RelaxIV Documentation

version 1.80

date 11 - 02 - 2011

Operations Research Group

Dipartimento di Informatica

Università di Pisa

Chapter 1

Manual

1.1 Introduction

This file is a short user manual for our distribution of RelaxIV, a C++ implementation of the latest variants of Relaxation algorithms for linear Min Cost Flow problems.

1.2 Standard Disclaimer

This code is provided "as is", without any explicit or implicit warranty that it will properly behave or it will suit you needs. Although codes reaching the distribution phase have usually been extensively tested, we cannot guarantee that they are absolutely bug-free (who can?). Any use of the codes is at you own risk: in no case we could be considered liable for any damage or loss you would evenctually suffer, either directly or indirectly, for having used this code. More details about the non-warranty attached to this code are available in the license description file.

The code also comes with a "good will only" support: feel free to contact us for any comments/critics/bug report/request help you may have, we will be happy to try to answer and help you. But we cannot spend much time solving your problems, just the time to read a couple of e-mails and send you fast suggestions if the problem is easily solvable. Apart from that, we can't offer you any support.

1.3 License

This code is provided free of charge for academic purposes under the "academic license": see the file academicl.txt. Commercial use of this code is only allowed at a fee: please contact the original author dimitrib@mit.edu for further information.

1.4 Release

Current version of RelaxIV is: 1.80 (after a major renumbering) Current date is: February 11, 2011

1.5 How to Use It

This release comes out with the following files:

- Manual This file.

- doc/academicl.txt Description of the "academic license".

- doc/refman.pdf Pdf version of the full manual.

- doc/html/* Html version of the full manual.

- OPTUtils.h Contains some small utilities such as random number generators and timing routines.

- MCFClass.h Contains the declaration of class MCFClass; it is an "abstract" class with "pure virtual" methods, so that you cannot declare objects of type MCFClass, but only of derived class of its. MCFClass offers a general interface for MCF solver codes, with methods for setting and reading the data of the problems, for solving it and retrieving solution informations, and so on. The actual Min Cost Flow solver distributed in this package conforms with the interface, i.e., it derives from MCFClass; however, the idea is that applications using this interface would require almost no changes if any other of the available solvers implementing the interface is used. Carefully read the public interface of the class to understend how to use the public methods of the class. Also, some compile time switches let you choose important features of the algorithm, such as if the node "names" are the integers 0 .. n - 1 rather than 1 .. n.

- RelaxIV.h Contains the declarations of class RelaxIV. Being a derived class of MCFClass, most of its interface is defined and discussed in MCFClass.h; however, some implementation dependent details (and compile-time switches) which are worth knowing are described in this file.

- RelaxIV.C Contains the implementation of the RelaxIV class, which uses the Relaxation algorithm version IV (due to Dimitri P. Bertsekas) to solve the Min Cost Flow problem. You should not need to read it.

The following files are not a part of the actual MCF solver, they are only provided as an example of how to use it:

- Main.C Contains an example of use of RelaxIV; it reads the data from a file in DIMACS standard format, constructs the problem, solves it and prints the results.

- Config.txt Allows (if exists) to set some important parameters for the solver.

- sample.dmx An example of a tiny (4 nodes, 4 arcs) network in DIMACS standard format; the comments in the file also describe the format itself.

- makefile A makefile of the example.

1.6 Testing the distribution

To compile the example, edit the makefile and do the proper changes (such as the compiler name), then just type "make". To execute the example

MCFSolve sample.dmx

The output should be something like Optimal Objective Function value = 4 Solution time (s): user 0, system 0

Contents

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Module Documentation

6.1 Compile-time switches in MCFClass.h

Defines

- #define USENAME0 0

6.1.1 Detailed Description

These macros control some important details of the class interface. Although using macros for activating features of the interface is not very C++, switching off some unused features may allow some implementation to be more efficient in running time or memory.

6.1.2 Define Documentation

6.1.2.1 #define USENAME0 0

Decides if 0 or 1 is the "name" of the first node. If USENAME0 == 1, (warning: it has to be ∗exactly∗ 1), then the node names go from 0 to n - 1, otherwise from 1 to n. Note that this does not affect the position of the deficit in the deficit vectors, i.e., the deficit of the i-th node - be its "name" ‘i’ or ‘i - 1’ - is always in the i-th position of the vector.

6.2 Classes in MCFClass.h

Classes

- class MCFClass

6.3 Compile-time switches in OPTUtils.h

Defines

- #define OPT_USE_NAMESPACES 0
- #define OPT_TIMERS 5
- #define OPT_RANDOM 1

6.3.1 Detailed Description

These macros control how the classes OPTTimers and OPTrand are implemented; choose the appropriate value for your environment, or program a new version if no value suits you. Also, namespaces can be eliminated if they create problems.

6.3.2 Define Documentation

6.3.2.1 #define OPT_RANDOM 1

The class OPTrand is defined below to give an abstract interface to the different random generators that are used in different platforms. This is needed since random generators are one of the less standard parts of the C[++] library. The value of the OPT_RANDOM constant selects among the different timing routines:

- 0 = an hand-made implementation of a rather good random number generator is used; note that this assumes that long ints >= 32 bits
- 1 = standard rand() / srand() pair, common to all C libreries but not very sophisticated
- 2 = drand48() / srand48(), common on Unix architectures and pretty good.

Any unsupported value would simply make the functions to report constant zero, which is not nice but useful to quickly fix problems if you don't use random numbers at all.

6.3.2.2 #define OPT_TIMERS 5

The class OPTtimers is defined below to give an abstract interface to the different timing routines that are used in different platforms. This is needed since time-related functions are one of the less standard parts of the C[++] library. The value of the OPT_TIMERS constant selects among the different timing routines:

- 1 = Use the Unix times() routine in sys/times.h
- 2 = As 1 but uses sys/timeb.h (typical for Microsoft(TM) compilers)
- 3 = Still use times() of sys/times.h, but return wallclock time rather than CPU time
- 4 = As 3 but uses sys/timeb.h (typical for Microsoft(TM) compilers)
- 5 = return the user time obtained with ANSI C clock() function; this may result in more accurate running times w.r.t. but may be limited to $\sim$ 72 hours on systems where ints are 32bits.

- 6 = Use the Unix gettimeofday() routine of sys/time.h.

Any unsupported value would simply make the class to report constant zero as the time.

The values 1 .. 4 rely on the constant CLK_TCK for converting between clock ticks and seconds; for the case where the constant is not defined by the compiler -- should not happen, but it does -- or it is defined in a wrong way, the constant is re-defined below.

6.3.2.3 #define OPT_USE_NAMESPACES 0

Setting OPT_USE_NAMESPACES == 0 should instruct all codes that use OPTUtils stuff to avoid using namespaces; to start with, the common namespace OPTUtils_di_unipi_it, that contains all the types defined herein, is ∗not∗ defined.

6.4 Classes in OPTUtils.h

Classes

- class OPTrand

6.5 Functions in OPTUtils.h

Functions

- template<class T >
  void DfltdSfInpt (istream ∗iStrm, T &Param, const T Dflt, const char cmntc= ’#’)

6.5.1 Function Documentation

6.5.1.1 template<class T > void DfltdSfInpt (istream ∗ *iStrm*, T & *Param*, const T *Dflt*, const char *cmntc* = `'#'`) `[inline]`

Template function for reading parameters from a istream. The function is "safe" because it works also if the istream is not given, is not be long enough or contains erroneous things.

Given a &istream (possibly NULL), DfltdSfInpt() attempts to read Param out of it, skipping any line that begins with the comment carachter (defaulted to ’#’), any blank line and any line starting with anything that can not be interpreted as a ‘T’. If, for any reason, the read operation fails, then the parameter is given the default value ‘Dflt’. Otherwise, all the rest of the line up to the nearest newline (’

’) carachter is flushed.

Note

lines should not be longer than 255 carachters.

6.6 Compile-time switches in RelaxIV.h

Defines

- #define SAME_GRPH_RIV 0
- #define DYNMC_MCF_RIV 0
- #define AUCTION 0
- #define RELAXIV_STATISTICS 0

6.6.1 Detailed Description

These macros control some important details of the implementation. Although using macros for activating features of the implementation is not very C++, switching off some unused features may make the code more efficient in running time or memory.

6.6.2 Define Documentation

6.6.2.1 #define AUCTION 0

Decides if the auction/shortest paths inizialization procedure is used. If AUCTION == 1, then an auction/shortest paths inizialization procedure is provided [see SetAuction() below] that has been reported to make the RelaxIV algorithm run faster on some classes of instances. The auction initialization essentially "spreads" the imbalances around the graph by performing some steps of the "pure" epsilon-relaxation method: this should produce "short" augmenting steps, that seem to be the best situation for RelaxIV.

By setting AUCTION == 0, some memory is saved.

6.6.2.2 #define DYNMC_MCF_RIV 0

Decides if the graph topology (arcs, nodes) can be changed. If DYNMC_MCF_RIV $>$ 0, the methods of the public interface of class that allow to change the topology of the underlying network are actually implemented. Possible values of this macro are:

- 0 => the topology of the graph cannot be changed;
- 1 => the methods that "close" arcs and delete nodes are implemented;
- 2 => the methods that "open" previously closed arcs and add nodes are implemented;
- 3 => the methods that change the start and end node of a (possibly "closed") arc, delete and create new arcs are implemented.

As long as DYNMC_MCF_RIV $<$ 3, SAME_GRPH_RIV can possibly be $>$ 0; less and less data structures can be shared as DYNMC_MCF_RIV increases. DYNMC_MCF_RIV == 3 implies SAME_GRPH_RIV == 0.

6.6.2.3 #define RELAXIV_STATISTICS 0

If RELAXIV_STATISTICS $>$ 0, then statistic information about the behaviour of the Relaxation algorithm is computed.

6.6.2.4 #define SAME_GRPH_RIV 0

Decides if all MCFClass instances share the same graph. If SAME_GRPH_RIV $>$ 0, then all the instances of the class will work on the same "topological" network, while the costs, capacities and supplies can change from one instance to another. This allows implementations to share some data structures describing the graph, e.g. by declaring them "static", saving memory when multiple instances of the solver are active at the same time. However, this also obviously inhibits some type of changes in the topology of the graph [see DYNMC_MCF_RIV below].

6.7 Classes in RelaxIV.h

Classes

- class RelaxIV

Chapter 7

Class Documentation

7.1 MCFClass::Eps< T > Class Template Reference

```
#include <MCFClass.h>
```

Public Member Functions

- **operator T** ()

7.1.1 Detailed Description

template<typename T> class MCFClass::Eps< T >

Very small class to simplify extracting the "machine epsilon" for a basic type (FNumber, CNumber); just use Eps<type>().

The documentation for this class was generated from the following file:

- MCFClass.h

7.2 MCFClass::Inf< T > Class Template Reference

```
#include <MCFClass.h>
```

Public Member Functions

- **operator T** ()

7.2.1 Detailed Description

template<typename T> class MCFClass::Inf< T >

Very small class to simplify extracting the "+ infinity" value for a basic type (FNumber, CNumber, Index); just use Inf<type>().

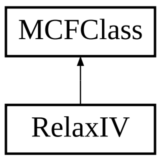
The documentation for this class was generated from the following file:

- MCFClass.h

7.3 MCFClass Class Reference

`#include <MCFClass.h>`

Inheritance diagram for MCFClass:



Classes

- class Eps
- class Inf
- class MCFException
- class MCFState

Public Types

Public types

The MCFClass defines four main public types:

- *Index, the type of arc and node indices;*
- *FNumber, the type of flow variables, arc capacities, and node deficits;*
- *CNumber, the type of flow costs, node potentials, and arc reduced costs;*
- *FONumber, the type of objective function value.*

By re-defining the types in this section, most MCFSolver should be made to work with any reasonable choice of data type (= one that is capable of properly representing the data of the instances to be solved). This may be relevant due to an important property of MCF problems: ∗if all arc capacities and node deficits are integer, then there exists an integral optimal primal solution∗, and ∗if all arc costs are integer, then there exists an integral optimal dual solution∗. Even more importantly, ∗many solution algorithms will in fact produce an integral primal/dual solution for free∗, because ∗every primal/dual solution they generate during the solution process is naturally integral∗. Therefore, one can use integer data types to represent everything connected with flows and/or costs if the corresponding data is integer in all instances one needs to solve. This directly translates in significant memory savings and/or speed improvements.

