# Approximation Algorithms and Semidefinite Programming

Jiří Matoušek Bernd Gärtner

# Contents

**1 Introduction: MAX-CUT via Semidefinite Programming 5**

1.1 The MAX-CUT Problem . . . 5
1.2 Approximation Algorithms . . . 5
1.2.1 What is Polynomial Time? . . . 7
1.3 A Randomized 0.5-Approximation Algorithm for MAX-CUT . . . 8
1.4 Semidefinite Programming . . . 9
1.4.1 From Linear to Semidefinite Programming . . . 9
1.4.2 Positive Semidefinite Matrices . . . 10
1.4.3 Cholesky Factorization . . . 10
1.4.4 Semidefinite Programs . . . 12
1.5 A Randomized 0.878-Approximation Algorithm for MAX-CUT . . . 12
1.5.1 The Semidefinite Programming Relaxation . . . 13
1.5.2 Rounding the Vector Solution . . . 15
1.5.3 Getting the Bound . . . 17
1.6 Exercises . . . 18

**2 Cone Programming 19**

2.1 Closed Convex Cones . . . 19
2.1.1 The Ice Cream Cone in $\mathbb{R}^n$ . . . 20
2.1.2 The Toppled Ice Cream Cone in $\mathbb{R}^3$ . . . 21
2.2 Dual Cones . . . 22
2.3 The Farkas Lemma, Cone Version . . . 24
2.3.1 A Separation Theorem for Closed Convex Cones . . . 24
2.3.2 Adjoint Operators . . . 25
2.3.3 The Farkas Lemma, Bogus Version . . . 26
2.3.4 The Farkas Lemma, Corrected Version . . . 27
2.4 Cone Programs . . . 28
2.4.1 Reachability of the Value . . . 29
2.4.2 The Subvalue . . . 30
2.4.3 Value vs. Subvalue . . . 30
2.5 Cone Programming Duality . . . 32
2.5.1 Weak Duality . . . 34
2.5.2 Regular Duality . . . 34

2.5.3 Strong Duality . . . . . . . . . . . . . . . . . . . . . . . . 36
2.6 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

**3 Some Applications 39**
3.1 The Cone of Positive Semidefinite Matrices . . . . . . . . . . . . 39
3.1.1 Self-Duality . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.1.2 Generating Matrices . . . . . . . . . . . . . . . . . . . . . 40
3.2 Largest Eigenvalue . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.3 Shannon Capacity and Theta Function of a Graph . . . . . . . . . 43
3.3.1 The Shannon Capacity . . . . . . . . . . . . . . . . . . . . 45
3.3.2 The Theta Function . . . . . . . . . . . . . . . . . . . . . 46
3.3.3 The Lovász Bound . . . . . . . . . . . . . . . . . . . . . . 47
3.3.4 The 5-Cycle . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.3.5 Two Semidefinite Programs for the Theta Function . . . . 51
3.3.6 The Sandwich Theorem and Perfect Graphs . . . . . . . . 53
3.4 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

**4 Interior-Point Methods 59**
4.1 The Auxiliary Problem . . . . . . . . . . . . . . . . . . . . . . . 59
4.2 Uniqueness of Solution . . . . . . . . . . . . . . . . . . . . . . . 60
4.3 Necessary Conditions for Optimality . . . . . . . . . . . . . . . . 61
4.3.1 The Method of Lagrange Multipliers . . . . . . . . . . . . 62
4.3.2 Application to the Auxiliary Problem . . . . . . . . . . . . 62
4.3.3 The Symmetric Case . . . . . . . . . . . . . . . . . . . . . 64
4.3.4 A Primal-Dual Interpretation . . . . . . . . . . . . . . . . 66
4.4 Sufficient Conditions for Optimality . . . . . . . . . . . . . . . . 66
4.5 Central Path and The Algorithm . . . . . . . . . . . . . . . . . . 69
4.5.1 Newton's Method . . . . . . . . . . . . . . . . . . . . . . . 69
4.5.2 Application to the Central Path Funtion . . . . . . . . . . 70
4.5.3 Making $\Delta X$ Symmetric . . . . . . . . . . . . . . . . . . . 71
4.5.4 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 72
4.6 Complexity: Theory Versus Practice . . . . . . . . . . . . . . . . 74
4.6.1 A Semidefinite Program with Only Huge Feasible Solutions 75
4.7 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

