A Lifted Linear Programming Branch-and-Bound Algorithm for Mixed Integer Conic Quadratic Programs

Juan Pablo Vielma, Shabbir Ahmed, George L. Nemhauser,
H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, 765 Ferst Drive NW, Atlanta, GA 30332-0205, USA, {jvielma@isye.gatech.edu, shabbir.ahmed@isye.gatech.edu, george.nemhauser@isye.gatech.edu}

This paper develops a linear programming based branch-and-bound algorithm for mixed integer conic quadratic programs. The algorithm is based on a higher dimensional or lifted polyhedral relaxation of conic quadratic constraints introduced by Ben-Tal and Nemirovski. The algorithm is different from other linear programming based branch-and-bound algorithms for mixed integer nonlinear programs in that, it is not based on cuts from gradient inequalities and it sometimes branches on integer feasible solutions. The algorithm is tested on a series of portfolio optimization problems. It is shown that it significantly outperforms commercial and open source solvers based on both linear and nonlinear relaxations.

Key words: nonlinear integer programming; branch and bound; portfolio optimization
History: February 2007.

---

1. Introduction

This paper deals with the development of an algorithm for the class of mixed integer nonlinear programming (MINLP) problems known as mixed integer conic quadratic programming problems. This class of problems arises from adding integrality requirements to conic quadratic programming problems (Lobo et al., 1998), and is used to model several applications from engineering and finance. Conic quadratic programming problems are also known as second order cone programming problems, and together with semidefinite and linear programming (LP) problems are special cases of the more general conic programming problems (Ben-Tal and Nemirovski, 2001a). For ease of exposition, we will refer to conic quadratic and mixed integer conic quadratic programming problems simply as conic programming (CP) and mixed integer conic programming (MICP) problems respectively.

We are interested in solving MICP problems of the form

$$z_{\mathsf{MICPP}} := \max_{x,y} \quad cx + dy \tag{1}$$
$$s.t.$$
$$Dx + Ey \leq f \tag{2}$$
$$(x, y) \in \mathcal{CC}_i \qquad i \in \mathcal{I} \tag{3}$$
$$(x, y) \in \mathbb{R}^{n+p} \tag{4}$$
$$x \in \mathbb{Z}^n \tag{5}$$

where $c \in \mathbb{R}^n$, $d \in \mathbb{R}^p$, $D \in \mathbb{R}^{m\times n}$, $E \in \mathbb{R}^{m\times p}$, $f \in \mathbb{R}^m$, $\mathcal{I} \subset \mathbb{Z}_+$, $|\mathcal{I}| < \infty$ and for each $i \in \mathcal{I}$ the set $\mathcal{CC}_i$ is a conic constraint of the form

$$\mathcal{CC} := \{(x, y) \in \mathbb{R}^{n+p} \,:\, ||Ax + By + \delta||_2 \leq ax + by + \delta_0\} \tag{6}$$

for some $A \in \mathbb{R}^{r\times n}$, $B \in \mathbb{R}^{r\times p}$, $\delta \in \mathbb{R}^r$, $a \in \mathbb{R}^n$, $b \in \mathbb{R}^p$, $\delta_0 \in \mathbb{R}$ and where $||\cdot||_2$ is the Euclidean norm. We denote the MICP problem given by (1)–(5) as MICPP and its CP relaxation given by (1)–(4) as CPP.

MICPP includes many portfolio optimization problems (see for example Ben-Tal and Nemirovski (1999), Ceria and Stubbs (2006), Lobo et al. (1998) and Lobo et al. (2007)). A specific example is the portfolio optimization problem with cardinality constraints (see for example Bienstock (1996), Chang et al. (2000), Maringer and Kellerer (2003) and Bertsimas and Shioda (2004)) which can be formulated as

$$\max_{x,y} \qquad \bar{a}y \tag{7}$$
$$s.t.$$
$$||Q^{1/2}y||_2 \leq \sigma \tag{8}$$
$$\sum_{j=1}^{n} y_j = 1 \tag{9}$$
$$y_j \leq x_j \qquad \forall j \in \{1, \ldots, n\} \tag{10}$$
$$\sum_{j=1}^{n} x_j \leq K \tag{11}$$
$$x \in \{0, 1\}^n \tag{12}$$
$$y \in \mathbb{R}^n_+, \tag{13}$$

where n is the number of assets available, y indicates the fraction of the portfolio invested in each asset, $\bar{a} \in \mathbb{R}^n$ is the vector of expected returns of the stocks, $Q^{1/2}$ is the positive

semidefinite square root of the covariance matrix of the returns of the stocks, σ is the maximum allowed risk and $K < n$ is the maximum number of stocks that can be held in the portfolio. Objective (7) is to maximize the expected return of the portfolio, constraint (8) limits the risk of the portfolio, and constraints (10)–(12) limit the number of stocks that can be held in the portfolio to K. Finally, constraints (9) and (13) force the investment of the entire budget in the portfolio.

Most algorithms for solving MICP problems (and in general MINLP problems) can be classified into two major groups depending on what type of continuous relaxations they use (see for example Bonami et al. (2005) and Grossmann (2002)).

The first group only uses the nonlinear relaxation CPP in a branch-and-bound procedure (Borchers and Mitchell, 1994; Gupta and Ravindran, 1985; Leyffer, 2001; Stubbs and Mehrotra, 1999). This procedure is the direct analog of the LP based branch-and-bound procedure for mixed integer linear programming (MILP) problems and is the basis for the MICP solver in CPLEX 9.0 and 10.0 (ILOG, 2005) and the I-BB solver in Bonmin (Bonami et al., 2005). We refer to these algorithms as *NLP based branch-and-bound algorithms*.

The second group uses polyhedral relaxations of the nonlinear constraints of MICPP, possibly together with the nonlinear relaxation CPP. These polyhedral relaxations are usually updated after solving an associated MILP problem or inside a branch-and-bound procedure. Additionally the nonlinear relaxation of MICPP is sporadically solved to obtain integer feasible solutions, to improve the polyhedral relaxations, to fathom nodes in a branch-and-bound procedure or as a local search procedure. Some of the algorithms in this group include outer approximation (Duran and Grossmann, 1986; Fletcher and Leyffer, 1994), generalized Benders decomposition (Geoffrion, 1972), LP/NLP-based branch-and-bound (Quesada and Grossmann, 1992) and the extended cutting plane method (Westerlund and Pettersson, 1995; Westerlund et al., 1994). This approach is the basis for the I-OA, I-QG and I-Hyb solvers in Bonmin (Bonami et al., 2005) and the MINLP solver FilMINT (Abhishek et al., 2006). We refer to these algorithms as *polyhedral relaxation based algorithms*.

For algorithms in the second group to perform efficiently, it is essential to have polyhedral relaxations of the nonlinear constraints that are both tight and have few constraints. To the best of our knowledge, the polyhedral relaxations used by all the algorithms proposed so far are based on gradient inequalities for the nonlinear constraints. This approach yields a polyhedral relaxation which is constructed in the space of the original variables of the problem. The difficulty with these types of polyhedral relaxations is that they can require

an unmanageable number of inequalities to yield tight approximations of the nonlinear constraints. In particular, it is known that obtaining a tight polyhedral approximation of the Euclidean ball without using extra variables requires an exponential number of inequalities (Ball, 1997). To try to resolve this issue, current polyhedral based algorithms generate the relaxations dynamically.

In the context of CP problems, an alternative polyhedral relaxation that is not based on gradient inequalities was introduced in 1999 by Ben-Tal and Nemirovski (Ben-Tal and Nemirovski, 2001b). This approach uses the projection of a higher dimensional or *lifted* polyhedral set to generate a polyhedral relaxation of a conic quadratic constraint of the form $\mathcal{CC}$. By exploiting the fact that projection can significantly multiply the number of facets of a polyhedron, this approach constructs a relaxation that is "efficient" in the sense that it is very tight and yet it is defined using a relatively small number of constraints and extra variables. The relaxation of Ben-Tal and Nemirovski has been further studied by Glineur (Glineur, 2000) who also tested it computationally on continuous CP problems. These tests showed that solving the original CP problem with state of the art interior point solvers was usually much faster than solving the polyhedral relaxation.

Although the polyhedral relaxation of Ben-Tal and Nemirovski (2001b) and Glineur (2000) might not be practical for solving purely continuous CP problems, it could be useful for polyhedral relaxation based algorithms for solving MICP problems. In particular, solving the polyhedral relaxation in a branch-and-bound procedure instead of the original CP relaxations could benefit from the "warm start" capabilities of the simplex algorithm for LP problems and the various integer programming enhancements such as cutting planes and preprocessing that are available in commercial MILP solvers. The objective of this paper is to develop such an algorithm and to demonstrate that this approach can significantly outperform other methods. The algorithm is conceptually valid for general MINLP problems, but we only test it on MICP problems as we are only aware of the existence of an efficient lifted polyhedral relaxation for this case.

The remainder of the paper is organized as follows. In section 2 we introduce a branch-and-bound algorithm based on a lifted polyhedral relaxation. In section 3 we describe the polyhedral relaxation of Ben-Tal and Nemirovski (2001b) and Glineur (2000) we use in our test. Then, in section 4 we present computational results which demonstrate that the algorithm significantly outperforms other methods. Finally, in section 5 we give some conclusions and possible future work in this area.

