Mathematical Programming 74 (1996) 121–140

Computational study of a family of mixed-integer quadratic programming problems

Daniel Bienstock [1]

Department of Industrial Engineering and Operations Research, Columbia University, Room 331 Mudd, New York, NY 10027-6699, USA

Received 1 February 1995; revised manuscript received 31 July 1995

Abstract

We present computational experience with a branch-and-cut algorithm to solve quadratic programming problems where there is an upper bound on the number of positive variables. Such problems arise in financial applications. The algorithm solves the largest real-life problems in a few minutes of run-time.

Keywords: Branch-and-cut; Mixed-integer; Quadratic programming; Parallel computing

1. Introduction

We are interested in optimization problems **QMIP** of the form:

$$\min\ x^{\mathrm{T}}Qx + c^{\mathrm{T}}x,$$

s.t.

$$Ax \le b, \tag{1}$$

$$|\mathrm{supp}(x)| \le K, \tag{2}$$

$$0 \le x_j \le u_j, \quad \text{all } j, \tag{3}$$

where x is an n-vector, Q is a symmetric positive-semidefinite matrix, $\mathrm{supp}(x) = \{j : x_j > 0\}$ and K is a positive integer. Problems of this type are of interest in portfolio optimization. Briefly, variables in the problem correspond to commodities to be bought,

[1] Email: dano@ieor.columbia.edu.

the objective is a measure of "risk", the constraints (1) prescribe levels of "performance", and constraint (2) specifies that not too many different types of commodities can be chosen. All data is derived from statistical information. A good deal of previous work has focused on solving the continuous version of the above problem, that is, without constraint (2). Using this constraint, the feasible region becomes a mixed-integer set, and in this paper we describe a branch-and-cut algorithm for the resulting problem. We note that in practice this problem may include additional mixed-integer constraints. For example, there may be *minimum transaction levels*. This constraint states that if a variable is positive, then it must be "large enough", for example, at least 0.01. This kind of constraint can easily be incorporated into a branch-and-cut framework, and we will report on computational results involving it as well. See [13] for extensive background on these optimization problems, known generically as "mean-variance" models.

The problems that arise in practical applications have several important properties. The number of variables is typically in the thousands, but not very large—at most 10000, say, while K will be far smaller, say 20 to 50. The matrix A has few rows, and most of these are totally or almost totally dense, and badly conditioned.

1.1. Structure of problems

Most important is the fact that while the matrix Q is totally dense, it has very special structure. In fact, as we discuss below, Q frequently has small rank (say, at most 20). In fact, one knows an explicit factorization of Q as a product of low-rank matrices. As has been observed by several other authors (see [10, 16]) by adding new variables and some additional constraints the quadratic part of the objective may then be reformulated as:

$$\min y^{\mathrm{T}} M y, \tag{4}$$

with y an r-vector (r = rank of Q) of additional variables and M an $r \times r$ symmetric positive-definite matrix, which is of course preferable.

In an older and different version of the mean-variance model (see [13]), the objective quadratic has rank n and through some manipulation similar to the above it can be written as

$$x^{\mathrm{T}} D x + z^{\mathrm{T}} N z, \tag{5}$$

where D is a nonnegative diagonal matrix, and (again) z is a small vector of added variables and N is a symmetric positive-definite matrix. However, recent work ([10, 16]) has shown that a mean-variance problem can always be reformulated with an objective of the form (4), with a different constraint matrix A.

As of this writing, most of the real-life data available to us is of the compact form (4). The non-compact problems we obtained have been much smaller and in fact, much easier to solve. Moreover, the pure quadratic programs behaved just like those arising from the compact problems, lending experimental backing to the results in [10, 16].

To challenge our algorithm, we artificially constructed high-rank problems by adding a large diagonal term to the quadratic of some of our problems. It is not clear which of the two versions will yield a more difficult mixed-integer program. On the one hand, the large-rank version should give rise to more difficult quadratic programs, which may also have optimal solutions with many positive and small entries. Further, it is likely that the optima may be in the integer hull, so that no cutting is possible, resulting in larger enumeration trees (which we observed with our artificial problems). On the other hand, part of the price of using the compact formulation with low-rank objective is that the linear constraints become badly behaved and make cutting numerically difficult. Typically the ratio of the largest to the smallest entry on a row is of the order of 10^5, even allowing for column scaling. Furthermore, the objective is very "flat" for these problems, in the sense that the value of the mixed-integer program is the same as the value of the initial quadratic program to at least seven digits of accuracy. In the course of the algorithm the lower bound essentially remains constant. We will elaborate on this later. At any rate, the code described in this paper has been primarily optimized for solving compact problems.

1.2. Difficulty of problem

As stated, the problem may appear deceptively easy, especially when in the compact form. If q is the rank of the objective, and r is the rank of the constraint matrix, then after solving the initial quadratic program we may expect to have roughly $r + q$ variables strictly between bounds, which in the compact version will not be very large. More important, however, is the set of variables attaining their upper bound. This may well be the "wrong set". In fact it may even be the case that the set of variables that are positive in the solution to the initial quadratic program may not contain a mixed-integer feasible subset (testing this condition is NP-hard, see below) although this is rare. What is not rare is that some of the variables at their upper bounds have to be decreased. In any case, for the problems we handled the quantity K was between 20 and 25, whereas the number of positive variables in the continuous relaxation was between 35 and 50.

