Approximating Quadratic Programs with Semidefinite Relaxations

Michael Dinitz
Advisor: Moses Charikar
3 May 2004

Abstract

Given an arbitrary matrix $A$ in which all of the diagonal elements are zero, we would like to find $x_1, x_2, \ldots, x_n \in \{-1, 1\}$ such that $\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j$ is maximized. This problem has an important application in correlation clustering, and is also related to the well-known inequality of Grothendieck in functional analysis. While solving quadratic programs is NP-hard, we can approximate the solution by using the canonical semidefinite relaxation. We would like to know how good an approximation this provides. In this paper we explore this problem computationally in an attempt to find matrices $A$ that result in a large gap between the semidefinite relaxation and the quadratic program.

1 Introduction

This project is about determining how well a semidefinite program might be used to approximate a particular quadratic program. The quadratic program we are interested in is actually fairly general: given a $n \times n$ matrix $A$, maximize

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j, \quad \text{s.t. } x_i \in \{-1, 1\} \text{ and } a_{ii} = 0 \text{ for all } i.$$  \(1\)

This maximization problem is known as \textsc{MaxQP} [4]. Until recently, there was no known good approximation algorithm for this problem, but Charikar and Wirth recently found a polynomial time algorithm that gives a $\Omega(1/\log n)$ approximation [4]. Their algorithm is based on solving the canonical semidefinite relaxation of \textsc{MaxQP}:
maximize
\[ \sum_{i,j} a_{ij} v_i \cdot v_j, \quad \text{s.t. } v_i \cdot v_i = 1 \text{ and } v_i \in \mathbb{R}^n \text{ for all } i. \] (2)

If \( n = 1 \) then this is clearly identical to (1), but in higher dimensions it seems to allow more flexibility.

The goal of this project was to investigate the relationship between MAXQP and its semidefinite relaxation in order to find out what sort of approximation factor the semidefinite relaxation might provide. We do this by trying to find matrices \( A \) for which the gap between the quadratic program and the semidefinite program is large. Simply trying out various matrices is impractical because we cannot efficiently solve the quadratic program. On the other hand, we can find certain “bad” matrices by creating a linear program whose solution is the constraint matrix for our quadratic and semidefinite programs. Finding matrices that result in a large gap will give at least a lower bound on the size of the gap in general.

We begin by presenting the previous work on this and related problems in Section 2. We then discuss our approach in Section 3, our actual implementation and methodology in Section 4, and then our results in Section 5.

2 Previous Research

This quadratic maximization problem has not been the subject of much research. In fact, the only people who have examined it are Charikar and Wirth, who studied it in [4] and found a \( \Omega(1/\log n) \) approximation algorithm for the quadratic program that uses the semidefinite program. However, while this particular program has not been extensively studied, a similar problem has received slightly more attention. This is the problem of maximizing
\[ \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_i y_j, \quad \text{s.t. } x_i, y_j \in \{-1, 1\} \text{ for all } i, j. \] (3)
The canonical semidefinite relaxation of this problem is maximizing

\[ \sum_{i,j} a_{ij} (u_i \cdot v_j), \quad \text{s.t. } u_i \cdot u_i = v_j \cdot v_j = 1 \text{ and } u_i, v_j \in \mathbb{R}^n \text{ for all } i, j. \]  

(4)

An important result in functional analysis is Grothendieck’s inequality, which states that the maximum value of (4) is always within a constant factor of the maximum of (3) [6]. The precise value of this constant, called Grothendieck’s constant, is not known, but it is at most 1.782... and at least \( \pi/2 = 1.570... \) [1].

This problem is more than just similar to ours – it is actually a special case [4]. To see this, just let \( z = (x, y) \) and \( B = [0 \ A \ 0] \). Then (3) becomes \( z^T B z \), and so is an instance of (1). Unfortunately this means that we do not have an obvious proof that the ratio between \text{MaxQP} and its semidefinite relaxation is bounded by a constant, but it does imply that the lower bound for Grothendieck’s constant \( \pi/2 \) is also a lower bound for the gap between \text{MaxQP} and its semidefinite relaxation.

