algorithm. The compilers `f77` and `cc` are used with optimization level `-O2 -Olimit 800`. Running times are measured with the UNIX routine `times()`. The times reported exclude time to input the problem.

5.2. DLNET **Parameter Settings.** DLNET obtains from a specification file run-time parameters that control the execution of the algorithm. Fine tuning of these parameters for each individual problem could lead to faster execution times. However, for the experiments reported here, we run DLNET on all instances with identical parameter settings listed in Figure 5.1. The following is a description of DLNET parameters:

- `mode`: optimization mode (minimization or maximization)
- `seed`: random number generator seeds
- `maximum iterations`: maximum DAS iterations
- `maximum perturbation`: perturbation parameter ρ_p. Cost vector is randomly perturbed
$$c_p = c + \text{unif}(-\epsilon_p, \epsilon_p)$$
where $\epsilon_p = \rho_p/(2\ \|c\|_2)$.
- `warm iterations`: minimum number of DAS iterations with diagonal preconditioned CG
- `maximum switch iterations`: maximum number of DAS iterations with diagonal preconditioned CG
- `active tolerance`: tolerance used to set "non-basic" variables
- `zero tolerance`: tolerance for 0
- `primal basic optimality`: set primal estimates testing; initial DAS iteration to begin testing; DAS iteration frequency of testing; testing tolerance
- `absolute gap optimality`: set absolute gap optimality testing; initial DAS iteration to begin testing; DAS iteration frequency of testing; testing tolerance
- `dual interior optimality`: set dual interior optimality testing; initial DAS iteration to begin testing; DAS iteration frequency of testing; testing tolerance
- `max flow optimality`: set max flow optimality testing; initial DAS iteration to begin testing; DAS iteration frequency of testing; testing tolerance
- `sort buckets`: number of buckets in sort $\div\ n$
- `cg tolerance`: $\epsilon_{\cos}$ tolerance for CG convergence
- `cg maximum iterations`: maximum number of CG iterations $\div\ \sqrt{m}$
- `cg maximum diagonal`: maximum number of diagonal preconditioner CG iterations $\div\ \sqrt{m}$
- `cg residual check`: l_{cos} frequency of CG stopping criterion checking

Finally, the DAS step back factor γ is hard-wired in the code and cannot be set through the specification file. For the first 10 iterations $\gamma = 0.99$. After that, $\gamma = 0.95$.

To solve maximum flow problems on the restricted network described in Section 4 we have interfaced the implementation of Dinic's algorithm of Goldfarb and Grigoriadis [11] with DLNET.

5.3. Experimental Results. In this sub-section we present experimental results on the five problem classes listed earlier. Several of the classes are further broken down into sub-classes. For each sub-class we summarize the experiments with two tables and a figure. The first table summarizes running times and iteration counts for the three codes. Values listed are averaged over one or more runs, corresponding to different instances generated with different seeds. In most cases, three instances are generated, though this is not always the case, given the expensive nature of running the codes on some of the instances. The second table summarizes DLNET runs. Specifically, it lists averages for conjugate gradient iterations and running times, running times for spanning tree approximate sorting and Kruskal's algorithm, and number of calls to the maximum flow optimality testing procedure, the average running time of

```
begin
       mode                          minimize
       seed                          999 333
       maximum iterations            300
       maximum perturbation          1.e-3
       warm iterations               2
       maximum switch iteration      15
       active tolerance              1.e-3 1.e-4
       zero tolerance                1.e-20
       primal basic optimality       yes 10 1 1.e-8
       absolute gap optimality       yes 10 1 1.
       dual interior optimality      yes 10 1 1.e-15
       max flow optimality           yes 10 10 1.e-8
       sort buckets                  1
       cg tolerance                  1.e-3
       cg maximum iterations         1
       cg maximum diagonal           0.5
       cg residual check             5
end
```

FIG. 5.1. DLNET *specification file*

each call, the size of the restricted graph ($|\tilde{E}|$) and a measure of the density of the restricted graph ($|\tilde{E}|/(|V|-1)$). The figure shows running time ratios, in log-log scale, for NETFLO to DLNET, and RELAXT-3 to DLNET.

In all runs carried out in this experiment the optimal objective function values found by DLNET (primal value), NETFLO and RELAXT-3 were identical. In most cases, the dual objective function value obtained by DLNET was equal to the primal value. However, because we allowed DLNET to stop with a primal dual gap of less than one, but not necessarily zero, in a few cases, the dual objective value did not equal the primal. By changing the specification file to disallow stopping with absolute duality gap, all DLNET solutions could be made primal-dual complementary. Table 5.1 summarizes how DLNET stopped for all problem instances. For each problem class, the table lists the number of instances solved, the number of instances in which DLNET stopped with the primal estimate stopping criterion, the corresponding percentage of all instances for which this occurred, the number of instances in which DLNET stopped with the maximum flow stopping criterion, the corresponding percentage, the number of instances that DLNET produced a primal-dual complementary solution and the percentage of instances for which this occurred.

5.3.1. Transshipment Problems on a Grid. The networks in this class of problems are obtained by removing the minimum number of edges from a grid graph embedded on a torus such that all vertical paths wrap around and no horizontal path wraps around, and adding a source and sink vertex with edges going from the source to all vertices on one side of the resulting tube network and from all vertices on the other side to the sink. Network density is controlled by adding extra edges in both the horizontal and vertical directions. All vertices other than the source and sink vertices are transshipment vertices. Costs and capacities are uniformly distributed in the intervals $[0, 4096]$ and $[0, 16384]$, respectively.

These instances are generated with the MCNF problem generator `goto.c` of Goldberg [10]. Three sub-classes of problems are generated: Grid-Density-8, Grid-Density-

TABLE 5.1
Summary of DLNET *optimal solutions*

Problem Class	Instances	PB-Stopping		MF-Stopping		PD-Opt Sol'n	
		#	%	#	%	#	%
Grid-Density-8	22	19	86	3	14	22	100
Grid-Density-16	18	5	28	13	72	18	100
Grid-Increasing-Density	18	3	17	15	83	18	100
RLG-Wide	7	0	0	7	100	7	100
Grid-Square	14	12	86	2	14	14	100
Grid-Wide	27	27	100	0	0	25	93
Grid-Long	24	23	96	1	4	24	100
Mesh_1	16	14	88	2	12	16	100
Mesh_2	15	11	73	4	27	15	100
Mesh_4	15	10	67	5	33	15	100
Mesh_8	12	8	67	4	33	12	100
Netgen-Lo	27	19	70	8	30	22	81
Netgen-Hi	27	23	85	4	15	25	93
All instances	242	174	72	68	28	233	96

16 and Grid-Increasing-Density. Let m and n denote the number of vertices and edges of the network, respectively. For Grid-Density-8, $n = 8\,m$; for Grid-Density-16, $n = 16\,m$; and for Grid-Increasing-Density, $n = m^{1.5}$. For Grid-Density-8 and Grid-Density-16, instances having $256, 512, \ldots, 32768$ vertices are generated. For Grid-Increasing-Density the networks have $256, 512, \ldots, 8192$ vertices.

The random number generator seeds 270001, 270002, and 270003 were used to generate different instances for each problem size. For Grid-Density-8, 3 instances of sizes $256, 512, \ldots, 16384$ vertices were generated. A single 32768 vertex instance was generated. RELAXT-3 was not run on that instance. For Grid-Density-16, 3 instances of sizes $256, 512, \ldots, 8192$ vertices were generated. Single instances of sizes 16384 and 32768 vertices were generated. RELAXT-3 was not run on the 32768 vertex instance. For Grid-Increasing-Density, 3 instances were generated for each problem size. RELAXT-3 was run on only a single 8192 vertex instance.

Tables 5.2-5.3 and Figure 5.2 summarize runs for problem class Grid-Density-8. Tables 5.4-5.5 and Figure 5.3 summarize runs for problem class Grid-Density-16. Tables 5.6-5.7 and Figure 5.4 summarize runs for problem class Grid-Increasing-Density.

We make the following observations regarding this family of problems.

- DLNET was faster than RELAXT-3 on all instances in sub-classes Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density. A speedup ratio of over 85 was observed, where RELAXT-3 took 60.4 hours while DLNET took 42.4 minutes.
- DLNET was faster than NETFLO on the larger instances. For those, a speedup ratio of over 15 was observed, where NETFLO took 81.33 hours and DLNET took 5.2 hours.
- DLNET-to-RELAXT-3 and DLNET-to-NETFLO solution time ratios decrease with network density.
- Let T_D, T_R, T_N denote running time (in CPU seconds) of DLNET, RELAXT-3 and NETFLO, respectively. A linear regression of the running times produced

TABLE 5.2
CPU times for problem class Grid-Density-8

PROB	SIZE		DLNET		NETFLO		RELAXT-3	
	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	256	2048	5.6	22.7	1.6	5613.7	18.8	68620.0
n_512	512	4096	12.5	23.7	7.0	15472.0	106.1	214289.7
n_1024	1024	8192	39.7	29.0	26.8	29750.0	432.5	600647.3
n_2048	2048	16384	102.4	32.7	112.8	68479.7	1625.5	1248580.7
n_4096	4096	32768	296.8	36.0	721.3	360894.0	6695.2	1890436.7
n_8192	8192	65536	842.7	35.3	2740.7	330211.0	43694.8	9761998.3
n_16384	16384	131072	2545.6	40.0	12116.9	755820.3	217432.2	29034906.7
n_32768	32768	262144	18882.0	90.0	293822.2	18718769.0	DID NOT RUN	

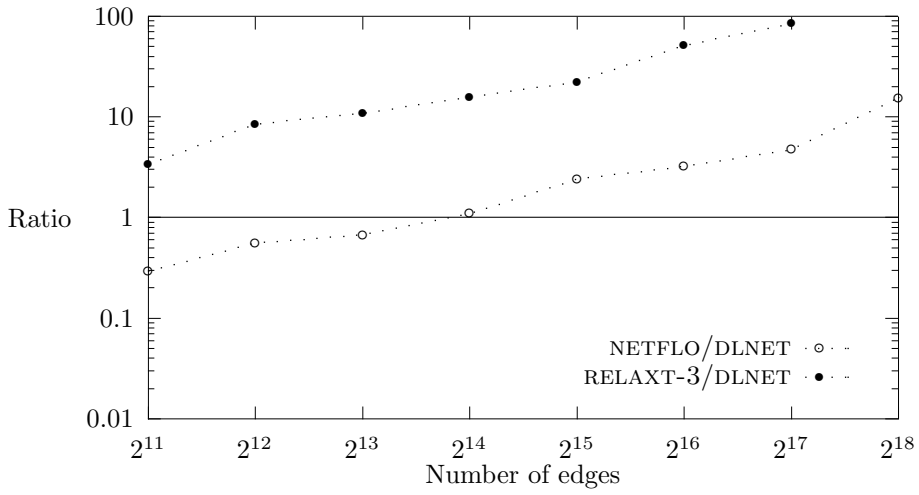

FIG. 5.2. *CPU time ratios for problem class Grid-Density-8*

TABLE 5.3
DLNET statistics for problem class Grid-Density-8

PROB	Conj. Grad.		Span. Tree		Max Flow			
	itr	time	sort	Kruskal	calls	time	$\|\tilde{E}\|$	$\frac{\|\tilde{E}\|}{\|V\|-1}$
n_256	6.6	0.07	0.03	0.01	2.0	0.06	257.3	1.01
n_512	6.2	0.15	0.06	0.01	2.0	0.15	514.3	1.01
n_1024	7.0	0.38	0.13	0.03	2.3	0.42	1029.9	1.01
n_2048	7.4	0.99	0.28	0.08	3.0	1.37	2063.3	1.01
n_4096	8.5	2.95	0.59	0.06	3.3	5.09	4147.7	1.01
n_8192	10.8	9.85	1.24	0.42	3.0	43.92	8210.2	1.00
n_16384	12.2	27.55	2.64	0.97	3.3	155.31	16424.3	1.00
n_32768	19.6	133.87	5.83	1.14	9.0	266.45	33131.6	1.01

