Interior point algorithms for network flow problems*

Mauricio G.C. Resende† Panos M. Pardalos‡

Abstract

Computational algorithms for the solution of network flow problems are of great practical significance. In the last decade, a new class of computationally efficient algorithms, based on the interior point method, has been proposed and applied to solve large scale network flow problems. In this chapter, we review interior point approaches for network flows, with emphasis on computational issues.

Key Words: Network flow problems, interior point methods, computational testing, computer implementation.

1 Introduction

A large number of problems in transportation, communications, and manufacturing can be modeled as network flow problems. In these problems one seeks to find the most efficient, or optimal, way to move flow (e.g. materials, information, buses, electrical currents) on a network (e.g. postal network, computer network, transportation grid, power grid). Among these optimization problems, many are special classes of linear programming problems, with combinatorial properties that enable development of efficient solution techniques. In this chapter, we limit our discussion to these linear network flow problems. For a treatment of classes of nonlinear network flow problems, the reader is referred to [17, 28, 29, 48] and references therein.

Given a directed graph $G = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N}$ is a set of m nodes and $\mathcal{A}$ a set of n arcs, let (i, j) denote a directed arc from node i to node j. Every node is classified in one of the following three categories. *Source* nodes produce more flow than they consume. *Sink* nodes consume more flow than they produce. *Transshipment* nodes produce as much flow as they consume. Without loss of generality, one can assume that the total flow produced in the network equals the total flow consumed. Each arc has associated with it an origination node and a destination node, implying a direction for flow to follow. Arcs have limitations (often called capacities or bounds) on how much flow can move through them. The flow on arc (i, j) must be no less than l_{ij} and can be no greater than u_{ij}. To setup the problem in the framework of an optimization problem, a unit flow cost c_{ij}, incurred by each unit of flow moving through arc (i, j), must be defined. Besides being restricted by lower and upper bounds at each arc, flows must satisfy another important condition, known

*Last revision: January 30, 2003

†AT&T Bell Laboratories, Murray Hill, NJ 07974-2070 USA

‡The University of Florida, Gainesville, FL 32611-6595 USA

as Kirchhoff's Law (conservation of flow), which states that for every node in the network, the sum of all incoming flow together with the flow produced at the node must equal the sum of all outgoing flow and the flow consumed at the node. The objective of the *minimum cost network flow problem* is to determine the flow on each arc of the network, such that all of the flow produced in the network is moved from the source nodes to the sink nodes in the most cost-effective way, while not violating Kirchhoff's Law and flow limitations on the arcs. The minimum cost network flow problem can be formulated as the following linear program:

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

subject to:

$$\sum_{(j,k)\in\mathcal{A}} x_{jk} - \sum_{(k,j)\in\mathcal{A}} x_{kj} = b_j, \;\; j \in \mathcal{N} \tag{2}$$

$$l_{ij} \le x_{ij} \le u_{ij}, \;\; (i,j) \in \mathcal{A}. \tag{3}$$

In this formulation, x_{ij} denotes the flow on arc (i,j) and c_{ij} is the cost of transporting one unit of flow on arc (i,j). For each node $j \in \mathcal{N}$, let b_j denote a quantity associated with node j that indicates how much flow is produced or consumed at the node. If $b_j > 0$, node j is a source. If $b_j < 0$, node j is a sink. Otherwise ($b_j = 0$), node j is a transshipment node. For each arc $(i,j) \in \mathcal{A}$, as before, let l_{ij} and u_{ij} denote, respectively, the lower and upper bounds on flow on arc (i,j). The case where $u_{ij} = \infty$, for all $(i,j) \in \mathcal{A}$, gives rise to the *uncapacitated* network flow problem. Without loss of generality, l_{ij} can be set to zero. Most often, the problem data (i.e. c_{ij}, u_{ij}, l_{ij}, for $(i,j) \in \mathcal{A}$ and b_j, for $j \in \mathcal{N}$) are assumed to be integer, and many codes adopt this assumption. However, there can exist applications where the data are real numbers, and algorithms should be capable of handling problems with real data.

Constraints of type (2) are referred to as the flow conservation equations, while constraints of type (3) are called the flow capacity constraints. In matrix notation, the above network flow problem can be formulated as a linear program of the special form

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

where A is the $m \times n$ *node-arc incidence matrix* of the graph $G = (\mathcal{N}, \mathcal{A})$, i.e. for each arc (i,j) in $\mathcal{A}$ there is an associated column in matrix A with exactly two nonzero entries: an entry 1 in row i and an entry -1 in row j. Note that from the mn entries of A, only $2n$ are nonzero and because of this, the node-arc incidence matrix is not a space-efficient representation of the network. There are many other ways to represent a network. A popular representation is the *node-node adjacency* matrix B. This is an $m \times m$ matrix with an entry 1 in position (i,j) if arc $(i,j) \in \mathcal{A}$ and 0 otherwise. Such a representation is efficient for dense networks, but is inefficient for sparse networks. A more efficient representation of sparse networks is the *adjacency list*, where for each node $i \in \mathcal{N}$ there exists a list of arcs emanating from node i, i.e. a list of nodes j such that $(i,j) \in \mathcal{A}$. The *forward star* representation is a multi-array implementation of the adjacency list data structure. The adjacency list enables easy access to the arcs emanating from a given node, but not the incoming arcs. The *reverse star* representation enables easy access to the list of arcs incoming into i. Another representation that is much used in interior point network flow implementations is a simple *arc list*, where the arcs are stored in a linear array. The complexity of an algorithm for solving network flow problems depends greatly on the network representation and the data structures used for maintaining and updating the intermediate computations.

We denote the i-th column of A by A_i, the i-th row of A by $A_{.i}$ and a submatrix of A formed by columns with indices in set S by A_S. If graph G is disconnected and has p connected 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

$$\sum_{j \in \mathcal{N}^k} b_j = 0, \;\; k = 1, \ldots, p, \tag{4}$$

where $\mathcal{N}^k$ is the set of nodes for the k-th component of G.

Often it is further required that the flow x_{ij} be integer, i.e. we replace (3) with

$$l_{ij} \leq x_{ij} \leq u_{ij}, \; x_{ij} \text{ integer}, \; (i,j) \in \mathcal{A}. \tag{5}$$

Since the node-arc incidence matrix A is totally unimodular, when the data is integer all vertex solutions of the linear program are integer. An algorithm that finds a vertex solution, such as the simplex method, will necessarily produce an integer optimal flow. In certain types of network flow problems, such as the assignment problem, one may be only interested in solutions having integer flows, since fractional flows do not have a logical interpretation.

In the remainder of this chapter we assume, without loss of generality, that $l_{ij} = 0$ for all $(i,j) \in \mathcal{A}$ and that $c \neq 0$. A simple change of variables can transform the original problem into an equivalent one with $l_{ij} = 0$ for all $(i,j) \in \mathcal{A}$. The case where $c = 0$ is a simple feasibility problem, and can be handled by solving a maximum flow problem [3].

Many important combinatorial optimization problems are special cases of the minimum cost network flow problem. Such problems include the linear assignment and transportation problems, and the maximum flow and shortest path problems. In the transportation problem, the underlying graph is bipartite, i.e. there exist two sets $\mathcal{S}$ and $\mathcal{T}$ such that $\mathcal{S} \cup \mathcal{T} = \mathcal{N}$ and $\mathcal{S} \cap \mathcal{T} = \emptyset$ and arcs occur only from nodes of $\mathcal{S}$ to nodes of $\mathcal{T}$. Set $\mathcal{S}$ is usually called the set of source nodes and set $\mathcal{T}$ is the set of sink nodes. For the transportation problem, the right hand side vector in (2) is given by

$$b_j = \begin{cases} s_j & \text{if } j \in \mathcal{S} \\ -t_j & \text{if } j \in \mathcal{T}, \end{cases}$$

where s_j is the supply at node $j \in \mathcal{S}$ and t_j is the demand at node $j \in \mathcal{T}$. The assignment problem is a special case of the transportation problem, in which $s_j = 1$ for all $j \in \mathcal{S}$ and $t_j = 1$ for all $j \in \mathcal{T}$.

The computation of the maximum flow from node s to node t in $G = (\mathcal{N}, \mathcal{A})$ can be done by computing a minimum cost flow in $G' = (\mathcal{N}', \mathcal{A}')$, where $\mathcal{N}' = \mathcal{N}$ and $\mathcal{A}' = \mathcal{A} \cup (t,s)$, where

$$c_{ij} = \begin{cases} 0 & \text{if } (i,j) \in \mathcal{A} \\ -1 & \text{if } (i,j) = (t,s), \end{cases}$$

and

$$u_{ij} = \begin{cases} \text{cap}(i,j) & \text{if } (i,j) \in \mathcal{A} \\ \infty & \text{if } (i,j) = (t,s), \end{cases}$$

where $\text{cap}(i,j)$ is the capacity of arc (i,j) in the maximum flow problem.

The shortest paths from node s to all nodes in $\mathcal{N} \setminus \{s\}$ can be computed by solving an uncapacitated minimum cost network flow problem in which c_{ij} is the length of arc (i,j) and the right hand side vector in

(2) is given by

$$b_j = \begin{cases} m-1 & \text{if } j = s \\ -1 & \text{if } j \in \mathcal{N} \setminus \{s\}. \end{cases}$$

Although all of the above combinatorial optimization problems are formulated as minimum cost network flow problems, several specialized algorithms have been devised for solving them efficiently.

In many practical applications, flows in networks with more than one commodity need to be optimized. In the multicommodity network flow problem, k commodities are to be moved in the network. The set of commodities is denoted by $\mathcal{K}$. Let x_{ij}^k denote the flow of commodity k in arc (i,j). The multicommodity network flow problem can be formulated as the following linear program:

$$\min \sum_{k \in \mathcal{K}} \sum_{(i,j) \in \mathcal{A}} c_{ij}^k x_{ij}^k \tag{6}$$

subject to:

$$\sum_{(j,l) \in \mathcal{A}} x_{jl}^k - \sum_{(l,j) \in \mathcal{A}} x_{lj}^k = b_j^k, \ \ j \in \mathcal{N}, \ \ k \in \mathcal{K} \tag{7}$$

$$\sum_{k \in \mathcal{K}} x_{ij}^k \leq u_{ij}, \ \ (i,j) \in \mathcal{A}, \tag{8}$$

$$x_{ij}^k \geq 0, \ \ (i,j) \in \mathcal{A}, \ \ k \in \mathcal{K}. \tag{9}$$

The minimum cost network flow problem is a special case of the multicommodity network flow problem, in which there is only one commodity.

In the 1940s, Hitchcock [30] proposed an algorithm for solving the transportation problem and later Dantzig [14] developed the Simplex Method for linear programming problems. In the 1950s, Kruskal [41] developed a minimum spanning tree algorithm and Prim [51] devised an algorithm for the shortest path problem. During that decade, commercial digital computers were introduced widely. The first book on network flows was published by Ford and Fulkerson [19] in 1962. Since then, active research produced a variety of algorithms, data structures, and software for solving network flow problems. For an introduction to network flow problems and applications, see the books [3, 7, 17, 19, 39, 42, 58, 62].

The focus of this chapter is on recent computational approaches, based on interior point methods, for solving large scale network flow problems. In the last two decades, many other approaches have been developed. A history of computational approaches up to 1977 is summarized in [11]. In addition, several computational studies established the fact that specialized network simplex algorithms are orders of magnitude faster than the best linear programming codes of that time. (See, e.g. the studies in [23, 27, 39, 43]). A collection of FORTRAN codes of efficient algorithms of that period can be found in [59]. Another important class of network optimization algorithms and codes are the relaxation methods described in [8]. More recent computational research is included in [31] and [25].

In 1984 N. Karmarkar introduced a new polynomial time algorithm for solving linear programming problems [37]. This algorithm and many of its variants, known as interior point methods, have been used to efficiently solve network flow problems. This is the main topic of the remainder of this chapter.

The chapter is organized as follows. In Section 2 we provide a brief survey of the literature of the field of interior point network flow methods. A discussion of complexity issues is made. In Section 3 we focus on several important components of interior point network flow implementations, including a discussion on iterative methods for solving the large linear systems that occur in interior point methods, preconditioners,

identification of the optimal partition, and recovery of the optimal flow. These points are illustrated on a particular interior point method, the dual affine scaling algorithm. In Section 4 we present some computational results, comparing an interior point network flow method with an efficient, commercially available, network simplex code. Concluding remarks are made in Section 5.

2 Implementation of interior point network flow methods

In this section, we present issues related to efficient implementation of interior point methods for solving network flow problems. We present a brief review of research literature of interior point network flow methods and discuss the computational complexity of interior point network flow methods.

