MCF

— Version 1.2 (January 7, 2000) —

A network simplex implementation.

Andreas Löbel

Contents

1 Introduction.

This is the documentation of **MCF**. MCF is an implementation of a primal and a dual network simplex algorithm. The documentation gives a description of the minimum cost flow problem, the used data structures, and the library interface. How the functions of the library can be used is briefly shown in the file mcflight.c or, in details, in main.c. Please notify the ZIB Academic License conditions at the end of this documentation.

Performance.
MCF has been tested with several classes of artificially generated NETGEN problems and with real-world problems arising from vehicle scheduling (e. g., see Löbel [1998]) and telecommunication problems (e. g., see Eisenblätter [1998] and Borndörfer, Eisenblätter, Grötschel, and Martin [1997]). Our computational experiments have always shown a good polynomial behaviour of our code. Even truly large-scale real-world instances with several thousand nodes and several million arcs can be solved quickly. The code was checked with *Purify* making this implementation quite reliable and robust, for more information see `http://www.rational.com`.

Commercial Applications.
MCF is used commercially for vehicle scheduling in public transit, e. g., in the MICROBUS system of the IVU AG, Berlin, for information see `http://www.ivu.de`. Moreover, a simplified vehicle scheduling solver, employing MCF as the workhorse, has become a CINT2000 integer benchmark of the SPEC CPU2000 Benchmark suite, for more information see `http://www.spec.org`.

Literature.
The network simplex algorithm with upper bound technique is a specialised revised simplex algorithm, see Dantzig [1963] or Chvátal [1980], that exploits the structure of network flow problems. The linear algebra of the simplex algorithm is replaced by simple network operations. Helgason and Kennington [1995] and Ahuja, Magnanti, and Orlin [1989,1993] describe the (primal) network simplex algorithm and give pseudocodes, implementation hints, etc. Veldhorst [1993] compiled a bibliography containing 370 references to network flow papers published by 1993.

Problem definition.
Given a connected digraph $D = (V, A)$, a linear cost function $c \in \mathbb{Q}^A$, lower and upper bounds $l \in \mathbb{Q}$ and $u \in \mathbb{Q}$ such that $l \leqslant u$, and **node imbalances** $b \in \mathbb{Q}$ such that $\mathbb{1}^{\mathrm{T}} b = 0$. Unbounded lower or upper bounds can be defined, but are simulated in MCF by sufficiently small and large values. A node i is called a **supply** node, a **demand** node, or a **transshipment** node depending upon whether b_i is larger than, smaller than, or equal to zero, respectively. The minimum-cost flow problem is to find a flow vector $x^* \in \mathbb{Q}^A$ such that x^* is an optimal solution of the linear program

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

subject to

$$\begin{aligned}\sum_{(i,j)\in A} x_{ij} \quad - \sum_{(j,i)\in A} x_{ji} \quad &= \quad b_i, \qquad \forall\, i \in V, &\text{(1b)}\\ l_{ij} \quad \leqslant \quad x_{ij} \quad &\leqslant \quad u_{ij}, \qquad \forall\,(i,j) \in A. &\text{(1c)}\end{aligned}$$

The equations (1b) are the so-called **flow conservation constraints** and the inequalities (1c) are the **flow capacities** on x. A flow x is called a **feasible flow** if it satisfies (1b) and (1c). Let $\mathcal{N}$ denote the node-arc incidence matrix of D. In matrix notation, (1) reads

$$\min\{c^{\mathrm{T}}x \mid \mathcal{N}x = b,\ l \leqslant x \leqslant u\}. \tag{2}$$

It is well known that $\mathcal{N}$ and, thus, the constraint matrix of (2) are totally unimodular. For integer vectors l, u, and b, there exists always an integer optimal flow (see Grötschel, Lovász, and Schrijver [1988]).

Let $\pi \in \mathbb{Q}^V$ (the so-called **node potentials**), $\lambda \in \mathbb{Q}^A$, and $\eta \in \mathbb{Q}^A$ be the dual multipliers for the flow conservation constraints and the lower and upper bounds. The dual problem of (2) is

$$\max\{\pi^{\mathrm{T}}b + \lambda^{\mathrm{T}}l - \eta^{\mathrm{T}}u \mid \pi^{\mathrm{T}}\mathcal{N} + \lambda^{\mathrm{T}} - \eta^{\mathrm{T}} \leqslant c^{\mathrm{T}},\ \eta \geqslant 0,\ \lambda \geqslant 0\}, \tag{3}$$

Although it is possible to use arbitrary lower bounds, we strongly recommend to use the faster version of MCF with lower bounds fixed to zero. Each nonzero lower bound can easily be transformed to 0 by substituting the flow vector x by $x' + l$, $x' \in \mathbb{Q}^A$. Thus, the system $l \leqslant x \leqslant u$ transforms to $0 \leqslant x' \leqslant u - l$. The system $\mathcal{N}x = b$ transforms to $\mathcal{N}x' = b - \mathcal{N}l$, which is equivalent to decrease b_i and to increase b_j by l_{ij} for all $(i,j) \in A$. The objective $c^{\mathrm{T}}x$ transforms to $c^{\mathrm{T}}l + \min c^{\mathrm{T}}x'$. Figure 1, which is taken from Ahuja, Magnanti, and Orlin [1993], displays such a lower bound transformation. Note, there is currently no support to transform nonzero lower bounds to zero, you have to do it by yourself!