It is the user's responsibility to ensure that these types are set to reasonable values∗. So, the experienced user may want to experiment with setting this data properly if memory footprint and/or speed is a primary concern. Note, however, that ∗not all solution algorithms will happily accept integer data∗; one example are Interior-Point approaches, which require both flow and cost variables to be continuous (float). So, the viability of setting integer data (as well as its impact on performances) is strictly related to the specific kind of algorithm used. Since these types are common to all derived classes, they have to be set taking into account the needs of all the solvers that are going to be used, and adapting to the "worst case"; of course, FNumber == CNumber == double is going to always be an acceptable "worst case" setting. MCFClass may in a future be defined as a template class, with these as template parameters, but this is currently deemed overkill and avoided.

Finally, note that the above integrality property only holds for ∗linear∗ MCF problems. If any arc has a nonzero quadratic cost coefficient, optimal flows and potentials may be fractional even if all the data of the problem (comprised quadratic cost coefficients) is integer. Hence, for ∗quadratic∗ MCF solvers, a setting like FNumber == CNumber == double is actually mandatory∗, for any reasonable algorithm will typically misbehave otherwise.

- enum MCFParam {
  kMaxTime = 0, kMaxIter, kEpsFlw, kEpsDfct,
  kEpsCst, kReopt, kLastParam }
- enum MCFStatus {
  kUnSolved = -1, kOK = 0, kStopped, kUnfeasible,
  kUnbounded, kError }
- enum MCFAnswer { kNo = 0, kYes }
- enum MCFFlFrmt { kDimacs = 0, kQDimacs, kMPS, kFWMPS }
- typedef unsigned int Index

  index of a node or arc (>= 0)
- typedef Index ∗ Index_Set

  set (array) of indices
- typedef const Index cIndex

  a read-only index
- typedef cIndex ∗ cIndex_Set

  read-only index array
- typedef int SIndex

  index of a node or arc
- typedef SIndex ∗ SIndex_Set

  set (array) of indices
- typedef const SIndex cSIndex

  a read-only index
- typedef cSIndex ∗ cSIndex_Set

  read-only index array
- typedef double FNumber

  type of arc flow
- typedef FNumber ∗ FRow

  vector of flows
- typedef const FNumber cFNumber

  a read-only flow
- typedef cFNumber ∗ cFRow

  read-only flow array
- typedef double CNumber

  type of arc flow cost

- typedef CNumber ∗ CRow

  vector of costs

- typedef const CNumber cCNumber

  a read-only cost

- typedef cCNumber ∗ cCRow

  read-only cost array

- typedef double FONumber
- typedef const FONumber cFONumber

  a read-only o.f. value

- typedef MCFState ∗ MCFStatePtr

  pointer to a MCFState

Public Member Functions

Constructors

- MCFClass (cIndex nmx=0, cIndex mmx=0)

Other initializations

- virtual void LoadNet (cIndex nmx=0, cIndex mmx=0, cIndex pn=0, cIndex pm=0, cFRow pU=NULL, cCRow pC=NULL, cFRow pDfct=NULL, cIndex_Set pSn=NULL, cIndex_Set pEn=NULL)=0
- virtual void LoadDMX (istream &DMXs, bool IsQuad=false)
- virtual void PreProcess (void)
- virtual void SetPar (int par, int val)
- virtual void SetPar (int par, double val)
- virtual void GetPar (int par, int &val)
- virtual void GetPar (int par, double &val)
- virtual void SetMCFTime (bool TimeIt=true)

Solving the problem

- virtual void SolveMCF (void)=0
- int MCFGetStatus (void)

Reading flow solution

- virtual void MCFGetX (FRow F, Index_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual cFRow MCFGetX (void)
- virtual bool HaveNewX (void)

Reading potentials

- virtual void MCFGetPi (CRow P, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual cCRow MCFGetPi (void)
- virtual bool HaveNewPi (void)

Reading reduced costs

- virtual void MCFGetRC (CRow CR, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual cCRow MCFGetRC (void)
- virtual CNumber MCFGetRC (cIndex i)=0

Reading the objective function value

- virtual FONumber MCFGetFO (void)=0
- virtual FONumber MCFGetDFO (void)

Getting unfeasibility certificate

- virtual FNumber MCFGetUnfCut (Index_Set Cut)

Getting unboundedness certificate

- virtual Index MCFGetUnbCycl (Index_Set Pred, Index_Set ArcPred)

Saving/restoring the state of the solver

- virtual MCFStatePtr MCFGetState (void)
- virtual void MCFPutState (MCFStatePtr S)

Time the code

- void TimeMCF (double &t_us, double &t_ss)
- double TimeMCF (void)

Check the solutions

- void CheckPSol (void)
- void CheckDSol (void)

Reading graph size

- Index MCFnmax (void)
- Index MCFmmax (void)
- Index MCFn (void)
- Index MCFm (void)

Reading graph topology

- virtual void MCFArcs (Index_Set Startv, Index_Set Endv, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual Index MCFSNde (cIndex i)=0
- virtual Index MCFENde (cIndex i)=0
- virtual cIndex_Set MCFSNdes (void)
- virtual cIndex_Set MCFENdes (void)

Reading arc costs

- virtual void MCFCosts (CRow Costv, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0

- virtual CNumber MCFCost (cIndex i)=0
- virtual cCRow MCFCosts (void)
- virtual void MCFQCoef (CRow Qv, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- virtual CNumber MCFQCoef (cIndex i)
- virtual cCRow MCFQCoef (void)

Reading arc capacities

- virtual void MCFUCaps (FRow UCapv, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual FNumber MCFUCap (cIndex i)=0
- virtual cFRow MCFUCaps (void)

Reading node deficits

- virtual void MCFDfcts (FRow Dfctv, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual FNumber MCFDfct (cIndex i)=0
- virtual cFRow MCFDfcts (void)

Write problem to file

- virtual void WriteMCF (ostream &oStrm, int frmt=0)

Changing the costs

- virtual void ChgCosts (cCRow NCost, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual void ChgCost (Index arc, cCNumber NCost)=0
- virtual void ChgQCoef (cCRow NQCoef=NULL, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- virtual void ChgQCoef (Index arc, cCNumber NQCoef)

Changing the capacities

- virtual void ChgUCaps (cFRow NCap, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual void ChgUCap (Index arc, cFNumber NCap)=0

Changing the deficits

- virtual void ChgDfcts (cFRow NDfct, cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())=0
- virtual void ChgDfct (Index node, cFNumber NDfct)=0

Changing graph topology

- virtual void CloseArc (cIndex name)=0
- virtual bool IsClosedArc (cIndex name)=0
- virtual void DelNode (cIndex name)=0
- virtual void OpenArc (cIndex name)=0
- virtual Index AddNode (cFNumber aDfct)=0
- virtual void ChangeArc (cIndex name, cIndex nSN=Inf< Index >(), cIndex nEN=Inf< Index >())=0

- virtual void DelArc (cIndex name)=0
- virtual bool IsDeletedArc (cIndex name)=0
- virtual Index AddArc (cIndex Start, cIndex End, cFNumber aU, cCNumber aC)=0

Destructor

- virtual ~MCFClass ()

Protected Member Functions

Managing comparisons.

The following methods are provided for making it easier to perform comparisons, with and without tolerances.

- template<class T >
  bool ETZ (T x, const T eps)
- template<class T >
  bool GTZ (T x, const T eps)
- template<class T >
  bool GEZ (T x, const T eps)
- template<class T >
  bool LTZ (T x, const T eps)
- template<class T >
  bool LEZ (T x, const T eps)
- template<class T >
  bool GT (T x, T y, const T eps)
- template<class T >
  bool LT (T x, T y, const T eps)

Protected Attributes

- Index n

  total number of nodes

- Index nmax

  maximum number of nodes

- Index m

  total number of arcs

- Index mmax

  maximum number of arcs

- int status
- bool Senstv

  true <=> the latest optimal solution should be exploited

- OPTtimers ∗ MCFt

  timer for performances evaluation

- FNumber EpsFlw

precision for comparing arc flows / capacities

- FNumber EpsDfct

  precision for comparing node deficits

- CNumber EpsCst

  precision for comparing arc costs

- double MaxTime
- int MaxIter

7.3.1 Detailed Description

This abstract base class defines a standard interface for (linear or convex quadartic separable) Min Cost Flow (MCF) problem solvers.

The data of the problem consist of a (directed) graph G = (N , A) with n = |N| nodes and m = |A| (directed) arcs. Each node 'i' has a deficit b[i], i.e., the amount of flow that is produced/consumed by the node: source nodes (which produce flow) have negative deficits and sink nodes (which consume flow) have positive deficits. Each arc '(i, j)' has an upper capacity U[i , j], a linear cost coefficient C[i , j] and a (non negative) quadratic cost coefficient Q[i , j]. Flow variables X[i , j] represents the amount of flow to be sent on arc (i, j). Parallel arcs, i.e., multiple copies of the same arc '(i, j)' (with possibly different costs and/or capacities) are in general allowed. The formulation of the problem is therefore:

$$\min \sum_{(i,j)\in A} C[i,j]X[i,j] + Q[i,j]X[i,j]^2/2$$

$$(1) \sum_{(j,i)\in A} X[j,i] - \sum_{(i,j)\in A} X[i,j] = b[i] \qquad i \in N$$

$$(2) 0 \leq X[i,j] \leq U[i,j] \qquad (i,j) \in A$$

The n equations (1) are the flow conservation constraints and the 2m inequalities (2) are the flow non-negativity and capacity constraints. At least one of the flow conservation constraints is redundant, as the demands must be balanced ($\sum_{i\in N} b[i] = 0$); indeed, exactly n - ConnectedComponents(G) flow conservation constraints are redundant, as demands must be balanced in each connected component of G. Let us denote by QA and LA the disjoint subsets of A containing, respectively, "quadratic" arcs (with Q[i , j] > 0) and "linear" arcs (with Q[i , j] = 0); the (MCF) problem is linear if QA is empty, and nonlinear (convex quadratic) if QA is nonempty.

The dual of the problem is:

$$\max \sum_{i\in N} Pi[i]b[i] - \sum_{(i,j)\in A} W[i,j]U[i,j] - \sum_{(i,j)\in AQ} V[i,j]^2/(2 * Q[i,j])$$

$$(3.a) C[i,j] - Pi[j] + Pi[i] + W[i,j] - Z[i,j] = 0 \qquad (i,j) \in AL$$

$$(3.b) C[i,j] - Pi[j] + Pi[i] + W[i,j] - Z[i,j] = V[i,j] \qquad (i,j) \in AQ$$

$$(4.a) W[i,j] \geq 0 \qquad (i,j) \in A$$

$$(4.b) Z[i,j] \geq 0 \qquad (i,j) \in A$$

Pi[] is said the vector of node potentials for the problem, W[] are bound variables and Z[] are slack variables. Given Pi[], the quantities

$$RC[i,j] = C[i,j] + Q[i,j] * X[i,j] - Pi[j] + Pi[i]$$

are said the "reduced costs" of arcs.

A primal and dual feasible solution pair is optimal if and only if the complementary slackness conditions

$$RC[i,j] > 0 \Rightarrow X[i,j] = 0$$

$$RC[i,j] < 0 \Rightarrow X[i,j] = U[i,j]$$

are satisfied for all arcs (i, j) of A.