Now consider a set of constraints of the form,

$$\text{(P)}\quad \begin{aligned} &Ax \geq b, && (6)\\ &|\text{supp}(x)| \leq K, && (7)\\ &0 \leq x_j \leq 1, \quad \text{all } j. && (8) \end{aligned}$$

where one of the constraints implied by (6) is $\sum_j x_j \geq K$. It follows that every feasible solution for this system is $\{0, 1\}$-valued. For many combinatorial optimization problems the number of variables set to 1 is the same in all feasible solutions, and known. This argument shows that a (linear) optimization problem with feasible set of type (P) is NP-hard. More to the point, we have:

Theorem 1. *Testing feasibility of a system (P) is NP-complete even if A has three rows.*

Proof. Consider a "partition"-like problem of the type: "given $2n$ integers $a_1, a_2, \ldots, a_{2n}$, is there a partition of them into two sets with n elements each and with equal sums?" This problem is known to be NP-complete. Let $S = \frac{1}{2}\sum_i a_i$. We can write the partition problem in the form (P), where the constraints are $\sum_i a_i x_i \leq S$, $\sum_i a_i x_i \geq S$, and $\sum_i x_i \geq n$; and $K = n$. This concludes the proof. Clearly the difficulty of the problem is due to badly behaved numbers a_i. □

2. Formulation and branching

One possible approach for solving problem QMIP is to add $\{0, 1\}$-variables z_j, together with the constraints

$$x_j \leq u_j z_j, \tag{9}$$

$$\sum_j z_j \leq K \tag{10}$$

and then study the polyhedral structure of the resulting mixed-integer set. We feel that this is not desirable, however, not just because it doubles the number of variables, but because it will drastically raise the rank of the constraint matrix, especially after adding cutting planes. As a result, the quadratic programs to be solved will become much more difficult. It is worth noting that the quadratic programs arising directly from QMIP are not particularly difficult to solve, especially those with low quadratic rank. Using an appropriate code, the largest require only a few seconds of computation, and reoptimization (for example, when forcing a variable to zero) takes much less than that.

The convexity of the objective implies that even with a "good" formulation, a certain amount of branching will be required, particularly when the objective is of low rank and as a result there are multiple optima for the mixed-integer program. Increasing the dimension of the (continuous relaxation of) the feasible region may only make things worse. As a result, our approach has been to preserve the fast solvability of the quadratic programs, even at the cost of increased branching. We model constraint (2) using the "surrogate" constraint,

$$\sum_j x_j/u_j \leq K, \tag{11}$$

which is clearly valid.

Branching is done as follows: at a given node, let F be the set of variables that have not been branched on, and let t denote the number of variables that have been branched "up". The surrogate constraint at this node becomes:

$$\sum_{j \in F} x_j/u_j \leq K - t. \tag{12}$$

If we choose variable $h \in F$ to branch on, then in the up branch the surrogate constraint will become

$$\sum_{j \in F-h} x_j/u_j \leq K - t - 1. \tag{13}$$

Notice that in an enumeration tree for this problem, a node can be fathomed as soon as it corresponds to K up branches. Experimentally, we have noticed that using (12), nodes are fathomed after a much smaller number of up branches, frequently due to infeasibility. In the current implementation we always branch down first, which seems to be the better choice.

As is standard, when we solve a node the underlying quadratic optimizer (which is primal feasible) reoptimizes using a warm start from the vector that gave rise to the node; initially this vector will usually be infeasible, and this is handled by assigning a quadratic penalty to the objective. For example, if we branch down on variable x_h, then we add to the objective a term of the form $\pi_h x_h^2$, where the parameter π_h is adjusted continuously in the course of the optimization (more on this in the Appendix). Once x_h has been reduced to 0 the optimization continues, with x_h fixed at 0. See [9, 12] for more background on this generic approach to handling infeasibilities. In general, we observed that the warm start speeded up optimization (as compared to optimization from scratch) by several orders of magnitude.

2.1. *Numerical issues*

The combination of a quadratic objective and the specific mixed-integer sets that we handle potentially leads to numerical difficulties. One possible "bad" scenario is the following: after solving a node, many variables are positive but very small, and a number of variables are positive but not very small. We note that this is unlikely to happen with a linear objective. In fact, if it happened most likely it would be due to roundoff errors, but it would be simple to recover from the errors by recomputing the last basis. With a quadratic objective it may well be that there is no error. Moreover, the larger the curvature of the objective the more likely it is that the optimal solution will show this behavior (and at the same time, the more difficult and more susceptible to numerical instability the quadratic program will be).

From a combinatorial viewpoint, it makes more sense to pick as branching variable one that is not very small. This way, generally the current solution will be infeasible in both branches. As a result of this strategy, variables set at very small values will be "pushed" down the branch-and-cut tree, i.e., the same bad scenario will reoccur at descendant nodes. As we continue branching, we may obtain nodes where this phenomenon is pronounced. This is undesirable as it makes difficult to make good branching choices and it renders cutting ineffective, as well as making the warm start reoptimizations more difficult. However, it may well be an unavoidable feature of the problem. Ideally, we may just leave the small variables alone in the hope that as we continue branching and strengthening the formulation the bad behavior will disappear. This is most often the case. On the other hand, we may need a systematic procedure for handling very bad nodes. Ideally, when at a node an excessive number of variables take small values, one would like to "compress" the solution into an equally good one with fewer small components. The "squeeze" heuristic that we describe later attempts to do just that, among other things.

There is another difficulty that is closely related to this, namely, when solving QMIP, when do we declare that a variable is positive? The underlying optimizer will use a small number ϵ as *infeasibility tolerance* for variables, that is, the largest amount by which a variable bound is allowed to be exceeded. Say $\epsilon = 10^{-9}$. At first glance, it may make sense to count a variable as positive precisely when its value exceeds ϵ. However, we feel that this is not the best approach, for two reasons. The first is that the infeasibility tolerance is a parameter that plays a fundamental role in the optimizer. Any algorithm that uses the optimizer as a subroutine should not rely on any kind of threshold of the same order of magnitude as ϵ (this may be particularly unstable when branching) the difficulty being how to handle values that are only one order of magnitude larger than ϵ, for example. A more important reason is purely practical: a practicioner would accept a much larger value γ as the threshold for being positive, say $\gamma = 10^{-6}$.

