Tree-width Driven SDP for The Max-Cut Problem

Ivan Voronin

Supervisor: Alexander Bulkin

Moscow Institute of Physics and Technology

May 23, 2024

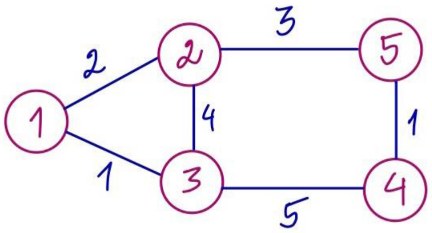
Outline

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 define the weight of this cut as the sum of the weights of edges which endpoints are contained in different parts

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

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



S	$W(S)$
$\varnothing$	0
$\{1,2,4\}$	$1+4+3+5+1=14$
$\{1,5,4\}$	$1+2+3+5=11$
$\{1,3,5\}$	$2+4+3+1+5=15$

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

SDP

Let us paraphrase the issue within the context of an optimization problem.
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 $\mathbf{x}$ represent the vector $\mathbf{x} = (x_1, \dots, x_n)^\top$ where $n = |V|$.

Now the Max-Cut can be represented as

$$\textbf{MaxCut} := \max_{y_{ij} \in \{-1,+1\}} \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - y_{ij}) =$$

$$= \max_{x_i^2 = 1} \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - x_i x_j) = \max_{\substack{X = \mathbf{x}\mathbf{x}^\top \\ X_{ii} = 1}} \frac{1}{4} \sum_{i \in V} \sum_{j \in V} w_{ij}(1 - X_{ij})$$

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:
   $\forall \mathbf{v} \in \mathbb{R}^n : \quad \mathbf{v}^\top X \mathbf{v} = \mathbf{v}^\top \mathbf{x}\mathbf{x}^\top \mathbf{v} = (\mathbf{x}^\top \mathbf{v})^\top (\mathbf{x}^\top \mathbf{v}) = (\mathbf{x}^\top \mathbf{v})^2$

Finally, let us consider the following problem

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

The difference between the **MaxCut** and **SDP** is as follows

	MaxCut	**SDP**
X	$X = \mathbf{x}\mathbf{x}^\top,\ \mathbf{x}\in\mathbb{R}^n$	$X = VV^\top,\ \mathbf{x}\in\mathbb{R}^{n\times m}$
X_{ij}	$X_{ij} = x_i x_j \in \{-1,+1\}$	X_{ij} is an arbitrary element

Hence **MaxCut** $\leqslant$ **SDP** (requirement $X = \mathbf{x}\mathbf{x}^\top$ is a particular case of $X = VV^\top$ therefore maximization in **MaxCut** is being performed on the subspace of constraint space of **SDP**)

```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(cp.sum(cp.multiply(W, (1-X))))
prob = cp.Problem(objective, constraints)
prob.solve()
print("The optimal value is ", 0.25*round(prob.value))
print("The solution is", X.value)
```

The optimal value is 15
The 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

Lemma

$$\textbf{MaxCut} = \frac{1}{4}\max_{x_i^2=1} \mathbf{x}^\top L\mathbf{x}, \text{ where } L \text{ is the Laplacian of the graph.}$$

$$\textbf{MaxCut} = \frac{1}{4}\max_{x_i^2=1} \mathbf{x}^\top L\mathbf{x} = \frac{1}{4}\max_{\boldsymbol{\lambda} \geqslant 0}\min_{\mathbf{x}} \left(\mathbf{x}^\top L\mathbf{x} + \sum_{i=1}^{n} \lambda_i(1-x_i^2)\right) =$$

$$= \frac{1}{4}\max_{\boldsymbol{\lambda} \geqslant 0}\min_{\mathbf{x}} \left(\mathbf{x}^\top (L - Diag(\boldsymbol{\lambda}))\mathbf{x} + \mathbf{1}^\top\boldsymbol{\lambda}\right) \leqslant$$

$$\leqslant \frac{1}{4}\min_{\boldsymbol{\lambda} \geqslant 0}\max_{\mathbf{x}} \left(\mathbf{x}^\top (L - Diag(\lambda))\mathbf{x} + \mathbf{1}^\top\boldsymbol{\lambda}\right) = \max_{\substack{Diag(\boldsymbol{\lambda})\succcurlyeq L \\ \boldsymbol{\lambda} \geqslant 0}} \frac{\mathbf{1}^\top\boldsymbol{\lambda}}{4} =: \textbf{Dual}$$

```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, 10, -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(0.25 * cp.trace(X))
prob = cp.Problem(objective, constraints)
prob.solve()
print("The optimal value is", round(prob.value))
print(" The solution is\n", X.value.toarray())
```

The optimal value is 15
The solution X is

$$\begin{pmatrix} 3.99998606 & 0 & 0 & 0 & 0 \\ 0 & 17.99998614 & 0 & 0 & 0 \\ 0 & 0 & 17.99998614 & 0 & 0 \\ 0 & 0 & 0 & 11.9999861 & 0 \\ 0 & 0 & 0 & 0 & 7.99998608 \end{pmatrix}$$

The question arises as to how to retrieve real cut from solution matrix X.

