Maximum Cut: Learning-based approaches

Victor Pekkari
[epekkari@ucsd.edu]

University of California, San Diego

June 2025

Abstract

This paper explores deep learning approaches to the Maximum cut (max-cut) problem, motivated by its NP-hardness and the practical limits of classical approximations. While Goemans–Williamson (GW) delivers a 0.878 guarantee and is often near-optimal, it can still underperform on certain graph families. We ask three questions: (i) how to generate principled training data for neural solvers, (ii) which architectures work best in practice, and (iii) where, if anywhere, learned methods can outperform state-of-the-art polynomial-time approximations. We construct training sets solely via planting of optimal solution, and evaluate encoder–decoder models. In addition to fast, highly parallel inference, Many neural models can offer lower effective time complexity than SDP-based pipelines in practice. Our results suggest a trade-off: GW is uniformly strong across graph classes but follows a fixed rounding scheme, whereas learned models can adapt to specific graph families and capitalize on structure—providing a complementary tool where GW nears its worst case. This work is a meant as a step toward a recipe for learning-based max-cut solvers.

Code can be found at `https://github.com/victor99pekk/ML_4_MAXCUT`.

1 Introduction

The max-cut problem concerns partitioning the vertex set of a graph into two subsets so as to maximize the total weight of the edges crossing between them.

Definition 1.1 (Maximum Cut)**.** Let $G = (V, E, w)$ be an undirected weighted or unweighted graph with edge weights $w_{ij} \in \mathbb{R}$. For $S \subseteq V$, the *maximum cut value* is

$$\max_{S \subseteq V} \sum_{\{i,j\} \in E} w_{ij} \, \mathbf{1}_{[i \in S,\, j \notin S]}.$$

Since the Max-Cut problem is NP-hard, exact algorithms are computationally infeasible for large instances, and approximation methods become indispensable. This motivates the exploration of learning-based approaches. Neural networks have the potential to discover high-quality cuts more efficiently than classical state of the art approximation algorithms. However, before attempting to surpass sota approximation algorithms with learned models, it is essential to understand the practical performance of the approximation algorithms them self.

Geomanss & Williamson's algorithm for max-cut relaxes the max-cut problem into a semidefinite programming problem (SDP). Solving this relaxed problem and translating the solution from a set of unit vectors $\{X_i : ||X_i|| = 1\}$ into the set of the solution for the max-cut $x \in \{0, 1\}^n$. We can mathematically prove that we will achieve at worst a cut score of 0.878 of the max-cut. This algorithm comes closer to the optimum in practice however, frequently exceeding 99% of the optimal value. For some families of graphs it is even exact like for complete bipartite graphs, it is known to be exact [3]. Consequently, if GW were consistently near-optimal on all graphs, the motivation for neural approaches would be limited. Crucially however, there exist graph constructions where the performance of GW is closer to it's theoretical lower bound. These hard instances are the kinds of graphs where learning-based approaches can offer an advantage in max-cut value. Furthermore neural networks can be highly parallelized, making inference fast and scalable. This is also a potential edge over the GW algorithm. Therefore, even on graph families where GW is known to perform well, it is still worthwhile to explore learning-based methods, since they can achieve comparable or better performance at significantly lower computational cost, or faster inference.

2 Geomanss Williamson's and Feige Schectman graphs

2.1 Geomanns & Willimson's lower bound

The SDP relaxation not only gives us a lower bound of 0.878 but also the actual partition to achieve it. The lower bound and partition comes from maximizing the semidefinite problem below.

$$SDP(G) := \frac{1}{4} \max\{ \sum_{i,j=0}^{n} A_{ij}(1 - \langle X_i X_j \rangle) : X \in \mathbb{R}^n, ||X||_2 = 1 \text{ for all i}\}$$

To translate the unit sphere vectors from the SDP problem we used randomized rounding with a hyperplane.

$$g \sim \mathcal{N}(0, I_n), \qquad x_i = sign\langle X_i, g \rangle \quad \text{i = 1,2,..., n}$$

It is helpful to study the derivation of the GW lower bound, as knowing what type of assumptions and what features of the SDP probelm we use to derive the lower bound, can help us understand for what type of graphs the GW algorithm might be close to its lower bound.

Theorem 2.1 (0.878-approximation rounding for max-cut)**.** *Let G be a graph with adjacency matrix A, further let x_i be the randomized rounding for the solution X_i to the semidefinite program.*

$$\mathbb{E}\left[CUT(G,x)\right] \geq 0.878\, SDP(G) \geq 0.878\, MAX\text{-}CUT(G)$$

Proof.

$$\mathbb{E}\left[\mathrm{CUT}(G,x)\right] = \frac{1}{4}\sum_{i,j=1}^{n} A_{ij}(1 - \mathbb{E}x_i x_j)$$

By using the definitions of the x_i labels from the randomized rounding

$$\begin{aligned} 1 - \mathbb{E}\, x_i xj &= 1 - \mathrm{sign}\langle g, X_i\rangle \cdot \mathrm{sign}\langle g, X_j\rangle \\ &= 1 - \frac{2}{\pi}\arcsin\langle X_i, X_j\rangle \qquad \text{Grothendieck's identity} \\ &\geq 0.878\,(1 - \langle X_i, X_j\rangle) \qquad (*) \end{aligned}$$

This conclcudes the proof that $\mathbb{E}\left[\mathrm{CUT}(G,x)\right] \geq 0.878\,\mathbb{E}\left[\mathrm{SDP}(G)\right]$, The proof that $\mathbb{E}\left[\mathrm{SDP}(G)\right] \geq \mathbb{E}\left[\text{MAX-CUT}(G,x)\right]$ is trivial since max-cut operates over a subset of the domain of the semidefinite program.

An important observation from the proof is that $\langle X_i, X_j\rangle$ is seldom zero, meaning the true lower bound often exceeds 0.878. This further proves the claim that the GW algorithm most often performs close to the optimum.

2.2 Feige & Schechtman graphs

We concluded in the previous section that the SDP relaxation is most often predicted to perform better than the lower bound 0.878, something we did not note is that there is a family of graphs where the SDP relaxation performs close to the lower bound. This family of graphs is called Feige & Schechtman graphs (FS graphs).