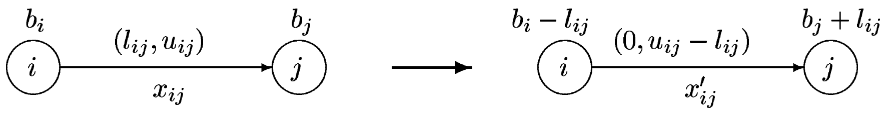


Figure 1: Transformation to zero lower bounds.

To apply the network simplex algorithm, we need a full rank constraint matrix. For a connected network D, the rank of the flow conservation constraints is equal to $|V| - 1$, and the flow conservation constraint for one node, the so-called **root node**, can be eliminated. We will assume that we have chosen a root node and have eliminated its flow conservation constraint, i.e., the reduced node-arc incidence matrix has full rank. For notational simplicity, we also denote the reduced node-arc incidence matrix by $\mathcal{N}$. It is well known that every nonsingular basis matrix B of $\mathcal{N}$ corresponds to a spanning tree of A in D and vice versa.

Let $T \subseteq A$ be a spanning tree in D. The variables x_{ij}, $(i,j) \in T$, are called the **basic variables** corresponding to the basis matrix $B := \mathcal{N}_{\cdot,T}$. Let L and U denote the arcs that correspond to the **nonbasic variables** whose values are set to their lower and upper bound, respectively. The triple (T, L, U) is called a **basis structure**. For given nonbasic arc sets L and U, the right hand side b transforms to

$$b' := b - \sum_{(i,j)\in U} \mathcal{N}_{\cdot,ij}u_{ij} - \sum_{(i,j)\in L} \mathcal{N}_{\cdot,ij}l_{ij}.$$

The associated basic solution is the solution of the system $Bx_T = b'$, the values of the node potentials are determined by the system $\pi^{\mathrm{T}} B = c_T^{\mathrm{T}}$. Let $\overline{c}_{ij} := c_{ij} - \pi_i + \pi_j$ denote the **reduced costs** of an arc $(i, j) \in A$. The dual multipliers λ and η are determined by

$$\lambda_{ij} := \begin{cases} \overline{c}_{ij} & \text{if } (i,j) \in L, \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

$$\eta_{ij} := \begin{cases} -\overline{c}_{ij} & \text{if } (i,j) \in U, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

A basis structure (T, L, U) is called **primal feasible** if the associated basic solution x satisfies the flow bounds (1 c) and is called **dual feasible** if for all $(i, j) \in A$:

$$\overline{c}_{ij} > 0 \quad \Rightarrow \quad (i,j) \in L, \tag{6}$$
$$\overline{c}_{ij} < 0 \quad \Rightarrow \quad (i,j) \in U, \tag{7}$$

A basis structure is called **optimal** if it is both primal and dual feasible.

Input and Output Format.
Problems and their solutions are expected to be in DIMACS format, which is for our purposes described in the MCF interface. Further information including network generators and minimum-cost flow codes can be received via anonymous ftp from dimacs.rutgers.edu in the directory /pub/netflow/general-info, see also DIMACS [1990], DIMACS [1993], and Johnson and McGeoch [1993].

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1989). *Network Flows.* In Nemhauser, Rinnooy Kan, and Todd [1989], chapter IV, pages 211–369.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L., editors (1995). *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science.* Elsevier Science B.V., Amsterdam.

Borndörfer, R., Eisenblätter, A., Grötschel, M., and Martin, A. (1997). Frequency assignment in cellular phone networks. ZIB Preprint SC 97-35, available at www.zib.de.

Chvátal, V. (1980). *Linear programming.* W. H. Freeman and Company, New York.

Dantzig, G. B. (1963). *Linear Programming and Extensions.* Princeton University Press, Princeton.

DIMACS (1990). The first DIMACS international algorithm implementation challenge: Problem definitions and specifications. Available via WWW at URL `ftp://dimacs.rutgers.edu/pub/netflow/general-info`.

DIMACS (1993). The first DIMACS international algorithm implementation challenge. Available via WWW at URL `ftp://dimacs.rutgers.edu/pub/netflow`.

Du, D.-Z. and Pardalos, P. M., editors (1993). *Network Optimization Problems: Algorithms, Applications and Complexity*, volume 2 of *Series on Applied Mathematics*, Singapore, New York, London. World Scientific Publishing Co. Pte. Ltd.

Eisenblätter, A. (1998). A frequency assignment problem in cellular phone networks. In Du, D. and Pardalos, P. M., editors, *DIMACS series in discrete mathematics and theoretical computer science,*, volume 40, pages 109–115. American Mathematical Society, Providence, RI. Available as ZIB Preprint SC 97-27 at www.zib.de

Grötschel, M., Lovász, L., and Schrijver, A. (1988). *Geometric algorithms and combinatorial optimization.* Springer Verlag, Berlin.

Helgason, R. V. and Kennington, J. L. (1995). *Primal Simplex Algorithms for Minimum Cost Network Flows.* In Ball, Magnanti, Monma, and Nemhauser [1995], chapter 2, pages 85–133.

Johnson, D. S. and McGeoch, C. C., editors (1993). *Network Flows and Matching*, volume 12 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society.

Löbel, A. (1998). *Optimal Vehicle Scheduling in Public Transit.* PhD thesis, Technische Universität Berlin, November 1997. Shaker Verlag, 1998. Available via WWW at URL `ftp://ftp.zib.de/pub/zib-publications/books/Loebel.disser.ps`, but observe the publisher's copyright restrictions.