# Chapter 1

# Introduction: MAX-CUT via Semidefinite Programming

Let us start with a (by now) classic result of Michel X. Goemans and David P. Williamson that they first presented at the prestigious *Symposium on Theory of Computing* (STOC) in 1994 and then published as a journal paper in 1995 [6].

In this paper, we see *semidefinite programming* being used for the first time in the context of *approximation algorithms*. In reviewing the particular result concerning the MAX-CUT problem, we will try to get the reader acquainted with both concepts.

## 1.1 The MAX-CUT Problem

Given a graph $G = (V, E)$ and a subset $S \subseteq V$ of the vertices, the pair $(S, V \setminus S)$ is a *cut* of $G$. The *size* of the cut is the number of *cut edges*

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

i.e. the set of edges with one endpoint in $S$ and one in $V \setminus S$, see Figure 1.1.

The MAX-CUT problem is the following:

> Given a graph $G = (V, E)$, compute a cut $(S, V \setminus S)$ such that $|E(S, V \setminus S)|$ is as large as possible.

The decision version of this problem (given $G$ and $k \in \mathbb{N}$, is there a cut of size at least $k$?) has been shown to be NP-complete by Garey, Johnson, and Stockmeyer [3]. The above optimization version is consequently NP-hard.

## 1.2 Approximation Algorithms

Given an NP-hard optimization problem, an *approximation algorithm* for it is a *polynomial-time algorithm* that computes for *every instance* of the problem a

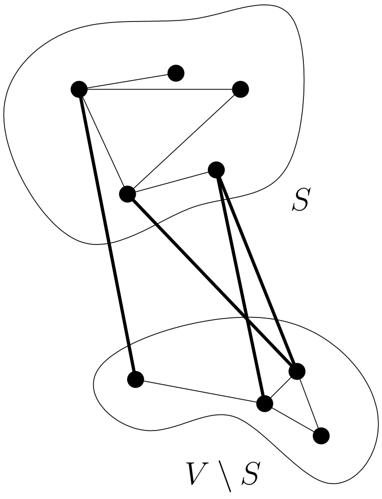


Figure 1.1: The cut edges (bold) induced by a cut $(S, V \setminus S)$

solution with some guaranteed quality. Here is a reasonably formal definition for maximization problems.

A maximization problem consists of a set $\mathcal{I}$ of *instances*. Every instance $I$ comes with a set $F(I)$ of feasible (or 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

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

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

The MAX-CUT 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 $\delta : \mathcal{I} \rightarrow \mathbb{R}_+$ be a function.*

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

*(i) There exists a polynomial $p$ such that for all $I \in \mathcal{I}$, the runtime of $\mathcal{A}$ on instance $I$ is bounded by $p(|I|)$, where $|I|$ is the encoding size of instance $I$.*

*(ii) For all instances $I \in \mathcal{I}$, $\omega(\mathcal{A}(I)) \geq \delta(I)\,\mathrm{opt}(I)$.*

An interesting special case occurs if $\delta$ 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 is the approximation. We can smoothly extend this definition to randomized algorithms (algorithms that may use internal unbiased coin flips to guide their decisions). A randomized $\delta$-approximation algorithm must have *expected* polynomial runtime and must always return a solution such that

$$E(\omega(\mathcal{A}(I))) \geq \delta(I)\,\mathrm{opt}(I).$$

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

For minimization problems, we replace sup by inf in the definition of $\mathrm{opt}(I)$ and require that $\omega(\mathcal{A}(I)) \leq \delta(I)\,\mathrm{opt}(I)$ for all $I \in \mathcal{I}$. This leads to $c$-approximation algorithms with $c \geq 1$. In the randomized case, the requirement of course is $E(\omega(\mathcal{A}(I))) \leq \delta(I)\,\mathrm{opt}(I)$.

