FAST AND PARALLEL ALGORITHM FOR NEURAL COMBINATORIAL OPTIMIZATION WITH CONSTRAINTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-supervised learning for combinatorial optimization offers a promising way to train neural solvers without requiring optimal solutions as supervision. This requires informative gradients while respecting discrete feasibility. Recent geometric approaches use Carathéodory decompositions to decompose a neural network output into a distribution over feasible solutions, but the resulting decomposition procedures can be sequential and computationally expensive. We propose a parallel alternative based on linear optimization oracles. A neural network generates multiple query directions, whose samples are passed independently through the oracle to produce feasible solutions in parallel. We then obtain a probability distribution over these feasible solutions through a regularized simplex-constrained quadratic program and use its KKT conditions to derive the gradients of the designed loss, without differentiating through the discrete oracle. The resulting method relies mainly on parallel oracle calls and GPU-friendly linear algebra. Experiments across constrained combinatorial optimization problems show that it produces high-quality feasible solutions while substantially reducing the computational cost of decomposition-based self-supervised learning.

1 INTRODUCTION

Neural combinatorial optimization (NCO) aims to learn algorithms for solving combinatorial optimization problems from distributions of problem instances. A central challenge is to design neural solvers that are both broadly applicable across different combinatorial structures and computationally competitive with specialized solvers and heuristics. While neural networks naturally benefit from massively parallel GPU computation, the surrounding optimization pipeline—including constraint handling, solution recovery, and other combinatorial subroutines—may not be equally well suited to the same hardware. As a result, these components can dominate runtime and limit the scalability of the overall solver. This motivates designing NCO pipelines whose training and solution-generation steps can both exploit GPU parallelism.

Scalability also concerns the cost of supervision. Neural methods typically learn from large collections of problem instances, and procuring high-quality solution labels for these instances can require expensive calls to combinatorial solvers. Self-supervised NCO offers an appealing direction because it uses the combinatorial objective itself as the training signal, avoiding the need to generate labels of optimal or near-optimal solutions. Combined with algorithms that exploit GPU parallelism, this paradigm offers a route to neural solvers that scale both computationally and with respect to the cost of supervision.

A central challenge in self-supervised constrained NCO is how to incorporate discrete feasibility constraints into a differentiable learning pipeline. A common approach is to optimize a continuous surrogate while penalizing constraint violations in the loss. Other approaches rely on continuous relaxations, differentiable projections, or problem-specific layers that map the network output toward the feasible region (Karalias & Loukas, 2020; Wang et al., 2022a; Donti et al., 2021; Wang et al., 2023; Zeng et al., 2024). While effective in many settings, these methods may require problem-specific loss design, additional hyperparameter tuning, or a separate rounding or solution-recovery procedure to obtain a feasible discrete solution. Thus, the choices made for constraint handling, loss computation, and solution recovery can substantially affect the efficiency and utility of the overall solver.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

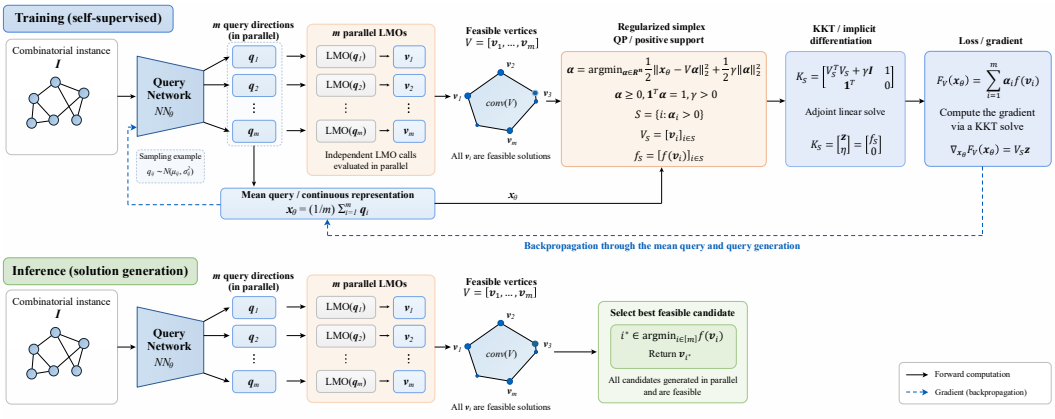

Figure 1: Overview of our approach. The query network produces m query directions, and independent LMOs generate feasible vertices V in parallel. During training, the mean query $\mathbf{x}_\theta$ is mapped to a convex combination of V via the regularized simplex QP and differentiated through its KKT system. At inference, we return the best feasible vertex among the m candidates.

A recent line of work connects these components through *extensions* of the discrete objective constructed from the geometry of the feasible set (Karalias et al., 2022; 2025; Nerem et al., 2025; Rafiey et al., 2026). These methods represent, or approximate, a continuous point using weighted combinations of feasible discrete solutions. The resulting weights define a distribution over feasible solutions and induce a continuous surrogate through the expected value of the discrete objective. Under suitable regularity conditions, this surrogate can be differentiated and used directly for self-supervised training. Crucially, the same decomposition also produces feasible candidates for discrete inference. Extensions are therefore algorithmically appealing because a single geometric construction addresses both loss design and solution recovery. However, existing geometric and permutation-based constructions generate these feasible solutions and distributions through sequential decomposition procedures, creating a computational bottleneck shared by training and inference.

For constrained NCO, Karalias et al. (2025) develop a general framework based on geometric decompositions of the feasible-set polytope. Their generic construction builds on the classical Grötschel–Lovász–Schrijver (GLS) framework, while more efficient variants exploit problem-specific structure. For permutation problems, Nerem et al. (2025) construct a related extension through successive decompositions over the Birkhoff polytope. More recently, Rafiey et al. (2026) propose a simpler oracle-based Frank–Wolfe decomposition that requires only one linear optimization oracle (a.k.a LMO) call per iteration and applies across a broad class of polytopes. Despite these improvements, all of these constructions remain sequential: the feasible solution generated at one step requires calling a combinatorial LMO and influences the point or residual used to generate the next.

This sequential dependence can become a computational bottleneck in neural optimization pipelines. Even when an individual oracle call is efficient, the calls cannot be executed simultaneously, and repeated CPU–GPU communication can introduce additional overhead. Since the same decomposition is used during both training and inference, removing this dependency could improve both stages. This motivates our central question: *Can we design parallel decomposition algorithms for self-supervised constrained NCO that make both loss computation and rounding efficient on GPUs?* More specifically, can we replace long sequences of dependent oracle calls with a small number of batched computations that generate multiple feasible candidates in parallel, while retaining feasibility, differentiability, and the principled connection between continuous training and discrete inference provided by geometric extensions?

Our contributions. We address this bottleneck through parallel oracle-based candidate generation, illustrated in Figure 1. Given a problem instance, a neural network generates multiple query directions, and independent oracle calls map these directions to feasible combinatorial solutions. During training, a geometric extension over the generated candidates turns their objective values into a gradient for updating the network. At inference, the same candidate-generation procedure is used, and we simply return the candidate with the best objective value. Thus, feasibility is enforced by the oracle, while training and inference share a parallel solution-generation mechanism. The framework accommodates

both exact and approximate linear optimization oracles. Its central technical contribution is a regularized geometric loss whose gradient is derived from the associated KKT conditions, enabling training without differentiating through the discrete oracle. We evaluate the framework on Maximum Coverage, Quadratic Assignment, MaxCut, and Densest k-Subgraph, demonstrating competitive solution quality and low inference cost across different combinatorial structures, including graphs with millions of vertices.

2 Problem Definition and Learning Setup

We consider combinatorial optimization (CO) problems over a finite feasible set $\mathcal{C} \subseteq D^n$, where D is a finite discrete domain. A feasible solution $\mathbf{x} \in \mathcal{C}$ assigns a value in D to each of the n decision variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$, and is evaluated by an objective $f : \mathcal{C} \to \mathbb{R}$. We write the problem as

$$\max f(\mathbf{x}) \quad \text{s.t.} \quad \mathbf{x} \in \mathcal{C}. \tag{1}$$

We associate $\mathcal{C}$ with its combinatorial polytope $\mathcal{P} = \mathrm{conv}(\mathcal{C})$, i.e., the convex hull of the feasible solutions. Although $\mathcal{P}$ may have exponentially many constraints or vertices, many important CO problems admit efficient linear optimization over $\mathcal{P}$. We assume access to an efficient Linear Maximization Oracle (LMO). Given a query vector $\mathbf{q} \in \mathbb{R}^n$, an exact LMO returns a feasible solution

$$\mathbf{v}^\star \in \arg\max_{\mathbf{v} \in \mathcal{C}} \langle \mathbf{q}, \mathbf{v} \rangle,$$

while an approximate LMO returns a feasible solution that approximately maximizes the same linear objective. Many important combinatorial polytopes admit efficient LMOs. Examples include the hypersimplex, spanning-tree polytopes, and, more generally, matroid polytopes, and the Birkhoff polytope for which exact deterministic LMOs can be implemented using standard combinatorial algorithms. Our framework also allows approximate LMOs when exact linear optimization is unavailable or computationally expensive.

Self-Supervised CO and training. Given a problem instance $\mathcal{I}$ with input features $\mathbf{Z}_\mathcal{I}$, we use a neural network NN_θ to generate m distributions over query directions $\mathbf{q}_1, \ldots, \mathbf{q}_m \in \mathbb{R}^n$. We sample the queries and apply the LMO independently to obtain feasible solutions $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathcal{C}$. Hence, unlike standard neural CO approaches, the network is not required to directly predict a feasible solution, and no rounding step is needed to obtain the generated candidates.

Training is self-supervised: no optimal solutions are provided as labels. Instead, we evaluate the combinatorial objective on the generated feasible solutions and construct a differentiable loss of these values. In particular, we define the interpolation point $\mathbf{x} = \frac{1}{k} \sum_{i=1}^{m} \mathbf{q}_i$ and associate it with a distribution over $\mathbf{v}_1, \ldots, \mathbf{v}_m$. The resulting self-supervise loss provides a gradient that is backpropagated through the sampled queries to train NN_θ. At inference time, the same parallel LMO procedure directly produces feasible discrete solutions, from which the best candidate can be selected according to the objective.

3 Method

We introduce a parallel alternative to sequential decomposition-based extensions. The main idea is simple: rather than constructing feasible vertices one at a time, we generate a collection of query directions in parallel and solve the corresponding linear maximization problems simultaneously. The resulting feasible vertices define a local surrogate extension around the current neural network output. Importantly, training does not require explicitly computing the convex-combination coefficients: we obtain the gradient of the loss directly through implicit differentiation.