Consider the last line of the proof of Theorem 2.1.

$$1 - \mathbb{E}\, x_i xj \geq 0.878\,(1 - \langle X_i, X_j\rangle) \qquad (*)$$

If we insert this into the fomula for the expected cut we get

$$\mathbb{E}\left[\mathrm{CUT}(G,x)\right] \geq \frac{1}{4}\sum_{i,j=1}^{n} A_{ij}(1 - 0.878\,(1 - \langle X_i, X_j\rangle)) = \mathbb{E}\left[\mathrm{SDP}(G)\right]$$

In order for the expected cut to approach the worst–case factor 0.878·MAX-CUT(G), we want the SDP vectors to be as decorrelated as possible: if $\langle X_i, X_j \rangle \approx 0$ for many pairs, the inequality $(*)$ is nearly tight and the guarantee stays close to 0.878. Note, however, that making vectors orthogonal does not force the rounded cut (or the SDP value) to equal $0.878 \cdot$ MAX-CUT(G); it only prevents the right–hand side of $(*)$ from being larger. Any nonzero correlations $\langle X_i, X_j \rangle \neq 0$ lift the bound above 0.878, so to keep the guarantee near 0.878 we need many inner products to be close to zero.

Definition 2.2. (Feige–Schechtman graphs)

Let $FS(n, \theta)$ denote a Feige–Schechtman graph with n nodes and angle threshold θ. Let each vertex v_i be a point uniformly chosen from the unit sphere S^{n-1}

$$v_i \sim S^{n-1}$$

Let vertices v_i, v_j be connected if there inner product is below a chosen threshold θ.

$$(i, j) \in E \quad \text{iff} \quad \langle v_i, v_j \rangle \leq \cos(\pi - \theta)$$

The edges can either be un-weighted or weighted with a decreasing function

$$f\left(\langle v_i, v_j \rangle\right) = A_{ij}$$

Something import to note about any graph and not just FS-graphs is that for the graph to have a non-trivial solution the graph cannot just contain nodes that are connected in pairs. The solution would then be trivial (separate the pairs). The reason this is worth noting in contact with the FS graphs is that as we decrease the angle threshold, we lower the chance of vertices being close to orthogonal and therefore connected, and even more so we decrease the chance of nodes being connected to more than one vertice, which makes we might converge to the trivial example of a sparse graph were vertices are connected to a low number of other vertices and then solution becomes trivial.

2.2.1 Density of FS graphs

It is in fact possible to prove that two random vectors on the unit sphere tend to be *almost orthogonal* in high dimensional space. This is very good as this implies that there are FS graphs that aren'r sparse and have a non-trivial solution, a potential family of graphs where neural networks might beat the GW algorithm in performance. To prove this *almost orthogonality* between random vectors on the unit sphere we use the proprties of isotropic vectors.

Definition 2.3. (Isotropic vectors)

A vector $X \in \mathbb{R}^n$ is isotropic iff

$$\mathbb{E}\langle x, X \rangle = ||x||_2^2 \quad \text{for all } x \in \mathbb{R}^n$$

Theorem 2.4. *let X, Y be two isotropic vectors sampled form the scaled unit sphere; $X, Y \sim$ Unif $\left(\sqrt{n}S^{n-1}\right)$*

$$|\langle X, Y\rangle| \sim \frac{1}{\sqrt{n}}$$

Proof.

$$\begin{aligned}\mathbb{E}\langle X, Y\rangle^2 &= \mathbb{E}_Y\, \mathbb{E}_X\left[\langle X, Y\rangle^2 | Y\right] \\ &= \mathbb{E}_X ||X||_2^2 \qquad \text{using definition of isotropic vectors} \\ &= n\end{aligned}$$

We already know by definition that

$$\mathbb{E}||X||_2^2, \quad \mathbb{E}||Y||_2^2 = \sqrt{n}$$

That concludes the proof that the vectors $X, Y \sim$ Unif $\left(\sqrt{n}S^{n-1}\right)$ are isotropic.

Now consider the normalization of the vectors sampled from Unif $\left(\sqrt{n}S^{n-1}\right)$ in theorem 2.4

$$X^* = \frac{X}{||X||_2}, \qquad Y^* = \frac{Y}{||Y||_2}$$

$$|\langle \frac{X}{||X||_2}, \frac{Y}{||Y||_2}\rangle| = \frac{1}{n}|\langle X, Y\rangle| \sim \frac{1}{\sqrt{n}}$$

Here we can see that the inner product of the two normalized vectors X^*, Y^* approaches 0 as n grows. Proving the *almost orthognality* between two random vectors on the unit sphere in high dimension.

2.2.2 Numerical experiment of *almost orthogonality* in high dimension

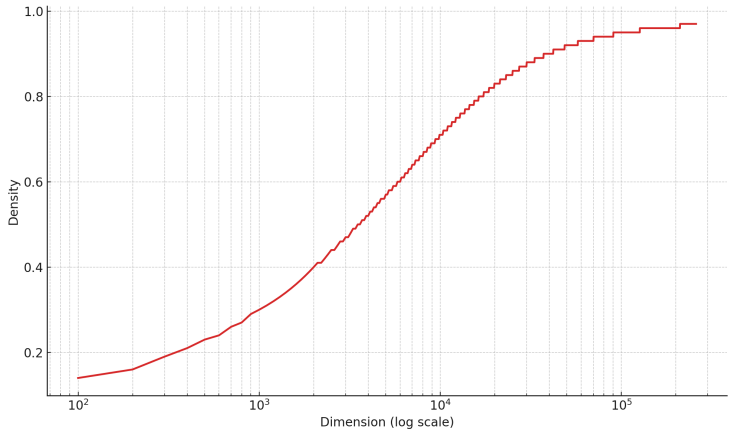

Figure 1: density as a function of log(dimension) for FS(50, 1°) graph

This was done in `python` by generating FS(50, 1°) graphs (50 vertices, angle threshold 1°) with vertices as vectors on the unit sphere, and only connecting them if the angle between them was in the range $(89°, 91°)$. The density was then calculated by the ratio of non-zero entries in the adjacency matrix.

$$\text{Density} = \frac{\#\{A_{ij} : A_{ij} \neq 0\}}{\#\{A_{ij} : A_{ij} = 0 \vee A_{ij} \neq 0\} - n}$$

where A is the adjacency matrix. n is the number of vertices in the graph, to account for the zero filled diagonal.

This empirical experiments shows that we can easily control the density of randomly generated FS graphs by adjusting the dimension from what the unit sphere vectors are sampled from.