3 Approach

As mentioned before, our basic idea was to find matrices that result in a large SDP gap. However, we couldn’t just compare the SDP and QP optimums for various matrices, as that would involve actually solving \text{MaxQP}, which we do not know how to do efficiently since \text{MaxQP} is NP-hard. So we took a different approach: instead of fixing the constraint matrix, we fixed the vector solution to the semidefinite program. We can then solve a linear program to find a constraint matrix that will maximize the gap between the semidefinite program and the quadratic program if the solution to the SDP is the one we provide. This linear program is the following: minimize \( c \) such that \( \sum_{i,j} a_{ij} (v_i \cdot v_j) = 1, \sum_{i,j} a_{ij} x_i x_j \leq c \) for all \( x \in \{-1, 1\}^n \), and \( a_{ij} = 0 \) for all \( i, j \) with \( i \geq j \). We included the last constraint because we only want one variable for each \( i, j \) pair. In this program the variables we are solving for are the \( a_{ij} \) values and \( c \),
while the $v_i$ vectors are fixed parameters. Since this is a linear program we can solve it in time polynomial in the size of the program using well-known methods. However, note that there are an exponential number of constraints, as there is a constraint for every $x \in \{-1, 1\}^n$. Thus the size of the program is exponential in the size of the input, so even though we can solve a linear program in polynomial time we will actually need exponential time since the size of the program grows exponentially.

The logic behind this linear program is fairly intuitive. The first requirement guarantees that the SDP optimum is at least 1, and the second constraint simply finds the matrix that gives the worst performance on the quadratic program given that the SDP optimum is at least 1. Since the optimum of the SDP is at least as good as the vector solution we are providing (which gives a value of 1), the inverse of this program will be a lower bound on the SDP gap for the matrix, which will be the matrix with the largest gap possible given the vector solution we provide.

It is not immediately obvious why this approach is any better than simply trying different constraint matrices, since we need exponential time with our method as well. The main reason is that we have no idea what kind of matrix would result in a large SDP gap, so we would not know what to try. On the other hand, we do have some idea of what kind of vectors might be the solution to a problem with a large gap. Since the quadratic program is simply the semidefinite program restricted to one dimension, intuitively the semidefinite program should be able to do much better when its solution takes advantage of the extra dimensions. So by choosing vectors that approximate the entire space (equally spaced vectors in 2-dimensions, vectors corresponding to the vertices of the Platonic solids in three dimensions, and random vectors in any dimension) we would expect the semidefinite program to do much better than the quadratic program. So instead of looking for constraint matrices that give a large gap without any real method of finding these matrices, by using this approach we have at least a few things to try that seem promising. This also
allows the possibility of finding similarities between matrices that give large gaps, after which we can test individual matrices that might seem interesting.

An added benefit of this approach is in the implementation. Much work has been done on solving linear programs, and current industrial strength linear program solvers are extremely fast. Semidefinite programs, on the other hand, have not yet received this amount of attention. There are semidefinite program solvers that run in polynomial time, but they have not been optimized nearly as much as LP solvers have been. We might as well take advantage of existing high-quality implementations, and our strategy of solving a single linear program rather than a semidefinite program and a quadratic program lets us make more efficient use of pre-existing tools.

4 Methodology

4.1 Solving linear programs

While there are many good linear program solvers available, we decided on a higher level approach and used AMPL, the mathematical programming modelling language developed by Fourer, Gay, and Kernighan [5]. AMPL is nice because it is a modelling language, and so is much easier to use than any direct linear program solvers. This also makes it easier to write code that is very adaptable, allowing us to easily carry out many tests in various dimensions and with varying numbers of vectors. The modularity it affords by separating the model from the data also makes it possible to simply write a script to generate different input data without changing the model at all, which is a nice feature. We did not have to worry about the mechanics of solving the linear program at all – a 20-line model file and a script to generate inputs was all that was needed.

Using a solver directly, on the other hand, would have required a significant amount of work and time that could have been better spent running tests and exam-
ining results. At first we were considering using CPLEX, a library for solving linear programs that would have allowed us to combine creating an instance of the problem and solving it in one C program. However, we would have had to explicitly construct an instance of the problem by creating the constraint matrix. While not extremely difficult, this step did not seem necessary since AMPL could do it for us. It would also have been slightly more difficult to preserve modularity between the model and the instance, and thus write code that would be easily adaptable for different instances of the program. Fortunately we ended up with the best of both worlds, since AMPL has the ability to use a variety of different solvers, CPLEX being one of them. The version of AMPL we used actually used CPLEX by default, so we got the power and speed of CPLEX without having to actually delve into the details of the linear program, instead using a simple AMPL model.