Nemhauser, G. L., Rinnooy Kan, A. H. G., and Todd, M. J., editors (1989). *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science.* Elsevier Science B.V., Amsterdam.

Veldhorst, M. (1993). A bibliography on network flow problems. In Du and Pardalos [1993], pages 301–331.

2 Changes.

Version 1.2:

- Minor bug fix in main.c.
- Correct handling of time.h, sys/time.h, and sys/times.h.
- Better dependencies handling on unix/linux systems (requires GNU make).
- Removed prototyp.h and made function prototyping as standard.
- Added prefix MCF_ to all names and identifiers.
- Windows support for a Microsoft Visual C++ 6.0 environment.

Version 1.1:

- MCF is now stable for 64-bit architectures.
- The objective value is now be computed correctly by mcflight.
- Fixed arc values are handled correctly.
- Windows support for a Microsoft Visual C++ 5.0 environment

3

MCF data structures.

Names

	typedef long	**MCF_flow_t**	*Default flow type definition*	
	typedef long*	**MCF_flow_p**	*Default flow pointer definition*	
	typedef long	**MCF_cost_t**	*Default cost type definition*	
	typedef long*	**MCF_cost_p**	*Default cost pointer type definition*	
	typedef double	**MCF_flow_t**	*Flow type definition if MCF_FLOAT is defined*	
	typedef double*	**MCF_flow_p**	*Flow pointer definition if MCF_FLOAT is defined*	
	typedef double	**MCF_cost_t**	*Cost type definition if MCF_FLOAT is defined*	
	typedef double*	**MCF_cost_p**	*Cost pointer definition if MCF_FLOAT is defined*	
	typedef struct MCF_node	**MCF_node_t**	*Node type definition*	
	typedef struct MCF_node*	**MCF_node_p**	*Node pointer definition*	
	typedef struct MCF_arc	**MCF_arc_t**	*Arc type definition*	
	typedef struct MCF_arc*	**MCF_arc_p**	*Arc pointer definition*	
	typedef struct MCF_network	**MCF_network_t**	*Network type definition*	
	typedef struct MCF_network*	**MCF_network_p**	*Network pointer definition*	
3.1	struct	**MCF_node**	*Node description*	10
3.2	struct	**MCF_arc**	*Arc description*	13
3.3	struct	**MCF_network**	*Network description*	16

In the following, we give a description of the variable types and the data structures of MCF, which are defined in the file "mcfdefs.h". For costs and flows, it is possible either to use the faster integer arithmetic restricted to (4-byte) integers or to use floating point arithmetic with double precision.

For the network simplex algorithm, the input network is assumed to be connected, which is ensured by the following simple procedue: Having red a problem from file, we add to V one artificial root node, denoted by "0". Each original node i of V is then connected to the root node 0 either by the artificially generated arc $(i, 0)$ if i is a supply or transshipment node or by the artificially generated arc $(0, i)$ if i is a demand node.

Node, arc, and network information are stored in the following data structures.