2. A Branch-and-Bound Algorithm for MINLP

We describe the algorithm for general MINLP problems because it allows us to simplify the notation. The algorithm is somewhat similar to other polyhedral relaxation algorithms and in particular to enhanced versions of the LP/NLP-based branch-and-bound algorithm such as Bonmin's I-Hyb solver and FilMINT. The main differences between the proposed algorithm and existing polyhedral relaxation based algorithms are that:

(i) it is based on a lifted polyhedral relaxation instead of one constructed using gradient inequalities,

(ii) it does not update the relaxation using gradient inequalities, and

(iii) it will sometimes branch on integer feasible solutions.

The MINLP we solve is of the form

$$z_{\mathsf{MINLPP}} := \max_{x,y} \quad cx + dy \tag{14}$$
$$s.t.$$
$$(x, y) \in \mathcal{C} \subset \mathbb{R}^{n+p} \tag{15}$$
$$x \in \mathbb{Z}^n \tag{16}$$

where $\mathcal{C}$ is a compact convex set. We denote the problem given by (14)–(16) by MINLPP. We also denote by NLPP the continuous relaxation of MINLPP given by (14)–(15) and we assume for simplicity that MINLPP is feasible.

We further assume that we have a lifted polyhedral relaxation of the convex set $\mathcal{C}$. In other words there exists $q \in \mathbb{Z}_+$ and a bounded polyhedron $\mathcal{P} \subset \mathbb{R}^{n+p+q}$ such that

$$\mathcal{C} \subset \{(x, y) \in \mathbb{R}^{n+p} \,:\, \exists\, v \in \mathbb{R}^q \text{ s.t. } (x, y, v) \in \mathcal{P}\}.$$

Thus we have the linear programming relaxation of MINLPP given by

$$z_{\mathsf{LPP}} := \max_{x,y,v} \quad cx + dy \tag{17}$$
$$s.t.$$
$$(x, y, v) \in \mathcal{P}, \tag{18}$$

which we denote by LPP. We also denote by MILPP the problem obtained by adding the integrality requirements to LPP, in other words MILPP is the problem given by (17)–(18) and (16).

Note that we could very well choose $q = 0$ in the construction of LPP, but as we will discuss in section 3, the key idea for the effectiveness of our algorithm is the use of a tight LP relaxation that requires $q > 0$.

The final problem we use in the algorithm is defined for any $\hat{x} \in \mathbb{Z}^n$ as

$$\begin{aligned} z_{\mathsf{NLPP}(\hat{x})} := \max_{y} \quad & c\hat{x} + dy \\ s.t. \quad & \\ & (\hat{x}, y) \in \mathcal{C} \subset \mathbb{R}^{n+p}. \end{aligned}$$

We denote this problem by $\mathsf{NLPP}(\hat{x})$.

We use these auxiliary problems to construct a branch-and-bound algorithm for solving MINLPP as follows. For any $(l^k, u^k) \in \mathbb{Z}^{2n}$ we denote by $\mathsf{LPP}(l^k, u^k)$ and $\mathsf{NLPP}(l^k, u^k)$ the problems obtained by adding constraints $l^k \leq x \leq u^k$ to LPP and NLPP respectively. We also adopt the convention that a node k in a branch-and-bound tree is defined by some $(l^k, u^k, \mathrm{UB}^k) \in \mathbb{Z}^{2n} \times (\mathbb{R} \cup \{+\infty\})$ where (l^k, u^k) are the bounds defining the node and UB^k is an upper bound on $z_{\mathsf{NLPP}(l^k,u^k)}$. Furthermore, we denote by LB the global lower bound on z_{MINLPP} and by $\mathcal{H}$ the set of active branch-and-bound nodes. We give in Figure 1 a lifted LP branch-and-bound algorithm for solving MINLPP.

A pure NLP based branch-and-bound algorithm solves $\mathsf{NLPP}(l^k, u^k)$ at each node k of the branch-and-bound tree. The idea of the lifted LP branch-and-bound algorithm of Figure 1 is to replace each call to $\mathsf{NLPP}(l^k, u^k)$ in an NLP based branch-and-bound algorithm by a call to $\mathsf{LPP}(l^k, u^k)$. After this replacement special care has to be taken when fathoming by integrality as an integer feasible solution to $\mathsf{LPP}(l^k, u^k)$ is not necessarily an integer feasible solution to $\mathsf{NLPP}(l^k, u^k)$. This is handled by the algorithm in lines 11–28. The first step is to solve $\mathsf{NLPP}(\hat{x}^k)$ to attempt to correct an integer feasible solution $(\hat{x}^k, \hat{y}^k)$ to $\mathsf{LPP}(l^k, u^k)$ into an integer feasible solution to $\mathsf{NLPP}(l^k, u^k)$. If the correction is successful and $z_{\mathsf{NLPP}(\hat{x}^k)} > \mathrm{LB}$ we can update LB. This step is carried out in lines 11–14 of the algorithm. Another complication arises when the optimal solution to $\mathsf{LPP}(l^k, u^k)$ is integer feasible, but $l^k \neq u^k$. The problem in this case is that integer optimal solutions to $\mathsf{LPP}(l^k, u^k)$ and $\mathsf{NLPP}(\hat{x}^k)$ may not be solutions to $\mathsf{MINLPP}(l^k, u^k)$. In fact, in this case, it is possible for $\mathsf{NLPP}(\hat{x}^k)$ to be infeasible and for $\mathsf{MINLPP}(l^k, u^k)$ to be feasible. To resolve this issue, the algorithm of

Set global lower bound LB := $-\infty$.
Set $l_i^0 := -\infty$, $u_i^0 := +\infty$ for all $i \in \{1, \ldots, n\}$.
Set $\mathrm{UB}^0 = +\infty$.
Set node list $\mathcal{H} := \{(l^0, u^0, \mathrm{UB}^0)\}$.
while $\mathcal{H} \neq \emptyset$ **do**
    Select and remove a node $(l^k, u^k, \mathrm{UB}^k) \in \mathcal{H}$.
    Solve $\mathsf{LPP}(l^k, u^k)$.
    if $\mathsf{LPP}(l^k, u^k)$ *is feasible* **and** $z_{\mathsf{LPP}(l^k,u^k)} > \mathrm{LB}$ **then**
        Let $(\hat{x}^k, \hat{y}^k)$ be the optimal solution to $\mathsf{LPP}(l^k, u^k)$.
        if $\hat{x}^k \in \mathbb{Z}^n$ **then**
            Solve $\mathsf{NLPP}(\hat{x}^k)$.
            if $\mathsf{NLPP}(\hat{x}^k)$ *is feasible* **and** $z_{\mathsf{NLPP}(\hat{x}^k)} > \mathrm{LB}$ **then**
                LB := $z_{\mathsf{NLPP}(\hat{x}^k)}$.
            end
            if $l^k \neq u^k$ **and** $z_{\mathsf{LPP}(l^k,u^k)} > \mathrm{LB}$ **then**
                Solve $\mathsf{NLPP}(l^k, u^k)$.
                if $\mathsf{NLPP}(l^k, u^k)$ *is feasible* **and** $z_{\mathsf{NLPP}(l^k,u^k)} > \mathrm{LB}$ **then**
                    Let $(\tilde{x}^k, \tilde{y}^k)$ be the optimal solution to $\mathsf{NLPP}(l^k, u^k)$.
                    if $\tilde{x}^k \in \mathbb{Z}^n$ **then** `/* Fathom by Integrality */`
                        LB := $z_{\mathsf{NLPP}(l^k,u^k)}$.
                    else `/* Branch on` $\tilde{x}^k$ `*/`
                        Pick i_0 in $\{i \in \{1, \ldots, n\} : \tilde{x}_i^k \notin \mathbb{Z}\}$.
                        Let $l_i = l_i^k$, $u_i = u_i^k$ for all $i \in \{1, \ldots, n\} \setminus \{i_o\}$.
                        Let $u_{i_0} = \lfloor \tilde{x}_{i_0}^k \rfloor$, $l_{i_0} = \lfloor \tilde{x}_{i_0}^k \rfloor + 1$.
                        $\mathcal{H} := \mathcal{H} \cup \{(l^k, u, z_{\mathsf{NLPP}(l^k,u^k)}), (l, u^k, z_{\mathsf{NLPP}(l^k,u^k)})\}$
                    end
                end
            end
        else `/* Branch on` $\hat{x}^k$ `*/`
            Pick i_0 in $\{i \in \{1, \ldots, n\} : \hat{x}_i^k \notin \mathbb{Z}\}$.
            Let $l_i = l_i^k$, $u_i = u_i^k$ for all $i \in \{1, \ldots, n\} \setminus \{i_o\}$.
            Let $u_{i_0} = \lfloor \hat{x}_{i_0}^k \rfloor$, $l_{i_0} = \lfloor \hat{x}_{i_0}^k \rfloor + 1$.
            $\mathcal{H} := \mathcal{H} \cup \{(l^k, u, z_{\mathsf{LPP}(l^k,u^k)}), (l, u^k, z_{\mathsf{LPP}(l^k,u^k)})\}$
        end
    end
    Remove every node $(l^k, u^k, \mathrm{UB}^k) \in \mathcal{H}$ such that $\mathrm{UB}^k \leq \mathrm{LB}$.
end

Figure 1: A Lifted LP Branch-and-Bound Algorithm.

Figure 1 solves $\mathsf{NLPP}(l^k, u^k)$ to process the node in the same way it would be processed in an NLP based branch-and-bound algorithm for MINLPP. This last step is carried out in lines 15–28.

Note that, in lines 21–26, the algorithm is effectively branching on a variable x_i such that $\hat{x}_i^k$ is integer but for which $l_i^k < u_i^k$. This idea of branching on integer feasible variables is not a new idea in MILP (it can be used for example to find alternative optimal solutions), but to the best of our knowledge it has never been used in the context of MINLP solvers.