2.1 Literature review

After the introduction of Karmarkar's algorithm in 1984, many groups of researchers hurried to implement the method. One of the first implementations is described in Adler et al. [1, 2]. They implemented what is now known as the dual affine scaling algorithm using direct factorization for solving, at each iteration, a direction-finding system of linear equations, of the form

$$ADA^\top u = t, \tag{10}$$

where A is an $m \times n$ constraint matrix, D is a diagonal $n \times n$ scaling matrix, and u and t are m-vectors. Though that implementation was shown to compare favorably with an efficient implementation of the simplex method [44] on general linear programming problems from the test problem set NETLIB [20], the code was orders of magnitude slower than the network simplex implementation NETFLO [39] on small assignment problems.

At the same time, Aronson et al. [5] implemented PTANET, a variant of Karmarkar's projective transformation algorithm, for solving network flow problems. The code, an implementation of a primal projective algorithm, used the LSQR subroutine of Paige and Saunders [46] to solve (10). Their implementation took advantage of the network structure to build the Q-factor in the LSQR method. No preconditioners were used, possibly explaining the large number of LSQR iterations taken. For example, a 20×20 linear assignment problem took 6931 LSQR iterations in 7 interior point iterations before stalling at 93% of the optimum. They limited their computational testing on small dense assignment problems of size up to 80×80. PTANET was compared with NETFLO and IBM MPSX/370 and was found to be hundreds of times slower than NETFLO and tens of times slower than IBM MPSX/370.

An implementation of the dual affine scaling method with centering, called the approximate dual projective algorithm (ADP) was described in 1988 at the Tokyo Symposium of the Mathematical Programming Society by Karmarkar and Ramakrishnan [38]. Their code had a key ingredient for solving network flow problems. They introduced a preconditioned conjugate gradient algorithm for solving (10). ADP obtained a feasible dual solution in an initial phase, an ε-optimal dual solution in a second phase, terminating in a third phase, with an ε-optimal primal-dual solution. They reported computational results on maximum flow, minimum cost network flow problems on grid graphs and compared their results only with the general linear programming code MINOS [44]. On a 100×100 grid graph, ADP was 123 times faster than MINOS and on a 200×200 graph, ADP was 252 times faster.

At the same time, a computational study [52] comparing the dual affine scaling implementation in AT&T's KORBX mathematical programming system [12] with the implementation of Bertsekas' relaxation algorithm, RELAXT-3 [10], on a set of problems generated with NETGEN [40]. RELAXT-3 was up to two orders of magnitude faster than KORBX. The KORBX implementation solved the system in (10) using direct factorization and takes no advantage of the structure of the network.

Armacost and Mehrotra [4] report a comparison of their implementations of a network simplex method and the DAS algorithm using direct factorization for solving (10). Their DAS implementation produces an approximate dual optimal solution y^*, stopping when $b^\top(y^{k+1} - y^k)/\max(1, |b^\top y^k|) \leq 10^{-6}$. They generate assignment, transportation, and minimum cost network flow problems using NETGEN. On problems having 100 to 400 nodes and 150 to 1600 arcs, the network simplex code was 2 to 20 times faster than the interior point code. On dense 1000-node problems, the simplex code was up to 80 times faster, while on 1000-node sparse problems, it was at most 3 times faster. They conclude that the results by Karmarkar and Ramakrishnan on grid graphs suggest that the preconditioned conjugate gradient approach for solving (10) may be the most promising way to proceed.

Yeh [66] proposed and implemented the reduced dual affine scaling (RDAS) interior point algorithm for solving assignment and transportation problems. This method uses a reduced working sub-matrix of the constraint matrix to compute the direction. System (10) is solved via a preconditioned conjugate gradient method with a diagonal preconditioner M of the form $M = \text{diag}(AGA^\top)$. The method switches to a network simplex algorithm after a feasible primal basis is identified. In a comparative study with Yeh's network simplex implementation on assignment problems of dimension $n = 200, 300, 500$, and 2500, the simplex-to-RDAS CPU time ratios were, respectively, $.53, 1.01, 1.39$, and 3.57, indicating a relative improvement of the interior point method with the dimension of the problem.

Resende and Veiga [55] report computational results on a modification of the implementation of the dual affine scaling algorithm described in [1, 2] where the optimal primal basic solution $\tilde{x}$ is obtained by solving the linear system $A_{\text{MST}}\tilde{x} = b$, where A_{MST} is the sub-matrix of A corresponding to the edges of a maximum weighted spanning tree of G, with appropriately defined arc weights. The algorithm stops when

$$c^\top \tilde{x} - b^\top y^k < 1 \text{ and } \tilde{x} \geq 0,$$

where y^k is the dual iterate at step k. They compare two codes that differ only by how (10) is solved: one by direct factorization and the other by a conjugate gradient method using diagonal and maximum weighted spanning tree preconditioners. On a 3000×22000 assignment problem, they show a 116 speedup factor of the iterative solver with respect to the solver using the direct method. This study further indicated that direct methods may not be appropriate for solving systems of type (10) that occur in network flow problems.

Prior to the paper by Resende and Veiga [57], most studies that compared interior point methods with network simplex methods used their own implementations of the simplex method. In [57] an implementation of the DAS algorithm for bipartite uncapacitated networks was compared with NETFLO and RELAX on large assignment problems. The interior point code used was the one based on preconditioned conjugate gradient in [55]. The matrix-vector multiplication of the conjugate gradient method was implemented in parallel on an Alliant FX/80 computer. On a 500×500 assignment problem, the CPU time observed with eight processors was 6.5 times faster than on a single processor for the conjugate gradient and 5.5 times faster for the overall DAS algorithm. The parallel interior point code was compared with NETFLO and RELAX, both running on a single processor on the same machine. The relative performance of the DAS code with respect to the

other codes improved with the size of the network problem solved. The break even points were 20000-node networks for NETFLO and 50000-node networks for RELAX.

Kaliski and Ye [33] implemented a build-up variant of the dual potential reduction algorithm for solving transportation problems with $n \gg m$. Their implementation used the preconditioned conjugate gradient with the maximum weighted spanning tree preconditioner to solve (10). They differ from previous implementations in the way they compute the spanning tree, using a hybrid QuickSort procedure. To speed up the convergence of the duality gap, they use an indicator function to guess a dual face, and project the dual interior solution on the face and test optimality. On small problems, with 50 to 2000 nodes and arcs, they report that NETFLO was 2 to 10 times faster than their code.

In 1991, the first year-long DIMACS algorithm implementation challenge [31] was initiated. This challenge focussed on network flow and matching algorithms. A large set of network flow test problems was proposed and collected for the challenge. All codes solved a subset of those instances. Three entries in the network flow challenge involved interior point approaches.

The first approach, by Joshi, Goldstein, and Vaidya [32] described the implementation and testing of a primal path following interior point algorithm using a preconditioned conjugate gradient with the maximum weighted spanning tree preconditioner to solve (10). Their code was compared with NETFLO and RELAXT-3. Though the performance of this code, relative to NETLFO and RELAXT-3, was not competitive on most instances, it was shown to improve with problem size. The largest speedups with respect to NETFLO and RELAXT-3 were 2.4 and 25, respectively.

The second approach, by Ramakrishnan, Karmarkar, and Kamath [53], extended the ADP code by adding an initial solution module and a stopping heuristic module for handling assignment problems. The code, called ADP/A, was used to solve assignment problems having up to 32,768 nodes and was compared with the code AUCTION of Bertsekas and Castañon [9]. Ramakrishnan, Karmarkar, and Kamath report that, unlike AUCTION, their code has little CPU-time sensitivity to problem instances. Because of its high CPU-time sensitivity, AUCTION is much faster than ADP/A on some problem classes, while being much slower on others. That code was also used to solve large dense assignment problems to verify conjectures on the value of the optimal assignments of randomly distributed cost data [47]. Computational results on dense assignment problems with up to 20,000 nodes are reported.

The third approach, by Resende and Veiga [56], describes DLNET, an implementation of a dual affine scaling algorithm using diagonal and maximum weighted spanning tree preconditioned conjugate gradient algorithm to solve (10). That code was compared with NETFLO, RELAXT-3, CPLEX, and CPLEX NETOPT on 250 instances in 13 problem classes, having from 256 to 262,144 nodes. Compared to RELAXT-3 and NETFLO, DLNET had the slowest CPU-time growth rates in 11 of the 13 classes. It was up to 152 times faster than RELAXT-3 and 18 times faster than NETFLO. The study also pointed out that the variance in the CPU times for fixed size problems was smallest for the interior point code. The study also compared CPLEX and CPLEX NETOPT with NETFLO and showed that overall NETOPT was twice as fast as NETFLO, but was slower in 4 of the 13 problem classes. NETOPT was up to 180 times faster that CPLEX primal simplex algorithm.

A new indicator to identify the optimal face of network linear programs was introduced in [54] and incorporated in DLNET. Using the new indicator DLNET was able to find complementary primal-dual integer optimal solutions for all of the DIMACS instances. The DLNET version used in [56] found primal-optimal only solutions in 4% of the instances.

Using a new preconditioner (incomplete QR decomposition), Portugal et al. [49] implemented a pre-

conditioned conjugate gradient based DAS, primal dual, and predictor corrector interior point algorithms for solving linear transportation problems. On this class of problems, the new preconditioner outperforms both the diagonal and maximum weighted spanning tree preconditioners. They limit their computational experiments to instances with up to 2000 nodes, and conclude that the primal dual and predictor corrector algorithms are more appropriate for well-scaled transportation problems, while DAS is more appropriate for well-scaled assignment problems.

In [50], the infeasible-primal feasible-dual interior point algorithm for linear programming was presented and its implementation to solve network flow problems described. Using a standard primal-dual indicator function [21, 18], along with the stopping criterion, based on solving a maximum flow problem, the code PDNET, was shown to outperform DLNET, CPLEX NETOPT (ver. 3.0) [13], and CS (ver. 1.2) [25] on most classes of problems tested.

2.2 Complexity issues

Long before the discovery of polynomial-time algorithms for linear programming, strongly polynomial algorithms for many network flow problems were known [3]. The best known result for minimum cost network flow, due to Orlin [45], is $O(n^2 \log m + mn(\log m)^2)$, where, as before, m and n denote the number of nodes and arcs, respectively.

Recently, Vavasis and Ye [65] proposed a strongly polynomial interior point algorithm for minimum cost network flow problems having complexity $O(n^{6.5} \log n)$. Contrasted with the result of Tardos [61], which requires that all data be integer, the new algorithm generalizes to real-number data.

Prior to the result of Vavasis and Ye, the best known complexity result for minimum cost network interior point methods, due to Vaidya [63], was $O(m^2 \sqrt{n} \log(m\Gamma))$, where it is assumed that all costs and upper and lower bounds are integers whose magnitudes are bounded above by Γ.

Regarding complexity of multicommodity flow problems, the best known algorithm is the interior point method analyzed by Kamath and Palmon [34]. One variant of this interior point algorithm finds an exact solution in $O(k^{2.5} n^{1.5} mL \log(nDU))$, where $L = km + n$, D is the largest demand, U is the largest capacity, and k is the number of commodities. This variant uses the conjugate gradient method. The second variant, which has complexity bound of $O((k^{0.5} n^3 + km^{1.5} n^{1.5}) L \log(mDU))$, can be improved to $O((k^{0.5} n^{2.7} + km^{1.2} n^{1.5} + kn^{2.5}) L \log(mDU))$, using fast matrix multiplication. This variant uses matrix inversion and rank one updates.

Previously, the fastest known algorithm for multicommodity flow was also an interior point algorithm, due to Kapoor and Vaidya [36, 63]. This algorithm achieves a complexity bound of $O(k^3 n^{0.5} m^3 L \log(mDU))$ and reduces to $O(k^{2.5} n^{0.5} m^2 L \log(mDU))$ using fast matrix multiplication. It should be noted, however, that algorithms using fast matrix multiplication are not of practical interest, unless the dimensions are very large.

3 Components of interior point network flow methods

As seen in the literature review, since Karmarkar's breakthrough in 1984, many variants of his algorithm, including the dual affine scaling, with and without centering, reduced dual affine scaling, primal path following (method of centers), primal-dual path following, predictor-corrector primal-dual path following, and the infeasible primal-dual path following, have been used to solve network flow problems. Though these

algorithms are, in some sense, different, they share many of the same computational requirements. The key ingredients for efficient implementation of these algorithms are:

1. The solution of the linear system $ADA^\top u = t$, where D is a diagonal $n \times n$ scaling matrix, and u and t are m-vectors. This requires an iterative algorithm for computing approximate directions, preconditioners, stopping criteria for the iterative algorithm, etc.

2. The recovery of the desired optimal solution. This may depend on how the problem is presented (integer data or real data), and what type of solution is required (fractional or integer solution, ε-optimal or exact solution, primal optimal or primal-dual optimal solution, etc.).

In this section, we present in detail these components, illustrating their implementation in the dual affine scaling network flow algorithm DLNET of Resende and Veiga [56]. To illustrate the use of some of the techniques described we present plots of several quantities obtained by running DLNET on a 16384-node, 131072-arc minimum cost network flow problem. We follow closely the presentations in [56, 49, 54].