3.1 struct **MCF_node**

Node description

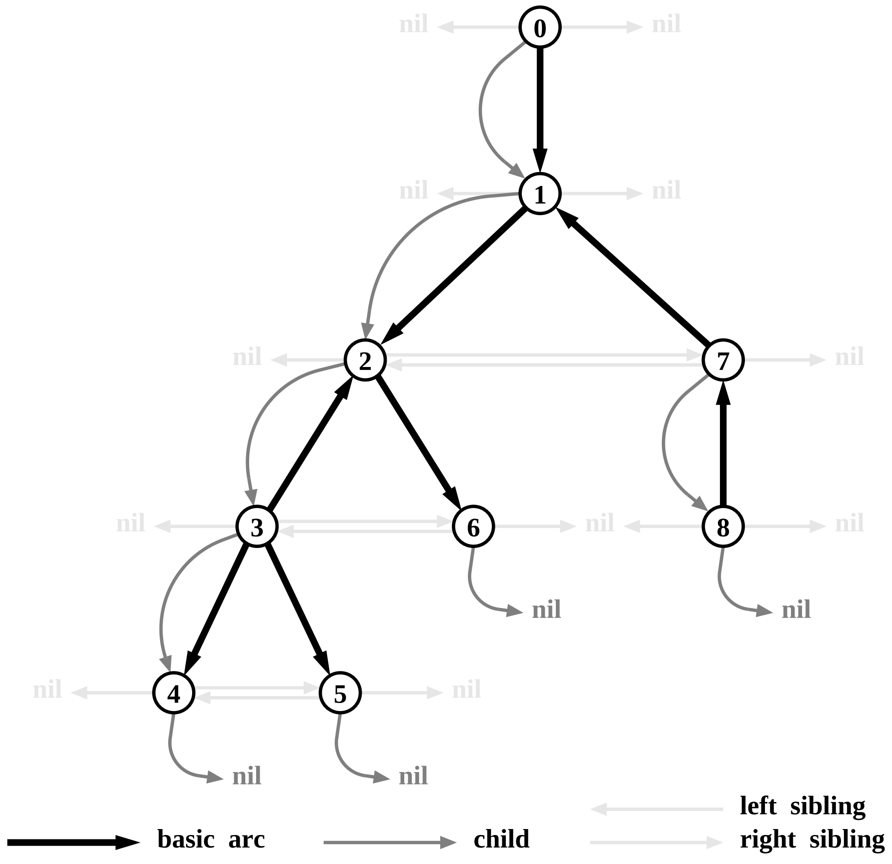
Members

3.1.1	long	**number**	*Node identifier*	12
	MCF_node_p	**pred**	*predecessor node*	
	MCF_node_p	**child**	*First child node*	
	MCF_node_p	**right_sibling**	*Next child of predecessor*	
	MCF_node_p	**left_sibling**	*Previous child of predecessor*	
	long	**subtreesize**	*Number of nodes (including this one) up to the root node*	
	MCF_arc_p	**basic_arc**	*The node's basic arc*	
3.1.2	long	**orientation**	*Orientation of the node's basic arc*	12
	MCF_arc_p	**firstout**	*First arc of the neighbour list of arcs leaving this node*	
	MCF_arc_p	**firstin**	*First arc of the neighbour list of arcs entering this node*	
3.1.3	MCF_flow_t	**balance**	*Supply/Demand b_i of this node*	12
3.1.4	MCF_cost_t	**potential**	*Dual node multipliers*	13
	MCF_flow_t	**flow**	*Flow value of the node's basic arc*	
3.1.5	long	**mark**	*Temporary variable*	13

Node description.

Let $T \subseteq A$ be a spanning tree in D, and consider some node $v \in V \setminus \{0\}$. There is an unique (undirected) path, denoted by $P(v)$, defined by T from v to the root node 0. The arc in $P(v)$, which is incident to v, is called the **basic arc** of v. The other terminal node u of this basic arc is called the **predecessor** (**node**) of v. The basic arc of v is called **upward** (**downward**) oriented if v is the tail (head) node of its basic arc. If v is the predecessor of some other node u, we call u a **child** (**node**) of v. Given some order of all childs of v, and let u and w be two different childs of v. If u is smaller than w with respect to the given order, we call u the **left sibling** of w and w the **right sibling** of u. If there is no child u being smaller (greater) than a given child w, then w has no left (right) sibling. Each node has at most one child reference, the other children of a node can be reached by traversing the sibling links. The number of nodes in $P(V)$ is called the **subtree size** of v.

The subtree size and predecessor variables are used by the ratio test. The orientation, child, and sibling variables are used for the computation of the node potentials. Figure 2 shows a small example of a rooted basis tree for our data structures (the underlying network is a copy from Ahuja, Magnanti, and Orlin [1993]).



node	0	1	2	3	4	5	6	7	8
subtree size	9	8	5	2	1	1	1	2	1
predecessor	nil	0	1	2	3	3	2	1	7
child	1	2	3	4	nil	nil	nil	8	nil
right sibling	nil	nil	7	6	5	nil	nil	nil	nil
left sibling	nil	nil	nil	nil	nil	4	3	2	nil
orientation	-	down	down	up	down	down	down	up	up

Figure 2: Rooted basis tree.

3.1.1

long **number**

Node identifier

Node identifier.

This variable is only used to assign some identification to each node. Typically, as for the DIMACS format, nodes are indexed from 1 to n, where n denotes the number of nodes.

3.1.2

long **orientation**

Orientation of the node's basic arc

Orientation of the node's basic arc.

This variable stands for the orientation of the node's basic arc. The value UP (= 1) means that the arc points to the father, and the value DOWN (= 0) means that the arc points from the father to this node.

3.1.3

MCF_flow_t **balance**

Supply/Demand b_i of this node

Supply/Demand b_i of this node.

A node i is called a supply node, a demand node, or a transshipment node depending upon whether b_i is larger than, smaller than, or equal to zero, respectively.

3.1.4

MCF_cost_t **potential**

Dual node multipliers

Dual node multipliers.

This variable stands for the node potential corresponding with the flow conservation constrait of this node.

3.1.5

long **mark**

Temporary variable

Temporary variable.

This is a temporary variable, which you can use as you like.

3.2

struct **MCF_arc**

Arc description

Members

	MCF_node_p	**tail**	*Tail node*	
	MCF_node_p	**head**	*Head node*	
	MCF_arc_p	**nextout**	*Next arc of the neighbour list of arcs leaving the tail node*	
	MCF_arc_p	**nextin**	*Next arc of the neighbour list of arcs entering the head node*	
3.2.1	MCF_cost_t	**cost**	*Arc costs*	14

3.2.1

MCF_cost_t **cost**

Arc costs

Arc costs.

This variable stands for the arc cost (or weight).

Our primal feasible starting basis consists just of artificial arcs (corresponding to a slack basis), and all originally defined arcs are first nonbasic at their lower bounds. The costs of the artificial arcs are set to MAX_ART_COST, which is defined in the file mcfdefs.h. It is easy to see that any feasible and optimal solution with artificial arcs is also optimal and feasible for the original problem without artificials iff no artifical arc yields a nonzero flow value. If, however, a solution contains an artificial arc with positive flow, the original problem is either indeed infeasible or the MAX_ART_COST is just too small compared to the cost coefficients of the original arcs. If the latter is the case, increase MAX_ART_COST, but we also strongly recommend to use then floating point arithmetic!

3.2.2

MCF_flow_t **upper**

Arc upper bound

Arc upper bound.

This variable stands for the arc upper bound value. Note that an unbounded upper bound is set to UNBOUNDED, which is defined in the file mcfdefs.h. Per default, UNBOUNDED is set to 10^9. Note, this value may be too small for your purposes, and you should increase it appropriately. However, we strongly recommend to use then floating point arithmetic (define MCF_FLOAT)!