3 Generating Training Data

A key challenge in applying supervised learning to max-cut is the lack of labeled data. Supervised learning (SL) methods require pairs of inputs (graphs) and outputs (optimal partitions), but computing optimal solutions is intractable for all but small instances due to the NP-hardness of Max-Cut. which makes training data generation a big problem for SL.

To address this, one approach is to *plant* optimal solutions, i.e., generate graphs from with a known optimal max-cut partition of vertices. This makes large-scale data generation feasible, though at the cost of reduced control over graph

structure. In particular, planted instances may not capture the special graph families where classical heuristics, such as the GW algorithm, perform poorly.

To target such harder cases, we also generate graphs without optimal labels, using instead the GW solution as the training signal. Our overall strategy is thus twofold: (i) pretrain the network in a supervised manner on GW-labeled data, and (ii) fine-tune it via reinforcement learning to surpass GW's performance.

All instances of graphs generated was created with `gen_maxcut_data.py`, where every generated row was one instance of a graph. The graphs were stored as a flattened adjacency matrix, followed by the optimal partition (a vector of 0's and 1's where the value at index i in this vector indicated whether node i was in partition 0 or 1), and lattice followed by the cut value from either the planted solutions or Geomans williamsons.

3.1 Planting solutions

We have two approaches to planting a solution, one of which involves reformulating an uncontrained boolean quadratic problem (ubqp) into the max-cut problem while keeping equivalence to the original ubqp. The other one involves projecting the solution of a relaxed ubqp to be optimal for a given adajcency matrix and partition of vertices.

3.1.1 Reformulating an unconstrained boolean quadratic problem

Our goal is to reformulate the problem below into the max-cut problem formulation, while keeping equivalence to the ubqp.

$$\min\{\, f(x) = x^\top Q x - c^\top x \;:\; x \in \{-1,1\}^n \,\}, \tag{1}$$

one obtains the following Lagrangian stationarity system [4]:

$$\begin{aligned} \text{find} \quad & Q,\, c,\, x,\, \lambda \\ \text{s.t.} \quad & (Q + \operatorname{diag}\lambda)\, x = c, \\ & Q + \operatorname{diag}\lambda \succ 0, \\ & x \in \{-1,1\}^n. \end{aligned} \tag{2}$$

Here $Q \in \mathbb{R}^{n\times n}$ is symmetric (possibly indefinite) and $c \in \mathbb{R}^n$. Choosing

$$\lambda_i \;\geq\; \sum_{j\neq i} |Q_{ij}| \quad (1 \leq i \leq n)$$

makes $Q + \operatorname{diag}\lambda$ diagonally dominant (hence positive semidefinite). Given any "planted" $x \in \{-1,1\}^n$, setting

$$c = (Q + \operatorname{diag}\lambda)\, x$$

enforces the stationarity equation and certifies x as an optimal solution of (1) under the standard Lagrangian argument.

Algorithm 1 maxcut data generator (while keeping optimality) [1]

Require: Dimension n; `base` $= 10$
Ensure: Matrix Q; Vector x

Generate an $n \times n$ matrix Q with i.i.d. entries from standard normal
$Q \leftarrow \texttt{base} \times Q$
$Q \leftarrow \frac{1}{2}(Q + Q^\top)$ ▷ make Q symmetric
Generate $x \in (0,1)^n$ uniformly at random
$x \leftarrow 2x - 1$
Compute $\lambda \leftarrow \sum \mathrm{abs}(Q)$ row-wise
Form Q' with λ on the diagonal and zeros elsewhere
$c \leftarrow (Q + Q') \cdot x$
Set $w_{0j}, w_{oj} \leftarrow \frac{1}{4}\left(\sum_{j=1}^{i-1} q_{ji} + \sum_{j=i+1}^{n} q_{ij}\right) + \frac{1}{2}c_i, \quad 1 \le j \le n$
Set $w_{ij} \leftarrow \frac{1}{4}q_{ij}, \quad 1 \le i < j \le n$
Update $Q \leftarrow \left(q_{1j}^\top, q_{2j}^\top, \ldots, q_{(n+1)j}^\top\right) \leftarrow \left(w_{0j}^\top, w_{1j}^\top, \ldots, w_{nj}^\top\right)$
Set $x_0 \leftarrow 1$ and update $x \leftarrow 2x + 1$
Update $x \leftarrow (x_1, x_2, \ldots, x_{n+1}) \leftarrow (x_0, x_1, \ldots, x_n)$

Flaw. This construction tends to yield weight/adjacency matrices with a disproportionately large first row and first column. Intuitively, the vector c aggregates row magnitudes of Q (via the diagonal–dominance choice of λ) and aligns with x. How disproportionate the final adjacency matrix becomes depends on the dimension of the adjacency matrix as well as the probability for x being part of group one since that is what our new augmented node will be. The graphs generated from Algorithm 1 becomes more and more disproportionate as the graph size increases.

The graphs represented by these adjacency matrix are *hub-like* with one vertice that edges that very large in magnitude compared to other edges between other vertices in the graph.

3.1.2 Magnitude inflation in the first row/column

Let n be the number of original vertices, and let the augmentation introduce a special vertex 0. For $i \in [n]$, the construction in Algorithm 1 produces

$$w_{0i} \;=\; \frac{1}{4}\sum_{\substack{j=1 \\ j\neq i}}^{n} q_{ij} \;+\; \frac{1}{2}\,c_i, \qquad w_{ij} \;=\; \tfrac{1}{4}\,q_{ij} \quad (i \neq j), \tag{3}$$

where $c = (Q + Q')x$ with $Q' = \mathrm{diag}(\lambda)$ and $\lambda_i \geq \sum_{j\neq i} |Q_{ij}|$ (diagonal dominance). We want to understand why entries in the first row/column $\{w_{0i}, w_{i0}\}$ tend to dominate the remaining w_{ij}.

Modelling assumptions. (used only for *expectation/upper bound* estimates):

(A1) *Off-diagonal homogeneity:* $|q_{ij}| \approx |q|$ for $i \neq j$ (write q for the typical magnitude).

(A2) *Tight diagonal dominance:* $\lambda_i \approx \sum_{j \neq i} |q_{ij}| \approx (n-1)|q|$.