Representation and parallel query generation. Let $\mathcal{I}$ an an instance of the CO problem. We first generate m query directions $\mathbf{q}_1, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ using a neural network and define $\mathbf{x}_\theta = \frac{1}{m} \sum_{i=1}^{m} \mathbf{q}_i \in \mathbb{R}^n$ to be the continuous representation of the instance $\mathcal{I}$ produced by the neural model. (More generally, any differentiable aggregation of the query directions could be used.) The queries are generated jointly and therefore do not depend sequentially on one another. There are several possible parameterizations for the query directions. Importantly, each query $\mathbf{q}_i$ must have the same dimension as a feasible solution, i.e., $\mathbf{q}_i \in \mathbb{R}^n$ when $\mathcal{C} \subseteq \mathbb{R}^n$.

For example, for graph problems, one implementation is to associate m Gaussian distributions with each coordinate of the feasible solution. For instance, when a feasible solution is represented by one coordinate per graph vertex, the neural network outputs, for every vertex $j \in [n]$,

$(\mu_{1j}, \sigma_{1j}), \ldots, (\mu_{kj}, \sigma_{kj})$. We then sample

$$q_{ij} = \mu_{ij} + \sigma_{ij}\epsilon_{ij}, \qquad \epsilon_{ij} \sim \mathcal{N}(0,1), \tag{2}$$

and form the ith query as $\mathbf{q}_i = (q_{i1}, \ldots, q_{in}) \in \mathbb{R}^n$, for all $i \in [m]$. Thus, the network parameterizes m distributions over directions in the same space as the feasible solutions, and all m queries can be sampled in parallel.

Alternatively, the network can directly output $(\mathbf{q}_1, \ldots, \mathbf{q}_m) = \mathrm{NN}_\theta(\mathcal{I})$, without introducing a stochastic parameterization. Our framework is agnostic to the particular parameterization; the important property is that all m query directions are generated simultaneously and can therefore be passed to the LMO in parallel. This provides a stochastic mechanism for exploring different regions of the feasible polytope.

Parallel linear maximization and stability. For every query q_i, we solve use the deterministic LMO to obtain $\mathbf{v}_i \in \mathcal{C}$ that (approximatly) maximizes $\langle q_i, v \rangle$. These m LMO calls are independent and can therefore be evaluated in parallel. This is the main computational difference from sequential decomposition methods. In those methods, the query at iteration $t+1$ depends on the vertex returned at iteration t. Here, all $\mathbf{q}_i$ are available before any LMO is evaluated, and all $\mathbf{v}_i$ can consequently be computed in one batched parallel stage. We collect the resulting feasible vertices into the matrix

$$V := [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_m] \in \mathbb{R}^{n \times m}.$$

Although the query directions $\mathbf{q}_1, \ldots, \mathbf{q}_m$ vary continuously with the neural network output, the feasible solutions selected by the oracle need not change with every small perturbation. For example, a top-k oracle returns the same set as long as the perturbation does not cause a selected element to fall below an unselected one. Thus, the generated vertex matrix V can remain fixed even as the queries and their mean $\mathbf{x}_\theta$ change.

We assume that the oracle, together with its fixed tie-breaking rule, has this local stability for almost every collection of query directions: sufficiently small perturbations leave all selected vertices unchanged. The exceptional inputs, where an arbitrarily small perturbation can change a selected vertex, are assumed to form a set of Lebesgue measure zero. This property holds for many combinatorial oracles: top-k selection, standard matroid greedy algorithms such as Kruskal's algorithm, and exact assignment or matching oracles with fixed tie-breaking.

The loss over the generated vertices. The generated vertices define the restricted polytope $\mathcal{P}_V := \mathrm{conv}\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$. A natural way to associate $\mathbf{x}_\theta$ with a distribution over these vertices is to project $\mathbf{x}_\theta$ onto $\mathcal{P}_V$. In particular, define

$$\boldsymbol{\alpha}(\mathbf{x}_\theta) \in \operatorname*{arg\,min}_{\boldsymbol{\alpha} \in \mathbb{R}^m} \quad \frac{1}{2}\|\mathbf{x}_\theta - V\boldsymbol{\alpha}\|_2^2 \qquad \text{s.t.} \qquad \mathbf{1}^\top \boldsymbol{\alpha} = 1, \;\; \boldsymbol{\alpha} \geq 0. \tag{3}$$

The coefficients $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ define a distribution over the generated feasible solutions. This would naturally induce the expected loss defined as

$$F_V(\mathbf{x}_\theta) := \sum_{i=1}^{m} \boldsymbol{\alpha}_i(\mathbf{x}_\theta) f(\mathbf{v}_i). \tag{4}$$

To support gradient-based training, we seek a surrogate loss F_V that is differentiable almost everywhere (a.e.) with respect to $\mathbf{x}_\theta$. Next, we establish this and then derive a direct procedure for computing the gradient from the associated KKT conditions.

Computing the gradient directly. The key observation is that, once the generated vertices V are fixed followed by our oracle stability assumption, the coefficients $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ are locally differentiable functions of the continuous point $\mathbf{x}_\theta$. Therefore, to optimize $F_V(\mathbf{x}_\theta) := \sum_{i=1}^m \boldsymbol{\alpha}_i(\mathbf{x}_\theta) f(\mathbf{v}_i)$ it is sufficient to compute the Jacobian $J(\mathbf{x}_\theta) = \partial \boldsymbol{\alpha}(\mathbf{x}_\theta)/\partial \mathbf{x}_\theta$.

The generated vertices $\mathbf{v}_1, \ldots, \mathbf{v}_m$ need not be affinely independent, and therefore the coefficients representing a point in their convex hull need not be unique. This breaks differentiability. To obtain a unique distribution, we add a small quadratic regularizer with $\gamma > 0$ and define

$$\boldsymbol{\alpha}(\mathbf{x}_\theta) := \operatorname*{arg\,min}_{\boldsymbol{\alpha} \in \mathbb{R}^m} \frac{1}{2}\|\mathbf{x}_\theta - V\boldsymbol{\alpha}\|_2^2 + \frac{\gamma}{2}\|\boldsymbol{\alpha}\|_2^2 \qquad \text{s.t.} \qquad \mathbf{1}^\top \boldsymbol{\alpha} = 1, \;\; \boldsymbol{\alpha} \geq 0. \tag{5}$$

Since the Hessian with respect to $\boldsymbol{\alpha}$ is $V^\top V + \gamma I \succ 0$ (a positive definite matrix), the objective is strongly convex and (5) admits a unique solution. Moreover, $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ lies on the probability simplex and therefore defines a distribution over the generated feasible solutions.

To differentiate the coefficients $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ with respect to $\mathbf{x}_\theta$, we first need a set of equations that characterizes the optimal solution of (5). For constrained optimization problems, these are given by the Karush–Kuhn–Tucker (KKT) optimality conditions, which generalize the usual condition of setting the gradient of an unconstrained objective to zero. The constraint $\mathbf{1}^\top \boldsymbol{\alpha} = 1$ is associated with a scalar Lagrange multiplier λ, while each nonnegativity constraint $\boldsymbol{\alpha}_i \geq 0$ has a multiplier $\boldsymbol{\mu}_i \geq 0$. Since the optimal coefficients depend on $\mathbf{x}_\theta$, these multipliers generally depend on $\mathbf{x}_\theta$ as well. The stationarity condition for (5) is $(V^\top V + \gamma I)\boldsymbol{\alpha} - V^\top \mathbf{x}_\theta + \lambda \mathbf{1} - \boldsymbol{\mu} = 0$. Together with the original constraints, the KKT conditions additionally require complementary slackness, $\boldsymbol{\mu}_i \boldsymbol{\alpha}_i = 0$ for every i. This last condition has a simple interpretation. If a coefficient is strictly positive, $\boldsymbol{\alpha}_i > 0$, then its nonnegativity constraint is not active and therefore $\boldsymbol{\mu}_i = 0$. Let $S = S(\mathbf{x}_\theta) := \{i : \alpha_i(\mathbf{x}_\theta) > 0\}$ denote the set of such coefficients, and let $V_S = [\mathbf{v}_i]_{i \in S}$. On a region where this support remains unchanged, coefficients outside S remain zero, while the positive coefficients satisfy

$$\underbrace{\begin{bmatrix} V_S^\top V_S + \gamma I & \mathbf{1} \\ \mathbf{1}^\top & 0 \end{bmatrix}}_{K_S} \begin{bmatrix} \boldsymbol{\alpha}_S(\mathbf{x}_\theta) \\ \lambda \end{bmatrix} = \begin{bmatrix} V_S^\top \mathbf{x}_\theta \\ 1 \end{bmatrix}. \tag{6}$$

The first row of (6) is the stationarity condition restricted to the positive coefficients, while the second row enforces $\mathbf{1}^\top \boldsymbol{\alpha}_S = 1$. Thus, once the support S is fixed, the optimal coefficients are characterized by a linear system whose right-hand side depends linearly on $\mathbf{x}_\theta$. This form will allow us to differentiate $\boldsymbol{\alpha}_S(\mathbf{x}_\theta)$ implicitly without differentiating through the optimization procedure itself.

Because $\gamma > 0$, K_S is nonsingular. Differentiating (6) with respect to $\mathbf{x}_\theta$ gives

$$\begin{bmatrix} \dfrac{\partial \boldsymbol{\alpha}_S(\mathbf{x}_\theta)}{\partial \mathbf{x}_\theta} \\ \dfrac{\partial \lambda}{\partial \mathbf{x}_\theta} \end{bmatrix} = K_S^{-1} \begin{bmatrix} V_S^\top \\ 0 \end{bmatrix}. \tag{7}$$

The derivative of the Lagrange multiplier $\partial\lambda/\partial\mathbf{x}_\theta$ is only an auxiliary quantity; we retain the first block of the solution, which gives $J_S = \partial\boldsymbol{\alpha}_S(\mathbf{x}_\theta)/\partial\mathbf{x}_\theta \in \mathbb{R}^{|S| \times n}$. Let $f_S = (f(\mathbf{v}_i))_{i \in S}$ collect the objective values of the vertices with positive probability. The resulting loss is $F_V(\mathbf{x}_\theta) = \boldsymbol{\alpha}(\mathbf{x}_\theta)^\top f$, and within a region where the support and generated vertices remain fixed,

$$\nabla_{\mathbf{x}_\theta} F_V(\mathbf{x}_\theta) = J_S^\top f_S. \tag{8}$$

Importantly, we do not need to explicitly construct the Jacobian J_S. From (7), $J_S = [I \quad 0]\, K_S^{-1} \begin{bmatrix} V_S^\top \\ 0 \end{bmatrix}$. Substituting this expression into (8) gives $\nabla_{\mathbf{x}_\theta} F_V(\mathbf{x}_\theta) = [V_S \quad 0]\, K_S^{-1} \begin{bmatrix} f_S \\ 0 \end{bmatrix}$. Hence, rather than forming K_S^{-1} or J_S, we solve the linear system

$$K_S \begin{bmatrix} \mathbf{z} \\ \eta \end{bmatrix} = \begin{bmatrix} f_S \\ 0 \end{bmatrix}, \tag{9}$$

where η is an auxiliary scalar. The extension gradient is then obtained directly as

$$\nabla_{\mathbf{x}_\theta} F_V(\mathbf{x}_\theta) = V_S \mathbf{z}. \tag{10}$$

Therefore, after determining the positive support, computing the backward signal requires only a linear solve with the KKT matrix; neither the Jacobian nor its matrix inverse is formed explicitly.