3.1 The dual affine scaling algorithm

The dual affine scaling (DAS) algorithm [6, 15, 60, 64] was one of the first interior point methods to be shown to be compete computationally with the simplex method [1, 2]. As before, 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

$$\max \{b^\top y - u^\top z \mid A^\top y - z + s = c,\ z \geq 0, s \geq 0\}, \tag{11}$$

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

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

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

$$\alpha_z = \min\{-z_i^k / (\Delta z)_i \mid (\Delta z)_i < 0,\ i = 1, \ldots, n\}$$

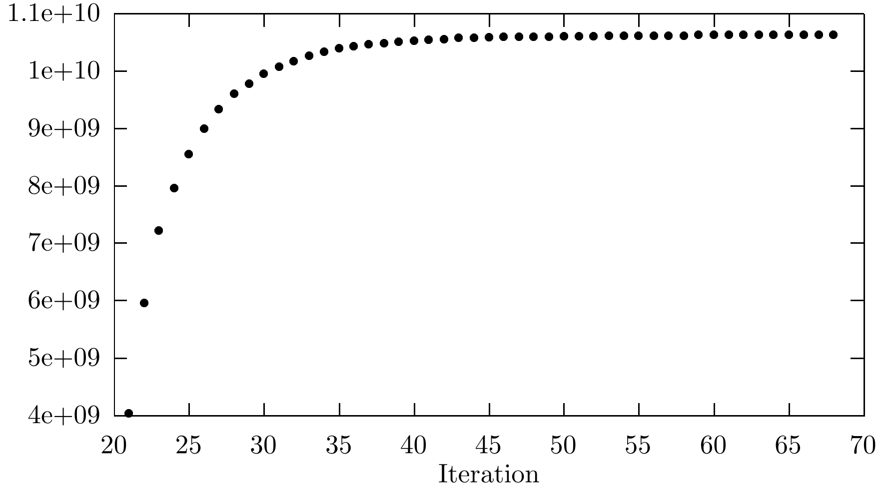

Figure 1: Interior point dual objective function (positive values)

$$\alpha_s = \min\{-s_i^k/(\Delta s)_i \mid (\Delta s)_i < 0,\ i = 1,\dots,n\}.$$

The dual problem (11) has a readily available initial interior point solution:

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

where λ is a scalar such that $\lambda > 0$ and $\lambda > -c_i$, $i = 1,\dots,n$. The algorithm described above has two important parameters, γ and λ. For example, in DLNET, $\gamma = 0.95$ and $\lambda = 2\ \|c\|_2$. Figure 1 shows the progress of the interior point dual objective function solving the 16384-node, 131072-arc example. The optimal objective function value is 10,631,080,962 and was found at iteration 69. The objective function values prior to iteration 21 are all negative and are not plotted.

3.2 Computing the direction

The computational efficiency of interior point network flow methods relies heavily on a preconditioned conjugate gradient algorithm to solve the direction finding system at each iteration. The preconditioned conjugate gradient algorithm is used to solve

$$M^{-1}(AD_kA^\top)\Delta y = M^{-1}\bar{b} \tag{12}$$

where M is a positive definite matrix and, in the case of the DAS algorithm, $\bar{b} = b - AZ_k^2 D_k^u$, and $D_k = (Z_k^2 + S_k^2)^{-1}$ is a diagonal matrix of positive elements. The objective is to make the preconditioned matrix

$$M^{-1}(AD_kA^\top) \tag{13}$$

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

```
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;
```

Figure 2: The preconditioned conjugate gradient algorithm

The pseudo-code for the preconditioned conjugate gradient algorithm is presented in Figure 2. The computationally intensive steps in the preconditioned conjugate gradient algorithm are lines 3, 7 and 11 of the pseudo-code. These lines correspond to a matrix-vector multiplication (7) and solving linear systems of 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 are of the form $AD_kA^\top\ p_i$, carried out without forming $AD_kA^\top$ explicitly. One way to compute the above matrix-vector multiplication is to decompose it into three sparse matrix-vector multiplications. Let

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

Then

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

The complexity of this matrix-vector multiplication is $O(n)$, involving n additions, $2n$ subtractions and n floating point multiplications.

The preconditioned residual is computed in lines 3 and 11 when the system of linear equations

$$Mz_{i+1} = r_{i+1}, \tag{14}$$

is solved, where M is a positive definite matrix. An efficient implementation requires a preconditioner that can make (14) easy to solve. On the other hand, one needs a preconditioner that makes (13) well conditioned. In the next subsection, we show several preconditioners that satisfy, to some extent, these two criteria.

To determine when the approximate direction Δy_i produced by the conjugate gradient algorithm is satisfactory, one can compute the angle θ between $(AD_kA^\top)\Delta y_i$ and $\bar{b}$ and stop when $|1-\cos\theta| < \varepsilon_{cos}$, where ε_{cos} is some small tolerance. In practice, one can initially use $\varepsilon_{cos} = 10^{-3}$ and tighten the tolerance as the interior point iterations proceed, as $\varepsilon_{cos} = \varepsilon_{cos} \times 0.95$. The exact computation of

$$\cos\theta = \frac{|\bar{b}^\top (AD_kA^\top)\Delta y_i|}{\|\bar{b}\|_2 \cdot \|(AD_kA^\top)\Delta y_i\|_2}$$

has the complexity of one conjugate gradient iteration and is therefore expensive if computed at each conjugate gradient iteration. One way to proceed is to compute the cosine every l_{cos} conjugate gradient iterations. A more efficient procedure follows from the observation that $(AD_kA^\top)\Delta y_i$ is approximately equal to $\bar{b} - r_i$, where r_i is the estimate of the residual at the i-th conjugate gradient iteration. Using this approximation, the cosine can be estimated by

$$\cos\theta = \frac{|\bar{b}^\top (\bar{b} - r_i)|}{\|\bar{b}\|_2 \cdot \|(\bar{b} - r_i)\|_2}.$$

Since, in practice, the conjugate gradient method finds good directions in few iterations, this estimate has been shown to be effective and can be computed at each conjugate gradient iteration. Figure 3 shows the accuracy of the direction computed by the preconditioned conjugate gradient algorithm on the 16384-node, 131072-arc example. The tolerance ε_{cos} was set to 10^{-3} on all interior point iterations, except iteration 13. On that iteration, the conjugate gradient reached the limit of 256 ($= 2 \times \sqrt{16384}$) iterations without attaining the desired precision. Stopping was tested every 5 conjugate gradient iterations, and because of that the $|1-\cos\theta|$ value was alway smaller than 10^{-3}. The diagonal preconditioner was used from iteration 1 to iteration 13. From iteration 14 on the spanning tree preconditioner took effect.

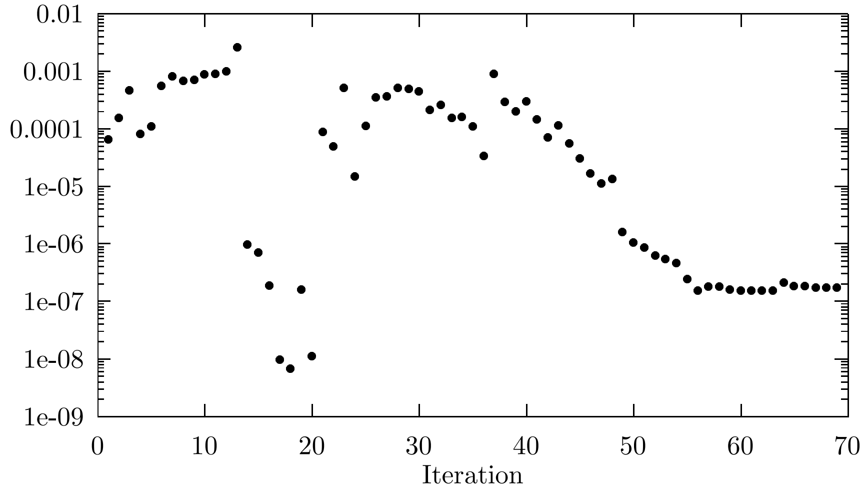

Figure 3: $|1-\cos\theta|$ value at termination of conjugate gradient

3.3 Network preconditioners for conjugate gradient method

A useful preconditioner for the conjugate gradient algorithm must be one that allows the efficient solution of (14), while at the same time causing the number of conjugate gradient iterations to be small. Four preconditioners have been found useful in conjugate gradient based interior point network flow methods: diagonal, maximum weighted spanning tree, incomplete QR decomposition, and the Karmarkar-Ramakrishnan preconditioner for general linear programming.

A diagonal matrix constitutes the most straightforward preconditioner used in conjunction with the conjugate gradient algorithm [26]. They are simple to compute, taking $O(n)$ double precision operations, and can be effective [55, 57, 66]. In the diagonal preconditioner, $M = \text{diag}\ (AD_kA^\top)$, and the preconditioned residue systems of lines 3 and 11 of the conjugate gradient pseudo-code in Figure 2 can each be solved in $O(m)$ double precision divisions.

A preconditioner that is observed to improve with the number of interior point iterations is the maximum weighted spanning tree preconditioner. Since the underlying graph G is not necessarily connected, one can identify a maximal forest using as weights the diagonal elements of the current scaling matrix,

$$w = D_k e, \tag{15}$$

where e is a unit n-vector. In practice, Kruskal's and Prim's algorithm have been used to compute the maximal forest. Kruskal's algorithm, implemented with the data structures in [62] has been applied to arcs, ordered approximately with a bucket sort [32, 56], or exactly using a hybrid QuickSort [33]. Prim's algorithm is implemented in [49] using the data structures presented in [3].

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

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

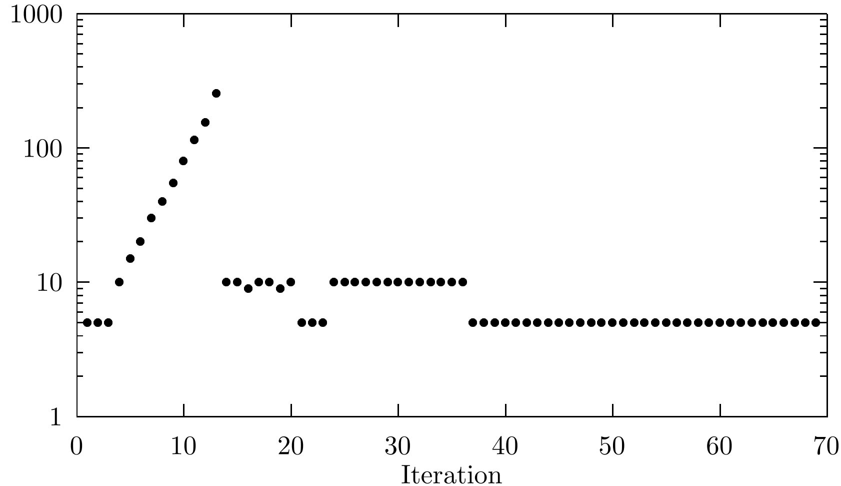

Figure 4: Conjugate gradient iterations

where, for example, in the DAS algorithm

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

For simplicity of notation, we include in $\mathcal{S}_k$ the linear dependent rows corresponding to the redundant flow conservation constraints. At each conjugate gradient iteration, the preconditioned residue system

$$(\mathcal{S}_k \mathcal{D}_k \mathcal{S}_k^\top) z_{i+1} = r_{i+1} \tag{16}$$

is solved with the variables corresponding to redundant constraints set to zero. As with the diagonal preconditioner, (16) can be solved in $O(m)$ time, as the system coefficient matrix can be ordered into a block triangular form. Figure 4 shows the number of iterations taken by the preconditioned conjugate gradient during each interior point iteration on the 16384-node, 131072-arc example. The diagonal preconditioner was used from iteration 1 to iteration 13. From iteration 14 on the spanning tree preconditioner took effect. Besides checking the angle every five iterations the conjugate gradient was also stopped by a residual stopping criterion. This occurred in iterations 16 and 19.