The AMPL model we ended up with (see Listing 1) is both simple and flexible, easily allowing various dimensions and vectors. It requires that all possible \( x \) parameters be given, but handles all possible dimensions and vector counts. AMPL does not have a built-in function for taking the dot product of vectors, but it was easy to implement by simply taking the sum of the products of each coordinate. In order to make it general enough to handle arbitrary dimensions and vector counts, we combined all of the vectors into one two-dimensional parameter indexed by \( i \) and \( j \) with \( i \leq n \) specifying the vector and \( j \leq d \) specifying the coordinate (where \( d \) is the dimension of the space). We also combined all of the \( x \) vectors into a different two-dimensional parameter, this one indexed by \( i \) and \( j \) with \( i \leq 2^n \) specifying the \( x \) vector and \( j \leq n \) specifying the element of the vector. Designing the model this way allows us to leave the model unchanged for different problem instances – we can simply provide new data for the same model.

Listing 1: AMPL model

\[
\text{param n >= 1;}
\]
param dim >= 1;
param u {1..n, 1..dim};
param x {1..2^n, 1..n};
var a {1..n, 1..n};
var c;
minimize min_value: c;
subject to SDP: 
  sum {i in 1..n, j in 1..n} 
    (a[i,j] * (sum {d in 1..dim} u[i,d] * u[j,d])) = 1;
subject to main {d in 1..2^n}:
  sum {i in 1..n, j in 1..n} (a[i,j] * x[d,i] * x[d,j]) <= c;
subject to extra {i in 1..n, j in 1..n: i>=j}: a[i,j] = 0;

4.2 Generating problem instances

Once we had the model, the next step was to create some input data for it. A problem instance is specified in AMPL by simply giving values for the parameters of the model, which for us is the dimension, the number of vectors, the actual vectors, and every $x \in \{-1,1\}^n$. An input for two dimensions and three vectors might look as follows:

Listing 2: Sample data file

param dim := 2;
param n := 3;
param u:
  1 2 :=
  1 1.000000 0.000000
  2 -0.500000 0.866025
  3 -0.500000 -0.866025 ;
param x:
  1 2 3 :=
  1 -1 -1 -1
  2 1 -1 -1
  3 -1 1 -1
  4 1 1 -1
  5 -1 -1 1
  6 1 -1 1
  7 -1 1 1
  8 1 1 1 ;
As previously mentioned, we thought that equally spaced vectors in two dimensions might result in an $A$ matrix with a large gap, so that was the first type of input we tried. We wrote a C program that took an angle (measured in degrees) as an argument, and then generated unit vectors starting at $(1,0)$ that were offset by the specified angle from the previous vector. So if the supplied degree was a divisor of 360, this resulted in equally spaced unit vectors in the plane. It finds the vectors through some pretty obvious trigonometry: if the angle in degrees between adjacent vectors is $d$, then the $x$ coordinate of the $i$th vector is $\cos(i \cdot d \cdot \pi/180)$ and the $y$ coordinate is $\sin(i \cdot d \cdot \pi/180)$, where we converted to radians because the C sin and cos functions take radians as arguments rather than degrees. To generate the $x$ values, we iterated over all integers from 0 to $2^n - 1$ and for each one used bit masking to find out which bits in the representation were 0 and which were 1. Every time the program saw 0 it printed out $-1$, and each time is saw 1 it printed out 1. Since each number from 0 to $2^n - 1$ has a unique binary representation in $n$ bits, this clearly results in all possible strings in $\{-1, 1\}^n$.

