Semidefinite Diagonal Directions Algorithm for Detecting Necessary Linear Matrix Inequality Constraints

Jackie Van Ryzin

St. Norbert College, De Pere, WI 54115.

July 21, 2006

Abstract

In operations research, it is often useful to pre-solve a linear or semidefinite program to reduce the size of the original program, thus saving on computation time. This can be done by determining redundant or necessary constraints in the program. This is especially valuable to corporations that frequently require solving large optimization problems.

The most efficient Monte Carlo method for detecting necessary linear matrix inequality constraints in semidefinite programming is the Semidefinite Coordinate Directions (SCD) algorithm. We develop the Semidefinite Diagonal Directions (SDD) algorithm by using diagonal directions instead of coordinate directions as in SCD. We analyze the benefits and costs of this change in comparison with the SCD method.

Acknowledgement

This research was made possible by the National Science Foundation funding for the Research Experience for Undergraduate Students (REU Program). Special thanks to Dr. Shafiu Jibrin for his motivation, support, and ideas as he supervised this research.

1 Introduction

We study linear matrix inequality constraints that arise in the field of semidefinite programming. This is an area of operations research that is still under active research and investigation as it is a relatively recent development in the field of optimization. A *semidefinite program (SDP)* is an optimization problem where one optimizes a linear function subject to a set of constraints called *linear matrix inequalities (LMIs)*. An SDP can be expressed in the following form:

$$\begin{aligned} &min \quad c^T x \\ &s.t. \quad A^{(j)}(x) := A_0^{(j)} + \sum_{i=1}^{n} x_i A_i^{(j)} \succeq 0, \quad 1 \leq j \leq q \end{aligned}$$

where all the $A_i^{(j)}$ are $m_j \times m_j$ symmetric matrices and $x, c \in \mathbb{R}^n$. We assume that $m_1 \leq m_2 \leq \ldots \leq m_q$.

A matrix is *positive semidefinite* ($\succeq$) if all its eigenvalues are non-negative. Likewise, $\succ$ refers to a *positive definite* matrix which means that all of its eigenvalues are positive. A matrix is positive semidefinite (positive definite) if and only each of its minors is non-negative (positive) [12]. A semidefinite program is a convex optimization problem [12], [13].

Semidefinite programming is a generalization of linear programming, quadratic programming and other useful nonlinear convex optimization problems. There are many connections between the theory of semidefinite programming and that of linear programming [13]. Problems in engineering, statistics, geometry, combinatorics and many other areas can be naturally converted to semidefinite programs [13], [9]. Semidefinite programming can also be used to approximate the solution of certain NP-hard combinatorial optimization problems [13].

Semidefinite Program Example: $q = 5$ LMI constraints with $n = 2$ variables:

$$\min\ 2x_1 - 3x_2$$

s.t.

$$A^{(1)}(x) := \begin{bmatrix} 5 & -1 \\ -1 & 2 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0$$

$$A^{(2)}(x) := \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0$$

$$A^{(3)}(x) := \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} + x_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \succeq 0$$

$$A^{(4)}(x) := \begin{bmatrix} 3.85 \end{bmatrix} + x_1 \begin{bmatrix} -0.4 \end{bmatrix} + x_2 \begin{bmatrix} 1 \end{bmatrix} \geq 0$$

$$A^{(5)}(x) := \begin{bmatrix} 2.75 \end{bmatrix} + x_1 \begin{bmatrix} 0.8 \end{bmatrix} + x_2 \begin{bmatrix} 1 \end{bmatrix} \geq 0.$$

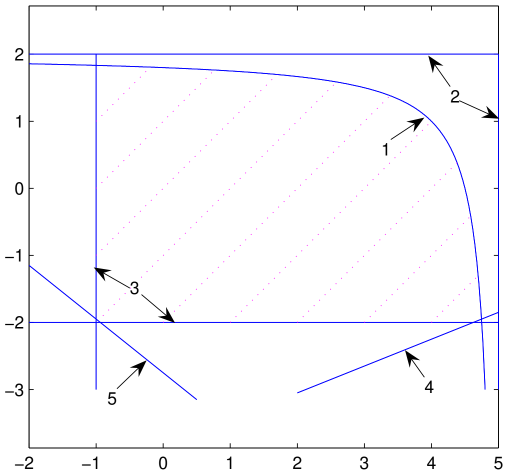

Figure 1: Example SDP

Because SDP problems can be very large, especially in practice within industry, it is valuable to pre-solve an SDP to reduce its size. This can help to save on computation time, thus saving money and resources for companies. As [1] describes, it is also valuable to reduce the size of an SDP because common SDP solvers such as SDPSOL, SDPpack, and SeDuMi are restricted to problems of moderate size. Algorithms exist to solve SDP problems, but they are restricted to particular classes of semidefinite programs; a few are given in [11]. To reduce the size of a semidefinite program, one can eliminate the redundant or identify the necessary LMI constraints. An LMI constraint is called *redundant* if its removal would not change the feasible region of the SDP. Otherwise, it is called *necessary.* In our example, constraints 1, 3, 4, and 5 are necessary and constraint 2 is redundant.