Algorithm 1 Gradient Computation

Require: Instance $\mathcal{I}$, query network NN_θ, LMO, objective f, number of queries m, regularization $\gamma > 0$

1: Generate and sample query directions $\mathbf{q}_1, \ldots, \mathbf{q}_m \leftarrow \mathrm{NN}_\theta(\mathcal{I})$
2: Set the interpolation point $\mathbf{x}_\theta \leftarrow \frac{1}{m} \sum_{i=1}^m \mathbf{q}_i$
3: **in parallel:** $\mathbf{v}_i \leftarrow \mathrm{LMO}(\mathbf{q}_i)$ for $i = 1, \ldots, m$
4: $V \leftarrow [\mathbf{v}_1, \ldots, \mathbf{v}_m]$
5: Compute: $\boldsymbol{\alpha} \leftarrow \arg\min_{\boldsymbol{\alpha} \geq 0} \frac{1}{2}\|\mathbf{x}_\theta - V\boldsymbol{\alpha}\|_2^2 + \frac{\gamma}{2}\|\boldsymbol{\alpha}\|_2^2 \quad \text{s.t.} \quad \mathbf{1}^\top \boldsymbol{\alpha} = 1$
6: $S \leftarrow \{i : \boldsymbol{\alpha}_i > 0\}$, $V_S \leftarrow [\mathbf{v}_i]_{i \in S}$, and $f_S \leftarrow (f(\mathbf{v}_i))_{i \in S}$
7: Form $K_S \leftarrow \begin{bmatrix} V_S^\top V_S + \gamma I & \mathbf{1} \\ \mathbf{1}^\top & 0 \end{bmatrix}$
8: Solve for $\mathbf{z}$: $K_S \begin{bmatrix} \mathbf{z} \\ \eta \end{bmatrix} = \begin{bmatrix} f_S \\ 0 \end{bmatrix}$
9: $\nabla_{\mathbf{x}_\theta} F_V(\mathbf{x}_\theta) \leftarrow V_S \mathbf{z}$
10: **return** $\nabla_{\mathbf{x}_\theta} F_V(\mathbf{x}_\theta)$

Note that, for any fixed support S, the KKT system shows that $\boldsymbol{\alpha}_S(\mathbf{x}_\theta)$ is an affine function of $\mathbf{x}_\theta$. As $\mathbf{x}_\theta$ varies, the support may change, giving different affine pieces. Hence $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ is piecewise affine and $F_V(\mathbf{x}_\theta)$ is differentiable almost everywhere. The procedure is given in Algorithm 1.

Proposition 1 (A.e. differentiability and gradient correctness)**.** *Let $\mathcal{P} = \mathrm{conv}(\mathcal{C})$ admit a deterministic LMO with fixed tie-breaking, and let $\gamma > 0$. Assume the network queries $\mathbf{q}_i(\theta)$ are locally Lipschitz and the generated vertices $V(\theta)$ are locally constant for a.e. θ. Let $\mathbf{x} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{q}_i(\theta)$. For fixed V, (5) admits unique coefficients $\boldsymbol{\alpha}(\mathbf{x}) \geq 0$ with $\mathbf{1}^\top \boldsymbol{\alpha}(\mathbf{x}) = 1$, and $\boldsymbol{\alpha}(\mathbf{x})$ is piecewise affine and hence differentiable a.e. in $\mathbf{x}$. Consequently, $F_{V(\theta)}(\mathbf{x}_\theta)$ is differentiable a.e. in θ, and Algorithm 1, followed by backpropagation through $\mathbf{x}_\theta$, computes the exact gradient a.e.*

Inference. At inference time, the network generates m query directions, which are passed independently through the LMO to obtain feasible candidates $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathcal{C}$. Since feasibility is enforced by the oracle, no rounding step is required. We evaluate the combinatorial objective on the generated candidates and return the best one. See Figure 1 for the overall pipeline in inference and training.

Remark 2. The constraint $\boldsymbol{\alpha} \geq 0$ is required only if the coefficients are to define a probability distribution over the generated vertices. If it is removed while retaining $\mathbf{1}^\top \boldsymbol{\alpha} = 1$, the coefficients instead define an affine combination of the vertices, giving a regularized affine interpolation over their affine hull. In this case there is no active set: the KKT matrix is fixed for a given V, and the gradient can be computed directly from a linear KKT solve **without** first solving for $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ (5). Thus, the nonnegative formulation provides a probabilistic interpretation, while the equality-only formulation yields a simpler affine extension and backward computation. This makes a speed difference in training, but is irrelevant to inference.

4 Experiments

We evaluate our framework on Maximum Coverage (MC), Quadratic Assignment (QAP), MaxCut, and Densest k-Subgraph (DKS). Our framework applies unchanged across problems; only the LMO changes. The network need only produce m query vectors and support backpropagation through them, allowing architectures and data pipelines tailored to each problem. Our implementations scale from small assignment problems to graphs with millions of vertices, achieve SOTA solution quality in several comparisons, and lie on the quality–time Pareto frontier across all four problems. We exploit GPU parallelism to generate queries in one network forward pass and obtain feasible solutions through independent, parallel LMO calls. Compared with sequential decomposition or iterative neural improvement, this reduces sequential computation and repeated network evaluations during training and inference. This efficiency enables fast large-scale solution generation and opens opportunities to learn better representations from larger, more diverse training instances. Code is available at Link.

4.1 Maximum Cut

Let $G = (V, E)$ be an undirected graph with n vertices and edge weights w_{uv}. MaxCut seeks a partition of the vertices into two parts that maximizes the total weight of crossing edges. Representing the partition by $\mathbf{s} \in \{-1, 1\}^n$, the problem is $\max_{\mathbf{s}\in\{-1,1\}^n} f_{\text{cut}}(\mathbf{s})$, where $f_{\text{cut}}(\mathbf{s}) = \sum_{\{u,v\}\in E} w_{uv}(1-\mathbf{s}_u\mathbf{s}_v)/2$ and each edge is counted once. Note that, MaxCut has no coupling constraint between variables: every sign vector is feasible, so $\mathcal{C} = \{-1, 1\}^n$.

Neural network, oracle, and training. A width-32 residual message-passing network with 512 layers processes BFS parity features and random node features. A learned mapping combines the resulting node embeddings with fixed Sobol vectors to produce $\mathbf{q}_1, \ldots, \mathbf{q}_m \in \mathbb{R}^n$. We use the node-assignment hypercube $\mathcal{P}_{\text{cut}} = \mathrm{conv}(\{-1, 1\}^n) = [-1, 1]^n$. Its LMO acts independently on each coordinate: $\mathbf{v}_{i,u} = 1$ if $\mathbf{q}_{i,u} \geq 0$, and $\mathbf{v}_{i,u} = -1$ otherwise. Each returned vertex therefore defines a cut.

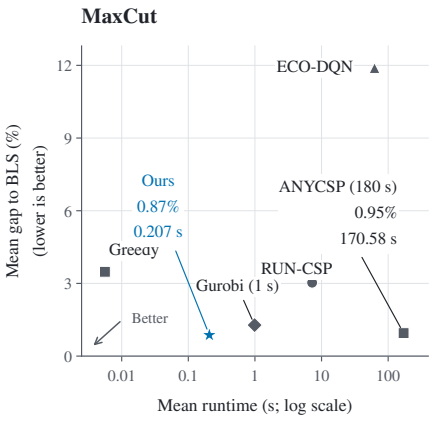

Figure 2: MaxCut on 35 Gset graphs: mean gap to the BLS reference versus mean runtime (*log scale*), with equal weight per graph. Lower and left are better.

We supply normalized negative cut values to Algorithm 1 and propagate the gradient through the mean query to the learned mapping and message-passing network. Training uses randomly generated graphs of mixed families and sizes, with $m = 16$. Following ANYCSP (Tönshoff et al., 2022), we evaluate on 35 unweighted Gset instances with 800–10000 vertices. At inference, we use $m = 1000$ and return the assignment attaining $\max_j f_{\text{cut}}(\mathbf{v}_j)$.

Results. Figure 2 summarizes the quality–runtime trade-off across the 35 Gset graphs. Points represent results at the evaluated budgets, not time to a common target quality. SDP is omitted because an overall mean over all 35 graphs is unavailable. Our method achieves the lowest reported overall mean gap to the BLS (Benlic & Hao, 2013) reference, 0.87%, with a mean runtime of 0.207 seconds per graph. This improves over ANYCSP's mean gap of 0.95% while being more than $800\times$ faster, and also improves over the 1-second Gurobi baseline both in solution quality and runtime. Greedy is faster but produces substantially worse solutions, with a mean gap of 3.48%. The full results, including the breakdown by graph size, are provided in Table 3 in Appendix D. The quality advantage is particularly pronounced on the largest instances Gset3000+: our method achieves a mean gap of 0.58%, compared with 1.35% for Gurobi and 1.61% for ANYCSP, while requiring only 0.206 seconds per graph on average. Further experimental details are provided in Appendix D, this includes ablation on m and raw numbers.

4.2 Quadratic Assignment Problem

QAP seeks a bijection between n facilities and n locations that minimizes the total flow-weighted distance, given the flows between facilities and distances between locations. Such a bijection can be represented by a permutation matrix. Given flow and distance matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$, respectively, the problem is $\min_{\mathbf{P}\in\Pi_n} f_{\text{qap}}(\mathbf{P})$, where $f_{\text{qap}}(\mathbf{P}) = \operatorname{trace}(\mathbf{A}^\top \mathbf{P}\mathbf{B}\mathbf{P}^\top)$ and Π_n denotes the set of $n \times n$ permutation matrices. Thus, $\mathcal{C} = \Pi_n$.

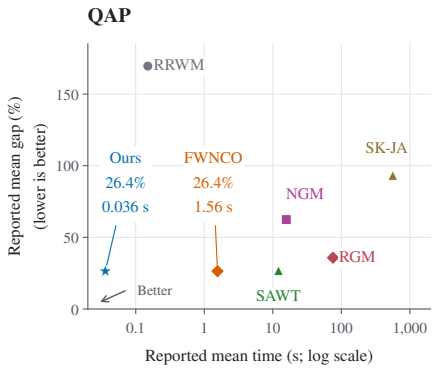

Figure 3: QAPLIB: mean optimality gap across 14 classes versus reported mean inference time (*log scale*), with equal weight per class. Lower and left are better.