We also wanted to investigate what would happen with random vector sets, since for higher dimensions the concept of equally spaced vectors is much trickier, and the easiest way to approximate the entire space is simply to use a large number of random vectors. To do this we wrote another C program, one which takes three parameters: the dimension ($d$), the number of vectors ($n$), and a random seed. Printing out the dimension and number of vectors is trivial, and the $x$ values are obtained in the same way they were in the equally spaced case. Then for each vector we simply pick $d$ random values from the Gaussian distribution with mean 0 and variance 1, then scale the whole vector to make it unit length. We have to pick from a Gaussian distribution in order to make sure the distribution of vectors is spherically symmetric. Unfortunately there is no standard C function that picks random numbers from a Gaussian distribution, only ones that get random numbers from a uniform distribution. So we
used the C function drand to obtain uniformly distributed random numbers between 0 and 1, and then used the standard Box-Muller transformation [2] to change these into random numbers from the unit Gaussian distribution.

As with anything random, we wanted to repeat this experiment many times to witness a variety of cases. It impractical to run each experiment by hand, so we had to automate the process. We created a shell script that took the dimension $d$, the number of vectors $n$, a random seed, and the number of problem instances to be generated as parameters. The script then calls the C program described in the last paragraph once for each problem, getting a new random seed each time through the shell’s built-in random number generator. The output of each call to the program gets written to its own file and put in a special directory called testcases. We then iterate over each file in testcases, printing the AMPL commands to first read in new data from each file and then solve and display the results. We finally call AMPL with this list of commands as its input.

The last family of problem instances we wanted to check were those in which the vectors were the vertices of the Platonic solids. These vectors are the three dimensional analog of equally spaced vectors in two dimensions. The five Platonic solids are convex polyhedra in which every face is an identical polygon: the tetrahedron, the octahedron, and the icosahedron are made out of triangles, the cube is made out of squares, and the dodecahedron is made out of pentagons. Since there are only five, there was no reason to write a program to generate them – we could simply construct the input files by hand, one for each solid, and feed them into AMPL ourselves. Unfortunately the dodecahedron proved to be too large for our program to handle (more on this in the next section).
5 Experimental Results

Unfortunately, we were not able to find a family of vectors that caused the gap to increase past a constant. Thus we were not able to show that the gap between MaxQP and its semidefinite relaxation is not bounded by a constant, which would show that the $\Omega(1/\log n)$ approximation algorithm designed by Charikar and Wirth [4] might be about as good as can be expected. Note this does not imply that the gap is bounded by a constant, simply that we were unable to find a counterexample. The largest gap we observed was 1.5, which is not even as good a lower bound as the $\pi/2$ lower bound we get from Grothendieck’s inequality. However, achieving $\pi/2$ requires a very large number of dimensions and a large number of vectors [1], so 1.5 might be a bound for lower dimensions.

5.1 Two dimensions

The first vector sets we tried were equally spaced vectors in two dimensions. Table 1 summarizes our results. Unfortunately we could not test above 15 equally spaced vectors, since the exponential running time makes it impractical to test any more than 15 vectors. An input of 15 vectors takes 20 minutes or so to run, and 16 takes almost 5 minutes just to load in the data – we never ran it to completion, but it would have taken at least a day.

The most obvious observation is that there is a pattern: the gap is 1.5 whenever the number of vectors $n$ is a multiple of 3. This suggests that there is something special about having three vectors all at 120 degrees from each other, and that having extra vectors does not matter as long as we have those three. Examining the matrices found for these vectors that maximize the gap supports this: there are always only three nonzero entries, and these entries are always in locations $a_{ij}$, $a_{jk}$, and $a_{ik}$, where $i$, $j$, and $k$ are vectors that are all at 120 degrees from each other. The absolute value
Table 1: Equally Spaced 2-D Vectors

<table>
<thead>
<tr>
<th>Vectors</th>
<th>Angle</th>
<th>LP value</th>
<th>SDP gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>120</td>
<td>0.666666</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>72</td>
<td>0.700745</td>
<td>1.427053</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>0.666666</td>
<td>1.5</td>
</tr>
<tr>
<td>7</td>
<td>51.429</td>
<td>0.675572</td>
<td>1.48</td>
</tr>
<tr>
<td>8</td>
<td>45</td>
<td>0.707106</td>
<td>1.414215</td>
</tr>
<tr>
<td>9</td>
<td>40</td>
<td>0.666666</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
<td>0.700745</td>
<td>1.427053</td>
</tr>
<tr>
<td>11</td>
<td>32.72</td>
<td>0.67006</td>
<td>1.4924</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
<td>0.666666</td>
<td>1.5</td>
</tr>
<tr>
<td>13</td>
<td>27.6923</td>
<td>0.668554</td>
<td>1.4958</td>
</tr>
<tr>
<td>14</td>
<td>25.714</td>
<td>0.671514</td>
<td>1.48917</td>
</tr>
<tr>
<td>15</td>
<td>24</td>
<td>0.666666</td>
<td>1.5</td>
</tr>
</tbody>
</table>