The feasible region of the SDP is given by

$$\mathcal{R} = \{x \in \mathbb{R}^n \mid A^{(j)}(x) \succeq 0, \quad 1 \leq j \leq q\}$$

In Figure 1, the feasible region is shaded with dotted lines. A point x_0 is in the interior of the feasible region if and only if $A^{(j)}(x_0) \succ 0, \quad 1 \leq j \leq q$. Likewise, the point x_0 is on the boundary of the feasible region if and only if $A^{(j)}(x_0) \succeq 0$ for some j and $A^{(j)}(x_0) \not\succ 0$. We assume that $\mathcal{R}$ is bounded and has a non-empty interior.

Eliminating all redundant constraints will give us the minimal representation of the SDP as described in [3]. To determine if the k^{th} constraint, $A^{(k)}(x) \succeq 0$, is redundant, we solve the associated *semidefinite redundancy problem.* Let $\lambda_{max}(B)$ and $\lambda_{min}(B)$ denote the maximum and minimum eigenvalues of a symmetric matrix B.

Theorem 1.1 *[7] $A^{(k)}(x) \succeq 0$ is redundant (not necessary) if and only if the semidefinite program*

SDP_k:

$min \quad \lambda_{min}(A^{(k)}(x))$

$x \in \mathcal{R}_k$

has an optimal solution, say x^*, *satisfying*

$$min \;\; \lambda_{min}(A^{(k)}(x^*)) \geq 0.$$

where $\mathcal{R}_k = \{x \mid A^{(j)}(x) \succeq 0, \; \jmath = 1, 2, \ldots, k-1, k+1, \ldots, q\}$ $\Box$

The problem SDP_k is *NP-complete*, which implies that it can be very difficult to solve. However, in the special case where all the LMI constraints are linear, SDP_k is a linear program and can be solved using standard LP techniques. A survey of methods for determining redundant or necessary linear constraints in linear programming is given in [2], [6] and [8]. Because of the difficulty of determining redundant LMI constraints, we focus on the detection of necessary constraints from the system of LMI's of the SDP.

There are three common probabilistic hit-and-run methods for detecting necessary LMI constraints, related to those used to detect necessary linear constraints in [5]. As described in [7], they are based on passing randomly generated lines through the feasible region. These methods are the Semidefinite Stand-and-Hit (SSH), the Semidefinite Hypersphere Direction (SHD), and the Semidefinite Coordinate Direction (SCD) methods. Of the three methods, SCD is considered to be the most efficient.

The SCD algorithm begins by selecting a *standing point* x_0 in the interior of the feasible region $\mathcal{R}$. From this standing point, we generate a direction vector s_0 parallel to one of the coordinate directions. The standing point together with the generated direction s_0 define a line $\{x_0 + \sigma s_0 \mid \sigma \in \mathcal{R}\}$. The line meets the boundary of $\mathcal{R}$ at the points $(x_0 + \sigma^+ s_0)$ and $(x_0 - \sigma^- s_0)$, where σ^+ and σ^- denote the distances from x_0 to the boundary of $\mathcal{R}$ in either direction $\pm s_0$. These two points are called *hit points* and identify at least one LMI as necessary. A random point x_1 is then chosen uniformly from the line segment joining the two hit points and used as the standing point for the next iteration. In the next iteration a new random coordinate direction vector s_1 is chosen and the process is repeated until a stopping condition is reached. Stopping conditions for hit-and-run methods are discussed in [4].

The following theorem gives the calculations for σ^+ and σ^- along $s_k = e_r$.

Theorem 1.2 *[7] The distances σ^+ and σ^- are given by*

$$\begin{aligned}\sigma^+ &= \min\{1/\lambda_{max}(B_j(s_k, x_k)) \mid 1 \leq j \leq q\} \\ \sigma^- &= \max\{-1/\lambda_{min}(B_j(s_k, x_k)) \mid 1 \leq j \leq q\}\end{aligned}$$

where

$$B_j(s_k, x_k) = -A^{(j)}(x_k)^{-\frac{1}{2}} A_r^{(j)} A^{(j)}(x_k)^{-\frac{1}{2}} \ \textit{for} \ 1 \leq j \leq q. \ \square$$

Note that if $x_{k+1} = x_k + \sigma s_k$, then $(x_{k+1})_r = (x_k)_r + \sigma$ and $(x_{k+1})_i = (x_k)_i$ if $i \neq r$. From this we also have,

$$\begin{aligned}A^{(j)}(x_{k+1}) &= A_0^{(j)} + \sum_{i=1}^{n}(x_k + \sigma s_k)_i A_i^{(j)} & (1) \\ &= A_0^{(j)} + \sum_{i=1}^{n}(x_k)_i A_i^{(j)} + \sigma \sum_{i=1}^{n}(s_k)_i A_i^{(j)} & (2) \\ &= A^{(j)}(x_k) + \sigma A_r^{(j)} & (3)\end{aligned}$$

To compute the matrix $A^{(j)}(x_k)$ requires m_j^2 additions and m_j^2 multiplications. Also, computing each $A^{(j)}(x_k)^{-\frac{1}{2}}$ is of order $\bigcirc(m_j^3)$. We see that excluding the cost of calculating $A^{(j)}(x_k)^{-\frac{1}{2}}$, computing $B_j(s_k, x_k)$ requires $2m_j^3$ multiplications. To compute the eigenvalues of $B_j(s_k, x_k)$ is also of order $\bigcirc(m_j^3)$. Hence, including the costs of all of these computations, SCD takes $\bigcirc(qm_q^3)$ computations per iteration.