## 1.2.1 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, or *unit cost* model is used instead.

The real RAM is a hypothetical computer, each of its memory cells being 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, 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 in fact 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 due to exploding encoding sizes of intermediate results.

For example, even Gaussian elimation, 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 [8, Section 1.4].

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 only depends 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 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 bitlengths that are also polynomial in $n$ [8, Section 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 MAX-CUT

To illustrate the previous definitions, let us describe a concrete (randomized) approximation algorithm for the MAX-CUT problem, given an instance $G = (V, E)$.

```
ApproximateMaxCut($G$) :
  begin
    $S := \emptyset$;
    foreach $v \in V$ do
      choose a random bit $b \in \{0,1\}$;
      if $b = 1$ then
        $S := S \cup \{v\}$
      end
    end
    return $(S, V \setminus S)$;
  end
```

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

**1.3.1 Theorem**. *Algorithm* `ApproximateMaxCut` *is a randomized* 1/2*-approximation algorithm for the MAX-CUT problem.*

**Proof.** The algorithm clearly runs in polynomial time. Now we compute

$$E(\omega(\texttt{ApproximateMaxCut}(G))) = E(|E(S, V \setminus S)|)$$

$$\begin{aligned} &= \sum_{e\in E} \mathrm{prob}(e \in E(S, V\setminus S)) \\ &= \sum_{e\in E} \frac{1}{2} = \frac{1}{2}|E| \geq \frac{1}{2}\,\mathrm{opt}(G), \end{aligned}$$

and the theorem is proved. 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 by the way the algorithm chooses $S$, the probability for this event is exactly $1/2$. □

It is possible to "derandomize" this algorithm and come up with a deterministic 0.5-approximation algorithm for MAX-CUT (see Exercise 1.6.1). Minor improvements are possible. For example, there exists a $0.5(1+1/m)$ approximation algorithm, where $m = |E|$, see Exercise 1.6.2.

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

## 1.4 Semidefinite Programming

Let us start with the familiar concept of *linear programming.* A linear program is the problem of maximizing (or minimizing) a linear function in $n$ variables subject to linear (in)equality 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, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is a vector of $n$ variables,[1] $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ is the objective function vector, $\mathbf{b} = (b_1, b_2, \ldots, b_m)$ is the right-hand side, and $A \in \mathbb{R}^{m\times n}$ is the constraint matrix. $\mathbf{0}$ 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 $\mathbf{x} \geq \mathbf{0}$ (the feasible solutions), we are looking for an $\mathbf{x}^*$ with highest value $\mathbf{c}^T\mathbf{x}^*$.

### 1.4.1 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

$$\mathcal{S}_n = \{X \in \mathbb{R}^{n\times n} : x_{ij} = x_{ji}\ \forall i,j\}$$

of symmetric $n \times n$ matrices, and we replace the matrix $A$ by a linear operator between $\mathcal{S}_n$ and $\mathbb{R}^m$.

---

[1] vectors are column vectors, but in writing them explicitly, we use 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

$$\langle X, Y \rangle = \mathrm{Tr}(X^T Y) = \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}y_{ij}$$

over $\mathcal{S}_n$. Let us remind the reader that for a square matrix $M$, $\mathrm{Tr}(M)$ (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

$$X \succeq 0 \quad :\Leftrightarrow \quad X \text{ is positive semidefinite.}$$

The formal definition of a semidefinite program appears below in Section 1.4.4.

## 1.4.2 Positive Semidefinite Matrices

A symmetric matrix $M$ (as such, $M$ has only real eigenvalues) is called positive semidefinite if and only if all its eigenvalues are nonnegative. Let us summarize a number of equivalent characterizations that easily follow from basic machinery in linear algebra (diagonalization of symmetric matrices).

