Grothendieck's Inequality and the MAX-CUT Problem

Alex Rambasek

January 6, 2023

1 Problem Formulation

Let $\mathcal{G} = (V, E)$ be an undirected graph, $|V| = n$, $|E| = m$. For simplicity we will assume that $\mathcal{G}$ is acyclic. A **cut** $(S, V \setminus S)$ of $\mathcal{G}$ is a partitioning of all of the vertices V into two disjoint subsets. All graph edges joining vertices in different subsets is called the **cut set**: $C = \{(v_1, v_2) \in E | v_1 \in S, v_2 \in V \setminus S\}$. If $(x, y) \in C$, we say that the edge *crosses the cut*. The *size* of the cut is the cardinality of C. The MAX-CUT problem is to find a graph cut of maximal size: MAX-CUT$(\mathcal{G}) = \mathbf{max}_{(S,V\setminus S)}|C|$.

This is a perfectly acceptable way of thinking about the MAX-CUT, but it will be useful to introduce more notation so that we can formulate the problem as a numerical system that we can solve. We call $\mathbf{A} \in \mathbb{R}^{n\times n}$ the **adjacency** matrix of $\mathcal{G}$ if $\mathbf{A}_{ij} = 1$ whenever vertices i and j share an edge, and 0 otherwise. In a more general setting, $\mathbf{A}$ can contain weights that are real numbers representing the strength of a connection between two vertices, with a weight of 0 if and only if no connection exists. For our purposes, it will be sufficient to think about $\mathbf{A}$ over a binary field, with the understanding that the generalization to a weighted adjacency matrix and subsequently weighted MAX-CUT (find the cut of maximal edge weight sum) is straightforward. It follows that

$$\text{MAX-CUT}(\mathcal{G}) = \mathbf{max}\frac{1}{2}\sum_{i,j:x_i x_j=-1} \mathbf{A}_{ij} \tag{1}$$

where $x_i = 1$ if $v_i \in \mathbf{S}$ and -1 if $v_i \in \mathbf{S} \setminus \mathbf{V}$. The size of the cut is exactly the count of the number of edges crossing the cut (the factor of $\frac{1}{2}$ prevents us from double counting (v_i, v_j) and (v_j, v_i)). Now, let us introduce decision variables. Notice that for $i \neq j$, $x_i x_j = 1$ if v_i and v_j are on the same side of the cut, and -1 if they are on opposite sides. So, the quantity $\frac{\mathbf{A}_{ij}(1-x_i x_j)}{2}$ is 1 for vertices on opposite sides of the cut, and 0 otherwise. This expression is more useful because summing over all vertex pairs gives the size of the cut. This is easily checked. With this new notation, our linear program for MAX-CUT is:

$$\text{MAX-CUT}(\mathcal{G}) = \mathbf{max}\left\{\frac{1}{4}\sum_{i,j}\mathbf{A}_{ij}(1-\mathbf{x}_i\mathbf{x}_j) : \mathbf{x}_i = \pm 1 \; \forall \; i\right\}. \tag{2}$$

1.1 Linear Programming

Definition 1 (Linear Program)**.** *A linear program is an optimization problem of the following form:*

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0} \end{array}$$

The $\leq$ symbol denotes entry-wise comparison. This is the **canonical form** of a linear program. An **integer linear program** is a linear program with the additional constraint that $x_i \in \mathbb{Z}$. Our last formulation of the MAX-CUT problem can be modeled as a linear program, with the added constraint that the decision variables are integer (an **integer linear program**). However, despite its simplicity, solving general integer linear programs is NP-complete: unless $P = NP$, there are no polynomial-time algorithms to find an optimum. This is an issue because, for large graphs, a brute-force will quickly become computationally intractable. In fact, MAX-CUT is APX-hard, meaning that in addition to the lack of polynomial solutions, there are no polynomial approximations, either. If we could get rid of the integrality constraint, then

the program is solvable in polynomial time by an interior point method. [1]

As it turns out, we can do better. It has long been known that convex optimization problems can be solved efficiently in theory, but existing methods were slow in practice. Let us write our linear program in **standard form**:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{Ax} = \mathbf{b}, \mathbf{x} \in \mathbb{R}_n^+. \end{array} \tag{3}$$

where $\mathbb{R}_n^+$ is the set of all real non-negative vectors. Every ILP can be written in standard form by introducing slack variables. It was discovered that $\mathbb{R}_n^+$ can be replaced by the convex cone of positive semi-definite (PSD) matrices, and that doing so still allows for good theoretical guarantees from interior point solvers.

