Tree-width Driven SDP for The Max-Cut Problem

Иван Воронин

Научный руководитель: Александр Булкин

Moscow Institute of Physics and Technology

2 апреля 2024 г.

Outline

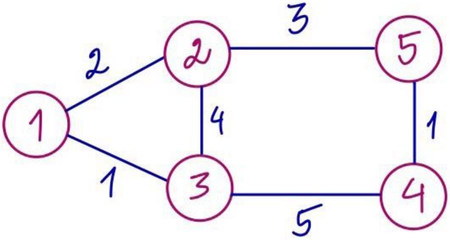
Problem statement

Given a weighted, undirected graph $G = (V, E)$ i.e. each edge, (i, j), has a weight, $w_{ij} = w_{ji}$. The set of vertices is partitioned into two parts, S and $\bar{S} := V \setminus S$. Let us call the weight of this "cut" the sum of the weights of edges whose endpoints lie in different parts

$$W(S) := \sum_{(i,j) \in S \times \bar{S}} w_{ij}$$

The goal is to find the cut with maximal possible weight.

Example



S	$W(S)$
$\{1,2,3\}$	$3+5=8$
$\{1,2,3,4\}$	$3+1=4$
$\{1,5,4\}$	$2+3+4+1+5=15$
$\{1,3,5\}$	$2+4+3+1+5=15$

The maximum cut is 15. Actually, the graph is not bipartite, the total weight of all edges is 16, and there are no edges lighter than 1.

Semidefinite Program

Let us rephrase the problem in the context of Integer Linear Programming (ILP) and reduce it to Semidefinite Programming (SDP)
For each vertex i in the graph, we define the indicator $x_i \in \{-1, +1\}$, characterizing the affiliation of $i \in S$ or $i \in \bar{S}$, respectively.
Similarly, for each edge (i, j), we define the indicator $y_{ij} = y_{ji} = x_i x_j \in \{-1, +1\}$ characterizing the belonging of the edge to the cut. Let x represent the vector $x = (x_1, \ldots, x_n)$, where $n = |V|$.

Now the Max-Cut can be represented as

$$ILP = \max_{\substack{x_i \in \{-1,+1\} \\ y_{ij} = x_i x_j}} \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - y_{ij}) = \max_{x_i^2 = 1} \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - x_i x_j) =$$

$$= \max_{\substack{X = xx^T \\ X_{ii} = 1}} \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - X_{ij}) = SDP$$

Clearly, $X_{ij} = x_i x_j$, hence $x_i^2 = X_{ii} = 1 \implies x_i \in \{-1, +1\}$

In particular, the matrix X is

1. Symmetric with units on the diagonal
2. Positive semi-definite, indeed
   $\forall v \in \mathbb{R}^n : \quad v^T X v = v^T x x^T v = (x^T v)^T (x^T v) = (x^T v)^2$

Finally, let us consider the following problem

$$SDP^* = \max \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - X_{ij}), \text{ где } X = \begin{pmatrix} 1 & x_{12} & \dots & x_{1n} \\ x_{12} & 1 & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1n} & x_{2n} & \dots & 1 \end{pmatrix} \succcurlyeq 0$$

Such formulation is a relaxation of the Max-Cut problem (matrix X still meets properties 1 and 2).
The difference between the SDP and SDP^* is as follows

	SDP	SDP^*
X	$X = xx^T, \ x \in \mathbb{R}^n$	$X = LL^T, \ x \in \mathbb{R}^{n \times m}$
X_{ij}	$X_{ij} = x_i x_j \in \{-1, +1\}$	X_{ij} arbitrary element

```python
import numpy as np
import cvxpy as cp

n = 5
W = np.array([[0, 2, 1, 0, 0],
              [2, 0, 4, 0, 3],
              [1, 4, 0, 5, 0],
              [0, 0, 5, 0, 1],
              [0, 3, 0, 1, 0]])


X = cp.Variable((n, n), symmetric=True)
constraints = [X >> 0]
constraints += [X[i][i] == 1 for i in range(n)]
objective = cp.Maximize(0.25 * cp.sum(cp.multiply(W,
    (1 - X))))
prob = cp.Problem(objective, constraints)
prob.solve()
```

The optimal value is 15.000002187529262
A solution X is

$$\begin{pmatrix} 1 & -1.00000047 & 1.0000005 & -1.00000052 & 1.00000057 \\ -1.00000047 & 1 & -1.00000027 & 1.00000038 & -1.00000037 \\ 1.0000005 & -1.00000027 & 1 & -1.00000026 & 1.0000004 \\ -1.00000052 & 1.00000038 & -1.00000026 & 1 & -1.00000042 \\ 1.00000057 & -1.00000037 & 1.0000004 & -1.00000042 & 1 \end{pmatrix}$$

