

# Chapter 1

## Introduction: MAXCUT Via Semidefinite Programming

Semidefinite programming is considered among the most powerful tools in the theory and practice of approximation algorithms. We begin our exposition with the Goemans–Williamson algorithm for the MAXCUT problem (i.e., the problem of computing an edge cut with the maximum possible number of edges in a given graph). This is the first approximation algorithm (from 1995) based on semidefinite programming and it still belongs among the simplest and most impressive results in this area.

However, it should be said that semidefinite programming entered the field of combinatorial optimization considerably earlier, through a fundamental 1979 paper of Lovász [Lov79], in which he introduced the *theta function* of a graph. This is a somewhat more advanced concept, which we will encounter later on.

In this chapter we focus on the Goemans–Williamson algorithm, while semidefinite programming is used as a black box. In the next chapter we will start discussing it in more detail.

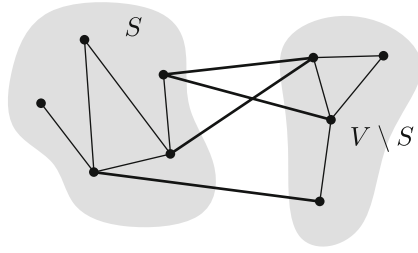
### 1.1 The MAXCUT Problem

MAXCUT is the following computational problem: We are given a graph  $G = (V, E)$  as the input, and we want to find a partition of the vertex set into two subsets,  $S$  and its complement  $V \setminus S$ , such that the number of edges going between  $S$  and  $V \setminus S$  is maximized.

More formally, we define a *cut* in a graph  $G = (V, E)$  as a pair  $(S, V \setminus S)$ , where  $S \subseteq V$ . The *edge set* of the cut  $(S, V \setminus S)$  is

$$E(S, V \setminus S) = \{e \in E : |e \cap S| = |e \cap (V \setminus S)| = 1\}$$

(see Fig. 1.1), and the *size* of this cut is  $|E(S, V \setminus S)|$ , i.e., the number of edges. We also say that the cut is *induced* by  $S$ .



**Fig. 1.1** The cut edges (*bold*) induced by a cut  $(S, V \setminus S)$

The decision version of the MAXCUT problem (given  $G$  and  $k \in \mathbb{N}$ , is there a cut of size at least  $k$ ?) was shown to be NP-complete by Garey et al. [GJS76]. The above optimization version is consequently NP-hard.

## 1.2 Approximation Algorithms

Let us consider an optimization problem  $\mathcal{P}$  (typically, but not necessarily, we will consider NP-hard problems). An *approximation algorithm* for  $\mathcal{P}$  is a *polynomial-time algorithm* that computes a solution with some guaranteed quality for *every instance* of the problem. Here is a reasonably formal definition, formulated for maximization problems.

A maximization problem consists of a set  $\mathcal{I}$  of *instances*. Every instance  $I \in \mathcal{I}$  comes with a set  $F(I)$  of *feasible solutions* (sometimes also called *admissible solutions*), and every  $s \in F(I)$  in turn has a nonnegative real value  $\omega(s) \geq 0$  associated with it. We also define

$$\text{Opt}(I) = \sup_{s \in F(I)} \omega(s) \in \mathbb{R}_+ \cup \{-\infty, \infty\}$$

to be the optimum value of the instance. Value  $-\infty$  occurs if  $F(I) = \emptyset$ , while  $\text{Opt}(I) = \infty$  means that there are feasible solutions of arbitrarily large value. To simplify the presentation, let us restrict our attention to problems where  $\text{Opt}(I)$  is finite for all  $I$ .

The MAXCUT problem immediately fits into this setting. The instances are graphs, feasible solutions are subsets of vertices, and the value of a subset is the size of the cut induced by it.

**1.2.1 Definition.** Let  $\mathcal{P}$  be a maximization problem with set of instances  $\mathcal{I}$ , and let  $\mathcal{A}$  be an algorithm that returns, for every instance  $I \in \mathcal{I}$ , a feasible solution  $\mathcal{A}(I) \in F(I)$ . Furthermore, let  $\delta: \mathbb{N} \rightarrow \mathbb{R}_+$  be a function.

We say that  $\mathcal{A}$  is a  $\delta$ -approximation algorithm for  $\mathcal{P}$  if the following two conditions hold.

- (i) There exists a polynomial  $p$  such that for all  $I \in \mathcal{I}$ , the runtime of  $\mathcal{A}$  on the instance  $I$  is bounded by  $p(|I|)$ , where  $|I|$  is the encoding size of instance  $I$ .
- (ii) For all instances  $I \in \mathcal{I}$ ,  $\omega(\mathcal{A}(I)) \geq \delta(|I|) \cdot \text{Opt}(I)$ .

Encoding size is not a mathematically precise notion; what we mean is the following: For any given problem, we fix a reasonable “file format” in which we feed problem instances to the algorithm. For a graph problem such as MAXCUT, the format could be the number of vertices  $n$ , followed by a list of pairs of the form  $(i, j)$  with  $1 \leq i < j \leq n$  that describe the edges. The encoding size of an instance can then be defined as the number of characters that are needed to write down the instance in the chosen format. Due to the fact that we allow runtime  $p(|I|)$ , where  $p$  is any polynomial, the precise format usually does not matter, and it is “reasonable” for every natural number  $k$  to be written down with  $O(\log k)$  characters.

An interesting special case occurs when  $\delta$  is a constant function. For  $c \in \mathbb{R}$ , a  $c$ -approximation algorithm is a  $\delta$ -approximation algorithm with  $\delta \equiv c$ . Clearly,  $c \leq 1$  must hold, and the closer  $c$  is to 1, the better the approximation.

We can smoothly extend the definition to randomized algorithms (algorithms that may use internal coin flips to guide their decisions). A randomized  $\delta$ -approximation algorithm must have *expected* polynomial runtime and must satisfy

$$\mathbf{E}[\omega(\mathcal{A}(I))] \geq \delta(|I|) \cdot \text{Opt}(I) \quad \text{for all } I \in \mathcal{I}.$$

For randomized algorithms,  $\omega(\mathcal{A}(I))$  is a random variable, and we require that its *expectation* be a good approximation of the true optimum value.

For minimization problems, we replace  $\sup$  by  $\inf$  in the definition of  $\text{Opt}(I)$  and we require that  $\omega(\mathcal{A}(I)) \leq \delta(|I|)\text{Opt}(I)$  for all  $I \in \mathcal{I}$ . This leads to  $c$ -approximation algorithms with  $c \geq 1$ .

## What Is Polynomial Time?

In the context of complexity theory, an algorithm is formally a Turing machine, and its runtime is obtained by counting the elementary operations (head movements), depending on the number of bits used to encode the problem on the input tape. This model of computation is also called the *bit model*.

The bit model is not very practical, and often the *real RAM* model, also called the *unit cost* model, is used instead.

The real RAM is a hypothetical computer, each of its memory cells capable of storing an arbitrary real number, including irrational ones like  $\sqrt{2}$  or  $\pi$ .

Moreover, the model assumes that arithmetic operations on real numbers (including computations of square roots, trigonometric functions, random numbers, etc.) take constant time. The model is motivated by actual computers that approximate the real numbers by floating-point numbers with fixed precision.