TABLE 5.4
CPU times for problem class Grid-Density-16

PROB	SIZE		DLNET		NETFLO		RELAXT-3	
	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	256	4096	15.9	31.7	4.3	14224.7	85.3	137897.7
n_512	512	8192	56.8	39.7	15.7	32477.7	355.5	335681.7
n_1024	1024	16384	134.6	48.0	64.9	74794.3	1655.0	1092594.3
n_2048	2048	32768	394.2	60.0	275.0	172897.7	5905.8	2614008.3
n_4096	4096	65536	921.6	63.3	1307.2	453328.0	27352.2	6324750.3
n_8192	8192	131072	2878.9	70.0	5082.8	839747.0	110404.4	13928929.0
n_16384	16384	262144	7099.3	80.0	26550.8	1948729.0	536569.2	46306590.0
n_32768	32768	524288	26572.8	90.0	281157.8	15453366.0	DID NOT RUN	

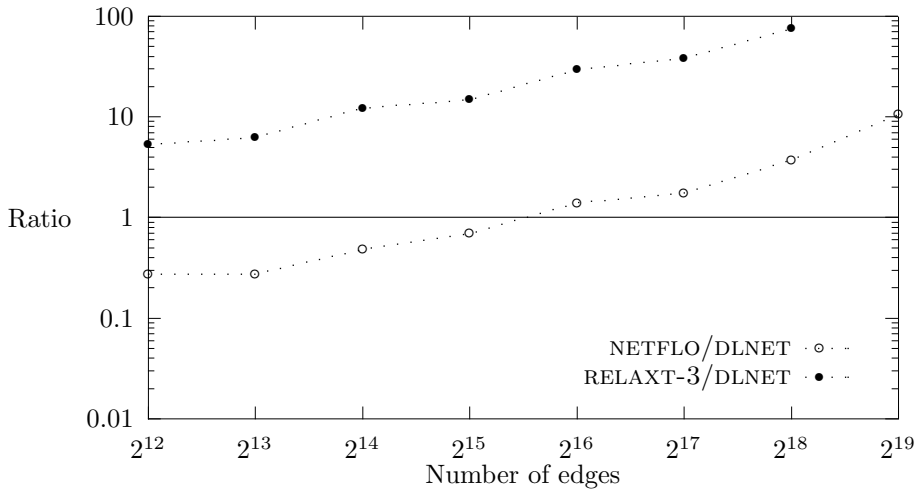

FIG. 5.3. *CPU time ratios for problem class Grid-Density-16*

TABLE 5.5
DLNET statistics for problem class Grid-Density-16

PROB	Conj. Grad.		Span. Tree		Max Flow			
	itr	time	sort	Kruskal	calls	time	$\|\tilde{E}\|$	$\frac{\|E\|}{\|V\|-1}$
n_256	9.0	0.19	0.06	0.01	3.0	0.07	271.6	1.06
n_512	14.1	0.62	0.13	0.01	3.7	0.16	544.6	1.06
n_1024	10.0	1.09	0.27	0.06	4.7	0.45	1085.2	1.06
n_2048	10.8	2.48	0.55	0.14	6.0	1.37	2243.4	1.10
n_4096	10.8	6.13	1.19	0.08	6.3	4.03	4445.6	1.08
n_8192	13.6	20.59	2.60	0.89	7.0	19.81	8762.4	1.07
n_16384	12.3	41.48	5.35	1.69	8.0	80.39	17036.8	1.04
n_32768	17.2	171.12	12.30	1.01	9.0	302.67	34520.9	1.05

TABLE 5.6
CPU times for problem class Grid-Increasing-Density

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	256	4096	17.0	31.7	6.3	14224.7	99.4	137897.7
n_512	512	11585	79.4	50.0	38.5	47668.0	716.2	444212.0
n_1024	1024	32768	513.9	70.0	297.2	188971.3	4411.3	1480909.3
n_2048	2048	92682	2214.6	103.3	2191.1	679631.7	23886.3	3826721.3
n_4096	4096	262144	9277.8	150.0	18383.2	2516757.3	132558.0	14827709.0
n_8192	8192	741455	44623.1	200.0	121526.8	7467246.7	1138682.0	30883918.0

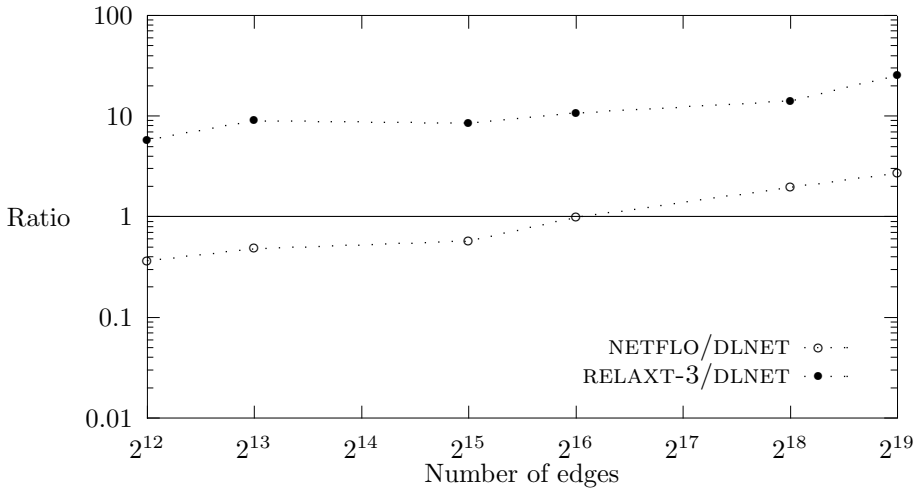

FIG. 5.4. *CPU time ratios for problem class Grid-Density-Increasing*

the following models for:

- Grid-Density-8:

$$\log_{10} T_D = .46 \log_2 |E| - 4.449 \ \ (R^2 = .9903)$$
$$\log_{10} T_R = .66 \log_2 |E| - 6.041 \ \ (R^2 = .9954)$$
$$\log_{10} T_N = .68 \log_2 |E| - 7.354 \ \ R^2 = .9890),$$

- Grid-Density-16:

$$\log_{10} T_D = .44 \log_2 |E| - 4.072 \ \ (R^2 = .9967)$$
$$\log_{10} T_R = .63 \log_2 |E| - 5.615 \ \ (R^2 = .9975)$$
$$\log_{10} T_N = .66 \log_2 |E| - 7.331 \ \ (R^2 = .9944),$$

- Grid-Increasing-Density:

$$\log_{10} T_D = .46 \log_2 |E| - 4.213 \ \ (R^2 = .9976)$$
$$\log_{10} T_R = .52 \log_2 |E| - 4.272 \ \ (R^2 = .9964)$$
$$\log_{10} T_N = .58 \log_2 |E| - 6.179 \ \ (R^2 = .9996),$$

TABLE 5.7
DLNET statistics for problem class Grid-Density-Increasing

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	9.0	0.20	0.06	0.01	3.0	0.06	271.6	1.06
n_512	10.1	0.70	0.19	0.01	5.0	0.17	569.3	1.11
n_1024	15.3	3.30	0.60	0.07	7.0	0.75	1203.3	1.18
n_2048	20.2	13.35	1.87	0.33	10.3	2.46	2643.2	1.29
n_4096	12.9	27.88	5.58	0.13	15.0	6.74	4735.1	1.16
n_8192	22.2	150.68	18.36	3.97	20.0	20.65	12561.4	1.53

where R^2 is the coefficient of multiple determination.

- The conjugate gradient algorithm took on average 19.6, 17.2 and 22.2 iterations for the largest instances of classes Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density, respectively. Those instances have, respectively 32768, 32768, and 8192 vertices.
- For the largest instances of classes Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density the computation of the maximum weight spanning tree (sorting and Kruskal's algorithm) took respectively 0.5%, 0.7% and 13% of the total time taken by the conjugate gradient algorithm (computing the spanning tree plus carrying out the conjugate gradient iterations). Sorting accounted for 88%, 92% and 84% of the total spanning tree computation time for classes Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density, respectively.
- On the largest instances of Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density, DLNET spent on solving maximum flow problems in the maximum flow stopping test the equivalent of respectively 18.9%, 16.4% and 1.2% of the time spent on the conjugate gradient algorithm.
- The density of the restricted network of the maximum flow stopping criterion increased with the density of the original network. The largest restricted network densities for Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density had $|\tilde{E}|/(|V|-1)$ values of 1.01, 1.10 and 1.53, respectively.
- For Grid-Density-8, DLNET stopped with primal estimate stopping in 86% of the instances and with maximum flow stopping in the remaining 14%. For Grid-Density-16, DLNET stopped with primal estimate stopping in 28% of the instances and with maximum flow stopping in the remaining 72%. For Grid-Increasing-Density, DLNET stopped with primal estimate stopping in 17% of the instances and with maximum flow stopping in the remaining 83%. Primal-dual complementary solutions were produced for all instances.
- The instances had optimal objective function values with 10 to 12 digits.

5.3.2. Maximum Flow Problems. The class RLG-Wide consists of finding the maximum flow across a wide random leveled network. In these networks, vertices are arranged on a grid in 2^{x-6} rows by 64 columns ($x = 11, 12, \ldots$), with an additional source vertex and sink vertex. Edges go from the source vertex to each vertex in the first column of the grid and from each vertex in the last column to the sink. Furthermore, each vertex in the first 63 columns has an edge going to exactly three randomly selected vertices in the next column. Edge capacities in the grid are uniformly distributed in $[1, 10000]$. Edges out of the source and into the sink edges are uncapacitated.

TABLE 5.8
CPU times for problem class RLG-Wide

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
x_32	2050	6113	40.9	20.0	37.5	6848.0	67.3	2283.0
x_64	4098	12225	125.9	20.0	181.7	17640.0	287.9	4460.0
x_128	8194	24449	493.0	30.0	917.0	45469.0	1582.8	9824.0
x_256	16386	48897	1328.7	30.0	4806.0	120058.0	7891.2	19472.0
x_512	32770	97793	3854.5	30.0	25584.1	325919.0	52689.2	46386.0
x_1024	65538	195585	14496.8	50.0	141157.0	849836.0	252447.2	95147.0
x_2048	131074	391169	51293.8	70.0	962987.5	2408010.0	1460680.0	189577.0

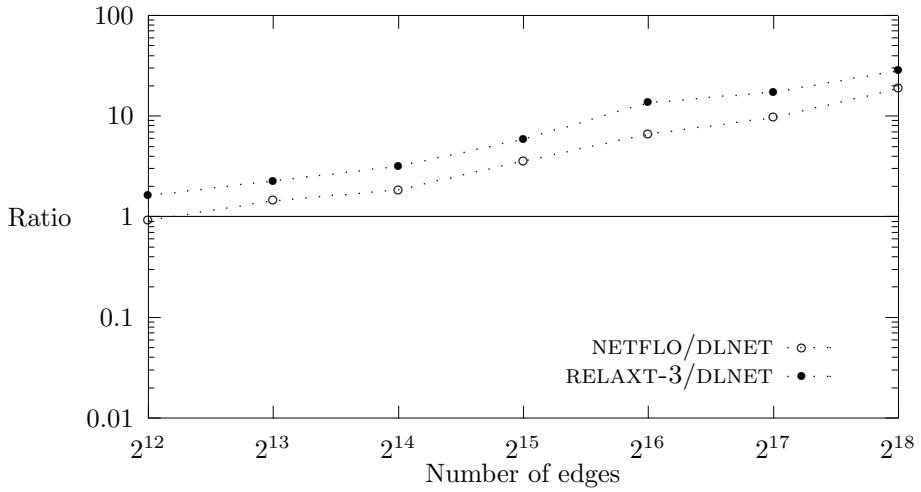

FIG. 5.5. *CPU time ratios for problem class RLG-Wide*

We transform the maximum flow problem into a MCNF problem by adding an uncapacitated edge from the sink to the source with cost -1. All other edges have cost zero. These instances are used as a test for the codes on highly dual degenerate problems. In practice, however, one should use specialized maximum flow codes to solve maximum flow problems.

