The `CQKnPClass` Project

Version: 1.05
Date: 23/12/2012

Antonio Frangioni

Operations Research Group
Dipartimento di Informatica
Università di Pisa

Enrico Gorgone

Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria

Contents

0.1 <H1>The CQKnPClass Project Documentation</H1>

0.1.1 Introduction

This file is a short user manual for the `CQKnPClass` project, a small collection of solvers for Continuous (Convex) Quadratic Knapsack Problems (CQKnP).

0.1.1.1 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 eventually 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.

0.1.1.2 License

This code is provided free of charge under the "GNU Lesser General Public License", see the file doc/LGPL.txt.

0.1.1.3 Release

Current version is: 1.05

Current date is: December 22, 2012

0.1.2 Package description

This release comes out with the following files:

- **CQKnPClass.h:** Contains the declaration of class `CQKnPClass`. It is an *abstract class* with *pure virtual methods*, so that you cannot declare objects of type `CQKnPClass`, but only of derived classes of its. `CQKnPClass` offers a general interface for CQKnP 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 CQKnP solvers distributed in this package conforms with the interface, i.e., they derive from `CQKnPClass`; however, the idea is that applications using this interface would require almost no changes if any other solvers implementing the interface is used. Carefully read the public interface of the class to understand how to use the public methods of the class.

- **CQKnPClone/CQKnPClone.h**: Contains the declarations of class `CQKnPClone`, implementing a "fake" CQKnP solver that takes two "real" ones and does everything on both; useful for testing the solvers (either for correctness or for efficiency) when used within "complex" approaches

- **CQKnPClone/makefile**: "recursive" makefile for the `CQKnPClone` class, see lib and extlib below

- **CQKnPCplex/CQKnPCplex.h**: Contains the declarations of class `CQKnPCplex`, implementing a CQKnP solver conforming to the `CQKnPClass` interface based on calls to the commercial (but now free for academic purposes) `Cplex` solver from IBM. You need to separately obtain a `Cplex` distribution and link it with this code (see QKnPCplex/makefile below) to make it work. This does not make much of a sense in that `Cplex` is far slower than the other options, unless one is interested in checking correctness and efficiency of some CQKnP solver.

- **CQKnPCplex/CQKnPCplex.C**: Contains the implementation of the `CQKnPCplex` class. You should not need to read it.
- **CQKnPCplex/makefile**: "recursive" makefile for the `CQKnPCplex` class, see lib and extlib below
- **doc/LGPL.txt**: Description of the LGPL license.
- **doc/refman.pdf**: Pdf version of the manual.
- **doc/html/***: Html version of the manual.
- **DualCQKnP/DualCQKnP.h**: Contains the declarations of class `DualCQKnP`, implementing a CQKnP solver based on the standard dual-ascent approach. This class derives from `CQKnPClass`, hence most of its interface is defined and discussed in CQKnPClass.h; however, for efficiency it is actually restricted to instances where *all* items have *strictly positive quadratic costs* and *finite bounds* (both lower and upper), so in fact it does not fully implement the interface. Furthermore, a few implementation-dependent details (and compile-time switches) which are worth knowing are described in this file.
- **DualCQKnP/DualCQKnP.C**: Contains the implementation of the `DualCQKnP` class. You should not need to read it.
- **DualCQKnP/makefile**: Makefile for the `DualCQKnP` class.
- **ExDualCQKnP/ExDualCQKnP.h**: Contains the declarations of class `ExDualCQKnP`, derived from `DualCQKnP` (and hence from `CQKnPClass`) and extending it to support all the features of the problem (possibly zero quadratic costs, possibly infinite upper and lower bounds). It is slightly less efficient, so `DualCQKnP` should be preferred for the instances that it can solve.
- **ExDualCQKnP/ExDualCQKnP.C**: Contains the implementation of the `ExDualCQKnP` class. You should not need to read it.
- **ExDualCQKnP/makefile**: Makefile for the `ExDualCQKnP` class.
- **extlib/makefile-libCPX**: the makefile where Cplex path libraries are declared, edit it to insert your own
- **lib/***: Makefiles for creating the libCQK.a library file to be linked by the code using it (in this case the main files in the Main directory); in particular, makefile-o is where the modules that will be part of the library are decided (`CQKnPCplex` comes commented out by default)
- **Main/Main.C**: Contains an example of use of the provided CQKnP solvers: creates pne of them (according to the some macros) and use it to solve a chosen instance read from file.
- **Main/MainRnd.C**: Contains an example of use of the provided CQKnP solvers: creates two of them (according to the some macros) and use them to solve a slew of randomly-generated instances, checking that the results agree (and comparing the running times).
- **Main/makefile**: A makefile for Main.C and MainRnd.C. You can easily decide which Main file to use, while in order to decide which solvers will be in the library you have to work with lib/makefile-o; of course you have to be careful to include these required according to the macro settings in Main[Rnd].C (assuming you care anything about these main files, which is unclear why you should).
- **Main\tests\rnd.sh**: Script file for using MainRnd.C.
- **Main\tests\spp.sh**: Script file for using Main.C together with the generator of one-dimensional Sensor Placement Problems that can be found at `http://www.di.unipi.it/optimize/Data/RDR.html`

0.2 Module Index

0.2.1 Modules

Here is a list of all modules:

0.3 Class Index

0.3.1 Class Hierarchy

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

0.4 Class Index

0.4.1 Class List

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

0.5 File Index

0.5.1 File List

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

0.6 Module Documentation

0.6.1 Compile-time switches in CQKnPClass.h

These macros control some important details of the class interface.