of each nonzero entry is also always equal to 0.666666, which is the minimum LP value obtained. The matrix is essentially picking out three vectors and taking the dot product of each possible combination.

Interestingly enough, this behavior is not unique to values of $n$ that are multiples of three. When $n \neq 4$ we see similar behavior, just with not as good minimum objective values and with vectors at angles other than 120 degrees from each other. The $n = 4$ is obviously different, as is discussed below. In the other cases, though, the linear program is trying to approximate three equally spaced vectors by picking out the three vectors available that come closest to being separated by 120 degrees. There is something fundamentally better about 3 equally spaced vectors than other configurations.

The $n = 4$ case is somewhat different from the others since every vector is either perpendicular or antiparallel to every other vector. It has a gap of 1, indicating that the semidefinite program can not do any better than the quadratic program. This is intuitive since the dot product of perpendicular vectors is 0 and the dot product of antiparallel vectors is $-1$, but we also present a formal proof of this:

**Theorem 1.** If the vector solution to the semidefinite program is four equally spaced
vectors in two dimensions, then the SDP gap is 1.

Proof. Since the four vectors are equally spaced in two dimensions, they’re spaced at 90 degrees. Let vectors 1 and 3 be antiparallel and vectors 2 and 4 be antiparallel. Then vectors 1 and 2 are perpendicular, as are 1 and 4, 2 and 3, and 3 and 4. Then by the SDP constraint, \(-a_{13} - a_{24} = 1\). Assume \(c < 1\). If we let \(x_1 = -1, x_2 = -1, x_3 = 1, \) and \(x_4 = 1\) we get the constraint \(a_{12} - a_{14} - a_{23} + a_{34} - a_{13} - a_{24} < 1\). Then by substituting and doing some algebra we get that \(a_{12} + a_{34} < a_{14} + a_{23}\). Now if we let \(x_1 = 1, x_2 = -1, x_3 = -1, \) and \(x_4 = 1\) and then perform similar steps we get that \(a_{12} + a_{34} > a_{14} + a_{23}\). This is a contradiction, so \(c \geq 1\). We know that \(c \leq 1\), though, since there are two antiparallel vectors so at worst it can collapse into the 1 dimensional case. Hence the gap is 1.

These results naturally made us wonder about the three vector situation, since it results in the largest gap of any vector solution we tried. We designed another mathematical program to find vectors that resulted in the largest gap:

Listing 3: three vector optimum

```plaintext
var theta12; var theta13; var theta23; var a12; var a13; var a23; var c;
minimize min_value: c;
subject to extra1: theta12 >= 0.001;
subject to extra2: theta13 >= 0.001;
subject to extra3: theta23 >= 0.001;
subject to extra4: cos(theta13) = cos(theta12+theta23);
subject to SDP: (a12*cos(theta12)+(a13*cos(theta13))+
(a23*cos(theta23)))=1;
subject to one: a12+a13+a23 <= c;
subject to two: -a12-a13+a23 <= c;
subject to three: -a12+a13-a23 <= c;
subject to four: a12-a13-a23 <= c;
```

Unfortunately AMPL cannot handle strict inequalities, so we had to set each angle to greater than or equal to 0.001 instead of greater than 0. Also, this program is nonlinear due to the cosines in the constraints, which is clearly a problem. We could
have used the same formulation of the dot product as we have previously (the sum of products of coordinates), but that would be quadratic and so still nonlinear, and this way we immediately get the relationship between the vectors rather than just a random-looking set of vectors. CPLEX cannot solve nonlinear programs, but AMPL can be hooked up to other solvers that can. We set the options in AMPL so that it fed its output into MINOS, a solver developed at the Stanford System Optimization Lab by Bruce Murtagh and Michael Saunders that can solve nonlinear programs.