Neural network, oracle, and training. GraphSAGE-based encoders process the flow and distance matrices and predict Gaussian parameters for m assignment-score matrices $\mathbf{Q}_j \in \mathbb{R}^{n\times n}$. The feasible polytope is the Birkhoff polytope whose vertices are permutations: $\mathcal{B}_n = \operatorname{conv}(\Pi_n) = \{\mathbf{P} \in \mathbb{R}^{n\times n}_{\geq 0} : \mathbf{P}\mathbf{1} = \mathbf{1}, \mathbf{P}^\top\mathbf{1} = \mathbf{1}\}$. We use a greedy weighted-matching algorithm as an approximate LMO. It selects high-scoring assignments while enforcing one assignment per row and column, producing a permutation $\mathbf{P}_j$ for each query. As in Algorithm 1, the KKT gradient at the mean assignment-score matrix is then calculated and backpropagated through Gaussian sampling and the encoders. We follow SAWT (Tan & Mu, 2024) in training on synthetic QAP32 instances and evaluating on 14 QAPLIB classes with $12 \leq n \leq 64$. Inference returns the permutation with cost $\min_j f_{\text{qap}}(\mathbf{P}_j)$.

Results. Figure 3 compares seven methods on QAPLIB using the mean optimality gap across 14 classes and mean inference time. Our method and FWNCO (Rafiey et al., 2026) both achieve a mean class gap of 26.4%, computed from the rounded per-class values in Appendix Table 6. Our method has the lowest reported inference time at 0.036 seconds, compared with 1.56 seconds for FWNCO, a roughly $43\times$ difference. Baseline runtimes are taken from prior publications, while ours was measured locally on different hardware, so the comparison is not fully controlled. However, our method would remain faster than all baselines even if their reported runtimes were reduced tenfold. It also achieves the lowest gap on five classes (`rou`, `scr`, `ste`, `tho`, and `wil`) and the second-best gap on several others, combining competitive solution quality with low reported inference time. Full results and an ablation on m are provided in Appendix E.

4.3 Maximum Coverage

Let $G = (L \cup R, E)$ be a bipartite graph with $L = [n]$ and nonnegative weights w_u on vertices $u \in R$. MC seeks k vertices from L that maximize the total weight of their neighbors in R. For

a selection vector $\mathbf{x} \in \{0,1\}^n$, the objective is $f_{\text{cov}}(\mathbf{x}) = \sum_{u \in R} w_u \mathbf{1}[\sum_{i:(i,u)\in E} \mathbf{x}_i \geq 1]$. The problem is $\max_{\mathbf{x}\in\mathcal{C}} f_{\text{cov}}(\mathbf{x})$, where $\mathcal{C} = \{\mathbf{x} \in \{0,1\}^n : \mathbf{1}^\top \mathbf{x} = k\}$.

Neural network, oracle, and training. A GraphSAGE (Hamilton et al., 2017) network predicts $\mathbf{q}_1, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ as explained in (2). The feasible polytope is the hypersimplex $\mathcal{P}_{n,k} = \{\mathbf{x} \in [0,1]^n : \mathbf{1}^\top \mathbf{x} = k\}$. Its LMO selects the k largest entries of each query, yielding the indicator vector $\mathbf{v}_j$ of a feasible collection of sets. Following GeoNCO (Karalias et al., 2025), we train on synthetic random instances and evaluate on Random500, Random1000, Rail, and Twitch with no test time optimization. At inference time, we evaluate all candidates generated in parallel and return $\arg\max_j f_{\text{cov}}(\mathbf{v}_j)$. We use $m = 50$ parallel queries (see F.4 for an ablation on this).

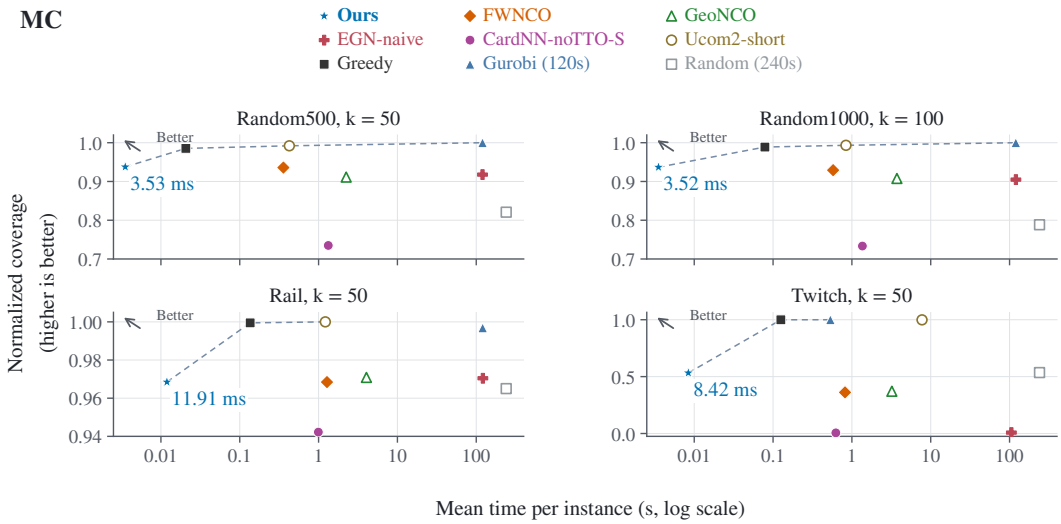

Figure 4: MC at the larger evaluated cardinality bound for each dataset. Coverage is normalized by the best mean in each panel; dashed lines show the quality–runtime Pareto frontier.

Results. Figure 4 shows normalized coverage and inference time for the larger of the two evaluated cardinality bounds k for each dataset. Full results for both values of k are reported in Appendix Table 10. Our method has the lowest inference time in all eight settings, four of which are shown in th figure, requiring only 0.0035–0.0119 seconds per instance. Despite this substantially lower runtime, its solution quality remains competitive with the strongest neural baselines. In particular, it obtains the highest neural coverage on Random1000 with $k = 100$ and on both Twitch settings, while matching the best neural result in several other cases. Because it is the fastest method in every setting, our method lies on the quality–runtime Pareto frontier throughout the benchmark.

Greedy is a strong baseline for MC: its monotone submodular objective gives Greedy a $1 - 1/e$ approximation guarantee under a cardinality constraint. We evaluate on GreedyTrap instances designed to expose limitations of greedy selection. Our method achieves 21–30% higher coverage than Greedy and is faster in three of four settings, reaching an $8.5\times$ speedup on the largest case. We also test scalability on the much larger Random10000 and Random50000 instances. Our method retains 94.8–96.6% of Greedy's coverage while using only 0.01–0.33% of its runtime, demonstrating high coverage at substantially lower inference cost; see Table 1.

Table 1: MC normalized coverage and runtime in seconds. Coverage is normalized by best mean in each row.

		Greedy		Ours	
Dataset	k	Cov.↑	Time↓	Cov.↑	Time↓
GreedyTrap500	10	0.77	0.00198	1.00	0.00362
	50	0.82	0.00710	1.00	0.00352
GreedyTrap1000	20	0.78	0.00823	1.00	0.00382
	100	0.82	0.02950	1.00	0.00346
Random10000	200	1.00	1.94177	0.97	0.00633
	1000	1.00	8.10018	0.95	0.00619
Random50000	1000	1.00	51.06446	0.95	0.02492
	5000	1.00	252.55409	0.95	0.02515

4.4 Densest k-Subgraph

Let $G = (V, E)$ be a simple undirected graph with n vertices and adjacency matrix $\mathbf{A}$. DKS seeks k vertices that induce the largest number of edges. For a selection vector $\mathbf{v} \in \{0,1\}^n$,

the objective is $f_{\text{dks}}(\mathbf{v}) = \sum_{\{u,w\}\in E} \mathbf{v}_u \mathbf{v}_w = \frac{1}{2}\mathbf{v}^\top \mathbf{A}\mathbf{v}$. The problem is $\max_{\mathbf{v}\in\mathcal{C}} f_{\text{dks}}(\mathbf{v})$, where $\mathcal{C} = \{\mathbf{v} \in \{0,1\}^n : \mathbf{1}^\top \mathbf{v} = k\}$.

Neural network, oracle, and training. To scale to million-vertex graphs, we compute compact node features from sampled neighborhoods of the input graph. A shared MLP with hidden widths 32 and 16 combines learned and structural scores to generate full-graph queries $\mathbf{q}_j \in \mathbb{R}^n$. As in MC, the LMO operates over the hypersimplex $\mathcal{P}_{n,k}$ and selects the k highest-scoring vertices for each query. Thus, the neural network processes only a small portion of the graph, while the LMO produces feasible k-vertex subsets over the full graph. In Algorithm 1, we use negative induced-edge density as the k-vertex subsets cost and backpropagate through the mean query to the shared MLP. We train on random 500-vertex graphs with $k \in 20, 50, 100, 200$ and $m = 32$, and evaluate at $k = 100$ on the 12 SNAP networks (Leskovec & Sosič, 2016) used by FW-DL (Lu et al., 2025) and EP-Prox (Liu et al., 2026). At inference, we use $m = 100$ or $m = 1000$ and return the subset maximizing $f_{\text{dks}}(\mathbf{v}_j)$. Queries are processed in chunks of 16 to control memory usage while retaining the full graph.

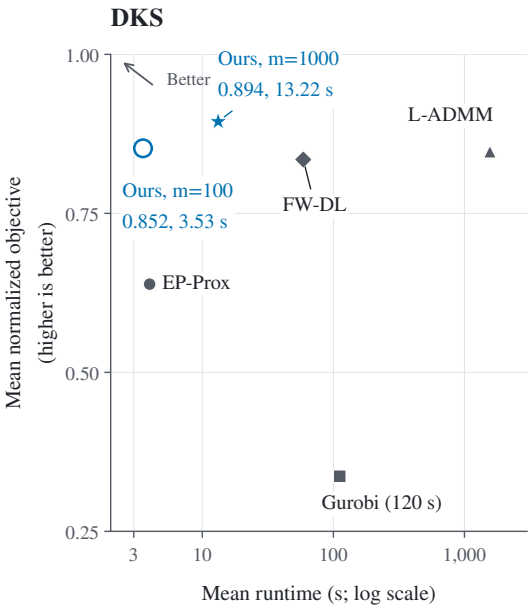

Figure 5: DKS on 12 SNAP graphs (k = 100): mean normalized objective versus mean run-time (*log scale*), with equal weight per graph. Higher and left are better

Results. Figure 5 summarizes solution quality and runtime on the 12 SNAP graphs. With only $m = 100$ queries, our method achieves a mean normalized objective of 0.85, matching L-ADMM for the second-best solution quality, while having the lowest mean runtime at 3.53 seconds. This is about $17\times$ faster than FW-DL and over $400\times$ faster than L-ADMM. Increasing the budget to $m = 1000$ improves the mean objective to 0.89, the best among all compared methods, with a mean runtime of 13.22 seconds. This remains about $4.5\times$ faster than FW-DL and over two orders of magnitude faster than L-ADMM. It also achieves the best or tied-best objective on several individual graphs, including web-Stanford, soc-Pokec, ca-HepTh, com-DBLP, and soc-LiveJournal1. Overall, m provides a clear quality–runtime trade-off: $m = 100$ favors speed, while $m = 1000$ improves solution quality at modest additional cost. Full results appear in Table 13, with additional results and ablations in Appendix G.