The instances are generated with the generator `washington.c` of Anderson [10]. Seven instances on $2^{x-6} \times 64$ grids ($x = 11, 12, \ldots, 17$) are generated. All three codes solved all instances. Tables 5.8-5.9 and Figure 5.5 summarize runs for problem class RLG-Wide.

We make the following observations regarding this family of problems.

- Except for the smallest instance (with $x = 11$) DLNET was faster than the other codes on all instances. On the largest instance (131074 vertices and 391169 edges) DLNET was over 18 times faster than NETFLO and 28 times faster than RELAXT-3. On that instance DLNET took 14.2 hours to solve the problem while NETFLO and RELAXT-3 took 267.5 hours and 405.7 hours,

TABLE 5.9
DLNET statistics for problem class RLG-Wide

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
x_32	20.0	1.26	0.10	0.04	2.0	1.81	5869.0	2.86
x_64	25.8	4.56	0.20	0.09	2.0	5.10	11684.0	2.85
x_128	27.7	11.45	0.41	0.22	3.0	24.09	23076.0	2.82
x_256	29.3	31.07	0.87	0.54	3.0	75.74	46680.3	2.85
x_512	39.2	95.16	1.86	1.29	3.0	213.00	93035.0	2.84
x_1024	27.6	151.17	4.11	3.31	5.0	1125.56	184545.2	2.82
x_2048	31.6	385.24	8.64	7.46	7.0	2917.88	366764.4	2.80

respectively.

- Let T_D, T_R, T_N denote running time (in CPU seconds) of DLNET, RELAXT-3 and NETFLO, respectively. A linear regression of the running times produced the following models:

$$\log_{10} T_D = .51 \log_2 |E| - 4.821 \quad (R^2 = .9986)$$
$$\log_{10} T_R = .73 \log_2 |E| - 7.407 \quad (R^2 = .9992)$$
$$\log_{10} T_N = .73 \log_2 |E| - 7.663 \quad (R^2 = .9992),$$

  where R^2 is the coefficient of multiple determination.
- On the largest instance the conjugate gradient algorithm took on average 31.6 iterations.
- On the largest instance the computation of the spanning tree (sorting plus Kruskal's algorithm) accounted for 0.4% of the total time taken by the conjugate gradient (computing the preconditioner plus conjugate gradient iterates). Sorting accounted for 54% of the total spanning tree computation time.
- On the largest instance of RLG-Wide, DLNET spent on solving maximum flow problems in the maximum flow stopping test the equivalent of 72.7% of the time spent on the conjugate gradient algorithm.
- Since these instances are highly dual degenerate it is expected that the restricted networks of the maximum flow stopping criterion will be dense. In fact, the least dense restricted network had on average 2.8 ($|V| - 1$) edges.
- In all instances, DLNET stopped using the maximum flow stopping criterion. All solutions produced by DLNET were primal-dual complementary.
- The instances had optimal objective function values with 6 to 8 digits.

5.3.3. Minimum Cost-Maximum Flow Problems on a Grid. The networks in this class are formed on a grid of vertices of height h and width w. Two additional vertices complete the vertex set: a source vertex S and a sink vertex T. Edges go from the source to each vertex in the first column of the grid and from each vertex in the last column of the grid to the sink vertex. On the grid, edges go from vertex to nearest neighbor vertex oriented left to right and top to bottom. Grid edge costs and capacities are generated uniformly in the interval $[1, 10000]$. Edges from the source and into the sink have cost zero and are uncapacitated. All vertices, except the source and the sink are transshipment vertices. The source has a supply of M_{ST}, the maximum flow from S to T, and the sink has a demand of M_{ST}.

These instances are generated with the MCNF problem generator `ggraph1.f` of Resende [10]. Three sub-classes of problems are generated: Grid-Square, Grid-Wide,

TABLE 5.10
CPU times for problem class Grid-Square

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	258	512	1.6	15.0	0.1	355.3	0.3	1354.3
n_1024	1026	2048	12.7	25.0	0.7	1940.3	4.2	7417.3
n_4096	4098	8192	149.2	38.3	8.6	10393.7	103.8	18453.0
n_16384	16386	32768	2192.8	61.3	159.2	72847.3	2295.4	392862.0
n_65536	65538	131072	17618.4	90.0	2625.9	486330.0	36947.8	2715011.0
n_262144	262146	524288	255332.0	180.0	67189.7	3482255.0	DID NOT RUN	

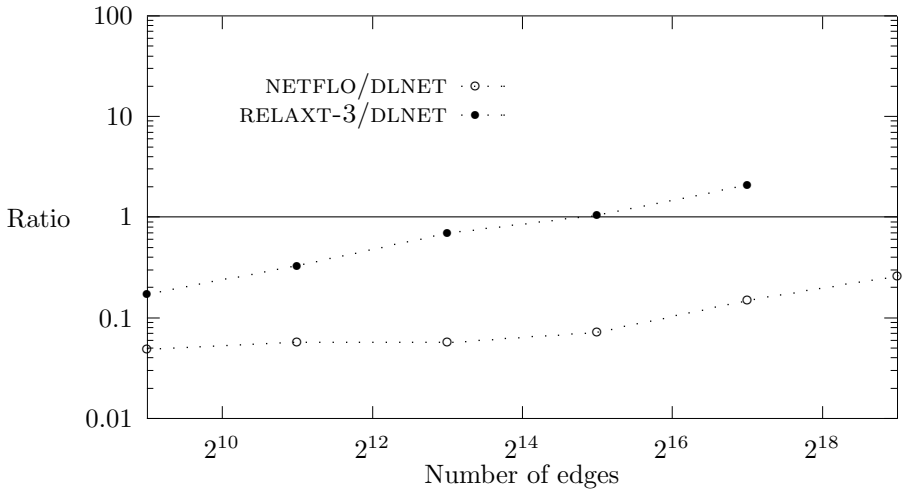

FIG. 5.6. *CPU time ratios for problem class Grid-Square*

and Grid-Long. Grid-Square has $h = w = 16, 32, \ldots$ Grid-Wide has $w = 16$ and $h = 32, 64, \ldots$ Grid-Long has $h = 16$ and $w = 32, 64, \ldots$ Karmarkar and Ramakrishnan, in [18], solved instances similar to those of Grid-Square.

The random number generator seeds 270001, 270002, and 270003 were used to generate different instances for each problem size. For Grid-Square, 3 instances of sizes $258, 1026, \ldots, 16386$ vertices were generated. Single 65538 and 262146 vertex instances were generated. RELAXT-3 was not run on the largest instance. For Grid-Wide, 3 instances of sizes $514, 1026, \ldots, 131074$ vertices were generated. RELAXT-3 was not run on one of the three 65538 vertex instances and on all three of the 131074 vertex instances. For Grid-Long, 3 instances of sizes $258, 1026, \ldots, 65538$ vertices were generated.

Tables 5.10-5.11 and Figure 5.6 summarize runs for problem class Grid-Square. Tables 5.12-5.13 and Figure 5.7 summarize runs for problem class Grid-Long. Tables 5.14-5.15 and Figure 5.8 summarize runs for problem class Grid-Wide.

We make the following observations regarding this family of problems.

- On both Grid-Square and Grid-Wide, the interior point code's performance

TABLE 5.11
DLNET statistics for problem class Grid-Square

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	9.9	0.04	0.01	0.00	1.0	0.03	257.0	1.00
n_1024	12.7	0.28	0.04	0.01	2.0	0.19	1025.0	1.00
n_4096	17.4	2.61	0.14	0.07	3.3	1.83	4052.2	0.99
n_16384	32.6	27.70	0.67	0.43	5.7	20.22	16159.8	0.99
n_65536	47.0	158.46	2.62	1.97	9.0	146.94	64214.6	0.98
n_262144	70.0	1193.90	12.46	10.19	18.0	1185.79	254344.3	0.97

TABLE 5.12
CPU times for problem class Grid-Long

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\lvert V\rvert$	$\lvert E\rvert$	time	itr	time	itr	time	itr
n_512	514	1008	4.7	20.3	0.2	656.7	0.7	1952.0
n_1024	1026	2000	14.3	27.0	0.4	1207.3	2.2	696.7
n_2048	2050	3984	50.2	35.3	1.2	2771.3	6.4	902.0
n_4096	4098	7952	191.9	49.7	3.0	5259.7	19.0	545.3
n_8192	8194	15888	599.0	59.3	6.9	8808.3	56.9	883.7
n_16384	16386	31760	2318.5	88.3	18.1	17732.7	159.5	1878.0
n_32768	32770	63504	7328.8	111.0	61.2	32102.0	411.2	2650.3
n_65536	65538	126992	18340.7	142.3	199.3	65849.3	1294.9	4258.0

relative to the other codes improved with problem size. On Grid-Long, it initially decreased slowly with size, but later leveled off and began increasing. On Grid-Square, a speedup ratio of 2.1 relative to RELAXT-3 was observed on the 65538 vertex instance. RELAXT-3 took 10.3 hours to solve the instance, while DLNET took 4.9 hours. On all instances NETFLO was faster than DLNET, with a speedup ratio of 3.8 on the largest instance tested. On that instance, DLNET took 70.9 hours, while NETFLO solved the problem in 18.7 hours. On Grid-Wide, DLNET was up to 46.9 times faster than RELAXT-3, solving a 65538 vertex instance in 1.0 hours while RELAXT-3 took 48.9 hours. On the largest instance (131074 vertices) DLNET was 2.2 times faster then NETFLO, solving the problem in 3.1 hours while NETFLO took 6.8 hours. On Grid-Long, NETFLO was the fastest code, solving the 65538 vertex instances on average in only 3.3 minutes while RELAXT-3 took 21.6 minutes and DLNET took 5.1 hours.

- Let T_D, T_R, T_N denote running time (in CPU seconds) of DLNET, RELAXT-3 and NETFLO, respectively. A linear regression of the running times produced the following models for:
  - Grid-Square:

$$\log_{10} T_D = .52 \log_2 |E| - 4.591 \ \ (R^2 = .9982)$$
$$\log_{10} T_N = .58 \log_2 |E| - 6.496 \ \ (R^2 = .9950)$$
$$\log_{10} T_R = .65 \log_2 |E| - 6.503 \ \ (R^2 = .9986),$$

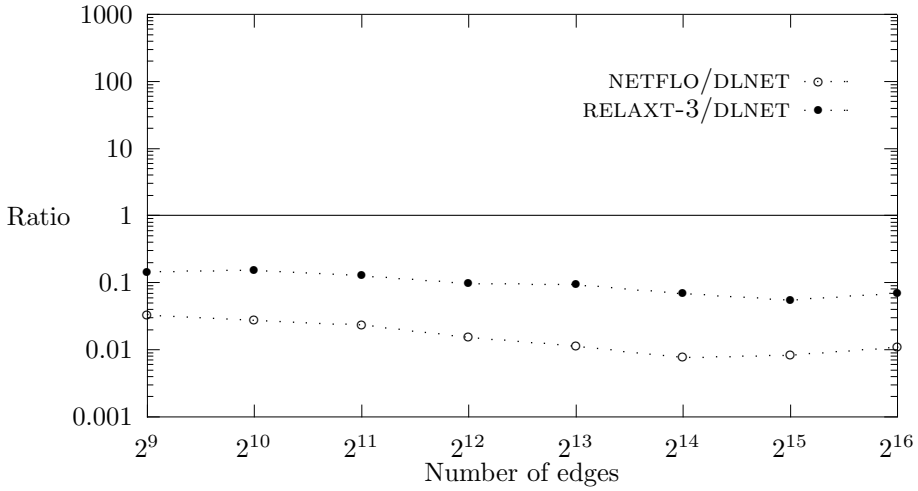

FIG. 5.7. *CPU time ratios for problem class Grid-Long*

TABLE 5.13
DLNET statistics for problem class Grid-Long

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_512	12.4	0.11	0.02	0.01	1.3	0.06	511.8	1.00
n_1024	13.7	0.31	0.03	0.01	2.0	0.19	1025.2	1.00
n_2048	18.4	0.94	0.07	0.03	3.0	0.52	2041.7	1.00
n_4096	19.5	2.84	0.14	0.06	4.7	1.62	3858.1	0.94
n_8192	23.1	7.90	0.29	0.13	5.3	4.64	7796.4	0.95
n_16384	25.8	21.53	0.61	0.35	8.7	11.78	15116.0	0.92
n_32768	29.4	55.26	1.23	0.73	10.7	29.19	29620.6	0.90
n_65536	33.5	109.16	2.39	1.69	13.3	56.24	58405.7	0.89