Lemma

$$\textbf{SDP} = \textbf{Dual}$$

k-diagonal extension

Instead of solving

$$\mathbf{Dual} = \min_{\substack{Diag(\boldsymbol{\lambda}) \succcurlyeq L \\ \boldsymbol{\lambda} \geqslant 0}} \frac{\mathbf{1}^\top \boldsymbol{\lambda}}{4} = \min_{\substack{Diag(\boldsymbol{\lambda}) \succcurlyeq L \\ \boldsymbol{\lambda} \geqslant 0}} \max_{x_i^2=1} \frac{\mathbf{x}^\top Diag(\boldsymbol{\lambda})\mathbf{x}}{4}$$

Let us expand the space of minimization to the k-diagonal matrices (symmetric matrices having at most $\frac{k-1}{2}$ leading diagonals on both sides of main diagonal)

$$\mathbf{D_k} := \min_{\substack{T:\ T=T^\top \succcurlyeq L \\ T \text{ is } k\text{-diagonal}}} \frac{1}{4} \max_{x_i^2=1} x^\top T x, \qquad k \underset{2}{\equiv} 1$$

Therefore,

$$\mathbf{SDP} = \mathbf{Dual} = \mathbf{D_1} \geqslant \mathbf{D_3} \geqslant \mathbf{D_5} \geqslant \dots \geqslant \mathbf{D_{2n-1}} = \mathbf{MaxCut}$$

Let us define an inner function as

$$f(T) := \frac{1}{4} \max_{x_i^2=1} x^\top T x$$

Hence, the aim is to solve

$$\min_{\substack{T:\ T=T^\top \succcurlyeq L \\ T \text{ is } k\text{-diagonal}}} f(T)$$

The obstacle here is that $f(T)$ is not easily differentiable. Let us now consider derivative-free methods.
Note that $f(T)$ is computable in $O(2^k \cdot n)$ time using dynamic programming, where T is a k-diagonal matrix. Therefore, optimization for this problem could be performed by using a polynomial-time oracle to compute $f(T)$ on each iteration.

Optimizers comparison

Let us now turn our attention to the nevergrad package in Python with ready-made implementations of a wide range of gradient-free optimizers. Visualization of some optimizers' work here.

1. CMA
2. Powell's method
3. TBPSA
4. TwoPointsDE

We have selected the Powell's method due to its numerous benefits.
The pertinent issue is how to derive a real cut from the optimal matrix T^* for $\mathbf{D_k}$. The following approach is suggested: we will run our dynamic oracle on T^* remembering all transitions that led to the optimal value.

k-treewidth

Another approach is to perform minimization among k-treewidth graphs (having a treewidth of at most k).

$$\mathbf{H_k} := \min_{\substack{T:\ T=T^\top \succcurlyeq L,\\ T \text{ is } k\text{-treewidth}}} \frac{1}{4} \max_{x_i^2=1} x^\top T x$$

Here again, inner function can be solved by dynamic programming. Therefore,

$$\mathbf{SDP} = \mathbf{Dual} = \mathbf{D_1} \geqslant \mathbf{H_1} \geqslant \mathbf{H_2} \geqslant \ldots \geqslant \mathbf{H_n} = \mathbf{D_{2n-1}} = \mathbf{MaxCut}$$

Moreover, $\mathbf{D_k} \geqslant \mathbf{H_k}, \quad k \underset{2}{\equiv} 1$

And again, the question arises as to how to derive a real cut from the optimal matrix T^* for $\mathbf{H_k}$.

This time we proceed with a similar to the k-diagonal case approach, with the exception of the new dynamic oracle.