We modify the SCD algorithm in the choice of the direction vectors to create the semidefinite diagonal directions (SDD) algorithm. Instead of choosing a random direction parallel to a coordinate axis, we choose a random direction that is parallel to a diagonal. Our numerical experiments show that SDD is more efficient than SCD in finding necessary LMI constraints for some problems. The algorithm is explained in detail in the following section.

2 Semidefinite Diagonal Direction (SDD) Method

Here we introduce the semidefinite diagonal direction (SDD) method. It is the same as SCD except in the choice of a search direction. In the SDD method we only choose directions parallel to the diagonals, as opposed to the coordinate directions.

We begin as with the SCD method by setting $k = 0$ and choosing a starting point x_0 from the interior of the feasible region $\mathcal{R}$. If an interior point is unknown, it is possible to use a method such as the one described in [10] to determine an interior point. One can also use the following result.

Theorem 2.1 *Consider the following SDP:*

$$\begin{array}{rl} min & t \\ s.t. & A^{(j)}(x) + tI \succeq 0, \quad 1 \le j \le q \\ & t \ge -1 \end{array}$$

Let (x^*, t^*) *be an optimal solution to this SDP. Then*

(i) If $t^* < 0,\ A^{(j)}(x) \succ 0$ *for* $1 \le j \le q$.

(ii) If $t^* \ge 0,\ \mathcal{R}$ *has no interior point.*

Proof:

(i) Assume $t^* < 0$. Then

$$\begin{aligned} & A^{(j)}(x^*) + t^* I \succeq 0 \quad for\ 1 \le j \le q \\ \Rightarrow\ & \lambda_{min}(A^{(j)}(x^*) + t^* I) \ge 0 \quad for\ 1 \le j \le q \\ \Rightarrow\ & \lambda_{min}(A^{(j)}(x^*)) + t^* \ge 0 \quad for\ 1 \le j \le q \\ \Rightarrow\ & \lambda_{min}(A^{(j)}(x^*)) > 0 \quad for\ 1 \le j \le q \\ \Rightarrow\ & A^{(j)}(x^*) \succ 0 \quad for\ 1 \le j \le q \end{aligned}$$

(ii) Suppose $\mathcal{R}$ has an interior point y. Then

$$A^{(j)}(y) \succ 0 \quad for\ 1 \le j \le q \Rightarrow \exists \epsilon_j < 0\ s.t.\ A^{(j)}(y) + \epsilon_j I \succ 0 \quad for\ 1 \le j \le q.$$

Let $\epsilon = max\{\epsilon_j \mid 1 \le j \le q\}$. Then $t^* \le \epsilon < 0$. □

Once we have an interior point, we generate a random diagonal direction vector s_k by choosing the first $n-1$ entries each randomly from $\{-1, 1\}$ and setting n^{th} entry to 1. This gives us a random diagonal direction vector in the upper half of the hyperspace $\mathbb{R}^n$. Then we look for hit points in both directions ($\pm s_k$) as determined by Theorem 1.2, where in this case

$$B_j(s_k, x_k) = -A^{(j)}(x_k)^{-\frac{1}{2}} C_j(s_k) A^{(j)}(x_k)^{-\frac{1}{2}}\ for\ 1 \le j \le q \tag{4}$$

where

$$C_j(s_k) = \sum_{i=1}^{n} (s_k)_i A_i^{(j)}. \tag{5}$$

Following this format, a possibility for the first s_k in our example from the previous section could be $s_k = [1\ \ 1]^T$ as shown in Figure 2. This direction would be parallel to $y = x$ and appear in the first quadrant. Suppose our $x_0 = [0\ \ 0]^T$, a point

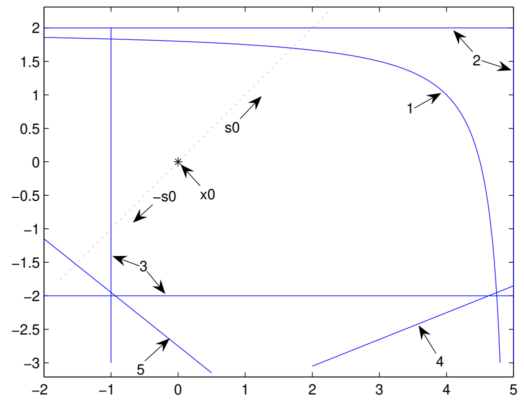

Figure 2: First Iteration of SDD

in the interior of the feasible region. Then we would have the picture in Figure 2 for our first iteration of SDD.