The MCFClass class provides an interface with methods for managing and solving problems of this kind. Actually, the class can also be used as an interface for more general NonLinear MCF problems, where the cost function either nonseparable (C(X)) or arc-separable ($\sum_{(i,j)\in A} C_{i,j}(X[i,j])$). However, solvers for NonLinear MCF problems are typically objective-function-specific, and there is no standard way for inputting a nonlinear function different from a separable convex quadratic one, so only the simplest form is dealt with in the interface, leaving more complex NonLinear parts to the interface of derived classes.

7.3.2 Member Typedef Documentation

7.3.2.1 typedef double MCFClass::FONumber

type of the objective function: has to hold sums of products of FNumber(s) by CNumber(s)

7.3.3 Member Enumeration Documentation

7.3.3.1 enum MCFClass::MCFAnswer

Public enum describing the possible reoptimization status of the MCF solver.

Enumerator:

kNo	no
kYes	yes

7.3.3.2 enum MCFClass::MCFFlFrmt

Public enum describing the possible file formats in WriteMCF().

Enumerator:

kDimacs	DIMACS file format for MCF.
kQDimacs	quadratic DIMACS file format for MCF
kMPS	MPS file format for LP.
kFWMPS	"Fixed Width" MPS format

7.3.3.3 enum MCFClass::MCFParam

Public enum describing the possible parameters of the MCF solver, to be used with the methods SetPar() and GetPar().

Enumerator:

kMaxTime	max time

kMaxIter max number of iteration

kEpsFlw tolerance for flows

kEpsDfct tolerance for deficits

kEpsCst tolerance for costs

kReopt whether or not to reoptimize

kLastParam dummy parameter: this is used to allow derived classes to "extend" the set of parameters.

7.3.3.4 enum MCFClass::MCFStatus

Public enum describing the possible status of the MCF solver.

Enumerator:

kUnSolved no solution available

kOK optimal solution found

kStopped optimization stopped

kUnfeasible problem is unfeasible

kUnbounded problem is unbounded

kError error in the solver

7.3.4 Constructor & Destructor Documentation

7.3.4.1 MCFClass::MCFClass (cIndex *nmx* = `0`, cIndex *mmx* = `0`) `[inline]`

Constructor of the class.

nmx and mmx, if provided, are taken to be respectively the maximum number of nodes and arcs in the network. If nonzero values are passed, memory allocation can be anticipated in the constructor, which is sometimes desirable. The maximum values are stored in the protected fields nmax and mmax, and can be changed with LoadNet() [see below]; however, changing them typically requires memory allocation/deallocation, which is sometimes undesirable outside the constructor.

After that an object has been constructed, no problem is loaded; this has to be done with LoadNet() [see below]. Thus, it is an error to invoke any method which requires the presence of a problem (typicall all except those in the initializations part). The base class provides two protected fields n and m for the current number of nodes and arcs, respectively, that are set to 0 in the constructor precisely to indicate that no instance is currently loaded.

7.3.4.2 virtual MCFClass::∼MCFClass () `[inline, virtual]`

Destructor of the class. The implementation in the base class only deletes the MCFt field. It is virtual, as it should be.

7.3.5 Member Function Documentation

7.3.5.1 virtual Index MCFClass::AddArc (cIndex *Start*, cIndex *End*, cFNumber *aU*, cCNumber *aC*) `[pure virtual]`

Add the new arc (Start , End) with cost aC and capacity aU, returning its name. Inf<Index>() is returned if there is no room for a new arc. Remember that arc names go from 0 to mmax - 1.

Implemented in RelaxIV.

7.3.5.2 virtual Index MCFClass::AddNode (cFNumber *aDfct*) `[pure virtual]`

Add a new node with deficit aDfct, returning its name. Inf<Index>() is returned if there is no room for a new node. Remember that the node names are either { 0 .. nmax - 1 } or { 1 .. nmax }, depending on the value of USENAME0.

Implemented in RelaxIV.

7.3.5.3 virtual void MCFClass::ChangeArc (cIndex *name*, cIndex *nSN* = Inf< Index >(), cIndex *nEN* = Inf< Index >()) `[pure virtual]`

Change the starting and/or ending node of arc ‘name’ to nSN and nEN. Each parameter being Inf<Index>() means to leave the previous starting or ending node untouched. When this method is called ‘name’ can be either the name of a "normal" arc or that of a "closed" arc [see CloseArc() above]: in the latter case, at the end of ChangeArc() the arc is ∗still closed∗, and it remains so until OpenArc(name) [see above] is called.

Implemented in RelaxIV.

7.3.5.4 void MCFClass::CheckDSol (void) `[inline]`

Check that the dual solution returned by the solver is dual feasible. (to within the tolerances set by SetPar(kEps∗∗∗∗) [see above], if any). Also, check that the objective function value is correct.

This method is implemented by the base class, using the above methods for collecting the solutions and the methods of the next section for reading the data of the problem; as such, they will work for any derived class that properly implements all these methods.

7.3.5.5 void MCFClass::CheckPSol (void) `[inline]`

Check that the primal solution returned by the solver is primal feasible. (to within the tolerances set by SetPar(kEps∗∗∗∗) [see above], if any). Also, check that the objective function value is correct.

This method is implemented by the base class, using the above methods for collecting the solutions and the methods of the next section for reading the data of the problem; as such, they will work for any derived class that properly implements all these methods.

7.3.5.6 virtual void MCFClass::ChgCost (Index *arc*, cCNumber *NCost*) `[pure virtual]`

Change the cost of the i-th arc.

Note

changing the costs of arcs that ∗do not exist∗ is ∗not allowed∗; only arcs which have not been closed/deleted [see CloseArc() / DelArc() below and LoadNet() above about C_INF costs] can be touched with these methods.

7.3.5.7 virtual void MCFClass::ChgCosts (cCRow *NCost*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Change the arc costs. In particular, change the costs that are:

- listed in into the vector of indices ‘nms’ (ordered in increasing sense and Inf<Index>()-terminated),
- ∗and∗ whose name belongs to the interval [‘strt’, ‘stp’).

That is, if strt <= nms[i] < stp, then the nms[i]-th cost will be changed to NCost[i]. If nms == NULL (as the default), ∗all∗ the entries in the given range will be changed; if stp > MCFm(), then the smaller bound is used.

Note

changing the costs of arcs that ∗do not exist∗ is ∗not allowed∗; only arcs which have not been closed/deleted [see CloseArc() / DelArc() below and LoadNet() above about C_INF costs] can be touched with these methods.

7.3.5.8 virtual void MCFClass::ChgDfct (Index *node*, cFNumber *NDfct*) `[pure virtual]`

Change the deficit of the i-th node.

Note that, in ChgDfct[s](), node "names" (i, strt/ stp or those contained in nms[]) go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit can be changed e.g. with ChgDfcts(0 , new_deficit).

Note

changing the capacities of nodes that ∗do not exist∗ is ∗not allowed∗; only nodes that have not been deleted [see DelNode() below] can be touched with these methods.

7.3.5.9 virtual void MCFClass::ChgDfcts (cFRow *NDfct*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Change the node deficits. In particular, change the deficits that are:

- listed in into the vector of indices ‘nms’ (ordered in increasing sense and Inf<Index>()-terminated),
- ∗and∗ whose name belongs to the interval [‘strt’, ‘stp’).

That is, if strt <= nms[i] < stp, then the nms[i]-th deficit will be changed to NDfct[i]. If nms == NULL (as the default), ∗all∗ the entries in the given range will be changed; if stp > MCFn(), then the smaller bound is used.

Note that, in ChgDfcts(), node "names" (strt, stp or those contained in nms[]) go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit can be changed e.g. with ChgDfcts(&new_deficit , NULL , 0 , 1).

Note

changing the capacities of nodes that ∗do not exist∗ is ∗not allowed∗; only nodes that have not been deleted [see DelNode() below] can be touched with these methods.

7.3.5.10 virtual void MCFClass::ChgQCoef (Index *arc*, cCNumber *NQCoef*) `[inline, virtual]`

Parameters

NQCoef Change the quadratic coefficient of the cost of the i-th arc.

Note that the method is ∗not∗ pure virtual: an implementation is provided for "pure linear" MCF solvers that only work with all zero quadratic coefficients.

Note

changing the costs of arcs that ∗do not exist∗ is ∗not allowed∗; only arcs which have not been closed/deleted [see CloseArc() / DelArc() below and LoadNet() above about C_INF costs] can be touched with these methods.

7.3.5.11 virtual void MCFClass::ChgQCoef (cCRow *NQCoef* = `NULL`, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf<Index>`()`) `[inline, virtual]`

Parameters

stp Change the quadratic coefficients of the arc costs. In particular, change the coefficients that are:

- listed in into the vector of indices ‘nms’ (ordered in increasing sense and Inf<Index>()-terminated),
- ∗and∗ whose name belongs to the interval [‘strt’, ‘stp’).

That is, if strt <= nms[i] < stp, then the nms[i]-th cost will be changed to NCost[i]. If nms == NULL (as the default), ∗all∗ the entries in the given range will be changed; if stp > MCFm(), then the smaller bound is used. If NQCoef == NULL, all the specified coefficients are set to zero.

Note that the method is ∗not∗ pure virtual: an implementation is provided for "pure linear" MCF solvers that only work with all zero quadratic coefficients.

Note

changing the costs of arcs that ∗do not exist∗ is ∗not allowed∗; only arcs which have not been closed/deleted [see CloseArc() / DelArc() below and LoadNet() above about C_INF costs] can be touched with these methods.

7.3.5.12 virtual void MCFClass::ChgUCap (Index *arc*, cFNumber *NCap*) `[pure virtual]`

Change the capacity of the i-th arc.

Note

changing the capacities of arcs that ∗do not exist∗ is ∗not allowed∗; only arcs that have not been closed/deleted [see CloseArc() / DelArc() below and LoadNet() above about C_INF costs] can be touched with these methods.

7.3.5.13 virtual void MCFClass::ChgUCaps (cFRow *NCap*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Change the arc capacities. In particular, change the capacities that are:

- listed in into the vector of indices 'nms' (ordered in increasing sense and Inf<Index>()-terminated),
- ∗and∗ whose name belongs to the interval ['strt', 'stp').

That is, if strt <= nms[i] < stp, then the nms[i]-th capacity will be changed to NCap[i]. If nms == NULL (as the default), ∗all∗ the entries in the given range will be changed; if stp > MCFm(), then the smaller bound is used.

Note

changing the capacities of arcs that ∗do not exist∗ is ∗not allowed∗; only arcs that have not been closed/deleted [see CloseArc() / DelArc() below and LoadNet() above about C_INF costs] can be touched with these methods.

7.3.5.14 virtual void MCFClass::CloseArc (cIndex *name*) `[pure virtual]`

"Close" the arc 'name'. Although all the associated information (name, cost, capacity, end and start node) is kept, the arc is removed from the problem until OpenArc(i) [see below] is called.

"closed" arcs always have 0 flow, but are otherwise counted as any other arc; for instance, MCFm() does ∗not∗ decrease as an effect of a call to CloseArc(). How this closure is implemented is solver-specific.

Implemented in RelaxIV.

7.3.5.15 virtual void MCFClass::DelArc (cIndex *name*) `[pure virtual]`

Delete the arc 'name'. Unlike "closed" arcs, all the information associated with a deleted arc is lost and 'name' is made available as a name for new arcs to be created with AddArc() [see below].

Il furthermore 'name' is the last arc, the number of arcs as reported by MCFm() is reduced by at least one, until the m-th arc is not a deleted one. Otherwise, the flow on the arc is always ensured to be 0.

Implemented in RelaxIV.

7.3.5.16 virtual void MCFClass::DelNode (cIndex *name*) `[pure virtual]`

Delete the node 'name'.