**1.4.1 Fact**. *Let $M \in \mathcal{S}_n$. 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 $M = U^T U$.*

## 1.4.3 Cholesky Factorization

A tool that is often needed in the context of semidefinite programming is the *Cholesky factorization* of a positive semidefinite matrix $M$, meaning the computation of $U$ such that $M = U^T U$, see Fact 1.4.1(iii). As this is simple, let us explicitly do it here. The following recursive method is called the *outer product Cholesky factorization* [7, Section 4.2.8], and it requires $O(n^3)$ arithmetic operations for a given $n \times n$ matrix $M$.

If $M = (\alpha) \in \mathbb{R}^{1\times 1}$, we obtain a Cholesky factorizationx with $U = (\sqrt{\alpha})$, where $\alpha \geq 0$ by nonnegativity of eigenvalues.

Otherwise, as $M$ is symmetric, we can write $M$ in the form

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

We also have $\alpha \geq 0$, since otherwise $\mathbf{e}_1^T M \mathbf{e}_1 = \alpha < 0$ would show that $M$ is not positive semidefinite, see Fact 1.4.1 (ii). Here, $\mathbf{e}_i$ denotes the $i$-th unit vector of the appropriate dimension.

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

$$M = \begin{pmatrix} \sqrt{\alpha} & \mathbf{0}^T \\ \frac{1}{\sqrt{\alpha}}\mathbf{q} & I_{n-1} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & N - \frac{1}{\alpha}\mathbf{q}\mathbf{q}^T \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & \frac{1}{\sqrt{\alpha}}\mathbf{q}^T \\ \mathbf{0} & I_{n-1} \end{pmatrix}. \tag{1.1}$$

The matrix $N - \frac{1}{\alpha}\mathbf{q}\mathbf{q}^T$ is again positive semidefinite (Exercise 1.6.3), 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 $M = U^T U$, so we have found a Cholesky factorization of $M$.

In the other case ($\alpha = 0$), we also have $\mathbf{q} = \mathbf{0}$ (Exercise 1.6.3). The matrix $N$ is positive semidefinite (apply Fact 1.4.1 (ii) with vectors $\mathbf{x}$ of the form $(0, x_2, \ldots, x_n)$), so we can recursively compute $V$ satisfying $N = V^T V$. Setting

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

then gives $M = U^T U$.

Exercise 1.6.4 asks you to show that the above method can be modified to check whether a given matrix $M$ is positive semidefinite.

Note that the above algorithm is a polynomial-time algorithm in the real RAM model only. 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 [10] implies the following: when we run a modified version of the above algorithm (the modification is to base the decomposition (1.1) not on $m_{11}$ but 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 $\|U^T U - M\|_2 / \|M\|_2$ is bounded by $2^{-n}$.

### 1.4.4 Semidefinite Programs

**1.4.2 Definition**. *A* semidefinite program *in equational form is an optimization problem of the form*

$$\begin{array}{ll} \textit{Maximize} & \mathrm{Tr}(C^T X) \\ \textit{subject to} & A(X) = \mathbf{b} \\ & X \succeq 0, \end{array} \tag{1.2}$$

*where*

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{pmatrix} \in \mathcal{S}_n$$

*is a matrix of $n^2$ variables,*

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix} \in \mathcal{S}_n$$

*is the objective function matrix, $\mathbf{b} = (b_1, b_2, \ldots, b_m) \in \mathbb{R}^m$ is the right-hand side, and $A : \mathcal{S}_n \mapsto \mathbb{R}^m$ is a linear operator.*

Following the linear programming case, we call the semidefinite program (1.2) *feasible* if there is some *feasible solution*, a matrix $\tilde{X} \in \mathcal{S}_n$ with $A(\tilde{X}) = \mathbf{b}, \tilde{X} \succeq 0$. The *value* of a feasible semidefinite program is defined as

$$\sup\{\mathrm{Tr}(C^T X) : A(X) = \mathbf{b}, X \succeq 0\}, \tag{1.3}$$

which includes the possibility that the value is $\infty$.