From this picture, we can see that SDD detected constraints 1 and 3 as necessary LMI's. From here, we will generate a new diagonal direction s_{k+1} in the upper half of the hyperspace $\mathbb{R}^n$, different from the previous s_k. To generate s_{k+1}, we first generate a random number between 1 and $n-1$. This will be the number of entries in the s_{k+1} vector that will differ from those of the s_k vector; they will differ only by their sign. This guarantees that s_{k+1} will be different than s_k by at least one entry while remaining in the upper half of $\mathbb{R}^n$. We randomly generate a list of the actual entries whose signs we will change. We denote the list of entries to change by (G); the list G will later be used to reduce computations in the calculation of $B_j(s_{k+1}, x_{k+1})$.

In the second iteration of SDD for our example, $s_1 = [-1 \quad 1]^T$. As shown in Figure 2, we have detected constraint 4 as another necessary LMI. We can see from the picture that our x_1 was chosen from the line segment connecting the two hit

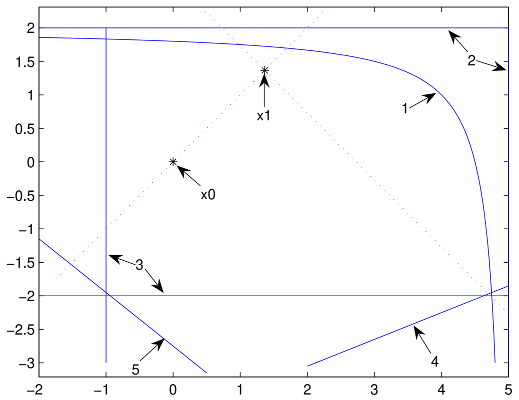

Figure 3: Second Iteration of SDD

points from our first iteration as in the SCD algorithm, and that our s_{k+1} vector is a different direction than s_k so that we could detect constraints different from those detected in the first iteration.

We find the distance to the boundary as in the SCD algorithm. We use the symmetric matrix $B_j(s_k, x_k)$ as defined in (4) and Theorem 1.2 to calculate the distances. We have the following result:

Lemma 2.1 *[7] If the line $\{x_k + \sigma s_k \mid \sigma \in \mathcal{R}\}$ intersects the boundary of $A^{(j)}(x) \succeq 0$ at the point $x_k + \sigma^+ s_k$ or $x_k - \sigma^- s_k$, respectively, then*

$$\begin{aligned} \sigma^+ &= min\{\sigma_j^+ \mid 1 \leq j \leq q\} \\ \sigma^- &= min\{\sigma_j^- \mid 1 \leq j \leq q\} \end{aligned}$$

where

$$\sigma_j^+ = 1/\lambda_{max}^+(B_j(s_k, x_k)) \tag{6}$$

$$\sigma_j^- = -1/\lambda_{min}^-(B_j(s_k, x_k)). \ \square \tag{7}$$

The following is the algorithm for our SDD method based on the above calculations and definitions.

SDD Algorithm

Initialization:

Let J be the set of detected necessary constraints and initialize $J = \emptyset$.

Choose x_0 in the interior of the feasible region.

Generate a random diagonal direction vector s_0 in the upper half of $\mathbb{R}^n$ with each entry chosen from $\{-1, 1\}$.

Set $k \longleftarrow 0$.

Repeat:

Hitting Step:

Calculate $A^{(j)}(x_k) = A_0^{(j)} + \sum_{i=1}^{n} (x_k)_i A_i^{(j)}$ for $1 \leq j \leq q$.

Calculate $A^{(j)}(x_k)^{-\frac{1}{2}}$ for $1 \leq j \leq q$.

Calculate $C_j(s_k) = \sum_{i=1}^{n} (s_k)_i A_i^{(j)}$ for $1 \leq j \leq q$.

Calculate $B_j(s_k, x_k) = -A^{(j)}(x_k)^{-\frac{1}{2}} C_j(s_k) A^{(j)}(x_k)^{-\frac{1}{2}}$ for $1 \leq j \leq q$.

Find σ_j^+, σ_j^- for $1 \leq j \leq q$ as described in (6) and (7).

Determine $\sigma^+ = min\{\sigma_j^+ \mid 1 \leq j \leq q\}$ and $\sigma^- = min\{\sigma_j^- \mid 1 \leq j \leq q\}$

For $1 \leq \ell \leq q$, if $\sigma_\ell^+ = \sigma^+$ or $\sigma_\ell^- = \sigma^-$ and $\ell \notin J$, then set $J = J \cup \{\ell\}$.

Search Direction:

Generate a random diagonal direction vector s_{k+1} in the upper half of $\mathbb{R}^n$ different from s_k with each entry chosen from $\{-1, 1\}$.

Update:

Choose $u \in U(0, 1)$.

Calculate $\alpha = \sigma^+ + u(\sigma^- - \sigma^+)$.

Set $x_k \longleftarrow x_k + \alpha s_k$.

Set $k \longleftarrow k + 1$.

Until a stopping condition is reached.

At the end of the algorithm, all of the LMI constraints listed in J are declared necessary.

3 Reducing the Time Complexity of SDD

In this section, we discuss the time complexity of SDD for each iteration. We also show how to use update formulas to reduce the time complexity. Many people would not consider additions to be time consuming or relevant to the cost per iteration because they are generally much faster in implementation than multiplications. However, with modern computers, it seems that this difference in computation time between multiplications and additions is shrinking. For this reason, we analyze multiplications as well as additions for each iteration of SDD.