For any value of 'name', all incident arcs to that node are closed [see CloseArc() above] (∗not∗ Deleted, see DelArc() below) and the deficit is set to zero.

Il furthermore 'name' is the last node, the number of nodes as reported by MCFn() is reduced by at least one, until the n-th node is not a deleted one.

Implemented in RelaxIV.

7.3.5.17 template<class T > bool MCFClass::ETZ (T *x*, const T *eps*) `[inline, protected]`

true if flow x is equal to zero (possibly considering tolerances).

7.3.5.18 void MCFClass::GetPar (int *par*, double & *val*) `[inline, virtual]`

This method returns one of the integer parameter of the algorithm.

Parameters

par is the parameter to return [see SetPar(double) for comments];

val upon return, it will contain the value of the parameter.

The base class implementation handles the parameters kEpsFlw, kEpsDfct, kEpsCst, and kMaxTime.

7.3.5.19 void MCFClass::GetPar (int *par*, int & *val*) `[inline, virtual]`

This method returns one of the integer parameter of the algorithm.

Parameters

par is the parameter to return [see SetPar(int) for comments];

val upon return, it will contain the value of the parameter.

The base class implementation handles the parameters kMaxIter and kReopt.

Reimplemented in RelaxIV.

7.3.5.20 template<class T > bool MCFClass::GEZ (T *x*, const T *eps*) `[inline, protected]`

true if flow x is greater than or equal to zero (possibly considering tolerances).

7.3.5.21 template<class T > bool MCFClass::GT (T *x*, T *y*, const T *eps*) `[inline, protected]`

true if flow x is greater than flow y (possibly considering tolerances).

7.3.5.22 template<class T > bool MCFClass::GTZ (T *x*, const T *eps*) `[inline, protected]`

true if flow x is greater than zero (possibly considering tolerances).

7.3.5.23 virtual bool MCFClass::HaveNewPi (void) `[inline, virtual]`

Return true if a different (approximately) optimal dual solution is available. If the method returns true, then any subsequent call to (any form of) MCFGetPi() will return a different dual solution, and MCFGetRC() [see below] will return the corresponding reduced costs. The new solution need not be optimal (although, ideally, it has to be "good); this can be checked by comparing its objective function value, that will be returned by a call to MCFGetDFO() [see below].

Any subsequent call of HaveNewPi() that returns true produces a new solution, until the first that returns false; from then on, no new solutions will be generated until something changes in the problem's data.

Note that a default implementation of HaveNewPi() is provided which is good for those solvers that only produce one optimal dual solution.

7.3.5.24 virtual bool MCFClass::HaveNewX (void) `[inline, virtual]`

Return true if a different (approximately) optimal primal solution is available. If the method returns true, then any subsequent call to (any form of) MCFGetX() will return a different primal solution w.r.t. the one that was being returned ∗before∗ the call to HaveNewX(). This solution need not be optimal (although, ideally, it has to be "good); this can be checked by comparing its objective function value, that will be returned by a call to MCFGetFO() [see below].

Any subsequent call of HaveNewX() that returns true produces a new solution, until the first that returns false; from then on, no new solutions will be generated until something changes in the problem's data.

Note that a default implementation of HaveNewX() is provided which is good for those solvers that only produce one optimal primal solution.

7.3.5.25 virtual bool MCFClass::IsClosedArc (cIndex *name*) `[pure virtual]`

IsClosedArc() returns true if and only if the arc 'name' is closed.

Implemented in RelaxIV.

7.3.5.26 virtual bool MCFClass::IsDeletedArc (cIndex *name*) `[pure virtual]`

Return true if and only if the arc 'name' is deleted. It should only be called with name < MCFm(), as every other arc is deleted by definition.

Implemented in RelaxIV.

7.3.5.27 template<class T > bool MCFClass::LEZ (T *x*, const T *eps*) `[inline, protected]`

true if flow x is less than or equal to zero (possibly considering tolerances).

7.3.5.28 void MCFClass::LoadDMX (istream & *DMXs*, bool *IsQuad* = `false`) `[inline, virtual]`

Read a MCF instance in DIMACS standard format from the istream. The format is the following. The first line must be

p min <number of="" nodes>=""> <number of="" arcs>="">

Then the node definition lines must be found, in the form

n <node number>=""> <node supply>="">

Not all nodes need have a node definition line; these are given zero supply, i.e., they are transhipment nodes (supplies are the inverse of deficits, i.e., a node with positive supply is a source node). Finally, the arc definition lines must be found, in the form

a <start node>=""> <end node>=""> <lower bound>=""> <upper bound>=""> <flow cost>="">

There must be exactly <number of="" arcs>=""> arc definition lines in the file.

This method is ∗not∗ pure virtual because an implementation is provided by the base class, using the LoadNet() method (which ∗is∗ pure virtual). However, the method ∗is∗ virtual to allow derived classes to implement more efficient versions, should they have any reason to do so.

Note

Actually, the file format accepted by LoadDMX (at least in the base class implementation) is more general than the DIMACS standard format, in that it is allowed to mix node and arc definitions in any order, while the DIMACS file requires all node information to appear before all arc information.
Other than for the above, this method is assumed to allow for quadratic∗ Dimacs files, encoding for convex quadratic separable Min Cost Flow instances. This is a simple extension where each arc descriptor has a sixth field, <quadratic cost>="">. The provided istream is assumed to be quadratic Dimacs file if IsQuad is true, and a regular linear Dimacs file otherwise.

7.3.5.29 virtual void MCFClass::LoadNet (cIndex *nmx* = `0`, cIndex *mmx* = `0`, cIndex *pn* = `0`, cIndex *pm* = `0`, cFRow *pU* = `NULL`, cCRow *pC* = `NULL`, cFRow *pDfct* = `NULL`, cIndex_Set *pSn* = `NULL`, cIndex_Set *pEn* = `NULL`) `[pure virtual]`

Inputs a new network.

The parameters nmx and mmx are the new max number of nodes and arcs, possibly overriding those set in the constructor [see above], altough at the likely cost of memory allocation and deallocation. Passing nmx == mmx == 0 is intended as a signal to the solver to deallocate everything and wait for new orders; in this case, all the other parameters are ignored.

Otherwise, in principle all the other parameters have to be provided. Actually, some of them may not be needed for special classes of MCF problems (e.g., costs in a MaxFlow problem, or start/end nodes in a problem defined over a graph with fixed topology, such as a complete graph). Also, passing NULL is allowed to set default values.

The meaning of the parameters is the following:

- pn is the current number of nodes of the network (<= nmax).
- pm is the number of arcs of the network (<= mmax).
- pU is the m-vector of the arc upper capacities; capacities must be nonnegative, but can in principle be infinite (== F_INF); passing pU == NULL means that all capacities are infinite;
- pC is the m-vector of the arc costs; costs must be finite (< C_INF); passing pC == NULL means that all costs must be 0.
- pDfct is the n-vector of the node deficits; source nodes have negative deficits and sink nodes have positive deficits; passing pDfct == NULL means that all deficits must be 0 (a circulation problem);
- pSn is the m-vector of the arc starting nodes; pSn == NULL is in principle not allowed, unless the topology of the graph is fixed;
- pEn is the m-vector of the arc ending nodes; same comments as for pSn.

Note that node "names" in the arrays pSn and pEn must go from 1 to pn if the macro USANAME0 [see above] is set to 0, while they must go from 0 to pn - 1 if USANAME0 is set to 1. In both cases, however, the deficit of the first node is read from the first (0-th) position of pDfct, that is if USANAME0 == 0 then the deficit of the node with name 'i' is read from pDfct[i - 1].

The data passed to LoadNet() can be used to specify that the arc 'i' must not "exist" in the problem. This is done by passing pC[i] == C_INF; solvers which don't read costs are forced to read them in order to check this, unless they provide alternative solver-specific ways to accomplish the same tasks. These arcs are "closed", as for the effect of CloseArc() [see below]. "invalid" costs (== C_INF) are set to 0 in order to being subsequently capable of "opening" them back with OpenArc() [see below]. The way in which these non-existent arcs are phisically dealt with is solver-specific; in some solvers, for instance, this could be obtained by simply putting their capacity to zero. Details about these issues should be found in the interface of derived classes.

Note that the quadratic part of the objective function, if any, is not dealt with in LoadNet(); it can only be separately provided with ChgQCoef() [see below]. By default, the problem is linear, i.e., all coefficients of the second-order terms in the objective function are assumed to be zero.

Implemented in RelaxIV.

7.3.5.30 template<class T > bool MCFClass::LT (T *x*, T *y*, const T *eps*) `[inline, protected]`

true if flow x is less than flow y (possibly considering tolerances).

7.3.5.31 template<class T > bool MCFClass::LTZ (T *x*, const T *eps*) `[inline, protected]`

true if flow x is less than zero (possibly considering tolerances).

7.3.5.32 virtual void MCFClass::MCFArcs (Index_Set *Startv*, Index_Set *Endv*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Write the starting (tail) and ending (head) nodes of the arcs in Startv[] and Endv[]. If nms == NULL, then the information relative to all arcs is written into Startv[] and Endv[], otherwise Startv[i] and Endv[i] contain the information relative to arc nms[i] (nms[] must be Inf<Index>()-terminated).

The parameters 'strt' and 'stp' allow to restrict the output of the method to all and only the arcs 'i' with strt <= i < min(MCFm() , stp). 'strt' and 'stp' work in "&&" with nms; that is, if nms != NULL then only the values corresponding to arcs which are ∗both∗ in nms and whose index is in the correct range are returned.

Startv or Endv can be NULL, meaning that only the other information is required.

Note

If USENAME0 == 0 then the returned node names will be in the range 1 .. n, while if USENAME0 == 1 the returned node names will be in the range 0 .. n - 1.
If the graph is "dynamic", be careful to use MCFn() e MCFm() to properly choose the dimension of nodes and arcs arrays.

7.3.5.33 virtual CNumber MCFClass::MCFCost (cIndex *i*) `[pure virtual]`

Return the cost of the i-th arc.

Implemented in RelaxIV.

7.3.5.34 virtual cCRow MCFClass::MCFCosts (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the arc costs. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented in RelaxIV.

7.3.5.35 virtual void MCFClass::MCFCosts (CRow *Costv*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Write the arc costs into Costv[]. If nms == NULL, then all the costs are written, otherwise Costv[i] contains the information relative to arc nms[i] (nms[] must be Inf<Index>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the arcs ‘i’ with strt <= i < min(MCFm() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != NULL then only the values corresponding to arcs which are ∗both∗ in nms and whose index is in the correct range are returned.

7.3.5.36 virtual FNumber MCFClass::MCFDfct (cIndex *i*) `[pure virtual]`

Return the deficit of the i-th node.

Note

Here node "names" go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit is returned by MCFDfct(0).

Implemented in RelaxIV.

7.3.5.37 virtual cFRow MCFClass::MCFDfcts (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the node deficits. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Note

Here node "names" go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit is contained in MCFDfcts()[0].

Reimplemented in RelaxIV.

7.3.5.38 virtual void MCFClass::MCFDfcts (FRow *Dfctv*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Write the node deficits into Dfctv[]. If nms == NULL, then all the defcits are written, otherwise Dfctvv[i] contains the information relative to node nms[i] (nms[] must be Inf<Index>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the nodes ‘i’ with strt <= i < min(MCFn() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != NULL then only the values corresponding to nodes which are ∗both∗ in nms and whose index is in the correct range are returned.

Note

Here node "names" (strt and stp, those contained in nms[] or ‘i’ in MCFDfct()) go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit is returned by MCFDfcts(Dfctv , NULL , 0 , 1).

7.3.5.39 virtual Index MCFClass::MCFENde (cIndex *i*) `[pure virtual]`

Return the ending (head) node of the arc 'i'.

Note

If USENAME0 == 0 then the returned node names will be in the range 1 .. n, while if USENAME0 == 1 the returned node names will be in the range 0 .. n - 1.