An *optimal solution* is a feasible solution $X^*$ such that $\mathrm{Tr}(C^T X^*) \geq \mathrm{Tr}(C^T X)$ for all feasible solutions $X$. Consequently, if there is an optimal solution, the value of the semidefinite program is finite, and that value is attained, meaning that the supremum in (1.3) is a maximum.

## 1.5 A Randomized 0.878-Approximation Algorithm for MAX-CUT

Here, we describe a 0.878-approximation algorithm for the MAX-CUT problem, based on the fact that "semidefinite programs can be solved up to any desired accuracy $\varepsilon$, where the runtime depends polynomially on the sizes of the input numbers, and on $\log(R/\varepsilon)$, where $R$ is the maximum size $\|X\|$ of a feasible solution". We refrain from specifying this further, as a detailed statement would be somewhat technical and also unnecessary at this point. For now, let us continue with the Goemans-Williamson approximation algorithm, using the above "fact" as a black box.

We start by formulating the MAX-CUT problem as a constrained optimization problem (which we will then turn into a semidefinite program). Let us for the whole section fix the graph $G = (V, E)$, where we assume w.l.o.g. that $V = \{1, 2, \ldots, n\}$. Then we introduce variables $x_1, x_2, \ldots, x_n \in \{-1, 1\}$. Any assignment of these variables encodes a cut $(S, V \setminus S)$ where $S = \{i \in V : x_i = 1\}$. If we define

$$\omega_{ij} = \begin{cases} 1, & \text{if } \{i,j\} \in E \\ 0, & \text{otherwise} \end{cases},$$

the term

$$\omega_{ij}\left(\frac{1 - x_i x_j}{2}\right)$$

is exactly the contribution of the pair $\{i,j\}$ to the size of the above cut. Indeed, if $\{i,j\}$ is not an edge at all, or not a cut edge, we have $\omega_{ij} = 0$ or $x_i x_j = 1$, and the contribution is 0. If $\{i,j\}$ is a cut edge, then $x_i x_j = -1$, and the contribution is $\omega_{ij} = 1$. It follows that we can reformulate the MAX-CUT problem as follows.

$$\begin{array}{ll} \text{Maximize} & \sum_{i<j} \omega_{ij}\left(\frac{1-x_i x_j}{2}\right) \\ \text{subject to} & x_i \in \{-1, 1\}, \quad i = 1, \ldots, n. \end{array} \tag{1.4}$$

The value of this program is $\operatorname{opt}(G)$, the size of a maximum cut.

### 1.5.1 The Semidefinite Programming Relaxation

Here is the crucial step: We write down a semidefinite program whose value is an *upper bound* for the value $\operatorname{opt}(G)$ of (1.4). To get it, we first replace each real variable $x_i$ with a vector variable $\mathbf{x}_i \in S^{n-1} = \{\mathbf{x} \in \mathbb{R}^n \mid \|x\| = 1\}$, the $(n-1)$-dimensional unit sphere:

$$\begin{array}{ll} \text{Maximize} & \sum_{i<j} \omega_{ij}\left(\frac{1-\mathbf{x}_i^T \mathbf{x}_j}{2}\right) \\ \text{subject to} & \mathbf{x}_i \in S^{n-1}, \quad i = 1, 2, \ldots, n. \end{array} \tag{1.5}$$

From the fact that the sphere $S^0 = \{-1, 1\}$ can be embedded into $S^{n-1}$ via the mapping $\ell : x \rightarrow (0, 0, \ldots, 0, x)$, we derive the following important property: If $(x_1, x_2, \ldots, x_n)$ is a feasible solution of (1.4) with objective function value

$$z = \sum_{i<j} \omega_{ij}\left(\frac{1 - x_i x_j}{2}\right),$$

then $(\ell(x_1), \ell(x_2), \ldots, \ell(x_n))$ is a feasible solution of (1.5) with objective function value

$$\sum_{i<j} \omega_{ij}\left(\frac{1 - \ell(x_i)^T \ell(x_j)}{2}\right) = \sum_{i<j} \omega_{ij}\left(\frac{1 - x_i x_j}{2}\right) = z.$$

In other words, program (1.5) is a *relaxation* of (1.4), a program with "more" feasible solutions, and it therefore has value *at least* opt($G$). It is also clear that this value is still finite since $\mathbf{x}_i^T\mathbf{x}_j$ is lower-bounded by $-1$ for all $i, j$.