3.2.3

MCF_flow_t **lower**

Arc lower bound

Arc lower bound.

This variable stands for the arc lower bound value. Note, this variable is only active if the MCF_LOWER_BOUNDS variable is defined! An negative unbounded lower bound is set to -UNBOUNDED, see also the arc upper bound.

3.2.4

MCF_flow_t **flow**

Arc flow value

Arc flow value.

This variable stands for the arc's flow value. Note that the flow value is not set within the main (primal or dual) iteration loop; actually, it can only be computed using the function primal_obj().

3.2.5

long **ident**

Arc status

Arc status.

This variable shows the current arc status. Feasible is BASIC (for basic arcs), MCF_AT_LOWER_BOUND (nonbasic arcs set to lower bound), MCF_AT_UPPER_BOUND (nonbasic arcs set to the upper bound), MCF_AT_ZERO (nonbasis arcs set to zero), or FIXED (arcs fixed to zero and being not considered by the optimization).

3.3 struct **MCF_network**

Network description

Members

3.3.1	long	**n**	*Number of nodes*	16
3.3.2	long	**m**	*Number of arcs*	17
3.3.3	long	**primal_unbounded**	*Primal unbounded indicator*	17
3.3.4	long	**dual_unbounded**	*Dual unbounded indiciator*	17
3.3.5	long	**feasible**	*Feasible indicator*	18
3.3.6	double	**optcost**	*Costs of current basis solution*	18
3.3.7	MCF_node_p	**nodes**	*Vector of nodes*	18
3.3.8	MCF_node_p	**stop_nodes**	*First infeasible node address*	19
3.3.9	MCF_arc_p	**arcs**	*Vector of arcs*	19
3.3.10	MCF_arc_p	**stop_arcs**	*First infeasible arc address*	19
3.3.11	MCF_arc_p	**dummy_arcs**	*Vector of artificial slack arcs*	20
3.3.12	MCF_arc_p	**stop_dummy**	*First infeasible slack arc address*	20
3.3.13	long	**iterations**	*Iteration count*	20
3.3.14	MCF_node_p	**(*find_iminus)** (long n, MCF_node_p nodes, MCF_node_p stop_nodes, MCF_flow_p delta)	*Dual pricing rule*	21
3.3.15	MCF_arc_p	**(*find_bea)** (long m, MCF_arc_p arcs, MCF_arc_p stop_arcs, MCF_cost_p red_cost_of_bea)	*Primal pricing rule*	21

3.3.1 long **n**

Number of nodes

Number of nodes.

This variable stands for the number of originally defined nodes without the artificial root node.

3.3.2

long **m**

Number of arcs

Number of arcs.

This variable stands for the number of arcs (without the artificial slack arcs).

3.3.3

long **primal_unbounded**

Primal unbounded indicator

Primal unbounded indicator.

This variable is set to one iff the problem is determined to be primal unbounded.

3.3.4

long **dual_unbounded**

Dual unbounded indiciator

Dual unbounded indiciator.

This variable is set to one iff the problem is determined to be dual unbounded.

3.3.5

long **feasible**

Feasible indicator

Feasible indicator.

This variable is set to zero if the problem provides a feasible solution. It can only be set by the function primal_feasible() or dual_feasible() and is not set automatically by the optimization.

3.3.6

double **optcost**

Costs of current basis solution

Costs of current basis solution.

This variable stands for the costs of the current (primal or dual) basis solution. It is set by the return value of primal_obj() or dual_obj().

3.3.7

MCF_node_p **nodes**

Vector of nodes

Vector of nodes.

This variable points to the $n + 1$ node structs (including the root node) where the first node is indexed by zero and represents the artificial root node.

3.3.8

MCF_node_p **stop_nodes**

First infeasible node address

First infeasible node address.

This variable is the address of the first infeasible node address, i. e., it must be set to $nodes + n + 1$.

3.3.9

MCF_arc_p **arcs**

Vector of arcs

Vector of arcs.

This variable points to the m arc structs.

3.3.10

MCF_arc_p **stop_arcs**

First infeasible arc address

First infeasible arc address.

This variable is the address of the first infeasible arc address, i. e., it must be set to $nodes + m$.

3.3.11

MCF_arc_p **dummy_arcs**

Vector of artificial slack arcs

Vector of artificial slack arcs.

This variable points to the artificial slack (or dummy) arc variables and contains n arc structs. The artificial arcs are used to build (primal) feasible starting bases. For each node i, there is exactly one dummy arc defined to connect the node i with the root node.

3.3.12

MCF_arc_p **stop_dummy**

First infeasible slack arc address

First infeasible slack arc address.

This variable is the address of the first infeasible slack arc address, i. e., it must be set to $nodes + n$.

3.3.13

long **iterations**

Iteration count

Iteration count.

This variable contains the number of main simplex iterations performed to solve the problem to optimality.

3.3.14

MCF_node_p **(*find_iminus)** (long n, MCF_node_p nodes, MCF_node_p stop_nodes, MCF_flow_p delta)

Dual pricing rule

Dual pricing rule.

Pointer to the dual pricing rule function that is used by the dual simplex code.

3.3.15

MCF_arc_p **(*find_bea)** (long m, MCF_arc_p arcs, MCF_arc_p stop_arcs, MCF_cost_p red_cost_of_bea)

Primal pricing rule

Primal pricing rule.