The solution MINOS finds to this problem is, surprisingly, not the three equally spaced vectors we had expected, but is instead one vector at 0 degrees, one at 60 degrees, and another at 120 degrees (e.g. (1, 0), (0.5, 0.866025), (−0.5, 0.866023)). We will refer to this vector configuration as the second configuration. However, this solution is equal to the three equally spaced vectors in that they both result in a $c$ value of $2/3$, and thus a gap of 1.5. So when the number of vectors is a multiple of three it can use the equally spaced vectors, and when the number of vectors is a multiple of 6 it can use either three equally spaced vectors or the second configuration. It gives the same gap in either case. This shows that with a three vector solution the gap is at most 1.5.

Since we had no other intuition for configurations in 2 dimensions that might be better than three equally spaced vectors, we turned to generating random vectors, as described in the previous section. Table 2 summarizes the results of random vector testing in 2 dimensions. We generated 100 random vector sets for each value of $n$ and found the average value of the linear program, the minimum value of the LP, and the maximum gap (the inverse of the min LP value).

While some of the instances resulted in a gap of close to 1.5, none of them quite achieved it. Adding more vectors seems to help slightly, but not enough to have a significant effect. It is also worth noting that in every case we observed the same behavior in the resulting matrix as in the equally spaced case (picking three vectors...
Table 2: Random 2D Vectors

out, every other coefficient equal to zero). There were always exactly three nonzero entries in the matrix, corresponding to the three combinations of three vectors. These three vectors were always close to either being equally spaced or at 60, 60, and 120 degrees from each other (the second configuration), which are the two cases we found that give a gap of 1.5. This is added evidence that it is simply trying to pick vectors to achieve a gap of 1.5, and if those vectors are not available it picks three vectors that are closest to one of the two cases that give a gap of 1.5.

5.2 Three dimensions

In three dimensions we had two things we wanted to try: random vectors and the Platonic solids. We used the same program to generate and analyze random 3d vectors as we did to generate and analyze random 2d vectors, and again we tried 100 random instances for each value of $n$. Table 3 summarizes the results of these experiments.

Table 3: Random 3D Vectors

With the same number of vectors, the maximum gap in three dimensions is notice-
ably smaller than in two dimensions. While initially surprising, since it might seem
that the higher the dimension the more freedom available in picking vectors and thus
the better the semidefinite program can do, this actually makes sense. The larger the
space the more vectors will be needed to cover the same fraction of it, so 12 random
vectors do a much better job at simulating the entire two dimensional space than at
simulating the entire three dimensional space. So we should not be surprised that
we did not find any program that results in a large gap when the SDP is in three
dimensions, since we did not find one in two dimensions either.

We also noticed the same behavior in the resulting $A$ matrix of each of the in-
estances we generated as in the two dimensional case. It was still picking out three
vectors and taking the dot products of each of the three combinations. It is not quite
as clear in these cases whether it was attempting to approximate either of the two
dimensional cases that give gaps of 1.5 (three vectors all at 120 degrees from each
other or three vectors at 60, 60, and 120 degrees from each other), but they seem
like likely possibilities, especially since we could not find any construction in three
dimensions that results in a larger gap.

The last group of vector sets we tested experimentally were the vertices of the
Platonic solids. These are analogs of the equally spaced vectors in two dimensions –
since every face is identical and the same number of faces intersect at every vertex,
there is a kind of regularity governing the angles between vectors that is similar to
the equally spaced vectors in 2D. Namely, any vector is the same as any other vector
in terms of how it interacts with the rest of the vectors. Because of the exponential
size of the linear program in terms of the number of vectors, we were unable to test
the dodecahedron, which has 20 vectors. We were able to test the rest of the Platonic
solids, though, and the results are summarized in Table 4.

The most obvious observation when looking at this table is that three of the four
solids result in a gap of 1, so the semidefinite program cannot do any better than the
Theorem 2. If the vector solution to the semidefinite program is the set of vectors corresponding to the vertices of the tetrahedron, then the SDP gap is at least 1.