The real RAM is a very convenient model, since it frees us from thinking about how to encode a real number, and what the resulting encoding size is. On the downside, the real RAM model is not always compatible with the Turing machine model. It can happen that we have a polynomial-time algorithm in the real RAM model, but when we translate it to a Turing machine, it becomes exponential.

For example, Gaussian elimination, one of the simplest algorithms in linear algebra, is not a polynomial-time algorithm in the Turing machine model if a naive implementation is used [GLS88, Sect. 1.4]. The reason is that in the naive implementation, intermediate results may require exponentially many bits.

Vice versa, a polynomial-time Turing machine may not be transferable to a polynomial-time real RAM algorithm. Indeed, the runtime of the Turing machine may tend to infinity with the encoding size of the input numbers, in which case there is no bound at all for the runtime that depends only on the *number* of input numbers.

In many cases, however, it *is* possible to implement a polynomial-time real RAM algorithm in such a way that all intermediate results have encoding lengths that are polynomial in the encoding lengths of the input numbers. In this case we also get a polynomial-time algorithm in the Turing machine model. For example, in the real RAM model, Gaussian elimination is an  $O(n^3)$  algorithm for solving  $n \times n$  linear equation systems. Using appropriate representations, it can be guaranteed that all intermediate results have bit lengths that are also polynomial in  $n$  [GLS88, Sect. 1.4], and we obtain that Gaussian elimination is a polynomial-time method also in the Turing machine model.

We will occasionally run into real RAM vs. Turing machine issues, and whenever we do so, we will try to be careful in sorting them out.

### 1.3 A Randomized 0.5-Approximation Algorithm for MAXCUT

To illustrate previous definitions, let us describe a concrete (randomized) approximation algorithm **RandomizedMaxCut** for the MAXCUT problem.

Given an instance  $G = (V, E)$ , the algorithm picks  $S$  as a random subset of  $V$ , where each vertex  $v \in V$  is included in  $S$  with probability  $1/2$ , independent of all other vertices.

In a way this algorithm is stupid, since it never even looks at the edges. Still, we can prove the following result:

**1.3.1 Theorem.** *Algorithm RandomizedMaxCut is a randomized 0.5-approximation algorithm for the MAXCUT problem.*

**Proof.** It is clear that the algorithm runs in polynomial time. The value  $\omega(\text{RandomizedMaxCut}(G))$  is the size of the cut (number of cut edges) generated by the algorithm (a random variable). Now we compute

$$\begin{aligned} \mathbf{E}[\omega(\text{RandomizedMaxCut}(G))] &= \mathbf{E}[|E(S, V \setminus S)|] \\ &= \sum_{e \in E} \text{Prob}[e \in E(S, V \setminus S)] \\ &= \sum_{e \in E} \frac{1}{2} = \frac{1}{2}|E| \geq \frac{1}{2}\text{Opt}(G). \end{aligned}$$

Indeed,  $e \in E(S, V \setminus S)$  if and only if exactly one of the two endpoints of  $e$  ends up in  $S$ , and this has probability exactly  $\frac{1}{2}$ .  $\square$

The main trick in this simple proof is to split the complicated-looking quantity  $|E(S, V \setminus S)|$  into the contributions of individual edges; then we can use the linearity of expectation and account for the expected contribution of each edge separately. We will also see this trick in the analysis of the Goemans–Williamson algorithm.

It is possible to “derandomize” this algorithm and come up with a deterministic 0.5-approximation algorithm for MAXCUT (see Exercise 1.1). Minor improvements are possible. For example, there exists a  $0.5(1 + 1/m)$  approximation algorithm, where  $m = |E|$ ; see Exercise 1.2.

But until 1994, no  $c$ -approximation algorithm was found for any factor  $c > 0.5$ .

## 1.4 The Goemans–Williamson Algorithm

Here we describe the **GWMaxCut** algorithm, a 0.878-approximation algorithm for the MAXCUT problem, based on semidefinite programming. In a nutshell, a semidefinite program (SDP) is the problem of maximizing a linear function in  $n^2$  variables  $x_{ij}$ ,  $i, j = 1, 2, \dots, n$ , subject to linear equality constraints *and* the requirement that the variables form a positive semidefinite matrix  $X$ . We write  $X \succeq 0$  for “ $X$  is positive semidefinite.”

For this chapter we assume that a semidefinite program can be solved in polynomial time, up to any desired accuracy  $\varepsilon$ , and under suitable conditions that are satisfied in our case. We refrain from specifying this further here; a detailed statement appears in Chap. 2. For now, let us continue with the

Goemans–Williamson approximation algorithm, using semidefinite programming as a black box.

We start by formulating the MAXCUT problem as a constrained optimization problem (which we will then turn into a semidefinite program). For the whole section, let us fix the graph  $G = (V, E)$ , where we assume that  $V = \{1, 2, \dots, n\}$  (this will be used often and in many places). Then we introduce variables  $z_1, z_2, \dots, z_n \in \{-1, 1\}$ . Any assignment of values from  $\{-1, 1\}$  to these variables encodes a cut  $(S, V \setminus S)$ , where  $S = \{i \in V : z_i = 1\}$ . The term

$$\frac{1 - z_i z_j}{2}$$

is exactly the contribution of the edge  $\{i, j\}$  to the size of the above cut. Indeed, if  $\{i, j\}$  is not a cut edge, we have  $z_i z_j = 1$ , and the contribution is 0. If  $\{i, j\}$  is a cut edge, then  $z_i z_j = -1$ , and the contribution is 1. It follows that we can reformulate the MAXCUT problem as follows.

$$\begin{aligned} & \text{Maximize} && \sum_{\{i,j\} \in E} \frac{1 - z_i z_j}{2} \\ & \text{subject to} && z_i \in \{-1, 1\}, \quad i = 1, \dots, n. \end{aligned} \tag{1.1}$$

The optimum value (or simply value) of this program is  $\text{Opt}(G)$ , the size of a maximum cut. Thus, in view of the NP-completeness of MAXCUT, we cannot expect to solve this optimization problem exactly in polynomial time.

## Semidefinite Programming Relaxation

Here is the crucial step: We write down a semidefinite program whose value is an *upper bound* for the value  $\text{Opt}(G)$  of (1.1). To get it, we first replace each real variable  $z_i$  with a vector variable  $\mathbf{u}_i \in S^{n-1} = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\| = 1\}$ , the  $(n - 1)$ -dimensional unit sphere:

$$\begin{aligned} & \text{Maximize} && \sum_{\{i,j\} \in E} \frac{1 - \mathbf{u}_i^T \mathbf{u}_j}{2} \\ & \text{subject to} && \mathbf{u}_i \in S^{n-1}, \quad i = 1, 2, \dots, n. \end{aligned} \tag{1.2}$$

This is called a *vector program* since the unknowns are vectors.<sup>1</sup>

From the fact that the set  $\{-1, 1\}$  can be embedded into  $S^{n-1}$  via the mapping  $x \mapsto (0, 0, \dots, 0, x)$ , we derive the following important property: for every solution of (1.1), there is a corresponding solution of (1.2) with the same value. This means that the program (1.2) is a *relaxation* of (1.1), a program with “more” solutions, and it therefore has value *at least*  $\text{Opt}(G)$ . It is also

---

<sup>1</sup> We consider vectors in  $\mathbb{R}^n$  as column vectors, i.e., as  $n \times 1$  matrices. The superscript  $T$  denotes matrix transposition, and thus  $\mathbf{u}_i^T \mathbf{u}_j$  is the standard scalar product of  $\mathbf{u}_i$  and  $\mathbf{u}_j$ .

clear that this value is still finite, since  $\mathbf{u}_i^T \mathbf{u}_j$  is bounded from below by  $-1$  for all  $i, j$ .