TABLE 5.14
CPU times for problem class Grid-Wide

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\lvert V\rvert$	$\lvert E\rvert$	time	itr	time	itr	time	itr
n_512	514	1040	3.7	19.7	0.2	956.3	1.1	3758.3
n_1024	1026	2096	10.8	26.0	1.0	2471.3	5.6	13963.7
n_2048	2050	4208	32.1	34.0	3.6	6143.7	29.4	48901.7
n_4096	4098	8432	88.3	35.7	14.0	14625.0	167.3	140421.0
n_8192	8194	16880	241.5	44.7	59.7	33214.0	1013.0	467138.7
n_16384	16386	33776	720.6	49.7	232.0	71172.7	6341.8	1607827.3
n_32768	32770	67568	1511.8	57.0	1003.3	156852.7	35670.5	5464625.0
n_65536	65538	135152	3756.6	65.3	4207.0	351092.3	176147.8	12211461.0
n_131072	131074	270320	11012.5	85.3	24619.6	786468.0	DID NOT RUN	

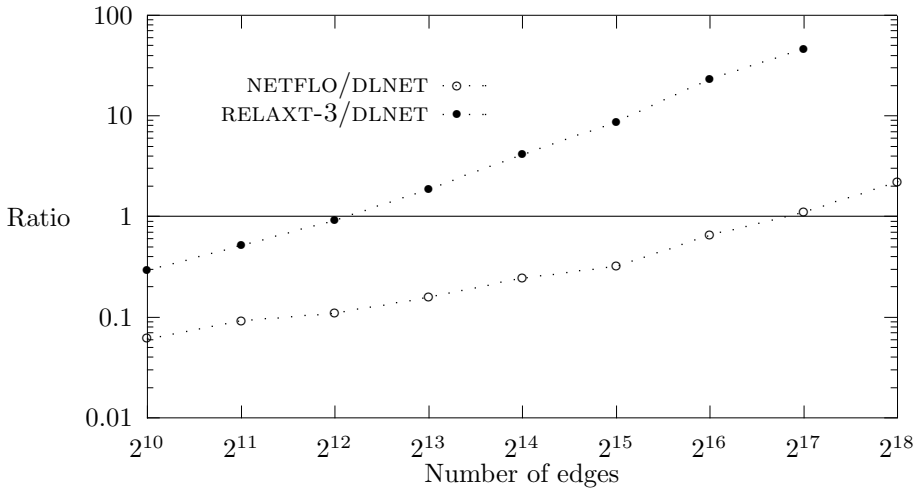

FIG. 5.8. *CPU time ratios for problem class Grid-Wide*

TABLE 5.15
DLNET statistics for problem class Grid-Wide

PROB	Conj. Grad.		Span. Tree		Max Flow			
	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_512	9.9	0.08	0.02	0.00	1.3	0.06	512.7	1.00
n_1024	10.8	0.20	0.03	0.01	2.0	0.16	1025.0	1.00
n_2048	11.2	0.48	0.06	0.03	3.0	0.50	2047.3	1.00
n_4096	10.3	1.47	0.14	0.07	3.0	1.27	4097.3	1.00
n_8192	8.7	3.02	0.29	0.16	4.0	5.31	8189.5	1.00
n_16384	9.3	8.61	0.63	0.40	4.3	14.73	16363.7	1.00
n_32768	7.6	14.79	1.35	0.91	5.0	33.14	32765.0	1.00
n_65536	6.6	28.97	3.01	2.33	6.0	89.13	65424.5	1.00
n_131072	5.5	62.24	6.86	5.73	7.7	210.68	131020.2	1.00

- Grid-Wide:

$$\log_{10} T_D = .43 \log_2 |E| - 3.675 \ \ (R^2 = .9974)$$
$$\log_{10} T_N = .62 \log_2 |E| - 6.866 \ \ (R^2 = .9987)$$
$$\log_{10} T_R = .75 \log_2 |E| - 7.532 \ \ (R^2 = .9992),$$

- Grid-Long:

$$\log_{10} T_N = .44 \log_2 |E| - 5.185 \ \ (R^2 = .9924)$$
$$\log_{10} T_R = .46 \log_2 |E| - 4.737 \ \ (R^2 = .9949)$$
$$\log_{10} T_D = .53 \log_2 |E| - 4.585 \ \ (R^2 = .9981),$$

where R^2 is the coefficient of multiple determination.

- The conjugate gradient algorithm took on average 70, 5.5, and 33.5 iterations on the largest instances of classes Grid-Square, Grid-Wide and Grid-Long,

respectively. Those instances have, respectively 262146, 131074 and 65538 vertices.

- For the largest instances of classes Grid-Square, Grid-Wide and Grid-Long, the computation of the maximum weight spanning tree (sorting and Kruskal's algorithm) took respectively 1.8%, 17% and 3.7% of the total time taken by the conjugate gradient algorithm (computing the spanning tree plus carrying out the conjugate gradient iterations). Sorting accounted for 55%, 63% and 57% of the total spanning tree computation time for classes Grid-Density-8, Grid-Density-16 and Grid-Increasing-Density, respectively.
- On the largest instances of Grid-Square, Grid-Wide and Grid-Long, DLNET spent on solving maximum flow problems in the maximum flow stopping test the equivalent of respectively 9.7%, 25.4% and 4.6% of the time spent on the conjugate gradient algorithm.
- All restricted networks of the maximum flow stopping criterion were very sparse. The largest restricted network densities for Grid-Square, Grid-Wide and Grid-Long had $|\tilde{E}|/(|V|-1)$ values of 1.00, 1.00 and 1.00, respectively.
- For Grid-Square, DLNET stopped with primal estimate stopping in 86% of the instances and with maximum flow stopping in the remaining 14%. For Grid-Wide, DLNET stopped with primal estimate stopping in all of the instances and with maximum flow stopping in none. For Grid-Long, DLNET stopped with primal estimate stopping in 96% of the instances and with maximum flow stopping in the remaining 4%. Primal-dual complementary solutions were produced for all instances of Grid-Square and Grid-Long and in 93% of the instances of Grid-Wide.
- The instances had optimal objective function values with 10 to 13 digits.

5.3.4. Minimum Cost Circulation Problems. Networks in this class are formed on a grid of vertices embedded on a torus. Edges connect vertices in the same row or column of the grid. All horizontal edges have the same orientation. Similarly, all vertical edges are oriented in the same direction. Networks in the four sub-classes Mesh-1, Mesh-2, Mesh-4 and Mesh-8 differ only with respect to vertex degree. All vertices in a network from a specific sub-class have the same degree. In Mesh-1, each vertex has an edge going to each nearest neighbor (one in the horizontal direction, the other in the vertical direction). In Mesh-2, Mesh-4 and Mesh-8, edges go from a vertex to, respectively, its 4,8 and 16 nearest neighbors. Costs are generated uniformly in the interval $[-1000, 1000]$. Capacities are generated in the interval $[-1000, 1000]$ with a bias that makes longer edges have smaller capacities.

The instances are generated with the MCNF generator `mesh.c` of Goldberg [10]. All vertex sets are $h \times w$ grids. We generate all instances with $h = w$. In Mesh-1, instances are generated having $h = w = 16, 32, \ldots, 512$. In Mesh-2, instances have $h = w = 16, 32, \ldots, 256$. In Mesh-4, $h = w = 16, 32, \ldots, 256$ and in Mesh-8 $h = w = 16, 32, \ldots, 128$.

The random number generator seeds 270001, 270002, and 270003 were used to generate different instances for each problem size. For Mesh-1, 3 instances having $256, 1024, \ldots, 65536$ vertices were generated. A single 262144 vertex instance was generated. For Mesh-2, 3 instances of sizes $256, 1024, \ldots, 65536$ vertices were generated. For Mesh-4, 3 instances of sizes $256, 1024, \ldots, 65536$ vertices were generated. NETFLO was not run on two of the 65536 vertex instances. For Mesh-8, 3 instances of sizes $256, 1024, \ldots, 16384$ vertices were generated.

Since NETFLO requires lower bounds $l = 0$ we applied the change of variables

TABLE 5.16
CPU times for problem class Mesh-1

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	256	512	1.6	15.7	0.2	1022.3	0.1	671.7
n_1024	1024	2048	8.6	17.7	3.7	5833.7	0.9	3077.3
n_4096	4096	8192	110.8	25.0	64.9	28962.3	10.3	13083.7
n_16384	16384	32768	1291.6	32.0	1426.1	151025.7	98.6	56668.7
n_65536	65536	131072	11462.6	49.3	29567.3	780700.7	857.2	226535.3
n_262144	262144	524288	95445.9	80.0	506716.0	4174524.0	13122.6	958396.0

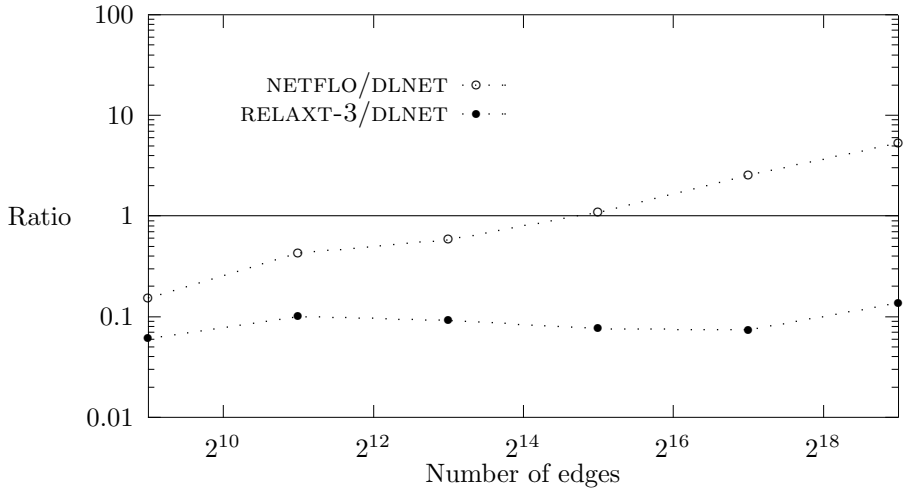

FIG. 5.9. *CPU time ratios for problem class Mesh-1*

$x' = x - l$ to

$$\min \{c^\top x \mid Ax = b,\ l \le x \le u\},$$

resulting in

$$\min \{c^\top x' + c^\top l \mid Ax' = b - Al,\ 0 \le x' \le u - l\}.$$

Tables 5.16-5.17 and Figure 5.9 summarize runs for problem class Mesh-1. Tables 5.18-5.19 and Figure 5.10 summarize runs for problem class Mesh-2. Tables 5.20-5.21 and Figure 5.11 summarize runs for problem class Mesh-4. Tables 5.22-5.23 and Figure 5.12 summarize runs for problem class Mesh-8.

We make the following observations regarding this family of problems.