We show the correctness of the algorithm in the following proposition.

Proposition 1. *For any polyhedral relaxation* LPP *of* NLPP *using a bounded polyhedron* $\mathcal{P}$*, the lifted LP branch-and-bound algorithm of Figure 1 terminates with* LB *equal to the optimal objective value of* MINLPP*.*

Proof. Finiteness of the algorithm is direct from the fact that $\mathcal{P}$ is bounded. However after branching in lines 21–26, solution $(\hat{x}^k, \hat{y}^k)$ could be repeated in one of the newly created nodes, which could cause $(\hat{x}^k, \hat{y}^k)$ to be generated again in several nodes. This can only happen a finite number of times though, as the branching will eventually cause $l^k = u^k$ or $\mathsf{LPP}(l^k, u^k)$ will become infeasible.

All that remains to prove is that the sub-tree rooted at a fathomed node cannot contain an integer feasible solution to MINLPP which has an objective value strictly larger than the current incumbent integer solution. The algorithm fathoms a node only in lines 8, 15, 17 and 19. In line 8, the node is fathomed if $\mathsf{LPP}(l^k, u^k)$ is infeasible or if $z_{\mathsf{LPP}(l^k,u^k)} \leq \text{LB}$. Because $\mathsf{LPP}(l^k, u^k)$ is a relaxation of $\mathsf{NLPP}(l^k, u^k)$ we have that infeasibility of $\mathsf{LPP}(l^k, u^k)$ implies infeasibility of $\mathsf{NLPP}(l^k, u^k)$ and $z_{\mathsf{NLPP}(l^k,u^k)} \leq z_{\mathsf{LPP}(l^k,u^k)}$, hence in both cases we have that the sub-tree rooted at node (l^k, u^k) cannot contain an integer feasible solution strictly better than the incumbent. In line 15, the node is fathomed if $l^k = u^k$ or if $z_{\mathsf{LPP}(l^k,u^k)} \leq \text{LB}$. In the first case, $\mathsf{NLPP}(l^k, u^k) = \mathsf{NLPP}(\hat{x}^k)$ and hence processing node k is correctly done by lines 12–14. In the second case, the node is correctly fathomed for the same reasons for correctness in line 8. In line 17, the node is fathomed if $\mathsf{NLPP}(l^k, u^k)$ is infeasible or if $z_{\mathsf{NLPP}(l^k,u^k)} \leq \text{LB}$, in either case the sub-tree rooted at the fathomed node cannot contain a integer feasible solution strictly better that the incumbent. Finally, in line 19 the node is fathomed because solution $(\tilde{x}^k, \tilde{y}^k)$ to $\mathsf{NLPP}(l^k, u^k)$ is integer feasible and hence it is the best integer feasible solution that can be found at the sub-tree rooted at the fathomed node. □

We note that, as in other branch-and-bound algorithms, at any point in the execution of the algorithm we have a lower bound of z_{MINLPP} given by LB and an upper bound given by $\max\{\mathrm{UB}^k : (l^k, u^k, \mathrm{UB}^k) \in \mathcal{H}\}$. This can be used for early termination of the algorithm given a target optimality gap.

3. Lifted Polyhedral Relaxations

The key idea for the effectiveness of the lifted LP branch-and-bound algorithm is the use of a lifted polyhedral relaxation ($q > 0$) for the construction of LPP. For the algorithm to be effective we need $\mathsf{NLPP}(l^k, u^k)$ to be called in as few nodes as possible, so we need LPP to be a tight approximation of NLPP. On the other hand we need to solve $\mathsf{LPP}(l^k, u^k)$ quickly, which requires the polyhedral relaxation to have relatively few constraints and extra variables. The problem is that using a relaxation with $q = 0$, such as those constructed using gradient inequalities, can require a polyhedron $\mathcal{P}$ with an exponential number of facets to approximate the convex set $\mathcal{C}$ tightly. In fact, it is known (see for example Ball (1997)) that for any $\varepsilon > 0$ approximating the d-dimensional unit euclidean ball $\mathcal{B}^d$ with a polyhedron $\mathcal{P} \subset \mathbb{R}^d$ such that $\mathcal{B}^d \subset \mathcal{P} \subset (1+\varepsilon)\mathcal{B}^d$ requires $\mathcal{P}$ to have at least $\exp(d/(2(1+\varepsilon)^2))$ facets. This is one of the main reason why current polyhedral relaxation based algorithms do not use a fixed polyhedral relaxation of $\mathcal{C}$ and instead dynamically refine the relaxation as needed.

On the other hand, when we allow for a polyhedron $\mathcal{P}$ in a higher dimensional space we can take advantage of the fact that a lifted polyhedron with a polynomial number of constraints and extra variables can have the same effect as a polyhedron in the original space with an exponential number of facets. Exploiting this property, it is sometimes possible to have a tight lifted polyhedral relaxation of $\mathcal{C}$ that can be described by a reasonable number of inequalities and extra variables. Ben-Tal and Nemirovski (2001b) introduced such a lifted polyhedral relaxation for MICP problems. We now give a compact description of the version of the lifted polyhedral relaxation of Ben-Tal and Nemirovski (2001b) and Glineur (2000) we use in this study.

We start by noting that a set $\mathcal{CC}$ given by (6) can be written as

$$\begin{aligned}\mathcal{CC} = \{(x,y) \in \mathbb{R}^{n+p} : \exists (z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \quad \text{s.t.} \quad & Ax + By + \delta = z, \\ & ax + by + \delta_0 = z_0, \\ & (z_0, z) \in \mathcal{L}^r\}\end{aligned}$$

where $\mathcal{L}^r$ is the $(r+1)$-dimensional Lorentz cone given by

$$\mathcal{L}^r := \{(z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \, : \, \sum_{k=1}^{r} z_k^2 \leq z_0^2\}.$$

Hence a polyhedral relaxation of $\mathcal{L}^r$ induces a polyhedral relaxation of $\mathcal{CC}$. Then, for a given tightness parameter $\varepsilon > 0$ we want to construct a polyhedron $\mathcal{L}^r_\varepsilon$ such that

$$\mathcal{L}^r \subsetneq \mathcal{L}^r_\varepsilon \subsetneq \{(z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \, : \, ||z||_2 \leq (1+\varepsilon) z_0\}. \tag{19}$$

To describe this polyhedral relaxation of $\mathcal{L}^r$ we assume at first that $r = 2^p$ for some $p \in \mathbb{Z}_+$. We then begin by grouping variables z in $\mathcal{L}^r$ into $k = r/2$ pairs and associate a new variable ρ_k for the kth pair. We can then rewrite $\mathcal{L}^r$ as

$$\begin{aligned}\mathcal{L}^r = \{(z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \, : & \exists \rho \in \mathbb{R}^{r/2} \quad \text{s.t.} \\ & \sum_{k=1}^{r/2} \rho_k^2 \leq z_0^2 \\ & z_{2k-1}^2 + z_{2k}^2 \leq \rho_k^2 \text{ for } k \in \{1, \ldots, r/2\}\}.\end{aligned}$$

In other words, we can rewrite $\mathcal{L}^r$ using $(r/2)$ 3-dimensional Lorentz cones and one $(r/2+1)$-dimensional Lorentz cone as

$$\begin{aligned}\mathcal{L}^r = \{(z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \, : & \exists \rho \in \mathbb{R}^{r/2} \quad \text{s.t.} \\ & (\rho, z_0) \in \mathcal{L}^{r/2} \\ & (z_{2k-1}, z_{2k}, \rho_k) \in \mathcal{L}^2 \text{ for } k \in \{1, \ldots, r/2\}\}.\end{aligned}$$

By recursively applying this procedure to the $(r/2+1)$-dimensional Lorentz cone we can rewrite $\mathcal{L}^r$ using only $(n-2)$ 3-dimensional Lorentz cones. We can then replace each of these 3-dimensional Lorentz cones with the polyhedral relaxation of $\mathcal{L}^2$ given by

$$\begin{aligned}\mathcal{W}_s := \{(z_0, z_1, z_2) \in \mathbb{R}_+ \times \mathbb{R}^2 \, : \, & \exists (\alpha, \beta) \in \mathbb{R}^{2s} \quad \text{s.t.} \\ & z_0 = \alpha_s \cos\left(\frac{\pi}{2^s}\right) + \beta_s \sin\left(\frac{\pi}{2^s}\right) \\ & \alpha_1 = z_1 \cos(\pi) + z_2 \sin(\pi) \\ & \beta_1 \geq |z_2 \cos(\pi) - z_1 \sin(\pi)| \\ & \alpha_{i+1} = \alpha_i \cos\left(\frac{\pi}{2^i}\right) + \beta_i \sin\left(\frac{\pi}{2^i}\right) \\ & \beta_{i+1} \geq \left|\beta_i \cos\left(\frac{\pi}{2^i}\right) - \alpha_i \sin\left(\frac{\pi}{2^i}\right)\right| \\ & \text{for } i \in \{1, \ldots, s-1\}\},\end{aligned}$$

for some $s \in \mathbb{Z}$.