Defines

- #define CQKnPClass_LOG 0

  If CQKnPClass_LOG $>$ 0, data structures and methods to log the activities of the (actual) solver are added to the (abstract) interface.

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.

0.6.1.2 Define Documentation

0.6.1.2.1 #define **CQKnPClass_LOG** 0

If CQKnPClass_LOG $>$ 0, data structures and methods to log the activities of the (actual) solver are added to the (abstract) interface.

0.6.2 Compile-time switches in DualCQKnP.h

These macros control some important details of the class interface.

Defines

- #define DualCQKnP_WHCH_QSORT 1

  If DualCQKnP_WHCH_QSORT == 0, the sort() function of the STL is used, otherwise a hand-made non-recursive quick-sort implementation is used.
- #define DualCQKnP_SANITY_CHECKS 0

  If DualCQKnP_SANITY_CHECKS == 1, sanity checks are done each time the data of the instance changes to pick up clearly bad values.

0.6.2.1 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.

0.6.2.2 Define Documentation

0.6.2.2.1 #define **DualCQKnP_WHCH_QSORT** 1

If DualCQKnP_WHCH_QSORT == 0, the sort() function of the STL is used, otherwise a hand-made non-recursive quick-sort implementation is used.

Note

Other than the performance impact, this choice has a consequence on the thread-safety of the code. If DualCQKnP_WHCH_QSORT == 1, some temporary data structures are created that are shared among all "active" instances of DualCQKnP to save on space in the (frequent) case where many instances are simultaneously in memory. This is *not* thread-safe, while DualCQKnP_WHCH_QSORT == 0 is.

0.6.2.2.2 #define **DualCQKnP_SANITY_CHECKS** 0

If DualCQKnP_SANITY_CHECKS == 1, sanity checks are done each time the data of the instance changes to pick up clearly bad values.

Otherwise, the user will have to be extra careful to avoid them.

0.7 Class Documentation

0.7.1 CQKnPClass::CQKException Class Reference

Small class for exceptions.

```
#include <CQKnPClass.h>
```

Inherits exception.

0.7.1.1 Detailed Description

Small class for exceptions.

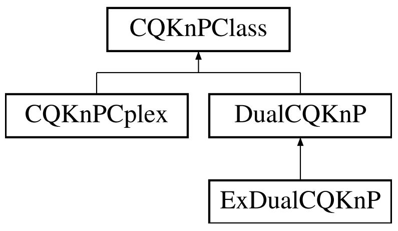
Derives from std::exception implementing the virtual method what() -- and since what is virtual, remember to always catch it by reference (catch exception &e) if you want the thing to work. CQKException class are thought to be of the "fatal" type, i.e., problems for which no solutions exists apart from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

0.7.2 CQKnPClass Class Reference

The class ContQKsk defines an interface for Continuous Quadratic Knapsack problems (CQKnP) solvers.

```
#include <CQKnPClass.h>
```

Inheritance diagram for CQKnPClass:



Classes

- class CQKException

  Small class for exceptions.
- class Inf

  Small class using std::numeric_limits to extract the "infinity" value of a basic type (just use Inf<type>()).

Public Types

Public types

- enum CQKStatus {
  kOK = 0, kStopped = 1, kUnfeasible = 2, kUnbounded = 3,
  kError = 4, kUnSolved = 5 }

  Public enum describing the possible status of the MCF solver.

Public Member Functions

- virtual void LoadSet (const int pn=0, const double ∗pC=0, const double ∗pD=0, const double ∗pA=0, const double ∗pB=0, const double pV=0, const bool sns=true)=0

Inputs a new CQKnP instance.

- virtual void ReadInstance (std::istream &inFile, bool RBV=false)

  Read the instance from file.
- virtual void SetEps (const double eps=1e-6)=0

  Defines the precision required to construct the solution of the knapsack problem.
- virtual CQKStatus SolveKNP (void)=0

  SolveKNP() attempts to solve the current CQKnP instance, returning the status indicating the outcome of the optimization.
- virtual const double ∗ KNPGetX (void)=0

  Return the best feasible solution of the CQKnP found so far (assuming CheckFsb() == true, see above), which is optimal if SolveKNP() has returned kOK (see above).
- virtual double KNPGetPi (void)=0

  Returns the optimal dual multiplier of the knapsack constraint; the returned value is dependable only if SolveKNP() has returned kOK (see above).
- virtual double KNPGetFO (void)=0

  Returns the optimal value of the objective function of the problem.
- virtual int KNPn (void)

  Returns the current number of items.
- virtual void KNPLCosts (double ∗csts, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  The linear costs of the items are written into csts[].
- virtual void KNPQCosts (double ∗csts, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  The quadratic costs of the items are written into csts[].
- virtual double KNPLCost (const int i)=0

  Return the linear cost of the i-th item (i = 0 .
- virtual double KNPQCost (const int i)=0

  Return the quadratic cost of the i-th item (i = 0 .
- virtual void KNPLBnds (double ∗bnds, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  The lower bounds of the items are written into bnds[].
- virtual void KNPUBnds (double ∗bnds, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  The upper bounds of the items are written into bnds[].
- virtual double KNPLBnd (const int i)=0

