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.

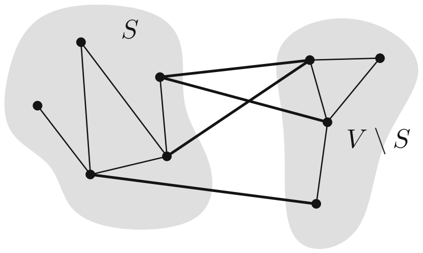

Fig. 1.1 The cut edges (*bold*) induced by a cut $(S, V \setminus S)$

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

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

to be the optimum value of the instance. Value $-\infty$ occurs if $F(I) = \emptyset$, while $\mathsf{Opt}(I) = \infty$ means that there are feasible solutions of arbitrarily large value. To simplify the presentation, let us restrict our attention to problems where $\mathsf{Opt}(I)$ 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 $\delta\colon \mathbb{N} \to \mathbb{R}_+$ be a function.*

We say that $\mathcal{A}$ is a δ-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 $I \in \mathcal{I}$, $\omega(\mathcal{A}(I)) \geq \delta(|I|) \cdot \mathsf{Opt}(I)$.*

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 δ is a constant function. For $c \in \mathbb{R}$, a c-approximation algorithm is a δ-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 δ-approximation algorithm must have *expected* polynomial runtime and must satisfy

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

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 $\mathsf{Opt}(I)$ and we require that $\omega(\mathcal{A}(I)) \leq \delta(|I|)\mathsf{Opt}(I)$ 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 π.

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 $\omega(\texttt{RandomizedMaxCut}(G))$ is the size of the cut (number of cut edges) generated by the algorithm (a random variable). Now we compute

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

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}, $i, j = 1, 2, \ldots, n$, 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 ε, 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 $V = \{1, 2, \ldots, n\}$ (this will be used often and in many places). Then we introduce variables $z_1, z_2, \ldots, z_n \in \{-1, 1\}$. 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.

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

The optimum value (or simply value) of this program is $\mathsf{Opt}(G)$, 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 $\mathsf{Opt}(G)$ 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:

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

This is called a *vector program* since the unknowns are vectors.[1]

From the fact that the set $\{-1, 1\}$ can be embedded into S^{n-1} via the mapping $x \mapsto (0, 0, \ldots, 0, x)$, 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* $\mathsf{Opt}(G)$. It is also

[1] 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:

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

To see that (1.3) is equivalent to (1.2), we first note that if $\mathbf{u}_1, \ldots, \mathbf{u}_n$ 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 $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$. 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 $\mathsf{SDP}(G) \geq \mathsf{Opt}(G)$ as (1.2). So we can find in polynomial time a matrix $X^* \succeq 0$ with $x_{ii}^* = 1$ for all i and with

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

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 ε. So let us assume that the factorization is exact.

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

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

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

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

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.

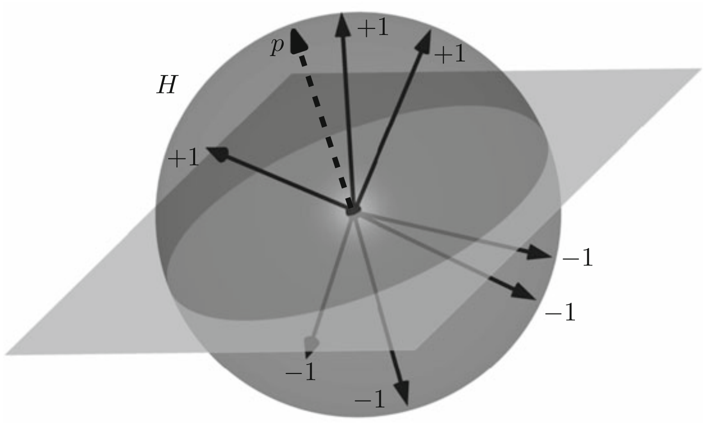

Fig. 1.2 Rounding vectors in S^{n-1} to $\{-1, 1\}$ through a vector $\mathbf{p} \in S^{n-1}$

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 α; see Fig. 1.3.

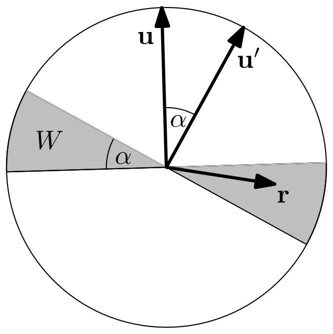

Fig. 1.3 Randomly rounding vectors: $\mathbf{u}$ and $\mathbf{u}'$ map to different values if and only if the projection $\mathbf{r}$ of $\mathbf{p}$ to the linear span of $\mathbf{u}$ and $\mathbf{u}'$ lies in the shaded region W ("half-open double wedge")

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 α/π. $\square$

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

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

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

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

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

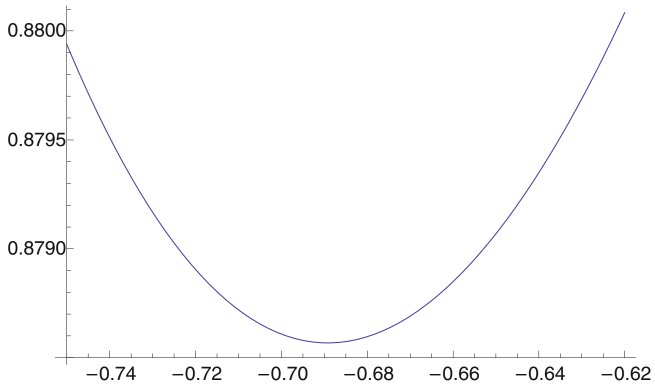

Fig. 1.4 The function $f(z) = 2\arccos(z)/\pi(1-z)$ and its minimum

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

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

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

   This is a solution that satisfies

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

   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

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

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.