For a general r, not necessarily a power of two, these ideas and some careful selection of the parameter s in $\mathcal{W}_s$ yield the polyhedral relaxation of $\mathcal{L}^r$ given by

$$\begin{aligned}
\mathcal{L}^r_\varepsilon := \{(z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \,:\, & \exists (\zeta^k)_{k=0}^K \in \mathbb{R}^{T(r)} \quad \text{s.t.} \\
& z_0 = \zeta_1^K \\
& \zeta_i^0 = z_i && \text{for } i \in \{1, \dots, r\}, \\
& (\zeta_{2i-1}^k, \zeta_{2i}^k, \zeta_i^{k+1}) \in \mathcal{W}_{s_k(\varepsilon)} && \text{for } i \in \{1, \dots, \lfloor t_k/2 \rfloor\}, \\
& && k \in \{0, \dots, K-1\}, \\
& \zeta_{t_k}^k = \zeta_{\lceil t_k/2 \rceil}^{k+1} && \text{for } k \in \{0, \dots, K-1\} \quad \text{s.t.} \\
& && t_k \text{ is odd}\}
\end{aligned}$$

where $K = \lceil \log_2(r) \rceil$, $\{t_k\}_{k=0}^K$ is defined by the recursion $t_0 = r$, $t_{k+1} = \lceil t_k/2 \rceil$ for $k \in \{0, \dots, K-1\}$, $T(r) = \prod_{k=0}^K t_k$ and

$$s_k(\varepsilon) = \left\lceil \frac{k+1}{2} \right\rceil - \left\lceil \log_4 \left(\frac{16}{9} \pi^{-2} \log(1+\varepsilon) \right) \right\rceil. \tag{20}$$

From Ben-Tal and Nemirovski (2001b) and Glineur (2000) we have that $\mathcal{L}^r_\varepsilon$ complies with (19) for any $\varepsilon > 0$ and has $O(n \log(1/\varepsilon))$ variables and constraints for any $0 < \varepsilon < 1/2$.

We can then use $\mathcal{L}^r_\varepsilon$ to define the relaxation of $\mathcal{CC}$ given by

$$\begin{aligned}
\mathcal{P}(\mathcal{CC}, \varepsilon) = \{(x, y) \in \mathbb{R}^{n+p} \,:\, \exists (z_0, z) \in \mathbb{R}_+ \times \mathbb{R}^r \quad \text{s.t.} \quad & Ax + By + \delta = z, \\
& ax + by + \delta_0 = z_0, \\
& (z_0, z) \in \mathcal{L}^r_\varepsilon\},
\end{aligned}$$

which complies with

$$\mathcal{CC} \subsetneq \mathcal{P}(\mathcal{CC}, \varepsilon) \subsetneq \{(x, y) \in \mathbb{R}^{n+p} \,:\, ||Ax + By + \delta||_2 \leq (1+\varepsilon)(ax + by + \delta_0)\}.$$

Using this relaxation we can construct the lifted polyhedral relaxation of CPP given by

$$z_{\mathsf{LP}(\varepsilon)} := \max_{x,y,v} \quad cx + dy \tag{21}$$

$$s.t.$$

$$Dx + Ey \leq f \tag{22}$$

$$(x, y, v) \in \overline{\mathcal{P}}(\mathcal{CC}_i, \varepsilon) \quad i \in \mathcal{I} \tag{23}$$

$$(x, y, v) \in \mathbb{R}^{n+p+q} \tag{24}$$

where $v \in \mathbb{R}^q$ are the auxiliary variables used to construct all $\mathcal{P}(\mathcal{CC}_i, \varepsilon)$'s and $\overline{\mathcal{P}}(\mathcal{CC}_i, \varepsilon)$ is the polyhedron in $\mathbb{R}^{n+p+q}$ whose projection to $\mathbb{R}^{n+p}$ is $\mathcal{P}(\mathcal{CC}_i, \varepsilon)$. We denote the problem given by (21)–(24) as $\mathsf{LP}(\varepsilon)$ and the problem given by (21)–(24) and (5) as $\mathsf{MILP}(\varepsilon)$.

We use $\mathsf{LPP} = \mathsf{LP}(\varepsilon)$ and $\mathsf{NLPP} = \mathsf{CPP}$ in the algorithm described in Figure 1 to obtain a lifted LP branch-and-bound algorithm for MICPP.

4. Computational Results

In this section we present the results of computational tests showing the effectiveness of the lifted LP branch-and-bound algorithm based on $\mathsf{LP}(\varepsilon)$. We begin by describing how the algorithm was implemented, then describe the problem instances we used in the tests and finally we present the computational results.

4.1. Implementation

We implemented the lifted LP branch-and-bound algorithm of Figure 1 for $\mathsf{LPP} = \mathsf{LP}(\varepsilon)$ and $\mathsf{NLPP} = \mathsf{CPP}$ by modifying CPLEX 10.0's MILP solver. We used the branch callback feature to implement branching on integer feasible solutions when necessary and we used the incumbent and heuristic callback features to implement the solve of $\mathsf{NLPP}(\hat{x}^k)$. All coding was done in C++ using Ilog Concert Technology. We used CPLEX's barrier solver to solve $\mathsf{CPP}(l^k, u^k)$ and $\mathsf{CPP}(\hat{x})$. In all cases we used CPLEX's default settings. We denote this implementation as $\mathsf{LP}(\varepsilon)$-BB .

There are some technical differences between this implementation and the lifted LP branch-and-bound algorithm of Figure 1. First, in the CPLEX based implementation, $\mathsf{NLPP}(\hat{x}^k)$ is solved for all integer feasible solutions found. This is a difference because the algorithm of Figure 1 only finds integer solutions when $\mathsf{LPP}(l^k, u^k)$ is integer feasible, but CPLEX also finds integer feasible solutions by using primal heuristics. Finally, the implementation benefits from other advanced CPLEX features such as preprocessing, cutting planes and sophisticated branching and node selection schemes. In particular, the addition of cutting planes conceptually modifies the algorithm as adding these cuts updates the polyhedral relaxation defining LPP. This updating does not use any information from the nonlinear constraints though, as CPLEX's cutting planes are only derived using the linear constraints of LPP and the integrality of the x variables.

4.2. Test Instances

Our test set consists of three different portfolio optimization problems with cardinality constraints from the literature (Ceria and Stubbs, 2006; Lobo et al., 1998, 2007). For most portfolio optimization problems only the continuous variables are present in the nonlinear constraints and hence the convex hull of integer feasible solutions to these problems is almost never a polyhedron. Furthermore, polyhedral relaxation based algorithms for the purely continuous versions of these problems are known to converge slowly. For these reasons we believe that portfolio optimization problems are a good set of problems to test the effectiveness of the lifted LP branch-and-bound algorithm based on $\mathsf{LP}(\varepsilon)$.

For all three problems we let a_i be the random return on stock i and let the expected value and covariance matrix of the joint distribution of $a = (a_1, \ldots, a_n)$ be $\bar{a} \in \mathbb{R}^n_+$ and Q respectively. Also, let y_i be the fraction of the portfolio invested in stock i and $Q^{1/2}$ be the positive semidefinite square root of Q.

The first problem is obtained by adding a cardinality constraint to the classical mean-variance portfolio optimization model to obtain the MICP problem already explained in (7)–(13). We refer to the set of instances of this problem as the *classical* instances.

The second problem is constructed by replacing the variance risk constraint (8) of the classical mean-variance model with two shortfall risk constraints of the form $\text{Prob}(\bar{a}y \geq W^{low}) \geq \eta$. Following Lobo et al. (1998) and Lobo et al. (2007) we formulate this model as a conic quadratic programming problem obtained by replacing constraint (8) in the classical mean-variance problem with

$$\Phi^{-1}(\eta_i)||Q^{1/2}y||_2 \leq \bar{a}y - W_i^{low} \qquad i \in \{1, 2\}$$

where $\Phi(\cdot)$ is the cumulative distribution function of a zero mean, unit variance Gaussian random variable. We refer to the set of instances of this problem as the *shortfall* instances.

The final problem is a robust portfolio optimization problem studied in Ceria and Stubbs (2006). This model assumes that there is some uncertainty in the expected returns $\bar{a}$ and that the true expected return vector is normally distributed with mean $\bar{a}$ and covariance matrix R. The model is similar to one introduced in Ben-Tal and Nemirovski (1999) and can be formulated as the conic quadratic programming problem obtained by replacing the objective function (7) of the classical mean-variance with $\max_{x,y,r} r$ and adding the constraint $\bar{a}y - \alpha||R^{1/2}y||_2 \geq r$ where $R^{1/2}$ is the positive semidefinite square root of R. The effect of this

change is the maximization of $\bar{a}y - \alpha||R^{1/2}y||_2$ which is a robust version of the maximization of the expected return $\bar{a}y$. We refer to the set of instances of this problem as the *robust* instances.

We generated the data for the classical instances in a manner similar to the test instances of Lobo et al. (2007). We first estimated $\bar{a}$ and Q from 251 daily closing prices of S&P 500 stocks starting with the 22nd of August 2005 and we scaled the distributions for a portfolio holding period of 20 days. Then, for each n we generated an instance by randomly selecting n stocks out of the 462 stocks for which we had closing price data. We also arbitrarily selected $\sigma = 0.2$ and $K = 10$.

For the shortfall instances we used the same data generated for the classical mean-variance instances, but we additionally included a risk-less asset with unit return to make these instances differ even more from the classical mean-variance instances. Also, in a manner similar to the test sets of Lobo et al. (2007) we arbitrarily selected $\eta_1 = 80\%$, $W_1^{low} = 0.9$, $\eta_2 = 97\%$, $W_2^{low} = 0.7$.