  Return the lower bound of the i-th item (i = 0 .
- virtual double KNPUBnd (const int i)=0

  Return the upper bound of the i-th item (i = 0 .
- virtual double KNPVlm (void)=0

  Returns the volume 'V' of the knapsack.
- virtual void WriteInstance (std::ostream &oFile, const int precc=16, const int precv=16)

  Write the instance to the provided ostream in the "complete" format read by ReadInstance() [see above].
- virtual void ChgLCosts (const double ∗csts, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  Change the linear cost coefficients that are:
- virtual void ChgQCosts (const double ∗csts, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  Change the quadratic cost coefficients that are:
- virtual void ChgLCost (int i, const double cst)=0

  Change the linear cost coefficient of item i to cst.
- virtual void ChgQCost (int i, const double cst)=0

  Change the quadratic cost coefficient of item i to cst.
- virtual void ChgLBnds (const double ∗bnds, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  Change the item lower bounds that are:
- virtual void ChgUBnds (const double ∗bnds, const int ∗nms=0, int strt=0, int stp=Inf< int >())=0

  Change the item upper bounds that are:
- virtual void ChgLBnd (int i, const double bnd)=0

Change the lower bound of item i to bnd.

- virtual void ChgUBnd (int i, const double bnd)=0

  Change the upper bound of item i to bnd.

- virtual void ChgVlm (const double NVlm)=0

  Change the volume.

Protected Attributes

- int n

  total number of items

- int status

  status of the algorithm: it is an int so that derived classes can use it to encode other information apart from the return value of SolveKNP().

0.7.2.1 Detailed Description

The class ContQKsk defines an interface for Continuous Quadratic Knapsack problems (CQKnP) solvers.

The data of the problem consists of a set of items = (0 , ... , n - 1), where each item i has:

- a linear cost C[i];
- a quadratic cost D[i] $\in R_+$;
- a lower bound A[i] $\in R \cup$ -INF, and
- an upper bound B[i] $\in R \cup$ +INF.

The problem requires to find the most valuable set of items which fit in a knapsack of given volume V, but *partly accepting items is allowed*. The formulation of the problem is therefore:

$$\min \sum_{i \in E} C[i] * X[i] + D[i] * X[i]^2$$

$$\sum_{i \in E} X[i] \leq V (= V)$$

$$A[i] \leq X[i] \leq B[i] \qquad i \in E$$

This is a convex quadratic problem, hence a "easy" one. However it is repeatedly solved as a subproblem in many applications, so a fast solver may ultimately be useful.

0.7.2.2 Member Enumeration Documentation

0.7.2.2.1 enum CQKStatus

Public enum describing the possible status of the MCF solver.

Enumerator:

kOK optimal solution found

kStopped optimization stopped

kUnfeasible problem is unfeasible

kUnbounded problem is unbounded

kError error in the solver

kUnSolved no solution available yet

0.7.2.3 Member Function Documentation

0.7.2.3.1 virtual void **LoadSet** (const int *pn* = `0`, const double ∗ *pC* = `0`, const double ∗ *pD* = `0`, const double ∗ *pA* = `0`, const double ∗ *pB* = `0`, const double *pV* = `0`, const bool *sns* = `true`) `[pure virtual]`

Inputs a new CQKnP instance.

The meaning of the parameters is the following:

- pn is the current number of items of the set: passing pn == 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.
- pC is the n-vector of the linear part of item costs; the linear costs must be finite (- INF < pC[i] < INF); pC == 0 means that all linear costs are 0.
- pD is the n-vector of the quadratic part of item costs; the quadratic costs must be non-negative and finite (0 <= pD[i] < INF); pD == 0 means that all quadratic costs are 0.
- pA is the n-vector of the item lower bounds; lower bound must be < + INF; pA == 0 means that all lower bound are - INF.
- pB is the n-vector of the item upper bounds; upper bounds must be > -INF; pB == 0 means that all upper bounds are + INF.
- pV is the knapsack volume, which must be finite (- INF < pV < + INF).
- sns is the sense of knapsack constraint: true is for an equality constraint (default), false for an inequality (<=) constraint.

This method *must* be called prior to invoking any other method of the class, with the exception of SetKNPLog() and SetEps() (not to mention the constructor, of course).

Referenced by CQKnPClass::ReadInstance().

0.7.2.3.2 void **ReadInstance** (std::istream & *inFile*, bool *RBV* = `false`) `[inline, virtual]`

Read the instance from file.

While virtual, the method is implemented in the base class: the instance data is read in temporary data structures and then LoadSet() [see above] is called using these. Thus, the method will work for any derived class with no modification, since it uses the class-provided implmentation of LoadSet(); however, being virtual it can be re-implemented for maximum efficiency if desired.

The method supports two formats of the file: a "simplified" one and a "complete" one. The simplified format, assumed when RBV is false, is:

<number of items (n)>

for i = 0 to n - 1 <linear cost="" of="" item="" i>="">

for i = 0 to n - 1 <quadratic cost="" of="" item="" i>="">

In this case all lower bounds are zero, all upper bounds are INF, and the volume is 1. Otherwise (RBV == true) the file must continue with

for i = 0 to n - 1 <lower bound="" of="" item="" i>="">

for i = 0 to n - 1 <upper bound="" of="" item="" i>="">

<volume of="" knapsack>="">

Note that this method is "alternative" to LoadSet() (in the sense that the latter is invoked inside), so the object is "ready to use" once this method returns.