Vectors may look more complicated than real numbers, and so it is quite counterintuitive that (1.2) should be any easier than (1.1). But semidefinite programming will allow us to solve the vector program efficiently, to any desired accuracy!

To see this, we perform yet another variable substitution, namely,  $x_{ij} = \mathbf{u}_i^T \mathbf{u}_j$ . This brings (1.2) into the form of a semidefinite program:

$$\begin{aligned} & \text{Maximize} && \sum_{\{i,j\} \in E} \frac{1-x_{ij}}{2} \\ & \text{subject to} && x_{ii} = 1, \quad i = 1, 2, \dots, n, \\ & && X \succeq 0. \end{aligned} \tag{1.3}$$

To see that (1.3) is equivalent to (1.2), we first note that if  $\mathbf{u}_1, \dots, \mathbf{u}_n$  constitute a feasible solution to (1.2), i.e., they are unit vectors, then with  $x_{ij} = \mathbf{u}_i^T \mathbf{u}_j$ , we have

$$X = U^T U,$$

where the matrix  $U$  has the columns  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n$ . Such a matrix  $X$  is positive semidefinite, and  $x_{ii} = 1$  follows from  $\mathbf{u}_i \in S^{n-1}$  for all  $i$ . So  $X$  is a feasible solution of (1.3) with the same value.

Slightly more interesting is the opposite direction, namely, that every feasible solution  $X$  of (1.3) yields a solution of (1.2), with the same value. For this, one needs to know that every positive semidefinite matrix  $X$  can be written as the product  $X = U^T U$  (see Sect. 2.2). Thus, if  $X$  is a feasible solution of (1.3), the columns of such a matrix  $U$  provide a feasible solution of (1.2); due to the constraints  $x_{ii} = 1$ , they are actually unit vectors.

Thus, the semidefinite program (1.3) has the same finite value  $\text{SDP}(G) \geq \text{Opt}(G)$  as (1.2). So we can find in polynomial time a matrix  $X^* \succeq 0$  with  $x_{ii}^* = 1$  for all  $i$  and with

$$\sum_{\{i,j\} \in E} \frac{1-x_{ij}^*}{2} \geq \text{SDP}(G) - \varepsilon,$$

for every  $\varepsilon > 0$ .

We can also compute in polynomial time a matrix  $U^*$  such that  $X^* = (U^*)^T U^*$ , up to a tiny error. This is a *Cholesky factorization* of  $X^*$ ; see Sect. 2.3. The tiny error can be dealt with at the cost of slightly adapting  $\varepsilon$ . So let us assume that the factorization is exact.

Then the columns  $\mathbf{u}_1^*, \mathbf{u}_2^*, \dots, \mathbf{u}_n^*$  of  $U^*$  are unit vectors that form an almost-optimal solution of the vector program (1.2):

$$\sum_{\{i,j\} \in E} \frac{1 - \mathbf{u}_i^{*T} \mathbf{u}_j^*}{2} \geq \text{SDP}(G) - \varepsilon \geq \text{Opt}(G) - \varepsilon. \tag{1.4}$$

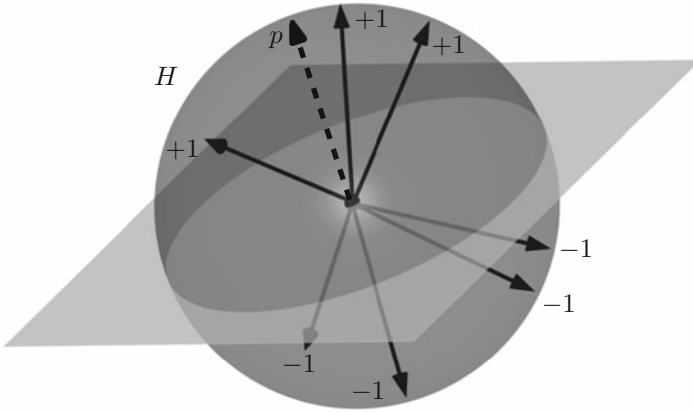
## Rounding the Vector Solution

Let us recall that what we actually want to solve is program (1.1), where the  $n$  variables  $z_i$  are elements of  $S^0 = \{-1, 1\}$  and thus determine a cut  $(S, V \setminus S)$  via  $S := \{i \in V : z_i = 1\}$ .

What we have is an almost optimal solution of the relaxed program (1.2) where the  $n$  vector variables are elements of  $S^{n-1}$ . We therefore need a way of mapping  $S^{n-1}$  back to  $S^0$  in such a way that we do not “lose too much.” Here is how we do it. Choose  $\mathbf{p} \in S^{n-1}$  and consider the mapping

$$\mathbf{u} \mapsto \begin{cases} 1 & \text{if } \mathbf{p}^T \mathbf{u} \geq 0, \\ -1 & \text{otherwise.} \end{cases} \quad (1.5)$$

The geometric picture is the following:  $\mathbf{p}$  partitions  $S^{n-1}$  into a closed hemisphere  $H = \{\mathbf{u} \in S^{n-1} : \mathbf{p}^T \mathbf{u} \geq 0\}$  and its complement. Vectors in  $H$  are mapped to 1, while vectors in the complement map to  $-1$ ; see Fig. 1.2.



**Fig. 1.2** Rounding vectors in  $S^{n-1}$  to  $\{-1, 1\}$  through a vector  $\mathbf{p} \in S^{n-1}$

It remains to choose  $\mathbf{p}$ , and we will do this *randomly* (we speak of *randomized rounding*). More precisely, we sample  $\mathbf{p}$  uniformly at random from  $S^{n-1}$ . To understand why this is a good thing, we need to do the computations, but here is the intuition. We certainly want that a pair of vectors  $\mathbf{u}_i^*$  and  $\mathbf{u}_j^*$  with large value

$$\frac{1 - \mathbf{u}_i^{*T} \mathbf{u}_j^*}{2}$$

is more likely to yield a cut edge  $\{i, j\}$  than a pair with a small value. Since the contribution grows with the angle between  $\mathbf{u}_i^*$  and  $\mathbf{u}_j^*$ , our mapping to



$\{-1, +1\}$  should be such that pairs with large angles are more likely to be mapped to different values than pairs with small angles.

As we will see, this is how the function (1.5) with randomly chosen  $\mathbf{p}$  is going to behave.