Finally, we generated the data for the robust instances in a manner similar to the test instances of Ceria and Stubbs (2006). We used the same daily closing prices used for the classical mean-variance and shortfall risk constraints instances, but we randomly selected different groups of n stocks and we generated the data in a slightly different way. For stock i we begin by calculating μ_i as the mean daily return from the first 120 days available. We then let $\bar{a}_i = 0.1\mu_i + 0.9r$ where r is the daily return for the 121st day. Finally Q is estimated from the same first 120 days and following Ceria and Stubbs (2006) we let $R = (0.9/120)Q$. We also arbitrarily selected $\alpha = 3$ and we again selected $\sigma = 0.2$ and $K = 10$.

For the three sets of instances we generated 100 instances for each n in $\{20, 30, 40, 50\}$ and 10 instanced for each n in $\{100, 200\}$.

4.3. Results

All computational tests where done on a dual 2.4GHz Xeon workstation with 2GB of RAM running Linux Kernel 2.4. The first set of experiments show calibration results for different values of ε. We then study how $\mathsf{LP}(\varepsilon)$-BB compares to other algorithms. Finally, we study some factors that might affect the effectiveness of $\mathsf{LP}(\varepsilon)$.

4.3.1. Selection of ε

Note that as ε gets smaller the size of $\mathsf{LP}(\varepsilon)$ grows as $O(n\log(1/\varepsilon))$, on the other hand the relaxation gets tighter. To select the value of ε for subsequent runs we first studied the sizes of $\mathsf{LP}(\varepsilon)$ for n in $\{20, 30\}$ and for values of ε in $\{1, 0.1, 0.01, 0.001, 0.0001\}$. Table 1 presents the number of columns, rows and non-zero coefficients for the different values of n and ε. The table also includes the same information for CPP.

		classical		shortfall		robust	
n	ε	cols+rows	nz	cols+rows	nz	cols+rows	nz
20	1	484	1172	908	2310	956	2368
	0.1	579	1343	1098	2652	1156	2728
	0.01	769	1685	1478	3336	1556	3448
	0.001	959	2027	1858	4020	1956	4168
	0.0001	1054	2198	2048	4362	2156	4528
	CPP	105	501	150	968	154	948
30	1	734	2076	1378	4098	1426	4146
	0.1	879	2337	1668	4620	1726	4686
	0.01	1169	2859	2248	5664	2326	5766
	0.001	1459	3381	2828	6708	2926	6846
	0.0001	1604	36427	3118	7230	3226	7386
	CPP	155	1051	220	2048	224	2018

Table 1: Problem Sizes for Different Values of ε

We see that the sizes of $\mathsf{LP}(\varepsilon)$ are considerable larger that the sizes of CPP. On the other hand we confirm that sizes only grow logarithmically with ε.

We additionally devised the following simple computational experiment for selecting the appropriate value of ε. For n equal to 20 and 30 we selected the first 10 instances of each instance class and tried to solve them with values of ε in $\{1, 0.1, 0.01, 0.001, 0.0001\}$. A time limit of 100 seconds was set. Note that $\varepsilon = 1$ is a value that we would probably never select, but we decided to test it anyway to illustrate that the procedure works even for this extreme choice of ε.

Table 2 shows the minimum, average, maximum and standard deviation of the number of nodes needed by $\mathsf{LP}(\varepsilon)$-BB to solve the instances. Tables 3 and 4 show the same statistics for solve times in seconds and the number of branch-and-bound nodes in which nonlinear relaxation $\mathsf{CPP}(l^k, u^k)$ is solved.

Figure 2 shows the performance profile (see Dolan and Moré (2002)) for all the instances

stat	$\varepsilon = 1$	$\varepsilon = 0.1$	$\varepsilon = 0.01$	$\varepsilon = 0.001$	$\varepsilon = 0.0001$
min	0	0	0	0	0
avg	**7760**	**1497**	**166**	**193**	**239**
max	36443	14281	2390	2228	3995
std	3087	1342	196	303	289

Table 2: Number of Nodes for Different Values of ε

stat	$\varepsilon = 1$	$\varepsilon = 0.1$	$\varepsilon = 0.01$	$\varepsilon = 0.001$	$\varepsilon = 0.0001$
min	0.14	0.07	0.09	0.12	0.18
avg	**37.18**	**3.71**	**1.10**	**3.19**	**5.79**
max	100.31	21.28	8.64	35.38	71.16
std	21.61	3.33	1.16	5.53	10.53

Table 3: Solve Time for Different Values of ε [s]

stat	$\varepsilon = 1$	$\varepsilon = 0.1$	$\varepsilon = 0.01$	$\varepsilon = 0.001$	$\varepsilon = 0.0001$
min	1	1	1	0	0
avg	**1700**	**74**	**4**	**3**	**3**
max	6178	367	18	22	23
std	436	25	2	3	1

Table 4: Number of Nodes that Solve $\mathsf{CPP}(l^k, u^k)$

using solve time as a performance metric. For a given value m on the horizontal axis and a given method, the value f plotted in the performance profile indicates the fraction f of the instances that were solved by that method within m times the length of time required by the fastest method for each instance. In particular, the intercepts of the plot (if any) with the vertical axis to the left and right indicate the fraction of the instances for which the method was the fastest and the fraction of the instances which the method could solve in the alloted time, respectivelly. For example, the method for $\varepsilon = 1$ was the fastest in about 5% of the instances, could solve almost 80% of the instances and could solve under 30% of the instances within 10 times the length of time required by the fastest solver. We refer the reader to Dolan and Moré (2002) for more details about the construction and interpretation of performance profiles.

We see that $\varepsilon = 0.01$ is the best choice on average. It also has the best performance profile and it yields the fastest method for 80% of the instances. Furthermore, for $\varepsilon = 0.01$ we have very few nodes solving $\mathsf{CPP}(l^k, u^k)$.

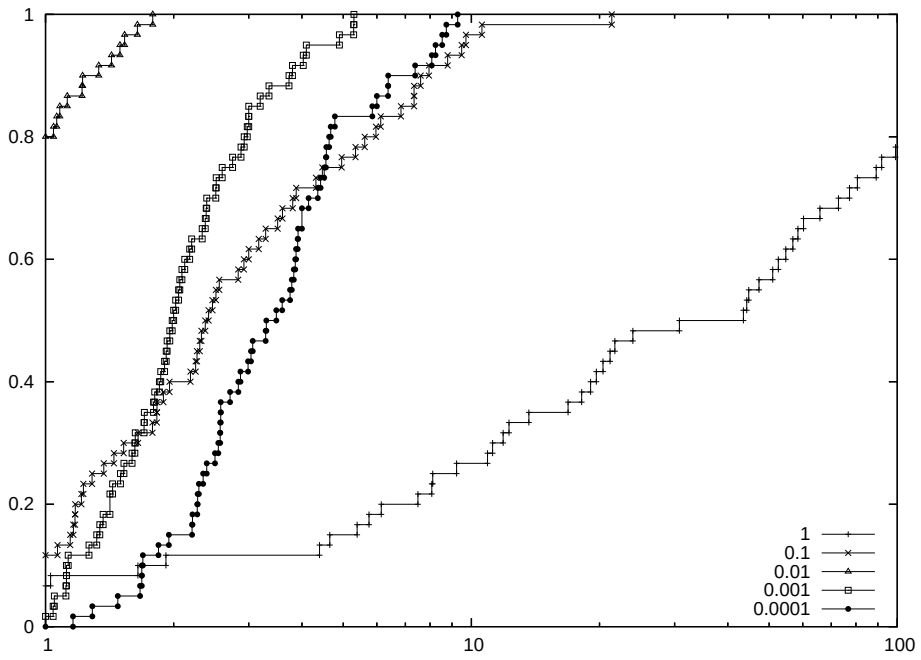

Figure 2: Performance Profile for Different Values of ε

It is also interesting to note that the procedure still works fairly well for values of ε as big as 0.1 and even for the extreme case of $\varepsilon = 1$ the procedure was still able to solve almost 80% of the instances in the alloted time of 100 seconds. For this last case though, the high number of nodes that solve $\mathsf{CPP}(l^k, u^k)$ makes the algorithm behave almost like an NLP based branch-and-bound algorithm.

Finally, we note that the result that $\varepsilon = 0.01$ requires the smallest number of branch-and-bound nodes on average is somewhat unexpected. This contradicts the belief that the algorithm of Figure 1 should require fewer branch-and-bound nodes if tighter relaxations are used. An explanation of this apparent contradiction comes from the difference between the algorithm of Figure 1 and the implementation of $\mathsf{LP}(\varepsilon)$-BB as discussed in section 4.1, since the use of CPLEX's advanced features in $\mathsf{LP}(\varepsilon)$-BB makes it hard to predict the behavior of the algorithm. In particular, slightly larger problems might cause CPLEX's default limits for advanced features such as preprocessing, heuristics, and cut generation to be reached quicker which could significantly reduce their effectiveness.

4.3.2. Performance of LP(ε)-BB against other methods

In this section we compare LP(ε)-BB with $\varepsilon = 0.01$ against other solvers. The solvers we choose for the comparison are the NLP based branch-and-bound algorithms CPLEX 10 MICP solver and Bonmin's I-BB and the polyhedral relaxation based algorithms I-QG and I-Hyb from Bonmin. We did not include Bonmin's I-OA algorithm as it performed very badly in preliminary tests.

For CPLEX we used its default settings and for the Bonmin solvers we set parameters `allowable_gap` and `allowable_fraction_gap` to 10^{-6} and 10^{-4} respectively to have the same target gaps as CPLEX. All tests were run with a time limit of 10000 seconds.

We first tested all solvers for all instances for n in $\{20, 30\}$. We denote this set of instances as the *small instances*. Table 5 shows the minimum, average, maximum and standard deviations of the solve times. Figure 3 shows the performance profile for all the instances using solve time as a performance metric.