This approach leads to an obvious numerical difficulty. Namely, we are essentially operating with two definitions of zero, and we have to decide what to do with values between ϵ and γ. This is related to the problem described at the beginning of this section. What do we do if at a node several variables are of value between ϵ and γ? From a "practical" point of view we could apparently reduce them to zero, but doing so may well result on a small but numerically significant right-hand side infeasibility, or a similarly significant increase in the objective value.

In our experiments, these numerical difficulties were not significant. However, one possible way to handle them is the following: first scale all variables so that γ becomes much larger, say $\gamma = 10^{-2}$, and then treat γ as a minimum transaction level, i.e., when we branch up on a variable x_j, we impose the condition $x_j \geq \gamma$. The scale factor would be somewhat large, and as a result this approach could generate other numerical problems.

3. Cutting planes

We have considered four types of cutting planes for this problem: mixed-integer rounding inequalities, knapsack cuts, intersection cuts and disjunctive cuts. In the existing code we have only implemented the last kind, primarily because separation is straightforward, at least in theory, and also because of recent success with these inequalities in solving linear integer programs. We will describe the disjunctive cuts in a separate section. We refer the reader to [11] for background on polyhedral theory and cutting planes.

3.1. Mixed-integer rounding cuts

The mixed-integer rounding procedure (MIR) [11] is a general-purpose method for generating valid inequalities for mixed-integer programs. Roughly speaking, it involves taking a nonnegative linear combination of valid inequalities and strengthening it essentially by rounding the coefficients of the integer variables. This method has been very effective for combinatorial mixed-integer programs, where the knowledge of the structure of the problem suggests the multipliers to be used.

For problem QMIP, we would add to the formulation $\{0,1\}$-variables z_j that indicate when x_j is positive, and constraints (9), (10); use the MIR procedure to generate cuts, and then project these cuts back to x-space. The difficulty lies in how to automatically choose the multipliers, because the constraints we deal with have no particular structure. As an example, consider the system

$$2x_1 + 3x_2 + 4x_3 \geq 1, \tag{14}$$

$$4x_1 + 2x_2 + 5x_3 \geq 1, \tag{15}$$

$$6x_1 + 3x_2 + 2x_3 \geq 1, \tag{16}$$

$$|\mathrm{supp}(x)| \leq 2, \tag{17}$$

$$0 \leq x_j \leq 1, \quad \text{all } j. \tag{18}$$

Using the MIR procedure, from constraints (14) and (15) one obtains

$$2x_1 + 3x_2 + z_3 \geq 1 \quad \text{and} \tag{19}$$

$$4x_1 + 2x_2 + z_3 \geq 1. \tag{20}$$

Multiplying these two inequalities by 1/2, and adding, we obtain

$$3x_1 + \tfrac{5}{2}x_2 \geq 1 - z_3 \quad \text{and similarly,} \tag{21}$$

$$\tfrac{5}{2}x_2 + \tfrac{7}{2}x_3 \geq 1 - z_1 \quad \text{and} \tag{22}$$

$$3x_1 + \tfrac{7}{2}x_3 \geq 1 - z_2. \tag{23}$$

Since one of the z_i (and the corresponding x_i) must be zero, we obtain

$$3x_1 + \tfrac{5}{2}x_2 + \tfrac{7}{2}x_3 \geq 1. \tag{24}$$

It can be shown that this inequality, together with (14), (15), (16) and (18) define the projection to x-space of the convex hull of feasible mixed-integer points. The approach in the example can be generalized to cut off an extreme point with $K+1$ positive variables where at most K are allowed. In any case, as the example shows, a proper choice of multipliers is critical. How to find them efficiently is a matter for further research.

3.2. *Knapsack cuts*

A classical approach for solving pure integer programs consists of strengthening individual constraints through the use of valid inequalities for the $0-1$ knapsack problem. For background on the knapsack problem, see [11]. Here we discuss the strengthening of a single linear inequality, given upper bounds on the variables and an upper bound on the number of positive variables.

Consider the system

$$
\text{(P)}\quad
\begin{aligned}
&\sum_{j\in N} a_j x_j \geq \beta, && (25)\\
&|\text{supp}(x)| \leq K, && (26)\\
&0 \leq x_j \leq u_j, \quad \text{all } j, && (27)
\end{aligned}
$$

where $a_j > 0$ for $j \in N$. To simplify notation, we assume

$$u_j = 1, \quad \text{all } j, \tag{28}$$

which can be achieved by rescaling. We say that a subset C of indices is *critical* if it has the property that:

$$(*)\quad \text{For every } J \subset C \text{ with } |J| = K, \quad \sum_{j\in J} a_j < \beta.$$

If C is critical, we clearly must have that at most $K-1$ variables from C can be positive. In other words,

$$\sum_{j\in C} x_j \leq K-1 \tag{29}$$

is valid for system (P).

Example 2. Suppose $a = (8,7,6,4,12,12)^{\mathrm{T}}$, $\beta = 22$ and $K = 3$. Let $C = \{1,2,3,4\}$ which is critical. Consider the point $v = (1,1,1/2,1/4,1/4,0)^{\mathrm{T}}$. Then v satisfies (25) with equality, and in fact v satifies the surrogate constraint $\sum_j x_j \leq K$ with equality. However, v violates (29).