5 Conclusion

We present a parallel, GPU-friendly framework for self-supervised neural combinatorial optimization under constraints. Independent oracle queries generate feasible candidates in parallel, while a regularized geometric function supplies training gradients without requiring differentiation through the discrete oracle. Compared with earlier neural approaches, this design reduces sequential computation and produces feasible solutions directly at inference time, without rounding. Across MaxCut, Quadratic Assignment, Maximum Coverage, and Densest k-Subgraph, the framework achieves competitive solution quality with low inference latency. These results suggest a scalable approach to label-free learning for constrained combinatorial optimization. The framework relies on a well-characterized feasible polytope and an efficient exact or approximate linear minimization oracle (LMO), which may limit its applicability to less structured constraint families. Moreover, feasibility alone does not imply an approximation guarantee for the original objective. Extending the framework to broader classes of constraints and establishing corresponding approximation guarantees are important directions for future work.

AI USE STATEMENT

We used generative AI tools to polish the writing and assist with implementing our method and experiments. We did **not** use AI to develop research ideas, formulate mathematical claims, write proofs, design experiments, or interpret results. The authors reviewed all AI-edited text, and one author reviewed and tested all AI-generated code for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The appendix provides experimental details, including training data, hyperparameters, and baseline configurations. Code and data are available in Link.

REFERENCES

Ahmed Abbas and Paul Swoboda. Fastdog: Fast discrete optimization on gpu. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 439–449. IEEE, 2022.

Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning to solve circuit-sat: An unsupervised differentiable approach. In *International Conference on Learning Representations (ICLR)*, 2018.

Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 3243–3250, 2020.

Una Benlic and Jin-Kao Hao. Breakout local search for the max-cutproblem. *Engineering Applications of Artificial Intelligence*, 26(3):1162–1173, 2013.

Fanchen Bu, Hyeonsoo Jo, Soo Yong Lee, Sungsoo Ahn, and Kijung Shin. Tackling prevalent conditions in unsupervised combinatorial optimization: Cardinality, minimum, covering, and more. In *41st International Conference on Machine Learning (ICML)*, pp. 4696–4729, 2024.

Minsu Cho, Jungmin Lee, and Kyoung Mu Lee. Reweighted random walks for graph matching. In *European conference on Computer vision*, pp. 492–505. Springer, 2010.

Chase Cook, Hengyang Zhao, Takashi Sato, Masayuki Hiromoto, and Sheldon X-D Tan. Gpu-based ising computing for solving max-cut combinatorial optimization problems. *Integration*, 69: 335–344, 2019.

Priya L. Donti, David Rolnick, and J Zico Kolter. DC3: A learning method for optimization with hard constraints. In *International Conference on Learning Representations (ICLR)*, 2021.

Zhang-Hua Fu, Kai-Bin Qiu, and Hongyuan Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 7474–7482, 2021.

Michel X Goemans and David P Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6): 1115–1145, 1995.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

Nikolaos Karalias and Andreas Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. *Advances in Neural Information Processing Systems*, 33: 6659–6672, 2020.

Nikolaos Karalias, Joshua Robinson, Andreas Loukas, and Stefanie Jegelka. Neural set function extensions: Learning with discrete functions in high dimensions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Nikolaos Karalias, Akbar Rafiey, Yifei Xu, Zhishang Luo, Behrooz Tahmasebi, Connie Jiang, and Stefanie Jegelka. Geometric algorithms for neural combinatorial optimization with constraints. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Nicolas Keriven and Samuel Vaiter. What functions can graph neural networks compute on random graphs? the role of positional encoding. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:11823–11849, 2023.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 6348–6358, 2017.

Aritra Konar and Nicholas D Sidiropoulos. Exploring the subgraph density-size trade-off via the lovaśz extension. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, pp. 743–751, 2021.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations (ICLR)*, 2019.

Yam Kushinsky, Haggai Maron, Nadav Dym, and Yaron Lipman. Sinkhorn algorithm for lifted assignment problems. *SIAM Journal on Imaging Sciences*, 12(2):716–735, 2019.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in neural information processing systems*, 33:21188–21198, 2020.

Marius Leordeanu and Martial Hebert. A spectral technique for correspondence problems using pairwise constraints. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pp. 1482–1489. IEEE, 2005.

Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1–20, 2016.

Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. *Advances in Neural Information Processing Systems*, 33:4465–4478, 2020.

Derek Lim, Joshua David Robinson, Lingxiao Zhao, Tess Smidt, Suvrit Sra, Haggai Maron, and Stefanie Jegelka. Sign and basis invariant networks for spectral graph representation learning. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

Chang Liu, Zetian Jiang, Runzhong Wang, Lingxiao Huang, Pinyan Lu, and Junchi Yan. Revocable deep reinforcement learning with affinity regularization for outlier-robust graph matching. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

Ya Liu, Junbin Liu, Wing-Kin Ma, and Aritra Konar. A scalable and exact relaxation for densest k-subgraph via error bounds. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 15360–15368, 2026.

Qiheng Lu, Nicholas D Sidiropoulos, and Aritra Konar. Densest k-subgraph mining via a provably tight relaxation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 12291–12299, 2025.

Jiale Ma, Wenzheng Pan, Yang Li, and Junchi Yan. Coexpander: Adaptive solution expansion for combinatorial optimization. In *ICML*, 2025.

Yimeng Min, Frederik Wenkel, Michael Perlmutter, and Guy Wolf. Can hybrid geometric scattering networks help solve the maximum clique problem? *Advances in Neural Information Processing Systems (NeurIPS)*, 35:22713–22724, 2022.

Konrad Mundinger, Max Zimmer, Aldo Kiem, Christoph Spiegel, and Sebastian Pokutta. Neural discovery in mathematics: Do machines dream of colored planes? In *Forty-second International Conference on Machine Learning (ICML)*, 2025.

George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.*, 14(1):265–294, 1978.

Robert R Nerem, Zhishang Luo, Akbar Rafiey, and Yusu Wang. Differentiable extensions with rounding guarantees for combinatorial optimization over permutations. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Ruizhong Qiu, Zhiqing Sun, and Yiming Yang. Dimes: A differentiable meta solver for combinatorial optimization problems. *Advances in Neural Information Processing Systems (NeurIPS)*, 35: 25531–25546, 2022.

Akbar Rafiey, Yifei Xu, and Nikolaos Karalias. Self-supervised combinatorial optimization with constraints via frank-wolfe. 2026.

Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Approximation ratios of graph neural networks for combinatorial problems. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

Martin JA Schuetz, J Kyle Brubaker, and Helmut G Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022.

Richard Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The annals of mathematical statistics*, 35(2):876–879, 1964.

Haoran Sun, Etash K Guha, and Hanjun Dai. Annealed training for combinatorial optimization on graphs. *arXiv preprint arXiv:2207.11542*, 2022.

Zhiqing Sun and Yiming Yang. Difusco: Graph-based diffusion solvers for combinatorial optimization. *Advances in Neural Information Processing Systems (NeurIPS)*, 36:3706–3731, 2023.

Zhentao Tan and Yadong Mu. Learning solution-aware transformers for efficiently solving quadratic assignment problem. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pp. 47627–47648, 2024.

Jan Toenshoff, Martin Ritzert, Hinrikus Wolf, and Martin Grohe. Graph neural networks for maximum constraint satisfaction. *Frontiers in artificial intelligence*, 3:580607, 2021.

Jan Tönshoff, Berke Kisin, Jakob Lindner, and Martin Grohe. One model, any csp: graph neural networks as fast global search heuristics for constraint satisfaction. *arXiv preprint arXiv:2208.10227*, 2022.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2692–2700, 2015.

Haoyu Peter Wang, Nan Wu, Hang Yang, Cong Hao, and Pan Li. Unsupervised learning for combinatorial optimization with principled objective relaxation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022a.

Runzhong Wang, Junchi Yan, and Xiaokang Yang. Neural graph matching network: Learning lawler's quadratic assignment problem with extension to hypergraph and multiple-graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5261–5279, 2021.

Runzhong Wang, Li Shen, Yiting Chen, Xiaokang Yang, Dacheng Tao, and Junchi Yan. Towards one-shot neural combinatorial solvers: Theoretical and empirical notes on the cardinality-constrained case. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2022b.

Runzhong Wang, Yunhao Zhang, Ziao Guo, Tianyi Chen, Xiaokang Yang, and Junchi Yan. Linsatnet: the positive linear satisfiability neural networks. In *International Conference on Machine Learning (ICML)*, pp. 36605–36625. PMLR, 2023.

Yu-Xiang Wang, Veeranjaneyulu Sadhanala, Wei Dai, Willie Neiswanger, Suvrit Sra, and Eric Xing. Parallel and distributed block-coordinate frank-wolfe algorithms. In *International conference on machine learning*, pp. 1548–1557. PMLR, 2016.

Yifan Xia, Xianliang Yang, Zichuan Liu, Zhihao Liu, Lei Song, and Jiang Bian. Position: Rethinking post-hoc search-based neural approaches for solving large-scale traveling salesman problems. In *International Conference on Machine Learning (ICML)*, pp. 54178–54190. PMLR, 2024.

Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. *Advances in Neural Information Processing Systems*, 34:7472–7483, 2021.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. Deepaco: Neural-enhanced ant systems for combinatorial optimization. *Advances in neural information processing systems*, 36: 43706–43728, 2023.

Hongtai Zeng, Chao Yang, Yanzhen Zhou, Cheng Yang, and Qinglai Guo. Glinsat: The general linear satisfiability neural network layer by accelerated gradient descent. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:122584–122615, 2024.

Jingyi Zhao, Linxin Yang, Haohua Zhang, Qile He, and Tian Ding. From sequential to parallel: Reformulating dynamic programming as gpu kernels for large-scale stochastic combinatorial optimization. *arXiv preprint arXiv:2602.05179*, 2026.

A Related work

Self-supervised neural CO. Neural CO methods learn from distributions of problem instances using supervised learning, reinforcement learning, or objectives derived directly from the combinatorial problem (Vinyals et al., 2015; Khalil et al., 2017; Kool et al., 2019; Amizadeh et al., 2018). Self-supervised methods use the discrete objective as a training signal without requiring optimal solutions as labels. They commonly optimize a continuous surrogate, often expressed as the expected objective value under a distribution parameterized by the network (Karalias & Loukas, 2020; Toenshoff et al., 2021; Wang et al., 2022a; Bu et al., 2024). Besides the choice of neural architecture and input features (Min et al., 2022; Sato et al., 2019; Lim et al., 2023; Keriven & Vaiter, 2023), the design of this surrogate and the recovery of a feasible solution are central parts of the solver.