Datasets

In our research we are using the Biq Mac Library

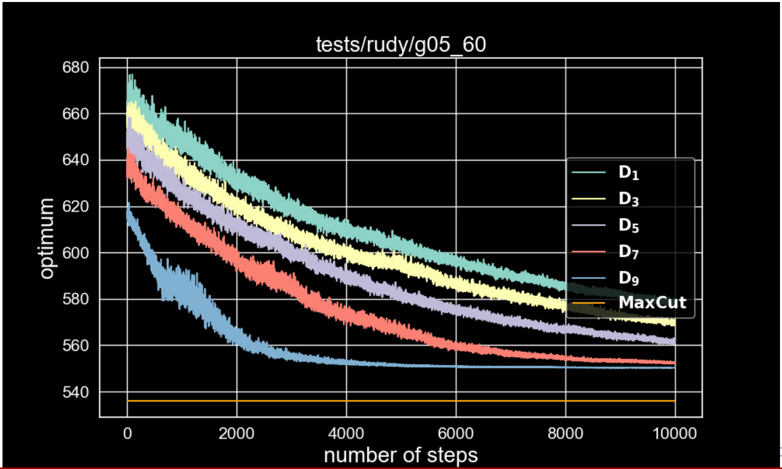
1. g05_60.i: For each dimension ten unweighted graphs with edge probability 0.5. n=60
2. g05_80.i: For each dimension ten unweighted graphs with edge probability 0.5. n=80
3. g05_100.i: For each dimension ten unweighted graphs with edge probability 0.5. n=100
4. pwd_100.i: For each density ten graphs with integer edge weights chosen from [0,10]
   and density d=0.1, 0.5, 0.9. n=100

Roadmap

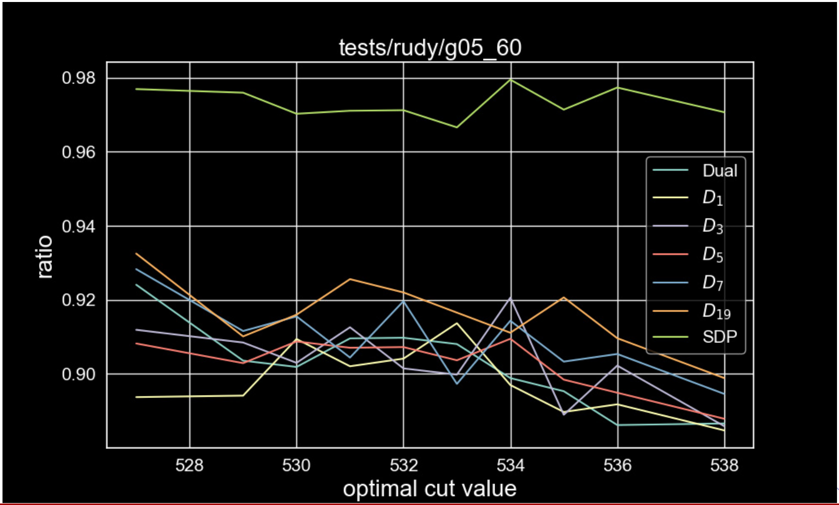
1. For each graph from dataset
   1.1 Use Goemans and Williamson approach to solve **SDP** and retrieve cut, then calculate ratio $\frac{\mathbf{SDP}^{cut}}{\mathbf{MaxCut}}$.
   1.2 For $k \in \{1, 3, 5, 7, 19\}$ run k-diagonal solver and retrieve cut from optimum for $\mathbf{D_k}$, then calculate ratio $\frac{\mathbf{D_k}^{cut}}{\mathbf{MaxCut}}$.
   1.3 Solve **Dual** and retrieve cut using 1-diagonal solver, then calculate ratio $\frac{\mathbf{Dual}^{cut}}{\mathbf{MaxCut}}$.
2. Examine error analysis on a graph of each type.
3. Compare the obtained results on the plots and in the table.