**1.4.1 Lemma.** *Let  $\mathbf{u}, \mathbf{u}' \in S^{n-1}$ . The probability that (1.5) maps  $\mathbf{u}$  and  $\mathbf{u}'$  to different values is*

$$\frac{1}{\pi} \arccos \mathbf{u}^T \mathbf{u}'.$$

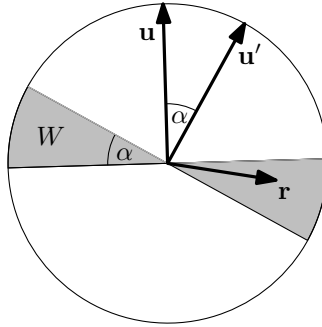
**Proof.** Let  $\alpha \in [0, \pi]$  be the angle between the unit vectors  $\mathbf{u}$  and  $\mathbf{u}'$ . By the law of cosines, we have

$$\cos(\alpha) = \mathbf{u}^T \mathbf{u}' \in [-1, 1],$$

or, in other words,

$$\alpha = \arccos \mathbf{u}^T \mathbf{u}' \in [0, \pi].$$

If  $\alpha = 0$  or  $\alpha = \pi$ , meaning that  $\mathbf{u} \in \{\mathbf{u}', -\mathbf{u}'\}$ , the statement trivially holds. Otherwise, let us consider the linear span  $L$  of  $\mathbf{u}$  and  $\mathbf{u}'$ , which is a two-dimensional subspace of  $\mathbb{R}^n$ . With  $\mathbf{r}$  the projection of  $\mathbf{p}$  to that subspace, we have  $\mathbf{p}^T \mathbf{u} = \mathbf{r}^T \mathbf{u}$  and  $\mathbf{p}^T \mathbf{u}' = \mathbf{r}^T \mathbf{u}'$ . This means that  $\mathbf{u}$  and  $\mathbf{u}'$  map to different values if and only if  $\mathbf{r}$  lies in a “half-open double wedge”  $W$  of opening angle  $\alpha$ ; see Fig. 1.3.



**Fig. 1.3** Randomly rounding vectors:  $\mathbf{u}$  and  $\mathbf{u}'$  map to different values if and only if the projection  $\mathbf{r}$  of  $\mathbf{p}$  to the linear span of  $\mathbf{u}$  and  $\mathbf{u}'$  lies in the shaded region  $W$  (“half-open double wedge”)

Since  $\mathbf{p}$  is uniformly distributed in  $S^{n-1}$ , the direction of  $\mathbf{r}$  is uniformly distributed in  $[0, 2\pi]$ . Therefore, the probability of  $\mathbf{r}$  falling into the double wedge is the fraction of angles covered by the double wedge, and this is  $\alpha/\pi$ .  $\square$

## Getting the Bound

Let us see what we have achieved. If we round as above, the expected number of edges in the resulting cut equals

$$\sum_{\{i,j\} \in E} \frac{\arccos \mathbf{u}_i^{*T} \mathbf{u}_j^*}{\pi}.$$

Indeed, we are summing the probability that an edge  $\{i, j\}$  becomes a cut edge, as in Lemma 1.4.1, over all edges  $\{i, j\}$ . The trouble is that we do not know much about this sum. But we *do* know that

$$\sum_{\{i,j\} \in E} \frac{1 - \mathbf{u}_i^{*T} \mathbf{u}_j^*}{2} \geq \text{Opt}(G) - \varepsilon;$$

see (1.4). The following technical lemma allows us to compare the two sums termwise.

**1.4.2 Lemma.** *For all  $z \in [-1, 1]$ ,*

$$\frac{\arccos(z)}{\pi} \geq 0.8785672 \frac{1 - z}{2}.$$

The constant appearing in this lemma is the solution to a problem that seems to come from a crazy calculus teacher: what is the minimum of the function

$$f(z) = \frac{2 \arccos(z)}{\pi(1 - z)}$$

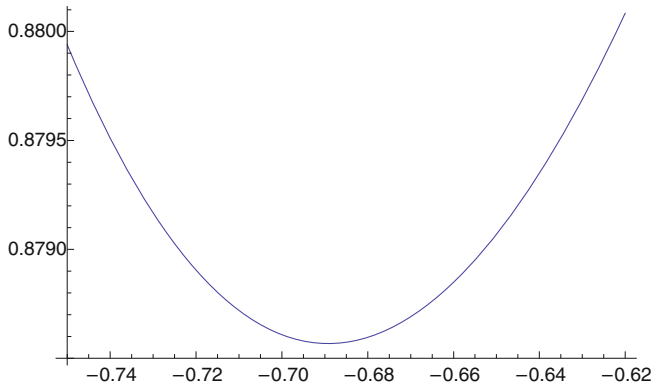
over the interval  $[-1, 1]$ ?

**Proof.** The plot in Fig. 1.4 below depicts the function  $f(z)$ ; the minimum occurs at the (unique) value  $z^*$  where the derivative vanishes. Using a numeric solver, you can compute  $z^* \approx -0.68915773665$ , which yields  $f(z^*) \approx 0.87856720578 > 0.8785672$ .  $\square$

Using this lemma, we can conclude that the expected number of cut edges produced by our algorithm satisfies

$$\begin{aligned} \sum_{\{i,j\} \in E} \frac{\arccos \mathbf{u}_i^{*T} \mathbf{u}_j^*}{\pi} &\geq 0.8785672 \sum_{\{i,j\} \in E} \frac{1 - \mathbf{u}_i^{*T} \mathbf{u}_j^*}{2} \\ &\geq 0.8785672(\text{Opt}(G) - \varepsilon) \\ &\geq 0.878 \text{Opt}(G), \end{aligned}$$

provided we choose  $\varepsilon \leq 5 \cdot 10^{-4}$ .



**Fig. 1.4** The function  $f(z) = 2 \arccos(z)/\pi(1-z)$  and its minimum

Here is a summary of the Goemans–Williamson algorithm **GWMaxCut** for approximating the maximum cut in a graph  $G = (\{1, 2, \dots, n\}, E)$ .

1. Compute an almost optimal solution  $\mathbf{u}_1^*, \mathbf{u}_2^*, \dots, \mathbf{u}_n^*$  of the vector program

$$\begin{array}{ll} \text{maximize} & \sum_{\{i,j\} \in E} \frac{1 - \mathbf{u}_i^T \mathbf{u}_j}{2} \\ \text{subject to} & \mathbf{u}_i \in S^{n-1}, \quad i = 1, 2, \dots, n. \end{array}$$

This is a solution that satisfies

$$\sum_{\{i,j\} \in E} \frac{1 - \mathbf{u}_i^{*T} \mathbf{u}_j^*}{2} \geq \text{SDP}(G) - 5 \cdot 10^{-4} \geq \text{Opt}(G) - 5 \cdot 10^{-4},$$

and it can be found in polynomial time by semidefinite programming and Cholesky factorization.

2. Choose  $\mathbf{p} \in S^{n-1}$  uniformly at random, and output the cut induced by

$$S := \{i \in \{1, 2, \dots, n\} : \mathbf{p}^T \mathbf{u}_i^* \geq 0\}.$$

We have thus proved the following result.