Portugal et al. [49] introduced a preconditioner based on an incomplete QR decomposition (IQRD) for use in interior point methods to solve transportation problems. They showed empirically, for that class of problems, that this preconditioner mimics the diagonal preconditioner during the initial iterations of the interior point method, and the spanning tree preconditioner in the final interior point method iterations, while causing the conjugate gradient method to take fewer iterations than either method during the intermediate iterations. In [50], the use of this preconditioner is extended to general minimum cost network flow problems. In the following discussion, we omit the iteration index k from notation for the sake of simplicity. Let $\bar{\mathcal{T}} = \{1,2,\ldots,n\} \setminus \mathcal{T}$ be the index set of the arcs not in the computed maximal spanning tree, and let

$$D = \begin{bmatrix} D_{\mathcal{T}} & \\ & D_{\bar{\mathcal{T}}} \end{bmatrix},$$

where $D_{\mathcal{T}} \in \mathbb{R}^{q\times q}$ is the diagonal matrix with the arc weights of the maximal spanning tree and $D_{\bar{\mathcal{T}}} \in \mathbb{R}^{(n-q)\times(n-q)}$ is the diagonal matrix with weights of the arcs not in the maximal spanning tree. Then

$$\begin{aligned} ADA^\top = & \begin{bmatrix} A_{\mathcal{T}} & A_{\bar{\mathcal{T}}} \end{bmatrix} \begin{bmatrix} D_{\mathcal{T}} & \\ & D_{\bar{\mathcal{T}}} \end{bmatrix} \begin{bmatrix} A_{\mathcal{T}}^\top \\ A_{\bar{\mathcal{T}}}^\top \end{bmatrix} \\ = & \begin{bmatrix} A_{\mathcal{T}} D_{\mathcal{T}}^{\frac{1}{2}} & A_{\bar{\mathcal{T}}} D_{\bar{\mathcal{T}}}^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} D_{\mathcal{T}}^{\frac{1}{2}} A_{\mathcal{T}}^\top \\ D_{\bar{\mathcal{T}}}^{\frac{1}{2}} A_{\bar{\mathcal{T}}}^\top \end{bmatrix}. \end{aligned}$$

The Cholesky factorization of $ADA^\top$ can be found by simply computing the QR factorization of

$$\bar{A} = \begin{bmatrix} D_{\mathcal{T}}^{\frac{1}{2}} A_{\mathcal{T}}^\top \\ D_{\bar{\mathcal{T}}}^{\frac{1}{2}} A_{\bar{\mathcal{T}}}^\top \end{bmatrix}. \tag{17}$$

In fact, if $Q\bar{A} = R$, then

$$ADA^\top = \bar{A}^\top \bar{A} = R^\top Q^\top QR = R^\top R.$$

The computation of the QR factorization is not recommended here, since besides being more expensive than a Cholesky factorization, it also destroys the sparsity of the matrix $\bar{A}$. Instead, Portugal et al. [49] propose an *incomplete* QR decomposition of $\bar{A}$. Applying Givens rotations [22] to $\bar{A}$, using the diagonal elements of $D_{\mathcal{T}}^{\frac{1}{2}} A_{\mathcal{T}}^\top$, the elements of $D_{\bar{\mathcal{T}}}^{\frac{1}{2}} A_{\bar{\mathcal{T}}}^\top$ become null. No fill-in is incurred in this factorization. See [49] for an example illustrating this procedure. After the factorization, we have the preconditioner

$$M = F\mathcal{D}F^\top,$$

where F is a matrix with a diagonal of ones that can be reordered to triangular form, and $\mathcal{D}$ is a diagonal matrix with positive elements.

To avoid square root operations, $\mathcal{D}$ and F are obtained without explicitly computing $\mathcal{D}^{\frac{1}{2}} F^\top$. Suppose that the maximum spanning tree is rooted at node r, corresponding to the flow conservation equation that has been removed from the formulation. Furthermore, let $\mathcal{A}_{\mathcal{T}}$ denote the subset of arcs belonging to the tree and let ρ_i represent the predecessor of node i in the tree. The procedure used to compute the nonzero elements of $\mathcal{D}$ and the off-diagonal nonzero elements of F is presented in the pseudo-code in Figure 5.

The computation of the preconditioned residual with $F\mathcal{D}F^\top$ requires $O(m)$ divisions, multiplications, and subtractions, since $\mathcal{D}$ is a diagonal matrix and F can be permuted into a triangular matrix with diagonal elements equal to one. The construction of F and $\mathcal{D}$, that constitute the preconditioner, requires $O(n)$ additions and $O(m)$ divisions.

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 $AD_kA^\top$. Arguments as to why this preconditioner is effective are given in [32, 49]. The DLNET implementation begins with the diagonal preconditioner and monitors 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. Upper and lower limits to the number of DAS iterations using a diagonal preconditioned conjugate gradient are specified.

Another preconditioner used in an interior point implementation is the one for general linear programming, developed by Karmarkar and Ramakrishnan and used in [38, 53]. This preconditioner is based on a

```
procedure iqrd(T, T̄, N, D, 𝒟, F)
1     do i ∈ N \ {r} →
2          j = ρ_i;
3          if (i, j) ∈ A_T → 𝒟_ii = D_ij fi;
4          if (j, i) ∈ A_T → 𝒟_ii = D_ji fi;
5     od;
6     do (i, j) ∈ A_T̄ →
7          if i ∈ N \ {r} → 𝒟_ii = 𝒟_ii + D_ij fi;
8          if j ∈ N \ {r} → 𝒟_jj = 𝒟_jj + D_ij fi;
9     od;
10    do i ∈ N \ {r} →
11         j = ρ_i;
12         if j ∈ N \ {r} →
13              if (i, j) ∈ A_T → F_ij = D_ij / 𝒟_ii fi;
14              if (j, i) ∈ A_T → F_ji = D_ji / 𝒟_ii fi;
15         fi;
16    od;
end iqrd;
```

Figure 5: Computing the F and $\mathcal{D}$ matrices in IQRD

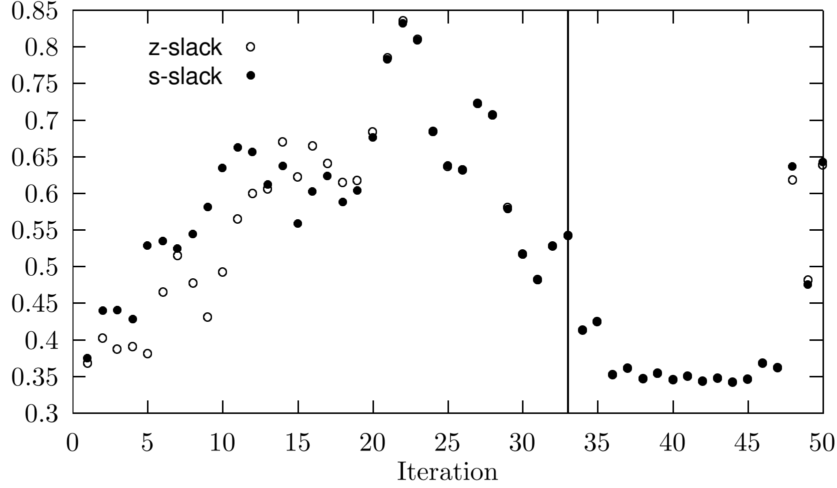

Figure 6: Dual Slack Ratio (Edge left active)

dynamic scheme to drop elements of the original scaled constraint matrix DA, as well as the from the factors of the matrix $ADA^\top$ of the linear system, and use the incomplete Cholesky factors as the preconditioner. Because of the way elements are dropped, this preconditioner mimics the diagonal preconditioner in the initial iterations and the tree preconditioner in the final iterations of the interior point algorithm.

3.4 Identifying the optimal partition

One way to stop an interior point algorithm before the (fractional) interior point iterates converge is to estimate (or guess) the optimal partition of arcs at each iteration, attempt to recover the flow from the partition and, if a feasible flow is produced, test if that flow is optimal. In the discussion that follows we describe a strategy to partition the set of arcs in the dual affine scaling algorithm. The discussion follows [54] closely, using a dual affine scaling method for uncapacitated networks to illustrate the procedure.

Let $A \in \mathbb{R}^{n\times n}$, $c,x,s \in \mathbb{R}^{n}$ and $b,y \in \mathbb{R}^{n}$. Consider the linear programming problem

$$\begin{aligned} &\text{minimize} \quad c^\top x \\ &\text{subject to} \quad Ax = b, \quad x \geq 0 \end{aligned} \tag{18}$$

and its dual

$$\begin{aligned} &\text{maximize} \quad b^\top y \\ &\text{subject to} \quad A^\top y + s = c, \quad s \geq 0. \end{aligned} \tag{19}$$

The dual affine scaling algorithm starts with an initial dual solution $y^0 \in \{y \,:\, s = c - A^\top y > 0\}$ and obtains iterate y^{k+1} from y^k according to $y^{k+1} = y^k + \alpha^k d_y^k$, where the search direction d_y is $d_y^k = (AD_k^{-2}A^\top)^{-1}b$ and $D_k = \operatorname{diag}(s_1^k,\ldots,s_n^k)$. A step moving a fraction γ of the way to the boundary of the feasible region is taken at

each iteration, namely,

$$\alpha^k = \gamma \times \min\{-s_i^k/(d_s^k)_i \ : \ (d_s^k)_i < 0, \ i = 1,...,n\}, \tag{20}$$

where $d_s^k = -A^\top d_y^k$ is a unit displacement vector in the space of slack variables. At each iteration, a tentative primal solution is computed by $x^k = D_k^{-2}A^\top(AD_k^{-2}A^\top)^{-1}b$. The set of optimal solutions is referred to as the *optimal face*. We use the index set N_* for the always-active index set on the optimal face of the primal, and B_* for its complement. It is well-known that B_* is the always-active index set on the optimal face of the dual, and N_* is its complement. An *indicator* is a quantity to detect whether an index belongs to N_* or B_*. We next describe three indicators that can be implemented in the DAS algorithm. For pointers to other indicators, see [18].

Under a very weak condition, the iterative sequence of the DAS algorithm converges to a relative interior point of a face on which the objective function is constant, i.e. the sequence $\{y^k\}$ converges to an interior point of a face on which the objective function is constant. Let B be the always-active index set on the face and N be its complement, and let b^∞ be the limiting objective function value. There exists a constant $C_0 > 0$ such that

$$\limsup_{k\to\infty} \frac{s_i^k}{b^\infty - b^\top y^k} \le C_0 \tag{21}$$

for all $i \in B$, while

$$\frac{s_i^k}{b^\infty - b^\top y^k} \tag{22}$$

diverges to infinity for all $i \in N$. Denote by s^∞ the limiting slack vector. Then $s_N^\infty > 0$ and $s_B^\infty = 0$. The vector

$$u^k \equiv \frac{(D^k)^{-1}d_s^k}{b^\infty - b^\top y^k} = \frac{D^k x^k}{b^\infty - b^\top y^k} \tag{23}$$

plays an important role, since

$$\lim_{k\to\infty} (u^k)^\top e = \lim_{k\to\infty} \frac{(s^k)^\top x^k}{b^\infty - b^\top y^k} = 1. \tag{24}$$

Consequently, in the limit $b^\infty - b^\top y^k$ can be estimated by $(s^k)^\top x^k$ asymptotically, and (21) can be stated as

$$\lim_{k\to\infty} \sup \frac{s_i^k}{(s^k)^\top x^k} \le C_0.$$

Then, if $i \in B$, for any β such that $0 < \beta < 1$,

$$\lim_{k\to\infty} \sup \frac{s_i^k}{((s^k)^\top x^k)^\beta} = 0,$$

since $((s^k)^\top x^k)^\beta$ converges to zero at a slower rate than $((s^k)^\top x^k)$ for any β such that $0 < \beta < 1$. Therefore, if $\beta = 1/2$, the following indicator has the property that $\lim_{k\to\infty} N^k = N_*$.

Indicator 1: Let $C_1 > 0$ be any constant, and define

$$N^k = \{i \in E \ : \ s_i^k \le C_1\sqrt{(s^k)^\top x^k}\}. \tag{25}$$

This indicator is available under very weak assumptions, so it can be used to detect B_* and N_* without any substantial restriction on step-size. On the other hand, it gives the correct partition only if the limit point y^∞ happens to be a relative interior point of the optimal face of the dual and thus lacks a firm theoretical justification. However, since we know by experience that y^∞ usually lies in the relative interior of the optimal

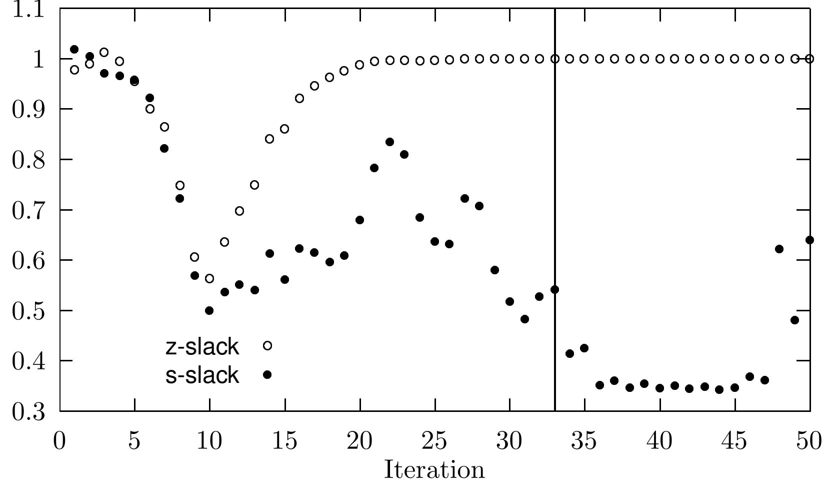

Figure 7: Dual Slack Ratio (Edge set to upper bound)

face, we may expect that it should work well in practice. Another potential problem with this indicator is that it is not scaling invariant, so that it will behave differently if the scaling of the problem is changed.