References CQKnPClass::LoadSet().

0.7.2.3.3 virtual void SetEps (const double *eps* = 1e-6) [pure virtual]

Defines the precision required to construct the solution of the knapsack problem.

0.7.2.3.4 virtual CQKStatus SolveKNP (void) [pure virtual]

SolveKNP() attempts to solve the current CQKnP instance, returning the status indicating the outcome of the optimization.

Possible values are:

- kUnSolved SolveKNP() 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" have been reached; this is not necessarily an error, as it might just be required to re-call SolveKNP() giving it more "resources" in order to solve the problem;
- kUnfeasible the current CQKnP instance is (primal) unfeasible;
- kUnbounded if the current CQKnP instance is (primal) unbounded: this can only happen if some of the items have +/-INF bounds and zero quadratic cost.
- kError 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 exhist.

0.7.2.3.5 virtual const double∗ KNPGetX (void) [pure virtual]

Return the best feasible solution of the CQKnP found so far (assuming CheckFsb() == true, see above), which is optimal if SolveKNP() has returned kOK (see above).

0.7.2.3.6 virtual double KNPGetPi (void) [pure virtual]

Returns the optimal dual multiplier of the knapsack constraint; the returned value is dependable only if SolveKNP() has returned kOK (see above).

0.7.2.3.7 virtual double KNPGetFO (void) [pure virtual]

Returns the optimal value of the objective function of the problem.

The method typically returns INF if SolveKNP() == kUnfeasible. It returns a finite feasible value if SolveKNP() == kOK or SolveKNP() == kStopped, but in the latter case it depends on the solver whether this is a lower or an upper bound on the optimal value. The return value is undefined in all other cases.

0.7.2.3.8 virtual int KNPn (void) [inline, virtual]

Returns the current number of items.

The class has its protected field for storing this information, so this method is *not* pure virtual, but it is indeed virtual to allow re-definition if needed.

References CQKnPClass::n.

0.7.2.3.9 virtual void KNPLCosts (double ∗ *csts*, const int ∗ *nms* = 0, int *strt* = 0, int *stp* = Inf< int >()) [pure virtual]

The linear costs of the items are written into csts[].

If nms == 0 then all the costs are written into csts[], otherwise cst[i] contains the informations relative to item nms[i] (nms must be Inf<int>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the items ‘i’ with strt <= i < min(KNPn() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != 0 then only the values corresponding to items which are *both* in nms and whose index is in the correct range are returned.

0.7.2.3.10 virtual void **KNPQCosts** (double ∗ *csts*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = **Inf**< `int` >`()`) `[pure virtual]`

The quadratic costs of the items are written into csts[].

If nms == 0 then all the costs are written into csts[], otherwise csts[i] contains the informations relative to item nms[i] (nms must be Inf<int>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the items ‘i’ with strt <= i < min(KNPn() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != 0 then only the values corresponding to items which are *both* in nms and whose index is in the correct range are returned (see above).

0.7.2.3.11 virtual double **KNPLCost** (const int *i*) `[pure virtual]`

Return the linear cost of the i-th item (i = 0 .

. n - 1).

Referenced by CQKnPClass::WriteInstance().

0.7.2.3.12 virtual double **KNPQCost** (const int *i*) `[pure virtual]`

Return the quadratic cost of the i-th item (i = 0 .

. n - 1).

Referenced by CQKnPClass::WriteInstance().

0.7.2.3.13 virtual void **KNPLBnds** (double ∗ *bnds*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = **Inf**< `int` >`()`) `[pure virtual]`

The lower bounds of the items are written into bnds[].

If nms == 0 then all the bounds are written, otherwise bnds[i] contains the information relative to item nms[i] (nms must be Inf<int>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the items ‘i’ with strt <= i < min(KNPn() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != 0 then only the values corresponding to items which are *both* in nms and whose index is in the correct range are returned.

0.7.2.3.14 virtual void **KNPUBnds** (double ∗ *bnds*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = **Inf**< `int` >`()`) `[pure virtual]`

The upper bounds of the items are written into bnds[].

If nms == 0 then all the bounds are written, otherwise bnds[i] contains the information relative to item nms[i] (nms must be Inf<int>()-terminated).

The parameters ‘strt’ and ‘stp’ allow to restrict the output of the method to all and only the items ‘i’ with strt <= i < min(KNPn() , stp). ‘strt’ and ‘stp’ work in "&&" with nms; that is, if nms != 0 then only the values corresponding to items which are *both* in nms and whose index is in the correct range are returned (see above).

0.7.2.3.15 virtual double **KNPLBnd** (const int *i*) `[pure virtual]`

Return the lower bound of the i-th item (i = 0 .

. n - 1).

Referenced by CQKnPClass::WriteInstance().

0.7.2.3.16 virtual double **KNPUBnd** (const int *i*) [pure virtual]

Return the upper bound of the i-th item (i = 0 .

. n - 1).

Referenced by CQKnPClass::WriteInstance().

0.7.2.3.17 virtual double **KNPVlm** (void) [pure virtual]

Returns the volume ‘V’ of the knapsack.

Referenced by CQKnPClass::WriteInstance().

0.7.2.3.18 void **WriteInstance** (std::ostream & *oFile,* const int *precc* = 16*,* const int *precv* = 16) [inline, virtual]

Write the instance to the provided ostream in the "complete" format read by ReadInstance() [see above].