In an iteration of SDD, the main cost per iteration results from computing $A^{(j)}(x_k)$, $A^{(j)}(x_k)^{-\frac{1}{2}}$, $B_j(s_k, x_k)$ and the eigenvalues of $B_j(s_k, x_k)$ for each constraint j. If we simply make these calculations as described in the algorithm, there are nm_j^2 multiplications and nm_j^2 additions in the calculation of the $A^{(j)}(x_k)$. In the calculation of $B_j(s_k, x_k)$, we find that there are $nm_j^2 + 2m_j^3$ multiplications and $(n-1)m_j^2$ additions. The calculation of $A^{(j)}(x_k)^{-\frac{1}{2}}$ and finding the eigenvalues of $B_j(s_k, x_k)$ are each of order $\bigcirc(m_j^3)$. Including all of these costs, the total computations per iteration of SDD is $\bigcirc(qnm_q^2 + qm_q^3)$. So we have:

Theorem 3.1 *Each iteration of SDD is $\bigcirc(qnm_q^2 + qm_q^3)$ time.*

Updates

The following results show how to compute the matrices $A^{(j)}(x_{k+1})$ and $B_j(s_{k+1}, x_{k+1})$ by updating $A^{(j)}(x_k)$ and $C_j(s_k)$. The updates provide a more efficient way to compute $A^{(j)}(x_{k+1})$ and $B_j(s_{k+1}, x_{k+1})$ than direct computation.

Theorem 3.2

$$\begin{aligned} A^{(j)}(x_{k+1}) &= A^{(j)}(x_k) + \alpha C_j(s_k) \\ C_j(s_{k+1}) &= C_j(s_k) - 2\Big(\sum_{i\in G, i\in P_k} A_i^{(j)} - \sum_{i\in G, i\in N_k} A_i^{(j)}\Big) \end{aligned}$$

where P_k is the set of entries of s_k that are positive and N_k is the set of those that are negative.

Proof: Note that $x_{k+1} = x_k + \alpha s_k$. Then

$$\begin{aligned} A^{(j)}(x_{k+1}) &= A_0^{(j)} + \sum_{i=1}^{n} (x_{k+1})_i A_i^{(j)} \\ &= A_0^{(j)} + \sum_{i=1}^{n} (x_k + \alpha s_k)_i A_i^{(j)} \\ &= A_0^{(j)} + \sum_{i=1}^{n} (x_k)_i A_i^{(j)} + \alpha \sum_{i=1}^{n} (s_k)_i A_i^{(j)} \\ &= A^{(j)}(x_k) + \alpha \sum_{i=1}^{n} (s_k)_i A_i^{(j)} \\ &= A^{(j)}(x_k) + \alpha C_j(s_k) \\ C_j(s_{k+1}) &= \sum_{i=1}^{n} (s_{k+1})_i A_i^{(j)} \\ &= \sum_{i\in G} (-s_k)_i A_i^{(j)} + \sum_{i\notin G} (s_k)_i A_i^{(j)} \\ &= \sum_{i\in G} (s_k)_i A_i^{(j)} - \sum_{i\in G} (s_k)_i A_i^{(j)} - \sum_{i\in G} (s_k)_i A_i^{(j)} + \sum_{i\notin G} (s_k)_i A_i^{(j)} \\ &= C_j(s_k) - 2\sum_{i\in G} (s_k)_i A_i^{(j)} \\ &= C_j(s_k) - 2\Big(\sum_{i\in G, i\in P_k} A_i^{(j)} - \sum_{i\in G, i\in N_k} A_i^{(j)}\Big) \ \square \end{aligned}$$

The following is the algorithm for our SDD method that utilizes the previous iterations and our update formulas described in Theorem 3.2. We refer to this algorithm as SDD(updates).

SDD(updates) Algorithm

Initialization:

Let J be the set of detected necessary constraints and initialize $J = \emptyset$.

Choose x_0 in the interior of the feasible region.

Generate a random diagonal direction vector s_0 in the upper half of $\mathbb{R}^n$ with each entry chosen from $\{-1, 1\}$.

Set $k \longleftarrow 0$.

Initial Hitting Step:

Calculate $A^{(j)}(x_k) = A_0^{(j)} + \sum_{i=1}^{n}(x_k)_i A_i^{(j)}$ for $1 \leq j \leq q$.

Calculate $A^{(j)}(x_k)^{-\frac{1}{2}}$ for $1 \leq j \leq q$.

Calculate $C_j(s_k) = \sum_{i=1}^{n}(s_k)_i A_i^{(j)}$ for $1 \leq j \leq q$.

Calculate $B_j(s_k, x_k) = -A^{(j)}(x_k)^{-\frac{1}{2}} C_j(s_k) A^{(j)}(x_k)^{-\frac{1}{2}}$ for $1 \leq j \leq q$.

Find σ_j^+, σ_j^- for $1 \leq j \leq q$ as described in (6) and (7).

Determine $\sigma^+ = min\{\sigma_j^+ \mid 1 \leq j \leq q\}$ and $\sigma^- = min\{\sigma_j^- \mid 1 \leq j \leq q\}$.