Pointer to the primal pricing rule function that is used by the primal simplex code.

4 MCF interface. (vec)

Names

4.1 Problem reading and writing.

Names

4.1.1 extern long **MCF_read_dimacs_min** (char *filename, MCF_network_p net)

Reading procedure

Reading procedure.

Reads minimum-cost flow problem (provided in an extended DIMACS format) from input file named *filename*, mallocs the necessary memory, and creates the network data structure. Each input data are assumed to have the following structure:

- For a network with n nodes it is assumed that the nodes are identified by the integers 1 through n.
- Capacities are integer valued. If you need floating point values, you have to change the source by yourself. Costs and flows are per default integer valued, but with the FLOAT definition in the Makefile, you can use doubles.

- It is assumed that $l_{ij} \leq u_{ij}$ for all $(i,j) \in A$.
- There is no *a priori* restriction on the number of nodes n or the number of arcs m.
- There may be multiple arcs (i,j) between any pair of nodes i and j. The arcs may have differing costs, capacities, and lower bounds.
- It is not necessarily the case that $(i,j) \in A$ implies $(j,i) \in A$.
- If both (i,j) and (j,i) are in A, it is not necessarily the case that $c_{ij} = -c_{ji}$.
- It is *not* assumed that the network has a feasible solution nor that the network is connected. This properties will be provided by the code by introducing an artificial root node that is connected via artifical slack arcs to each node of the input digraph.

The standard DIMACS file format for network input and output is as follows: All files contain ASCII characters. Input and output files contain several types of *lines*, described below. A line is terminated with an end-of-line character. Fields in each line are separated by at least one blank space. Each line begins with a one-character designator to identify the line type.

Input Files: First, for minimum-cost flow problems, we recommend to use suffixes **.min** to be conform with the DIMACS format. Second, files are assumed to be well-formed and internally consistent: node identifier values are valid, nodes are defined uniquely, exactly m arcs are defined, and so forth.

- **Comments.** Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character **c**.

  ```
  c This is an example of a comment line.
  ```

- **Problem line.** There is one problem line per input file. The problem line must appear before any node or arc descriptor lines. For network instances, the problem line has the following format.

  ```
  p min NODES ARCS
  ```

  The lower-case characters `p min` signify that this is a minimum-cost flow problem. The `NODES` field contains an integer value specifying n, the number of nodes in the network. The `ARCS` field contains an integer value specifying m, the number of arcs in the network.

- **Node Descriptors.** All node descriptor lines must appear before all arc descriptor lines. They must describe all supply and demand nodes, i. e., nodes i with a nonzero node imbalance b_i must appear. Transshipment nodes may be leaved out. There is at most one node descriptor line for each node having the following format.

  ```
  n ID FLOW
  ```

  The lower-case character `n` signifies that this is a node descriptor line. The `ID` field gives a node identification number, an integer between 1 and n. The `FLOW` field gives the node flow value b_i.

- **Arc Descriptors.** There is one arc descriptor line for each arc in the network. Arc descriptor lines are of the following form.

  ```
  a SRC DST LOW UPP COST
  ```

The lower-case character **a** signifies that this is an arc descriptor line. For a directed arc (i,j), the `SRC` field gives the identification number for the source vertex i, and the `DST` field gives the destination vertex j. Identification numbers are integers between 1 and n. The `LOW` field contains the lower capacity value l_{ij} and the `UPP` field contains the upper capacity value u_{ij}. Both value can be replaced by "free" or "FREE" identifying unbounded capacities. The value of the capacity is then set to UNBOUNDED defined in mcfdefs.h. The `COST` field contains c_{ij}.

Return Value: integer long value <> 0 indicating an error.

Parameters: `filename` — name of input file to be read.
`net` — reference to network data structure.

4.1.2

extern long **MCF_write_solution** (char *infile, char *outfile, MCF_network_p net, time_t sec)

Writing procedure

Writing procedure.

Writes solution vector in human readable DIMACS format to file. If the *infile* is equal to the *outfile* name, the solution is append to the input file. Otherwise, the solution is written to *outfile.* The output file should list the solution value and the nonzero flow assignments for all arcs (i,j) in A. Three types of lines may appear in output files.

- **Comment Lines.** Comment lines are identical in form to those defined for input files. If there is no feasible solution then the algorithm should report this fact on a comment line (in such a case, neither solution lines nor flow lines will appear in the output).
- **Solution Lines.** The solution line has the following format.

  ```
  s SOLUTION
  ```

  The lower-case character `s` signifies that this is a solution line. The `SOLUTION` field contains the solution value $\sum_{(i,j)\in A} c_{ij}x_{ij}$.
- **Flow Assignments.** There is one flow assignment line for each arc in the network. Flow assignment lines have the following format.

  ```
  f SRC DST FLOW
  ```

  The lower-case character `f` signifies that this is a flow assignment line. For arc (i,j), the `SRC` and `DST` fields give i and j, respectively. The `FLOW` field gives x_{ij}.

Return Value:	integer long value <> 0 indicating an error.
Parameters:	`infile` — Name of input file name
	`outfile` — name of output file name
	`net` — reference to network data structure.
	`sec` — running time of optimization process

4.2 Primal network simplex.

Names

4.2.1 Slack basis.

Names

4.2.1.1 extern long **MCF_primal_start_artificial** (MCF_network_p net)

Generate primal feasible slack basis

Generate primal feasible slack basis.