The parameter "precc" and "precv" allow to set the precision (number of decimal digits) of the costs (linear and quadratic) and the bounds (upper and lower) and volume, respectively, when printed to the ostream. Rhe default value is "all digits of a double", which results in pretty large files, but smaller precisions are possible if one e.g. knows that the instance data is integer. Since this can typically be different for costs-related and volume-related information, two different parameters are provided.

While virtual, the method is implemented in the base class: the instance data is read from the object using the class-provided implmentation of KNPLCost(), KNPQCost() and so on. Thus, the method will work for any derived class with no mdification; however, being virtual it can be re-implemented for maximum efficiency if desired.

References CQKnPClass::KNPLBnd(), CQKnPClass::KNPLCost(), CQKnPClass::KNPQCost(), CQKnPClass::KN-PUBnd(), CQKnPClass::KNPVlm(), and CQKnPClass::n.

0.7.2.3.19 virtual void **ChgLCosts** (const double ∗ *csts,* const int ∗ *nms* = 0*,* int *strt* = 0*,* int *stp* = **Inf**< int >()) [pure virtual]

Change the linear cost coefficients that are:

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

That is, if strt <= nms[i] < stp, then the coefficient of the nms[i]-th item will be changed reading from csts[i]. If nms == 0 (as the default), all the entries in the given range will be changed.

0.7.2.3.20 virtual void **ChgQCosts** (const double ∗ *csts,* const int ∗ *nms* = 0*,* int *strt* = 0*,* int *stp* = **Inf**< int >()) [pure virtual]

Change the quadratic cost coefficients that are:

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

That is, if strt <= nms[i] < stp, then the coefficient of the nms[i]-th item will be changed reading from csts[i]. If nms == 0 (as the default), all the entries in the given range will be changed (see above).

0.7.2.3.21 virtual void **ChgLCost** (int *i,* const double *cst*) [pure virtual]

Change the linear cost coefficient of item i to cst.

0.7.2.3.22 virtual void **ChgQCost** (int *i,* const double *cst*) [pure virtual]

Change the quadratic cost coefficient of item i to cst.

0.7.2.3.23 virtual void **ChgLBnds** (const double * *bnds*, const int * *nms* = 0, int *strt* = 0, int *stp* = Inf< int >()) [pure virtual]

Change the item lower bounds that are:

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

That is, if strt <= nms[i] < stp, then the bounds of the nms[i]-th item will be changed reading from bnds[i]. If nms == 0 (as the default), all the entries in the given range will be changed.

0.7.2.3.24 virtual void **ChgUBnds** (const double * *bnds*, const int * *nms* = 0, int *strt* = 0, int *stp* = Inf< int >()) [pure virtual]

Change the item upper bounds that are:

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

That is, if strt <= nms[i] < stp, then the bounds of the nms[i]-th item will be changed reading from bnds[i]. If nms == 0 (as the default), all the entries in the given range will be changed.

0.7.2.3.25 virtual void **ChgLBnd** (int *i*, const double *bnd*) [pure virtual]

Change the lower bound of item i to bnd.

0.7.2.3.26 virtual void **ChgUBnd** (int *i*, const double *bnd*) [pure virtual]

Change the upper bound of item i to bnd.

0.7.2.3.27 virtual void **ChgVlm** (const double *NVlm*) [pure virtual]

Change the volume.

0.7.2.4 Member Data Documentation

0.7.2.4.1 int **status** [protected]

status of the algorithm: it is an int so that derived classes can use it to encode other information apart from the return value of SolveKNP().

0.7.3 CQKnPClone< Master, Slave > Class Template Reference

Class for cross-testing solvers of the Continuous Quadratic Knapsack Problem implemented as derived object of the class CQKnPClass [see CQKnPClass.h].

```
#include <CQKnPClone.h>
```

Public Member Functions

- CQKnPClone (void)

  Construct both the Master and the Slave object.
- void LoadSet (const int pn=0, const double *pC=0, const double *pD=0, const double *pA=0, const double *pB=0, const double pV=0, const bool sns=true)

Load data in both the Master and the Slave object.

- void SetEps (const double eps=1e-6)

  Set tolerance in both the Master and the Slave object.

- CQKnPClass::CQKStatus SolveKNP (void)

  Solve the problem in both the Master and the Slave object, check that the solution status agrees.

- double KNPGetFO (void)

  Recover optimal value in both the Master and the Slave object, check that they agree.

- void ChgLCosts (const double ∗csts, const int ∗nms=0, int strt=0, int stp=Inf< int >())

  Change linear costs in both the Master and the Slave object.

- void ChgQCosts (const double ∗csts, const int ∗nms=0, int strt=0, int stp=Inf< int >())

  Change quadratic costs in both the Master and the Slave object.

- void ChgLCost (int i, const double cst)

  Change a linear cost in both the Master and the Slave object.

- void ChgQCost (int i, const double cst)

  Change a quadratic cost in both the Master and the Slave object.

- void ChgLBnds (const double ∗bnds, const int ∗nms=0, int strt=0, int stp=Inf< int >())

  Change lower bounds in both the Master and the Slave object.

- void ChgUBnds (const double ∗bnds, const int ∗nms=0, int strt=0, int stp=Inf< int >())

  Change upper bounds in both the Master and the Slave object.