For $1 \leq \ell \leq q$, if $\sigma_\ell^+ = \sigma^+$ or $\sigma_\ell^- = \sigma^-$ and $\ell \notin J$, then set $J = J \cup \{\ell\}$.

Repeat:

Update:

Choose $u \in U(0, 1)$.

Calculate $\alpha = \sigma^+ + u(\sigma^- - \sigma^+)$.

Calculate $A^{(j)}(x_k) \longleftarrow A^{(j)}(x_k) + \alpha C_j(s_k)$ for $1 \leq j \leq q$.

Calculate $A^{(j)}(x_k)^{-\frac{1}{2}}$ for $1 \leq j \leq q$.

Calculate $C_j(s_k) \longleftarrow C_j(s_k) - 2(\sum_{i\in G, i\in P_k} A_i^{(j)} - \sum_{i\in G, i\in N_k} A_i^{(j)})$ for $1 \leq j \leq q$.

Calculate $B_j(s_k, x_k) = -A^{(j)}(x_k)^{-\frac{1}{2}} C_j(s_k) A^{(j)}(x_k)^{-\frac{1}{2}}$ for $1 \leq j \leq q$.

Set $x_k \longleftarrow x_k + \alpha s_k$.

Search Direction:

Generate a random diagonal direction vector s_{k+1} in the upper half of $\mathbb{R}^n$ different from s_k with each entry chosen from $\{-1, 1\}$.

Save list G of indices i where $(s_{k+1})_i = -(s_k)_i$.

Hitting Step:

Find σ_j^+, σ_j^- for $1 \leq j \leq q$ as described in (6) and (7).

Determine $\sigma^+ = min\{\sigma_j^+ \mid 1 \leq j \leq q\}$ and $\sigma^- = min\{\sigma_j^- \mid 1 \leq j \leq q\}$.

For $1 \leq \ell \leq q$, if $\sigma_\ell^+ = \sigma^+$ or $\sigma_\ell^- = \sigma^-$ and $\ell \notin J$, then set $J = J \cup \{\ell\}$.

Update:

Set $k \longleftarrow k + 1$.

Until a stopping condition is reached.

Again, at the end of the algorithm, all of the LMI constraints listed in J are declared necessary.

Using the SDD(updates) algorithm, the total number of multiplications to calculate $A^{(j)}(x_{k+1})$ is m_j^2 and the additions total m_j^2. To compute $C_j(s_{k+1})$ requires $|G|m_j^2$ additions, where $|G|$ denotes the number of elements in G, as well as m_j^2 multiplications. Then the total number of computations, including the cost for calculating $A^{(j)}(x_k)^{-\frac{1}{2}}$ and the eigenvalues of $B_j(s_k, x_k)$, for SDD(updates) is $\bigcirc(qnm_q^2 + qm_q^3)$ per iteration. So, we have:

Theorem 3.3 *Each iteration of SDD(updates) is $\bigcirc(qnm_q^2 + qm_q^3)$ time.*

SDD with the update formulas is a significant improvement of SDD. Observe that $1 \leq |G| \leq (n-1)$. Using the updates in SDD, for each j, we save between $(3n-3)m_j^2$ and $(4n-5)m_j^2$ computations per iteration. This improvement is even more significant for problems with large n or m_j.

4 Numerical Experiments

In this section, we provide numerical experiments to compare SDD with SDD(updates) as well as SCD with SDD(updates). All numerical experiments were done using a Dell OptiPlex GX260 computer with codes written in MATLAB©7.0.4. We generated each test problem randomly using MATLAB©. In all of the problems, each LMI $A_0^{(j)} + \sum_{i=1}^{n} x_i A_i^{(j)} \succeq 0$ was generated as follows: $A_0^{(j)}$ is an $m_j \times m_j$ diagonal matrix with each diagonal entry chosen from $U(0,1)$. Each $A_i^{(j)}$ $(1 \leq i \leq n)$ is a random $m_j \times m_j$ symmetric and sparse matrix with approximately $0.8 * m_j^2$ nonzero entries; each entry is the sum of one or more normally distributed random samples. This ensures that each of our test problems are random and are not biased toward a desired result. Note that the origin is an interior point for each problem.

The test problems we used to collect our data can be seen in Table 1. The first problem, Data1, is the same problem as our example SDP; the others were all randomly generated.

Using these problems, we were able to compare the SDD algorithm without updates to that using updates. The results can be seen in Table 2. The time to detect the necessary constraints and the cost per iteration are given in seconds. For each simulation, we stopped the detection algorithms after $10 * n * q$ iterations.

According to the data we recorded, it appears that the SDD algorithm that uses updates is much more efficient in the cost per iteration and time to complete the

Example	No. LMI Constraints (q)	Dimension (n)	Size of Matrices (m_j)
Data1	5	2	2,2,2,1,1
Data2	7	2	3,4,4,2,5,2,1
Data3	50	2	10
Data4	50	4	3
Data5	50	5	5
Data6	50	10	5
Data7	50	15	5
Data8	50	5	10
Data9	50	10	10
Data10	50	15	10
Data11	100	5	5
Data12	100	10	5
Data13	100	15	5
Data14	100	5	10
Data15	100	10	10
Data16	100	15	10
Data17	200	5	5
Data18	200	10	5
Data19	200	15	5
Data20	200	5	10
Data21	200	10	10
Data22	200	15	10

Table 1: Test Problems.