Now, it takes just another variable substitution $x_{ij} = \mathbf{x}_i^T\mathbf{x}_j$ to bring (1.5) into the form of a semidefinite program:

$$\begin{array}{ll} \text{Maximize} & \sum_{i<j} \omega_{ij}\left(\frac{1-x_{ij}}{2}\right) \\ \text{subject to} & x_{ii} = 1, \quad i = 1, 2, \ldots, n \\ & X \succeq 0. \end{array} \tag{1.6}$$

Indeed, under $x_{ij} = \mathbf{x}_i^T\mathbf{x}_j$, we have

$$X = U^T U,$$

where the matrix $U$ has the the columns $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$. Hence, $X \succeq 0$ by Fact 1.4.1(iii), and $x_{ii} = 1$ follows from $\mathbf{x}_i \in S^{n-1}$ for all $i$. Vice versa, if $X$ is any feasible matrix solution of (1.6), the columns of any matrix $U$ with $X = U^T U$ yield feasible vector solutions of (1.5); due to $x_{ii} = 1$, these vectors are actually unit vectors.

Up to a constant term of $|E|/2$ in the objective function (that we may remove, of course), (1.6) assumes the form of a semidefinite program as in (1.2). We would like to remind the reader that the symmetry constraints $x_{ij} = x_{ji}$ are implicit in this program, since a semidefinite program "lives" over the vector space $\mathcal{S}_n$ of symmetric $n \times n$ matrices only.

Now, since (1.6) is feasible, with the same finite value $\gamma \geq \text{opt}(G)$ as (1.5), we can find in polynomial time a matrix $X^* \succeq 0$ with $x_{ii}^* = 1$ for all $i$, and with objective function value at least $\gamma - \varepsilon$, for any $\varepsilon > 0$. Here we use the fact that we can give a good upper bound $R$ on the maximum size $\|X\|$ of a feasible solution. Indeed, since all entries of a feasible solution $X$ are inner products of unit vectors, the entries are in $[-1, 1]$, and $R$ is polynomial in $n$.

As shown in Section 1.4.3, we can also compute in polynomial time a matrix $U^*$ such that $X^* = U^T U$, in the real RAM model. In the Turing machine model, we can do this only approximately, but with relative error $\|U^T U - X^*\| / \|X^*\| \leq 2^{-n}$. This will be sufficient for our purposes, since this tiny error can be dealt with at the cost of slightly adapting $\varepsilon$. Let us therefore assume that the factorization is in fact exact.

Then, the columns $\mathbf{x}_1^*, \mathbf{x}_2^*, \ldots, \mathbf{x}_n^*$ of $U$ are unit vectors that form an almost-optimal solution of the vector program (1.5):

$$\sum_{i<j} \omega_{ij}\left(\frac{1 - \mathbf{x}_i^{*T}\mathbf{x}_j^*}{2}\right) \geq \gamma - \varepsilon \geq \text{opt}(G) - \varepsilon. \tag{1.7}$$

### 1.5.2 Rounding the Vector Solution

Let us recall that what we actually want to solve is program (1.4) where the $n$ variables $x_i$ are elements of $S^0 = \{-1, 1\}$ and thus determine a cut $(S, V \setminus S)$ where $S = \{i \in V : x_i = 1\}$.