- void ChgLBnd (int i, const double bnd)

  Change a lower bound in both the Master and the Slave object.

- void ChgUBnd (int i, const double bnd)

  Change an upper bound in both the Master and the Slave object.

- void ChgVlm (const double NVlm)

  Change volume in both the Master and the Slave object.

- ∼CQKnPClone ()

  Destroy the Slave object together with the Master.

Public Attributes

- Slave ∗ SlvCQK

  pointer to the "slave" object; it is public in order to allow calling the methods of the specialized interface of the slave class

0.7.3.1 Detailed Description

template< class Master, class Slave> class CQKnPClone< Master, Slave >

Class for cross-testing solvers of the Continuous Quadratic Knapsack Problem implemented as derived object of the class CQKnPClass [see CQKnPClass.h].

The class is template over two objects, a "Master" one and a "Slave" one, that are assumed to be both derived from CQKnPClass; it derives from the Master class, and creates an object of the Slave class. Every call to a method is "reflected" on both objects. In some cases, checks are performed that the results agree.

The rationale for having CQKnPClone to derive from the Master class is that some methods (typically, these that read back the data of the problem) do not need to be redefined, as being CQKnPClone derived from Master the Master:: implementation of the method is automatically used (of course this implies that no checks are done that Master and Slave agree on which data they have, which perhaps should be done).

0.7.3.2 Constructor & Destructor Documentation

0.7.3.2.1 CQKnPClone (void) `[inline]`

Construct both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.2.2 ∼CQKnPClone ( ) `[inline]`

Destroy the Slave object together with the Master.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3 Member Function Documentation

0.7.3.3.1 void LoadSet (const int *pn* = `0`, const double ∗ *pC* = `0`, const double ∗ *pD* = `0`, const double ∗ *pA* = `0`, const double ∗ *pB* = `0`, const double *pV* = `0`, const bool *sns* = `true`) `[inline]`

Load data in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.2 void SetEps (const double *eps* = `1e-6`) `[inline]`

Set tolerance in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.3 CQKnPClass::CQKStatus SolveKNP (void) `[inline]`

Solve the problem in both the Master and the Slave object, check that the solution status agrees.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.4 double KNPGetFO (void) `[inline]`

Recover optimal value in both the Master and the Slave object, check that they agree.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.5 void ChgLCosts (const double ∗ *csts*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = `Inf<int>()`) `[inline]`

Change linear costs in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.6 void ChgQCosts (const double ∗ *csts*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = `Inf<int>()`) `[inline]`

Change quadratic costs in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.7 void ChgLCost (int *i*, const double *cst*) `[inline]`

Change a linear cost in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.8 void ChgQCost (int *i*, const double *cst*) `[inline]`

Change a quadratic cost in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.9 void **ChgLBnds** (const double ∗ *bnds*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = `Inf<int>()`) `[inline]`

Change lower bounds in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.10 void **ChgUBnds** (const double ∗ *bnds*, const int ∗ *nms* = `0`, int *strt* = `0`, int *stp* = `Inf<int>()`) `[inline]`

Change upper bounds in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.11 void **ChgLBnd** (int *i*, const double *bnd*) `[inline]`

Change a lower bound in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.12 void **ChgUBnd** (int *i*, const double *bnd*) `[inline]`

Change an upper bound in both the Master and the Slave object.

References CQKnPClone< Master, Slave >::SlvCQK.

0.7.3.3.13 void **ChgVlm** (const double *NVlm*) `[inline]`

Change volume in both the Master and the Slave object.

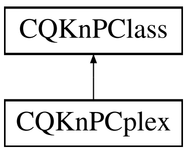
References CQKnPClone< Master, Slave >::SlvCQK.

0.7.4 CQKnPCplex Class Reference

Solver of Continuous Quadratic Knapsack Problems (CQKnP) based on calls to the `Cplex` callable library.

`#include <CQKnPCplex.h>`

Inheritance diagram for CQKnPCplex:



Public Member Functions

- CQKnPCplex (const int alg=0, const double eps=1e-6)

  The parameter alg controls which algorithm CPLEX uses to solve continuous quadratic knapsack problem.

- void SetCplexParam (int whichparam, double value)

  The first two methods allow to set the many algorithmic parameters of Cplex; see the documentation of CP-Xsetintparam() and CPXsetdblparam() in the Cplex manual for details.

- void SetAlg (const int alg=0)

  Allows to change the algorithm to be used in the next calls to SolveKNP(); see the comments to the constructor for details.

0.7.4.1 Detailed Description

Solver of Continuous Quadratic Knapsack Problems (CQKnP) based on calls to the `Cplex` callable library.

Derives from CQKnpClass and therefore it conforms to its interface.

0.7.4.2 Constructor & Destructor Documentation

0.7.4.2.1 CQKnPCplex (const int *alg* = `0`, const double *eps* = `1e-6`)

The parameter alg controls which algorithm CPLEX uses to solve continuous quadratic knapsack problem. Possible values of this parameter are:

```
0   [default] et CPLEX choose.
1   Primal simplex;
2   Dual simplex

[See the documentation of CPX_PARAM_QPMETHOD in the Cplex manual for
 details.]

The parameter Eps defines the convergence tolerance required to construct
the solution [default value is 1e-6].

[See the documentation of CPX_PARAM_BAREPCOMP in the Cplex manual for
 details.]

The choices can be changed at any time with SetAlg() and SetEps(),
respectively [see below].
```

0.7.4.3 Member Function Documentation

0.7.4.3.1 void SetCplexParam (int *whichparam*, double *value*) `[inline]`

The first two methods allow to set the many algorithmic parameters of Cplex; see the documentation of CPXsetintparam() and CPXsetdblparam() in the Cplex manual for details.