From Table 5 we see that LP(ε)-BB is the fastest algorithm on average for all but one set of instances and that this average can be up to five times better than the average for its closest competitor. Furthermore, as the standard deviation and maximum numbers show, LP(ε)-BB is far more consistent in providing good solve times than the other methods. From Figure 3 we can also see that LP(ε)-BB has the best performance profile, that it is the fastest solver in 60% of the instances and that it is almost never an order of magnitude slower than the best solver.

Our second set of tests include all instances for n in $\{40, 50\}$. We denote this set of instances as the *medium instances*. We did not include I-QG in these tests as it performed very poorly on the instances for $n = 30$ and reached the time limit in several instances. Although I-Hyb performed close to I-QC we included it in these tests as it had only reached the time limit in one instance and we wanted to have at least one of the original LP/NLP based branch-and-bound solvers in our tests. Table 6 shows the minimum, average, maximum and standard deviation of the solve times. Figure 4 shows the performance profile for all the instances using solve time as a performance metric.

We see from Table 6 that LP(ε)-BB is now the fastest algorithm on average for all instances and this average can be up to six times better than the average for its closest competitor. Again, as the standard deviation and maximum numbers show, LP(ε)-BB is far more consistent in providing good solve times than the other methods. From Figure 4

instance(n)	stat	LP(ε)-BB	I-QG	I-Hyb	I-BB	CPLEX
classical(20)	min	0.08	0.38	0.22	0.28	0.02
	avg	**0.29**	**26.41**	**24.84**	**1.28**	**1.31**
	max	1.06	222.19	164.71	13.33	7.95
	std	0.18	42.92	26.37	2.31	1.17
classical(30)	min	0.25	1.62	0.33	0.38	0.73
	avg	**1.65**	**1434.86**	**217.25**	**13.19**	**9.68**
	max	27.0	10005.2	10003.3	573.97	324.63
	std	3.21	2768.34	1016.68	59.17	33.68
shortfall(20)	min	0.19	0.18	0.26	0.34	0.03
	avg	**0.48**	**17.42**	**16.78**	**0.63**	**1.68**
	max	1.65	174.62	58.45	3.52	5.19
	std	0.21	30.77	17.96	0.52	0.89
shortfall(30)	min	0.4	1.25	0.57	0.47	1.26
	avg	**2.20**	**847.63**	**136.39**	**5.00**	**9.26**
	max	29.34	10003.1	5907.32	73.81	80.36
	std	3.21	1992.86	588.61	10.00	12.20
robust(20)	min	0.19	0.39	0.12	0.37	0.03
	avg	**0.39**	**4.99**	**15.51**	**2.57**	**1.03**
	max	1.05	33.85	599.46	57.22	3.5
	std	0.20	5.60	60.37	10.25	0.90
robust(30)	min	0.43	0.59	0.29	0.48	0.07
	avg	**1.20**	**75.07**	**23.43**	**1.02**	**3.54**
	max	4.72	2071.47	134.08	4.92	10.76
	std	0.81	284.39	25.49	0.87	2.45

Table 5: Solve Times for Small Instances [s]

we again see that LP(ε)-BB has the best performance profile, that it is the fastest solver in over 60% of the instances and that it is almost never an order of magnitude slower than the best solver. Moreover, it is the only solver with this last property.

Our last set of tests include instances for n in $\{100, 200\}$. We denote this set of instances as the *large instances*. We do not include the results for I-Hyb as it was not able to solve any of the instances in this group. Neither did we include results for the classical or shortfall instances for $n = 200$ as none of the methods could solve a single instance in the alloted time. Table 7 shows the minimum, average and maximumsome of the solve times. We did not include standard deviations as time limits were reached for too many instances. We instead include the number of instances (out of a total of 10 per instance class) that each method could solve in the alloted time. Figure 5 shows the performance profile for all the instances using solve time as a performance metric.

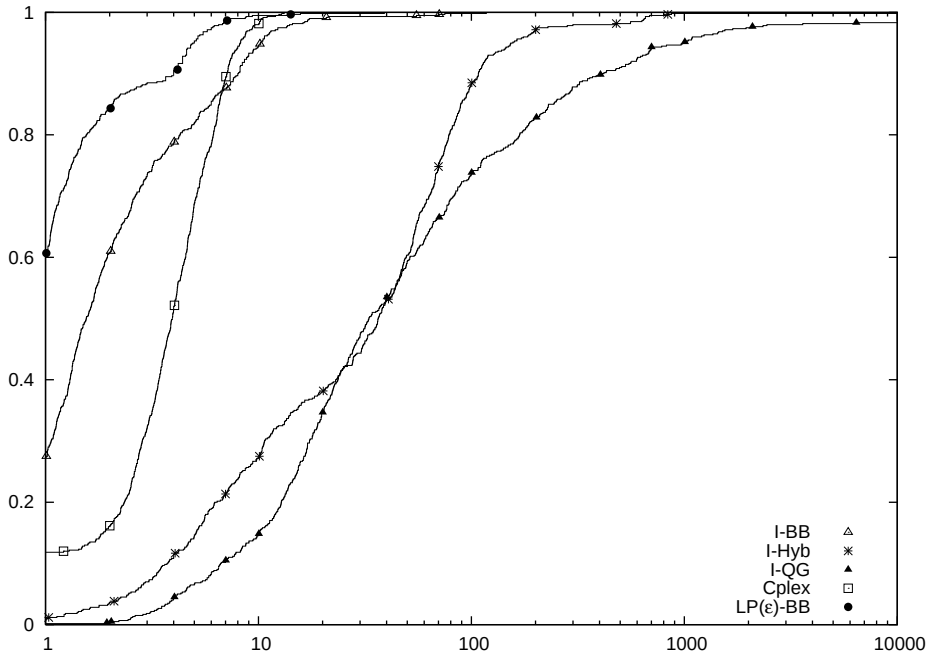

Figure 3: Performance Profile for Small Instances

From Table 7 we see that LP(ε)-BB is the fastest algorithm on average for all but one set of instances in this group. Furthermore, for all instance classes it is the method that solves the largest number of instances. From Figure 5 we can also see that LP(ε)-BB has the best performance profile, that it is the fastest solver in about 40% of the instances and that it is the method that is able to solve the greatest number of instances in the alloted time.

4.3.3. Factors that affect the effectiveness of LP(ε)-BB

In this subsection we study some factors that might affect the effectiveness of LP(ε) including solve times, accuracy and size of the polyhedral relaxation LP(ε), number of branch-and-bound nodes processed and number of calls to the nonlinear relaxations.

We begin by confirming the results from Glineur (2000) that solving the polyhedral relaxation LP(ε) is slower than solving CPP directly still hold for our tests instances. To confirm this we solved CPP with CPLEX 10's barrier solver and LP(0.01) with CPLEX 10's primal simplex, dual simplex and LP barrier solvers. We did this for all instances for n equal to 100 and 200. Table 8 shows the minimum, average, maximum and standard deviation of the solve times.

instance(n)	stat	LP(ε)-BB	I-Hyb	I-BB	CPLEX
classical(40)	min	0.56	35.04	0.61	1.55
	avg	**14.84**	**1412.23**	**144.17**	**63.41**
	max	554.52	10006.0	8518.95	2033.65
	std	56.64	2631.92	848.84	208.86
classical(50)	min	0.76	35.17	0.75	4.12
	avg	**102.88**	**4139.92**	**894.00**	**636.83**
	max	1950.81	12577.8	10030.1	10000.0
	std	270.96	4343.71	2048.96	1626.37
shortfall(40)	min	1.17	34.72	0.7	4.93
	avg	**16.60**	**956.98**	**92.85**	**111.97**
	max	389.57	10004.6	4888.26	4259.5
	std	43.85	2133.56	489.98	430.95
shortfall(50)	min	1.58	33.22	0.96	5.69
	avg	**163.10**	**3143.84**	**452.05**	**567.74**
	max	7674.86	10006.0	10034.1	10000.0
	std	771.98	3803.14	1285.52	1319.39
robust(40)	min	0.51	0.43	0.69	0.14
	avg	**3.82**	**59.10**	**4.31**	**11.17**
	max	42.57	1141.91	129.82	160.71
	std	6.04	130.37	14.64	18.58
robust(50)	min	0.92	0.65	0.93	0.25
	avg	**20.44**	**435.43**	**23.67**	**41.71**
	max	443.29	10002.1	746.37	876.31
	std	63.47	1702.15	95.68	120.24

Table 6: Solve Times for Medium Instances [s]

We see that solving LP(0.01) is slower than solving CPP. In fact, solving LP(0.01) with dual simplex is more than twice as slow as solving CPP with barrier. Hence, it is not the solve times of a single relaxation that gives the LP(ε)-BB algorithm the advantage over the NLP based branch-and-bound algorithms. Note that the "warm start" capabilities of an LP solver might still make solving a series of similar LP(0.01) problems faster than solving a series of similar CPP problems.