What we have is an almost optimal solution of the relaxed program (1.5) 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 of our objective function value. Here is how we do it. Choose $\mathbf{p} \in S^{n-1}$ and define

$$\begin{array}{rcll} \lambda_{\mathbf{p}} & : & S^{n-1} & \rightarrow S^0 \\ & & \mathbf{x} & \mapsto \begin{cases} 1, & \text{if } \mathbf{p}^T\mathbf{x} \geq 0 \\ -1, & \text{otherwise} \end{cases} \end{array} .$$

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

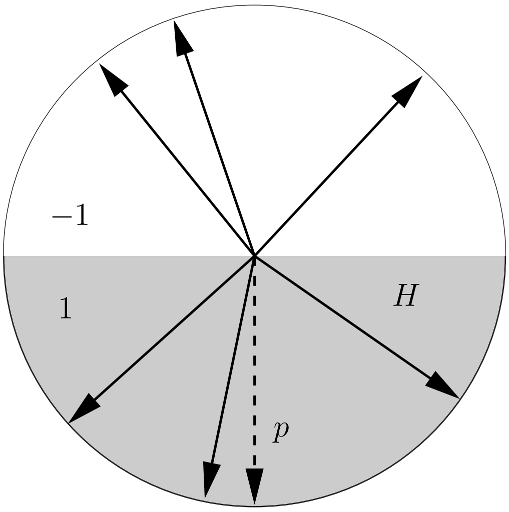


Figure 1.2: Rounding vectors in $S^{n-1}$ to $\{-1, 1\}$ through a vector $p \in S^{n-1}$

It remains to choose $\mathbf{p}$, and we will do this *randomly*. 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. In order not to lose too much of our objective function value, we certainly want that a pair of vectors $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$ with large contribution

$$\omega_{ij} \left( \frac{1 - \mathbf{x}_i^{*T}\mathbf{x}_j^*}{2} \right)$$

is more likely to yield a cut edge $\{i, j\}$ than a pair with a small contribution. Since the contribution grows with the angle between $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$, our mapping to $\{-1, +1\}$ should therefore be such that pairs with large angle are more likely to be mapped to different values than pairs with small angles.

The function $\lambda_\mathbf{p}$ with randomly chosen $\mathbf{p}$ has exactly this property as we show next.

**1.5.1 Lemma**. *Let* $\mathbf{x}_i^*, \mathbf{x}_j^* \in S^{n-1}$. *Then*

$$\text{prob}(\lambda_\mathbf{p}(\mathbf{x}_i^*) \neq \lambda_\mathbf{p}(\mathbf{x}_j^*)) = \frac{1}{\pi} \arccos \mathbf{x}_i^{*T} \mathbf{x}_j^*.$$

**Proof.** Let $\alpha \in [0, \pi]$ be the angle between $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$. By the law of cosines and because $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$ are unit vectors, we have

$$\cos(\alpha) = \mathbf{x}_i^{*T} \mathbf{x}_j^* \in [-1, 1],$$

meaning that

$$\alpha = \arccos \mathbf{x}_i^{*T} \mathbf{x}_j^* \in [0, \pi].$$

If $\alpha \in \{0, \pi\}$, meaning that $\mathbf{x}_i^* \in \{\mathbf{x}_j^*, -\mathbf{x}_j^*\}$, the statement trivially holds. Otherwise, let us consider the linear span $L$ of $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$ which is a twodimensional subspace of $\mathbb{R}^n$. With $\mathbf{r}$ being the projection of $\mathbf{p}$ to that subspace, we have $\mathbf{p}^T \mathbf{x}_i^* = \mathbf{r}^T \mathbf{x}_i^*$ and $\mathbf{p}^T \mathbf{x}_j^* = \mathbf{r}^T \mathbf{x}_j^*$. This means that $\lambda_\mathbf{p}(\mathbf{x}_i^*) \neq \lambda_\mathbf{p}(\mathbf{x}_i^*)$ if and only if $\mathbf{r}$ lies in a "half-open double wedge" of opening angle $\alpha$, see Figure 1.3.