1.2 Semi-definite Programming

Definition 2 (PSD Matrix)**.** *Let* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *be a symmetric matrix.* $\mathbf{A}$ *is said to be* ***positive definite*** *(written* $\mathbf{A} \succ 0$*) if* $\forall\ \mathbf{v} \in \mathbb{R}^n, \mathbf{v^T Av} > 0$*. If* $\mathbf{v^T Av} \geq 0$*, we say that* $\mathbf{A}$ *is* ***positive semi-definite*** *(written* $\mathbf{A} \succeq 0$*).*

Theorem 1. *(Convexity of PSD matrices) The set of all PSD matrices is convex. That is, if* $\mathbf{A}$ *and* $\mathbf{B}$ *are PSD, then* $\lambda\mathbf{A} + (1-\lambda)\mathbf{B}$ *for* $\lambda \in [0,1]$ *is PSD as well.*

Proof. By assumption, $\mathbf{v^T Av} \geq 0$ and $\mathbf{v^T Bv} \geq 0$.

$$\begin{aligned} v^T(\lambda\mathbf{A} + (1-\lambda)\mathbf{B})v &= \lambda\mathbf{v^T Av} + (1-\lambda)\mathbf{v^T Bv} \\ &\geq 0. \end{aligned}$$

□

Now, let us define the general form of a semi-definite program:

$$\begin{array}{ll} \text{maximize} & \langle \mathbf{A}, \mathbf{X} \rangle \\ \text{subject to} & \mathbf{X} \succeq 0, \langle \mathbf{B}_i, X \rangle = \mathbf{b}_i \text{ for } i = 1, ..., m. \end{array} \tag{4}$$

where $\langle \mathbf{A}, \mathbf{X} \rangle$ denotes the canonical inner product on the space of $n \times n$ matrices:

$$\langle \mathbf{A}, \mathbf{X} \rangle = \mathrm{tr}(\mathbf{A^T X}) = \sum_{i,j=1}^{n} A_{ij} X_{ij}. \tag{5}$$

From what we've already shown, every semidefinite program is a convex program. In order to write the MAX-CUT problem as a semi-definite program, we are going to relax the integrality constraints of our decision variables. That is, instead of requiring that $x_i \in \{1, -1\}$, we will instead opt for decision vectors of unit norm, replacing multiplication with a dot product.

Definition 3 (SDP for MAX-CUT)**.**

$$\text{MAX-CUT}(\mathcal{G}) = \frac{1}{4}\mathbf{max} \left\{ \sum_{i,j=1}^{n} \mathbf{A}_{ij}(1 - \langle \mathbf{X}_i, \mathbf{X}_j \rangle) : \mathbf{X}_i \in \mathbb{R}^n, ||\mathbf{X}_i||_2 = 1 \; \forall \; i \right\}.$$

It is not immediately obvious that this program is semi-definite as we've defined it, but it in fact is. Define $\mathbf{X}$ to be the *Gram matrix* of the vectors $\mathbf{X_i}$, which is the $n \times n$ matrix with entries $\langle \mathbf{X_i}, \mathbf{X_j} \rangle$. If we expand out the condition $\mathbf{X} \succeq 0 \equiv \mathbf{v^T X v} \geq \mathbf{0}$ using the fact that $\mathbf{X}$ is a Gram matrix, we get

$$\begin{aligned}
\mathbf{v^T X v} &= \sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{v}_i^T \mathbf{v}_j \langle \mathbf{X}_i, \mathbf{X}_j \rangle \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} \langle \mathbf{v}_i \mathbf{X}_i, \mathbf{v}_j \mathbf{X}_j \rangle \\
&= \left\langle \sum_i \mathbf{v}_i \mathbf{X}_i, \sum_j \mathbf{v}_j \mathbf{X}_j \right\rangle \\
&= \left|\left| \sum_i \mathbf{v}_i \mathbf{X}_i \right|\right|^2 \geq 0
\end{aligned}$$

so $\mathbf{X}$ is PSD. Then clearly the above program can be written

$$\begin{array}{ll} \text{maximize} & \frac{1}{4}\langle \mathbf{A}, \mathbf{X} \rangle \\ \text{subject to} & \mathbf{X} \succeq 0, \mathbf{X}_i \in \mathbb{R}^n, ||\mathbf{X}_i||_2 = 1 \; \forall \; i \end{array} \tag{6}$$