In the rest of the section, we discuss when (29) is facet defining for the convex hull of feasible solutions to (P), and how to lift in it in polynomial time to obtain a facet when (29) is not. To simplify notation, we assume $C = \{1,2,\ldots,|C|\}$ with $a_1 \geq a_2 \geq \cdots \geq a_{|C|}$; and that $a_{|C|+1} = \max\{a_j : j > |C|\}$.

Theorem 3. *Suppose system (P) is feasible. Inequality (29) is facet defining if the following conditions are satisfied:*

(1) *C is maximal subject to being critical and $|C| \geq K$,*

(2) $a_{|C|+1} + \sum_{j=2}^{K} a_j \geq \beta$,

(3) $a_{|C|+1} + \sum_{j=1}^{K-2} a_j + a_{|C|} \geq \beta$, *and*
(4) $K > |C| + 1$, *or* $a_{|C|+1} + \sum_{j=1}^{K-1} a_j > \beta$.

Proof. Similar to those used in the analysis of the knapsack problem, see [11]. □

It is not difficult to see that the above conditions are in fact necessary. This is clear for 1. If condition 2 does not hold then $x_1 = 1$ for all feasible points satisfying (29) with equality. We are most interested in condition 3. It guarantees that every index $j \in C$ is in the support of some feasible solution to (P) that satisfies (29) with equality. Finally, if condition 4 does not hold then the face is not proper. In the rest of this section, we describe an inequality that is facet defining when conditions 1 and 2 hold but 3 does not.

Define

$$Z = \{h \in C : a_{|C|+1} + \sum_{j=1}^{K-2} a_j + a_h < \beta\}.$$

Assuming (2) holds, we have that Z is of the form $t \leq j \leq |C|$ for some $t > K$. Clearly, for any x feasible for (P) that satisfies (29) with equality we have $x_j = 0$ for all $j \in Z$. Consequently,

$$\sum_{j=1}^{t-1} x_j \leq K - 1$$

is facet defining for the polyhedron

$$\text{conv}\{x : x \text{ feasible for (P) and } x_j = 0 \text{ for } j \in Z\}. \tag{30}$$

Thus, we can look for facet defining inequalities of the form,

$$\sum_{j=1}^{t-1} x_j + \sum_{j \in Z} \lambda_j x_j \leq K - 1, \tag{31}$$

with appropriate $\lambda_j \geq 0$.

One way to obtain such an inequality is to use the method of *sequential lifting*. Refer to [11] for details. Assuming that we have computed λ_j for $j \in W \subset Z$, then for a given $h \in Z - W$, λ_h is the largest value such that

$$\sum_{j \in C - Z} x_j + \sum_{j \in W} \lambda_j x_j + \lambda_h x_h \leq K - 1 \tag{32}$$

does not exclude any feasible x with $\text{supp}(x) \cap Z \subset W \cup h$. It can be shown that a simple dynamic programming recursion, similar to that used to solve the standard knapsack problem, can be used to compute λ_h.

Example 4. Suppose now that $a = \{8, 7, 6, 2, 1, 11, 10\}^{\mathrm{T}}$, $\beta = 22$ and $K = 3$. Consider $C = \{1, 2, 3, 4, 5\}$ which is critical. Here $Z = \{4, 5\}$. The corresponding facet defining inequality is

$$x_1 + x_2 + x_3 + 2x_4 + 2x_5 \leq 2. \tag{33}$$

Formally, we have the following:

Lemma 5. *For any lifting sequence, and for each $j \in Z$ λ_j is nonnegative, at most $K-1$ and integral, and can be computed in polynomial time.*

We note that this approach can also be used to obtain a facet when some of the a_i are nonpositive for $i \notin C$.

In any case, we do not have a good heuristic for identifying critical sets for which (29) is violated. This could be a good area for further work.

3.3. Intersection cuts

To describe the *intersection cuts* (see [1]), consider any valid formulation $Cx \leq d$ for QMIP, let x^* be an extreme point and let v^i, $1 \leq i \leq s$ be its neighboring vertices. If x^* has more than K positive variables, then every point of

$$\text{conv}(x^* \cup \{v^1, \cdots, v^s\}) \setminus \text{conv}(\{v^1, \cdots, v^s\}) \tag{34}$$

will likewise have more than K positive variables. As a result, any inequality that cuts off x^* but does not cut off any of the v^i is valid for QMIP. (We note that when dealing with an arbitrary mixed-0, 1 program, if we project the formulation to the space of the continuous variables, the preceding argument will not be valid. Strictly speaking, our cuts are not intersection cuts.) To implement this idea, one must enumerate the v^i (which is not difficult, modulo degeneracy) and choose an appropriate inequality. The inequality obtained above by means of the MIR procedure is an intersection cut.

The motivation for using intersection cuts is as follows. Typically, the quadratic programs arising in practice have nonnegative objective value. Essentially, we are finding a point in the polyhedron $\{x : Ax \leq b, 0 \leq x \leq u\}$ that is closest to the origin under some norm. As a result, the optimum will be attained by some point x^* in the nonnegative orthant with more than K positive coordinates, while all mixed-integer feasible points will be "farther away" from the origin than x^*. By adding an intersection cut we would hope not only to cut off x^* but to improve the lower bound on QMIP as well. This would certainly be the case if the normal to the cut has positive inner product with the gradient of the quadratic form at x^*.

A complication arising from QMIP is that the optimum for the quadratic program will in general not be an extreme point. Even if the optimum is in a low-dimensional face, this face may contain many extreme points, and we must in addition enumerate all of *their* neighbors. However, this may work to our benefit in that an entire face will be cut off.

4. Disjunctive cuts

The disjunctive procedure was developed by Balas (see [11]) as a general-purpose approach to generating valid inequalities for mixed-integer programs. Recently, this procedure has found great success in branch-and-cut algorithms; see [2,3,6].