Let $D' := (V \cup \{0\}, A')$ denote the network obtained by adding the artificial root node 0 to V and the artificial slack arcs $(i, 0)$ and $(0, i)$, respectivly, to A. Each artificial slack arc has a lower bound of 0, an upper bound of infinity, and a sufficiently large cost coefficient MAX_ART_COST, which is defined in the file mcfdefs.h. The initial basis tree consist of all artificial slack arcs, each original arc becomes nonbasic at its lower bound, and no arc becomes nonbasic at its upper bound. Such an initial basis structure is called **artificial basis structure**. Obviously, this artificial basis structure is primal feasible for D' and the original network D is feasible if the network D' has a

The use of an artificial basis tree has several advantages. First, it has a simple structure and can be generated quickly. Second, the ratio test and the basis update are quite fast for the first iterations. We have also tried to generate an initial basis structure using a crash procedure. The performance, however, was always slower than starting with an artificial basis tree. The only exceptions occur for special applications where particular problem knowledge can be exploited, for instance, using a delayed column generation.

Return Value:	integer long value <> 0 indicating an error.
Parameters:	`net` — reference to network data structure.

4.2.2

Main iteration loop.

Names

4.2.2.1

extern long **MCF_primal_net_simplex** (MCF_network_p net)

Primal network simplex main iteration loop

Primal network simplex main iteration loop.

For a detailed description of all these single network simplex steps, the reader is referred to Helgason and Kennington [1995].

Return Value:	integer long value <> 0 if the problem is primal unbounded.
Parameters:	`net` — reference to network data structure.

4.2.3

Pricing.

Names

From our point of view, the pricing rule has the most significant influence on the performance of a network simplex implementation. In the literature, there are some pricing rules described as Dantzig's rule, first eligible arc rule, or candidate list rules. We have implemented and tested these pricing rules in slightly modified ways and provide them in our implementation. It turned out that our by far fastest rules are special candidate list rules, called **multiple partial pricing**.

Return Value:	reference to the basis entering arc or NULL pointer if the current basis is optimal with respect to the optimality tolerance.
Parameters:	`net` — reference to network data structure.

4.2.3.1

Multiple partial pricing.

Names

extern MCF_arc_p
MCF_primal_bea_mpp_30_5 (long m, MCF_arc_p arcs,
MCF_arc_p stop_arcs,
MCF_cost_p red_cost_of_bea)
K = 30, B = 5

extern MCF_arc_p
MCF_primal_bea_mpp_50_10 (long m, MCF_arc_p arcs,
MCF_arc_p stop_arcs,
MCF_cost_p red_cost_of_bea)
K = 50, B = 10

extern MCF_arc_p
MCF_primal_bea_mpp_200_20 (long m, MCF_arc_p arcs,
MCF_arc_p stop_arcs,
MCF_cost_p red_cost_of_bea)
K = 200, B = 20

The declaration of the multiple partial pricing rules is extern arc_t *primal_bea_mpp_K_B (int m, arc_t *arcs, arc_t *stop_arcs, cost_t *red_cost_of_bea);

where K and B denote two natural numbers.

The arc set A (without the artificial slack arcs) is divided into $\lceil\frac{|A|}{K}\rceil$ candiadate lists, each of size at most K. If the arcs are indexed from 1 to $|A|$, the k^{th} candidate list includes all arcs i satisfying $(i-1)$ modulo $K = (k-1)$. There is a ***hot-list*** of at most $B+K$ arcs, which is initially empty. The candiate list number *next*, which defines the first to be examined candidate list in the initial pricing call, is set to 1. The candidate lists are always examined in a wraparound fashion. For a pricing call, the following steps are performed: First, the reduced costs of the arcs being currently in the hot-list are recomputed. If the new reduced costs of such an arcs becomes nonnegative, this arc is immediatly removed from the hot-list. Second, as long as the hot-list can be filled with at least K additional arcs and not all candidate lists have been examined in this pricing call, we price out all arcs of the *next* candidate list, add all nonbasic arcs of this list having negative reduced costs to the hot-list, and increment the *next* variable by 1 (if *next*>K, otherwise we reset *next* to 1). Third, if all candidate lists have been examined, but the hot-list is still empty, the current basis structure is optimal. Otherwise, some arc of the hot-list that violates the reduced cost criterion at most is selected as the basis entering arc. The last step of a pricing call is the preparation of the hot-list for the next pricing call: The new hot-list for the next pricing contains at most B arcs among those arcs of the current hot-list that are not the basis entering arc and that have the most invalid reduced costs.

Multiple partial pricing is very sensitive to the number of arcs making necessary a fine tuning for every problem class. We use and recommend the following default values for K and B depending on the number of arcs (note that a further fine tuning for your data may speed up the code):

Number of arcs	K	J
$\|A\|<10,000$	30	5
$10,000 \leqslant \|A\| \leqslant 100,000$	50	10
$\|A\|>100,000$	200	20

4.2.3.2

First eligible arc rule.

Names

4.2.3.2.1 extern MCF_arc_p
MCF_primal_bea_cycle (long m, MCF_arc_p arcs,
MCF_arc_p stop_arcs,
MCF_cost_p red_cost_of_bea)
First eligible arc pricing 29

4.2.3.2.1

extern MCF_arc_p **MCF_primal_bea_cycle** (long m, MCF_arc_p arcs, MCF_arc_p stop_arcs, MCF_cost_p red_cost_of_bea)

First eligible arc pricing

First eligible arc pricing. Searches for the basis entering arc in a wraparound fashion.