0.7.4.3.2 void SetAlg (const int *alg* = `0`) `[inline]`

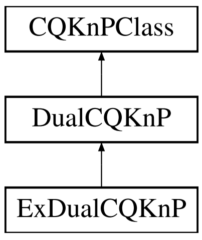
Allows to change the algorithm to be used in the next calls to SolveKNP(); see the comments to the constructor for details.

0.7.5 DualCQKnP Class Reference

Solver of Continuous Quadratic Knapsack Problems (CQKnP) based on the standard formulation of the dual problem as a piecewise-convex problem in the unique multiplier of the knapsack constraint and the corresponding obvious dual-ascent approach.

`#include <DualCQKnP.h>`

Inheritance diagram for DualCQKnP:



Public Member Functions

- DualCQKnP (const bool sort=true, const double eps=1e-6)

  The most important operation for solving the CQKnP with a dual method is the sorting of the items for nondecreasing elements.
- void SetSort (const bool WhchSrt=false)

  Allows to change the sorting procedures to be used in the next calls to SolveKNP(); see the comments to the constructor for details.

Protected Attributes

- double $*$ A

  vector of lower bounds
- double $*$ B

  vector of upper bounds
- double $*$ C

  vector of linear costs
- double $*$ D

  vector of quadratic costs
- double McB

  volume value
- bool sense

  sense of knapsack constraint
- double LB

  lower bound on dual variable
- double UB

  upper bound on dual variable
- int $*$ I

  optimal ordering
- int nSort

  how many elements we have to sort
- double $*$ OV

  values upon which to order
- double $*$ XSol

  primal solution
- double muStar

  optimal dual solution
- bool WSort

  which sorting procedure is used
- double OptVal

  The Optimal Value.
- double DefEps

  precision required to construct the solution

Static Protected Attributes

- static int $*$ QSStck

  the stack to simulate recursive calls in QS
- static int InstCntr

  number of active instances
- static int maxvl

  max value of items

0.7.5.1 Detailed Description

Solver of Continuous Quadratic Knapsack Problems (CQKnP) based on the standard formulation of the dual problem as a piecewise-convex problem in the unique multiplier of the knapsack constraint and the corresponding obvious dual-ascent approach.

This class is restricted to instances where *all* items have *strictly positive* quadratic costs and *finite* bounds (both lower and upper).

Derives from CQKnpClass and therefore it *mostly* conforms to its interface, except for refusing to solve instances without the required characteristics.

0.7.5.2 Constructor & Destructor Documentation

0.7.5.2.1 **DualCQKnP** (const bool *sort* = `true`, const double *eps* = `1e-6`)

The most important operation for solving the CQKnP with a dual method is the sorting of the items for nondecreasing elements.

2 ∗ A[i] ∗ D[i] + C[i] and 2 ∗ B[i] ∗ D[i] + C[i].

If the knapsack is a large one, this can be (relatively) time-consuming. Different sort procedures can be better in different situations, and the parameter 'sort' allows to decide which among the available sorting procedures has to be used. Possible values of this parameter are:

false Bubble Sort: this is O(n^2) on average, but it can be very fast - O(n) - if reoptimizing from a previous problem where the order was not very different (e.g., only few costs have changed).

true [default] Quick Sort: this is O(n lg n) on average and pretty efficient in practice, but can be very slow - O(n^2) - if the vector is already (almost) ordered, e.g. when reoptimizing from a previous problem where only few costs have changed.

The parameter Eps defines the precision required to construct the solution [default value is 1e-6].

The choices can be changed at any time with SetSort() and SetEps(), respectively [see below].

0.7.5.3 Member Function Documentation

0.7.5.3.1 void **SetSort** (const bool *WhchSrt* = `false`) `[inline]`

Allows to change the sorting procedures to be used in the next calls to SolveKNP(); see the comments to the constructor for details.

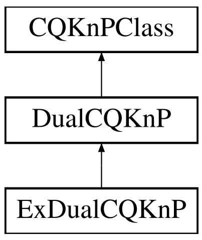
References DualCQKnP::WSort.

0.7.6 ExDualCQKnP Class Reference

Continuous Quadratic Knapsack Problems (CQKnP) solver derived from the DualCQKnP class (and therefore from CQKnPClass) and extending it, using the same standard dual-ascent approach, to non-negative quadratic costs and extended real bounds.

```
#include <ExDualCQKnP.h>
```

Inheritance diagram for ExDualCQKnP:



0.7.6.1 Detailed Description

Continuous Quadratic Knapsack Problems (CQKnP) solver derived from the DualCQKnP class (and therefore from CQKnPClass) and extending it, using the same standard dual-ascent approach, to non-negative quadratic costs and extended real bounds.

0.7.7 CQKnPClass::Inf< T > Class Template Reference

Small class using std::numeric_limits to extract the "infinity" value of a basic type (just use Inf<type>()).

```
#include <CQKnPClass.h>
```

0.7.7.1 Detailed Description

template<typename T>class CQKnPClass_di_unipi_it::CQKnPClass::Inf< T >

Small class using std::numeric_limits to extract the "infinity" value of a basic type (just use Inf<type>()).