For the purpose of this paper, the procedure may be described as follows. Consider a node of the branch-and-cut tree where t variables have been branched up and F is the set of variables not yet branched on, and let the current formulation consist of $Cx \leq d$ together with the surrogate constraint (12). For $h \in F$, let $P^0(h)$ denote the polyhedron defined by these constraints plus $x_h = 0$, and let $P^1(h)$ denote the polyhedron defined by $Cx \leq d$ and the strengthening (13) of the surrogate constraint. Suppose x^* is an optimal solution for the quadratic program at this node, and $h \in F$ is such that $0 < x^*_h < u_h$. If x^* is in the convex hull of feasible solutions to QMIP, it must be the case that it is a convex combination of a point in $P^0(h)$ and a point in $P^1(h)$. This condition can be described by system of linear inequalities; when this system is infeasible an application of the Farkas lemma yields an inequality valid for QMIP which is violated by x^*.

The disjunctive procedure can easily handle additional mixed-integer constraints for problem QMIP. Suppose, for example, that we have minimum transaction levels. This implies that in the up branch we will want to enforce (using the above notation) $x_h \geq \alpha$ for some $\alpha > 0$. This constraint can be added directly to the formulation of $P^1(h)$.

4.1. *Numerical difficulties*

The cuts produced by the disjunctive procedure can be extremely effective towards solving QMIP (although as we will see later our computational experience is not uniformly good). However, there are several noteworthy issues. The most important one concerns numerical instability. The disjunctive cuts cut-off very little: typically, the *scaled* violation (i.e. the violation after the inequality has been scaled so that the largest coefficient in absolute value is 1) is of the order of 10^{-4} or 10^{-3}, occasionally it is of the order of 10^{-2}, and almost never 10^{-1} or larger. This can make re-optimization slow and numerically unstable, but violations this small are to be expected because of the convex objective and the nature of the problem. Generally speaking, the cut coefficients can be badly behaved, and this is made worse by the fact that we are solving a quadratic program, i.e. the point we are trying to cut off may have several very small coordinates.

One early sign of numerical instability due to cutting is that the value of a problem may *decrease* by a small amount after adding cuts, say an improvement in the sixth or seventh significant digit. Since this is larger than the optimality tolerance we have set for the quadratic optimizer, it is meaningful. The way we currently handle it is to first resolve the quadratic program at the node from scratch (i.e., not using a warm start) and if the discrepancy in objective values remains, to then remove all newly added cuts. Fortunately, this seems to happen quite rarely.

A basic issue when implementing branch-and-cut concerns how to reoptimize when simultaneously adding several weakly violated inequalities. Obviously, some amount of experimentation is needed, but in the current implementation we use the following approach: (1) a cut is used if the scaled violation is at least 10^{-3} (small for linear integer programs [5]) (2) we reoptimize by adding all accepted cuts at once; if numerical problems are detected or if the reoptimization is taking too long, we handle the situation as described in the preceding paragraph. In terms of speed, it could be better to add one cut at a time and reoptimize, and whether this is the case remains to be tested.

Finally, there is a numerical difficulty inherent in using cuts whose coefficients are obtained by solving a linear program: small roundoff errors may yield cuts that "barely" cut off integer feasible points. This has been observed before [4,15]. As far as we know, no work has been done on how to avoid this problem. We have observed this difficulty when using very weakly violated cuts (scaled violation 10^{-4} or less).

4.2. Data management

Another issue concerns the density of the cuts. Recall that the initial formulation has very dense constraints. As a result, the disjunctive cuts themselves tend to be very dense. Moreover, the strengthening (13) of the surrogate inequality used in the formulation for $P^1(h)$, as compared to (12) must be used—without it, no violation will ever occur. Essentially, all the combinatorial information concerning branching is contained in the strengthening of the surrogate inequality. The disjunctive cuts are therefore only locally valid and avoiding this appears difficult. The combination of dense, locally valid cuts is quite tricky to handle. One alternative is to store cuts by storing the vector of multipliers obtained from the disjunctive procedure (i.e. the dual variables for the appropriate linear program) which is generally a less than fully dense vector, and use it to reconstruct a cut when needed. The difficulty here is that when we need to reconstruct a cut, we must have the formulation that gave rise to the cut, in addition to the multipliers. In essence, in order to reconstruct a cut we have to reconstruct the node of the branch-and-cut tree that gave rise to the cut (this node may not be in existence any longer) and it is not clear that this strategy will be space efficient. In the current implementation, the cuts generated at the root are removed only when they become very inactive, and cuts added in other nodes are removed when they become fairly inactive or "old" enough (i.e. a node does not inherit cuts generated L levels higher up the tree, for some small L unless they are still somewhat active). Further, there is a fixed limit on the total number of cuts (across the tree) and on the *average* number of cuts, also across the tree. This way storage becomes linear in the number of active nodes. Again, further experimentation is needed.

5. Upper bounds

Our upper bounds for QMIP are obtained through several heuristics. The primary thrust behind these heuristics is that they are designed to run fast, even at the cost of running them many times. At any given node, we will denote by x^* the current optimal solution to the quadratic program, and $J = \text{supp}(x^*)$.

Heuristic 1. In Heuristic 1 we run the complete branch-and-cut algorithm, restricted to columns of J. In general, we will expect $|J|$ not to be very large (say $|J| < 1000$) in particular if the overall branch-and-cut algorithm is working well. Consequently, this heuristic should run very quickly. In the current implementation, we run it at the root, and whenever $|J|$ is close enough to K. This heuristic has proved very useful for finding good upper bounds.