**1.4.3 Theorem.** *Algorithm **GWMaxCut** is a randomized 0.878-approximation algorithm for the MAXCUT problem.*

**Almost optimal vs. optimal solutions.** It is customary in the literature (and we will adopt this later) to simply call an almost optimal solution of a semidefinite or a vector program an “optimal solution.” This is justified, since

for the purpose of approximation algorithms an almost optimal solution is just as good as a truly optimal solution. Under this convention, an “optimal solution” of a semidefinite or a vector program is a solution that is accurate enough in the given context.

## Exercises

**1.1** Prove that there is also a deterministic 0.5-approximation algorithm for the MAXCUT problem.

**1.2** Prove that there is a  $0.5(1 + 1/m)$ -approximation algorithm (randomized or deterministic) for the MAXCUT problem, where  $m$  is the number of edges of the given graph  $G$ .

# Chapter 2

## Semidefinite Programming

Let us start with the concept of *linear programming*. A *linear program* is the problem of maximizing (or minimizing) a linear function in  $n$  variables subject to linear equality and inequality constraints. In *equational form*, a linear program can be written as

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}. \end{array}$$

Here  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  is a vector of  $n$  variables,<sup>1</sup>  $\mathbf{c} = (c_1, c_2, \dots, c_n)$  is the objective function vector,  $\mathbf{b} = (b_1, b_2, \dots, b_m)$  is the right-hand side, and  $A \in \mathbb{R}^{m \times n}$  is the constraint matrix. The bold digit  $\mathbf{0}$  stands for the zero vector of the appropriate dimension. Vector inequalities like  $\mathbf{x} \geq \mathbf{0}$  are to be understood componentwise.

In other words, among all  $\mathbf{x} \in \mathbb{R}^n$  that satisfy the matrix equation  $A\mathbf{x} = \mathbf{b}$  and the vector inequality  $\mathbf{x} \geq \mathbf{0}$  (such  $\mathbf{x}$  are called *feasible solutions*), we are looking for an  $\mathbf{x}^*$  with the highest value  $\mathbf{c}^T \mathbf{x}^*$ .

### 2.1 From Linear to Semidefinite Programming

To get a semidefinite program, we replace the vector space  $\mathbb{R}^n$  underlying  $\mathbf{x}$  by another real vector space, namely the vector space

$$\text{SYM}_n = \{X \in \mathbb{R}^{n \times n} : x_{ij} = x_{ji}, 1 \leq i < j \leq n\}$$

of symmetric  $n \times n$  matrices, and we replace the matrix  $A$  by a linear mapping  $A: \text{SYM}_n \rightarrow \mathbb{R}^m$ .

---

<sup>1</sup> Vectors are column vectors, but in writing them explicitly, we use the  $n$ -tuple notation.

The standard scalar product  $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y}$  over  $\mathbb{R}^n$  gets replaced by the standard scalar product

$$X \bullet Y := \sum_{i=1}^n \sum_{j=1}^n x_{ij} y_{ij}$$

over  $\text{SYM}_n$ . Alternatively, we can also write  $X \bullet Y = \text{Tr}(X^T Y)$ , where for a square matrix  $M$ ,  $\text{Tr}(M)$  (the *trace* of  $M$ ) is the sum of the diagonal entries of  $M$ .

Finally, we replace the constraint  $\mathbf{x} \geq \mathbf{0}$  by the constraint

$$X \succeq 0.$$

Here  $X \succeq 0$  stands for “the matrix  $X$  is positive semidefinite.”

Next, we will explain all of this in more detail.

## 2.2 Positive Semidefinite Matrices

First we recall that a *positive semidefinite matrix* is a real matrix  $M$  that is *symmetric* (i.e.,  $M^T = M$ , and in particular,  $M$  is a square matrix) and has all eigenvalues *nonnegative*. (The condition of symmetry is all too easy to forget. Let us also recall from Linear Algebra that a symmetric real matrix has only real eigenvalues, and so the nonnegativity condition makes sense.)

Here are several equivalent characterizations.

**2.2.1 Fact.** *Let  $M \in \text{SYM}_n$ . The following statements are equivalent.*

- (i)  *$M$  is positive semidefinite, i.e., all the eigenvalues of  $M$  are nonnegative.*
- (ii)  *$\mathbf{x}^T M \mathbf{x} \geq 0$  for all  $\mathbf{x} \in \mathbb{R}^n$ .*
- (iii) *There exists a matrix  $U \in \mathbb{R}^{n \times n}$  such that  $M = U^T U$ .*

This can easily be proved using diagonalization, which is a basic tool for dealing with symmetric matrices.

Using the condition (ii), we can see that a semidefinite program as introduced earlier can be regarded as a “linear program with infinitely many constraints.” Indeed, the constraint  $X \succeq 0$  for the unknown matrix  $X$  can be replaced with the constraints  $\mathbf{a}^T X \mathbf{a} \geq 0$ ,  $\mathbf{a} \in \mathbb{R}^n$ . That is, we have infinitely many linear constraints, one for every vector  $\mathbf{a} \in \mathbb{R}^n$ .

**2.2.2 Definition.**  $\text{PSD}_n$  is the set of all positive semidefinite  $n \times n$  matrices.

A matrix  $M$  is called *positive definite* if  $\mathbf{x}^T M \mathbf{x} > 0$  for all  $\mathbf{x} \neq \mathbf{0}$ . It can be checked that the positive definite matrices form the interior of the set  $\text{PSD}_n \subseteq \text{SYM}_n$ .

## 2.3 Cholesky Factorization

In semidefinite programming we often need to compute, for a given positive semidefinite matrix  $M$ , a matrix  $U$  as in Fact 2.2.1(iii), i.e., such that  $M = U^T U$ . This is called the computation of a *Cholesky factorization*. (The definition also requires  $U$  to be upper triangular, but we don't need this.)

We present a simple explicit method, the *outer product Cholesky Factorization* [GvL96, Sect. 4.2.8], which uses  $O(n^3)$  arithmetic operations for an  $n \times n$  matrix  $M$ .

If  $M = (\alpha) \in \mathbb{R}^{1 \times 1}$ , we set  $U = (\sqrt{\alpha})$ , where  $\alpha \geq 0$  by the nonnegativity of the eigenvalues. Otherwise, since  $M$  is symmetric, we can write it as

$$M = \begin{pmatrix} \alpha & \mathbf{q}^T \\ \mathbf{q} & N \end{pmatrix}.$$

We also have  $\alpha = \mathbf{e}_1^T M \mathbf{e}_1 \geq 0$  by Fact 2.2.1(ii). Here  $\mathbf{e}_i$  denotes the  $i$ -th unit vector of the appropriate dimension.

There are two cases to consider. If  $\alpha > 0$ , we compute

$$M = \begin{pmatrix} \sqrt{\alpha} & \mathbf{0}^T \\ \frac{1}{\sqrt{\alpha}} \mathbf{q} & I_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & N - \frac{1}{\alpha} \mathbf{q} \mathbf{q}^T \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & \frac{1}{\sqrt{\alpha}} \mathbf{q}^T \\ \mathbf{0} & I_{n-1} \end{pmatrix}. \quad (2.1)$$

The matrix  $N - \frac{1}{\alpha} \mathbf{q} \mathbf{q}^T$  is again positive semidefinite (Exercise 2.2), and we can recursively compute a Cholesky factorization

$$N - \frac{1}{\alpha} \mathbf{q} \mathbf{q}^T = V^T V.$$

Elementary calculations yield that

$$U = \begin{pmatrix} \sqrt{\alpha} & \frac{1}{\sqrt{\alpha}} \mathbf{q}^T \\ \mathbf{0} & V \end{pmatrix}$$

satisfies  $M = U^T U$ , and so we have found a Cholesky factorization of  $M$ .