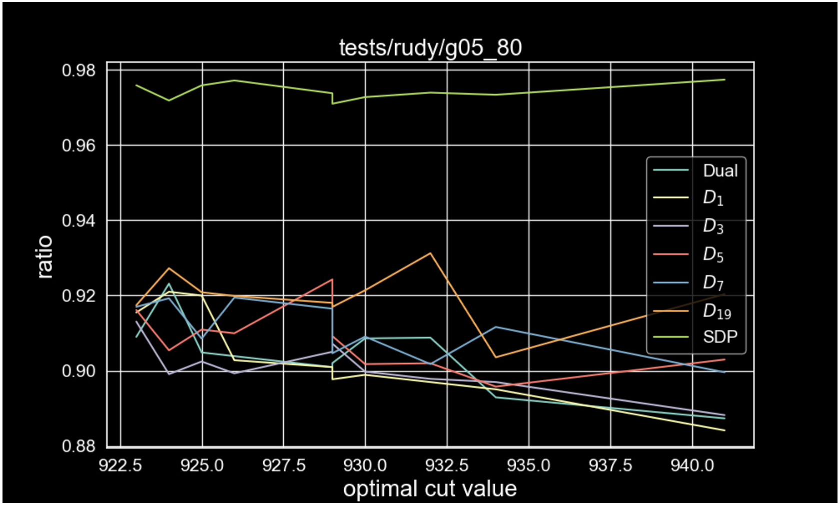
Error analysis

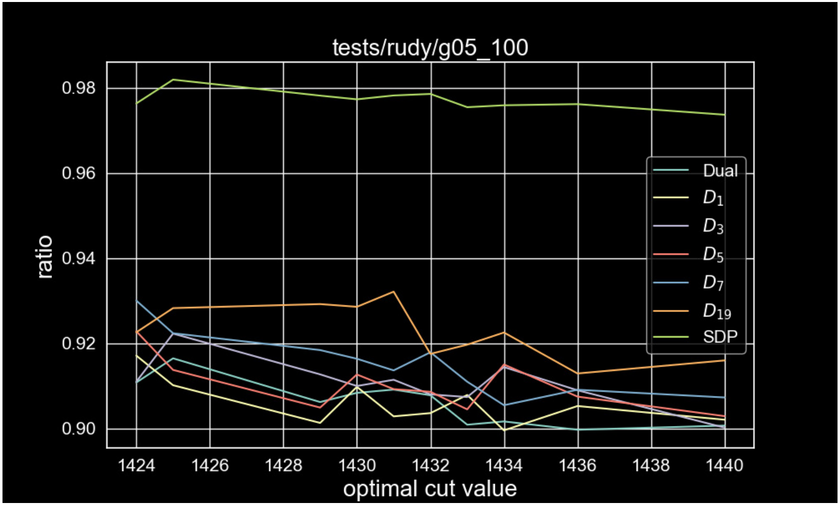
The optimizer demonstrates good convergence as the number of iterations increases.

