A Library for Continuous Convex Separable Quadratic Knapsack Problems

Antonio Frangioni
Dipartimento di Informatica, Università di Pisa
Largo B. Pontecorvo 1, 56127 Pisa – Italy
frangio@di.unipi.it)

Enrico Gorgone Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, 87036 Rende (CS) – Italy
egorgone@deis.unical.it

Abstract

The Continuous Convex Separable Quadratic Knapsack problem (CQKnP) is an easy but useful model that has very many different applications. Although the problem can be solved quickly, it must typically be solved very many times within approaches to (much) more difficult models; hence an efficient solution approach is required. We present and discuss a small open-source library for its solution that we have recently developed and distributed.

Keywords: *Quadratic Programming, Continuous Nonlinear Resource Allocation Problem, Lagrangian Relaxation, Optimization Software*

1 Introduction

We consider the Continuous Quadratic Knapsack problem where a convex separable quadratic objective function has to be minimized over a set defined by a single linear constraint plus box constraints. Upon appropriate scaling, a formulation is

$$\text{(CQKnP)} \qquad \min\left\{ \; \sum_{j=1}^{n} c_j x_j + d_j x_j^2 \; : \; \sum_{j=1}^{n} x_j = v \, , \, a_j \leq x_j \leq b_j \;\; j = 1, \ldots, n \; \right\}$$

where $d_j \geq 0$, $a_j \in \mathbb{R} \cup \{-\infty\}$, $b_j \in \mathbb{R} \cup \{+\infty\}$, and $a_j \leq b_j$; alternatively, the constraint can be a $\leq$ one. This is a particular (but representative) case of the Continuous Nonlinear Resource Allocation Problem that has very many and diverse applications [6], among which Lagrangian heuristics for ramp-constrained Unit Commitment problems in electrical power production [3], 1-D sensor placement problems [2], and deflected subgradient approaches [1] for Multicommodity Flow Problems [4].

One of the two main classes of approaches to (CQKnP) [6] is Lagrangian relaxation: the dual function

$$\phi(\mu) = v\mu + \textstyle\sum_{j=1}^{n} \min\{\ (c_j - \mu)x_j + d_j x_j^2\ :\ a_j \leq x_j \leq b_j\ \}$$

is piecewise-convex with at most $2n$ pieces, and its maximization (subject to a sign constraint if the knapsack constraint is a $\leq$ one) is well-known to solve (CQKnP). Assuming for simplicity that $d_j > 0$ and $-\infty < a_j < b_j < +\infty$ hold for each $j = 1, \ldots, n$ (but the argument can be generalized), the maximization can be performed in $O(n)$ once the $2n$ breakpoints $\gamma_j^- = 2a_j d_j + c_j$ and $\gamma_j^+ = 2b_j d_j + c_j$ of ϕ (the values of μ where the *unconstrained* minimizer of each single-variable subproblem, $x_j^*(\mu) = (\mu - c_j)/2d_j$, "hits the boundary" of the feasible region) are sorted; thus a solution of (CQKnP) can be obtained in $O(n \log n)$. This could be brought down to $O(n)$ by finding the *critical interval* $[\alpha, \beta]$ (its extremes being some of the above breakpoints) such that $\phi'(\alpha) < 0$ and $\phi'(\beta) > 0$ by a search procedure based on selecting the *median index*, which can be found without sorting [6]; however, the large constants hidden in the "O" notation are likely to make this competitive only for really huge-size instances, which are usually not particularly relevant since (CQKnP) is (repeatedly) solved as a (sub-)subproblem of much harder problems that are then, necessarily, not huge-sized.

While the implementation of this approach is not particularly difficult, it is not entirely straightforward either. We found ourselves in the need of using this in different applications [3, 2], and there was no useable code available anywhere. Because the implementation effort required for such a solver, while not huge, is not negligible, especially if one wants to have a "complete" and efficient solution, we developed an open-source `C++` library that is now available at

`http://www.di.unipi.it/optimize/Software/CQKnP.html`

This short note describes the library, its structure and its results.