- On all sub-classes, the interior point code's performance relative to the other codes improved with problem size. However, RELAXT-3 was the fastest code on all instances. On Mesh-1, a speedup ratio of 5.3 relative to NETFLO was observed on the largest instances. NETFLO took, on average, 140.8 hours, while DLNET took 26.5 hours. RELAXT-3 was 7.2 times faster than DLNET on those instances taking 3.6 hours. On Mesh-2, DLNET was up to 8.6 times faster than NETFLO, taking, on average, 3.4 hours to solve the largest instances,

TABLE 5.17
DLNET statistics for problem class Mesh-1

PROB	Conj. Grad.		Span. Tree		Max Flow			
	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	8.5	0.04	0.01	0.00	1.0	0.05	255.0	1.00
n_1024	11.2	0.25	0.03	0.01	1.0	0.23	1023.3	1.00
n_4096	13.4	2.99	0.14	0.06	2.0	3.85	4096.2	1.00
n_16384	16.5	28.79	0.65	0.40	3.0	48.12	16387.4	1.00
n_65536	20.1	169.16	3.15	2.33	4.3	313.55	65553.5	1.00
n_262144	19.7	772.43	13.70	11.41	8.0	2630.95	262212.4	1.00

TABLE 5.18
CPU times for problem class Mesh-2

PROB	SIZE		DLNET		NETFLO		RELAXT-3	
	$\lvert V\rvert$	$\lvert E\rvert$	time	itr	time	itr	time	itr
n_256	256	1024	2.7	17.0	0.7	3054.7	0.2	904.3
n_1024	1024	4096	16.3	21.7	13.9	18847.0	1.3	3645.0
n_4096	4096	16384	144.9	28.0	280.4	100593.7	16.4	16878.7
n_16384	16384	65536	1615.8	42.7	5201.7	527599.0	155.0	71540.7
n_65536	65536	262144	12297.3	53.3	106817.3	2828038.3	1408.6	295908.3

compared to 29.7 hours for NETFLO and 23.5 minutes for RELAXT-3. On average, RELAXT-3 was 8.7 times faster than DLNET on those largest instances. On the largest instances of class Mesh-4, DLNET was, on average, 15.6 times faster than NETFLO, taking 5.6 hours, compared to 83.9 hours for NETFLO and 41.9 minutes for RELAXT-3. In these instances RELAXT-3 was 7.7 times faster than DLNET. On the largest instances of Mesh-8, DLNET was, on average, 7.1 times faster than NETFLO, solving the problems in 1.6 hours, while NETFLO took 11.5 hours and RELAXT-3 8.8 minutes. RELAXT-3 was 11.0 times faster than DLNET in those instances.

- Let T_D, T_R, T_N denote running time (in CPU seconds) of DLNET, RELAXT-3 and NETFLO, respectively. A linear regression of the running times produced the following models for:
  - Mesh-1:

$$\begin{aligned}\log_{10} T_D &= .49 \log_2 |E| - 4.344 \;\; (R^2 = .9969)\\ \log_{10} T_R &= .50 \log_2 |E| - 5.570 \;\; (R^2 = .9982)\\ \log_{10} T_N &= .64 \log_2 |E| - 5.570 \;\; (R^2 = .9993),\end{aligned}$$

  - Mesh-2:

$$\begin{aligned}\log_{10} T_D &= .46 \log_2 |E| - 4.304 \;\; (R^2 = .9975)\\ \log_{10} T_R &= .50 \log_2 |E| - 5.810 \;\; (R^2 = .9955)\\ \log_{10} T_N &= .65 \log_2 |E| - 6.620 \;\; (R^2 = .9998),\end{aligned}$$

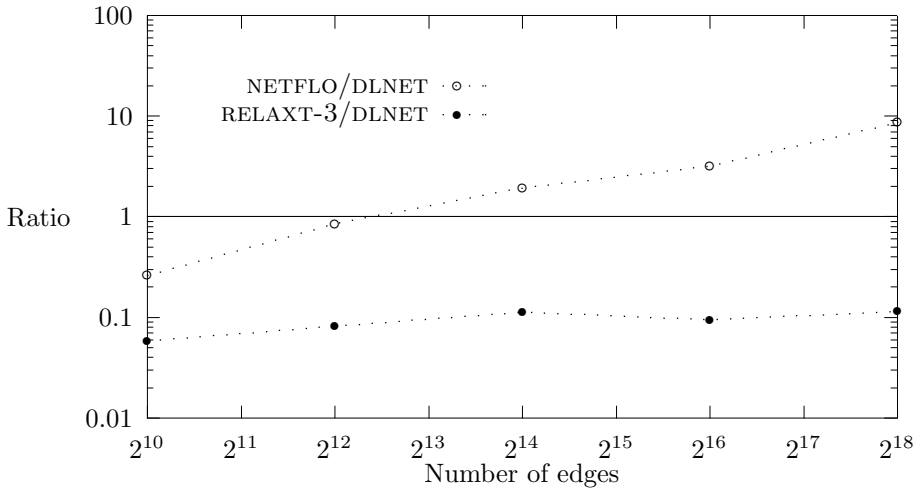

FIG. 5.10. *CPU time ratios for problem class Mesh-2*

TABLE 5.19
DLNET statistics for problem class Mesh-2

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	9.2	0.06	0.02	0.00	1.0	0.04	255.0	1.00
n_1024	9.4	0.32	0.07	0.01	1.7	0.26	1024.8	1.00
n_4096	12.2	2.86	0.30	0.08	2.0	2.61	4101.5	1.00
n_16384	14.8	22.77	1.40	0.49	3.7	33.92	16404.0	1.00
n_65536	17.1	139.79	6.46	3.00	5.3	310.95	65614.0	1.00

TABLE 5.20
CPU times for problem class Mesh-4

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\lvert V\rvert$	$\lvert E\rvert$	time	itr	time	itr	time	itr
n_256	256	2048	5.4	19.7	1.6	6761.0	0.3	994.3
n_1024	1024	8192	37.8	25.3	33.8	50395.0	2.7	4354.3
n_4096	4096	32768	309.1	34.3	798.4	285664.3	35.0	20556.3
n_16384	16384	131072	2554.1	45.0	18435.3	1512214.7	311.1	87593.0
n_65536	65536	524288	19409.6	63.3	302213.5	7659533.0	2511.8	358163.3

TABLE 5.21
DLNET statistics for problem class Mesh-4

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	9.7	0.10	0.03	0.00	1.7	0.05	255.3	1.00
n_1024	9.8	0.54	0.13	0.02	2.0	0.24	1025.8	1.00
n_4096	11.7	4.01	0.61	0.08	3.0	2.33	4108.0	1.00
n_16384	14.4	31.35	2.90	0.52	4.3	27.81	16430.7	1.00
n_65536	16.1	172.38	12.90	3.34	6.3	273.03	65737.2	1.00

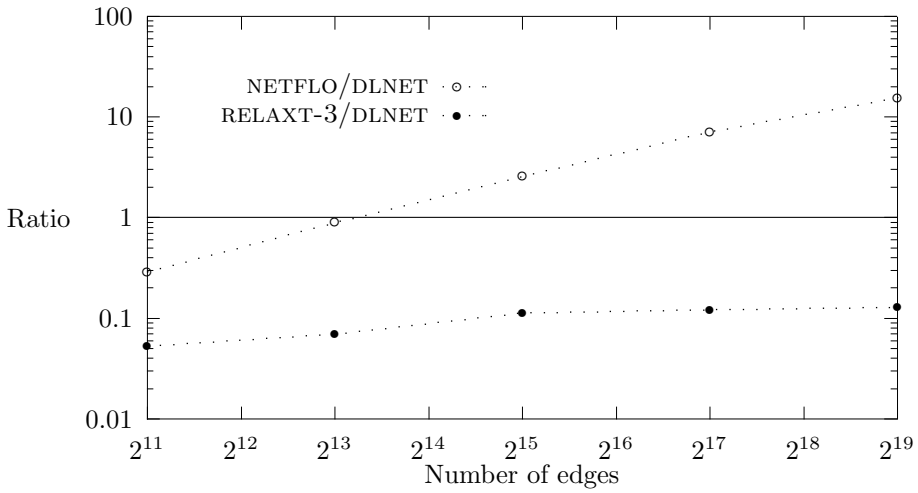

FIG. 5.11. *CPU time ratios for problem class Mesh-4*

TABLE 5.22
CPU times for problem class Mesh-8

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	256	4096	12.3	21.0	2.9	12368.0	0.6	1133.7
n_1024	1024	16384	93.1	27.7	74.7	115804.7	5.3	5365.3
n_4096	4096	65536	730.2	37.0	1806.6	712742.0	51.9	24169.3
n_16384	16384	262144	5822.2	56.7	41255.4	3688382.0	530.2	102671.0

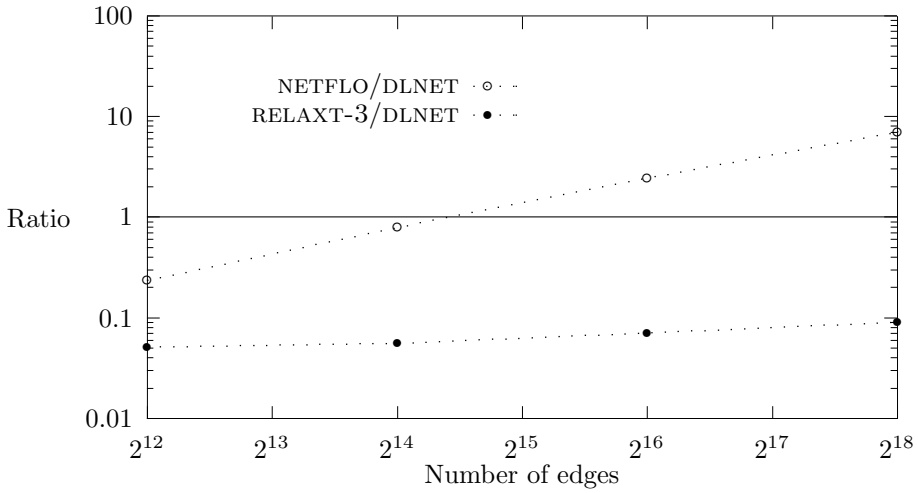

FIG. 5.12. *CPU time ratios for problem class Mesh-8*

TABLE 5.23
DLNET statistics for problem class Mesh-8

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	10.9	0.24	0.06	0.00	2.0	0.06	256.3	1.00
n_1024	12.7	1.50	0.28	0.02	2.0	0.31	1028.7	1.00
n_4096	16.4	9.96	1.32	0.11	3.3	2.96	4123.0	1.01
n_16384	19.5	56.67	5.88	0.58	5.7	30.26	16497.7	1.01

- Mesh-4:

$$\log_{10} T_D = .45 \log_2 |E| - 4.210 \ \ (R^2 = .9988)$$
$$\log_{10} T_R = .50 \log_2 |E| - 6.007 \ \ (R^2 = .9973)$$
$$\log_{10} T_N = .67 \log_2 |E| - 7.193 \ \ (R^2 = .9995),$$

- Mesh-8:

$$\log_{10} T_D = .44 \log_2 |E| - 4.268 \ \ (R^2 = .9994)$$
$$\log_{10} T_R = .49 \log_2 |E| - 6.089 \ \ (R^2 = .9982)$$
$$\log_{10} T_N = .69 \log_2 |E| - 7.831 \ \ (R^2 = .9999),$$

where R^2 is the coefficient of multiple determination.