(A3) *Labels as Bernoulli:* we analyse $x \in \{0,1\}^n$ with $\mathbb{E}[x_j] = p$ (we will set $p = \frac{1}{2}$ later) This makes sense since it just means a vertice has the same probability of belonging to either partition.

Lemma 3.1 (Expected scaling of the augmented weights)**.** *Under* (A1)–(A3) *with* $p = \frac{1}{2}$,

$$\mathbb{E}_x\big[c_i\big] = \sum_{j \neq i} q_{ij}\, \mathbb{E}_x[x_j] \;+\; \lambda_i\, \mathbb{E}_x[x_i] \;\approx\; \frac{n-1}{2}\, q \;+\; \frac{1}{2}\, \lambda_i, \tag{4}$$

$$\mathbb{E}_x\big[w_{0i}\big] \approx \frac{n-1}{4}\, q \;+\; \frac{1}{2}\, \mathbb{E}_x[c_i] \;\approx\; \frac{n-1}{2}\, q \;+\; \frac{1}{4}\, \lambda_i, \tag{5}$$

$$\mathbb{E}_x\big[|w_{0i}|\big] \;\lesssim\; \frac{3(n-1)}{4}\, |q|. \tag{6}$$

In contrast, for $i \neq j$, $|w_{ij}| = \frac{1}{4}|q_{ij}| \approx \frac{1}{4}|q|$.

Proof. Equation (4) is the definition of c_i. Insert (4) into (3) to obtain (5). For (6), apply the triangle inequality and use (A1)–(A2):

$$|w_{0i}| \;\leq\; \frac{1}{4} \sum_{j \neq i} |q_{ij}| + \frac{1}{2}\, |c_i| \;\leq\; \frac{n-1}{4}\, |q| + \frac{1}{2}\Big(\frac{n-1}{2}\, |q| + \frac{1}{2}\, \lambda_i\Big) \;\lesssim\; \frac{3(n-1)}{4}\, |q|.$$

□

Corollary 3.2 (First-row/column inflation factor)**.** *Comparing a typical augmented entry to a typical non-augmented one yields*

$$\frac{\mathbb{E}[|w_{0i}|]}{\mathbb{E}[|w_{ij}|]} \;\lesssim\; \frac{\frac{3(n-1)}{4}|q|}{\frac{1}{4}|q|} \;=\; 3(n-1).$$

Thus the first row/column can be up to $\mathcal{O}(n)$ *times larger in magnitude.*

It is convenient to summarise the upper-bound growth by the function

$$g(n) \;=\; 3(n-1) \;=\; |3n-3|,$$

which we use only as a *scale indicator* for the expected inflation.

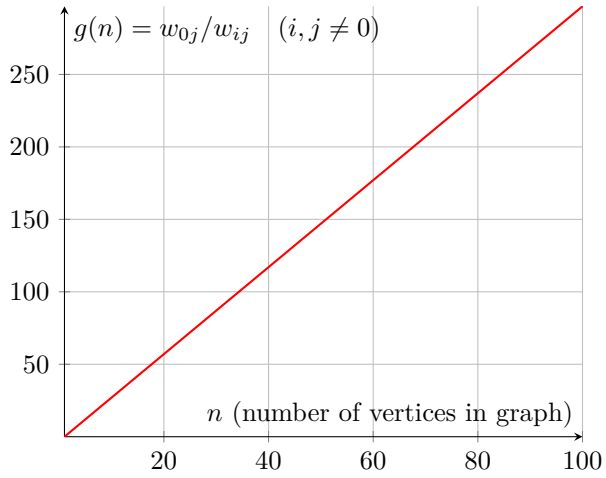

Figure 2: Upper-bound scaling $g(n)$ for the first row/column magnitude inflation.

3.1.3 Planting solutions by projecting

The unbalanced nature of the ubqp planting in section 3.1.1 is a problem if we want to train neworks with supervised learning to generalize to more "regular" graphs that dont have one vertice whose edge magnitudes are a lot larger than edge magnitudes between other vertices in the graph. For this we can use a projection planting instead. This method gives us a way to create training data that is more similar to graphs that we want our networks to generalize to.

To construct this projection we start with a formulation of the max-cut:

$$\max_{x \in \{0,1\}^n} \frac{1}{4} \sum_{i<j} W_{ij}(1 - x_i x_j) = C - \frac{1}{4} x^\top W x$$

If we instead consider the relaxed problem over the vector $[0, 1]^n$, we have a convex optimization problem.

Since the relaxed problem is a convex optimization problem we can guarantee that

$$Qx^* = 0 \quad \text{and} \quad Q \succeq 0 \implies x^* = argmin_{x \in [0,1]^n} \frac{1}{2} x^\top Q x$$

Furthermore we know that

$$\min_{x \in [0,1]^n} x^\top W x \leq \min_{x \in \{0,1\}^n} x^\top W x$$

This means that if we successfully plant a solution from the set $x^* \in \{0,1\}^n$ to the relaxed problem, then we know that it will also be the optimal partition for the maximum cut. We just have to construct an adjacency matrix that fulfills the conditions above.

Construction. Fix $x^\star \in \{\pm 1\}^n$, sample valid diag dominant adjacency matrix Q, set

$$P \;=\; I - \gamma\,\frac{x^\star (x^\star)^\top}{n}, \qquad W \;=\; P\,Q\,P, \quad Q \succeq 0,$$

Now consider the projection matrix P_{x^*} together with its corresponding $x^* \in \{0,1\}^n$, $Px^* = 0 \implies PQPx^* = 0$. The reason we multiply P on the left side is to keep the symmetric property that we need in a adjacency matrix.

$$PQPx^* \approx 0, \quad Q = Q^\top$$

We end by setting the diagonal $W = PQP$ to zero. This doesnt change the optimal partition or the max-cut value.

Algorithm 2 Projection-planting Max-Cut data

Require: $n \geq 0$;
Ensure: symmetric W; planted $x^\star \in \{\pm 1\}^n$
1: Sample $x^\star \sim \{\pm 1\}^n$
2: Sample $A \sim \mathbb{R}^{n\times n}$; set $Q \leftarrow AA^\top$
3: $P \leftarrow I_n - \frac{1}{n} x^\star (x^\star)^\top$ ($P^\top = P$, $Px^\star = 0$)
4: $W \leftarrow PQP$
5: $W_{ii} \leftarrow 0 \;\; \forall i$ (does not affect max-cut)
6: **return** W, $x^\star$

3.1.4 Control over Edges

To study how we can control the edge values in the adjacency matrix we start with some assumptions.