0.8 File Documentation

0.8.1 CQKnPClass.h File Reference

Definition of the abstract base class CQKnPClass, which implements a standard interface for Continuous Quadratic Knapsack Problem (CQKnP) solvers to be implemented as derived classes.

```
#include <limits>
#include <exception>
#include <iostream>
```

Classes

- class CQKnPClass

  The class ContQKsk defines an interface for Continuous Quadratic Knapsack problems (CQKnP) solvers.
- class CQKnPClass::CQKException

  Small class for exceptions.
- class CQKnPClass::Inf< T >

  Small class using std::numeric_limits to extract the "infinity" value of a basic type (just use Inf<type>()).

Defines

- #define CQKnPClass_LOG 0

  If CQKnPClass_LOG > 0, data structures and methods to log the activities of the (actual) solver are added to the (abstract) interface.

0.8.1.1 Detailed Description

Definition of the abstract base class CQKnPClass, which implements a standard interface for Continuous Quadratic Knapsack Problem (CQKnP) solvers to be implemented as derived classes.

Version

1.05

Date

22 - 12 - 2012

Author

Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Enrico Gorgone
Dipartimento di Elettronica Informatica e Sistemistica
Universita' della Calabria

0.8.2 CQKnPClone.h File Reference

Definition of implementation of the template class CQKnPClone.

```
#include "CQKnPClass.h"
```

Classes

- class CQKnPClone< Master, Slave >

  Class for cross-testing solvers of the Continuous Quadratic Knapsack Problem implemented as derived object of the class CQKnPClass [see CQKnPClass.h].

0.8.2.1 Detailed Description

Definition of implementation of the template class CQKnPClone.

Version

1.00

Date

23 - 12 - 2012

Author

Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Enrico Gorgone
Operations Research Group
Dipartimento di Elettronica Informatica e Sistemistica
Universita' della Calabria

0.8.3 CQKnPCplex.h File Reference

Continuous Quadratic Knapsack Problems (CQKnP) solver based on calls to the CPLEX callable library.

```
#include "CQKnPClass.h"
#include "cplex.h"
```

Classes

- class CQKnPCplex

  Solver of Continuous Quadratic Knapsack Problems (CQKnP) based on calls to the `Cplex` *callable library.*

0.8.3.1 Detailed Description

Continuous Quadratic Knapsack Problems (CQKnP) solver based on calls to the CPLEX callable library. Conforms to the standard interface for CQKnP solver defined by the abstract base class CQKnpClass.

Version

1.04

Date

22 - 12 - 2012

Author

Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Enrico Gorgone
Dipartimento di Elettronica Informatica e Sistemistica
Universita' della Calabria

0.8.4 DualCQKnP.h File Reference

Basis Continuous Quadratic Knapsack Problems (CQKnP) solver based on the standard dual-ascent approach.

```
#include "CQKnPClass.h"
```

Classes

- class DualCQKnP

  Solver of Continuous Quadratic Knapsack Problems (CQKnP) based on the standard formulation of the dual problem as a piecewise-convex problem in the unique multiplier of the knapsack constraint and the corresponding obvious dual-ascent approach.

Defines

- #define DualCQKnP_WHCH_QSORT 1

  If DualCQKnP_WHCH_QSORT == 0, the sort() function of the STL is used, otherwise a hand-made non-recursive quick-sort implementation is used.
- #define DualCQKnP_SANITY_CHECKS 0

  If DualCQKnP_SANITY_CHECKS == 1, sanity checks are done each time the data of the instance changes to pick up clearly bad values.

0.8.4.1 Detailed Description

Basis Continuous Quadratic Knapsack Problems (CQKnP) solver based on the standard dual-ascent approach. This class is restricted to instances where *all* items have *strictly positive* quadratic costs and *finite* bounds (both lower and upper).

Conforms to the standard interface for CQKnP solver defined by the abstract base class CQKnpClass.

Version

1.08

Date

21 - 12 - 2012

Author

Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Enrico Gorgone
Dipartimento di Elettronica Informatica e Sistemistica
Universita' della Calabria

0.8.5 ExDualCQKnP.h File Reference

Continuous Quadratic Knapsack Problems (CQKnP) solver based on the standard dual-ascent approach, which extends the DualCQKnP class to non-negative quadratic costs and extended real bounds (and therefore fully conforms to the standard interface for CQKnP solver defined by the abstract base class CQKnpClass).

```
#include "DualCQKnP.h"
```

Classes

- class ExDualCQKnP

  Continuous Quadratic Knapsack Problems (CQKnP) solver derived from the DualCQKnP class (and therefore from CQKnPClass) and extending it, using the same standard dual-ascent approach, to non-negative quadratic costs and extended real bounds.

0.8.5.1 Detailed Description

Continuous Quadratic Knapsack Problems (CQKnP) solver based on the standard dual-ascent approach, which extends the DualCQKnP class to non-negative quadratic costs and extended real bounds (and therefore fully conforms to the standard interface for CQKnP solver defined by the abstract base class CQKnpClass).

Version

1.06

Date

29 - 03 - 2012

Author

Antonio Frangioni
Operations Research Group
Dipartimento di Informatica
Universita' di Pisa
Enrico Gorgone
Dipartimento di Elettronica Informatica e Sistemistica
Universita' della Calabria

Index