Heuristic 2. This heuristic is only used at the root, and only after running Heuristic 1. We simultaneously force all variables in J to zero and (subject to this condition) reoptimize to obtain a point y^*, add violated inequalities as above (which will be globally valid), and run the branch-and-cut algorithm restricted to columns in $J \cup \text{supp}(y^*)$. The purpose of this heuristic is to find a good upper bound when Heuristic 1 failed to do so, and otherwise to add strong inequalities to the formulation.

Forcing all variables in J simultaneously to zero may be somewhat blunt. For some problems, this resulted in very poor upper bounds, and in some cases the quadratic program was infeasible. In general, we may want to force to zero a proper subset of J, for example, just those variables that are strictly between bounds, or just those variables at their upper bounds.

Squeeze heuristic. For a vector z, Let $L(z)$ denote the set of indices j with $0 < z_j < u_j$. This heuristic seeks to find a new optimal solution y^* for the current node with $|L(y^*)| < |L(x^*)|$ and $\text{supp}(y^*) \subset J$. To this effect we run the branch-and-cut algorithm restricted to columns of J, with branching restricted to columns of $L(x^*)$, and with maximum depth limited to some small number (for example, two or three). The objective of the heuristic is two-fold: first, we seek to limit the number of variables strictly between bounds (so as to obtain a "better" branch next time) and second, to obtain a good upper bound for the overall problem. This heuristic is run at every node, and although it is not guaranteed to succeed, in nearly every problem it has found the optimal solution to the mixed-integer program.

6. Computational experiments

We implemented the branch-and-cut algorithm as indicated above, with one round of cutting at each node and using the "best node" strategy to choose the next node to branch on. At a given node, from all variables not yet branched on we selected the one furthest from its bounds as the next branching variable. The underlying quadratic optimizer was run with a feasibility tolerance of 10^{-10} for variables and of 10^{-7} for (scaled) constraints, and an optimality tolerance of 5×10^{-9}. A variable was declared "positive" if its value exceeded 10^{-8}.

Table 1
Branch-and-cut on compact problems

Problem	Rank	Inequalities	Columns	α	Nodes	Cuts	Time (s)
1	7	15	3342	0.01	15	28	3.22
2	7	15	3897	0.01	181	0	23.62
3	7	15	3897	0.00	176	176	87.45
4	7	15	3897	0.00	25	38	13.65
5	7	15	3897	0.01	22	37	7.13
6	7	15	3897	0.00	86	179	53.25
7	7	15	3342	0.00	13	0	1.95
8	7	15	3342	0.04	623	0	665.37
9	7	15	3342	0.00	383	410	694.35
10	7	15	3342	0.00	623	0	756.33

Several real-life data sets were made available to us. As stated before, all of these problems where of the low-rank quadratic type. Each data set primarily consists of (1) a constraint matrix A, (2) a quadratic matrix M, (3) a right-hand side vector, and (4) a vector of upper bounds u_j for the variables. In practice, it is common to vary the right-hand sides of the inequalities and the u_j, and in particular, the parameter K so as to obtain different problems. For different values of K one can obtain significantly different problems. In addition, some of the problems were run with the condition that if a variable is positive, it must be at least a certain input value (minimum transaction level). Table 1 presents data for these runs. The column headed "rank" refers to the rank of the objective, "ineqs" stands for the number of inequalities in the original formulation, "cols" the number of columns, "nodes" is the number of branch-and-cut nodes needed to solve the problem, and "cuts" is the number of disjunctive inequalities separated by the algorithm, counted when generated (i.e., not counted when reused at descendant nodes). The column headed "α" indicates the minimum transaction level. All times are on a SUN Sparc 10/51 computer.

Some of the problems were solved by Heuristic 1, a few Heuristic 2, and the rest by branch-and-cut on all the columns. The value of K was generally 20–25, while the number of positive variables at the root was generally about 40, and higher if the surrogate constraint (11) was not used. As can be seen, the amount of cutting is somewhat erratic. Furthermore, the cuts were frequently concentrated at a few of the nodes, and were essentially used to fathom the nodes or to prove optimality at a certain node in conjuction with the squeeze heuristic. While this behavior can be partially explained by the fact that we are optimizing a quadratic objective, it remains to be seen whether a different approach to cutting or branching can be more effective. For those problems where cutting took place, the cutting did make a difference in terms of reducing the size of the enumeration tree.

One aspect of the problems that made them somewhat difficult is that the gap between the value at the root, and the value of the mixed integer program was very small, of the order of 10^{-7} or less, which for practical (and numerical) purposes is zero. The problems are fairly combinatorial in that at the root the "wrong" variables are set at their

Table 2
Branch-and-cut on non-compact problems

Problem	Rank	Inequalities	Columns	α	Nodes	Time (s)
9	2666	15	3342	0.01	232	111.33
10	2666	15	3342	0.04	522	364.90
11	2666	15	3342	0.01	242	13.82
12	3031	15	3897	0.01	32	36.40
13	3031	15	3897	0.00	448	170.75

upper bounds, and the task for the algorithm is to find the "right" set of variables to attain upper bounds, as well handling the nonlinearity of the objective, without changing the original objective value.

To further test our algorithm, we generated high rank objective problems by adding to the quadratic a large, artificially constructed, diagonal matrix. In this form we obtained problems resembling those of the form (5). This diagonal matrix was constructed by drawing numbers at random from (0, 1), rounding them to zero whenever smaller than a certain threshold value, and then scaling them up so that the contribution to the objective value arising from the diagonal term is of the same order of magnitude as the original value.

Not surprisingly, these problems were more difficult for our code, for the simple reason that *no* cutting ever took place, other than that resulting from strengthening the surrogate inequality (11) at each up branch. As a result, we ran the algorithm with the disjunctive cuts turned off.