(A1) *Off-diagonal homogeneity:* $|q_{ij}| \approx |q|$ for $i \neq j$ (write q for the typical magnitude).

(A2) *Tight diagonal dominance:* $\lambda_i \approx \sum_{j\neq i} |q_{ij}| \approx (n-1)|q|$.

(A3) *Labels as Bernoulli:* we analyse $x \in \{0,1\}^n$ with $\mathbb{E}[x_j] = p$ (we will set $p = \frac{1}{2}$ below) This makes sense since it jsut means a vertice has the same probability to belong to either partition.

Lemma 3.3 (Expected scaling of the augmented weights)**.** *Under* (A1)–(A3) *with* $p_i = \frac{1}{2}$,

$$\mathbb{E}_x[P] = \frac{1}{n}\begin{bmatrix} n - p_1p_1 & p_1p_2 & \cdots & p_1p_n \\ p_2p_1 & n - p_2p_2 & \cdots & p_2p_n \\ \vdots & \vdots & \ddots & \vdots \\ p_np_1 & p_np_2 & \cdots & n - p_np_n \end{bmatrix} \tag{7}$$

$$Q\,\mathbb{E}_x[P] \leq \frac{1}{n}\begin{bmatrix} n\lambda + \sum_i q\,p^2 & \cdots & n\lambda + \sum_i q\,p^2 \\ \vdots & \ddots & \vdots \\ n\lambda + \sum_{i\neq 1} q\,p^2 & \cdots & n\lambda + \sum_{i\neq 1} q\,p^2 \end{bmatrix} \tag{8}$$

$$\leq \frac{1}{n}\begin{bmatrix} n\lambda + (n-1)q\,p^2 & \cdots & n\lambda + (n-1)q\,p^2 \\ \vdots & \ddots & \vdots \\ n\lambda + (n-1)q\,p^2 & \cdots & n\lambda + (n-1)q\,p^2 \end{bmatrix} \tag{9}$$

$$\mathbb{E}_x[P]\,Q\,\mathbb{E}_x[P] \leq \frac{1}{n^2}\begin{bmatrix} n\big[n\lambda + (n-1)q\,p^2\big] & \cdots & n\big[n\lambda + (n-1)q\,p^2\big] \\ \vdots & \ddots & \vdots \\ n\big[n\lambda + (n-1)q\,p^2\big] & \cdots & n\big[n\lambda + (n-1)q\,p^2\big] \end{bmatrix} \tag{10}$$

If we know analyze one single entry and use our assummptions that nodes have equal chance of belonging to group 0 or 1, and our assumptions of the magnitude proprtions between diagonal elements and the rest in the Q matrix.

$$w_{ij} \leq \frac{n(n-1)q + (n-1)q\,0.25}{n} = \frac{n^2q - nq + 0.25nq - 0.25q}{n}$$

$$\leq nq - 0.75q - \frac{0.25q}{n}$$

We can see here that this projection planting gives us a lot more control over the edge values, since first of all, they all have the same magnitude, and second of all they are directly dependent on the q entries, which we have direct control over, which means we have direct control over the magnitude of the entries in the final adjacency matrix W, we just have to find a fitting q-value that fits our dimension n.

3.2 Training data without planted solutions

The con with using the methods above for planting solutions is like we previously mentioned that we loose control over the graph structure. We loose the ability to guarantee statistical properties for the graph such as probability of nodes being connected and the edge-value distribution.

When generating training data without planted solutions, we instead rely completely on reinforcement learning (RL). This allows us to train our network to

maximize some function (the maximum cut function in our case) without having an explicit target.

Without the need to plant solutions, we have more control over the structure of the graphs which lets us train our networks on graphs that are actually hard for current the GW algorithms like FS graphs.

4 Neural network architectures

The neural architectures used in this study were initially inspired by the LSTM-based model proposed in [1]. Building on this foundation, we explored two main extensions aimed at improving performance. First, we introduced a graph neural network encoder to process the adjacency matrix more effectively by exploiting the underlying graph structure. Second, we replaced the LSTM cells with Transformer layers, enabling parallelized sequence processing and potentially better modeling of long-range dependencies between nodes.

4.1 Long short-term term memory

LSTM:s solve the problems of exploding/vanishing gradient commonly encountered in simpler RNNs by introducing a separate memory cell C_t alongside the hidden state h_t, and by using gating mechanisms to control what information is stored, forgotten, and outputed at each time-step. All inputs going into the activation functions in the gates are mulitplied by a learnable weight. The LSTM cell can this way efficiently learn what forget, remember anc carry forward by ajdusting it's weights during training.

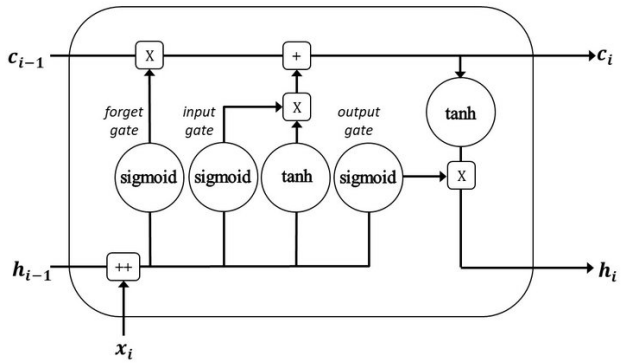

Figure 3: LSTM Cell

By explicitly maintaining the cell state C_t and using these gates, the LSTM can decide at each time step how much of the past information to carry forward (via

f_t), which new information to write (via i_t and $\tilde{C}_t$), and what to expose as the current hidden representation (via o_t).

$$
\begin{aligned}
f_t &= \sigma_g\left(W_f x_t + U_f h_{t-1} + b_f\right) \\
i_t &= \sigma_g\left(W_i x_t + U_i h_{t-1} + b_i\right) \\
o_t &= \sigma_g\left(W_o x_t + U_o h_{t-1} + b_o\right) \\
\tilde{c}_t &= \sigma_c\left(W_c x_t + U_c h_{t-1} + b_c\right) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