with the same matrix inner product as before. Recall that

$$\langle \mathbf{X}_i, \mathbf{X}_i \rangle = ||\mathbf{X}_i||_2^2.$$

So, the above condition that $||\mathbf{X}_i||_2 = 1 \; \forall \; i$ is exactly true when $\langle \mathbf{X}_i, \mathbf{X}_i \rangle = \mathbf{X}_{ii} = 1$. So, (6) is equivalent to

$$\begin{array}{ll} \text{maximize} & \frac{1}{4}\langle \mathbf{A}, \mathbf{X} \rangle \\ \text{subject to} & \mathbf{X} \succeq 0, \mathbf{X}_{ii} = 1 \; \forall \; i \end{array} \tag{7}$$

and we have shown that our program for MAX-CUT is indeed semi-definite. So, we have successfully translated MAX-CUT into a integer linear program, and found its semidefinite relaxation. However, it is not readily apparent as to how a solution to the SDP can be translated into a feasible solution for the MAX-CUT problem. How do we get a partition for $\mathcal{G}$ from the vectors X_i? Surprisingly, it turns out that we can partition the X_is at random, and get a good approximation to the optimal cut. This results from Grothendieck's inequality, and forms the basis of the Goemans Williamson algorithm for solving MAX-CUT.

1.3 Grothendieck's Inequality

Theorem 2 (Grothendieck's inequality)**.** *Let $\mathbf{A}_{ij}$ be a real $m \times n$ matrix. Assume that the following holds for any $x_i, y_j \in \{-1, 1\}$:*

$$\left| \sum_{i,j} a_{ij} x_i y_j \right| \leq 1.$$

Then, for any Hilbert space H and any unit vectors $\mathbf{u}_i, \mathbf{v}_j \in H$, we have

$$\left| \sum_{i,j} a_{ij} \langle u_i, v_j \rangle \right| \leq K,$$

where $K \leq 1.783$ is an absolute constant.

We will not prove Grothendieck's inequality here, but we will note its application in bounding the distance between the solution of an integer linear program and that of its semi-definite relaxation.

2 Approximating MAX-CUT

2.1 Approximation Algorithms

Computing the exact solution of MAX-CUT($\mathcal{G}$) is intractable for large n, so we will instead concern ourselves with finding an approximation, and bounding its distance to the true solution. Let us introduce the notion of an approximation algorithm.

Definition 4 (ρ-approximation)**.** *Let I be an instance of an optimization problem, and let $OPT(I) \in \mathbb{R}$ be the value of the (not necessarily unique) optimal solution of I. An approximation algorithm A of I is said to be a $\rho-$* **approximation** *of I if the value of any solution returned by A is at most a factor of ρ away from the optimum:*

$$\begin{cases} OPT(I) \leq A(I) \leq \rho OPT(I), & \text{if } \rho > 1 \\ \rho OPT(I) \leq A(I) \leq OPT(I), & \text{if } \rho < 1 \end{cases}$$

Consider a naïve algorithm to compute a graph cut $(S, V \setminus S)$: for every vertex, flip a coin. If it comes up heads, assign that vertex to S. If it's tails, $V \setminus S$. The partitioning can be modeled as a symmetric Bernoulli random vector $x \sim \text{Unif}\{-1, 1\}^n$. We can then calculate the expected magnitude of the cut:

$$\begin{aligned} \mathbb{E}[\text{CUT}(\mathcal{G})] &= \frac{1}{4}\sum_{i,j} \mathbf{A}_{ij}(1 - \mathbb{E}[\mathbf{x}_i\mathbf{x}_j]) \\ &= \frac{1}{4}\sum_{i,j} \mathbf{A}_{ij} \\ &= \frac{1}{2}|E|. \end{aligned}$$

Thus our naïve algorithm is a 0.5-approximation algorithm of MAX-CUT; the largest cut can't be larger than the number of edges in the graph.

Returning to our semi-definite program, let us consider Michel Goemans and David Williamson's algorithm to turn our vectors $\mathbf{X}_i$ into decision variables x_i for the original MAX-CUT problem. Sample a standard normal random vector $\mathbf{g} \sim \mathcal{N}(0, \mathbf{I}_n)$ and define $x_i = \text{sign}\langle \mathbf{X}_i, \mathbf{g} \rangle$. This can be visualized as choosing a random hyperplane in $\mathbb{R}^n$ that passes through the origin and divides the vectors into two halves. The resulting x_is produce a cut that is an approximation of the maximal cut.