Now we assume that the step-size is asymptotically less than or equal to 2/3. Then the limiting point exists in the interior of the optimal face and b^∞ is the optimal value. Specifically, $\{y^k\}$ converges to an interior point of the optimal face of the dual problem, $\{x^k\}$ converges to the analytic center of the optimal face of the primal problem, and $\{b^\top y^k\}$ converges linearly to the optimal value b^∞ asymptotically, where the (asymptotic) reduction rate is exactly $1-\gamma$. Furthermore, one can show that

$$\lim_{k\to\infty} u_i^k = 1/|B_*| \quad \text{for } i \in B_* \tag{26}$$

$$\lim_{k\to\infty} u_i^k = 0 \quad \text{otherwise.} \tag{27}$$

The vector u^k is not available because the exact optimal value is unknown a priori, but $b^\infty - b^\top y^k$ can be estimated by $(s^k)^\top x^k$ to obtain

$$\lim_{k\to\infty} \frac{s_i^k x_i^k}{(s^k)^\top x^k} = 1/|B_*| \quad \text{for } i \in B_* \tag{28}$$

$$\lim_{k\to\infty} \frac{s_i^k x_i^k}{(s^k)^\top x^k} = 0 \quad \text{otherwise.} \tag{29}$$

On the basis of this fact, the following procedure to construct N^k, which asymptotically coincides with N_*:
Indicator 2: Let δ be a constant between 0 and 1. We obtain N^k according to the following procedure:

- Step 1: Sort $g_i^k = s_i^k x_i^k/(s^k)^\top x^k$ according to its order of magnitude. Denote i_l the index for the l-th largest component.

- Step 2: For $p := 1, 2, ...$ compare g_{i_p} and δ/p, and let p^* be the first number such that $g_{i^*_p} \leq \delta/p^*$. Then set

$$N^k = \{i_1, i_2, ..., i_{p^*-1}\}. \tag{30}$$

To state the third, and most practical indicator, let us turn our attention to the asymptotic behavior of s_i^{k+1}/s_i^k. If $i \in N_*$, then s_i^k converges to a positive value, and hence

$$\lim_{k\to\infty} \frac{s_i^{k+1}}{s_i^k} = 1. \tag{31}$$

If $i \in B_*$, s_i^k converges to zero. Since

$$\lim_{k\to\infty} \frac{s_i^k x_i^k}{b^\infty - b^\top y^k} = \frac{1}{|B_*|}, \tag{32}$$

x_i^k converges to a positive number, and the objective function reduces with a rate of $1-\gamma$, then

$$\lim_{k\to\infty} \frac{s_i^{k+1}}{s_i^k} = 1-\gamma, \tag{33}$$

which leads to the following indicator:
Indicator 3: Take a constant η such that $1-\gamma < \eta < 1$. Then let

$$N^k = \{i \, : \, \frac{s_i^{k+1}}{s_i^k} \geq \eta\} \tag{34}$$

be defined as the index set. Then $N^k = N_*$ holds asymptotically.

Of the three indicators described here, Indicators 2 and 3 stand on the firmest theoretical basis. Furthermore, unlike Indicator 1, both are scaling invariant. The above discussion can be easily extended for the case of capacitated network flow problems. DLNET uses Indicator 3 to identify the set of active arcs defining the optimal face by examining the ratio between subsequent iterates of each dual slack. At the optimum, the flow on each arc can be classified as being at its upper bound, lower bound, or as active. From the discussion above, if the flow on arc i converges to its upper bound,

$$\lim_{k\to\infty} s_i^k/s_i^{k-1} = 1-\gamma \quad \text{and} \quad \lim_{k\to\infty} z_i^k/z_i^{k-1} = 1.$$

If the flow on arc i converges to its lower bound,

$$\lim_{k\to\infty} s_i^k/s_i^{k-1} = 1 \quad \text{and} \quad \lim_{k\to\infty} z_i^k/z_i^{k-1} = 1-\gamma.$$

If the flow on arc i is active,

$$\lim_{k\to\infty} s_i^k/s_i^{k-1} = 1-\gamma \text{ and } \lim_{k\to\infty} z_i^k/z_i^{k-1} = 1-\gamma.$$

From a practical point of view, scale invariance is the most interesting feature of this indicator. An implementable version can use constants which depend only on the step size factor γ. Let $\kappa_0 = .7$ and $\kappa_1 = .9$. At each iteration of DLNET, the arcs are classified as follows:

- If $s_i^k/s_i^{k-1} < \kappa_0$ and $z_i^k/z_i^{k-1} > \kappa_1$, arc i is set to its upper bound.
- If $s_i^k/s_i^{k-1} > \kappa_1$ and $z_i^k/z_i^{k-1} < \kappa_0$, arc i is set to its lower bound.

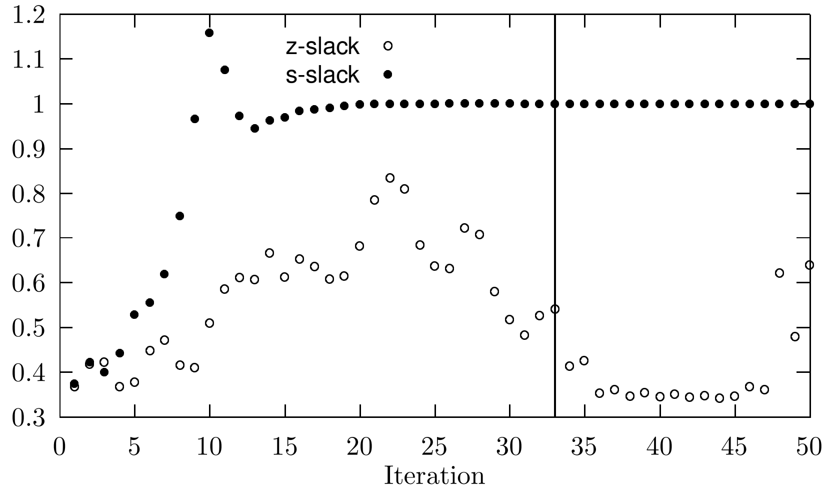

Figure 8: Dual Slack Ratio (Edge set to lower bound)

- Otherwise, arc i is set active, defining the tentative optimal face.

Figures 6, 7, and 8, illustrate the behavior of the ratios of dual slacks corresponding to arcs converging to a value away from its bounds, to its lower bound, and to its upper bound, respectively. The step size fraction used is $\gamma = 2/3$. The vertical line in each of the figures shows when the indicator successfully identified the optimal partition. The data points on the right of the vertical line illustrate the behavior of the indicator as the iterates of the algorithm converge to optimal solution.

3.5 Recovering the optimal flow

The simplex method restricts the sequence of solutions it generates to nodes of the linear programming polytope. Since the matrix A of the network linear program is totally unimodular, when a simplex variant is applied to a network flow problem with integer data, the optimal solution is also integer. On the other hand, an interior point algorithm generates a sequence of interior point (fractional) solutions. Unless the primal optimal solution is unique, the primal solution that an interior point algorithm converges to is not guaranteed to be integer. In an implementation of an interior point network flow method, one would like to be capable of recovering an integer flow even when the problem has multiple optima. We discuss below the stopping strategies implemented in DLNET and used to recover an integer optimal solution.

Besides the indicator described in subsection 3.4, DLNET uses the arcs of the spanning forest of the tree preconditioner as an indicator. If there exists a unique optimal flow, this indicator correctly identifies an optimal primal basic sequence, and an integer flow can be easily recovered by solving a triangular system of linear equations. In general, however, the arc indices do not converge to a basic sequence. Let $\mathcal{T} = \{t_1, \ldots, t_q\}$ denote the set of arc indices in the spanning forest. To obtain a tentative primal basic solution, first set flow

on arcs not in the forest to either their upper or lower bound, i.e. for all $i \in \mathcal{A} \setminus \mathcal{T}$:

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

where s^k and z^k are the current iterates of the dual slack vectors as defined in (11). The remaining basic arcs have flows that satisfy the linear system

$$A_{\mathcal{T}} x_{\mathcal{T}}^* = b - \sum_{i \in \Omega^-} u_i A_i, \tag{35}$$

where $\Omega^- = \{i \in \mathcal{A} \setminus \mathcal{T} : s_i^k \le z_i^k\}$. Because $A_{\mathcal{T}}$ can be reordered in a triangular form, (35) can be solved in $O(m)$ operations. If $u_{\mathcal{T}} \ge x_{\mathcal{T}}^* \ge 0$ then the primal solution is feasible and optimality can be tested.

Optimality can be verified producing a dual feasible solution (y^*, s^*, z^*) that is either complementary or that implies in a duality gap less than 1. The first step to build a tentative optimal dual solution is identify the set of dual constraints defining the supporting affine space of the dual face complementary to x^*,

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

i.e. the set of arcs with zero dual slacks. Since, in general, x^* is not feasible, $\mathcal{F}$ is usually determined by the indicators of subsection 3.4, as the index-set of active arcs. To ensure a complementary primal-dual pair, the current dual interior vector y^k is projected orthogonally onto this affine space. The solution y^* of the least squares problem

$$\min_{y^* \in \mathbb{R}^n} \{\|y^* - y^k\|_2 \;:\; A_{\mathcal{F}}^\top y^* = c_{\mathcal{F}}\} \tag{36}$$

is the projected dual iterate.

Let $G_{\mathcal{F}} = (\mathcal{N}, \mathcal{F})$ be the subgraph of G with $\mathcal{F}$ as its set of arcs. Since this subgraph is a forest, its incidence matrix, $A_{\mathcal{F}}$, can be reordered into a block triangular form, with each block corresponding to a tree in the forest. Assume $G_{\mathcal{F}}$ has p components, with $T_1, \ldots, T_p$ as the sets of arcs in each component tree. After reordering, the incidence matrix can be represented as

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

The supporting affine space of the dual face can be expressed as the sum of orthogonal one-dimensional subspaces. The operation in (36) can be performed by computing the orthogonal projections onto each individual subspace independently, and therefore can be completed in $O(m)$ time. For $i = 1, \ldots, p$, denote the number of arcs in T_i by m_i, and the set of nodes spanned by those arcs by $\mathcal{N}_i$. A_{T_i} is an $(m_i + 1) \times m_i$ matrix and each subspace

$$\Psi_i = \{y_{\mathcal{N}_i} \in \mathbb{R}^{m_i+1} \;:\; A_{T_i}^\top y_{\mathcal{N}_i} = c_{T_i}\}$$

has dimension one. For all $y_{\mathcal{N}_i} \in \Psi_i$,

$$y_{\mathcal{N}_i} = y_{\mathcal{N}_i}^0 + \alpha_i y_{\mathcal{N}_i}^h, \tag{37}$$

where $y_{\mathcal{N}_i}^0$ is a given solution in Ψ_i and $y_{\mathcal{N}_i}^h$ is a solution of the homogeneous system $A_{T_i}^\top y_{\mathcal{N}_i} = 0$. Since A_{T_i} is the incidence matrix of a tree, the unit vector is a homogeneous solution. The given solution $y_{\mathcal{N}_i}^0$ can be

computed by selecting $v \in \mathcal{N}_i$, setting $y_v^0 = 0$, removing the row corresponding to node v from matrix A_{T_i} and solving the resulting triangular system

$$\tilde{A}_{T_i}^\top y_{\mathcal{N}_i \setminus \{v\}} = c_{T_i}.$$

With the representation in (37), the orthogonal projection of $y_{\mathcal{N}_i}$ onto subspace Ψ_i is

$$y_{\mathcal{N}_i}^* = y_{\mathcal{N}_i}^0 + \frac{e_{\mathcal{N}_i}^\top (y_{\mathcal{N}_i} - y_{\mathcal{N}_i}^0)}{(m_i + 1)} e_{\mathcal{N}_i}$$

where e is the unit vector. The orthogonal projection, as indicated in (36), is obtained by combining the projections onto each subspace,

$$y^* = (y_{\mathcal{N}_i}^*, \ldots, y_{\mathcal{N}_q}^*).$$

A feasible dual solution is built 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 $\delta_i = c_i - A^\top y^*$.

If the solution of (35) is feasible, optimality can be checked at this point, using the projected dual solution as a lower bound on the optimal flow. The primal and dual solutions, x^* and (y^*, s^*, z^*), are optimal if complementary slackness is satisfied, i.e. if for all $i \in \mathcal{A} \setminus \mathcal{T}$ 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 $c^\top x^* - b^\top y^* + u^\top z^* < 1$.