In the other case ( $\alpha = 0$ ), we also have  $\mathbf{q} = \mathbf{0}$  (Exercise 2.2). The matrix  $N$  is positive semidefinite (apply Fact 2.2.1(ii) with  $\mathbf{x} = (0, x_2, \dots, x_n)$ ), so we can recursively compute a matrix  $V$  satisfying  $N = V^T V$ . Setting

$$U = \begin{pmatrix} 0 & \mathbf{0}^T \\ \mathbf{0} & V \end{pmatrix}$$

then gives  $M = U^T U$ , and we are done with the outer product Cholesky factorization.

Exercise 2.3 asks you to show that the above method can be modified to check whether a given matrix  $M$  is positive semidefinite.

We note that the outer product Cholesky factorization is a polynomial-time algorithm only in the real RAM model. We can transform it into a polynomial-time Turing machine, but at the cost of giving up the exact factorization. After all, a Turing machine cannot even exactly factor the  $1 \times 1$  matrix (2), since  $\sqrt{2}$  is an irrational number that cannot be written down with finitely many bits.

The error analysis of Higham [Hig91] implies the following: when we run a modified version of the above algorithm (the modification is to base the factorization (2.1) not on  $\alpha = m_{11}$  but rather on the largest diagonal entry  $m_{jj}$ ), and when we round all intermediate results to  $O(n)$  bits (the constant chosen appropriately), then we will obtain a matrix  $U$  such that the relative error  $\|U^T U - M\|_F / \|M\|_F$  is bounded by  $2^{-n}$ . (Here  $\|M\|_F = (\sum_{i,j=1}^n m_{ij}^2)^{1/2}$  is the *Frobenius norm*.) This accuracy is sufficient for most purposes, and in particular, for the Goemans–Williamson MAXCUT algorithm of the previous chapter.

## 2.4 Semidefinite Programs

**2.4.1 Definition.** A *semidefinite program in equational form* is the following kind of optimization problem:

$$\begin{aligned} &\text{Maximize} && \sum_{i,j=1}^n c_{ij} x_{ij} \\ &\text{subject to} && \sum_{i,j=1}^n a_{ijk} x_{ij} = b_k, \quad k = 1, \dots, m, \\ &&& X \succeq 0, \end{aligned} \tag{2.2}$$

where the  $x_{ij}$ ,  $1 \leq i, j \leq n$ , are  $n^2$  variables satisfying the symmetry conditions  $x_{ji} = x_{ij}$  for all  $i, j$ , the  $c_{ij}$ ,  $a_{ijk}$  and  $b_k$  are real coefficients, and

$$X = (x_{ij})_{i,j=1}^n \in \text{SYM}_n.$$

In a more compact form, the semidefinite program in this definition can be written as

$$\begin{aligned} &\text{Maximize} && C \bullet X \\ &\text{subject to} && A_1 \bullet X = b_1 \\ &&& A_2 \bullet X = b_2 \\ &&& \vdots \\ &&& A_m \bullet X = b_m \\ &&& X \succeq 0, \end{aligned} \tag{2.3}$$



where

$$C = (c_{ij})_{i,j=1}^n$$

is the matrix expressing the objective function,<sup>2</sup> and

$$A_k = (a_{ijk})_{i,j=1}^n, \quad k = 1, 2, \dots, m.$$

(We recall the notation  $C \bullet X = \sum_{i,j=1}^n c_{ij}x_{ij}$  introduced earlier.)

We can write the system of  $m$  linear constraints  $A_1 \bullet X = b_1, \dots, A_m \bullet X = b_m$  even more compactly as

$$A(X) = \mathbf{b},$$

where  $\mathbf{b} = (b_1, \dots, b_m)$  and  $A: \text{SYM}_n \mapsto \mathbb{R}^m$  is a linear mapping. This notation will be useful especially for general considerations about semidefinite programs.

Following the linear programming case, we call the semidefinite program (2.3) *feasible* if there is some *feasible solution*, i.e., a matrix  $\tilde{X} \in \text{SYM}_n$  with  $A(\tilde{X}) = \mathbf{b}$ ,  $\tilde{X} \succeq 0$ . The *value* of a feasible semidefinite program is defined as

$$\sup\{C \bullet X : A(X) = \mathbf{b}, X \succeq 0\}, \quad (2.4)$$

which includes the possibility that the value is  $\infty$ . In this case, the program is called *unbounded*; otherwise, we speak of a *bounded* semidefinite program.

An *optimal solution* is a feasible solution  $X^*$  such that  $C \bullet X^* \geq C \bullet X$  for all feasible solutions  $X$ . Consequently, if there is an optimal solution, the value of the semidefinite program is finite, and it is attained, meaning that the supremum in (2.4) is a maximum.

**Warning:** If a semidefinite program has finite value, generally we *cannot* conclude that the value is attained! We illustrate this with an example below. For applications, this presents no problem: All known efficient algorithms for solving semidefinite programs return only *approximately optimal* solutions, and these are the ones that we rely on in applications.

Here is the example. With  $X \in \text{SYM}_2$ , let us consider the problem

$$\begin{array}{ll} \text{Maximize} & -x_{11} \\ \text{subject to} & x_{12} = 1 \\ & X \succeq 0. \end{array}$$

The feasible solutions of this semidefinite program are all positive semidefinite matrices  $X$  of the form

$$X = \begin{pmatrix} x_{11} & 1 \\ 1 & x_{22} \end{pmatrix}.$$

---

<sup>2</sup> Since  $X$  is symmetric, we may also assume that  $C$  is symmetric, without loss of generality; similarly for the matrices  $A_k$ .

It is easy to see that such a matrix is positive semidefinite if and only if  $x_{11}, x_{22} \geq 0$  and  $x_{11}x_{22} \geq 1$ . Equivalently, if  $x_{11} > 0$  and  $x_{22} \geq 1/x_{11}$ . This implies that the value of the program is 0, but there is no solution that attains this value.

## 2.5 Non-standard Form

Semidefinite programs do not always look exactly as in (2.3). Besides the constraints given by linear equations, as in (2.3), there may also be inequality constraints, and one may also need extra real variables that are not entries of the positive semidefinite matrix  $X$ . Let us indicate how such more general semidefinite programs can be converted to the standard form (2.3).

First, introducing extra nonnegative real variables  $x_1, x_2, \dots, x_k$  not appearing in  $X$  can be handled by incorporating them into the matrix. Namely, we replace  $X$  with the matrix  $X' \in \text{SYM}_{n+k}$ , of the form

$$X' = \begin{pmatrix} X & 0 & 0 & \cdots & 0 \\ 0 & x_1 & 0 & \cdots & 0 \\ 0 & 0 & x_2 & \cdots & 0 \\ & & & \ddots & \\ 0 & 0 & 0 & & x_k \\ 0 & 0 & 0 & \cdots & x_k \end{pmatrix}.$$

We note that the zero entries really mean adding equality constraints to the standard form (2.3). We have  $X' \succeq 0$  if and only if  $X \succeq 0$  and  $x_1, x_2, \dots, x_k \geq 0$ .