Implemented in RelaxIV.

7.3.5.40 virtual cIndex_Set MCFClass::MCFENdes (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the ending (head) nodes for each arc. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented in RelaxIV.

7.3.5.41 virtual FONumber MCFClass::MCFGetDFO (void) `[inline, virtual]`

Return the objective function value of the dual solution currently returned by MCFGetPi() / MCFGetRC(). This value (possibly) changes after any call to HaveNewPi() that returns true. The relations between MCFGetStatus() and MCFGetDFO() are analogous to these of MCFGetFO(), except that a finite value corresponding to kStopped must be a lower bound on the optimal objective function value (typically, the objective function value one dual feasible solution).

A default implementation is provided for MCFGetDFO(), which is good for MCF solvers where the primal and dual optimal solution values always are identical (except if the problem is unfeasible/unbounded).

7.3.5.42 virtual FONumber MCFClass::MCFGetFO (void) `[pure virtual]`

Return the objective function value of the primal solution currently returned by MCFGetX().

If MCFGetStatus() == kOK, this is guaranteed to be the optimal objective function value of the problem (to within the optimality tolerances), but only prior to any call to HaveNewX() that returns true. MCFGetFO() typically returns Inf<FONumber>() if MCFGetStatus() == kUnfeasible and

- Inf<FONumber>() if MCFGetStatus() == kUnbounded. If MCFGetStatus() == kStopped and MCFGetFO() returns a finite value, it must be an upper bound on the optimal objective function value (typically, the objective function value of one primal feasible solution).

Implemented in RelaxIV.

7.3.5.43 virtual cCRow MCFClass::MCFGetPi (void) `[inline, virtual]`

Return a read-only pointer to an internal data structure containing the node potentials. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented in RelaxIV.

7.3.5.44 virtual void MCFClass::MCFGetPi (CRow *P*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Writes the optimal node potentials in the vector P[]. If nms == NULL, the node potential of node 'i' (i in 0 .. n - 1) is written in P[i] (note that here node names always start from zero, regardless to the value of USENAME0). If nms != NULL, it must point to a vector of indices in 0 .. n - 1 (ordered in increasing sense and Inf<Index>()-terminated), and the node potential of nms[i] is written in P[i]. Note that, unlike MCFGetX() above, nms is an ∗input∗ of the method.

The parameters 'strt' and 'stp' allow to restrict the output of the method to all and only the nodes 'i' with strt <= i < min(MCFn() , stp). 'strt' and 'stp' work in "&&" with nms; that is, if nms != NULL then only the values corresponding to nodes which are ∗both∗ in nms[] and whose index is in the correct range are returned.

7.3.5.45 virtual CNumber MCFClass::MCFGetRC (cIndex *i*) `[pure virtual]`

Return the reduced cost of the i-th arc. This information should be cheapily available in most implementations.

Note

the output of MCFGetRC() will change after any call to HaveNewPi() [see above] which returns true.

Implemented in RelaxIV.

7.3.5.46 virtual cCRow MCFClass::MCFGetRC (void) `[inline, virtual]`

Return a read-only pointer to an internal data structure containing the reduced costs. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Note

the output of MCFGetRC() will change after any call to HaveNewPi() [see above] which returns true.

Reimplemented in RelaxIV.

7.3.5.47 virtual void MCFClass::MCFGetRC (CRow *CR*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Write the reduced costs corresponding to the current dual solution in RC[]. If nms == NULL, the reduced cost of arc 'i' (i in 0 .. m - 1) is written in RC[i]; if nms != NULL, it must point to a vector of indices in 0 .. m - 1 (ordered in increasing sense and Inf<Index>()-terminated), and the reduced cost of arc nms[i] is written in RC[i]. Note that, unlike MCFGetX() above, nms is an ∗input∗ of the method.

The parameters 'strt' and 'stp' allow to restrict the output of the method to all and only the arcs 'i' with strt <= i < min(MCFm() , stp). 'strt' and 'stp' work in "&&" with nms; that is, if nms != NULL then only the values corresponding to arcs which are ∗both∗ in nms[] and whose index is in the correct range are returned.

Note

the output of MCFGetRC() will change after any call to HaveNewPi() [see above] which returns true.

7.3.5.48 virtual MCFStatePtr MCFClass::MCFGetState (void) `[inline, virtual]`

Save the state of the MCF solver. The MCFClass interface supports the notion of saving and restoring the state of the MCF solver, such as the current/optimal basis in a simplex solver. The "empty" class MCFState is defined as a placeholder for state descriptions.

MCFGetState() creates and returns a pointer to an object of (a proper derived class of) class MCFState which describes the current state of the MCF solver.

Reimplemented in RelaxIV.

7.3.5.49 int MCFClass::MCFGetStatus (void) `[inline]`

Returns an int describing the current status of the MCF solver. Possible return values are:

- kUnSolved SolveMCF() has not been called yet, or the data of the problem has been changed since the last call;
- kOK optimization has been carried out succesfully;
- kStopped optimization have been stopped before that the stopping conditions of the solver applied, e.g. because of the maximum allowed number of "iterations" [see SetPar(int)] or the maximum allowed time [see SetPar(double)] has been reached; this is not necessarily an error, as it might just be required to re-call SolveMCF() giving it more "resources" in order to solve the problem;
- kUnfeasible if the current MCF instance is (primal) unfeasible;
- kUnbounded if the current MCF instance is (primal) unbounded (this can only happen if the solver actually allows F_INF capacities, which is nonstandard in the interface);
- kError if there was an error during the optimization; this typically indicates that computation cannot be resumed, although solver-dependent ways of dealing with solver-dependent errors may exist.

MCFClass has a protected `int member status that` can be used by derived classes to hold status information and that is returned by the standard implementation of this method. Note that `status is` an `int and` not an `enum` , and that an `int is` returned by this method, in order to allow the derived classes to extend the set of return values if they need to do so.

7.3.5.50 virtual Index MCFClass::MCFGetUnbCycl (Index_Set *Pred*, Index_Set *ArcPred*) `[inline, virtual]`

Return an unboundedness certificate. In an unbounded MCF problem, unboundedness can always be reduced to the existence of a directed cycle with negative cost and all arcs having infinite capacity. When detecting unboundedness, MCF solvers are typically capable to provide one such cycle. This information can be useful, and this method is provided for getting it. It can be called only if MCFGetStatus() == kUnbounded, and writes in Pred[] and ArcPred[], respectively, the node and arc predecessor function of the cycle. That is, if node ‘i’ belongs to the cycle then ‘Pred[i]’ contains the name of the predecessor of ‘j’ of ‘i’ in the cycle (note that node names depend on USENAME0), and ‘ArcPred[i]’ contains the index of the arc joining the two (note that in general there may be multiple copies of each arc). Entries of the vectors for nodes not belonging to the cycle are in principle undefined, and the name of one node belonging to the cycle is returned by the method. Note that if there are multiple cycles with negative costs this method will return just one of them (finding the cycle with most negative cost is an NO-hard problem), although solvers should be able to produce cycles with "large negative" cost.

However, not all solvers may be (easily) capable of providing this information; thus, returning Inf<Index>() is allowed, as in the base class implementation, to signify that this information is not available.

7.3.5.51 virtual FNumber MCFClass::MCFGetUnfCut (Index_Set *Cut*) `[inline, virtual]`

Return an unfeasibility certificate. In an unfeasible MCF problem, unfeasibility can always be reduced to the existence of a cut (subset of nodes of the graph) such as either:

- the inverse of the deficit of the cut (the sum of all the deficits of the nodes in the cut) is larger than the forward capacity of the cut (sum of the capacities of forward arcs in the cut); that is, the nodes in the cut globally produce more flow than can be routed to sinks outside the cut;

- the deficit of the cut is larger than the backward capacity of the cut (sum of the capacities of backward arcs in the cut); that is, the nodes in the cut globally require more flow than can be routed to them from sources outside the cut.

When detecting unfeasibility, MCF solvers are typically capable to provide one such cut. This information can be useful - typically, the only way to make the problem feasible is to increase the capacity of at least one of the forward/backward arcs of the cut -, and this method is provided for getting it. It can be called only if MCFGetStatus() == kUnfeasible, and should write in Cut the set of names of nodes in the unfeasible cut (note that node names depend on USENAME0), Inf<Index>()-terminated, returning the deficit of the cut (which allows to distinguish which of the two cases above hold). In general, no special properties can be expected from the returned cut, but solvers should be able to provide e.g. "small" cuts.

However, not all solvers may be (easily) capable of providing this information; thus, returning 0 (no cut) is allowed, as in the base class implementation, to signify that this information is not available.

7.3.5.52 virtual cFRow MCFClass::MCFGetX (void) `[inline, virtual]`

Return a read-only pointer to an internal data structure containing the flow solution in "dense" format. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented in RelaxIV.

7.3.5.53 virtual void MCFClass::MCFGetX (FRow *F*, Index_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Write the optimal flow solution in the vector F[]. If nms == NULL, F[] will be in "dense" format, i.e., the flow relative to arc 'i' (i in 0 .. m - 1) is written in F[i]. If nms != NULL, F[] will be in "sparse" format, i.e., the indices of the nonzero elements in the flow solution are written in nms (that is then Inf<Index>()-terminated) and the flow value of arc nms[i] is written in F[i]. Note that nms is not∗ guaranteed to be ordered. Also, note that, unlike MCFGetRC() and MCFGetPi() [see below], nms is an ∗output∗ of the method.

The parameters 'strt' and 'stp' allow to restrict the output of the method to all and only the arcs 'i' with strt <= i < min(MCFm() , stp).

7.3.5.54 Index MCFClass::MCFm (void) `[inline]`

Return the number of arcs in the current graph. The implementation of the method in the base class returns the protected fields `m`, which is provided for derived classes to hold this information.

7.3.5.55 Index MCFClass::MCFmmax (void) `[inline]`

Return the maximum number of arcs for this instance of MCFClass. The implementation of the method in the base class returns the protected fields `mmax`, which is provided for derived classes to hold this information.

7.3.5.56 Index MCFClass::MCFn (void) `[inline]`

Return the number of nodes in the current graph. The implementation of the method in the base class returns the protected fields `n`, which is provided for derived classes to hold this information.

7.3.5.57 Index MCFClass::MCFnmax (void) `[inline]`

Return the maximum number of nodes for this instance of MCFClass. The implementation of the method in the base class returns the protected fields `nmax`, which is provided for derived classes to hold this information.

7.3.5.58 virtual void MCFClass::MCFPutState (MCFStatePtr *S*) `[inline, virtual]`

Restore the solver to the state in which it was when the state ‘S’ was created with MCFGetState() [see above].

The typical use of this method is the following: a MCF problem is solved and the "optimal state" is set aside. Then the problem changes and it is re-solved. Then, the problem has to be changed again to a form that is close to the original one but rather different from the second one (think of a long backtracking in a Branch & Bound) to which the current "state" referes. Then, the old optimal state can be expected to provide a better starting point for reoptimization [see ReOptimize() below].

Note, however, that the state is only relative to the optimization process, i.e., this operation is meaningless if the data of the problem has changed in the meantime. So, if a state has to be used for speeding up reoptimization, the following has to be done:

- first, the data of the solver is brought back to ∗exactly∗ the same as it was at the moment where the state ‘S’ was created (prior than this operation a ReOptimize(false) call is probably advisable);
- then, MCFPutState() is called (and ReOptimize(true) is called);
- only afterwards the data of the problem is changed to the final state and the problem is solved.

A "put state" operation does not "deplete" the state, which can therefore be used more than once. Indeed, a state is constructed inside the solver for each call to MCFGetState(), but the solver never deletes statuses; this has to be done on the outside when they are no longer needed (the solver must be "resistent" to deletion of the state at any moment).

Since not all the MCF solvers reoptimize (efficiently enough to make these operations worth), an "empty" implementation that does nothing is provided by the base class.