However, in general, the method proceeds attempting to find a feasible flow x^* that is complementary to the projected dual solution y^*. Based on the projected dual solution y^*, a refined tentative optimal face is selected by redefining the set of active arcs as

$$\tilde{\mathcal{F}} = \{i \in \mathcal{A} \,:\, |c_i - A_{\cdot i}^\top y^*| < \varepsilon\}.$$

Next, the method attempts to build a primal feasible solution, x^*, complementary to the tentative dual optimal solution by setting the inactive arcs to lower or upper bounds, i.e., for $i \in \mathcal{A} \setminus \tilde{\mathcal{F}}$,

$$x_i^* = \begin{cases} 0 & \text{if } i \in \Omega^+ = \{i \in \mathcal{A} \setminus \tilde{\mathcal{F}} \,:\, c_i - A_{\cdot i}^\top y^* > 0\} \\ u_i & \text{if } i \in \Omega^- = \{i \in \mathcal{A} \setminus \tilde{\mathcal{F}} \,:\, c_i - A_{\cdot i}^\top y^* < 0\}. \end{cases}$$

By considering only the active arcs, a *restricted network* is built, represented by the constraint set

$$A_{\tilde{\mathcal{F}}} x_{\tilde{\mathcal{F}}} = \tilde{b} = b - \sum_{i \in \Omega^-} u_i A_i, \tag{38}$$

$$0 \le x_i \le u_i, \;\; i \in \tilde{\mathcal{F}}. \tag{39}$$

Clearly, from the flow balance constraints (38), if a feasible flow $x_{\tilde{\mathcal{F}}}^*$ for the restricted network exists, it defines, along with $x_{\Omega^+}^*$ and $x_{\Omega^-}^*$, a primal feasible solution complementary to y^*. A feasible flow for the restricted network can be determined by solving a maximum flow problem on the *augmented network* defined by underlying graph $\tilde{G} = (\tilde{\mathcal{N}}, \tilde{\mathcal{A}})$, where

$$\tilde{\mathcal{N}} = \{\sigma\} \cup \{\theta\} \cup \mathcal{N}$$

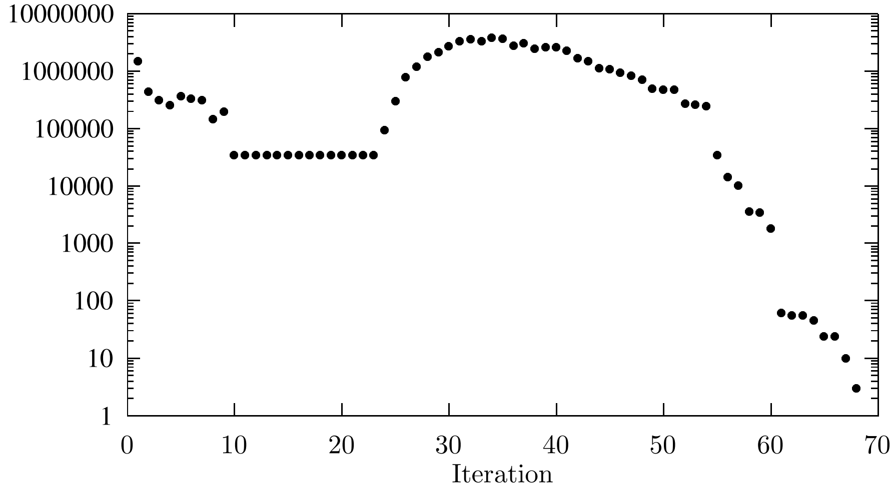

Figure 9: Difference between flow requested and maximum flow found

and

$$\tilde{\mathcal{A}} = \Sigma \cup \Theta \cup \tilde{\mathcal{F}}.$$

In addition, for each arc $(i, j) \in \tilde{\mathcal{F}}$ there is an associated capacity u_{ij}. The additional arcs are such that

$$\Sigma = \{(\sigma, i) \; : \; i \in \mathcal{N}^+\},$$

with associated capacity $\tilde{b}_i$ for each arc (σ, i), and

$$\Theta = \{(i, \theta) \; : \; i \in \mathcal{N}^-\},$$

with associated capacity $-\tilde{b}_i$ for each arc (i, θ), where $\mathcal{N}^+ = \{i \in \mathcal{N} \; : \; \tilde{b}_i > 0\}$ and $\mathcal{N}^- = \{i \in \mathcal{N} \; : \; \tilde{b}_i < 0\}$. It can be shown that if $\mathcal{M}_{\sigma,\theta}$ is the maximum flow value from σ to θ, and $\tilde{x}$ is a maximal flow on the augmented network, then $\mathcal{M}_{\sigma,\theta} = \sum_{i \in \mathcal{N}^+} \tilde{b}_i$ if and only if $\tilde{x}_{\tilde{\mathcal{F}}}$ is a feasible flow for the restricted network. Therefore, finding a feasible flow for the restricted network involves the solution a maximum flow problem. Furthermore, this feasible flow is integer, as we can select a maximum flow algorithm that provides an integer solution.

Figures 9 and 10 show, respectively, the difference between the flow requested and maximum flow found in the maximum flow termination criterion and the number of primal infeasibilities in the spanning tree on the 16384-node, 131072-arc example. On iteration 69, the requested flow of 18720642 units was found and a primal-dual complementary integer solution was recovered. That iteration still had four primal infeasibilities, the smallest number over all interior point iterations.

4 Computational challenge to the network simplex

In this section, we illustrate interior point network flow methods by comparing DLNET [56], an implementation of the dual affine scaling algorithm for network flows with CPLEX NETOPT, the network simplex

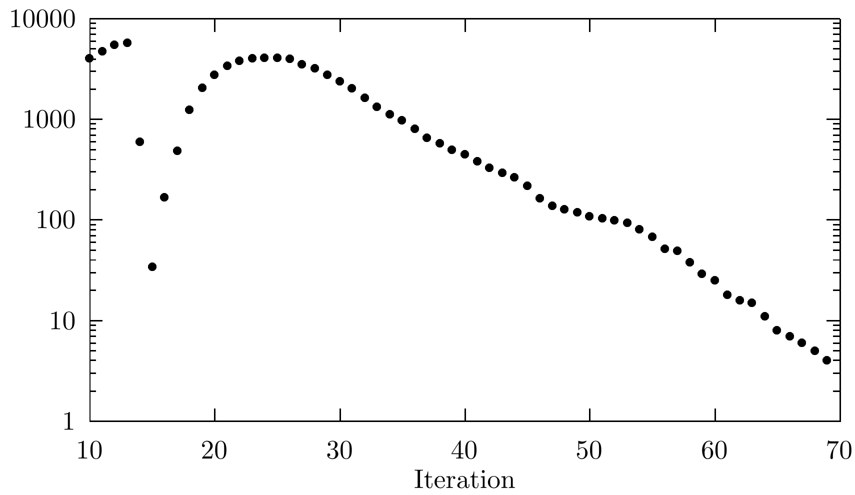

Figure 10: Number of primal infeasibilities in spanning tree

```
16 150 MHZ IP19 Processors
CPU: MIPS R4400 Processor Chip Revision:  5.0
FPU: MIPS R4010 Floating Point Chip Revision:  0.0
Data cache size:  16 Kbytes
Instruction cache size:  16 Kbytes
Secondary unified instruction/data cache size:  1 Mbyte
Main memory size:  1536 Mbytes, 8-way interleaved
```

Figure 11: Hardware configuration (partial output of system command `hinv`)

implementation in the CPLEX mathematical programming system [13]. The experiment was run on an SGI Challenge computer with the configuration summarized in Figure 11.

Version 2.2a of DLNET was used in the experiments. The stepsize parameter γ^k was set to 0.99 for the first 10 dual affine scaling iterations and at each iteration, the relative dual objective function improvement

$$I^k = \frac{(b^\top y^k - u^\top z^k) - (b^\top y^{k-1} - u^\top z^{k-1})}{|b^\top y^k - u^\top z^k|}$$

is computed. After iteration 10, the stepsize is determined as follows:

$$\gamma^k = \begin{cases} 0.95, & \text{if } I^k \geq 0.0001, \\ 0.66, & \text{if } 0.00001 \leq I^k < 0.0001, \\ 0.50, & \text{if } I^k < 0.00001. \end{cases}$$

Indicator 3, described in Section 3.4, with parameters $\kappa_0 = 0.7$ and $\kappa_1 = 0.9$ is used. DLNET uses diagonal and spanning tree preconditioners and the switching heuristic described in Section 3.3. The PB stopping criterion was checked every iteration on which the spanning tree preconditioner was used. The MF criterion was never checked during the first 5 iterations. Once $I^k < 10^{-10}$ the MF was checked every 10 iterations, and when $I^k < 10^{-12}$ it was checked every 5 iterations. The conjugate gradient cosine parameter ε_{cos} was set fixed to 10^{-3} throughout all iterations and the exact cosine rule was checked every 5 conjugate gradient iterations.

Version 3.0 (1994) of CPLEX NETOPT was used with default parameter settings.

All problems considered in this section are from the set of network flow benchmarks collected for the First DIMACS Algorithm Implementation Challenge [31]. The generators can be retrieved from the DIMACS FTP site: `dimacs.rutgers.edu`.

For each problem class, instances of increasing sizes were generated and run on both codes. For each class, the runs are summarized with a table and a figure. The tables list network dimensions, number of interior point iterations, total CPU time for the interior point algorithm, total number of conjugate gradient iterations, number of calls to the maximum flow (MF) stopping criterion, total CPU to run the MF stopping criterion, number of simplex iterations, and total CPU time for the network simplex code. The figures show, in a log-log plot, CPU time ratios between the simplex code and the interior point code.

The first class of problem is Grid-Density. The networks in this class of problems are obtained by removing the minimum number of arcs 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 node with arcs going from the source to all nodes on one side of the resulting tube network and from all nodes on the other side to the sink. Network density is controlled by adding extra arcs in both the horizontal and vertical directions. All nodes other than the source and sink nodes are transshipment nodes. Costs and capacities are uniformly distributed in the intervals $[0, 4096]$ and $[0, 16384]$, respectively.

These instances are generated with the minimum cost network flow test problem generator `goto.c` by Goldberg [16]. Two subclasses of problems are generated: Grid-Density-8 and Grid-Density-16. Let m and n denote the number of nodes and arcs of the network, respectively. For Grid-Density-8, $n = 8m$; and for Grid-Density-16, $n = 16m$. For Grid-Density-8, instances having $256, 1024, \ldots, 65536$ nodes are generated. For Grid-Density-16, instances with $512, 2048, \ldots, 32768$ nodes are generated. The random number generator seed 270001 was used to generate all instances for each problem size.

Table 1: Problem statistics: test problem class Grid-Density-8

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
256	2048	21	1.16	181	0	-	PB	4137	0.36
1024	8192	29	6.46	234	0	-	PB	34172	8.02
4096	32768	32	40.47	376	0	-	PB	184044	139.83
16384	131072	43	419.32	615	0	-	PB	875318	4162.20
65536	524288	65	6336.61	1114	1	255.56	MF	4244165	106956.76

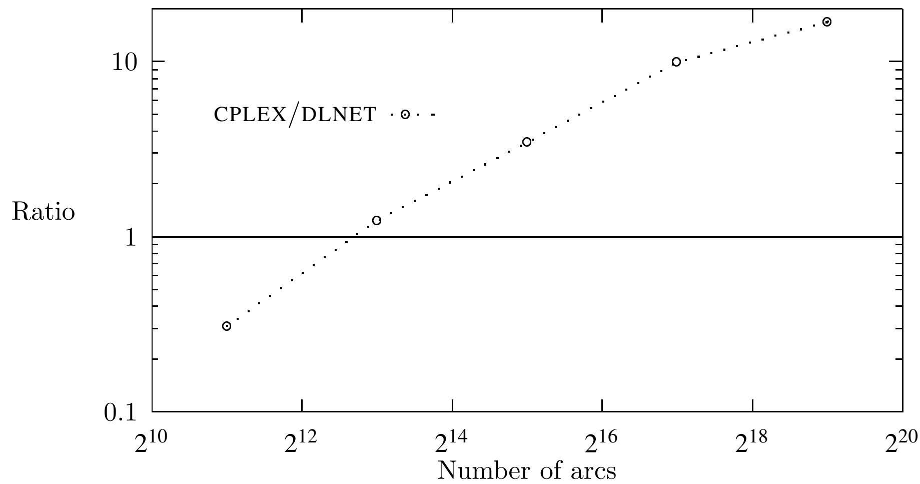

Figure 12: CPU time ratios for problem class Grid-Density-8

Table 2: Problem statistics: test problem class Grid-Density-16

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
512	8192	60	12.30	426	1	0.05	MF	24725	2.59
2048	32768	79	76.45	661	1	0.44	MF	148281	63.34
8192	131072	99	814.52	1052	1	4.06	MF	884825	1673.75
32768	524288	94	3939.72	1223	1	58.57	MF	5399631	46479.68

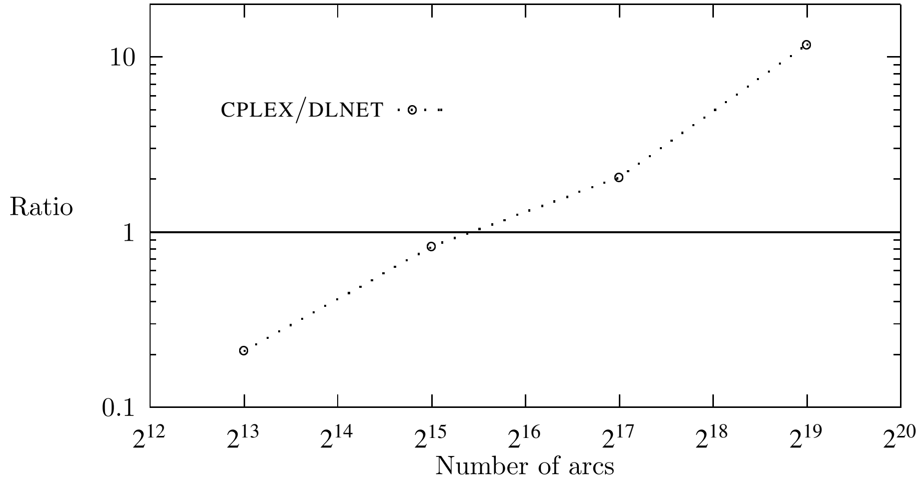

Figure 13: CPU time ratios for problem class Grid-Density-16

Table 1 and Figure 12 summarize the runs for Grid-Density-8. Table 2 and Figure 13 summarize the runs for Grid-Density-16.