Hard constraints complicate both steps. Penalty terms can discourage violations, but their weights may require tuning and a feasible output may still need to be recovered. Other methods use differentiable projections or specialized layers for cardinality, assignment, or more general linear constraints (Donti et al., 2021; Wang et al., 2023; Zeng et al., 2024; Sinkhorn, 1964; Wang et al., 2022b). Annealing and physics-inspired formulations offer further ways to shape the training objective (Sun et al., 2022; Schuetz et al., 2022; Mundinger et al., 2025). Geometric approaches instead define distributions supported on feasible solutions of a combinatorial polytope. The expected discrete objective then supplies a self-supervised loss, and the same feasible support can be used at inference (Karalias et al., 2022; 2025; Nerem et al., 2025; Rafiey et al., 2026). Our method follows this feasible-support perspective while constructing the support from independent oracle queries instead of building it through sequential decomposition.

Parallel approaches to CO. Parallelization, especially on GPUs, has long been an important route to faster CO algorithms. GPUs can evaluate many states, candidate solutions, or problem instances concurrently, and their throughput is especially useful when computations can be expressed as batched vector and matrix operations. Examples include GPU-based Max Cut search, massively parallel optimization of binary programs, parallel Frank–Wolfe variants, and GPU kernels for batched dynamic programming (Cook et al., 2019; Abbas & Swoboda, 2022; Wang et al., 2016; Zhao et al., 2026). Recent neural solvers also exploit GPU parallelism during prediction. Non-autoregressive graph models predict edge preferences for many decisions at once, while diffusion-based solvers update many variables simultaneously within each denoising step (Joshi et al., 2019; Sun & Yang, 2023). These approaches make the prediction stage more parallel, motivating closer attention to the subsequent steps that construct a feasible discrete solution.

Many neural CO methods use parallel neural computation to produce continuous predictions, such as heatmaps or points in a relaxation, but obtaining a high-quality feasible solution can require substantial sequential computation after the network output is available. Heatmap-based methods predict scores for many edges or variables simultaneously, then assemble solutions through beam search, conflict-aware greedy decoding, or heatmap-guided Monte Carlo tree search (MCTS) (Joshi et al., 2019; Fu et al., 2021; Qiu et al., 2022; Sun & Yang, 2023). Diffusion solvers also perform successive denoising steps before decoding (Qiu et al., 2022; Sun & Yang, 2023). Other learned predictors supply scores or heuristics to a subsequent combinatorial algorithm: NeuroLKH constructs candidate edge sets for LKH search, DeepACO guides ant-colony search (Xin et al., 2021; Ye et al., 2023). Autoregressive neural solvers take another route, repeatedly selecting the next decision conditioned on the partial solution (Kool et al., 2019; Kwon et al., 2020). Sequential dependence also appears in geometric methods: GeoNCO constructs vertices step by step within each decomposition, and FWNCO's next oracle query depends on the preceding decomposition step (Karalias et al., 2025; Rafiey et al., 2026). COExpander predicts heatmaps and determines variables over multiple adaptive expansion rounds, rather than completing solution construction in a single prediction step (Ma et al., 2025). Thus, although neural predictions can be computed in parallel, converting them into feasible solutions often requires sequential decoding, search, or decomposition. These steps can limit end-to-end speed, while solution quality may depend heavily on the choice of decoding or search procedure. Our method addresses the candidate-generation stage directly: independently generated oracle queries produce feasible solutions in parallel, which support the training extension and can be evaluated directly at inference.

B PROOF OF PROPOSITION 1

Proof. We first fix the generated vertex matrix $V = [\mathbf{v}_1, \dots, \mathbf{v}_m]$ and consider the dependence on $\mathbf{x}$. The Hessian of the objective in (5) with respect to $\boldsymbol{\alpha}$ is $V^\top V + \gamma I \succeq \gamma I \succ 0$. Hence the objective is strongly convex in $\boldsymbol{\alpha}$, while the feasible set $\{\boldsymbol{\alpha} \geq 0 : \mathbf{1}^\top \boldsymbol{\alpha} = 1\}$ is convex and compact. Therefore, for every $\mathbf{x}$, (5) has a unique minimizer $\boldsymbol{\alpha}(\mathbf{x})$.

We next show that the solution map is Lipschitz for fixed V. Let $\boldsymbol{\alpha} = \boldsymbol{\alpha}(\mathbf{x})$ and $\widetilde{\boldsymbol{\alpha}} = \boldsymbol{\alpha}(\widetilde{\mathbf{x}})$. First-order optimality of the two convex problems over the same feasible set gives

$$\begin{aligned}\left\langle (V^\top V + \gamma I)\boldsymbol{\alpha} - V^\top \mathbf{x}, \widetilde{\boldsymbol{\alpha}} - \boldsymbol{\alpha} \right\rangle \geq 0,\\ \left\langle (V^\top V + \gamma I)\widetilde{\boldsymbol{\alpha}} - V^\top \widetilde{\mathbf{x}}, \boldsymbol{\alpha} - \widetilde{\boldsymbol{\alpha}} \right\rangle \geq 0.\end{aligned}$$

Adding the two inequalities and using $V^\top V + \gamma I \succeq \gamma I$ yields

$$\begin{aligned}\gamma \|\boldsymbol{\alpha} - \widetilde{\boldsymbol{\alpha}}\|_2^2 &\leq \left\langle V^\top (\mathbf{x} - \widetilde{\mathbf{x}}), \boldsymbol{\alpha} - \widetilde{\boldsymbol{\alpha}} \right\rangle \\ &\leq \|V\|_2 \|\mathbf{x} - \widetilde{\mathbf{x}}\|_2 \|\boldsymbol{\alpha} - \widetilde{\boldsymbol{\alpha}}\|_2.\end{aligned}$$

Thus,

$$\|\boldsymbol{\alpha}(\mathbf{x}) - \boldsymbol{\alpha}(\widetilde{\mathbf{x}})\|_2 \leq \frac{\|V\|_2}{\gamma} \|\mathbf{x} - \widetilde{\mathbf{x}}\|_2.$$

In particular, $\boldsymbol{\alpha}(\mathbf{x})$ is continuous.

We now characterize the solution on a fixed positive support $S = \{i : \boldsymbol{\alpha}_i(\mathbf{x}) > 0\}$. As derived in the main text, the positive coefficients and the multiplier λ satisfy (6). The matrix K_S is nonsingular. Indeed, suppose

$$K_S \begin{bmatrix} \mathbf{u} \\ t \end{bmatrix} = 0.$$

Then $\mathbf{1}^\top \mathbf{u} = 0$ and $(V_S^\top V_S + \gamma I)\mathbf{u} + t\mathbf{1} = 0$. Multiplying the latter equality by $\mathbf{u}^\top$ gives

$$\mathbf{u}^\top (V_S^\top V_S + \gamma I)\mathbf{u} = 0.$$

Since $V_S^\top V_S + \gamma I \succ 0$, this implies $\mathbf{u} = 0$, and consequently $t = 0$. Hence K_S is invertible and

$$\begin{bmatrix} \boldsymbol{\alpha}_S(\mathbf{x}) \\ \lambda(\mathbf{x}) \end{bmatrix} = K_S^{-1} \begin{bmatrix} V_S^\top \mathbf{x} \\ 1 \end{bmatrix}.$$

Therefore, $\boldsymbol{\alpha}_S(\mathbf{x})$ is affine in $\mathbf{x}$ whenever the support S is fixed.

For $i \notin S$, stationarity gives

$$\boldsymbol{\mu}_i = \mathbf{v}_i^\top V_S \boldsymbol{\alpha}_S(\mathbf{x}) - \mathbf{v}_i^\top \mathbf{x} + \lambda(\mathbf{x}),$$

which is also affine in $\mathbf{x}$. Thus, for each support S, the KKT conditions $\boldsymbol{\alpha}_S \geq 0$ and $\boldsymbol{\mu}_i \geq 0$ for $i \notin S$ are affine inequalities in $\mathbf{x}$ and define a polyhedral region. Since there are finitely many supports $S \subseteq [m]$, $\boldsymbol{\alpha}(\mathbf{x})$ is piecewise affine. Consequently, it is differentiable except on the boundaries of finitely many polyhedral regions, which form a set of Lebesgue measure zero.

The same conclusion holds for $F_V(\mathbf{x}) = \sum_{i=1}^m \boldsymbol{\alpha}_i(\mathbf{x}) f(\mathbf{v}_i)$, since the values $f(\mathbf{v}_i)$ are fixed when V is fixed. On any region where the support is fixed, differentiating (6) gives (7). Combining this with (8) and the adjoint system (9) gives

$$\nabla_{\mathbf{x}} F_V(\mathbf{x}) = V_S \mathbf{z}.$$

Hence the backward signal returned by Algorithm 1 is the exact gradient on each fixed-support region, and therefore for a.e. $\mathbf{x}$.

It remains to consider the dependence on the network parameters θ. By assumption, the deterministic LMO makes $V(\theta)$ well-defined, and $V(\theta)$ is locally constant for a.e. θ. Fix such a point and a neighborhood on which $V(\theta) = V$ is constant. Since each $\mathbf{q}_i(\theta)$ is locally Lipschitz, $\mathbf{x}_\theta = m^{-1} \sum_i \mathbf{q}_i(\theta)$ is locally Lipschitz as well. The Lipschitz bound above then implies that $\boldsymbol{\alpha}(\mathbf{x}_\theta)$ is locally Lipschitz in θ, and therefore so is $F_V(\mathbf{x}_\theta)$. By Rademacher's theorem, $F_{V(\theta)}(\mathbf{x}_\theta)$ is differentiable for a.e. θ.

Finally, we verify that the backward computation in Algorithm 1 gives this derivative. Consider a point θ at which V is locally constant and $\mathbf{x}_\theta$, $\boldsymbol{\alpha}(\mathbf{x}_\theta)$, and the KKT multiplier λ are differentiable; these conditions fail only on a measure-zero set. Let $S = \{i : \boldsymbol{\alpha}_i(\mathbf{x}_\theta) > 0\}$. For every $i \notin S$, the scalar function $\boldsymbol{\alpha}_i(\mathbf{x}_\theta)$ is nonnegative and equals zero at the point under consideration. Hence, at every point where it is differentiable, $d\boldsymbol{\alpha}_i = 0$. For every $i \in S$, continuity implies $\boldsymbol{\alpha}_i > 0$ in a neighborhood, so complementary slackness gives $\boldsymbol{\mu}_i = 0$ there. Differentiating the corresponding stationarity equations together with $\mathbf{1}^\top \boldsymbol{\alpha} = 1$ therefore gives

$$K_S \begin{bmatrix} \partial \boldsymbol{\alpha}_S \\ \partial \lambda \end{bmatrix} = \begin{bmatrix} V_S^\top \partial \mathbf{x}_\theta \\ 0 \end{bmatrix}.$$

Here λ is locally Lipschitz as well: whenever $\boldsymbol{\alpha}_i > 0$, stationarity gives $\lambda = \mathbf{v}_i^\top \mathbf{x}_\theta - [(V^\top V + \gamma I)\boldsymbol{\alpha}]_i$.