Reimplemented in RelaxIV.

7.3.5.59 virtual cCRow MCFClass::MCFQCoef (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the arc costs. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready (such as "pure linear" MCF solvers that only work with all zero quadratic coefficients) does not need to implement the method.

7.3.5.60 virtual CNumber MCFClass::MCFQCoef (cIndex *i*) `[inline, virtual]`

Parameters

i Return the quadratic coefficients of the cost of the i-th arc. Note that the method is ∗not∗ pure virtual: an implementation is provided for "pure linear" MCF solvers that only work with all zero quadratic coefficients.

7.3.5.61 virtual void MCFClass::MCFQCoef (CRow *Qv*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf<Index>`()`) `[inline, virtual]`

Parameters

stp Write the quadratic coefficients of the arc costs into Qv[]. If nms == NULL, then all the costs are written, otherwise Costv[i] contains the information relative to arc nms[i] (nms[] must be Inf<Index>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the arcs ‘i’ with strt <= i < min(MCFm() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != NULL then only the values corresponding to arcs which are ∗both∗ in nms and whose index is in the correct range are returned.

Note that the method is ∗not∗ pure virtual: an implementation is provided for "pure linear" MCF solvers that only work with all zero quadratic coefficients.

7.3.5.62 virtual Index MCFClass::MCFSNde (cIndex *i*) `[pure virtual]`

Return the starting (tail) node of the arc ‘i’.

Note

If USENAME0 == 0 then the returned node names will be in the range 1 .. n, while if USENAME0 == 1 the returned node names will be in the range 0 .. n - 1.

Implemented in RelaxIV.

7.3.5.63 virtual cIndex_Set MCFClass::MCFSNdes (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the starting (tail) nodes for each arc. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented in RelaxIV.

7.3.5.64 virtual FNumber MCFClass::MCFUCap (cIndex *i*) `[pure virtual]`

Return the capacity of the i-th arc.

Implemented in RelaxIV.

7.3.5.65 virtual cFRow MCFClass::MCFUCaps (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the arc capacities. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented in RelaxIV.

7.3.5.66 virtual void MCFClass::MCFUCaps (FRow *UCapv*, cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`())` `[pure virtual]`

Write the arc capacities into UCapv[]. If nms == NULL, then all the capacities are written, otherwise UCapv[i] contains the information relative to arc nms[i] (nms[] must be Inf<Index>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the arcs ‘i’ with strt <= i < min(MCFm() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != NULL then only the values corresponding to arcs which are ∗both∗ in nms and whose index is in the correct range are returned.

7.3.5.67 virtual void MCFClass::OpenArc (cIndex *name*) `[pure virtual]`

Restore the previously closed arc ‘name’. It is an error to open an arc that has not been previously closed.

Implemented in RelaxIV.

7.3.5.68 virtual void MCFClass::PreProcess (void) `[inline, virtual]`

Extract a smaller/easier equivalent MCF problem. The data of the instance is changed and the easier one is solved instead of the original one. In the MCF case, preprocessing may involve reducing bounds, identifying disconnected components of the graph etc. However, proprocessing is solver-specific.

This method can be implemented by derived classes in their solver-specific way. Preprocessing may reveal unboundedness or unfeasibility of the problem; if that happens, PreProcess() should properly set the ‘status’ field, that can then be read with MCFGetStatus() [see below].

Note that preprocessing may destroy all the solution information. Also, it may be allowed to change the data of the problem, such as costs/capacities of the arcs.

A valid preprocessing is doing nothing, and that’s what the default implementation of this method (that is ∗not∗ pure virtual) does.

Reimplemented in RelaxIV.

7.3.5.69 virtual void MCFClass::SetMCFTime (bool *TimeIt* = `true`) `[inline, virtual]`

Allocate an OPTtimers object [see OPTtypes.h] to be used for timing the methods of the class. The time can be read with TimeMCF() [see below]. By default, or if SetMCFTime(false) is called, no timing is done. Note that, since all the relevant methods ot the class are pure virtual, MCFClass can only manage the OPTtimers object, but it is due to derived classes to actually implement the timing.

Note

time accumulates over the calls: calling SetMCFTime(), however, resets the counters, allowing to time specific groups of calls.
of course, setting kMaxTime [see SetPar() above] to any nonzero value has no effect unless SetMCF-Time(true) has been called.

7.3.5.70 void MCFClass::SetPar (int *par*, double *val*) `[inline, virtual]`

Set float parameters of the algorithm.

Parameters

par is the parameter to be set; the enum MCFParam can be used, but 'par' is an int (every enum is an int) so that the method can be extended by derived classes for the setting of their parameters

value is the value to assign to the parameter.

The base class implementation handles these parameters:

- kEpsFlw: sets the tolerance for controlling if the flow on an arc is zero to val. This also sets the tolerance for controlling if a node deficit is zero (see kEpsDfct) to val $*<$ max number of nodes $>$; this value should be safe for graphs in which any node has less than $<$ max number of nodes $>$ adjacent nodes, i.e., for all graphs but for very dense ones with "parallel arcs"

- kEpsDfct: sets the tolerance for controlling if a node deficit is zero to val, in case a better value than that autmatically set by kEpsFlw (see above) is available (e.g., val $*$ k would be good if no node has more than k neighbours)

- kEpsCst: sets the tolerance for controlling if the reduced cost of an arc is zero to val. A feasible solution satisfying eps-complementary slackness, i.e., such that

  $$RC[i,j] < -eps \Rightarrow X[i,j] = U[ij]$$

  and

  $$RC[i,j] > eps \Rightarrow X[i,j] == 0,$$

  is known to be (eps $*$ n)-optimal.

- kMaxTime: sets the max time (in seconds) in which the MCF Solver can find an optimal solution (default 0, which means no limit).

7.3.5.71 void MCFClass::SetPar (int *par*, int *val*) `[inline, virtual]`

Set integer parameters of the algorithm.

Parameters

par is the parameter to be set; the enum MCFParam can be used, but 'par' is an int (every enum is an int) so that the method can be extended by derived classes for the setting of their parameters

value is the value to assign to the parameter.

The base class implementation handles these parameters:

- kMaxIter: the max number of iterations in which the MCF Solver can find an optimal solution (default 0, which means no limit)

- kReopt: tells the solver if it has to reoptimize. The implementation in the base class sets a flag, the protected `bool` field `Senstv`; if true (default) this field instructs the MCF solver to to try to exploit the information about the latest optimal solution to speedup the optimization of the current problem, while if the field is false the MCF solver should restart the optimization "from scratch" discarding any previous information. Usually reoptimization speeds up the computation considerably, but this is not always true, especially if the data of the problem changes a lot.

Reimplemented in RelaxIV.

7.3.5.72 virtual void MCFClass::SolveMCF (void) `[pure virtual]`

Solver of the Min Cost Flow Problem. Attempts to solve the MCF instance currently loaded in the object.

Implemented in RelaxIV.

7.3.5.73 double MCFClass::TimeMCF (void) `[inline]`

Like TimeMCF(double,double) [see above], but returns the total time.

7.3.5.74 void MCFClass::TimeMCF (double & *t_us*, double & *t_ss*) `[inline]`

Time the code. If called within any of the methods of the class that are "actively timed" (this depends on the subclasses), this method returns the user and sistem time (in seconds) since the start of that method. If methods that are actively timed call other methods that are actively timed, TimeMCF() returns the (...) time since the beginning of the outer∗ actively timed method. If called outside of any actively timed method, this method returns the (...) time spent in all the previous executions of all the actively timed methods of the class.

Implementing the proper calls to MCFt->Start() and MCFt->Stop() is due to derived classes; these should at least be placed at the beginning and at the end, respectively, of SolveMCF() and presumably the Chg∗∗∗() methods, that is, at least these methods should be "actively timed".

7.3.5.75 void MCFClass::WriteMCF (ostream & *oStrm*, int *frmt* = `0`) `[inline, virtual]`

Write the current MCF problem to an ostream. This may be useful e.g. for debugging purposes.

The base MCFClass class provides output in two different formats, depending on the value of the parameter frmt:

- kDimacs the problem is written in DIMACS standard format, read by most MCF codes available;

- kMPS the problem is written in the "modern version" (tab-separated) of the MPS format, read by most LP/MIP solvers;

- kFWMPS the problem is written in the "old version" (fixed width fields) of the MPS format; this is read by most LP/MIP solvers, but some codes still require the old format.

The implementation of WriteMCF() in the base class uses all the above methods for reading the data; as such it will work for any derived class that properly implements this part of the interface, but it may not be very efficient. Thus, the method is virtual to allow the derived classes to either re-implement WriteMCF() for the above two formats in a more efficient way, and/or to extend it to support other solver-specific formats.

Note

None of the above two formats supports quadratic MCFs, so if nonzero quadratic coefficients are present, they are just ignored.

Reimplemented in RelaxIV.

7.3.6 Member Data Documentation

7.3.6.1 int MCFClass::MaxIter `[protected]`

max number of iterations in which MCF Solver can find an optimal solution (0 = no limits)

7.3.6.2 double MCFClass::MaxTime `[protected]`

max time (in seconds) in which MCF Solver can find an optimal solution (0 = no limits)

7.3.6.3 int MCFClass::status `[protected]`

return status, see the comments to MCFGetStatus() above. Note that the variable is defined int to allow derived classes to return their own specialized status codes

The documentation for this class was generated from the following file:

- MCFClass.h

7.4 MCFClass::MCFException Class Reference

```
#include <MCFClass.h>
```

Public Member Functions

- **MCFException** (const char ∗const msg=0)
- const char ∗ **what** (void) const throw ()

7.4.1 Detailed Description

Small class for exceptions. Derives from std::exception implementing the virtual method what() -- and since what is virtual, remember to always catch it by reference (catch exception &e) if you want the thing to work. MCFException class are thought to be of the "fatal" type, i.e., problems for which no solutions exists apart from aborting the program. Other kinds of exceptions can be properly handled by defining derived classes with more information.

The documentation for this class was generated from the following file:

- MCFClass.h

7.5 MCFClass::MCFState Class Reference

`#include <MCFClass.h>`

Inherited by RelaxIV::RIVState.

7.5.1 Detailed Description

Base class for representing the internal state of the MCF algorithm.

The documentation for this class was generated from the following file:

- MCFClass.h

7.6 OPTrand Class Reference

`#include <OPTUtils.h>`

Public Member Functions

- OPTrand (void)

    constructor of the class

- double rand (void)
- void srand (long seed)

    Seeds the random generator for this instance of OPTrand.

7.6.1 Detailed Description

Provide a common interface to the different random generators that are available in different platforms.

7.6.2 Member Function Documentation

7.6.2.1 double OPTrand::rand (void) `[inline]`

Returns a random number uniformly distributed in [0, 1).

Note

each object of class OPTrand has its own sequence, so that multiple OPTrand objects being used within the same program do not interfere with each other (as opposed to what C random routines would do).