Algorithm 1: Goemans Williamson Algorithm

Input: A graph G.
Output: A cut of the graph.

$D = \texttt{degreeMatrix}(G)$
$A = \texttt{adjacencyMatrix}(G)$
$L = D - A$
$X =$solve max $\frac{1}{4} \cdot \text{trace}(\langle L, X \rangle)$ s.t. $\text{diag}(X) == 1$
$g \sim \mathcal{N}(0, I_n)$
$newList = [\,]$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $newList.append(\text{sign}(\langle X[i], g \rangle))$
end for
return $newList$

To derive the relative performance guarantee of this randomized rounding step, we first need to prove Grothendieck's identity.

Lemma 1 (Grothendieck's identity)**.** *Consider a random vector* $\mathbf{g} \sim \mathcal{N}(0, I_n)$*. Then, for any fixed vectors* $\mathbf{u}, \mathbf{v} \in S^{n-1}$*,*

$$\mathbb{E}[\text{sign}\langle \mathbf{g}, \mathbf{u} \rangle \text{sign}\langle \mathbf{g}, \mathbf{v} \rangle] = \frac{2}{\pi} \arcsin\langle \mathbf{u}, \mathbf{u} \rangle.$$

Proof. $\text{sign}\langle \mathbf{g}, \mathbf{u} \rangle \text{sign}\langle \mathbf{g}, \mathbf{v} \rangle$ is 1 if and only if u and v are on the same side of the hyperplane partition. So we may write $\mathbb{E}[\text{sign}\langle \mathbf{g}, \mathbf{u} \rangle \text{sign}\langle \mathbf{g}, \mathbf{v} \rangle] = \mathbb{P}[\mathbf{u}, \mathbf{v} \text{ in same half}] - \mathbb{P}[\mathbf{u}, \mathbf{v} \text{ in different halves}] = 1 - 2\mathbb{P}[\mathbf{u}, \mathbf{v} \text{ in different halves}]$.

Since u and v are unit vectors, this is the same as u and v lying on different halves of the unit sphere. The probability that u and v are on different halves when it is divided by $\mathbf{g}$ is $\frac{\alpha}{\pi}$, where α is the angle between $\mathbf{u}$ and $\mathbf{v}$. This follows from the rotational invariance of $\mathbf{g}$ and a simple geometric argument. So, $\mathbb{E}[\text{sign}\langle \mathbf{g}, \mathbf{u}\rangle \text{sign}\langle \mathbf{g}, \mathbf{v}\rangle] = 1 - \frac{2\alpha}{\pi}$. Also, $\frac{2}{\pi}\arcsin(\langle \mathbf{u}, \mathbf{v}\rangle) = \frac{2}{\pi}\arcsin(\cos(\alpha)) = \frac{2}{\pi}\arcsin(\sin(\frac{\pi}{2} - \alpha)) = \frac{2}{\pi}(\frac{\pi}{2} - \alpha) = 1 - \frac{2\alpha}{\pi}$. □

With Grothendieck's identity proved, we now have the tools to bound the relative performance guarantee of the Goemans Williamson algorithm.

Theorem 3 (0.878-approximation algorithm for MAX-CUT)**.** *Let $\mathcal{G}$ be a graph with adjacency matrix $\mathbf{A}$. Let $\mathbf{x} = (x_i)$ be the result of a randomized rounding of the solution $(\mathbf{X}_i)$ of the semi-definite program for MAX-CUT. Then*

$$\mathbb{E}\mathit{CUT}(\mathcal{G}, \mathbf{x}) \geq 0.878 \cdot \mathit{SDP}(\mathcal{G}) \geq 0.878 \cdot \mathit{MAX\text{-}CUT}(\mathcal{G}).$$

Proof. The second inequality is trivial because $\text{SDP}(G) \geq \text{MAX-CUT}$: SDP is a relaxation, so the optimum of MAX-CUT can always be translated into a feasible solution for SDP.

$$\begin{aligned}
\mathbb{E}[\text{CUT}(\mathcal{G})] = & \quad \frac{1}{4}\sum_{i,j} \mathbf{A}_{ij}(1 - \mathbb{E}[\mathbf{x}_i\mathbf{x}_j]) \\
= & \quad \frac{1}{4}\sum_{i,j} \mathbf{A}_{ij}(1 - \mathbb{E}[\text{sign}\langle \mathbf{g}, \mathbf{u}\rangle \text{sign}\langle \mathbf{g}, \mathbf{v}\rangle]) && \text{(labels from rounding step)} \\
= & \quad 1 - \frac{2}{\pi}\arcsin\langle \mathbf{X}_i, \mathbf{X}_j\rangle && \text{(Grothendieck's identity)} \\
= & \quad 0.878(1 - \langle X_i, X_j\rangle) && \text{(numeric inequality for } \frac{2}{\pi}\arcsin(x))
\end{aligned}$$