To get rid of inequalities, we can add nonnegative slack variables, just as in linear programming. Thus, an inequality constraint  $x_{23} + 5x_{15} \leq 22$  is replaced with the equality constraint  $x_{23} + 5x_{15} + y = 22$ , where  $y$  is an extra nonnegative real variable that does not occur anywhere else. Finally, an unrestricted real variable  $x_i$  (allowed to attain both positive and negative values) is replaced by the difference  $x'_i - x''_i$ , where  $x'_i$  and  $x''_i$  are two new *nonnegative* real variables.

By these steps, a non-standard semidefinite program assumes the form of a standard program (2.3) over  $\text{SYM}_{n+k}$  for some  $k$ .

## 2.6 The Complexity of Solving Semidefinite Programs

In Chap.1 we claimed that under suitable conditions, satisfied in the Goemans–Williamson MAXCUT algorithm and many other applications, a semidefinite program can be solved in polynomial time up to any desired accuracy  $\varepsilon$ . Here we want to make this claim precise.

In order to claim that a semidefinite program is (approximately) solvable in polynomial time, we need to assume that it is “well-behaved” in some sense. Namely, we need that the feasible solutions cannot be too large: we will assume that together with the input semidefinite program, we also obtain an integer  $R$  bounding the Frobenius norm of all feasible matrices  $X$ .

We will be able to claim polynomial-time approximate solvability only in the case where  $R$  has polynomially many digits. As we will see later, one can construct examples of semidefinite programs where this fails and one needs exponentially many bits in order to write down any feasible solution.

**What the ellipsoid method can do.** The strongest known *theoretical* result on solvability of semidefinite programs follows from the *ellipsoid method* (a standard reference is Grötschel et al. [GLS88]). The ellipsoid method is a general algorithm for maximizing (or minimizing) a given linear function over a given *full-dimensional* convex set  $C$ .<sup>3</sup>

In our case, we would like to apply the ellipsoid method to the set  $C \subseteq \text{SYM}_n$  of all feasible solutions of the considered semidefinite program.

This set  $C$  is convex but not full-dimensional, due to the linear equality constraints in the semidefinite program. But since the affine solution space  $L$  of the set of linear equalities can be computed in polynomial time through Gaussian elimination, we may restrict  $C$  to this space and then we have a full-dimensional convex set. Technically, this can either be done through an explicit coordinate transformation, or dealt with implicitly (we will do the latter).

The ellipsoid method further requires that  $C$  should be enclosed in a ball of radius  $R$  and it should be given by a polynomial-time *weak separation oracle* [GLS88, Sect. 2.1]. In our case, this means that for a given symmetric matrix  $X$  that satisfies all the equality constraints, we can either certify that it is “almost” feasible (i.e., has small distance to the set  $\text{PSD}_n$ ), or find a hyperplane that almost separates  $X$  from  $C$ . Polynomial time is w.r.t. the encoding length of  $X$ , the bound  $R$ , and the amount of “almost.”

It turns out that a polynomial-time weak separation oracle is provided by the Cholesky factorization algorithm (see Sect. 2.3 and Exercise 2.3). The only twist is that we need to perform the decomposition “within”  $L$ , i.e., for a suitably transformed matrix  $X'$  of lower dimension.

Indeed, if the approximate Cholesky factorization goes through,  $X'$  is an almost positive semidefinite matrix, since it is close (in absolute terms) to a positive semidefinite matrix  $U^T U$ . The outer product Cholesky factorization guarantees a small *relative* error, but this can be turned into a small absolute error by computing with  $O(\log R)$  more bits.

Similarly, if the approximate Cholesky factorization fails at some point, we can reconstruct a vector  $\mathbf{v}$  (by solving a system of linear equations) such that  $\mathbf{v}^T X' \mathbf{v}$  is negative or at least very close to zero; this gives us an almost separating hyperplane.

---

<sup>3</sup> A set  $C$  is convex if for all  $\mathbf{x}, \mathbf{y} \in C$  and  $\lambda \in [0, 1]$ , we also have  $(1 - \lambda)\mathbf{x} + \lambda\mathbf{y} \in C$ .

To state the result, we consider a semidefinite program (P) in the form

$$\begin{aligned} & \text{Maximize} && C \bullet X \\ & \text{subject to} && A_1 \bullet X = b_1 \\ & && A_2 \bullet X = b_2 \\ & && \vdots \\ & && A_m \bullet X = b_m \\ & && X \succeq 0. \end{aligned}$$

Let  $L := \{X \in \text{SYM}_n : A_i \bullet X = b_i, i = 1, 2, \dots, m\}$  be the affine subspace of matrices satisfying all the equality constraints. Let us say that a matrix  $X \in \text{SYM}_n$  is an  $\varepsilon$ -deep feasible solution of (P) if all matrices  $Y \in L$  of (Frobenius) distance at most  $\varepsilon$  from  $X$  are feasible solutions of (P).

Now we can state a precise result about the solvability of semidefinite programs, which follows from general results about the ellipsoid method [GLS88, Theorem 3.2.1. and Corollary 4.2.7].

**2.6.1 Theorem.** *Let us assume that the semidefinite program (P) has rational coefficients, let  $R$  be an explicitly given bound on the maximum Frobenius norm  $\|X\|_F$  of all feasible solutions of (P), and let  $\varepsilon > 0$  be a rational number.*

*Let us put  $v_{\text{deep}} := \sup\{C \bullet X : X \text{ an } \varepsilon\text{-deep feasible solution of (P)}\}$ . There is an algorithm, with runtime polynomial in the (binary) encoding sizes of the input numbers and in  $\log(R/\varepsilon)$ , that produces one of the following two outputs.*

- (a) *A matrix  $X^* \in L$  (i.e., satisfying all equality constraints) such that  $\|X^* - X\|_F \leq \varepsilon$  for some feasible solution  $X$ , and with  $C \bullet X^* \geq v_{\text{deep}} - \varepsilon$ .*
- (b) *A certificate that (P) has no  $\varepsilon$ -deep feasible solutions. This certificate has the form of an ellipsoid  $E \subset L$  that, on the one hand, is guaranteed to contain all feasible solutions, and on the other hand, has volume so small that it cannot contain an  $\varepsilon$ -ball.*

One has to be careful here: This theorem does not yet imply the informal claim made in Chap. 1. It does so if  $R$  is not too large. Unfortunately,  $R$  may have to be very large in general, namely doubly-exponential in  $n$ , the matrix size; see the pathological example below. In such a case, the bound of Theorem 2.6.1 is exponential!

What saves us in the applications is that  $R$  is usually small. In the MAX-CUT application, for example, all entries of a feasible solution  $X$  are inner products of unit vectors. Hence the entries are in  $[-1, 1]$ , and thus  $\|X\|_F \leq n$ .