Proof. Note that the dot product between any two vectors in the tetrahedron is $-1/3$. We start by setting the SDP value to 1, which gives the equation $-1/3 a_{12} - 1/3 a_{13} - 1/3 a_{14} - 1/3 a_{23} - 1/3 a_{24} - 1/3 a_{34} = 1$. Then $a_{12} + a_{13} + a_{14} + a_{23} + a_{24} + a_{34} = -3$. If we let $x_1 = -1$, $x_2 = -1$, $x_3 = 1$, and $x_4 = 1$, we get the equation $a_{12} - a_{13} - a_{14} - a_{23} - a_{24} + a_{34} \leq c$, where $c$ is the optimum value of the quadratic program. Subtracting $a_{12}$ and $a_{34}$ then substituting from the SDP constraint and dividing gives $-2(a_{13} + a_{14} + a_{23} + a_{24}) \leq c + 3$. Dividing by $-2$ and thus switching the inequality gives $a_{13} + a_{14} + a_{23} + a_{24} \geq -\frac{c + 3}{2}$. Substituting again from the SDP constraint and doing some basic algebra yields the constraint $a_{12} + a_{34} \leq \frac{c - 3}{2}$. Assume that $c < 1$. Then $a_{12} + a_{34} < -1$. Note that we could have done this for any $a_{ij} + a_{kl}$, $i \neq j \neq k \neq l$, by setting $x_i$ and $x_j$ to $-1$ and $x_k$ and $x_l$ to 1. Then we also have that $a_{13} + a_{24} < -1$ and $a_{23} + a_{14} < -1$, so $a_{12} + a_{13} + a_{14} + a_{23} + a_{24} + a_{34} < 3$, which is a contradiction since that sum must be exactly $-3$ by the SDP constraint. Hence $c \geq 1$.}

The second observation about Table 4 is that the LP value of the icosahedron is not 1. It does not give quite as low an LP value as the three equally spaced vectors do, but it is much lower than the other Platonic solids. Furthermore, the matrix for

<table>
<thead>
<tr>
<th>Solid</th>
<th>$n$</th>
<th>LP Value</th>
<th>Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetrahedron</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Octahedron</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cube</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Icosahedron</td>
<td>12</td>
<td>0.672017</td>
<td>1.488057</td>
</tr>
</tbody>
</table>

Table 4: Platonic Solids

quadratic program. We formally show why this is true for the tetrahedron; proofs of the gap can for the octahedron and cube can be obtained by similar arguments. We leave the exact proofs as an exercise for the reader.
the icosahedron does not follow the pattern we have observed in every other case, namely there being only three nonzero values. In fact, every possible matrix entry is nonzero, not just some three that would indicate it picking out three vectors. At this point we are not sure why the icosahedron should be an exception to the rule, but it is.

The next natural question is whether the icosahedron is unique in this behavior. To test this, we tried changing the vectors slightly so that they almost formed an icosahedron but not quite. We found that, depending on the change, the resulting matrix was almost like the matrix for the icosahedron. Slightly perturbing any given vector resulted in a matrix with all zero entries in the row of the vector that had been perturbed but nonzero entries everywhere else, unless the vector was changed into one of the standard basis vectors, in which case the entire matrix reverted back to the usual behavior of only three nonzero values. But a few of the vectors could change significantly and it would still have many nonzero values in the rows other than the ones corresponding to those vectors.

So there is some class of vector sets that are similar to the icosahedron that have different behavior in the resulting matrices from almost every other case. We do not know exactly how big this class is, but it seems to be fairly large in that even fairly drastic changes to the icosahedron still lie in this class. At this point we don’t know why the icosahedron and its ”neighboring” class have this behavior, but it still does not seem to result in any gaps larger than 1.5.

6 Theoretical Results

We were not able to prove much about this problem, which is a disappointment and certainly an area where more progress could be made. However, inspired by our experimental results, Charikar and Makarychev did come up with a proof that the
gap in any given dimension is bounded by a constant [3]:

**Theorem 3.** The gap of the semidefinite relaxation is at most \(d\), where \(d\) is the dimension of the space the vectors lie in.