4.2 Transformers

Transformers further improve on the idea of having sequential input data, and generating sequentiel output data. Transformers removes the explicit recurrence altogether and instead rely on *self-attention* to let every position in the input sequence exchange information with every other position in a single, constant-depth layer. Each input token[1] $x_t \in \mathbb{R}^{d_{\text{model}}}$ is linearly projected to a *query*, *key*, and *value*. By representing the input-tokens with learnable key-, query- and value vectors, we can represent what a certain input-token is looking for (query), what it represents (key), and lastly what context it provides (value) at the same time.

$$
Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V, \qquad W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times d_k},
$$

and the pairwise compatibility between tokens is measured by the *scaled dot-product*

$$
\text{Attn}(Q, K, V) \;=\; \text{softmax}\big(\tfrac{QK^\top}{\sqrt{d_k}}\big)V.
$$

This produces a new representation in which each node is a weighted sum of *values* from every other node, with weights determined by how strongly its *query* matches their *keys*. By stacking several of these "attention heads'' in parallel (*multi-head attention*), followed by a position-wise feed-forward layer, a Transformer encoder layer can model long-range dependencies without the vanishing-gradient bottleneck that plagued vanilla RNNs and motivated LSTMs.

4.2.1 Benefits with transformer as opposed to LSTM

Compared with the LSTM encoder–decoder above, a Transformer offers two concrete advantages:

1. **Parallelism.** All nodes are processed simultaneously, cutting training time from $\mathcal{O}(n)$ per layer to $\mathcal{O}(1)$ on GPU/TPU hardware.

[1] In our Max-Cut setting a "token'' is a node, represented by its adjacency-row fingerprint introduced in the previous subsection.

2. **Global context in every layer.** Unlike an LSTM where each input can only interact with a summarization of past only past tokens and cannot interact with future inputs, a single encoder self-attention layer in a Transformer lets *every* token directly attend to *all* other tokens (past and future) in the sequence, and even the transformer decoder let's every input token directly interact with all past tokens. This full pairwise interaction makes it far easier to capture non-local structures[2]

4.3 Graph attention network

Graph attention networks (GATs) come from the idea of transformers. The idea is to update the vertice encodings based on it's neighbors. The intuition is that we this way can find a more accurate hidden represenation of the vertice than it´s original encoding.

4.3.1 Graph attention layers

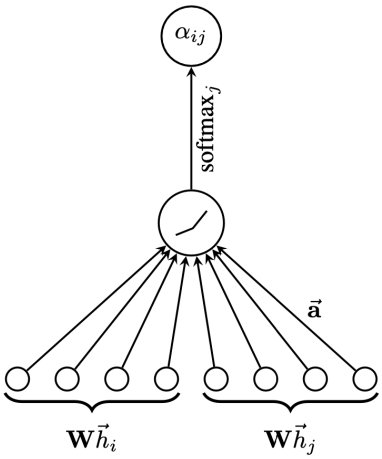

Figure 4: GAT architecture [2]

Consider the graph G vertices $v \in G$, where we denote the set of neighbors for a vertice as $\mathcal{N}(v)$. Every vertice has a feature vector $\vec{v} \in \mathbb{R}^N$ N is number of features, the set of feature vectors in the graph can be written as $\{\vec{v}|v \in G\}$. The graph attention layer produces a new set of vertice-feature vectors $\vec{v} \in \mathbb{R}^{N^*}$ where N^* not necessarily equals N.

We use a learnable weight matrix $\mathbf{W} \in \mathbb{R}^{F^* \times F}$ to project every vertice vector into the new dimension. We then perform self-attention by concatenating the new feature vectors for neighbouring vertices

[2]The context of other rows in the adjacency matrix is crucial for understanding the meaning of a row in our case

$$\mathbf{e}_{ij} = \begin{bmatrix} \mathbf{W}\,\vec{v}_i \\ \mathbf{W}\,\vec{v}_i \end{bmatrix}^{\top} \tilde{\mathbf{a}}\,, \qquad \mathbf{e}_{ij} \in \mathbb{R} \text{ denotes the importance of node j to node i}$$

Where $\tilde{\mathbf{a}} \in \mathbb{R}^{2 \cdot F^*}$ is a learnable vector, and nodes i and j are neighbors $v_i \in \mathcal{N}(v_j)$.

We lastly apply the leaky ReLu, followed by the softmax to complete the attention process

$$\alpha_{ij} = (\text{SoftMax}_i \circ \text{LReLU})\ (\mathbf{e}_{ij})$$

Last note. some graphs are dense and doing attention on all neighbors doesnt work. You can then slightly modify the criteria for doing attention from $[\, v_i \in \mathcal{N}(v_j)\,]$ to $[\, v_i \in \mathcal{N}(v_j) \quad \text{and} \quad (v_i, v_j) \geq \theta\,]$, where $\theta \in \mathbb{R}$ is an arbitrary threshold.

4.4 Complexity of inference | Transfomer and LSTM

The computational complexity and the inference time are two very important factors in this project since the whole purpose is to find a way to either find the maxcut faster than current approximation algorithms, or a more optimal cut.

Transformers: The dominating term in the timecomplexity for transformers occur in the self-attention, which consist of matrix mulitplications when we compute attention scores between all tokens. Calculating this consist of 2 matrix multiplications ($Q \cdot K^T$ followed by $Attn \cdot V$). The naive way for multiplying matrices has a complexity of $\mathcal{O}(n^3)$. You might think that this would mean that attention has a timecomplexity of $\mathcal{O}(n^3)$, but the timecomplexity is in fact (for one layer):

$$\mathcal{O}(n^2 \cdot d) \xrightarrow{\text{assuming} \quad n>d} \mathcal{O}(n^2)$$

(d is embedding size and n is the number of inputs)

This said, a GPU can drastically speed this up by calculating the parts of the matrix multiplications in parallel.

LSTM: The heaviest computations in the LSTM cell are the multiplications between the weight matrices and the input vectors, before we apply the sigmoid function $W \cdot x$. The dense products cost:

$$W_x \cdot x_t \implies \mathcal{O}(d_x d_h) \qquad W_h \cdot h_{t-1} \implies \mathcal{O}(d_h^2)$$

After removing all terms that aren't the heaviest we get the following timecomplexity for on LSTM cell for n timesteps:

$$\mathcal{O}(n(d_x d_h + d_h^2)$$

The matrix-vector multiplications can be speed up by a GPU the same was as the matrix multiplications in the transformer, but the linear increase with the input sequence length remains.