- The conjugate gradient algorithm took on average 19.7, 17.1, 16.1 and 19.5 iterations on the largest instances of classes Mesh-1, Mesh-2, Mesh-4, and Mesh-8, respectively. Those instances have, respectively 262144, 65536, 65536, and 16384 vertices.
- For the largest instances of classes Mesh-1, Mesh-2, Mesh-4 and Mesh-8, the computation of the maximum weight spanning tree (sorting and Kruskal's algorithm) took respectively 3.1%, 6.3%, 8.6% and 10.2% of the total time taken by the conjugate gradient algorithm (computing the spanning tree plus carrying out the conjugate gradient iterations). Sorting accounted for 54.6%, 68.3%, 79.4% and 91.0% of the total spanning tree computation time for classes Mesh-1, Mesh-2, Mesh-4, and Mesh-8, respectively.
- On the largest instances of Mesh-1, Mesh-2, Mesh-4, and Mesh-8, DLNET spent on solving maximum flow problems in the maximum flow stopping test the equivalent of respectively 33.0%, 20.7%, 14.4% and 4.8% of the time spent on the conjugate gradient algorithm.
- All restricted networks of the maximum flow stopping criterion were very sparse. The largest restricted network densities for Mesh-1, Mesh-2, Mesh-4, and Mesh-8, had $|\tilde{E}|/(|V|-1)$ values of 1.00, 1.00, 1.00 and 1.01, respectively.
- For Mesh-1, DLNET stopped with primal estimate stopping in 88% of the instances and with maximum flow stopping in the remaining 12%. For Mesh-2, DLNET stopped with primal estimate stopping in 73% of the instances and with maximum flow stopping in 27%. For Mesh-4, DLNET stopped with primal estimate stopping in 67% of the instances and with maximum flow stopping in the remaining 33%. For Mesh-8, DLNET stopped with primal estimate stopping in 67% of the instances and with maximum flow stopping in the remaining 33%. Primal-dual complementary solutions were produced for all instances.
- The instances had optimal objective function values with 8 to 11 digits.

`seed`	Random number seed:	270001, 270002, 270003
`problem`	Problem number (for output):	1
`nodes`	Number of nodes:	$m = 2^x$
`sources`	Number of sources:	2^{x-2}
`sinks`	Number of sinks:	2^{x-2}
`density`	Number of (requested) arcs:	2^{x+3}
`mincost`	Minimum arc cost:	0
`maxcost`	Maximum arc cost:	4096
`supply`	Total supply:	$2^{2\,(x-2)}$
`tsources`	Transshipment sources:	0
`tsinks`	Transshipment sinks:	0
`hicost`	Skeleton arcs with max cost:	100%
`capacitated`	Capacitated arcs:	100%
`mincap`	Minimum arc capacity:	1
`maxcap`	Maximum arc capacity:	16

FIG. 5.13. *NETGEN specification file for Netgen-Lo*

TABLE 5.24
CPU times for problem class Netgen-Lo

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\|V\|$	$\|E\|$	time	itr	time	itr	time	itr
n_256	256	2048	6.5	21.3	0.7	3517.0	0.3	958.3
n_512	512	4096	16.7	23.7	2.5	9258.7	1.4	2528.0
n_1024	1024	8204	41.0	29.3	9.4	21541.7	5.0	5707.3
n_2048	2048	16415	116.3	33.7	40.2	49142.7	28.2	13035.3
n_4096	4096	32877	308.4	37.3	202.5	113142.7	83.2	31832.3
n_8192	8192	65750	1025.8	47.0	1214.1	264955.0	306.1	69652.0
n_16384	16384	131442	3414.9	55.7	8447.5	601164.3	1445.2	148248.0
n_32768	32768	262909	11833.7	76.7	53656.2	1413134.3	3680.2	316751.3
n_65536	65536	525792	29366.2	83.3	309572.9	3273724.0	19888.8	656165.0

5.3.5. NETGEN Problems. Most computational studies of network optimization codes in the past have used the MCNF generator NETGEN [20] of Klingman, Napier and Stutz. In this sub-section we test the codes on two classes of networks generated with NETGEN: Netgen-Lo and Netgen-Hi. Figure 5.13 lists the NETGEN parameters used to generate the class Netgen-Lo. Networks in class Netgen-Hi are generated with the same parameters except for `maxcap`, which is set to 16384. For both sub-classes 3 instances of each size are generated. Sizes correspond to the settings $x = 8, 9, \ldots, 16$.

Tables 5.24-5.25 and Figure 5.14 summarize runs for problem class Netgen-Lo. Tables 5.26-5.27 and Figure 5.15 summarize runs for problem class Netgen-Hi.

We make the following observations regarding this family of problems.

- With respect to NETFLO, the relative speedup of DLNET increases with size for both sub-classes. On Netgen-Lo, an average speedup of 10.5 was observed for the 65526 vertex instances with DLNET taking an average of 8.2 hours while NETFLO took 86.0 hours. On Netgen-Hi, an average speedup of 6.6 was observed for the 65526 vertex instances with DLNET taking an average of 18.7 hours while NETFLO took 123.9 hours. With respect to RELAXT-3, the relative speedup of DLNET increased with size for Netgen-Lo, but decreased for Netgen-Hi. On the 65536 vertex Netgen-Lo instances, RELAXT-3 was

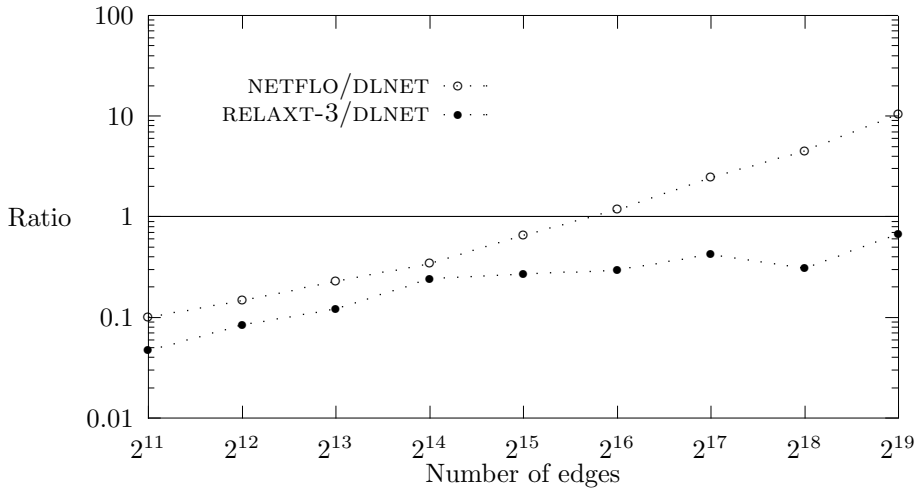

FIG. 5.14. *CPU time ratios for problem class Netgen-Lo*

TABLE 5.25
DLNET statistics for problem class Netgen-Lo

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	9.8	0.11	0.04	0.01	1.7	0.03	255.3	1.00
n_512	10.1	0.32	0.07	0.02	2.0	0.08	512.3	1.00
n_1024	9.6	0.62	0.14	0.03	2.3	0.19	1033.0	1.01
n_2048	10.2	1.78	0.31	0.08	3.0	0.69	2060.7	1.01
n_4096	11.1	4.86	0.60	0.17	3.0	2.21	4132.6	1.01
n_8192	12.6	13.60	1.31	0.46	4.3	12.13	8233.7	1.00
n_16384	14.2	40.32	2.95	1.28	5.3	46.18	16438.5	1.00
n_32768	15.8	101.37	6.41	3.03	7.7	158.25	33029.0	1.01
n_65536	17.5	237.19	12.83	6.58	8.3	409.82	66054.9	1.01

TABLE 5.26
CPU times for problem class Netgen-Hi

	SIZE		DLNET		NETFLO		RELAXT-3	
PROB	$\lvert V\rvert$	$\lvert E\rvert$	time	itr	time	itr	time	itr
n_256	256	2048	8.1	29.7	0.2	1102.0	0.2	431.3
n_512	512	4096	22.3	34.3	0.8	2980.3	0.6	806.3
n_1024	1024	8204	58.1	38.0	3.3	7768.0	2.5	1769.7
n_2048	2048	16415	172.5	42.7	18.0	20193.3	6.8	3955.0
n_4096	4096	32877	632.8	61.3	101.4	52000.7	16.6	8107.0
n_8192	8192	65750	1430.0	58.3	705.9	140144.0	50.3	17596.7
n_16384	16384	131442	5889.2	58.7	5010.3	397343.0	139.2	39109.0
n_32768	32768	262909	19176.1	124.7	57912.7	1151210.0	380.5	90430.0
n_65536	65536	525792	67475.8	200.0	446095.4	3462448.0	1136.5	211132.3

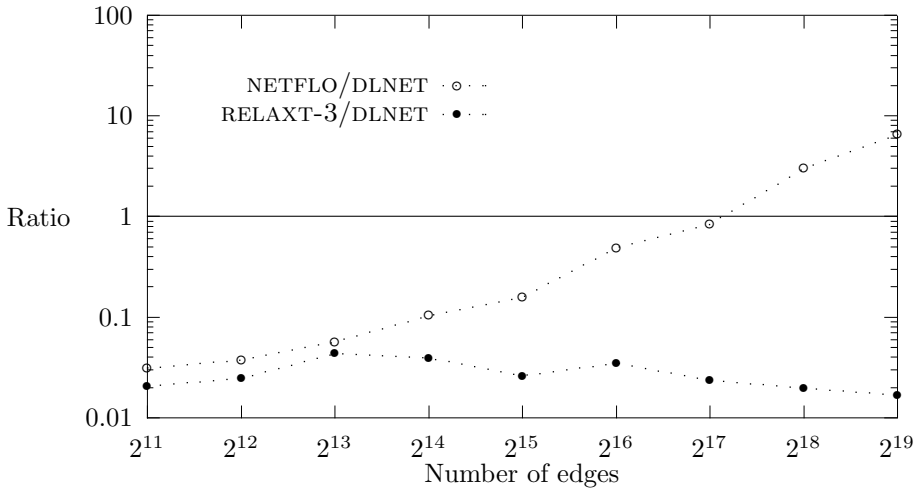

FIG. 5.15. *CPU time ratios for problem class Netgen-Hi*

TABLE 5.27
DLNET statistics for problem class Netgen-Hi

	Conj. Grad.		Span. Tree		Max Flow			
PROB	itr	time	sort	Kruskal	calls	time	$\lvert\tilde{E}\rvert$	$\frac{\lvert\tilde{E}\rvert}{\lvert V\rvert-1}$
n_256	10.5	0.11	0.03	0.01	2.3	0.01	247.6	0.97
n_512	10.9	0.30	0.06	0.02	3.0	0.03	499.8	0.98
n_1024	13.4	0.81	0.13	0.04	3.0	0.07	1023.2	1.00
n_2048	14.8	2.49	0.29	0.09	4.0	0.26	1944.3	0.95
n_4096	15.7	7.10	0.60	0.20	5.3	0.55	3980.9	0.97
n_8192	15.6	17.13	1.31	0.50	5.3	3.19	7768.2	0.95
n_16384	31.7	84.12	2.86	1.26	5.7	8.88	15598.0	0.95
n_32768	29.0	167.00	6.30	2.70	12.3	42.75	30670.5	0.94
n_65536	21.1	268.14	12.86	7.04	20.0	124.58	63939.0	0.98

on average 1.5 times faster than DLNET, with DLNET taking an average of 8.2 hours while RELAXT-3 took 5.5 hours. On the 65536 vertex Netgen-Hi instances, an average speedup of 59.4 was observed with DLNET taking an average of 18.7 hours while RELAXT-3 took only 18.9 minutes.

- RELAXT-3 was the most sensitive code to changes in the edge capacities. For the 65536 vertex networks RELAXT-3 was 17.5 faster on the Netgen-Hi networks than on the Netgen-Lo. DLNET was 2.3 times faster on the 65536 vertex Netgen-Lo instances than on Netgen-Hi. NETFLO was 1.4 times faster on the 65536 vertex Netgen-Lo instances than on Netgen-Hi.
- Let T_D, T_R, T_N denote running time (in CPU seconds) of DLNET, RELAXT-3 and NETFLO, respectively. A linear regression of the running times produced the following models for:

- Netgen-Lo:

$$\log_{10} T_D = .46 \log_2 |E| - 4.399 \ \ (R^2 = .9961)$$
$$\log_{10} T_R = .59 \log_2 |E| - 6.942 \ \ (R^2 = .9972)$$
$$\log_{10} T_N = .71 \log_2 |E| - 8.174 \ \ (R^2 = .9842),$$

- Netgen-Hi:

$$\log_{10} T_R = .47 \log_2 |E| - 5.883 \ \ (R^2 = .9914)$$
$$\log_{10} T_D = .49 \log_2 |E| - 4.534 \ \ (R^2 = .9944)$$
$$\log_{10} T_N = .77 \log_2 |E| - 9.418 \ \ (R^2 = .9900),$$

where R^2 is the coefficient of multiple determination.