Since V is locally constant, the values $f(\mathbf{v}_i)$ are also constant, and therefore $\partial F = f_S^\top \partial \boldsymbol{\alpha}_S$. Let $(\mathbf{z}, \eta)$ solve (9). Since K_S is symmetric,

$$\begin{aligned} dF &= \begin{bmatrix} f_S^\top & 0 \end{bmatrix} \begin{bmatrix} \partial \boldsymbol{\alpha}_S \\ \partial \lambda \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{z}^\top & \eta \end{bmatrix} K_S \begin{bmatrix} \partial \boldsymbol{\alpha}_S \\ \partial \lambda \end{bmatrix} \\ &= \mathbf{z}^\top V_S^\top \partial \mathbf{x}_\theta = (V_S \mathbf{z})^\top \partial \mathbf{x}_\theta. \end{aligned}$$

Thus $V_S \mathbf{z}$, exactly the signal returned by Algorithm 1, gives the correct differential through $\mathbf{x}_\theta$ for a.e. θ. Backpropagating this signal through $\mathbf{x}_\theta = m^{-1} \sum_i \mathbf{q}_i(\theta)$ therefore computes the gradient of $F_{V(\theta)}(\mathbf{x}_\theta)$ with respect to θ a.e. □

C HARDWARE AND VRAM USAGE

All our experiments except for the QAP baselines were conducted on a workstation equipped with an AMD Ryzen 9 9950X processor (16 cores, 32 threads) and an NVIDIA GeForce RTX 4090 GPU with 24 GB of memory. The implementation uses Python 3.12.14 and PyTorch 2.12.1 with CUDA 13.2, running on Debian 13 under WSL2. For QAP baselines, we use published results from previous studies. The reasons for this treatment are given in Appendix E.3. Table 2 reports peak GPU memory allocated by PyTorch during training and inference.

Table 2: Peak allocated GPU memory (GiB) of our method.

Problem	Dataset / setting	Training (GiB)	Inference (GiB)
MC	Random500 ($m = 50$)	0.081	0.014
MC	Random1000 ($m = 50$)	0.160	0.024
MC	Random5000 ($m = 200$)	0.581	0.120
MC	Random10000 ($m = 200$)	1.141	0.228
MC	Random50000 ($m = 200$)	5.596	1.094
MC	Rail	–	0.225
MC	Twitch	–	0.163
QAP	QAP32 ($m = 5000$)	2.951	0.111
QAP	QAPLIB ($n = 64$)	–	0.395
MaxCut	Mixed sizes ($n = 250/500/1000$, $m = 16$)	1.059	–
DKS	Synthetic ($n = 500$, $m = 32$)	0.140	–
DKS	Synthetic ($n = 500$, $m = 200$)	0.299	–
DKS	Orkut ($n = 3{,}072{,}441$, $\lvert E\rvert = 117{,}185{,}083$)	–	15.342
DKS	LiveJournal ($n = 4{,}847{,}571$, $\lvert E\rvert = 42{,}851{,}237$)	–	22.082

For MC and QAP, training batch sizes are 4 and 16, respectively; inference batch size is 1.

D Details and Further Experiments on MaxCut

D.1 Experiment Setup

Training and test graphs. Following ECO-DQN (Barrett et al., 2020) and ANYCSP (Tönshoff et al., 2022), we train on generated random graphs and evaluate transfer to established benchmarks. ECO-DQN uses Erdős–Rényi and Barabási–Albert graphs, and ANYCSP uses Erdős–Rényi graphs for MaxCut. We extend this approach by mixing four unweighted graph families: Erdős–Rényi, random regular, Barabási–Albert, and periodic two-dimensional grids with random shortcuts. Training interleaves batches of $n \in \{250, 500, 1000\}$, with one graph size per batch. For testing, we use Gset, an established MaxCut benchmark collection maintained by Yinyu Ye[1]. Following the instance selection and size grouping of ANYCSP (Tönshoff et al., 2022), we evaluate 35 unweighted graphs: nine graphs each with 800, 1000, and 2000 vertices, and eight graphs with 3000–10000 vertices. We report mean cut values and per-graph runtimes; the overall mean weights all 35 graphs equally.

Graph representation and neural network. Each vertex receives 16 breadth-first-search (BFS) parity features and 16 random normal features. A parity feature records whether the shortest-path distance from a sampled reference vertex is even or odd, with zero for unreachable vertices(Li et al., 2020). We use a width-32 residual message-passing network with 512 layers and produces an embedding matrix. The neural network combines this embedding with fixed Sobol vectors to generate m query vectors:

$$q_j = H_\theta\big(c_\theta(d_j) \odot z_j\big), \qquad c_\theta(d) = 0.75 + 0.5\ \mathrm{sigmoid}(d) + 0.2\tanh(W_q d),$$

where (d_j, z_j) is a pair of fixed input vectors.

Training and inference. We train for 30 epochs with 64 new graphs per epoch, batch size 4, and $m_{\text{train}} = 16$. AdamW uses learning rate 10^{-3}, weight decay 10^{-5}, and gradient clipping at norm 1. At inference, we score all m assignments and return the largest cut. The main results use $m = 1000$.

D.2 Related Work and Baselines

The traditional baselines are Greedy and semidefinite programming (SDP). The SDP relaxation follows Goemans and Williamson (Goemans & Williamson, 1995). RUN-CSP, ECO-DQN, and ANYCSP are neural-network methods. RUN-CSP (Toenshoff et al., 2021) uses recurrent message passing with unsupervised training. ECO-DQN (Barrett et al., 2020) uses a graph neural network and deep Q-learning to select vertex flips. ANYCSP (Tönshoff et al., 2022) uses a graph neural network trained with policy gradients to update complete assignments. Our comparison uses the implementations and bundled ER800 pretrained checkpoints in MaxCut-Bench[2]. Following the multi-run evaluation approach of ANYCSP (Tönshoff et al., 2022), we use 20 trajectories per neural baseline and a 180-second limit for ANYCSP, set RUN-CSP to 1600 recurrent steps and ECO-DQN to $2n$ steps. SDP is evaluated on the 800- and 1000-vertex groups with a three-hour limit per graph. Time columns report the recorded mean runtimes.

D.3 Scaled Gaps and Raw Cut Values

Table 4 provides the raw cuts corresponding to the scaled gaps in Table 3. Both tables weight all graphs equally within each group and across all 35 instances.

D.4 Effect of the Parallel Query Budget

The *parallel query budget* m is the number of query directions and corresponding feasible candidates generated by the neural network. We keep the trained weights fixed and evaluate $m \in \{200, 500, 1000, 2000, 5000\}$. For budgets above 1000, we extend the same fixed Sobol sequence, preserving the original vector prefix. Each budget is timed with a separate full forward pass, including query generation, sign decoding, scoring, and selection; graph loading and BFS preprocessing are outside this inference measurement.

[1] https://web.stanford.edu/~yyye/yyye/Gset/

[2] https://github.com/ankurnath/MaxCut-Bench

Table 3: MaxCut results on 35 Gset graphs. Mean gap (%) and time: lower is better. and indicate the best and second-best results, respectively.

	Gset800 (9 graphs)		Gset1000 (9 graphs)		Gset2000 (9 graphs)		Gset3000+ (8 graphs)		Mean (35 graphs)	
Method	Gap (%) ↓	Time (s) ↓	Gap (%) ↓	Time (s) ↓	Gap (%) ↓	Time (s) ↓	Gap (%) ↓	Time (s) ↓	Gap (%) ↓	Time (s) ↓
Greedy	2.93	0.0009	3.74	0.0010	3.75	0.0029	3.51	0.0190	3.48	0.0056
Gurobi (1s)	0.96	1.0378	1.20	1.0319	1.62	1.0368	1.35	0.8349	1.28	0.9896
SDP	3.22	502.1552	3.76	1929.5763	–	–	–	–	–	–
RUN-CSP	3.06	5.9356	3.22	4.4649	3.31	8.1989	2.47	10.6793	3.03	7.2237
ECO-DQN	9.86	16.0096	10.76	20.8582	11.69	48.7131	15.68	177.0220	11.89	62.4687
ANYCSP (180s)	0.49	180.1859	0.80	180.2433	0.96	180.1814	1.61	138.1003	0.95	170.5800
Ours	0.88	0.2509	0.99	0.0707	1.00	0.3002	0.58	0.2063	0.87	0.2070

We are using the BLS(Benlic & Hao, 2013) reference cuts and averaging equally over graphs. Raw cut values are provided in Table 4.

Table 4: Raw MaxCut values and inference time on 35 Gset graphs. Mean cut: higher is better; time: lower is better.

	Gset800 (9 graphs)		Gset1000 (9 graphs)		Gset2000 (9 graphs)		Gset3000+ (8 graphs)		Mean (35 graphs)	
Method	Cut ↑	Time (s)	Cut ↑	Time (s)	Cut ↑	Time (s)	Cut ↑	Time (s)	Cut ↑	Time (s)
Greedy	7630.89	0.0009	5212.56	0.0010	10433.00	0.0029	11748.25	0.0190	8670.69	0.0056
Gurobi (1s)	7774.00	1.0378	5349.00	1.0319	10664.89	1.0368	12053.62	0.8349	8872.00	0.9896
SDP	7594.00	502.1552	5205.33	1929.5763	–	–	–	–	–	–
RUN-CSP	7625.11	5.9356	5250.56	4.4649	10500.33	8.1989	11855.75	10.6793	8720.86	7.2237
ECO-DQN	7432.44	16.0096	4948.89	20.8582	9820.56	48.7131	10194.62	177.0220	8039.26	62.4687
ANYCSP (180s)	7800.00	180.1859	5368.56	180.2433	10730.33	180.1814	12048.62	138.1003	8899.40	170.5800
Ours	7751.07	0.2509	5353.74	0.0707	10719.04	0.3002	12174.29	0.2063	8908.83	0.2070

Table 5 shows modest gains as m increases: the mean cut rises from 8907.05 at $m = 200$ to 8910.09 at $m = 5000$. The measured mean inference time changes only from 0.0545 to 0.0562 seconds, consistent with sharing the graph encoder across all queries. The main setting $m = 1000$ achieves 8908.83, already close to the largest-budget result. The same trained model supports all five inference budgets.