Conclusion Both models benefit substantially from GPU acceleration, yet their scaling differs. Transformers are quadratically dependent on sequence length but fully parallelisable, whereas LSTMs are linearly dependent on sequence length but sequential across time-steps. For long sequences on GPU-hardware, a Transformer layer is often faster in practice despite its higher $O(n^2)$ arithmetic cost. But if we do have a shorter input sequence it is possible that a LSTM model will be faster.

5 Encoder–Decoder Architecture

All models tested were encoder-decoder models. We tried the encoder decoder originally from Gu and Yang [1][3], and later tried improving it the model by implementing the vertice-encoder with a GAT, and the adjacency matrix encoder and decoder with a transformer. That said all models we used, implented the high level architecture below, what varied was how they implemented each block.

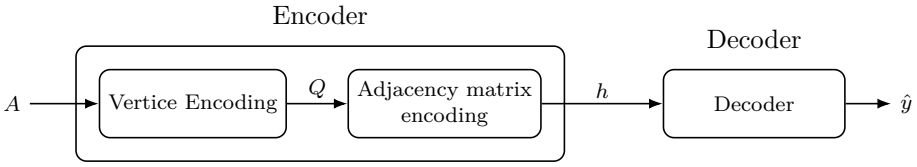

Figure 5: High-level architecture of all Neural models implemented

where $A \in \mathbb{R}^{n \times n}$ is the adjacency matrix, $Q \in R^{N \times N}$ is the new adjacency matrix that contains embedded vertices, $h \in \mathbb{R}^j$ is the encoding of Q, and $\hat{y}$ is the predicted partition. The input of all model implementations is a sequence of some rows from the adjacency matrix, and the output of all implementations is a sequence of vertices. This makes all models *Seq2Seq* models.

We implement the vertex encoder either with just a linear projection of the rows in the adjacency matrix. Or with GAT by computing attention scores between pairs of nodes, and then multiplying the edges in the adjacency matrix by the attention score between the vertices in question.

$$Q_{ij} = A_{ij} \cdot \alpha_{ij}$$

where Q is our new adjacency matrix, A is the old, and α_{ij} is the attention score between the two nodes.

[3]here the vertex encoding is a linear projection from row in adjacency matrix, adjacency encoding and decoder blocks are LSTM cells

5.1 Unfolding in time

Our models differ only in how the *encoder* consumes the adjacency sequence. The LSTM encoder processes rows autoregressively—one row at a time—so its hidden state h_t summarizes all past rows $(x_1, \ldots, x_t)$ when producing h_{t+1}. By contrast, the Transformer encoder is non-autoregressive: self-attention lets every row attend to all others in a single pass, yielding a parallel encoding of the entire matrix. In both architectures, the *decoder* is autoregressive: it generates the output sequence token by token (pointer selections), conditioning each step on the previously emitted tokens.

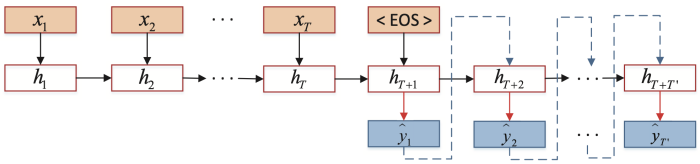

Figure 6: showing autoregressive encoding-decoding. from Gu and Yang paper [1]

5.1.1 Encoding

The input to our adjacency matrix encoder (a-encoder) is a sequence $x_{i:N} = x_1, x_2, ..x_N$, where x_i is the feature vector of vertice i in a graph with size N.

LSTM with the LSTM encoder we feed the whole input sequence $x_1, .., x_N$ and then use the last hidden state as the encoding for the adjacency matrix. This hidden state from the encoder will work as the starting state for the decoder.

Transfomer the transformer encoder works similarly to the LSTM. We feed the $x_1, .., x_N$ and use the last output as the encoding

5.1.2 Decoding

After feeding the model the start state it outputs an output sequence $y_{1:n} = y_1, y_2, .., y_n$ not necessarily the same length as the input. Our model is a pointer network, which means it will point to the index of the most probable next vertice in the partition of the previously outputed nodes.

The output at y_i is computed based on the idea of the conditional probability of

$$\max_{S(y),\ S(x)} \prod_{t=1}^{n} P(y_i \mid x_{j_1}, x_{j_2}, ..., x_{j_{i-1}})$$

where $S(y) = y_1, .., y_n$, $S(x) = x_1, .., x_N$, and all entries in the two sequences are unique.

The network can model this conditional probability (of belonging to the same partition as the already outputed vertices) as the attention score for every node i at timestep t

$$\text{Attention}_i(t) \approx P(y_i \mid x_{t_1}, x_{t_2}, ..., x_{t_{t-1}})$$

At every timestep we pick the vertice with the highest attention score (or highest conditional probability) after having masked vertices that we have already picked to avoid picking the same vertices several times. We then form one of the partitions for the max-cut problem by picking the first n^* vertices, this sequence is terminated by a learnable EOS output.

6 Experiments

Since our main goal with this project was to study possible benefits of using neural networks for solving maximum ut as opposed to using GW, we wanted to experiment with leaning-based approaches in areas of solving maximum ut where GW struggles. That is (i) inference speed, the maxcut is NP-hard and even though GW is just heuristically solving it, it can still be relatively slow, (ii) how much does further encoding of adajcency matrices with GAT help performance, (iii) which is better for max-cut LSTM or transformer

6.1 Neural network models

Training: The neural models were trained using the same Python script. The training was done on a T4 GPU by running the code on Google Colab. We trained each model for 15 minutes or until convergence due to limited GPU resources (google colab has a limit in the free tier). This early stopping is a flaw, as performance can plateau before improving or improve very slowly. So some of the runs may be undertrained.

Algorithm 3 Mini-batch SGD of pointer network

Input: training set $D = \{(x^{(n)}, y^{(n)})\}_{n=1}^{N}$; mini-batch size K; number of training steps L; learning rate α
Output: optimal W

random initial W;
repeat
  randomly reorder the samples in D;
  for $t = 1, \ldots, L$ **do**
    select samples $(x^{(n)}, y^{(n)})$ from the training set D;
    update parameters:
      $W_{t+1} \leftarrow W_t + \alpha\Big(\frac{1}{K}\sum_{n=1}^{N} x^{(n)}\big(y^{(n)} - \hat{y}_{W_t}^{(n)}\big)^{\top}\Big)$;
  end for
until the error rate of model $f(x; W)$ no longer decreases,
or the set time has passed

Inference: The test inference was done on a GPUs. We used Google Colab to access GPUs. We did not compile the models before we tested the inference.