It is noted in Glineur (2000) that although the relaxation used to construct LP(ε) gives accuracy guarantees on $\mathcal{L}^r_\varepsilon$, it is in general not possible to give *a priori* guarantees on the accuracy of LP(ε) or its optimal objective value $z_{\mathsf{LP}(\varepsilon)}$. For this reason we studied empirically the accuracy of $z_{\mathsf{LP}(\varepsilon)}$ on our set of test instances. To do this we calculated the value of the accuracy measure $100 \times (z_{\mathsf{LP}(\varepsilon)} - z_{\mathsf{CPP}})/z_{\mathsf{CPP}}$ for all of our test instances and for values of ε in $\{1, 0.1, 0.01, 0.001, 0.0001\}$. Table 9 shows the minimum, average, maximum and standard

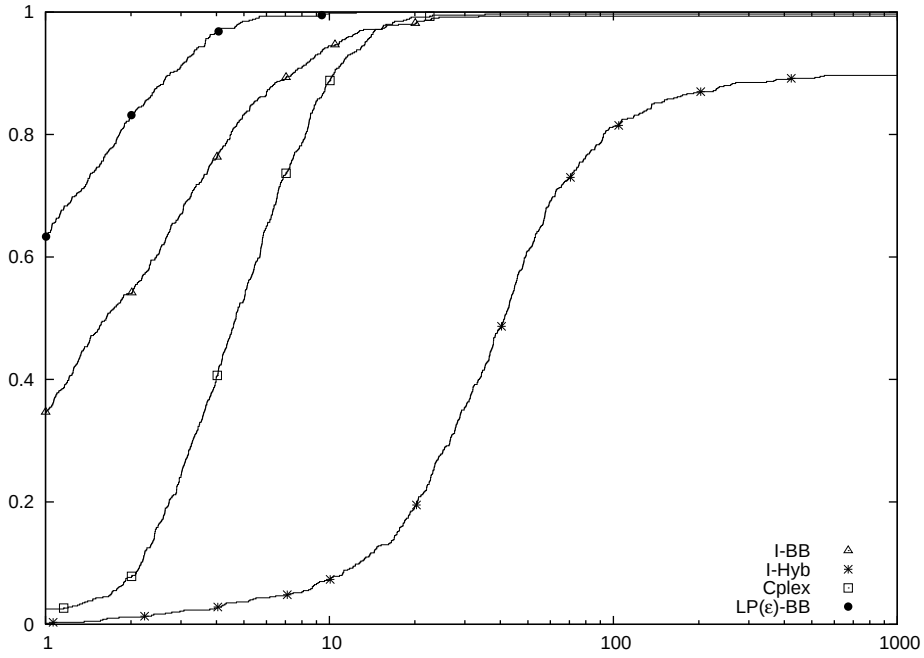

Figure 4: Performance Profile for Medium Instances

instance(n)	stat	LP(ε)-BB	I-BB	CPLEX
classical(100)	min	1653	3497	4503
	avg	**7443**	**8605**	**8767**
	max	10012	10035	10000
	solved	4	3	3
shortfall(100)	min	2014	4105	8733
	avg	**6660**	**8497**	**9818**
	max	10003	10163	10000
	solved	6	4	2
robust(100)	min	30	4	85
	avg	**956**	**612**	**1395**
	max	4943	2684	5294
	solved	10	10	10
robust(200)	min	1458	1775	9789
	avg	**6207**	**7346**	**9979**
	max	10138	10016	10000
	solved	6	5	1

Table 7: Solve Times for Large Instances [s]

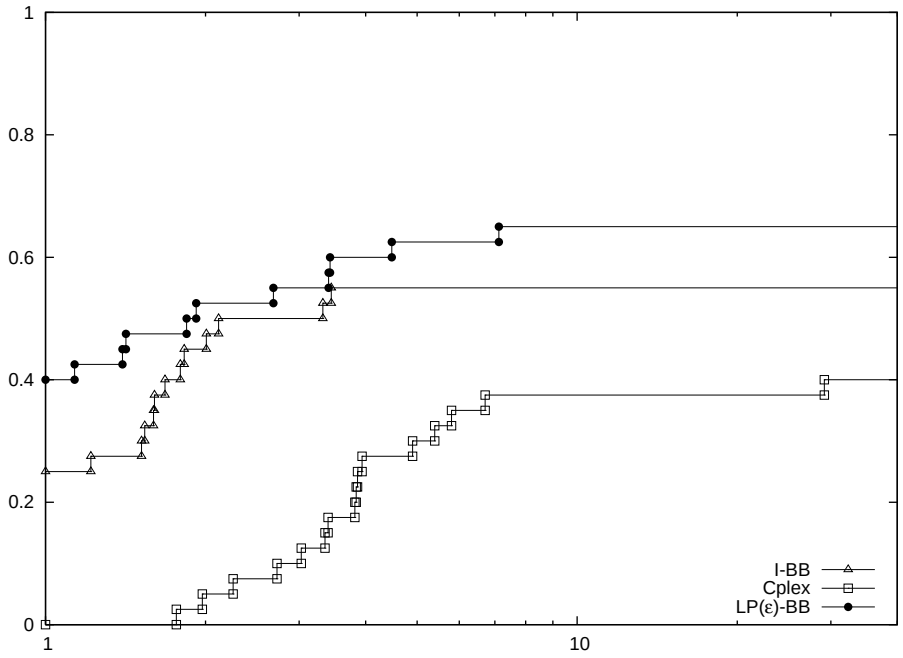

Figure 5: Performance Profile for Large Instances

	CPP	LP(0.01)		
stat	Barrier	P. Simplex	D. Simplex	Barrier
min	0.09	0.72	0.58	0.19
avg	**4.88**	**34.94**	**11.66**	**6.94**
max	20.39	174.45	49.69	29.7
std	1.50	10.45	2.86	2.05

Table 8: Solve Time for Root Relaxation [s]

deviation of this accuracy measure.

We see that even for the extreme case $\varepsilon = 1$ the accuracy of $z_{\mathsf{LP}(\varepsilon)}$ is fairly good and for our chosen value of $\varepsilon = 0.01$ the accuracy is extremely good. This accuracy is likely one of the reasons for the effectiveness of the $\mathsf{LP}(\varepsilon)$-BB algorithm.

We next study the sizes of $\mathsf{LP}(0.01)$ as n varies. Table 10 presents the number of columns, rows and non-zero coefficients for different values of n. Again, we also include the sizes of CPP. We see that $\mathsf{LP}(0.01)$ can be up to about an order of magnitude larger than CPP. However, we also confirm that the size of $\mathsf{LP}(0.01)$ only grows linearly with n.

Finally, we study the number of branch-and-bound nodes processed and the number of calls to the nonlinear relaxations. We begin by comparing $\mathsf{LP}(\varepsilon)$-BB to the two NLP based

stat	$\varepsilon = 1$	$\varepsilon = 0.1$	$\varepsilon = 0.01$	$\varepsilon = 0.001$	$\varepsilon = 0.0001$
min	0.25	0.07	0.00	0.00	0.00
avg	**4.30**	**1.14**	**0.07**	**0.00**	**0.00**
max	13.94	5.16	0.29	0.02	0.01
std	0.80	0.21	0.01	0.00	0.00

Table 9: Accuracy of Relaxation $z_{\mathsf{LP}(\varepsilon)}$ [%]

		classical		shortfall		robust	
n	Relaxation	cols+rows	nz	cols+rows	nz	cols+rows	nz
20	CPP	105	501	150	968	154	948
	LP(0.01)	769	1685	1478	3336	1556	3448
30	CPP	155	1051	220	2048	224	2018
	LP(0.01)	1169	2859	2248	5664	2326	5766
40	CPP	205	1801	290	3528	294	3488
	LP(0.01)	1574	4242	3028	8410	3116	8520
50	CPP	255	2751	360	5408	364	5358
	LP(0.01)	1979	5825	3808	11556	3886	11638
100	CPP	505	10501	710	20808	714	20708
	LP(0.01)	3994	16722	7688	33250	7766	33282
200	CPP	1005	41001	1410	81608	1414	81408
	LP(0.01)	8024	53516	15448	106638	15536	106588

Table 10: Problem Sizes for Different Values of n

branch-and-bound solvers. To do this, we selected all instances that solved within the time limit by I-BB, CPLEX and LP(ε)-BB . For these instances we present in Table 11 the total number of branch-and-bound nodes processed by each method and the total number of calls to nonlinear relaxations CPP(l^k, u^k) and CPP$(\hat{x})$ made by LP(ε)-BB .

We see that the total number of calls to the nonlinear relaxations made by LP(ε)-BB is only around 1% of the total number of branch-and-bound nodes processed. This is another reason for the effectiveness of LP(ε)-BB as it has to solve very few expensive nonlinear relaxations. An interesting observation is that CPP$(\hat{x})$ is solved more times than CPP(l^k, u^k). This is expected as CPLEX usually finds most of the integer feasible solutions with its primal heuristic than at integer feasible nodes. On the other hand, the fact that that LP(ε)-BB processed fewer branch-and-bound nodes than the two NLP based branch-and-bound methods is somewhat unexpected. Because LP(ε) is a relaxation of CPP we would expect that a pure branch-and-bound algorithm based on LP(ε) should process at least the same number of nodes as an algorithm based on CPP. We believe that the reason for this

B-and-b nodes I-BB	3007029
B-and-b nodes CPLEX	3224115
B-and-b nodes $\mathsf{LP}(\varepsilon)$-BB	2027332
$\mathsf{LP}(\varepsilon)$-BB calls to $\mathsf{CPP}(l^k, u^k)$	5818
$\mathsf{LP}(\varepsilon)$-BB calls to $\mathsf{CPP}(\hat{x})$	17784

Table 11: Total Number of Nodes and Calls to Relaxations for All Instances

unexpected behavior is that CPLEX is not a pure branch-and-bound solver. $\mathsf{LP}(\varepsilon)$-BB benefits from CPLEX being able to use some features which are currently available for MILP problems, but not for MICP problems, such as advanced preprocessing, branching rules, cutting planes and heuristics.