The next class is Grid-Graph. The networks in this class are formed on a grid of nodes of height h and width w. Two additional nodes complete the node set: a source node S and a sink node T. Arcs go from the source to each node in the first column of the grid and from each node in the last column of the grid to the sink node. On the grid, arcs go from node to nearest neighbor node oriented left to right and top to bottom. Grid arc costs and capacities are generated uniformly in the interval $[1, 10000]$. Arcs from the source and into the sink have cost zero and are uncapacitated. All nodes, except for source and sink, are transshipment nodes. 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 minimum cost network flow test problem generator `ggraph1.f` of Resende [16]. Two subclasses of problems are generated: Grid-Wide and Grid-Long. Grid-Wide uses parameters $w = 16$ and $h = 32, 64, \ldots$, while Grid-Long uses $h = 16$ and $w = 32, 64, \ldots$.

The random number generator seed 270001 was used to generate the instances. For both Grid-Long and Grid-Wide, instances having $1026, 4098, \ldots, 65538$ nodes were generated. Table 3 and Figure 14 summarize the runs for Grid-Long. Table 4 and Figure 15 summarize the runs for Grid-Wide.

Most computational studies of network optimization codes in the past have used the minimum cost network flow test generator NETGEN [40] of Klingman, Napier and Stutz. We tested the codes on two classes of networks generated with NETGEN: Netgen-Lo and Netgen-Hi. Figure 16 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 subclasses, an instance of each size was generated. Sizes correspond to the settings $x = 8, 9, \ldots, 16$. Table 5 and Figure 17 summarize the runs for Netgen-Hi. Table 6 and Figure 18 summarize the runs for Netgen-Lo.

The last class of problems considered is Mesh-4. Networks in this class are formed on a grid of nodes

Table 3: Problem statistics: test problem class Grid-Long

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
1026	2000	29	2.56	402	0	-	PB	2171	0.29
4098	7952	73	29.24	1112	1	0.33	MF	7776	4.59
16386	31760	84	284.88	2517	0	-	PB	29729	85.77
65538	126992	169	4925.54	4228	0	-	PB	92599	1471.92

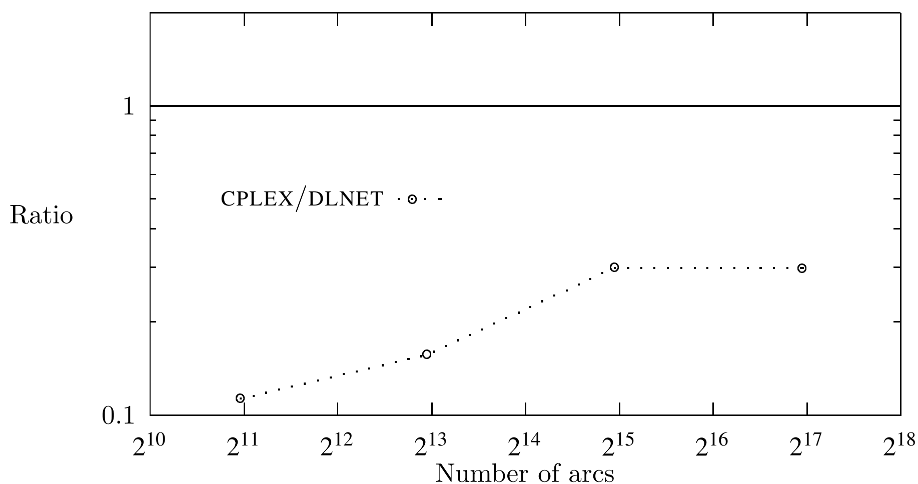

Figure 14: CPU time ratios for problem class Grid-Long

Table 4: Problem statistics: test problem class Grid-Wide

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
1026	2096	30	2.48	391	0	-	PB	2449	0.23
4098	8432	38	13.90	536	0	-	PB	10685	1.28
16386	33776	49	77.05	458	0	-	PB	37044	8.63
65538	135152	74	702.05	377	0	-	PB	140428	55.84

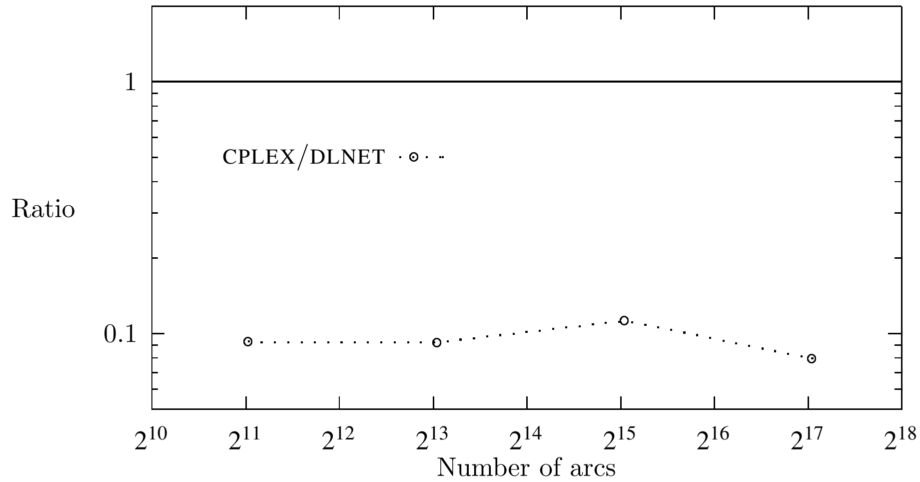

Figure 15: CPU time ratios for problem class Grid-Wide

`seed`	Random number seed:	270001
`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

Figure 16: NETGEN specification file for Netgen-Lo

Table 5: Problem statistics: test problem class Netgen-Hi

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
256	2048	29	1.55	255	0	-	PB	1033	0.18
1024	8214	41	10.43	537	0	-	PB	6679	1.22
4096	32877	61	82.97	989	0	-	PB	29330	13.77
16384	131409	74	835.42	2012	0	-	PB	137023	248.34
65536	525803	107	14861.27	5153	1	162.33	MF	1077281	21046.19

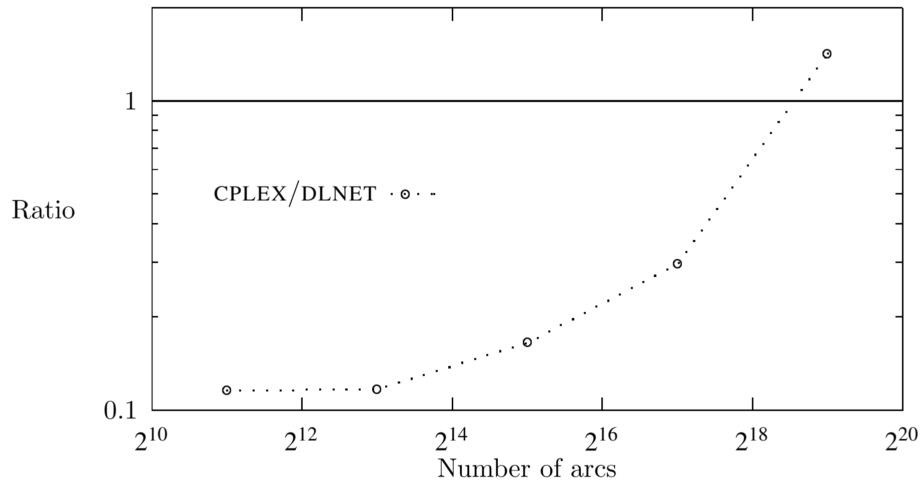

Figure 17: CPU time ratios for problem class Netgen-Hi

Table 6: Problem statistics: test problem class Netgen-Lo

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
256	2048	19	1.11	233	0	-	PB	2876	0.24
1024	8214	36	9.72	532	0	-	PB	16956	2.08
4096	32858	42	60.75	644	0	-	PB	63856	19.48
16384	131409	73	584.65	1220	1	10.54	MF	218163	231.59
65536	525803	102	5703.28	1685	1	190.26	MF	1212399	9459.77

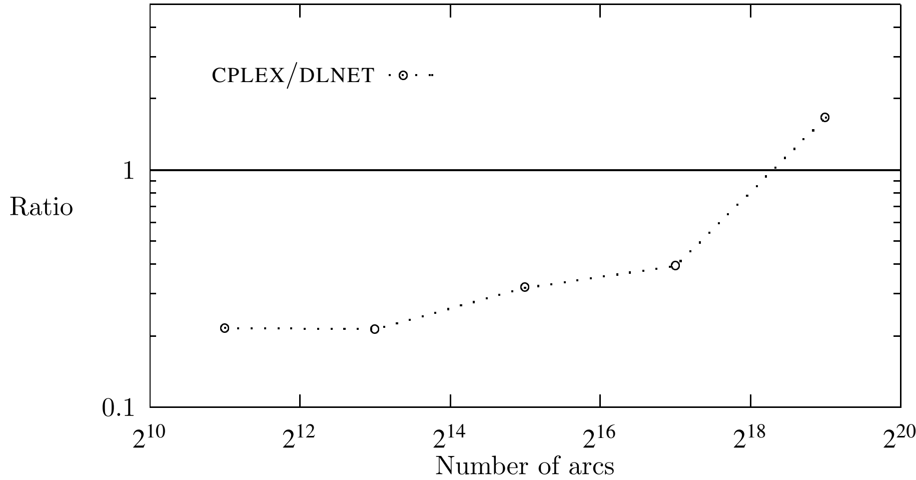

Figure 18: CPU time ratios for problem class Netgen-Lo

Table 7: Problem statistics: test problem class Mesh-4

network size		DLNET						CPLEX	
		int point alg		CG	max flow		stopping		
$\|\mathcal{N}\|$	$\|\mathcal{A}\|$	itrs	time	itrs	calls	time	criterion	itr	time
256	2048	21	1.18	245	0	-	PB	4790	0.36
1024	8192	56	11.67	475	1	0.10	MF	25459	4.13
4096	32768	88	90.02	779	1	0.68	MF	125315	68.34
16384	131072	60	520.57	662	1	7.82	MF	712187	1406.08
65536	524288	83	5972.02	1035	1	104.39	MF	3231833	28801.18

embedded on a torus. Arcs connect nodes in the same row or column of the grid. All horizontal arcs have the same orientation. Similarly, all vertical arcs are oriented in the same direction. Each node has arcs going to its 8 nearest neighbors (four in the horizontal direction, the other four in the vertical direction). Costs are generated uniformly in the interval $[-1000, 1000]$. Capacities are generated in the interval $[-1000, 1000]$ with a bias that makes longer arcs have smaller capacities.

The instances are generated with the minimum cost network flow test generator `mesh.c` by Goldberg [16]. All node sets are $h \times w$ grids. We generated all instances with $h = w$. Instances are generated with parameters $h = w = 16, 32, \ldots, 256$, having $256, 1024, \ldots, 65536$ nodes. The random number generator seed 270001 was used to generate the instances.

Table 7 and Figure 19 summarize the runs for Mesh-4.

We conclude this section with some remarks:

- Both codes found optimal solutions to all the instances tested.