2 The Library

As in several other cases [5, 7], the library is structured around one *abstract interface class*. In our case this is `CQKnPClass`, which provides all means for handling problem instances and solving them. The approach is the straightforward "the object is the instance" one: each object of the class (that has a `void` constructor in order to allow arrays) contains one instance. The data can be loaded at once either from memory (method `LoadSet`) or from file (method `ReadInstance`), but later on any element can be independently changed: linear and quadratic cost coefficients (methods `ChgLCosts` and `ChgQCosts`, respectively), lower and upper bounds (methods `ChgLBnds` and `ChgUBnds`, respectively), and the "volume" v of the knapsack (method `ChgVlm`). The methods can address any subset, not necessarily contiguous, of the data and specialized methods exist for changing single values. The data can be queried back from the object, either in memory or onto a file, so that the caller need not to keep a copy. The solution method `SolveKNP` reports status codes according to the outcome; the optimal value and primal/dual optimal solutions can then be obtained.

The distribution of the library currently comes with three "concrete" classes, deriving from `CQKnPClass` and therefore implementing the abstract interface. One is `CQKnPCplex` which, as the name suggests, solves the problem by calling the C API of the general-purpose MIQP solver `Cplex` (which is a commercial product, but currently free for academic purposes). This is basically only useful as a tool to verify the correctness of alternative implementations, as §3 shows. The other two classes are `DualCQKnP` and `ExDualCQKnP`, respectively, and they implement the Lagrangian approach sketched above, in particular the version using sorting, refereed to as "ranking" in [6, §3.1.1]. The difference between the two is that `DualCQKnP` only solves "pure QP" instances of (CQKnP) where $d_j > 0$ and $-\infty < a_j < b_j < +\infty$ hold for each $j = 1, \ldots, n$; thereby it does not exactly implement the interface, which allows to set $d_j = 0$ and/or infinite bounds for some (or all) of the items. However, by doing so it considerably simplifies the main logic of the dual optimization phase, usually resulting in a somewhat faster algorithm as shown in §3. The `ExDualCQKnP` derives from `DualCQKnP`, and therefore re-uses a large part of its implementation (in particular the sorting procedures, described below), while extending it to complete handling of "mixed QP" instances of (CQKnP) as the interface dictates, possibly at the cost of a slight performance hit. There is also a fourth "testing" class `CQKnPClone` included in the library, whose meaning and usefulness is described in §3.

As anticipated, the sorting routine is the crucial part of the approach, and therefore it deserves specific attention. Different solutions have been tested, in particular whether the values γ_j^- and γ_j^+ that need be sorted are computed a-priori and stored in a vector (thereby trading memory for speed) or re-computed each time a comparison is done from the original data (thereby trading speed for memory); despite the possible impact on the memory hierarchy, the former solution has shown to be consistently superior and it has been adopted. The code automatically takes care of avoiding useless re-computations when one instance is re-solved after some modification, such as sorting when only the volume v changes. Furthermore, the code has two different sorting routines: a QuickSort and a BubbleSort, which can be changed at any time with the method `SetSort`. The former is the default and it is usually much faster, unless when "only a few things change" in which case the vector of breakpoints is "almost sorted" already and the latter can be faster, as §3 shows. As for the QuickSort routine, a macro (`DualCQKnP_WHCH_QSORT` in `DualCQKnP.C`) allows to choose at compile time between the `sort()` routine of the C++ Standard Template Library and a "hand-made" QuickSort that we developed which avoids recursive calls by using a vector to hold a stack of indices. Somewhat surprisingly, the hand-made version was found to be often slightly but noticeably faster. As all the code concerning sorting is implemented in `DualCQKnP` and (almost) not touched in the derived `ExDualCQKnP`, the latter automatically inherits all these options.