□

3 Application: NAE 3-SAT

In the literature, there are two major applications of the MAX-CUT problem. The first is an equivalence between the MAX-CUT problem and the most

likely assignment to an Ising spin glass model, and the second is VLSI circuit design. I was not able to find suitable open-source datasets for either of these problems, so I opted to examine a niche but interesting application of MAX-CUT: boolean satisfiability.

The boolean satisfiability problem (SAT) is a classic problem in computer science. The problem is to find an assignment of variables in a boolean formula such that the entire formula evaluates to TRUE. Every boolean logic expression can be written as a conjunction of disjunctions (e.g., $a \implies (b \oplus \neg(c \vee \neg d)) \equiv (a \vee b \vee c \vee d) \wedge (a \vee b \vee \neg c \vee d) \wedge (a \vee b \vee \neg c \vee \neg d) \wedge (a \vee \neg b \vee c \vee \neg d)$). 3-SAT is a restriction of SAT that only permits length-3 clauses, and NAE 3-SAT is a restriction on 3-SAT that prevents any clause from having all TRUE or all FALSE literals. NAE 3-SAT reduces in polynomial time to MAX-CUT. To see this, support we have an instance of NAE 3-SAT. For every variable in the instance, add two vertices to $\mathcal{G}$: one for itself and one for the negated variable. Connect vertices of the same variable by an edge, and connect all members of a clause into a fully connected subgraph. Along "variable" edges, we assign capacity $M = 10 \cdot m$, where m is the number of clauses and n the number of variables. Then, $\mathcal{G}$ contains a cut with capacity at least $n \cdot M + 2 \cdot m$ if and only if the NAE 3-SAT instance is feasible. [2] A given assignment of this instance corresponds to a cut in $\mathcal{G}$, and by nature of being a satisfying assignment, exactly 2 edges in each clause cut the clause subgraph "triangle," because otherwise a clause would contain all of the same truth value. These edges collectively contribute $2 \cdot m$ to the cut capacity. All variable edges must cross the cut (exactly one of x_i and $\neg x_i$ is true), and these edges contribute $n \cdot M$ to the capacity of the cut. The converse follows similarly.

If we use the Goemans Williamson algorithm to find a cut of the graph described above, and the capacity of that cut is $\geq n \cdot M + 2 \cdot m$, then we know a satisfying assignment must exist by the above argument. I randomly generated 3-SAT instances, ran Goemans Williamson on the graph, and used a heuristic SAT solver to find a satisfying assignment if one exists.

4 Acknowledgements

Vershynin's *High Dimensional Probability* [3] book was invaluable in providing a theoretical framework for the material in this paper, and a number of theorems proved here appear as exercises in the textbook. I would like to thank Dr. Mark Meckes for providing helpful feedback on an earlier draft of this paper.

5 Appendix: Python code

```python
import cvxpy as cvx
import networkx as nx
import numpy as np

def goemans_williamson(g: nx.Graph) -> np.ndarray:
    # SDP formulation of MAX-CUT
    L = np.array(0.25 * nx.laplacian_matrix(g).todense())
    n = len(g.degree)
    M = cvx.Variable((n, n), PSD=True)
    op = cvx.Problem(cvx.Maximize(cvx.trace(L @ M)), [cvx.diag(M) == 1])
    op.solve(solver=cvx.CVXOPT)

    U = np.linalg.cholesky(M.value)

    # random hyperplane through origin
    r = np.random.randn(n)
    r = r / np.linalg.norm(r)

    return np.sign(U @ r)
```

References

[1] N. Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming". In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 302–311. ISBN: 0897911334. DOI: 10.1145/800057.808695. URL: https://doi.org/10.1145/800057.808695.

[2] David Steurer. *Cornell University*. https://www.cs.cornell.edu/courses/cs4820/2014sp/notes/reduction-maxcut.pdf. Mar. 2014.

[3] Roman Vershynin. *High-Dimensional Probability: An Introduction with Applications in Data Science*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2018. DOI: 10.1017/9781108231596.