The values of K that we used for these problems were, respectively, 25, 25, 43, 25 and 40 and the number of positive variables at the root ranged from 40 to 100, many of them being set to very small values, of the order of 10^{-7} or less.

In problem 11, the number of positive variables at the root was 43, i.e. equal to K, and thus the side-constraints in this problem were to force a minimum transaction level of 0.01 on all positive variables, without increasing the positive count. In Table 3 we analyze several problems which have the same constraints as problem 11 in Table 2, but for different values of K. The value at the root was 7.60805e+01 and the solution had 43 positive variables, both for all values of K reported in the table. Forcing all these variables simultaneously to zero resulted in an infeasible quadratic program; thus for these problems Heuristic 2 had no effect. Further, for all of these problems the optimal solution was found by Heuristic 1, and was proved so when running branch-and-cut on all variables. In the table, HTIME and HNODES are the time and nodes used by Heuristic 1, respectively, VALUE is the value of the mixed-integer program, and TIME and NODES are the overall time and nodes used by the algorithm.

From a purely practical point of view our heuristics may be good enough. However, as the table indicates, the heuristics may be unable to solve problems of this type to provable optimality in reasonable running time. In fact, it is a simple exercise to generate an artificial problem of the form (5) where the (unique) optimal solution to the

Table 3
Problem 11 with different values of K

K	VALUE	HTIME	HNODES	TIME	NODES
38	7.60875e+01	10.63	122	107.63	208
37	7.60884e+01	15.18	156	168.12	253
36	7.60902e+01	13.90	146	176.53	250
35	7.60962e+01	17.83	181	235.95	301
34	7.61131e+01	34.55	347	553.20	615
33	7.61347e+01	94.27	920	1830.62	1810
32	7.61645e+01	274.05	2580	5182.90	4909
31	7.62055e+01	862.33	8183	18759.83	16469
30	7.62542e+01	2460.00	21754	51214.57	44381

continuous problem has all variables positive. But even if only a few hundred variables are positive at the root, and if the objective is fairly flat, the mixed-integer program may be impossible to solve to optimality by a branch-and-cut algorithm as described in this paper. The difficult part is proving optimality, since quite likely no cutting will take place. Finding good upper bounds may not be so difficult if an appropriate heuristic is used. It is not clear to us whether the problems we generated are realistic or not; and the design of appropriate heuristics and appropriate branching schemes, such as branching by hyperplanes, should depend heavily on what the data is really like.

6.1. Parallel implementation

Here we describe preliminary computational experience with an implementation of our algorithm using a symmetric-multiprocessing, shared memory computer. A computer of this type contains a number of identical CPU processors connected to a common system memory. The different processors are able to simultaneously execute tasks. Furthermore, the processors can simultaneously "read" the system memory, but simultaneous "writes" (changes) are not allowed.

Parallelism is achieved through the use of "threads". Broadly speaking, a program creates, or launches, threads to execute in parallel specific computational tasks. Once a thread is created, it will run its appointed task in parallel with the main program. Once the task is completed, the thread enters a special "signaled" state. The main program can check to see if a thread has terminated by testing whether this signaled state holds. It is important to notice that the number of threads a program creates and the number of available processors are independent quantities (indeed, it is common to have multithreaded programs running on one processor—the processor alternates cyclically between the different threads). Further, the details of scheduling threads to processors, and of handling low-level details (such as detecting signals) are completely left to the operating system running the computer. All the algorithm designer has to do is choose *how* to parallelize an algorithm. In fact, the differences between the parallel and non-parallel versions of a program can be fairly small and localized. These facts make multithreaded programs easily portable from one computer to another, possibly with a different number of processors. Multi-processor computers of this kind are now becoming widely available.

In terms of our problem, we solve several nodes in parallel (this is similar to the approach used in [6, 8, 14] with very different parallel architectures). In the steady state of the algorithm, for some integer $M \geq 1$ chosen by the user, M nodes will be simultaneously being solved. Then we

- Wait until one of the current node-solving threads terminates.
- Post-process the corresponding node.
- Select a new node from among those remaining active.
- Create a new thread to solve that node.

Usually, but not always, the best strategy is to choose M equal to the number of CPU processors. More precisely, choosing M smaller than the number of processors will result in processors consistently experiencing idle periods. Choosing M much larger than the number of processors may result in much larger branch-and-cut trees, because we solve nodes that might otherwise have been fathomed. Choosing M about equal to the number of processors has yielded best results in our experiments.

For our implementation we used an ALR computer with four 100 MHz Pentium processors. In our testing, the branch-and-cut code with the multithreaded implementation was sped up by approximately a factor of three—sometimes more (but never close to a factor of four) and sometimes less (as little as a factor of 2.5). This piece of information is somewhat misleading. Even when running the program in single-threaded mode, the operating system attempts to parallelize low-level tasks, with unpredictable results.

Instead of focusing on speed-ups, we have chosen to analyze a different aspect of the implementation. It should be noted that the Pentium processor is not especially good in terms of floating point performance. For example, it is significantly slower than the processor in the Sparc 10/51 computer used in the computational experiments described in previous sections. An implementation of branch-and-cut entails a large amount of non-floating point computation but the floating point component dominates, and without parallelism the speed-up of the implementation in the Sparc 10/51 over that in the (one processor) Pentium was close to 40%. Table 4 compares the running times for various problems using a one-thread implementation in the one-processor Sparc 10 (column headed "$1 \times$ SS10/51") to those using a four-thread implementation in the $4 \times$ Pentium machine. Times are in seconds.