6.2 Results

The experiments performance evaluation was done on 1000 graphs (with 70 vertices) created with projection planting. The models that were tested were (i) transformer implementation of the encoder-decoder, (ii) LSTM implementation of the encoder-decoder, (iii) and lastly another transformer implementation of the encoder-decoder, but this time also a graph attention network used to improve the encoding of the vertex encodings, instead of representing vertices by a (learnable) linear projection from it's row in the adjacency matrix (like we do in the other two models.

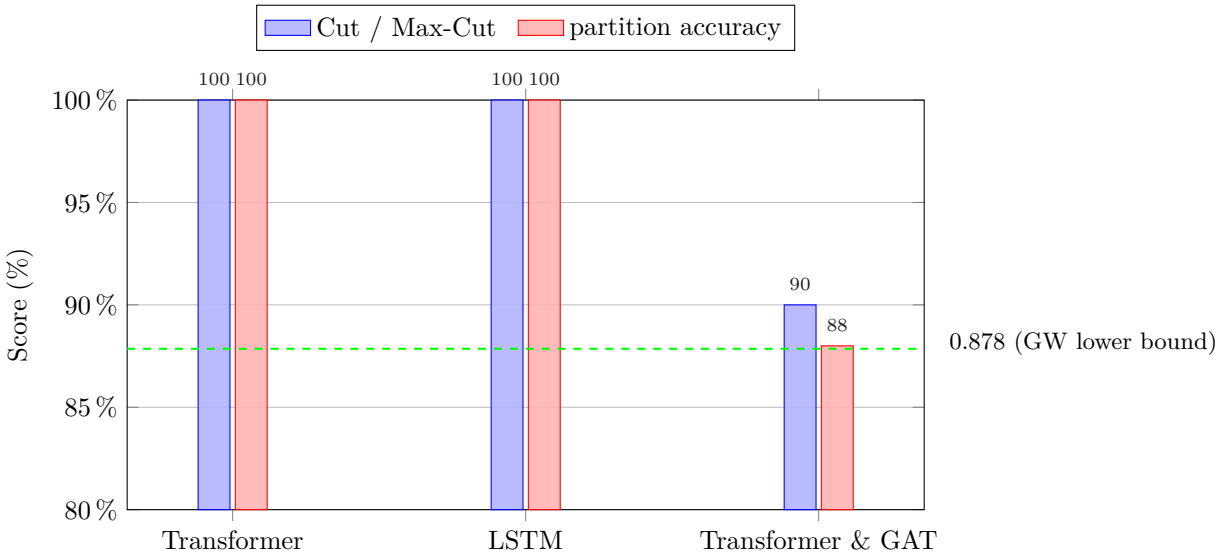

Figure 7: Comparison of three models with GW lower bound reference, on graphs with 70 vertices.

Partition accuracy. the accuracy for the partitions generated were defined by the *Hamming error code detection*.

Definition 6.1. Hamming error code metric as partition accuracy metric. let $y \in \{0,1\}^n$ be some optimal partition of vertices, and let $\hat{y} \in \{0,1\}^n$ be the optimal max-cut partition. We then define the partition accuracy as

$$\text{HamErr}(\hat{y}, y) = \frac{1}{n} \min\{||\hat{y} - y||_0, ||\hat{y} - (1-y)||_0\}$$

Cut optimality. We define the cut optimality as the summed cut over all 1000 graphs divided by the summed max-cut score obtained by the optimal planted solutions for all 1000 graphs.

7 Conclusion & Continued work

I didn't test the learning based models to their full extent. The aim was more to show that it is possible, and to lay the first foundation. The results clearly show that neural network has potential to compete with SDP as a way of solving max-cut problems.

Areas where Deep Learning has potential: We can see that the inference speed in the transformer is always the fastest, and gets faster compared to the other two as the graph size grows. If we can successfully scale and improve the transformers decent results on the small graphs we could get a good alternative to Goemans–Williamson if we need a fast estimation.

7.1 Improvements for Neural Networks

Several improvements could enhance the performance of the neural network models. Some obvious like hyperparameter optimization, and some less obvious like better (adjacency matrix) row encoding.

1. **More sophisticated row encoding** - We used either a linear projection from the row in the adjacency matrix, or a vertex encoding from a GAT network. One possible improvement could be to try and combine these instead of just using one. One could also consider using the pagerank algorithm as a way of encoding a vertex's importance by how many other nodes it is connected to, since that is what is important for the max-cut.
2. **Longer training duration** — especially for the Transformer, training for more epochs would likely benefit performance on larger graphs ($n = 20, 30, 50$).
3. **Larger training dataset** — generating more training would allow us to longer without overfitting.
4. **Better compilation and optimization of models**
5. **Weight pruning or compression** - inference speed
6. **Hyperparameter tuning** - performance

7.2 Continued work

Like mentioned earlier i didnt really push the models to their full capacity due to lack of GPU resources, it woudl be interesting how well these models could perform at larger graphs. Something that was never explored in this study

was how well these model can be trained with reinforcement learning. This is very interesting because of the high efficiency of thee GW algorithm for mot graphs, and because it then is hard to generate targets for training data for these graphs. just generating them and training with reinforcement learning could be interesting as it would let us train without targets. It would be really interesting to see how well the neural networks would perform on FS graphs compared to the GW algorithm.

References

[1] Shenshen Gu and Yue Yang. A deep learning algorithm for the max-cut problem based on pointer network structure with supervised learning and reinforcement learning strategies. *Mathematics*, 2020.

[2] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[3] E. A. Yıldırım and H. Wolkowicz. On the exactness of semidefinite relaxation for the maximum cut problem. *SIAM Journal on Optimization*, 12(2):350–366, 2001. Proves that Goemans–Williamson's SDP relaxation is exact for complete bipartite graphs $K(n, n)$ and graphs containing them as spanning subgraphs.

[4] Michael X. Zhou. A benchmark generator for boolean quadratic programming. *Unpublished manuscript / internal technical report*, 2024. Section 2, Lagrangian stationarity formulation and planted-solution construction. Available in `trainingData_paper.pdf`.