- The conjugate gradient algorithm took on average 17.5, and 21.1 iterations for the 65536 vertex instances of classes Netgen-Lo, and Netgen-Hi, respectively.
- For the 65536 vertex instances of classes Netgen-Lo and Netgen-Hi the computation of the maximum weight spanning tree (sorting and Kruskal's algorithm) took respectively 7.6%, and 6.9% of the total time taken by the conjugate gradient algorithm (computing the spanning tree plus carrying out the conjugate gradient iterations). Sorting accounted for 66% and 65% of the total spanning tree computation time for classes Netgen-Lo and Netgen-Hi, respectively.
- On the 65536 vertex instances of Netgen-Lo and Netgen-Hi, DLNET spent on solving maximum flow problems in the maximum flow stopping test the equivalent of respectively 15.9%, and 4.3% of the time spent on the conjugate gradient algorithm.
- The restricted networks of the maximum flow stopping criterion were very sparse. The largest restricted network densities for Netgen-Lo and Netgen-Hi had $|\tilde{E}|/(|V| - 1)$ values of 1.01 and 1.00, respectively.
- For Netgen-Lo, DLNET stopped with primal estimate stopping in 70% of the instances and with maximum flow stopping in the remaining 30%. For Netgen-Hi, DLNET stopped with primal estimate stopping in 85% of the instances and with maximum flow stopping in the remaining 15%. Primal-dual complementary solutions were produced for 81% of the Netgen-Lo instances and 93% of the Netgen-Hi instances.
- The instances had optimal objective function values with 7 to 13 digits.

5.4. Discussion. We conclude this section with a discussion based on statistics taken from all instances solved in the experiment. Two tables and two figures illustrate this discussion. Table 5.1 shows how DLNET stopped and what type of optimal solutions were generated. Table 5.28 shows the regression models for each problem class and for the combined set of problems. Figure 5.16 plots running times for all codes on all instances in the experiment. Figure 5.17 plots conjugate gradient iterations for all instances tested.

We make the following observations regarding the experimental results.

- All codes solved all 242 instances to optimality. Instances in the problem set had up to 13-digit optimal objective function values.
- Overall, DLNET was both the fastest code and had the most predictable running times. A linear regression of each family-code data set was made. The regression model used was

$$\log_{10} T = \beta_1 \ \log_2 |E| + \beta_0,$$

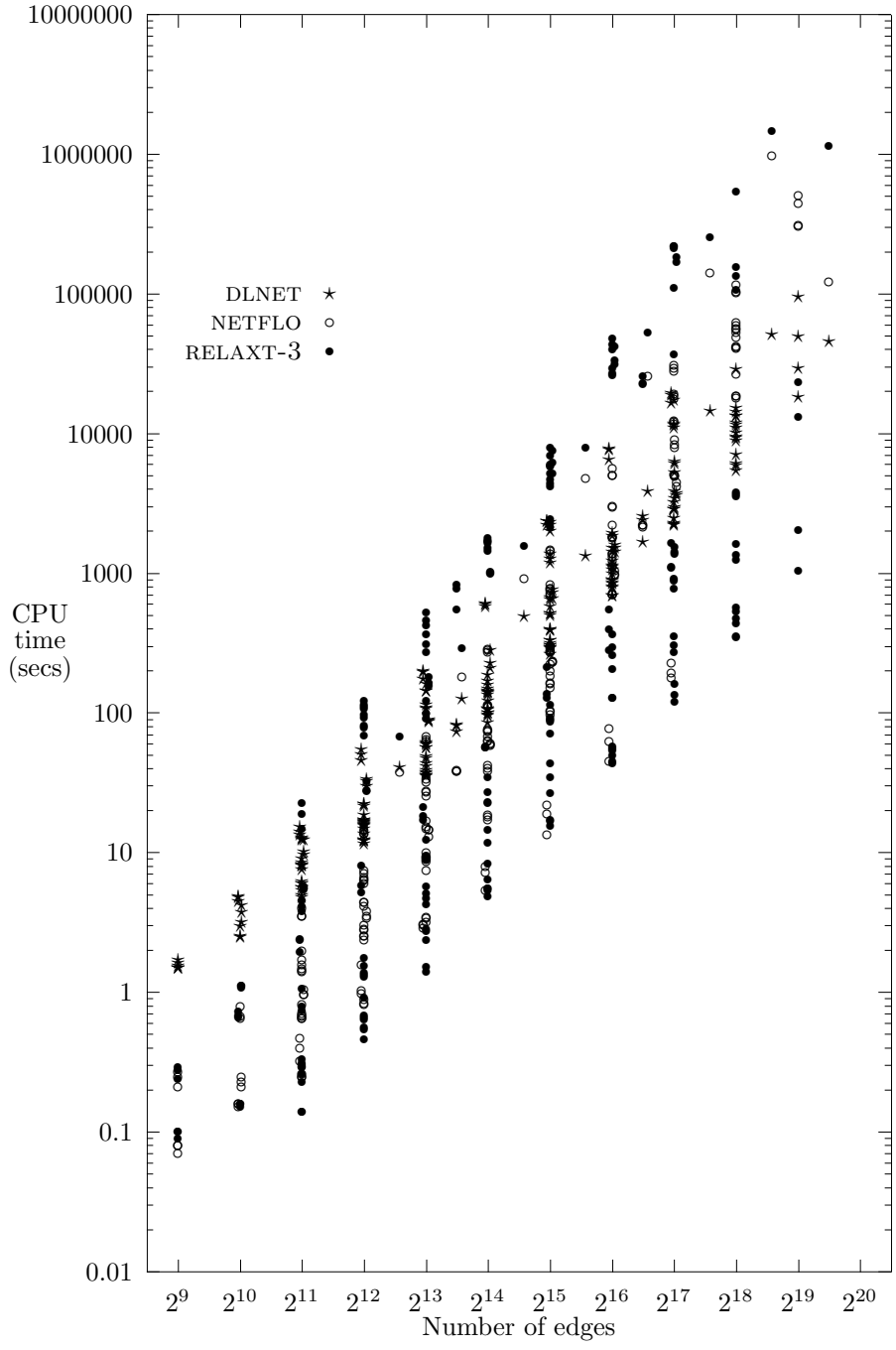

FIG. 5.16. *CPU times for all problem classes*

TABLE 5.28
Regression coefficients for all family-code combinations

Problem Class	Code	β_0	β_1	R^2
Grid-Density-16	DLNET	-4.072	0.44	0.9967
	NETFLO	-7.331	0.66	0.9944
	RELAXT-3	-5.615	0.63	0.9975
Grid-Density-8	DLNET	-4.449	0.46	0.9903
	NETFLO	-7.354	0.68	0.9890
	RELAXT-3	-6.041	0.66	0.9954
Grid-Increasing-Density	DLNET	-4.213	0.46	0.9976
	NETFLO	-6.179	0.58	0.9996
	RELAXT-3	-4.272	0.52	0.9964
RLG-Wide	DLNET	-4.821	0.51	0.9986
	NETFLO	-7.663	0.73	0.9992
	RELAXT-3	-7.407	0.73	0.9992
Grid-Square	DLNET	-4.591	0.52	0.9982
	NETFLO	-6.496	0.58	0.9950
	RELAXT-3	-6.503	0.65	0.9986
Grid-Wide	DLNET	-3.675	0.43	0.9974
	NETFLO	-6.866	0.62	0.9987
	RELAXT-3	-7.532	0.75	0.9992
Grid-Long	DLNET	-4.585	0.53	0.9981
	NETFLO	-5.185	0.44	0.9924
	RELAXT-3	-4.737	0.46	0.9949
Mesh-1	DLNET	-4.344	0.49	0.9969
	NETFLO	-6.419	0.64	0.9993
	RELAXT-3	-5.570	0.50	0.9982
Mesh-2	DLNET	-4.304	0.46	0.9975
	NETFLO	-6.620	0.65	0.9998
	RELAXT-3	-5.810	0.50	0.9955
Mesh-4	DLNET	-4.210	0.45	0.9988
	NETFLO	-7.193	0.67	0.9995
	RELAXT-3	-6.007	0.50	0.9973
Mesh-8	DLNET	-4.268	0.44	0.9994
	NETFLO	-7.831	0.69	0.9999
	RELAXT-3	-6.089	0.49	0.9982
Netgen-Lo	DLNET	-4.399	0.46	0.9961
	NETFLO	-8.174	0.71	0.9942
	RELAXT-3	-6.942	0.59	0.9972
Netgen-Hi	DLNET	-4.534	0.49	0.9944
	NETFLO	-9.418	0.77	0.9900
	RELAXT-3	-5.883	0.47	0.9914
All instances	DLNET	-4.058	0.45	0.9555
	NETFLO	-7.178	0.65	0.9253
	RELAXT-3	-5.316	0.52	0.6458

where T is the CPU time in seconds, $|E|$ is the number of edges and β_1, β_0 are the regression parameters to be estimated. Table 5.28 summarizes the results of the regression. The term R^2 is the coefficient of multiple determination. The fit was good ($R^2 > 0.99$) for all family-code combinations. On the