The distribution is completed by extensive documentation and two examples of main files. One (`Main.C`) just reads one (CQKnP) instance (in the format of [2]) and solves the corresponding problem. The second (`MainRnd.C`) instead randomly generates and solves on the fly a set of (CQKnP) instances (whose characteristics can be controlled with a small set of appropriate parameters) with *two* different solvers in order to cross-check the correctness of both solvers and compare their performances. This has been used for the results in the next section and may be useful to test alternative solution methods. In particular, one could reasonably want to test "pegging" algorithms [6, §3.2], that have been reported to be competitive. We only focussed on Lagrangian ones for three reasons: it is easier to derive efficient reoptimization procedures for all possible changes of data in the problem, they allow for both $d_j = 0$ and some bounds to be infinite for the *same* j, which may be useful in some applications, and they have a better complexity of $O(n \log n)$ (in our implementation, $O(n)$ in general) as opposed to $O(n^2)$ of pegging. Yet, "pegging" implementations could be easily added to the library.

3 Computational Results and Conclusions

The computational results have been obtained on a PC sporting an Intel Core i7-2600 CPU (3.40 GHz) with 8 Gb of RAM, running Ubuntu 11.04. All codes have been compiled with GNU `g++` 4.5.2 (with `-O3` optimization option), and we have used version 12.2 of `Cplex`.

We have performed three set of tests. The first has been on randomly-generated (CQKnP) instances, using `MainRnd.C`; a script file is included in the distribution to replicate the exact set of instances used in our tests. If one of the two solvers to be checked is `DualCQKnP` these are "pure QP" instances with $d_j > 0$ and $-\infty < a_j < b_j < +\infty$ for all $j = 1, \ldots, n$; otherwise, roughly 10% of the quadratic cost coefficients are zero and half of the bounds are ($\pm$) infinite. All the data is uniformly drawn from intervals of length 100, except the "volume" v that is uniformly drawn from $(0, 1000]$. Most instances are feasible, but unfeasible ones do happen; also, it may happen that $a_j = b_j$ for some j. Once one instance is generated, it is solved 16 times: one as it is, the remaining ones changing a selectable percentage of the data. All possible combinations of changes in costs, bounds and volume are tested twice. We remark that we did not aim at making these instances and the testing methodology particularly "realistic"; the aim of these experiments was primarily to check the correctness of the library, and to get a sense of the computational impact of the available algorithmic options.

The results of our experiments are reported in Table 1. Since the running times for a single instance are short, which could create problems with the accuracy of timing routines, we always solved and measured batches of instances; each entry of the table reports the time for solving 1000 instances. We remark that this actually means 1000 solutions "from scratch" and 15000 reoptimizations after changes in the data, as described above (which basically means 16000 runs when %c = 100). Also, each test has been repeated thrice, and the average results of the three runs are reported. We tested problems with the number of variables n varying from 100 to 100000, and with four choices for the percentage of data that is changed at each of the 15 reoptimizations (column "c%") after that the instance has been first solved. The leftmost half of the table is devoted to "pure QP" instances, while the rightmost part is devoted to "mixed QPs". For the former we compare all classes `DualCQKnP` (columns "Dual"), `ExDualCQKnP` (columns "Ex") and `CQKnPCplex` (columns "`Cplex`"). For the Lagrangian approaches, using the "specialized" `DualCQKnP` turned out to be most often—but, somewhat puzzlingly, not always—slightly better than using the "general" `ExDualCQKnP`, as expected. We also experiments with both variants of the sorting procedure; almost always the hand-made QS proved to be slightly but noticeably faster than the STL routine. Then, we also explored the combination of the QS, used during the first solution, and of the Bubble Sort (column "+BS"), used during reoptimization; this was avoided for %c = 100 because