Indeed, the matrix $X = \begin{pmatrix} 1 & -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 & 1 \end{pmatrix}$

corresponds to the maximum cut $S = \{1, 3, 5\}$ of the value $W(S) = 15$

Dual Problem

$$OPT = \max_{x_i^2=1} x^T L x = 4 \cdot MaxCut, \text{ where } L \text{ is the Laplacian of the graph}$$

$$Dual = \max_{\lambda} \min_{x} \sum_{i=1}^{n} \lambda_i (1 - x_i^2) - \sum_{i,j}^{n} x_i x_j L_{ij} =$$

$$= \max_{\lambda} \min_{x} \sum_{i=1}^{n} \lambda_i - \sum_{i,j}^{n} x_i x_j L_{ij} - \sum_{i=1}^{n} \lambda_i x_i^2 = \min_{Diag(\xi) \succcurlyeq L} \sum_{i=1}^{n} \xi_i =$$

$$= \min_{Diag(\xi) \succcurlyeq L} \max_{x_i^2=1} x^T Diag(\xi) x$$

```python
import numpy as np
import cvxpy as cp

n = 5
L = np.array([[ 3, -2, -1,  0,  0],
              [-2,  9, -4,  0, -3],
              [-1, -4,  9, -5,  0],
              [ 0,  0, -5,  6, -1],
              [ 0, -3,  0, -1,  4]])

X = cp.Variable((n, n), diag=True)
constraints = [X >> L]
objective = cp.Minimize(cp.trace(X))
prob = cp.Problem(objective, constraints)
prob.solve()
print("\nThe optimal value is", 0.25 * prob.value)
```

The optimal value is 14.749999991267886

Our Goal

Lemma

$$Dual = \min_{Diag(\xi) \succcurlyeq L} \max_{x_i^2=1} x^T Diag(\xi) x = \min_{L_T \succcurlyeq L} \max_{x_i^2=1} x^T L_T x = TreeRel$$

where L_T can be represented as $L_T = L_{tree} + Diagonal$, where L_{tree} corresponds to Laplacian of a tree graph and Diag is a diagonal matrix with non-negative values.

Proved ✓

Current aim

$$H_k = \min_{\substack{T:\ T=T^\top \succcurlyeq L \\ tw(T) \leqslant k}} \max_{x_i^2=1} x^\top T x, \qquad OPT = H_k \leqslant \ldots \leqslant H_1$$

where optimization is taken over all graph with tree-width less than k. That is internal problem can be solved by dynamic programming.

Instead of insisting on *treewidth* $\leqslant k$ matrix T can be restricted for having less or equal than k diagonals.

$$D_k = \min_{\substack{T:\ T=T^\top \succcurlyeq L \\ T \text{ is } \leqslant k-\text{diagonal}}} \max_{x_i^2=1} x^\top T x \qquad \text{then} \quad H_k \leqslant D_k$$

Optimization for this problem can be performed using Derivative-free methods. We will work with Hill climbing algorithm.

Possible implementation

```
def hill_climbing(f, x0):
    x = x0  # initial solution
    while True:
        neighbors = generate_neighbors(x)  # generate neighbors of x
        # find the neighbor with the highest function value
        best_neighbor = max(neighbors, key=f)
        if f(best_neighbor) <= f(x):  # if the best neighbor is not better than x, stop
            return x
        x = best_neighbor  # otherwise, continue with the best neighbor
```

Example

```
from gradient_free_optimizers import
    HillClimbingOptimizer
def convex_function(pos_new):
    score = -(pos_new["x1"] * pos_new["x1"] + pos_new[
    "x2"] * pos_new["x2"])
    return score
def constraint_1(para):
    return para["x1"]**2 > 1
search_space = {
    "x1": np.arange(-10, 10, 0.1),
    "x2": np.arange(-10, 10, 0.1),
}
constraints_list = [constraint_1]
opt = HillClimbingOptimizer(search_space, constraints=
    constraints_list)
opt.search(convex_function, n_iter=100)
search_data = opt.search_data
print("\n search_data \n", search_data, "\n")
```

Results: 'convex-function'
Best score: -1.000000000000064
Best parameter:
'x1' : -1.000000000000032
'x2' : -3.552713678800501e-14

GeoGebra

References

[1] 0.878-approximation for the Max-Cut problem, Lecture by Divya Padmanabhanx‘
[2] Ryan O’Donnell CS Theory Toolkit at CMU, YouTube
[3] gradient-free-optimizers package in Python, GitHub
[4] Convex Optimization, Lieven Vandenberghe, Stephen Boyd, Stanford University

The End