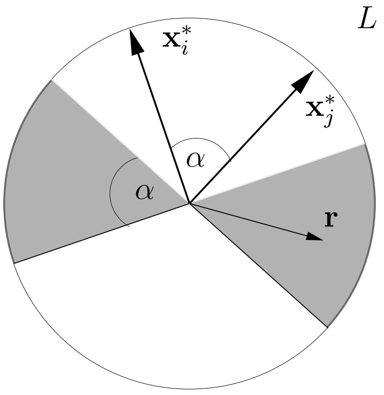


Figure 1.3: Rounding the vector solution: $\lambda_\mathbf{p}(\mathbf{x}_i^*) \neq \lambda_\mathbf{p}(\mathbf{x}_i^*)$ if and only if the projection $\mathbf{r}$ of $\mathbf{p}$ to the linear span of $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$ lies in the shaded region ("half-open double wedge")

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

### 1.5.3 Getting the Bound

Let's see what we have achieved. If we round as above, the expected number of cut edges being produced is

$$\sum_{i<j} \omega_{ij} \left( \frac{\arccos \mathbf{x}_i^{*T} \mathbf{x}_j^*}{\pi} \right).$$

Indeed, this sum contributes the "cut edge probability" of Lemma 1.5.1 for each edge $\{i, j\}$, and nothing otherwise.

The trouble is that we don't know much about this sum. But we *do* know that

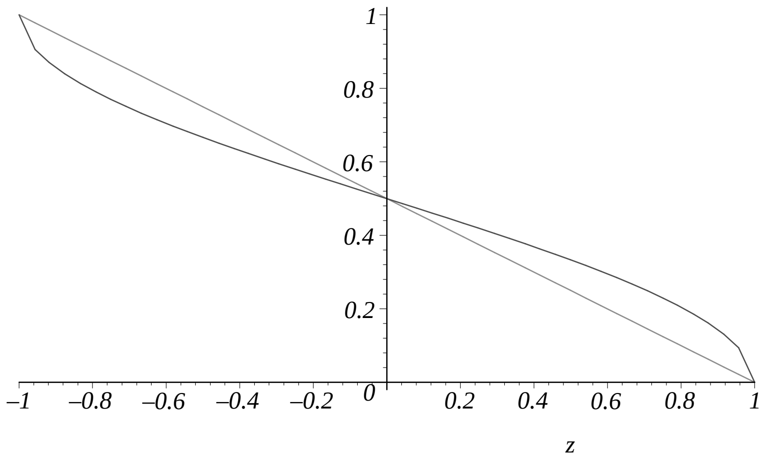
$$\sum_{i<1} \omega_{ij} \left( \frac{1 - \mathbf{x}_i^{*T} \mathbf{x}_j^*}{2} \right) \geq \mathrm{opt}(G) - \varepsilon,$$

see (1.7). But the following technical lemma allows us to compare the two sums termwise.

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

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

**Proof.** The plot below depicts the two functions $f(z) = \arccos(z)/\pi$ and $g(z) = (1 - z)/2$ over the interval $[-1, 1]$. The quotient becomes smallest at the unique value $z^*$ for which the derivative of $f(z)/g(z)$ vanishes. Using a numeric solver, you can compute $z^* \approx -0.68915774$ which yields $f(z^*)/g(z^*) \approx 0.87856723 > 0.8785672$.



□

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

$$\begin{aligned}\sum_{i<j} \omega_{ij} \left( \frac{\arccos \mathbf{x}_i^{*T} \mathbf{x}_j^*}{\pi} \right) &\geq 0.87856723 \sum_{i<j} \omega_{ij} \left( \frac{1 - \mathbf{x}_i^{*T} \mathbf{x}_j^*}{2} \right) \\ &\geq 0.87856723(\mathrm{opt}(G) - \varepsilon) \\ &\geq 0.878\, \mathrm{opt}(G),\end{aligned}$$

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

## 1.6 Exercises

**1.6.1 Exercise**. *Prove that there is also a deterministic* $0.5$*-approximation algorithm for the MAX-CUT problem.*

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

**1.6.3 Exercise**. *Fill in the missing details of the 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* $\mathbf{q} = \mathbf{0}$*.*

**1.6.4 Exercise**. *Provide a method for checking with* $O(n^3)$ *arithmetic operations whether a matrix* $M \in \mathbb{R}^{n \times n}$ *is positive semidefinite.*