		"pure QPs"							"mixed QPs"				
		QS			STL		`Cplex`		`ExDualCQKnP`			`Cplex`	
n	%c	+BS	Dual	Ex	Dual	Ex	Dual	Auto	STL	QS	+BS	Dual	Auto
1e5	100		113.43	117.60	121.20	124.53	13599.17	961.67	6.02	6.67		6807.33	208.07
	10	29.40	28.82	30.13	30.27	32.10	3917.27	275.37	4.73	5.00	4.62	81.70	90.60
	5	27.80	27.67	29.07	30.17	30.50	3154.27	252.67	4.73	4.57	4.52	82.70	89.50
	1	27.57	28.45	29.93	30.93	31.30	3211.47	257.33	4.75	5.47	4.80	82.63	87.93
1e4	100		8.16	8.61	9.11	9.10	201.06	127.91	0.87	0.86		164.08	135.10
	10	2.09	2.08	2.30	2.38	2.47	42.13	35.70	0.48	0.56	0.49	9.55	23.73
	5	2.10	2.15	2.15	2.35	2.41	39.82	34.58	0.52	0.56	0.45	9.54	23.76
	1	2.14	2.16	2.15	2.44	2.32	38.33	33.69	0.52	0.64	0.47	9.57	29.25
1e3	100		0.59	0.63	0.55	0.71	5.29	14.79	0.40	0.41		11.31	39.52
	10	0.14	0.16	0.15	0.17	0.16	1.35	7.21	0.08	0.09	0.08	1.56	12.98
	5	0.15	0.16	0.16	0.17	0.17	1.34	7.74	0.10	0.09	0.10	1.60	13.06
	1	0.17	0.20	0.19	0.16	0.22	1.35	7.54	0.15	0.16	0.11	1.82	18.66
1e2	100		0.04	0.03	0.04	0.04	0.53	8.70	0.04	0.06		0.99	42.47
	10	0.01	0.01	0.01	0.01	0.01	0.22	5.91	0.02	0.02	0.01	0.39	22.23
	5	0.02	0.02	0.01	0.02	0.01	0.22	6.21	0.02	0.03	0.02	0.42	27.00
	1	0.02	0.02	0.02	0.02	0.02	0.24	6.45	0.03	0.04	0.02	0.47	31.62

Table 1: Computational results on randomly-generated instances.

it resulted in unbearably long running times, as it could be expected. Using the BS in reoptimization is often, but not always, preferable; it could however be noted that real approaches may perform many more than 16 reoptimizations and more "consistent" ones (e.g. changing only one set of data), which may further improve the benefits of the BS. For `Cplex` we have experimented with all available solution algorithms; it turned out that two of them were "non dominated". In particular, forcing `Cplex` to always use the dual simplex turned out to be preferable if the size of the instances is small and "not too many things change", but it can be downright catastrophic otherwise. On the other hand, allowing it to automatically choose the algorithm can result in running times one orders of magnitude larger in several scenarios. Not that this matters much: the Lagrangian approaches are almost always at least one order of magnitude faster than the best that `Cplex` can do. The results for "mixed QPs" are by and large analogous, with a few differences. One is that these instances are significantly easier to solve for all approaches. Another is that the STL `sort()` in this case is often better than our QS; however, using the BS in reoptimization in this case has a more marked positive effect, so that option is most often the best. For `Cplex`, the dual algorithm is better than the automatic choice in all cases but for $n = 100000$ and $\%c = 100$, where it is worse by a somewhat surprising factor of 30. Yet, Lagrangian approaches consistently beat `Cplex` by over one order of magnitude.

The second set of tests has been on the continuous relaxation of the Sensor Placement Problem (cf. [2] and the references therein), that has the form (CQKnP) with $a_j = 0$, $b_j = \infty$, and $v = 1$. In this case we have only solved each instance once, corresponding to the root node relaxation in a B&B approach, to focus on the "one shot" performances of the solvers. The instances, with $n = 10000$ and $n = 100000$, can be generated with the generator freely available at `http://www.di.unipi.it/optimize/Data/RDR.html` ;

a script file for reproducing the tests, together with the corresponding `Main.C` file, is provided in the distribution of the library. The results are reported in Table 2, using the best settings we found, i.e., `DualCQKnP` (since the instances have the right structure for it to be used) with the "hand-made" QS and the primal algorithm for `Cplex`. The Table shows that even in a rigorously one-shot situation the specialized Lagrangian algorithm is substantially faster than the general-purpose one, as expected; we remark that running times exclude set-up procedures, that in the general-purpose solver can be costly.

n	`DualCQKnP`	`Cplex`
10000	0.009	0.043
100000	0.131	1.077

Table 2: Computational results on Sensor-Placement Problems.