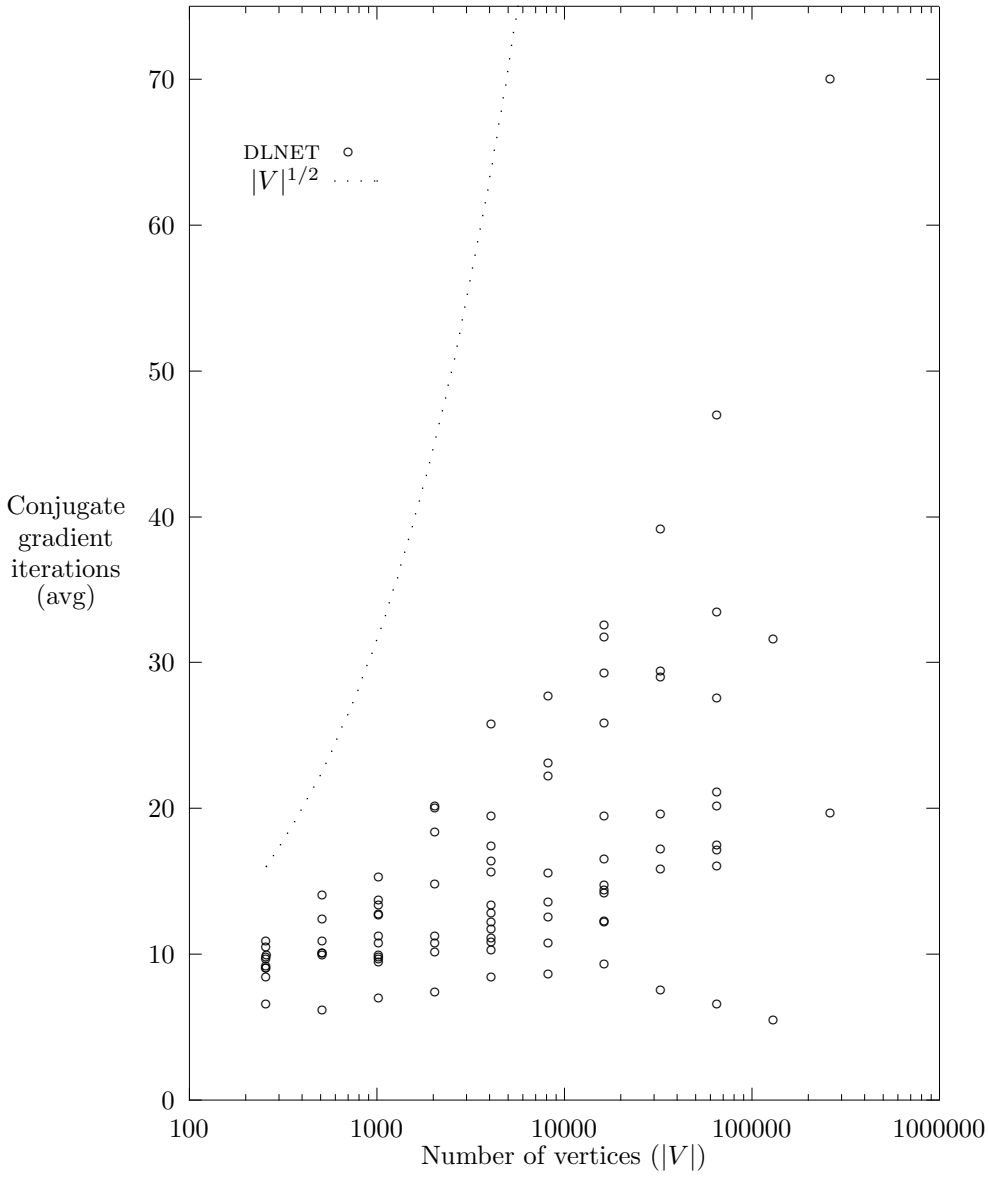

FIG. 5.17. *Average CG iterations for all problem classes*

complete set of data, the regression models were

$$\log_{10} T_D = .45 \log_2 |E| - 4.058 \ \ (R^2 = .9555)$$
$$\log_{10} T_R = .52 \log_2 |E| - 5.316 \ \ (R^2 = .6458)$$
$$\log_{10} T_N = .65 \log_2 |E| - 7.178 \ \ (R^2 = .9253),$$

indicating that asymptotically DLNET was the fastest, followed by RELAXT-

3 and then by NETFLO. For small networks, the model ranks the codes in the opposite order. The coefficients of multiple determination R^2 show that DLNET had the most predictable running times, while RELAXT-3 had the most variability. Figure 5.16 illustrates well the variability of the running times of RELAXT-3.

- It is well known that interior point algorithms take few iterations in practice. To take advantage of this phenomenon the computation of the ascent direction must be carried out efficiently. Figure 5.17 illustrates the effectiveness of the preconditioners implemented in DLNET. It shows that all average conjugate gradient iteration counts fell well below the $|V|^{1/2}$ level, considered to be the boundary between good and poor preconditioners.
- The computation of the maximum weight spanning tree took anywhere from 0.4 to 14% of the total conjugate gradient running time. Sorting, in most cases, accounted for over half of the spanning tree computation time, going from a minimum of 1.4% to a maximum of 91%. Kalinski and Ye [15] have implemented a sorting scheme that makes use of the fact that the dual slacks change very little from iteration to iteration. Using such a sorting scheme could reduce the conjugate gradient running time by perhaps 10% in some instances.
- Because the DAS algorithm allows for inexact ascent directions, a high conjugate gradient stopping tolerance of 10^{-3} was used successfully. Even though tightening this tolerance could reduce the DAS iteration count (by using better directions), the increased conjugate gradient running times do not justify this tightening.
- The primal estimate stopping scheme worked well on instances with no or little dual degeneracy. DLNET stopped in 72% of the instances with this scheme. In those, the projected dual solution identified the optimal face in 95% of the time. In the remaining 4%, DLNET produced an optimal primal integer solution and a dual interior solution with a duality gap of less than 1. By not allowing DLNET to stop with the absolute duality gap criterion, primal-dual complementary solutions could have been produced for all of the instances.
- For dual degenerate instances, DLNET stopped with the maximum flow stopping criterion. This occurred in 28% of the instances. By definition, all of those solutions are primal-dual complementary. The restricted networks used in the maximum flow stopping scheme were all very sparse (about the size of a tree) with the exception of the highly dual degenerate RLG-Wide instances. Dinic's algorithm is longer state-of-the-art for maximum flow computations and perhaps the maximum flow scheme running times could be improved by changing maximum flow algorithms. Perhaps the most waste was produced by starting the maximum flow stopping scheme at iteration 10 of DAS and repeating every 10 DAS iterations. DAS iterations were almost always greater than 40 and in some cases up to 200. By starting the maximum flow stopping scheme at iteration 40 and using a larger interval, of say 15 or 20 DAS iterations between tests, DLNET running times could be halved in some instances.

6. Concluding Remarks. Efficient implementations of variants of the network simplex algorithm and the relaxation method currently make up the set of tools used by analysts to solve large scale MCNF problems. In this study, we have introduced a new tool: the network interior point code DLNET.

In the computational experiments described in this study, DLNET proved to be robust, not failing to find an integer primal optimal solution a single time while using the same parameter settings throughout. Furthermore, it produced integer primal-dual complementary pairs for the majority of problems solved. When it did not produce a complementary pair, DLNET found a primal integer optimal solution. For those instances, forcing a few more DAS iterations would probably produce the primal-dual pair.

We showed that DLNET can be more efficient than NETFLO and RELAXT-3 in several classes of large MCNF problems. In the experiments DLNET was up to 85 times faster than RELAXT-3 and 18 times faster than NETFLO.

There were classes of problems where DLNET was not the fastest. Even for these instances DLNET proved to be competitive with the third code (with the exception of problem class Grid-Long).

Ongoing work on DLNET includes optimizing the conjugate gradient code, implementing a more efficient maximum flow algorithm, implementing better heuristics to control the transition from diagonal to spanning tree preconditioning and implementation of a centering step to avoid situations where the DAS algorithm encounters difficulty in converging.

Acknowledgment. One of the authors (M.G.C. Resende) acknowledges several insightful discussions with N. Karmarkar, K.G. Ramakrishnan and P. Vaidya. A discussion with S.T. McCormick led the authors to the idea of solving a maximum flow problem to find an integer primal-dual solution. This research was done as a part of the First DIMACS International Algorithm Implementation Challenge, organized by M.D. Grigoriadis, D.S. Johnson, C. McGeogh, C. Monma and R.E. Tarjan.

REFERENCES

[1] I. Adler, N. Karmarkar, M. Resende, and G. Veiga, *Data structures and programming techniques for the implementation of Karmarkar's algorithm*, ORSA Journal on Computing, 1 (1989), pp. 84–106.

[2] ———, *An implementation of Karmarkar's algorithm for linear programming*, Mathematical Programming, 44 (1989), pp. 297–335.

[3] A. Armacost and S. Mehrotra, *Computational comparison of the network simplex method with the affine scaling method*, Opsearch, 28 (1991), pp. 26–43.

[4] J. Aronson, R. Barr, R. Helgason, J. Kennington, A. Loh, and H. Zaki, *The projective transformation algorithm of Karmarkar: A computational experiment with assignment problems*, Tech. Report 85-OR-3, Department of Operations Research, Southern Methodist University, Dallas, TX, August 1985.

[5] E. Barnes, *A variation on Karmarkar's algorithm for solving linear programming problems*, Mathematical Programming, 36 (1986), pp. 174–182.

[6] D. Bertsekas and P. Tseng, *Relaxation methods for minimum cost ordinary and generalized network flow problems*, Operations Research, 36 (1988), pp. 93–114.

[7] A. Charnes, *Optimality and degeneracy in linear programming*, Econometrica, 20 (1952), pp. 160–170.

[8] G. Dantzig, *Maximization of a linear function of variables subject to linear inequalities*, in Activity Analysis of Production and Allocation, T. Koopsmans, ed., John Wiley and Sons, 1951, pp. 339–347.

[9] I. Dikin, *Iterative solution of problems of linear and quadratic programming*, Soviet Mathematics Doklady, 8 (1967), pp. 674–675.

[10] DIMACS, *The first DIMACS international algorithm implementation challenge: The benchmark experiments*, tech. report, DIMACS, New Brunswick, NJ, 1991.

[11] D. Goldfarb and M. Grigoriadis, *A computational comparison of the Dinic and network simplex methods for maximum flow*, Annals of Operations Research, 7 (1988), pp. 83–123.

[12] G. Golub and C. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1983.

[13] M. Grigoriadis, *An efficient implementation of the network simplex method*, Mathematical Programming Study, 26 (1986), pp. 83–111.

[14] A. Joshi, A. Goldstein, and P. Vaidya, *A fast implementation of a path-following algorithm for maximizing a linear function over a network polytope*, tech. report, Dept of Computer Science, University of Illinois, Urbana, IL, 1991.

[15] J. Kalinski and Y. Ye, *A decomposition variant of the potential reduction algorithm for linear programming*, Tech. Report 91-11, Dept of Management Sciences, The University of Iowa, Iowa City, Iowa, 1991.

[16] N. Karmarkar and K. Ramakrishnan, *Implementation and computational results of the Karmarkar algorithm for linear programming, using an iterative method for computing projections*, tech. report, AT&T Bell Laboratories, Murray Hill, NJ, 1988.

[17] ———, *Private communication*, 1988.

[18] ———, *Computational results of an interior point algorithm for large scale linear programming*, Mathematical Programming, 52 (1991), pp. 555–586.

[19] J. Kennington and R. Helgason, *Algorithms for network programming*, John Wiley and Sons, New York, NY, 1980.

[20] D. Klingman, A. Napier, and J. Stutz, *Netgen: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems*, Management Science, 20 (1974), pp. 814–821.

[21] K. McShane, C. Monma, and D. Shanno, *An implementation of a primal-dual interior point method for linear programming*, ORSA Journal on Computing, 1 (1989), pp. 70–83.

[22] S. Mehrotra and Y. Ye, *On finding the optimal facet of linear programs*, Tech. Report 91-10, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60208, 1991.

[23] C. Monma and A. Morton, *Computational experiments with a dual affine variant of Karmarkar's method for linear programming*, Operations Research Letters, 6 (1987), pp. 261–267.

[24] A. Rajan, *An empirical comparison of KORBX against RELAXT, a special code for network flow problems*, tech. report, AT&T Bell Laboratories, Holmdel, NJ, 1989.

[25] M. Resende and G. Veiga, *Computational study of two implementations of the dual affine scaling algorithm*, tech. report, AT&T Bell Laboratories, Murray Hill, NJ, 1990.

[26] ———, *An implementation of the dual affine scaling algorithm for minimum cost flow on bipartite uncapaciated networks*, tech. report, AT&T Bell Laboratories, Murray Hill, NJ, 1990. To appear in SIAM Journal on Optimization.

[27] R. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[28] M. Todd, *The effects of degeneracy and unbounded variables on variants of Karmarkar's linear programming algorithm*, in Large-scale Numerical Optimization, T. Coleman and Y. L, eds., SIAM, 1990, pp. 81–91.

[29] M. Todd and B. Burrell, *An extension to Karmarkar's algorithm for linear programming using dual variables*, Algorithmica, 1 (1986), pp. 409–424.

[30] T. Tsuchiya and M. Muramatsu, *Global convergence of the long-step affine scaling algorithm for degenerate linear programming problems*, tech. report, The Institute of Statistical Mathematics, Tokyo, January 1992.

[31] P. Vaidya, *Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners*, tech. report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1990.

[32] R. Vanderbei, M. Meketon, and B. Freedman, *A modification of Karmarkar's linear programming algorithm*, Algorithmica, 1 (1986), pp. 395–407.

[33] Y. Ye, *On the finite convergence of interior-point algorithms for linear programming*, Tech. Report 91-5, Dept of Management Sciences, The University of Iowa, Iowa City, Iowa, 1991. To appear in Mathematical Programming B.

[34] Q.-J. Yeh, *A reduced dual affine scaling algorithm for solving assignment and transportation problems*, PhD thesis, Columbia University, New York, NY, 1989.