Problem	SDD (without updates) No. Necessary Detected	No. Iterations	Time to Detect	Cost per Iteration	SDD(updates) No. Necessary Detected	No. Iterations	Time to Detect	Cost per Iteration
Data1	4	75	.5896	.0079	4	75	.4500	.0060
Data2	2	1	.1615	.1615	2	1	.0312	.0312
Data3	5	5	.5681	.1136	5	5	.4299	.0860
Data4	10	1531	92.4739	.0604	10	1531	78.3154	.0512
Data5	8	994	67.3789	.0678	8	994	55.8362	.0562
Data6	17	3691	307.0706	.0832	16	3606	215.7010	.0598
Data7	23	5374	514.7055	.0958	23	5441	345.7188	.0635
Data8	7	488	46.0921	.0945	7	488	39.6604	.0813
Data9	18	4919	530.3107	.1078	17	2623	221.9173	.0846
Data10	15	3334	408.7728	.1226	16	2971	261.4190	.0880
Data11	14	877	117.8032	.1343	14	877	97.5058	.1112
Data12	13	642	118.4216	.1845	13	642	102.7755	.1601
Data13	30	6704	1290.8000	.1925	31	5621	697.1669	.1240
Data14	9	904	168.2120	.1861	9	904	146.1798	.1617
Data15	18	911	195.2947	.2144	18	911	152.3518	.1672
Data16	25	11893	2915.2000	.2451	24	2048	358.1646	.1749
Data17	10	1945	521.5488	.2681	10	1945	434.1487	.2232
Data18	21	4644	1496.7000	.3223	22	18104	4254.7000	.2350
Data19	21	5046	1907.1000	.3780	21	5046	1243.6000	.2465
Data20	11	1184	438.1432	.3701	11	1184	380.7561	.3216
Data21	29	10955	4670.3000	.4263	29	12069	4038.9000	.3347
Data22	28	26457	12982.0000	.4873	28	13069	4529.8000	.3466

Table 2: Comparison Between SDD and SDD(updates)

algorithm. In every problem we tested, SDD(updates) had a lower cost per iteration than SDD without updates and, therefore, completed the algorithm more quickly. These are real problems that prove our theory that using updates would reduce the necessary computations and save time.

There are 9 test problems in the Table 2 that show differences in the number of necessary constraints detected or the number of iterations to detect them between the two SDD methods. All of the instances where the two methods differ involve a larger dimension (n), 10 or 15; the difference never occurred when we used $n \leq 5$. The higher dimension results in more iterations and, therefore, more calculations per iteration. We also know that SDD without updates does more calculations per iteration than SDD(updates) so there are more opportunities for round-off error in the algorithm that does not utilize updates. Since SDD(updates) does fewer calculations per iteration, it is a more accurate algorithm.

We also want to compare SDD with SCD. We use SDD(updates) in this comparison because we have already seen that it is the more efficient and accurate of the two SDD algorithms. Table 3 shows the results of these tests.

	SDD(updates)				SCD			
Problem	No. Necessary Detected	No. Iterations	Time to Detect	Cost per Iteration	No. Necessary Detected	No. Iterations	Time to Detect	Cost per Iteration
Data1	4	75	.4500	.0060	4	43	.2390	.0056
Data2	2	1	.0312	.0312	2	2	.0417	.0208
Data3	5	5	.4299	.0860	5	18	1.4082	.0782
Data4	10	1531	78.3154	.0512	9	1393	68.3314	.0491
Data5	8	994	55.8362	.0562	8	408	21.7229	.0532
Data6	16	3606	215.7010	.0598	16	4709	250.3382	.0532
Data7	23	5441	345.7188	.0635	23	1339	71.2683	.0532
Data8	7	488	39.6604	.0813	6	2456	191.4582	.0780
Data9	17	2623	221.9173	.0846	18	1267	97.1609	.0767
Data10	16	2971	261.4190	.0880	15	1573	120.8073	.0768
Data11	14	877	97.5058	.1112	14	126	13.2841	.1054
Data12	13	642	102.7755	.1601	13	107	16.4342	.1536
Data13	31	5621	697.1669	.1240	34	9837	1043.5000	.1061
Data14	9	904	146.1798	.1617	9	283	43.5360	.1538
Data15	18	911	152.3518	.1672	19	6184	953.0933	.1541
Data16	24	2048	358.1646	.1749	26	9706	1498.8000	.1544
Data17	10	1945	434.1487	.2232	10	3372	716.8151	.2126
Data18	22	18104	4254.7000	.2350	21	5226	1108.5000	.2121
Data19	21	5046	1243.6000	.2465	21	5397	1144.9000	.2121
Data20	11	1184	380.7561	.3216	11	2171	670.0674	.3086
Data21	29	12069	4038.9000	.3347	31	19403	5987.2000	.3086
Data22	28	13069	4529.8000	.3466	28	4129	1280.8000	.3102