As the problems become more difficult, the $4 \times$ Pentium implementation becomes approximately twice as fast as the Sparc 10/51 code, despite the processor handicap. We used the fifth problem for a more stringent test. The one-thread implementation was run on a one-processor Silicon Graphics Challenge machine, with a 200 MHz R4400 chip with a 4 Mb cache; a very competent (if somewhat expensive) platform for floating point computation. Here the running time was 1304.20—slower than that on the $4 \times$ Pentium. In our view, this highlights the power and potential of parallelism in branch-and-cut algorithms.

Table 4
Comparison of sequential and parallel codes

$1 \times$ SS10/51	$4 \times$ P5
214.87	133.23
224.55	140.13
279.55	171.38
653.00	370.69
2294.25	1297.17
6307.15	3530.12

Appendix A. The quadratic optimizer

The quadratic programming problems arising from QMIP, whether of the compact form (4) or the form (5) are far from "general", whatever this may mean. The practicioners who communicated problem QMIP to us had implemented a well-known dual algorithm for quadratic programming, without success: the algorithm would usually fail to achieve feasibility. This could be due to either a numerically unstable implementation or the bad conditioning of the constraints.

In any case, based on this experience, and considering the special structure of the problems, we implemented our own algorithm, which was used in the experiments described above. Here we will briefly describe this algorithm. A fuller description will appear in a future paper.

This algorithm is primal feasible; it consists of a standard Phase I combined with a heuristic for finding a good solution, plus a sequence of feasible descent steps, to achieve optimality within tolerance. The descent steps are of three types, all standard and well-known in the nonlinear programming literature. The first type consists of a Newton step taken in the null space of the active constraints. The Newton calculation yields a descent direction; a step is taken in this direction until we reach the global optimizer in the current null space or another constraint becomes active, whichever comes first. Typically, we will take a sequence of steps of this kind, each time in a space of dimension one smaller than before.

The other two types of steps that the algorithm takes are pure gradient descent steps. The first of these is a "steepest descent" step (Zoutendijk's method) which is determined by solving a linear program. The second step proceeds by replacing the objective by its gradient (i.e. linearizing the objective) and solving the resulting *linear* program to obtain a descent direction (we note that this method also yields a lower bound to the value of the quadratic program). Refer to [9, 12] for details. Typically, the algorithm will execute major iterations consisting of one of each of the three types of steps just described.

As stated before, the algorithm can handle warm starts with infeasibilities by using a quadratic penalty for each infeasibility. The multipliers associated with these quadratic terms are adjusted after each descent step. The multiplier calculation is done in two steps. Briefly, the multiplier associated with a specific infeasibility is chosen proportional to the value of the infeasibility. Next, the multipliers are rescaled so that the overall penalty is proportional to the value of the original quadratic at the current point (for example, we want the penalty to be at least one-half, but not more than twice the value of the unpenalized objective). When re-optimizing from an infeasible starting point, we use this approach for a small number of major iterations (say, no more than four major iterations). Typically by this point the remaining infeasibilities are small. We then switch to a Phase I linear program, beginning at the current iterate. We note that this step is necessary to detect infeasibility of the quadratic program.

We have done some limited testing to compare the performance of this code to other existing codes. The purpose of the testing is to ensure that our algorithm is not grossly inefficient or numerically unstable. We compared the code to recent versions of MINOS, a well-known general-purpose primal nonlinear optimizer, LoQo [16], a primal-dual linear or quadratic optimizer. Both these codes can explicitly take advantage of the special structure of the problems we are interested in (low rank quadratic and special constraint structure). In our experiments our code was significantly faster than both MINOS and LoQo. Undoubtedly both codes can be made to run faster on these problems. However, it appears that our code is at least competitive with both MINOS and LoQo; of course this conclusion is only valid with regards to the problems described above. We have also compared our code to an experimental QP-solver contained in the Cplex package [7]. This is a general-purpose code, but a very well designed one. Our algorithm was typically 20% to twice as fast as this code.

We should point out that from an experimental point of view, being familiar with the underlying optimizer was greatly useful when implementing the overall branch-and-cut algorithm.

Acknowledgements

The author wishes to thank T. Carpenter, S. Ceria, D. Goldfarb, E. Lee and R. Bixby for fruitful discussions.

References

[1] E. Balas, Intersection cuts – a new type of cutting planes for integer programming, *Operations Research* 19 (1971) 19–39.

[2] E. Balas, S. Ceria and G. Cornuéjols, A lift-and-project cutting plane algorithm for mixed 0–1 programs, *Mathematical Programming* 58 (1993) 295–324.

[3] E. Balas, S. Ceria and G. Cornuéjols, Mixed 0–1 programming by lift-and-project in a branch-and-cut framework, *Management Science* (to appear).

[4] W. Cook, personal communication.

[5] R. Bixby, personal communication.

[6] R. Bixby, W.J. Cook, A. Cox and E. Lee, Parallel mixed-integer programming, manuscript (1994).

[7] Cplex Optimization, Inc.

[8] J. Eckstein, Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5, *SIAM Journal on Optimization* 4 (1994) 794–814.

[9] R. Fletcher, *Practical Methods of Optimization*, Vol. 2 (Wiley, 1981).
[10] H. Konno and K. Suzuki, A fast algorithm for solving large scale mean-variance models by compact factorization of covariance matrices, Report IHSS 91-32, Institute of Human and Social Sciences, Tokyo Institute of Technology (1991).
[11] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization* (Wiley, New York, 1988).
[12] D.G. Luenberger, *Linear and Nonlinear Programming* (Addison Wesley, 1984).
[13] A.F. Perold, Large-scale portfolio optimization, *Management Science* 30 (1984) 1143–1160.
[14] M. Savelsbergh, personal communication (1995).
[15] L.A. Wolsey, personal communication.
[16] R.J. Vanderbei and T.J. Carpenter, Symmetric indefinite systems for interior point methods, *Mathematical Programming* 58 (1993) 1–32.