**A glance at other algorithms.** First we want to point out that the ellipsoid method is the *only known* method that provably yields polynomial runtime

in the Turing machine model, at least under suitable and fairly general conditions such as a good bound  $R$ .

On the other hand, the practical performance of the ellipsoid method is poor, and completely different algorithms have made semidefinite programming into an extremely powerful computational tool in practice.

Perhaps the most significant and most widely used class of algorithms are *interior-point methods*, which we will outline in Chap. 6. On the theoretical side, they are capable of providing polynomial-time bounds in the RAM model, but there is no control over the sizes of the intermediate numbers that come up in the computations, as far as we could find in the (huge) literature. Moreover, describing these methods in full detail is beyond the scope of this book.

In order to provide a simple and complete algorithm for semidefinite programming, we will present and analyze *Hazan's algorithm* in Chap. 5. This is a recent alternative method for approximately solving semidefinite programs, with a polynomial bound on the running time in the real RAM model. It comes with output guarantees similar to the ones in Theorem 2.6.1 above, and it is efficient in practice. However, the running time bound is polynomial only in  $1/\varepsilon$  and not in  $\log(1/\varepsilon)$ .

**A semidefinite program where all feasible solutions are huge.** To get such a pathological example, let us consider a semidefinite program with the following constraints:

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 2 & x_1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & x_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & x_1 & x_2 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & x_2 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & x_2 & x_3 & \cdots & 0 & 0 \\ & & & \vdots & & & \ddots & & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & x_{n-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & x_{n-1} & x_n \end{pmatrix} \succeq 0.$$

This is in fact a constraint of the form  $X \succeq 0$ , along with various equalities involving entries of  $X$ . Due to the block structure, we have  $X \succeq 0$  if and only if

$$\begin{pmatrix} 1 & x_{i-1} \\ x_{i-1} & x_i \end{pmatrix} \succeq 0, \quad i = 1, \dots, n,$$

where  $x_0 := 2$ . But this implies

$$\det \begin{pmatrix} 1 & x_{i-1} \\ x_{i-1} & x_i \end{pmatrix} = x_i - x_{i-1}^2 \geq 0, \quad i = 1, \dots, n,$$

equivalently  $x_i \geq x_{i-1}^2$ ,  $i = 1, \dots, n$ . It follows that

$$x_n \geq 2^{2^n}$$

for every feasible solution, which is doubly-exponential in  $n$ . Hence, the encoding size of  $x_n$  (when written as say a rational number) is exponential in  $n$  and also in the number of variables.

## Exercises

**2.1** Prove or disprove the following claim: For all  $A, B \in \text{SYM}_n$ , we also have  $AB \in \text{SYM}_n$ .

**2.2** Fill in the missing details of the outer product Cholesky factorization.

(i) If the matrix

$$M = \begin{pmatrix} \alpha & \mathbf{q}^T \\ \mathbf{q} & N \end{pmatrix}$$

is positive semidefinite with  $\alpha > 0$ , then the matrix

$$N - \frac{1}{\alpha} \mathbf{q} \mathbf{q}^T$$

is also positive semidefinite.

(ii) If the matrix

$$M = \begin{pmatrix} 0 & \mathbf{q}^T \\ \mathbf{q} & N \end{pmatrix}$$

is positive semidefinite, then also  $\mathbf{q} = \mathbf{0}$ .

**2.3** Show that the outer product Cholesky factorization can also be used to test whether a matrix  $M \in \mathbb{R}^{n \times n}$  is positive semidefinite.

**2.4** A *rank-constrained* semidefinite program is a problem of the form

$$\begin{array}{ll} \text{Maximize} & C \bullet X \\ \text{subject to} & A(X) = \mathbf{b} \\ & X \succeq 0 \\ & \text{rank}(X) \leq k, \end{array}$$

where  $k$  is a fixed integer. Show that the problem of solving a rank-constrained semidefinite program is NP-hard for  $k = 1$ .

**2.5** A matrix  $M \in \mathbb{R}^{n \times n}$  is called a *Euclidean distance matrix* if there exist  $n$  points  $\mathbf{p}_1, \dots, \mathbf{p}_n \in \mathbb{R}^n$ , such that  $M$  is the matrix of pairwise squared Euclidean distances, i.e.,

$$m_{ij} = \|\mathbf{p}_i - \mathbf{p}_j\|^2, \quad 1 \leq i, j \leq m.$$

Prove that a matrix  $M$  is a Euclidean distance matrix if and only if  $M$  is symmetric,  $m_{ii} = 0$  for all  $i$ , and

$$\mathbf{x}^T M \mathbf{x} \leq 0 \quad \text{for all } \mathbf{x} \text{ with } \sum_{i=1}^n x_i = 0.$$

**2.6** Let  $G = (\{1, \dots, n\}, E)$  be a graph with two edge weight functions  $\alpha_e \leq \beta_e$ ,  $e \in E$ . We want to know whether there exist points  $\mathbf{p}_1, \dots, \mathbf{p}_n \in \mathbb{R}^n$ , such that

$$\alpha_{\{i,j\}} \leq \|\mathbf{p}_i - \mathbf{p}_j\|^2 \leq \beta_{\{i,j\}}, \quad \text{for all } \{i, j\} \in E.$$

Show that this decision problem can be formulated as a semidefinite program!

### 2.7 (Sums of squares and minimization I)

- Let  $p(x) \in \mathbb{R}[x]$  be a univariate polynomial of degree  $d$  with real coefficients. We would like to decide whether  $p(x)$  is a *sum of squares*, i.e., if it can be written as  $p(x) = q_1(x)^2 + \dots + q_m(x)^2$  for some  $q_1(x), \dots, q_m(x) \in \mathbb{R}[x]$ . Formulate this problem as the feasibility of a semidefinite program.
- Let us call a polynomial  $p(x) \in \mathbb{R}[x]$  *nonnegative* if  $p(x) \geq 0$  for all  $x \in \mathbb{R}$ . Obviously, a sum of squares is nonnegative. Prove that the converse holds as well: Every nonnegative univariate polynomial is a sum of squares. (Hint: First factor into quadratic polynomials.)
- Let  $p(x) \in \mathbb{R}[x]$  be a given polynomial. Express its global minimum  $\min\{p(t) : t \in \mathbb{R}\}$  as the optimum of a suitable semidefinite program (use (b) and a suitable extension of (a)).

### 2.8 (Sums of squares and minimization II)

- Now let  $p(x_1, \dots, x_n)$  be a polynomial in  $n$  variables of degree  $d$  with real coefficients, and as in Exercise 2.7, we ask whether it can be expressed as a sum of squares (of  $n$ -variate real polynomials). Formulate this problem as the feasibility of a semidefinite program. How many variables and constraints are there in this SDP?
- Verify that the *Motzkin polynomial*  $p(x, y) = 1 + x^2y^2(x^2 + y^2 - 3)$  is nonnegative for all pairs  $(x, y) \in \mathbb{R}^2$ , but it is not a sum of squares.

Even though part (b) shows that for multivariate polynomials, nonnegativity is not equivalent to being a sum of squares, the multivariate version of the method from Exercise 2.7 constitutes a powerful tool in practice, which can find a global minimum in many cases (see, e.g., [Par06] or [Las10]).