4.2.3.3

Dantzig's rule.

Names

4.2.3.3.1

extern MCF_arc_p **MCF_primal_bea_all** (long m, MCF_arc_p arcs, MCF_arc_p stop_arcs, MCF_cost_p red_cost_of_bea)

Dantzig's rule

Dantzig's rule. Determins the arc violating the optimality condition at most.

4.3

Dual network simplex.

Names

4.3.1 Start basis.

Names

4.3.1.1 extern long **MCF_dual_start_artificial** (MCF_network_p net)

Generate dual feasible starting basis

Generate dual feasible starting basis.

Return Value: integer long value <> 0 if the problem is dual infeasible.
Parameters: `net` — reference to network data structure.

4.3.2 Main iteration loop.

Names

4.3.2.1

extern long **MCF_dual_net_simplex** (MCF_network_p net)

Dual network simplex main iteration loop

Dual network simplex main iteration loop.

Return Value:	integer long value <> 0 if the problem is dual infeasible or unbounded.
Parameters:	`net` — reference to network data structure.

4.3.3

Pricing.

Names

The dual pricing methods are similar to the primal ones, but in the dual code we have to search for basis leavings arcs violating their bounds instead of finding a basis entering arc instead of violating the reduced cost criterion.

4.3.3.1

Multiple partial pricing

Names

extern MCF_node_p
MCF_dual_iminus_mpp_30_5 (long n, MCF_node_p nodes,
MCF_node_p stop_nodes,
MCF_flow_p delta)
K = 30, B = 5

extern MCF_node_p
MCF_dual_iminus_mpp_50_10 (long n, MCF_node_p nodes,
MCF_node_p stop_nodes,
MCF_flow_p delta)
K = 50, B = 10

4.3.3.2

First eligible arc rule.

Names

4.3.3.2.1

extern MCF_node_p **MCF_dual_iminus_cycle** (long n, MCF_node_p nodes, MCF_node_p stop_nodes, MCF_flow_p delta)

First eligible arc pricing

First eligible arc pricing. Searches for the basis leaving arc in a wraparound fashion.

4.4

MCF utilities.

Names

4.4.1

extern long **MCF_free** (MCF_network_p net)

Frees malloced data structures

Frees malloced data structures.

Parameters: `net` — reference to network data structure.

4.4.2

extern double **MCF_primal_obj** (MCF_network_p net)

Primal objective $c^T x$

Primal objective $c^T x$.

Parameters: `net` — reference to network data structure.

4.4.3

extern double **MCF_dual_obj** (MCF_network_p net)

Dual objective $\pi^T b + \lambda^T l - \eta^T u$

Dual objective $\pi^T b + \lambda^T l - \eta^T u$.

Parameters: `net` — reference to network data structure.

4.4.4

extern long **MCF_primal_feasible** (MCF_network_p net)

Primal basis checking

Primal basis checking.

Checks whether a given basis is primal feasible.

Return Value:	value <> 0 indicates primal infeasible basis.
Parameters:	`net` — reference to network data structure.

4.4.5

extern long **MCF_dual_feasible** (MCF_network_p net)

Dual basis checking

Dual basis checking.

Checks whether a given basis is dual feasible.

Return Value:	value <> 0 indicates dual infeasible basis.
Parameters:	`net` — reference to network data structure.

4.4.6

extern long **MCF_is_basis** (MCF_network_p net)

Basis checking

Basis checking.

Checks whether the given basis structure is a spanning tree.

Return Value:	value <> 0 indicates infeasible spanning tree.
Parameters:	`net` — reference to network data structure.

4.4.7

extern long **MCF_is_balanced** (MCF_network_p net)

Flow vector checking

Flow vector checking.

Checks whether a given basis solution defines a balanced flow on each node.

Return Value:	value <> 0 indicates unbalanced flow vector.
Parameters:	`net` — reference to network data structure.

5

ZIB Academic License.

This license is designed for ZIB software to guarantee freedom to share and change software **for academic use**, but restricting commercial firms from exploiting our knowhow for their benefit. The precise terms and conditions for use, copy, distribution, and modification are as follows.

The term "Program" below refers to source, object, and executable code, and a the term "Work based on the Program" means either the Program or any derivative work under copyright law: that is a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. Each licensee is addressed as "you".

- This license applies to you only if you are a member of a noncommercial and academic institution, e.g., a university. The license expires as soon as you are no longer a member of this institution.
- Every publication and presentation for which the Work based on the Program or its output has been used must contain an appropriate citation and acknowledgement of the author(s) of the Program.
- You may copy and distribute the Program or Work based on the Program in source, object, or executable form provided that you also meet all of the following conditions:
  - You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge under the terms of this License. You must accompany it with this unmodified license text.

    These requirements apply to the Program or Work based on the Program as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered as independent and separate works in themselves, this License does not apply to those sections when you distribute them as separate works. But when you distribute the same sections as a Work based on the Program, the distribution of the whole must be on the terms of this license, whose permissions for other licensees extend to the entire whole and, thus, to each and every part regardless of who wrote it.
  - You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
  - You must keep track of access to the Program (e.g., similar to the registration procedure at ZIB).
  - You must accompany it with the complete corresponding human-readable source code. The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.
- You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute the Program is void and will automatically terminate your rights under this License. However, parties who have received copies or rights from you under this License will not have their licenses terminated so long as such parties remain in full compliance.