Table 3: Comparison Between SDD(updates) and SCD

According to our results comparing SDD(updates) with SCD, it appears that neither has a significant advantage over the other. In 10 or our 22 test problems, SDD(updates) out-performed SCD by either finding more necessary constraints or finding the same number of necessary constraints in fewer iterations. In 4 test problems, SDD(updates) detected more necessary LMI constraints than SCD. In 6 of our problems, SDD(updates) detected the same number of necessary LMI constraints as SCD but in fewer iterations and less time. On the other hand, SCD detected more constraints than SDD(updates) in 5 test problems and was faster than SDD(updates) in the remaining problems. It appears that each of the algorithms have their advantage. This advantage likely depends on the format of the feasible region of the SDP. Because our examples are generated randomly, we did not control the format of the region.

5 Conclusion

Using the most efficient Monte Carlo method for detecting necessary LMI constraints for an SDP, we adapted SCD to create the SDD method. After analyzing the cost per iteration, we also developed the SDD(updates) algorithm to improve on the time complexity. Using randomly generated problems of random LMI's, we compared these three methods. Our SDD(updates) method was clearly more efficient than our original SDD algorithm due to decreased computations per iteration by applying Theorem 3.2.

When we looked at the differences between SCD and SDD(updates), however, we were not able to determine such a clear result. Both algorithms appear to have strengths and weaknesses in comparison. The original SCD method always had a lower cost per iteration due to the convenient format of the direction vector s. On the other hand, both SDD(updates) and SCD were able to out-perform the other in various cases (roughly half for each algorithm) of our test problems. We attribute this to the format of the feasible region.

Because our test problems were randomly generated, we did not control any factors that might affect the format of the feasible region. For this reason, we were unable to draw any specific conclusions as to when SDD(updates) will perform better or worse than SCD. Future research into the format of the feasible region of an SDP and its effect on the performance of these two algorithms would be very beneficial. Knowing which algorithm will perform better in different situations would allow us to be able to choose the best one based on a specific optimization problem and its feasible region, thus further saving time and money. Along with determining which algorithm works best with different feasible region formats, it would be beneficial to note if there is a specific format that occurs more often than others on average. This information would allow us to also determine which algorithm would be better on average.

We are also interested in applying this algorithm using diagonal directions to the extreme point problem. In the past, various methods have been used to determine extreme versus redundant points. One method used is related to SCD; it is called the CD-EP algorithm. We could use similar conversions to create a DD-EP algorithm for the extreme point problem. Because finding extreme points is related to detecting necessary linear constraints, we would have $m_q = 1$. This would make the SDD(updates) and SCD algorithms (and their related extreme point algorithms) similar in cost and time per iteration. It would be interesting to see, then, how diagonal directions affect the detection of extreme points in comparison with the coordinate direction algorithm.

References

[1] S. J. Benson, Yinyu Ye and Xiong Zhang, "Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization", *SIAM Journal on Optimization*, Vol. 10, no. 2, 2000, pp. 443-461.

[2] A. Boneh, S. Boneh and R. J. Caron, "Constraint classification in mathematical programming", *Mathematical Programming*, vol. 61, no. 1-3, August 1993.

[3] R. J. Caron, "Redundancy in Nonlinear Programming", *Encyclopedia* January 1999.

[4] R. Caron, A. Boneh and S. Boneh, "Estimating the Prediction Function and the Number of Unseen Species in Sampling With Replacement", *JASA*, Vol. 93, no. 441, 1998, pp. 372-379.

[5] R. J. Caron, A. Boneh and S. Boneh, "Redundancy", *Advances in Sensitivity Analysis and Parametric Programming*, Chapter 13, 1997.

[6] J. W. Chinneck, "Discovering Characteristics of Mathematical Programs via Sampling", June 2000.

[7] S. Jibrin and I. Pressman, "Monte Carlo Algorithms for the Detection of Necessary Linear Matrix Inequality Constraints", *International Journal of Nonlinear Sciences and Numberical Simulation*, vol. 2, no. 2, 2001.

[8] M. H. Karwan, V. Lofti, J. Telgen, and S. Zionts, "Redundancy in Mathemathical Programming: A State-of-the-Art Survey", Springer-Verlag, Berlin, 1983.

[9] A. Lewis, "The mathematics of eigenvalue optimization", Simon Fraser University, Canada, April 2003.

[10] M. V. Ramana, "An algorithmic analysis of multiquadratic and semidefinite programming problems", Ph. D. Thesis, The John Hopkins University, Baltimore, 1993.

[11] K. C. Toh, R. H. Tütüncü and M. J. Todd, "Inexact primal-dual path-following algorithms for a special class of convex quadratic SDP and related problems", *Technical Report 1421*, School of Operations Research and Industrial Engineering, Cornell University, New York, February 2005.

[12] J. G. VanAntwerp and R. D. Braatz, "A tutorial on linear and bilinear matrix inequalities", *Journal of Process Control*, 2000.

[13] L. Vandenberghe and S. Boyd, "Semidefinite Programming", *Siam Review*, March 1996.