Finally, the third set of tests has been performed on the solution of Multicommodity Min-Cost Flow (MMCF) problems using a Resource Decomposition approach [4] driven by a (deflected) projected subgradient method [1]. In this case, for a problem with m arcs and k commodities, the approach has to solve $2m$ (CQKnP) problems at each step, each one with k variables: one is to compute the deflected subgradient, and the other is to project the final iterate on the feasible region (see [1] for details). In this case the ability to have multiple objects simultaneously present in memory is crucial, as we keep one solver for each of the $2m$ instances which we re-use along the iterations. The results reported in Table 3 refer to the 32 (MMCF) instances p33–p64 produced by the "Mulgen" generator found at

`http://www.di.unipi.it/optimize/Data/MMCF.html#Canad` .

The instances are organized in eight groups of four, the groups differing for several characteristics while the instances in the same group only differing for the seed of the random number generator used, and we thus report averaged results for each group. Exactly the same $2m$ problems have been solved for 100 iterations of the subgradient method with two different solvers, `Cplex` and `DualCQKnP`, each one tuned to its best (which turned out to be the same as in the previous case); the reported running times are only these directly incurred in the solution of the (CQKnP) problems. For this test we have developed a `template CQKnPClone` class that takes two solvers of type `CQKnPClass` and does exactly the same operations on both; this allows easy comparison (as well as correctness testing) of the solvers when used within more complex approaches, as in this case. Distributing the instances for this test is complicated; these depend on a rather complex code and each (MMCF) instance produces more than 50000 problems, so if we saved each (CQKnP) instance on a file these would then total to more than 150 Mb for each (MMCF(instance. In order to allow tests to be reproduced if necessary, the corresponding code and algorithmic parameters and/or the instances will be available on request from the authors.
Again, the Table clearly shows that the specialized solver can be significantly faster than the general-purpose one, up to more than two orders of magnitude in this case.

To conclude, this small library can be a useful, albeit by no means huge, contribution for all applications that require the solution of (CQKnP), especially if some care is exercised in

k	40	40	100	100	200	200	200	400
m	226	292	520	684	230	292	292	520
`DualCQKnP`	0.41	0.16	3.71	6.23	2.78	4.58	82.76	44.21
`Cplex`	89.35	134.97	517.31	828.08	139.38	197.25	551.49	328.06

Table 3: Computational results on Network Design Problems.

properly choosing among the available options. The library could easily be extended e.g. to include pegging algorithms [6], should the need arise. Also, extending the interface and the Lagrangian approaches to more general classes of nonlinear functions would not be too difficult. We believe that as optimization software becomes more and more complex, releasing ready-to-use and well-engineering libraries like this, even if they can only solve rather specialized problems, will be more and more important and should be actively encouraged.

References

[1] G. d'Antonio and A. Frangioni. Convergence Analysis of Deflected Conditional Approximate Subgradient Methods. *SIAM Journal on Optimization*, 20(1):357–386, 2009.

[2] A. Frangioni, C. Gentile, E. Grande, and A. Pacifici. Projected Perspective Reformulations with Applications in Design Problems. *Operations Research*, 59(5):1225–1232, 2010.

[3] A. Frangioni, C. Gentile, and F. Lacalandra. New Lagrangian Heuristics for Ramp-Constrained Unit Commitment Problems. In *Proceedings 19th Mini-EURO Conference in Operational Research Models and Methods in the Energy Sector –ORMMES 2006*. INESC Coimbra, 2006.

[4] J. Kennington and M. Shalaby. An Effective Subgradient Procedure for Minimal Cost Multicommodity Flow Problems. *Management Science*, 23(9):994–1004, 1977.

[5] *The MCFClass Project*. `http://www.di.unipi.it/optimize/Software/MCF.html`.

[6] M. Patriksson. A Survey on the Continuous Nonlinear Resource Allocation Problem. *European Journal on Operational Research*, 185:1–46, 2008.

[7] M. Saltzman, L. Ladǹyi, and T. Ralphs. *The COIN-OR Open Solver Interface*, 2004.