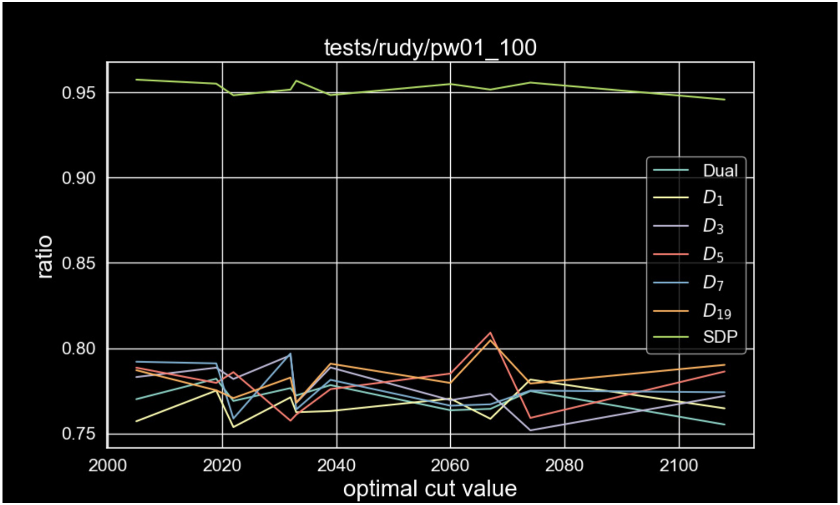


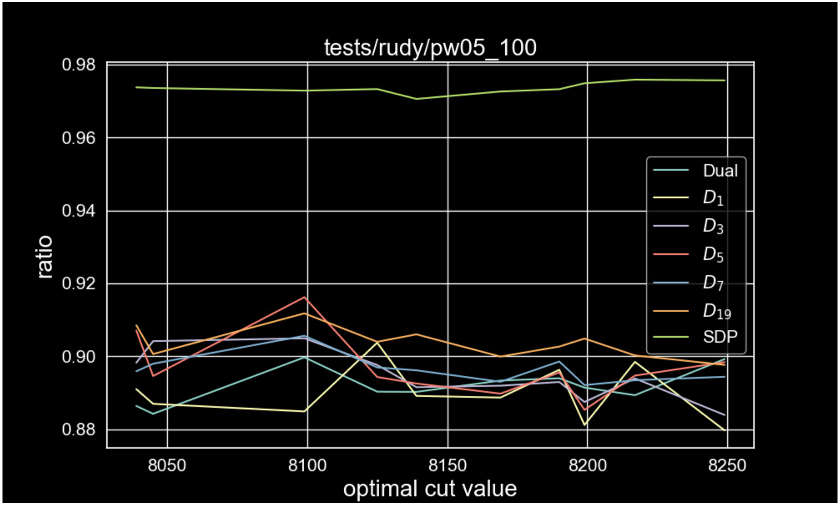
Results comparison

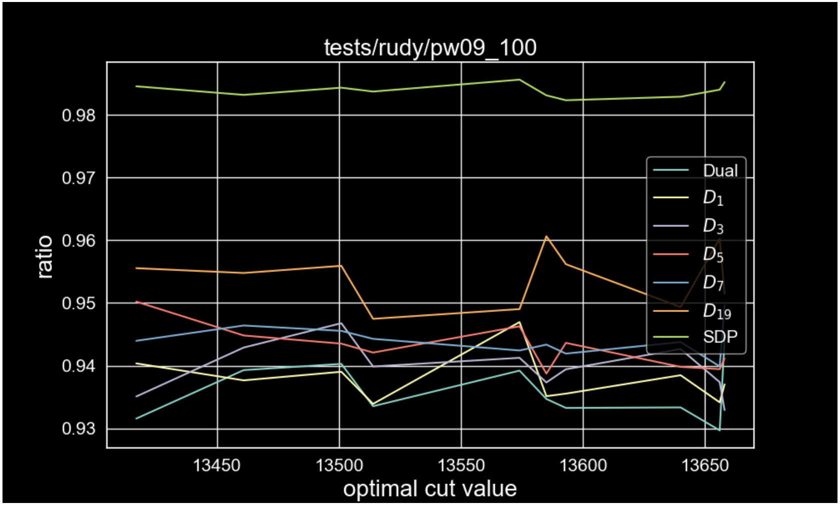



tests/rudy/g05_80
ratio
0.98
0.96
0.94
0.92
0.90
0.88
922.5
925.0
927.5
930.0
932.5
935.0
937.5
940.0
optimal cut value
Dual
D_1
D_3
D_5
D_7
D_{19}
SDP


tests/rudy/g05_100
ratio
0.98
0.96
0.94
0.92
0.90
1424
1426
1428
1430
1432
1434
1436
1438
1440
optimal cut value
Dual
D_1
D_3
D_5
D_7
D_{19}
SDP


tests/rudy/pw01_100
ratio
0.95
0.90
0.85
0.80
0.75
2000
2020
2040
2060
2080
2100
optimal cut value
Dual
D_1
D_3
D_5
D_7
D_{19}
SDP


tests/rudy/pw05_100
ratio
optimal cut value
0.98
0.96
0.94
0.92
0.90
0.88
8050
8100
8150
8200
8250
Dual
D_1
D_3
D_5
D_7
D_{19}
SDP


tests/rudy/pw09_100
ratio
0.98
0.97
0.96
0.95
0.94
0.93
13450
13500
13550
13600
13650
optimal cut value
Dual
D_1
D_3
D_5
D_7
D_{19}
SDP

Average ratios comparison

Test	**Dual**	**D_1**	**D_3**	**D_5**	**D_7**	**D_{19}**	**SDP**
g05-60	0.8951	0.8971	0.902	0.9055	0.9042	0.9187	0.9731
g05-80	0.9042	0.9033	0.9009	0.9079	0.9107	0.9197	0.9743
g05-100	0.9062	0.906	0.9107	0.9102	0.9152	0.923	0.9772
pw01-100	0.7706	0.7658	0.7772	0.7788	0.7767	0.7828	0.9525
pw05-100	0.8919	0.89	0.8947	0.8969	0.8964	0.9036	0.9736
pw09-100	0.9359	0.9378	0.9396	0.943	0.9441	0.9541	0.9839

Conclusion

In this paper, we have discussed a novel approach to solving the Max-Cut problem under specific constraints on the graph's Laplacian. Unfortunately, the implemented k-diagonal solver has not outperformed the **SDP**. In our opinion, the issue lies in the use of gradient-free optimizers which do not operate properly under the specified constraints.
Further research into this topic could be pursued in two directions. First, one could attempt to implement specific gradient-free technique tailored to the task. Apart from that, the solution of the k-treewidth case requires the implementation of a new oracle that solves $\mathbf{H_k}$, based on a polynomial tree decomposition.

The End