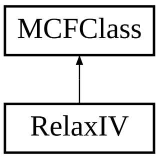
The documentation for this class was generated from the following file:

- OPTUtils.h

7.7 RelaxIV Class Reference

`#include <RelaxIV.h>`

Inheritance diagram for RelaxIV:



Classes

- class **RIVState**

Public Types

- enum MCFRParam { kAuction = kLastParam }
- enum RIVFlFrmt { kCLP = kMPS + 1, kRIV }
- typedef bool ∗ Bool_Vec

  vector of booleans

Public Member Functions

- RelaxIV (cIndex nmx=0, cIndex mmx=0)
- void LoadNet (cIndex nmx=0, cIndex mmx=0, cIndex pn=0, cIndex pm=0, cFRow pU=NULL, cCRow pC=NULL, cFRow pDfct=NULL, cIndex_Set pSn=NULL, cIndex_Set pEn=NULL)
- virtual void SetPar (int par, int val)
- virtual void GetPar (int par, int &val)
- void PreProcess (void)
- void SolveMCF (void)
- void **MCFGetX** (register FRow F, register Index_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- cFRow MCFGetX (void)
- void **MCFGetRC** (register CRow CR, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- cCRow MCFGetRC (void)
- CNumber MCFGetRC (cIndex i)
- void MCFGetPi (register CRow P, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- cCRow MCFGetPi (void)
- RelaxIV::FONumber MCFGetFO (void)
- MCFClass::MCFStatePtr MCFGetState (void)
- void MCFPutState (MCFClass::MCFStatePtr S)
- void **MCFArcs** (register Index_Set Startv, register Index_Set Endv, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- Index MCFSNde (cIndex i)

- Index MCFENde (cIndex i)
- cIndex_Set MCFSNdes (void)
- cIndex_Set MCFENdes (void)
- void **MCFCosts** (register CRow Costv, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- CNumber MCFCost (cIndex i)
- cCRow MCFCosts (void)
- void **MCFUCaps** (register FRow UCapv, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- FNumber MCFUCap (cIndex i)
- cFRow MCFUCaps (void)
- void **MCFDfcts** (register FRow Dfctv, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- FNumber MCFDfct (cIndex i)
- cFRow MCFDfcts (void)
- void WriteMCF (ostream &oStrm, int frmt=0)
- void **ChgCosts** (register cCRow NCost, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- void **ChgCost** (register Index arc, cCNumber NCost)
- void **ChgDfcts** (register cFRow NDfct, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- void **ChgDfct** (register Index nod, cFNumber NDfct)
- void **ChgUCaps** (register cFRow NCap, register cIndex_Set nms=NULL, cIndex strt=0, Index stp=Inf< Index >())
- void **ChgUCap** (register Index arc, cFNumber NCap)
- void CloseArc (cIndex name)
- bool IsClosedArc (cIndex name)
- void DelNode (cIndex name)
- void OpenArc (cIndex name)
- Index AddNode (cFNumber aDfct)
- void ChangeArc (cIndex name, cIndex nSS=Inf< Index >(), cIndex nEN=Inf< Index >())
- void DelArc (cIndex name)
- bool IsDeletedArc (cIndex name)
- Index AddArc (cIndex Start, cIndex End, cFNumber aU, cCNumber aC)
- int MCFiter ()

  Total number of (single-node or multinode) iterations.

- int MCFaug ()

  Number of flow augmentations.

7.7.1 Detailed Description

The RelaxIV class derives from the abstract base class MCFClass, thus sharing its (standard) interface, and implements a Relaxation algorithm for solving (Linear) Min Cost Flow problems.

7.7.2 Member Enumeration Documentation

7.7.2.1 enum RelaxIV::MCFRParam

Public enum describing the possible parameters of the MCF solver, "extended" from MCFClass::MCFParam, to be used with the methods SetPar() and GetPar().

Enumerator:

kAuction	crash initialization

7.7.2.2 enum RelaxIV::RIVFlFrmt

Public enum describing the more file formats in RelaxIV::WriteMCF().

Enumerator:

kCLP	the "LP" format
kRIV	RelaxIV-specific format.

7.7.3 Constructor & Destructor Documentation

7.7.3.1 RelaxIV::RelaxIV (cIndex *nmx* = `0`, cIndex *mmx* = `0`)

Constructor of the class, as in MCFClass::MCFClass().

7.7.4 Member Function Documentation

7.7.4.1 MCFClass::Index RelaxIV::AddArc (cIndex *Start*, cIndex *End*, cFNumber *aU*, cCNumber *aC*) `[virtual]`

Add the new arc (Start , End) with cost aC and capacity aU, returning its name. Inf<Index>() is returned if there is no room for a new arc. Remember that arc names go from 0 to mmax - 1.

Implements MCFClass.

7.7.4.2 MCFClass::Index RelaxIV::AddNode (cFNumber *aDfct*) `[virtual]`

Add a new node with deficit aDfct, returning its name. Inf<Index>() is returned if there is no room for a new node. Remember that the node names are either { 0 .. nmax - 1 } or { 1 .. nmax }, depending on the value of USENAME0.

Implements MCFClass.

7.7.4.3 void RelaxIV::ChangeArc (cIndex *name*, cIndex *nSN* = Inf< Index >`()`, cIndex *nEN* = Inf< Index >`()`) `[virtual]`

Change the starting and/or ending node of arc ‘name’ to nSN and nEN. Each parameter being Inf<Index>() means to leave the previous starting or ending node untouched. When this method is called ‘name’ can be either the name of a "normal" arc or that of a "closed" arc [see CloseArc() above]: in the latter case, at the end of ChangeArc() the arc is ∗still closed∗, and it remains so until OpenArc(name) [see above] is called.

Implements MCFClass.

7.7.4.4 void RelaxIV::CloseArc (cIndex *name*) `[virtual]`

"Close" the arc ‘name’. Although all the associated information (name, cost, capacity, end and start node) is kept, the arc is removed from the problem until OpenArc(i) [see below] is called.

"closed" arcs always have 0 flow, but are otherwise counted as any other arc; for instance, MCFm() does ∗not∗ decrease as an effect of a call to CloseArc(). How this closure is implemented is solver-specific.

Implements MCFClass.

7.7.4.5 void RelaxIV::DelArc (cIndex *name*) `[virtual]`

Delete the arc ‘name’. Unlike "closed" arcs, all the information associated with a deleted arc is lost and ‘name’ is made available as a name for new arcs to be created with AddArc() [see below].

Il furthermore ‘name’ is the last arc, the number of arcs as reported by MCFm() is reduced by at least one, until the m-th arc is not a deleted one. Otherwise, the flow on the arc is always ensured to be 0.

Implements MCFClass.

7.7.4.6 void RelaxIV::DelNode (cIndex *name*) `[virtual]`

Delete the node ‘name’.

For any value of ‘name’, all incident arcs to that node are closed [see CloseArc() above] (∗not∗ Deleted, see DelArc() below) and the deficit is set to zero.

Il furthermore ‘name’ is the last node, the number of nodes as reported by MCFn() is reduced by at least one, until the n-th node is not a deleted one.

Implements MCFClass.

7.7.4.7 void RelaxIV::GetPar (int *par*, int & *val*) `[inline, virtual]`

This method returns one of the integer parameter of the algorithm.

Parameters

par is the parameter to return [see SetPar(int) for comments];

val upon return, it will contain the value of the parameter.

Apart from the parameters of the base class, this method handles kAuction.

Reimplemented from MCFClass.

7.7.4.8 bool RelaxIV::IsClosedArc (cIndex *name*) `[inline, virtual]`

IsClosedArc() returns true if and only if the arc ‘name’ is closed.

Implements MCFClass.

7.7.4.9 bool RelaxIV::IsDeletedArc (cIndex *name*) `[inline, virtual]`

Return true if and only if the arc ‘name’ is deleted. It should only be called with name $<$ MCFm(), as every other arc is deleted by definition.

Implements MCFClass.

7.7.4.10 void RelaxIV::LoadNet (cIndex *nmx* = `0`, cIndex *mmx* = `0`, cIndex *pn* = `0`, cIndex *pm* = `0`, cFRow *pU* = `NULL`, cCRow *pC* = `NULL`, cFRow *pDfct* = `NULL`, cIndex_Set *pSn* = `NULL`, cIndex_Set *pEn* = `NULL`) `[virtual]`

Inputs a new network, as in MCFClass::LoadNet().

Arcs with pC[i] == Inf<CNumber>() do not "exist". If DYNMC_MCF_RIV > 0, these arcs are "closed".

If DYNMC_MCF_RIV == 0 but SAME_GRPH_RIV > 0, these arcs are dealt with exactly like as if they had pU[i] == 0, i.e., as "normal" arcs with zero capacity. These arcs can be put back into the formulation by simply changing their capacity (and cost). Note that, however, this is less efficient than eliminating them explicitly from the problem.

If DYNMC_MCF_RIV == 0 and SAME_GRPH_RIV == 0, these arcs are just removed from the formulation. However, they have some sort of a "special status" (after all, if the user wants to remove them completely he/she can just change the data), in that they are still counted into the number of arcs of the graph and they will always have 0 flow and Inf<CNumber>() reduced cost as "closed" or "deleted" arcs.

If SAME_GRPH_RIV == 1 (==> DYNMC_MCF_RIV < 3), pSn and pEn passed to any instance after the first that is constructed are ignored, thus, pSn == pEn == NULL is allowed in this case. Note that pn and pm are not∗ ignored.

Implements MCFClass.

7.7.4.11 MCFClass::CNumber RelaxIV::MCFCost (cIndex *i*) `[inline, virtual]`

Return the cost of the i-th arc.

Implements MCFClass.

7.7.4.12 MCFClass::cCRow RelaxIV::MCFCosts (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the arc costs. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented from MCFClass.

7.7.4.13 MCFClass::FNumber RelaxIV::MCFDfct (cIndex *i*) `[inline, virtual]`

Return the deficit of the i-th node.

Note

Here node "names" go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit is returned by MCFDfct(0).

Implements MCFClass.

7.7.4.14 MCFClass::cFRow RelaxIV::MCFDfcts (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the node deficits. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Note

Here node "names" go from 0 to n - 1, regardless to the value of USENAME0; hence, if USENAME0 == 0 then the first node is "named 1", but its deficit is contained in MCFDfcts()[0].

Reimplemented from MCFClass.

7.7.4.15 MCFClass::Index RelaxIV::MCFENde (cIndex *i*) `[inline, virtual]`

Return the ending (head) node of the arc ‘i’.

Note

If USENAME0 == 0 then the returned node names will be in the range 1 .. n, while if USENAME0 == 1 the returned node names will be in the range 0 .. n - 1.

Implements MCFClass.

7.7.4.16 MCFClass::cIndex_Set RelaxIV::MCFENdes (void) `[inline, virtual]`

Same meaning as MCFClass::MCFENdes().

Note

MCFENdes() returns a pointers to a (read-only) vector containing the arc end nodes ∗only if USENAME0 == 0∗; otherwise, it returns NULL.

Reimplemented from MCFClass.

7.7.4.17 MCFClass::FONumber RelaxIV::MCFGetFO (void) `[virtual]`

Return the objective function value of the primal solution currently returned by MCFGetX().

If MCFGetStatus() == kOK, this is guaranteed to be the optimal objective function value of the problem (to within the optimality tolerances), but only prior to any call to HaveNewX() that returns true. MCFGetFO() typically returns Inf<FONumber>() if MCFGetStatus() == kUnfeasible and

- Inf<FONumber>() if MCFGetStatus() == kUnbounded. If MCFGetStatus() == kStopped and MCFGetFO() returns a finite value, it must be an upper bound on the optimal objective function value (typically, the objective function value of one primal feasible solution).

Implements MCFClass.

7.7.4.18 MCFClass::cCRow RelaxIV::MCFGetPi (void) `[virtual]`

Same meaning as MCFClass::MCFGetPi().

Note

if MCFGetPi(void) returns a pointer, this is a pointer to a static shared data structure that can be corrupted by calls to methods (e.g. MCFGetPi() itself) for other active instances of RelaxIV.

Reimplemented from MCFClass.

7.7.4.19 void RelaxIV::MCFGetPi (register CRow *P*, register cIndex_Set *nms* = `NULL`, cIndex *strt* = `0`, Index *stp* = Inf< Index >`()`)

Same meaning as MCFClass::MCFGetPi().

Note

if both AUCTION == 0 and DYNMC_MCF_RIV <= 1, no internal memory for the vector of potentials is allocated; hence, even if nms != NULL MCFGetPi(P) first constructs the full vector of potentials in P and then selects only the components in nms. Therefore, memory can be written even after the | nms |-th element of P. For the same reason, memory can be written even after themore the (stp - strt)-th element of P.