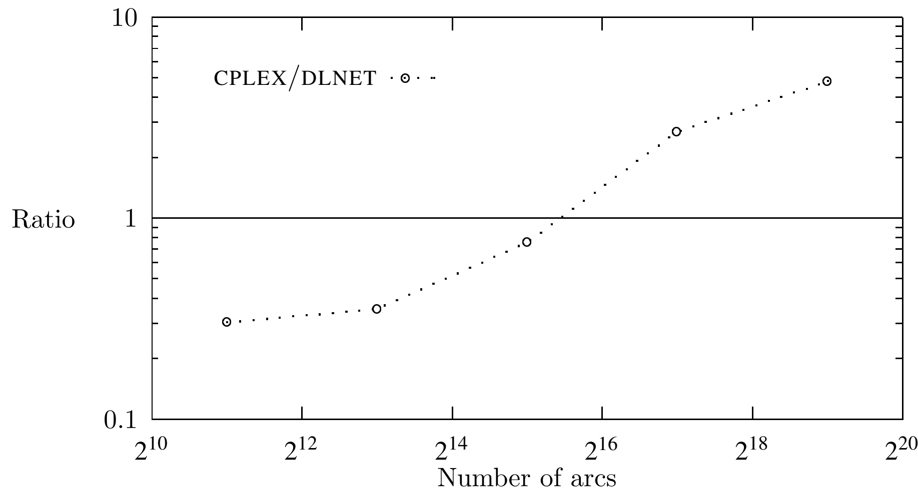

Figure 19: CPU time ratios for problem class Mesh-4

- Both codes produced integer flows as output.
- Of the 32 instances optimized, the interior point code stopped with the maximum flow (MF) criterion on 13 instances and with the primal basic (PB) criterion on 19 instances.
- On all 32 instances, the interior point code found optimal primal-dual complementary solutions.
- The triggering mechanism for the maximum flow stopping criterion, based on relative dual improvement, worked well, since on every instance MF was never called more than once.
- The preconditioners used resulted in small average number of conjugate gradient iterations. For the largest instance in each problem class, the average number of conjugate gradient iterations per interior point iteration was 17.1, 13.0, 25.0, 48.2, 16.5, and 12.5.
- On all but one of the problem classes, we observed an improvement in the performance of the interior point code with respect to the simplex code, as the sizes of the instances increase. On Grid-Wide, the class in which no improvement was observed, it is unclear which code is asymptotically superior.
- In all but two problem classes, the interior point code was faster than the simplex code on the largest instances tested. Interior point to simplex time ratios were 16.9, 11.8, 0.3, 0.1, 1.4, 1.7, and 4.8, for Grid-Density-8, Grid-Density-16, Grid-Long, Grid-Wide, Netgen-Hi, Netgen-Lo, and Mesh-4, respectively.
- Though the hardware used in these experiments is fast by today's standards, it was not specifically designed for floating point computations. A new processor recently introduced by SGI, the MIPS R8000, is about three times as fast as the R4400 processor (used in this study) on floating point computations. Because DLENT does a large amount of floating point computations, it is expected to benefit from the new hardware, while NETOPT, which does mainly integer arithmetic is not expected to benefit at all.

- Parallel implementation of the conjugate gradient algorithm has been shown to improve the performance of conjugate gradient based interior point codes more than parallel implementations speedup the simplex method. One should expect larger time ratios in favor of interior point codes on parallel architectures.

5 Concluding remarks

In this chapter, we discussed different issues related to the implementation of interior point methods for solving linear network flow problems. These methods provide, in addition to the plethora of existing solution techniques, new ways to solve large scale network flow problems. Though interior point methods have not, so far, been shown to outperform all other methods in several classes of large scale problems, there exist many classes where interior point codes are currently the most efficient computational solution approaches. In this way, interior point methods should not be thought of as a substitute for existing algorithms, but rather they are an addition to the class of computationally efficient network flow algorithms. It is expected that in the near future robust interior point codes for solving network flow problems will be widely available.

In addition to the codes developed for the single commodity network flow problem, initial attempts to use interior point methods to solve the more difficult multicommodity network flow problem can be found in [24, 35].

References

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

[2] I. Adler, N. Karmarkar, M.G.C. Resende, and G. Veiga. An implementation of Karmarkar's algorithm for linear programming. *Mathematical Programming*, 44:297–335, 1989.

[3] Navindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice Hall, Englewood Cliffs, NJ, 1993.

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

[5] 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. Technical Report 85-OR-3, Department of Operations Research, Southern Methodist University, Dallas, TX, August 1985.

[6] E.R. Barnes. A variation on Karmarkar's algorithm for solving linear programming problems. *Mathematical Programming*, 36:174–182, 1986.

[7] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali. *Linear Programming and Network Flows*. Wiley, New York, NY, 1990.

[8] D. Bertsekas. *Linear network optimization: Algorithms and codes*. MIT Press, Cambridge, MA, 1991.

[9] D.P. Bertsekas and D.A. Castañon. The auction algorithm for transportation problems. *Annals of Operations Research*, 20:67–96, 1989.

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

[11] G.H. Bradley, G.G. Brown, and G.W. Graves. Design and implementation of large scale primal transshipment algorithms. *Management Science*, 24:1–34, 1977.

[12] Y. C. Cheng, D. J. Houck, J. M. Liu, M. S. Meketon, L. Slutsman, R. J. Vanderbei, and P. Wang. The AT&T KORBX System. *AT&T Technical Journal*, 68:7–19, 1989.

[13] CPLEX Optimization, Inc. *Using the CPLEX™callable library and CPLEX™mixed integer library including the CPLEX™linear optimizer and CPLEX™mixed integer optimizer – Version 3.0*. Incline Village, NV, 1994.

[14] G.B. Dantzig. Application of the simplex method to a transportation problem. In T.C. Koopsmans, editor, *Activity Analysis of Production and Allocation*. John Wiley and Sons, 1951.

[15] I.I. Dikin. Iterative solution of problems of linear and quadratic programming. *Soviet Mathematics Doklady*, 8:674–675, 1967.

[16] DIMACS. The first DIMACS international algorithm implementation challenge: The benchmark experiments. Technical report, DIMACS, New Brunswick, NJ, 1991.

[17] Ding-Zhu Du and Panos M. Pardalos, editors. *Network Optimzation Problems: Algorithms, Applications and Complexity*. World Scientific, 1993.

[18] A.S. El-Bakry, R.A. Tapia, and Y. Zhang. A study of indicators for identifying zero variables in interior-point methods. Technical Report TR91-15, Department of Mathematical Sciences, Rice University, Houston, TX 77251, 1991.

[19] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1990.

[20] D. Gay. Electronic mail distribution of linear programming test problems. *COAL Newsletter*, 13:10–12, 1985.

[21] D.M. Gay. Stopping tests that compute optimal solutions for interior-point linear programming algorithms. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1989.

[22] A. George and M. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and Its Applications*, 34:69–83, 1980.

[23] F. Glover and D. Klingman. The simplex SON algorithm for LP/embedded network problems. *Mathematical Programming Study*, 15:148–176, 1981.

[24] J.-L. Goffin, J. Gondzio, R. Sarkissian, and J.-P. Vial. Solving nonlinear multicommodity flow problems by the analytic center cutting plane method. Technical report, GERARD, McGill University, Montreal, 1994.

[25] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. Technical report, Department of Computer Science, Stanford University, Stanford, CA 94305, 1992.

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

[27] M.D. Grigoriadis. An efficient implementation of the network simplex method. *Mathematical Programming Study*, 26:83–111, 1986.

[28] G.M. Guisewite. Network problems. In Reiner Horst and Panos M. Pardalos, editors, *Handbook of global optimization*. Kluwer Academic Publishers, 1995.

[29] G.M. Guisewite and P.M. Pardalos. Minimum concave cost network flow problems: Applications, complexity, and algorithms. *Annals of Operations Research*, 25:75–100, 1990.

[30] F.L. Hitchcock. The distribution of product from several sources to numerous facilities. *Journal of Mathematical Physics*, 20:224–230, 1941.

[31] D.S. Johnson and C.C. McGeoch, editors. *Network flows and matching: First DIMACS implementation challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1993.

[32] A. Joshi, A.S. Goldstein, and P.M. Vaidya. A fast implementation of a path-following algorithm for maximizing a linear function over a network polytope. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 267–298. American Mathematical Society, 1993.

[33] J.A. Kaliski and Y. Ye. A decomposition variant of the potential reduction algorithm for linear programming. *Management Science*, 39:757–776, 1993.

[34] Anil Kamath and Omri Palmon. Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. Technical report, Department of Computer Science, Stanford University, Stanford, CA 94305, 1994.

[35] A.P. Kamath, N.K. Karmarkar, and K.G. Ramakrishnan. Computational and complexity results for an interior point algorithm for multicommodity flow problems. Technical Report TR-21/93, Dipartimento di Informatica, Università di Pisa, 1993. Extended abstracts of *NETFLOW'93*.

[36] S. Kapoor and P.M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proceedings of 18th Annual ACM Symposium on the Theory of Computing*, pages 147–159, 1986.

[37] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[38] N.K. Karmarkar and K.G. Ramakrishnan. Computational results of an interior point algorithm for large scale linear programming. *Mathematical Programming*, 52:555–586, 1991.

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

[40] 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*, 20:814–821, 1974.

[41] J.B. Kruskal. On the shortest spanning tree of graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[42] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[43] J.M. Mulvey. Testing of a large scale network optimization program. *Mathematical Programming*, 15:291–315, 1978.

[44] B.A. Murtagh and M.A. Saunders. Minos 5.1 user's guide. Technical Report SOL 83-20R, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, CA 94305, 1983. revised 1987.

[45] J.B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of 20th Annual ACM Symposium on the Theory of Computing*, pages 377–387, 1988.

[46] C.C. Paige and M.A. Saunders. Algorithm 583 – LSQR: Sparse linear equations and least square problems. *ACM Transactions on Mathematical Software*, 8:195–209, 1982.

[47] P.M. Pardalos and K.G. Ramakrishnan. On the expected optimal value of random assignment problems: Experimental results and open questions. *Computational Optimization and Applications*, 2:261–271, 1993.

[48] P.M. Pardalos and H. Wolkowicz, editors. *Quadratic assignment and related problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1994.

[49] L. Portugal, F. Bastos, J. Júdice, J. Paixão, and T. Terlaky. An investigation of interior point algorithms for the linear transportation problem. Technical report, Department of Mathematics, University of Coimbra, Coimbra, Portugal, 1993. To appear in *SIAM J. Sci. Computing*.

[50] L. Portugal, M.G.C. Resende, G. Veiga, and J. Júdice. A truncated primal-infeasible dual-feasible network interior point method. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1994.

[51] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

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

[53] K.G. Ramakrishnan, N.K. Karmarkar, and A.P. Kamath. An approximate dual projective algorithm for solving assignment problems. In David S. Johnson and Catherine C. McGeoch, editors, *Network*

Flows and Matching: First DIMACS Implementation Challenge, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 431–451. American Mathematical Society, 1993.

[54] M.G.C. Resende, T. Tsuchiya, and G. Veiga. Identifying the optimal face of a network linear program with a globally convergent interior point method. In W.W. Hager, D.W. Hearn, and P.M. Pardalos, editors, *Large scale optimization: State of the art*, pages 362–387. Kluwer Academic Publishers, 1994.

[55] M.G.C. Resende and G. Veiga. Computing the projection in an interior point algorithm: An experimental comparison. *Investigación Operativa*, 3:81–92, 1993.

[56] M.G.C. Resende and G. Veiga. An efficient implementation of a network interior point method. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 299–348. American Mathematical Society, 1993.

[57] M.G.C. Resende and G. Veiga. An implementation of the dual affine scaling algorithm for minimum cost flow on bipartite uncapaciated networks. *SIAM Journal on Optimization*, 3:516–537, 1993.

[58] Günther Ruhe. *Algorithmic Aspects of Flows in Networks*. Kluwer Academic Publishers, Boston, MA, 1991.

[59] B. Simeone, P. Toth, G. Gallo, F. Maffioli, and S. Pallottino, editors. *Fortran codes for network optimization*, volume 13 of *Annals of Operations Research*. J.C. Baltzer, Basel, Switzerland, 1988.

[60] L.P. Sinha, B.A. Freedman, N.K. Karmarkar, A. Putcha, and K.G. Ramakrishnan. Overseas network planning. In *Proceedings of the Third International Network Planning Symposium – NETWORKS'86*, pages 8.2.1–8.2.4, June 1986.

[61] É. Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34:250–256, 1986.

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

[63] P.M. Vaidya. Speeding up linear programming using fast matrix multiplication. In *Proceedings of 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 332–337, 1989.

[64] R.J. Vanderbei, M.S. Meketon, and B.A. Freedman. A modification of Karmarkar's linear programming algorithm. *Algorithmica*, 1:395–407, 1986.

[65] Stephen A. Vavasis and Yinyu Ye. Accelerated interior point method whose running time depends only of A (extended abstract). In *Proceedings of 26th Annual ACM Symposium on the Theory of Computing*, pages 512–521, 1994.

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