Finally, we compared $\mathsf{LP}(\varepsilon)$-BB to the polyhedral relaxation based solvers by selecting all instances that were solved within the time limit by I-QG, I-Hyb and $\mathsf{LP}(\varepsilon)$-BB . For these instances we present in Table 12 the total number of branch-and-bound nodes processed by each method. Because the instances used to generate Table 12 are not the same as the ones used to generate Table 11 we also include, as a reference, the total number of branch-and-bound nodes processed by I-BB and CPLEX and the total number of calls to nonlinear relaxations $\mathsf{CPP}(l^k, u^k)$ and $\mathsf{CPP}(\hat{x})$ made by $\mathsf{LP}(\varepsilon)$-BB .

B-and-b nodes I-QG	3580051
B-and-b nodes I-Hyb	328316
B-and-b nodes I-BB	68915
B-and-b nodes CPLEX	85957
B-and-b nodes $\mathsf{LP}(\varepsilon)$-BB	57933
$\mathsf{LP}(\varepsilon)$-BB calls to $\mathsf{CPP}(l^k, u^k)$	2305
$\mathsf{LP}(\varepsilon)$-BB calls to $\mathsf{CPP}(\hat{x})$	7810

Table 12: Total Number of Nodes and Calls to Relaxations for Small Instances

We see that I-Hyb needed almost four times the number of nodes needed by CPLEX and I-QG needed over 40 times as many nodes as CPLEX. In contrast, $\mathsf{LP}(\varepsilon)$-BB was the algorithm that needed the fewest number of nodes. This confirms that the relaxation $\mathsf{LP}(\varepsilon)$ is extremely good for our set of instances.

5. Conclusions and Further Work

We have introduced a branch-and-bound algorithm for MINLP problems that is based on a lifted polyhedral relaxation, does not update the relaxation using gradient inequalities and sometimes branches on integer feasible variables. We have also demonstrated how this lifted LP branch-and-bound algorithm can be very effective when a good lifted polyhedral relaxation is available. More specifically, we have shown that the lifted LP branch-and-bound algorithm based on $\mathsf{LP}(\varepsilon)$ can significantly outperform other methods for solving a series of portfolio optimization problems with cardinality constraints. One reason for this good performance is that, for these problems, high accuracy of $\mathcal{L}_\varepsilon^r$ translates into high accuracy of $\mathsf{LP}(\varepsilon)$ which results in the construction of a tight but small polyhedral relaxation. Another factor is that by using a polyhedral relaxation of the nonlinear constraints we can benefit from "warm start" capabilities of the simplex LP algorithm and the many advanced features of CPLEX's MILP solver. It is curious to note that a statement similar to this last one can also be made for the other polyhedral relaxation based algorithms we tested and these were the worst performers in our tests. It seems then that using $\mathsf{LP}(\varepsilon)$ provides a middle point between NLP based branch-and-bound solvers and polyhedral relaxation based solvers which only use gradient inequalities by inheriting most of the good properties of this last class without suffering from slow convergence of the relaxations.

Although the lifted LP branch-and-bound algorithm based on $\mathsf{LP}(\varepsilon)$ we have presented is already very efficient there are many improvements that can be made to it. While the version of $\mathsf{LP}(\varepsilon)$ that we used achieves the best possible assymptotic order of magnitude of variables and constraints (see Ben-Tal and Nemirovski (2001b) and Glineur (2000)), it is shown in Glineur (2000) that for a fixed r and ε it can be improved further. Using a slightly smaller version of $\mathsf{LP}(\varepsilon)$ would probably not increase significantly the performance of the algorithm for our test instances, but it could provide an advantages for problems with many conic constraints of the form $\mathcal{CC}$.

The choice of value ε for $\mathsf{LP}(\varepsilon)$ is another aspect that can be studied further. The dependence of $\mathsf{LP}(\varepsilon)$ on ε is through the function $\left\lceil \log_4 \left(\frac{16}{9} \pi^{-2} \log(1+\varepsilon) \right) \right\rceil$ in (20). Hence, there is only a discrete set of possible of choices of ε in a certain interval that yield different relaxations $\mathsf{LP}(\varepsilon)$. This allows for a refinement of the calibration experiments of section 4.3.1. For example, in our calibration experiment the only different relaxations $\mathsf{LP}(\varepsilon)$ for values of ε in $[0.001, 0.1]$ are the ones corresponding to values of ε in $\{0.1, 0.03, 0.01, 0.004, 0.001\}$.

By re-running our calibration experiments for all of these values of ε we discovered that $\varepsilon = 0.01$ was still the best choice on average. This suggests the existence of, in some sense, an optimal choice of ε. The choice of this ε could become more complicated though when the more elaborate constructions of Glineur (2000) are used. An alternative to choosing ε *a priori* is to choose a moderate initial value and refine the relaxation inside the branch-and-bound procedure. It is not clear how to do this efficiently though.

We are currently studying some of these issues and the possibility of extending this work to other classes of MINLP problems.

Acknowledgments

This research was supported in part by the National Science Foundation under grants DMI-0121495, DMI-0522485 and DMI-0133943 and by a grant from ExxonMobil.

References

Abhishek, K., S. Leyffer, J. T. Linderoth. 2006. Filmint: An outer-approximation-based solver for nonlinear mixed integer programs. Preprint ANL/MCS-P1374-0906, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL.

Ball, K. M. 1997. An elementary introduction to modern convex geometry. S. Levy, ed., *Flavors of Geometry*, *Mathematical Sciences Research Institute Publications*, vol. 31. Cambridge University Press, Cambridge, 1–58.

Ben-Tal, A., A. Nemirovski. 1999. Robust solutions of uncertain linear programs. *Oper. Res. Lett.* **25** 1–13.

Ben-Tal, A., A. Nemirovski. 2001a. *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Ben-Tal, A., A. Nemirovski. 2001b. On polyhedral approximations of the second-order cone. *Math. Oper. Res.* **26** 193–205.

Bertsimas, D., R. Shioda. 2004. An algorithm for cardinality constrained quadratic optimization problems. Working Paper, http://web.mit.edu/dbertsim/www/papers.html.

Bienstock, D. 1996. Computational study of a family of mixed-integer quadratic programming problems. *Math. Program.* **74** 121–140.

Bonami, P., A. Waechter, L. T. Biegler, A. R. Conn, G. Cornuejols, I. E. Grossmann, C. D. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya. 2005. An algorithmic framework for convex mixed integer nonlinear programs. IBM Research Report RC23771, IBM, Yorktown Heights, NY.

Borchers, B., J. E. Mitchell. 1994. An improved branch and bound algorithm for mixed integer nonlinear programs. *Comput. Oper. Res.* **21** 359–367.

Ceria, S., R. A. Stubbs. 2006. Incorporating estimation errors into portfolio selection: Robust portfolio construction. *J. Asset Manage.* **7** 109–127.

Chang, T.-J., N. Meade, J. E. Beasley, Y. M. Sharaiha. 2000. Heuristics for cardinality constrained portfolio optimisation. *Comput. and Oper. Res.* **27** 1271–1302.

Dolan, E. D., J. J. Moré. 2002. Benchmarking optimization software with performance profiles. *Math. Program.* **91** 201–213.

Duran, M. A., I. E. Grossmann. 1986. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Math. Program.* **36** 307–339.

Fletcher, R., S. Leyffer. 1994. Solving mixed integer nonlinear programs by outer approximation. *Math. Program.* **66** 327–349.

Geoffrion, A. 1972. Generalized benders decomposition. *J. of Optim. Theory and Appl.* **10** 237–260.

Glineur, F. 2000. Computational experiments with a linear approximation of second order cone optimization. Image Technical Report 0001, Service de Mathématique et de Recherche Opérationnelle, Faculté Polytechnique de Mons, Mons, Belgium.

Grossmann, I. E. 2002. Review of nonlinear mixed-integer and disjunctive programming techniques. *Optim. and Engrg.* **3** 227–252.

Gupta, O. K., A. Ravindran. 1985. Branch and bound experiments in convex nonlinear integer programming. *Manage. Sci.* **31** 1533–1546.

ILOG. 2005. *Cplex 10: User's Manual and Reference Manual*. ILOG, S.A.

Leyffer, S. 2001. Integrating SQP and branch-and-bound for mixed integer nonlinear programming. *Comput. Optim. and Appl.* **18** 295–309.

Lobo, M. S., M. Fazel, S. Boyd. 2007. Portfolio optimization with linear and fixed transaction costs. *Ann. Oper. Res.* **To appear**.

Lobo, M. S., L. Vandenberghe, S. Boyd. 1998. Application of second-order cone programming. *Linear Algebra Appl.* **284** 193–228.

Maringer, D., H. Kellerer. 2003. Optimization of cardinality constrained portfolios with a hybrid local search algorithm. *OR Spectrum* **25** 481–495.

Quesada, I., I.E. Grossmann. 1992. An lp/nlp based branch and bound algorithm for convex minlp optimization problems. *Comput. and Chem. Engrg.* **16** 937–947.

Stubbs, R. A., S. Mehrotra. 1999. A branch-and-cut method for 0-1 mixed convex programming. *Math. Program.* **86** 515–532.

Westerlund, T., F. Pettersson. 1995. An extended cutting plane method for solving convex minlp problems. *Comput. and Chem. Engrg.* **19** S131–S136.

Westerlund, T., F. Pettersson, I.E. Grossmann. 1994. Optimization of pump configurations as a minlp problem. *Comput. and Chem. Engrg.* **18** 845–858.