7.7.4.20 MCFClass::CNumber RelaxIV::MCFGetRC (cIndex *i*) `[virtual]`

Return the reduced cost of the i-th arc. This information should be cheapily available in most implementations.

Note

the output of MCFGetRC() will change after any call to HaveNewPi() [see above] which returns true.

Implements MCFClass.

7.7.4.21 MCFClass::cCRow RelaxIV::MCFGetRC (void) `[virtual]`

Return a read-only pointer to an internal data structure containing the reduced costs. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Note

the output of MCFGetRC() will change after any call to HaveNewPi() [see above] which returns true.

Reimplemented from MCFClass.

7.7.4.22 MCFClass::MCFStatePtr RelaxIV::MCFGetState (void) `[virtual]`

Same meaning as MCFClass::MCFGetState().

The state of the algorithm is the pair S = (X[] , RC[]) of the arc flows and reduced costs.

Reimplemented from MCFClass.

7.7.4.23 MCFClass::cFRow RelaxIV::MCFGetX (void) `[virtual]`

Return a read-only pointer to an internal data structure containing the flow solution in "dense" format. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented from MCFClass.

7.7.4.24 void RelaxIV::MCFPutState (MCFClass::MCFStatePtr *S*) `[virtual]`

Restore the solver to the state in which it was when the state ‘S’ was created with MCFGetState() [see above].

The typical use of this method is the following: a MCF problem is solved and the "optimal state" is set aside. Then the problem changes and it is re-solved. Then, the problem has to be changed again to a form that is close to the original one but rather different from the second one (think of a long backtracking in a Branch & Bound) to which the current "state" referes. Then, the old optimal state can be expected to provide a better starting point for reoptimization [see ReOptimize() below].

Note, however, that the state is only relative to the optimization process, i.e., this operation is meaningless if the data of the problem has changed in the meantime. So, if a state has to be used for speeding up reoptimization, the following has to be done:

- first, the data of the solver is brought back to ∗exactly∗ the same as it was at the moment where the state ‘S’ was created (prior than this operation a ReOptimize(false) call is probably advisable);
- then, MCFPutState() is called (and ReOptimize(true) is called);
- only afterwards the data of the problem is changed to the final state and the problem is solved.

A "put state" operation does not "deplete" the state, which can therefore be used more than once. Indeed, a state is constructed inside the solver for each call to MCFGetState(), but the solver never deletes statuses; this has to be done on the outside when they are no longer needed (the solver must be "resistent" to deletion of the state at any moment).

Since not all the MCF solvers reoptimize (efficiently enough to make these operations worth), an "empty" implementation that does nothing is provided by the base class.

Reimplemented from MCFClass.

7.7.4.25 MCFClass::Index RelaxIV::MCFSNde (cIndex *i*) `[inline, virtual]`

Return the starting (tail) node of the arc ‘i’.

Note

If USENAME0 == 0 then the returned node names will be in the range 1 .. n, while if USENAME0 == 1 the returned node names will be in the range 0 .. n - 1.

Implements MCFClass.

7.7.4.26 MCFClass::cIndex_Set RelaxIV::MCFSNdes (void) `[inline, virtual]`

Same meaning as MCFClass::MCFSNdes().

Note

MCFSNdes() returns a pointers to a (read-only) vector containing the arc start nodes ∗only if USE-NAME0 == 0∗; otherwise, it returns NULL.

Reimplemented from MCFClass.

7.7.4.27 MCFClass::FNumber RelaxIV::MCFUCap (cIndex *i*) `[inline, virtual]`

Return the capacity of the i-th arc.

Implements MCFClass.

7.7.4.28 MCFClass::cFRow RelaxIV::MCFUCaps (void) `[inline, virtual]`

Return a read-only pointer to an internal vector containing the arc capacities. Since this may ∗not always be available∗, depending on the implementation, this method can (uniformly) return NULL. This is done by the base class already, so a derived class that does not have the information ready does not need to implement the method.

Reimplemented from MCFClass.

7.7.4.29 void RelaxIV::OpenArc (cIndex *name*) `[virtual]`

Restore the previously closed arc 'name'. It is an error to open an arc that has not been previously closed.

Implements MCFClass.

7.7.4.30 void RelaxIV::PreProcess (void) `[virtual]`

If this method is called, a preprocessing phase is performed trying to reduce the arc capacities. This may sometimes help in speeding up the solution of the problem, but may also change the capacities returned by MCFUCap[s]() [see below].

For this method to work properly, arc capacities, node deficits and the topology of the graph must have already been provided with LoadNet() [see above].

This method can be called more than once, for instance whenever the capacities of some arcs or the deficits of some nodes are changed; however, it destroys the provious optimal solution (if any), forcing the algorithm to restart from scratch.

Reimplemented from MCFClass.

7.7.4.31 void RelaxIV::SetPar (int *par*, int *val*) `[inline, virtual]`

Set integer parameters of the algorithm.

Parameters

par is the parameter to be set;

value is the value to assign to the parameter.

Apart from the parameters of the base class, this method handles:

- kAuction: if set to kYes, the auction/shortest paths initialization is used in SolveMCF() to generate the starting solution; if set to kNo (default), then the default initialization based on special single-node relaxation iterations is used instead. Note that this parameter is ∗ignored∗ if AUCTION == 0.

Reimplemented from MCFClass.

7.7.4.32 void RelaxIV::SolveMCF (void) `[virtual]`

Solver of the Min Cost Flow Problem. Attempts to solve the MCF instance currently loaded in the object.

Implements MCFClass.

7.7.4.33 void RelaxIV::WriteMCF (ostream & *oStrm*, int *frmt* = `0`) `[virtual]`

Write the current MCF problem to an ostream. This may be useful e.g. for debugging purposes.

The base MCFClass class provides output in two different formats, depending on the value of the parameter frmt:

- kDimacs the problem is written in DIMACS standard format, read by most MCF codes available;

- kMPS the problem is written in the "modern version" (tab-separated) of the MPS format, read by most LP/MIP solvers;

- kFWMPS the problem is written in the "old version" (fixed width fields) of the MPS format; this is read by most LP/MIP solvers, but some codes still require the old format.

The implementation of WriteMCF() in the base class uses all the above methods for reading the data; as such it will work for any derived class that properly implements this part of the interface, but it may not be very efficient. Thus, the method is virtual to allow the derived classes to either re-implement WriteMCF() for the above two formats in a more efficient way, and/or to extend it to support other solver-specific formats.

Note

None of the above two formats supports quadratic MCFs, so if nonzero quadratic coefficients are present, they are just ignored.

Reimplemented from MCFClass.

The documentation for this class was generated from the following files:

- RelaxIV.h
- RelaxIV.C

Chapter 8

File Documentation

8.1 Main.C File Reference

```
#include <fstream>
#include <sstream>
#include "RelaxIV.h"
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <iomanip>
```

Defines

- #define **MCFSOLVER** RelaxIV
- #define **PRINT_RESULTS** 0

Functions

- template<class T >
  T **ABS** (const T x)
- void **SkipComments** (ifstream &iParam, string &buf)
- void **SetParam** (MCFClass ∗mcf)
- int **main** (int argc, char ∗∗argv)

8.1.1 Detailed Description

Sample Main file to illustrate the use of any solver deriving from MCFClass. By changing just ∗two lines of code∗ and little more (see comment PECULIARITY, if exists) the file works with any derived solver.

An instance of a Min Cost Flow problem in DIMACS standard format is read from file and solved. In addition, the same problem can be written on a file in MPS format.

Version

4.00

Date

30 - 12 - 2009

Author

Alessandro Bertolini
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa

/*--

8.2 MCFClass.h File Reference

```
#include "OPTUtils.h"
#include <iomanip>
#include <sstream>
```

Classes

- class MCFClass
- class MCFClass::Inf< T >
- class MCFClass::Eps< T >
- class MCFClass::MCFException
- class MCFClass::MCFState

Defines

- #define USENAME0 0

8.2.1 Detailed Description

Header file for the abstract base class MCFClass, which defines a standard interface for (linear or convex quadratic separable) Min Cost Flow Problem solvers, to be implemented as derived classes.

Version

3.00

Date

30 - 12 - 2009

Author

Alessandro Bertolini
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Claudio Gentile
Istituto di Analisi di Sistemi e Informatica
Consiglio Nazionale delle Ricerche

8.3 OPTUtils.h File Reference

```
#include <sys/timeb.h>
#include <limits.h>
#include <iostream>
```

Classes

- class OPTrand

Defines

- #define OPT_USE_NAMESPACES 0
- #define OPT_TIMERS 5
- #define OPT_RANDOM 1
- #define **NULL** 0
- #define **CLK_TCK** 100

Functions

- template<class T >
  void DfltdSfInpt (istream ∗iStrm, T &Param, const T Dflt, const char cmntc= '#')

8.3.1 Detailed Description

Small classes are provided for:

- reading the time of a code;
- generating random numbers.

The classes can be adapted to different environments setting a compile-time switch in this file.

Additionally, a function is provided for safely reading parameters out of a stream.

Version

2.63

Date

29 - 08 - 2005

Author

Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa

8.4 RelaxIV.h File Reference

```
#include "MCFClass.h"
```

Classes

- class RelaxIV
- class **RelaxIV::RIVState**

Defines

- #define SAME_GRPH_RIV 0
- #define DYNMC_MCF_RIV 0
- #define AUCTION 0
- #define RELAXIV_STATISTICS 0

8.4.1 Detailed Description

Linear Min Cost Flow problems solver, based on the RelaxIV code by D. Bertsekas. Conforms to the standard (MCF) interface defined in MCFClass.h.

The RelaxIV is based on a primal-dual algorithm which essentially operates as follows: a pseudoflow (a flow vector which satisfies bound and non-negativity constraints but not necessarily flow conservation constraints) is kept which satisfies complementarity slackness conditions with the current vector of potentials; that is, only the flow on arcs whose reduced cost

$$RC[i,j] = C[i,j] - Pi[j] + Pi[i]$$

is zero can be chosen to any value between 0 and the capacity, while arcs with reduced cost < 0 are saturated (fixed to their capacity) and arcs with reduced cost > 0 are empty (fixed to 0).

The algorithm attempts to convert the pseudoflow into a flow (i.e., to satisfy the flow conservation constraints) by essentially running a max-flow algorithm of the augmenting path type. If the flow is found then this is an optimal solution of the problem and the algorithm is stopped. Otherwise, a saturated cut is identified which separates the origins (nodes not yet producing enough flow) to the destinations (nodes not yet consuming enough flow); this cut is used to modify the potentials, thereby creating new arcs with zero reduced cost, which can be used to push further flow from the origins to the destinations. If no such arcs can be created the problem is declared unfeasible. Much care is devoted to stop the max-flow computation as soon as a proof that the set of potentials is not optimal, in order to reach as soon as possible a dual optimal solution, and to re-use all available information to "warm start" the max-flow computation after a change in the potentials.

Warning

The original code has been written for integer data only. By properly setting the flow and cost tolerances [see SetEps$***$() in MCFClass.h] we have always been able to solve any MCF that we could throw at the solver, but in principle this kind of algorithm may fail to converge with nonintegral data, so consider yourselves warned.
A private type SIndex is defined which is intended to hold arc and node indices "with a sign", used to represent orientation. This has to be "in sync" with Index, in the sense that for every unsigned index value in Index, the two signed values should be feasible in SIndex. In other words, either Index is not using at least half of its feasible values, or SIndex has to be a "bigger" data type than Index. The default value for SIndex is int.

Version

1.80

Date

30 - 12 - 2009

Author

(original FORTRAN code)
Dimitri P. Bertsekas
M.I.T.
(C++ porting and polishing)
Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
(C++ porting and polishing)
Claudio Gentile
Istituto di Analisi di Sistemi e Informatica
Consiglio Nazionale delle Ricerche
Viale Manzoni, 30 Roma