**Proof.** Let \(\{u_i\}\) be a set of \(d\)-dimensional vectors, and let \(k\) be picked randomly and uniformly from \(\{1, 2, \ldots, d\}\). For each \(i\), let \(X_i\) equal \(u_i^k\), the \(k\)th coordinate of \(u_i\). Then for any \(z \in \{1, 2, \ldots, d\}\), the probability that \(k = z\) is \(1/d\), in which case the value of \(X_i X_j\) is \(u_i^z u_j^z\). So \(E[X_i X_j] = \sum_{k=1}^{d} \frac{u_i^k u_j^k}{d} = (u_i \cdot u_j)/d\). Now let \(Y_i\) be defined as follows: \(Y_i = 1\) with probability \((1 + X_i)/2\) and \(Y_i = -1\) with probability \((1 - X_i)/2\). We have that \(E[Y_i Y_j] = \frac{1 + X_i}{2} \cdot \frac{1 + X_j}{2} + \frac{1 - X_i}{2} \cdot \frac{1 - X_j}{2} - \left( \frac{1 + X_i}{2} \cdot \frac{1 - X_j}{2} + \frac{1 - X_i}{2} \cdot \frac{1 + X_j}{2} \right) = X_i X_j\), and since \(E[X_i X_j] = \frac{u_i \cdot u_j}{d}\) we have a \(d\)-approximation. Thus the gap of the SDP is at most \(d\). \(\square\)

7 Follow-up Work

There is room for a lot more work on the theory behind this problem. Our experiments in \(\mathbb{R}^d\) for small \(d\) lead us to believe that there is a constant gap between the semidefinite program and the quadratic program, but we were only able to prove that the gap is bounded by the dimension. While proving a general constant bound might be a little ambitious, it would be nice just to get a constant bound of less than \(d\) in low dimensions. Namely, since we never observed any gap of more than 1.5 in two dimensions and we did a very thorough search, we would like to prove that 1.5 is a bound on the gap in \(\mathbb{R}^2\).

Our technique could also be applied to equations (3) and (4) in an attempt to improve the lower bound on Grothendieck’s constant. Random vectors might not help much, but at the very least it would be interesting to see how close the gap obtained from equally spaced 2 dimensional vectors and the Platonic solids is to Grothendieck’s constant. Also, almost all of the literature refers to \(\pi/2\) as the best
lower bound on Grothendieck’s constant, but there is an unpublished manuscript by James Reeds that gives 1.6769... as a lower bound [7]. He uses a version of Grothendieck’s inequality that is phrased differently than ours, but it is the same inequality so theoretically there is some matrix $A$ that results in that large a gap. It would be interesting to try to adapt his proof to our way of phrasing the inequality and both check that he is right and look for an adaptation that might be better.

8 Conclusion

MaxQP is an interesting quadratic program with an important application in correlation clustering, so we are interested in trying to approximate it using its canonical semidefinite relaxation. Charikar and Wirth have designed a $\Omega(1/\log n)$ approximation algorithm that uses the semidefinite relaxation, but obviously a constant factor approximation would be better. In an attempt to show that the semidefinite relaxation does not lead to a constant factor approximation we tried to find $A$ matrices that resulted in large gaps between the semidefinite program and the quadratic program. We did this by designing and solving a certain linear program that, given a vector solution to the semidefinite program, found the matrix that resulted in the largest gap with the quadratic program. This approach took exponential time, but it allowed us to approach the problem from a perspective that we had some intuition for rather than trying to blindly guess matrices that result in large gaps.

We found two configurations of three vectors that result in a gap of 1.5, which was the largest gap we found in either two or three dimensions. We tried equally spaced vectors in 2 dimensions, the vertices of the platonic solids in three dimensions, and random vectors in both 2 and 3 dimensions, but we never saw any gap larger than 1.5. This led us to conjecture that 1.5 is a bound on the gap in low dimensions, which unfortunately we were unable to prove. Since Grothendieck’s inequality applies
to a special case of MAXQP we know that in general the gap can be at least $\pi/2$, but achieving $\pi/2$ requires many dimensions and many vectors. The best theoretical result we were able to find was that the gap is bounded by the dimension, which means that at least in any given dimension the gap is bounded by a constant. Unfortunately this project has to end now, but there is significant room for improvement, especially in the theory behind this problem.

References