Table 5: Effect of the parallel query budget m on MaxCut. Mean cut and inference time over three repetitions.

Method	m	Gset800		Gset1000		Gset2000		Gset3000+		All 35	
		Cut	Time (s)	Cut	Time (s)	Cut	Time (s)	Cut	Time (s)	Cut	Time (s)
Ours	200	7749.67	0.0544	5352.44	0.0548	10716.59	0.0549	12172.29	0.0537	8907.05	0.0545
Ours	500	7750.59	0.0542	5353.19	0.0547	10717.96	0.0549	12173.46	0.0535	8908.10	0.0543
Ours	1000	7751.07	0.0550	5353.74	0.0546	10719.04	0.0556	12174.29	0.0537	8908.83	0.0547
Ours	2000	7751.70	0.0549	5354.44	0.0551	10719.89	0.0555	12174.54	0.0543	8909.45	0.0550
Ours	5000	7752.15	0.0557	5354.93	0.0560	10721.11	0.0569	12174.92	0.0563	8910.09	0.0562

E Details and Further Experiments on QAP

E.1 Experiment Setup

We follow previous approaches prepare data on synthetic datasets and evaluate on the QAPLIB dataset. Training uses the same synthetic QAP32 dataset, with 5120 training instances generated at edge probability $p = 0.3$ and 256 held-out instances. The neural network processes the flow and distance matrices using the matrix features and GraphSAGE representation. It predicts Gaussian mean and variance matrices, from which queries are sampled.

We evaluate on the 14 QAPLIB classes with instance sizes $12 \leq n \leq 64$. The gap is $100(C/C_{\text{ref}} - 1)$, where C is the predicted cost and C_{ref} is the QAPLIB reference value. We report per-class mean gaps, their average and mean inference time per instance in Table 6.

Table 6: QAPLIB results for models trained on synthetic QAP32 instances. We report the optimality gap (%; lower is better) for each QAPLIB class, the average gap across classes, and mean inference time. and indicate the best and second-best results, respectively.

	Gap (%) ↓															
Method	**bur**	**chr**	**esc**	**had**	**kra**	**lipa**	**nug**	**rou**	**scr**	**sko**	**ste**	**tai**	**tho**	**wil**	**Avg.**	**Time (s) ↓**
RRWM	23.1	616.0	63.9	25.1	58.8	20.9	67.8	51.2	173.5	48.5	539.4	197.2	80.6	18.2	169.5	0.150
SK-JA	4.7	38.5	364.8	25.8	41.4	0.0	25.3	13.7	48.6	18.3	120.4	25.2	32.9	8.8	93.2	563.400
NGM	3.4	121.3	126.7	8.2	31.6	16.2	21.0	30.9	55.5	25.2	101.7	61.4	27.5	10.8	62.4	15.720
RGM	7.1	112.4	32.8	6.2	15.0	13.3	9.7	13.4	45.5	10.6	134.1	17.3	20.7	8.1	35.8	75.530
SAWT	2.8	110.7	13.5	3.8	30.1	0.4	9.2	10.8	28.5	17.7	93.5	16.5	24.8	8.1	26.8	12.110
FWNCO	2.3	97.7	19.0	2.8	37.0	13.5	11.5	10.9	23.6	15.0	83.5	20.7	23.0	9.2	26.4	1.560
Ours	2.4	127.6	18.5	3.2	33.6	13.3	10.1	9.5	19.7	13.5	66.4	23.6	20.6	7.6	26.4	0.036

E.2 Baselines

Classical methods. SM (Leordeanu & Hebert, 2005) uses the principal eigenvector of the affinity matrix to score candidate correspondences, followed by discrete matching. RRWM (Cho et al., 2010) performs random walks on an association graph and reweights correspondence scores to enforce matching constraints. SK-JA (Kushinsky et al., 2019) solves the Johnson–Adams relaxation of QAP using generalized Sinkhorn iterations and Bregman projections.

Neural methods. NGM (Wang et al., 2021) learns node embeddings on the association graph and predicts a matching with Sinkhorn normalization. RGM (Liu et al., 2023) uses deep reinforcement learning to construct node correspondences sequentially, with revocable actions that allow earlier matching decisions to be revised. SAWT (Tan & Mu, 2024) uses a solution-aware Transformer to encode the current assignment and learns actions that improve its QAP cost. FWNCO (Rafiey et al., 2026) predicts a continuous assignment matrix and applies sequential Frank–Wolfe decomposition with a greedy assignment oracle, returning the generated permutation with the lowest objective value.

E.3 Source of QAPLIB Baseline Results

We use the published QAPLIB baseline results from previous works. Reusing reported results is established practice in QAP evaluation: SAWT (Tan & Mu, 2024) follows RGM's (Liu et al., 2023) QAPLIB comparison protocol and reuses previously published baseline results.

As discussed in FWNCO (Rafiey et al., 2026, Appendix C.3), the public SAWT and RGM repositories do not provide the complete settings and evaluation pipeline needed to reproduce their reported QAPLIB experiments. That appendix also identifies inconsistencies in earlier QAPLIB summaries, including the WIL class, whose reported values cannot be reconciled with the stated instance selection. Reconstructing these experiments would therefore require unverified choices about the original evaluation. We retain the published baseline values to preserve the original comparisons, and evaluate Our method locally on QAPLIB.

Hardware and computational efficiency. To assess how much of our QAP efficiency advantage comes from the algorithm itself, we compare the hardware configurations used by previous methods.The reported platforms use an RTX 3080 Ti for SAWT (Tan & Mu, 2024), an RTX 8000 for NGM (Wang et al., 2021) and FWNCO (Rafiey et al., 2026), and an RTX 2080 Ti for RGM (Liu et al., 2023); our experiments use an RTX 4090. These GPUs differ substantially in performance: nominal peak FP32 throughput is approximately 34, 16.3, 14.2, and 82.6 TFLOPS for the RTX 3080 Ti, RTX 8000, RTX 2080 Ti, and RTX 4090, respectively. Meanwhile, the reported configurations for methods other than SAWT also use powerful workstation- or server-class CPUs and larger system memory capacities, which can affect runtime and the scale of instances they can process.

Table 6 reports 0.036 seconds per instance for Our method, compared with 1.560, 12.110, 15.720, and 75.530 seconds for FWNCO, SAWT, NGM, and RGM. As an illustrative sensitivity analysis, even granting each baseline a hypothetical $10\times$ reduction in its entire reported runtime leaves time ratios of $4.3\times$, $33.6\times$, $43.7\times$, and $209.8\times$, respectively, relative to Ours. This supports a substantial algorithmic efficiency advantage.

Our memory measurements in Table 2 also show a modest footprint: 2.951 GiB of peak allocated GPU memory for QAP32 training and 0.395 GiB for QAPLIB inference at $n = 64$, using $m = 5000$.

E.4 Effect of the Parallel Query Budget

We compare the *parallel query budget* $m \in \{100, 200, 500, 1000, 3000, 5000, 10000, 20000\}$. For QAP, this budget is implemented with m learned Gaussian mean/variance matrix pairs. Evaluation draws four samples from each distribution, yielding $4m$ assignment directions per instance. Table 7 shows that increasing m reduces the average gap from 35.11% to 25.10%, while inference time increases from 0.01248 to 0.11105 seconds. We use $m = 5000$ in the main comparison, achieving a 26.39% average gap in this experiment. Increasing the budget to $m = 20000$ reduces the gap by a further 1.29 percentage points at approximately $3.2\times$ the inference time.

Table 7: Effect of the parallel query budget m on QAPLIB, with four samples per distribution. Gap and time: lower is better.

m	Avg. gap (%)	Time (s)
100	35.11	0.01248
200	32.17	0.01251
500	29.82	0.01295
1000	28.99	0.01507
3000	26.97	0.02465
5000	26.39	0.03425
10000	25.75	0.05935
20000	25.10	0.11105

F Details and Further Experiments on Maximum Coverage

F.1 Experiment Setup

We follow GeoNCO (Karalias et al., 2025)for data preparation and evaluation on Random500, Random1000, Rail, and Twitch, using the same datasets and cardinality budgets. We report normalized coverage and mean inference time per instance in Table 11.

Neural network training and inference. We represent each instance as a bipartite graph connecting sets to the elements they contain, with node features based on degrees and element weights. The model uses GraphSAGE network with four bidirectional message-passing stages, hidden dimension 64, LeakyReLU activations, and dropout 0.1, it produces m diagonal Gaussian distributions that is later used to sample query vectors.

We train separate models for each Random dataset and cardinality budget. For Random500 and Random1000, training uses AdamW with batch size 4, peak learning rate 10^{-3}, weight decay 10^{-4}, and a 20-epoch warmup followed by cosine decay to 5×10^{-5}. During inference, we draw one query vector from each distribution produced by the neural network, run the LMO, and evaluate the weighted coverage of all m candidates in parallel. We return the candidate with the highest coverage.

No-TTO setting. We follow the no Test-time optimization (TTO) setting of FWNCO (Rafiey et al., 2026, Appendix E), using the short or no-TTO settings of the corresponding MC neural baselines. Test-time optimization can improve solution quality through per-instance fine-tuning or additional search, but it also adds computation and makes the contribution of the learned model harder to isolate. Xia et al. (2024) discuss this issue for heatmap-guided search in large-scale TSP, questioning how much the learned predictions contribute to the final solution. Our evaluation focuses on solution generation with fixed network parameters and the method's prescribed decoder.

F.2 Baselines

We compare against exact solvers, classical heuristics, and neural methods.

Exact solvers. Gurobi solves a mixed-integer programming formulation with a 120-second time limit per instance. We report the best feasible solution found within the limit.

Classical heuristics. Random samples subsets of size k uniformly and returns the best solution found within 240 seconds. Greedy (Nemhauser et al., 1978) repeatedly adds the set with the largest marginal coverage gain. For nonnegative element weights, it achieves a $(1 - 1/\mathrm{e})$ approximation under the cardinality constraint. Following the discussion in GeoNCO (Karalias et al., 2025) , we also classify UCOM2 (Bu et al., 2024) as a traditional baseline because its solution quality is primarily driven by incremental greedy derandomization, and we only report its short setting according to the no-TTO setting.

Neural methods. CardNN (Wang et al., 2022b) uses differentiable cardinality-constrained learning; we report CardNN-noTTO-S using the evaluation setting of Bu et al. (Bu et al., 2024). EGN-naive (Karalias & Loukas, 2020) uses a probabilistic objective with iterative rounding. GeoNCO (Karalias et al., 2025) projects neural predictions onto the feasible polytope and uses Carathéodory decomposition to produce feasible solutions. FWNCO (Rafiey et al., 2026) uses sequential Frank–Wolfe decomposition to generate feasible subsets from a neural prediction and returns the subset with the highest coverage.

F.3 Very Large Graphs

We further evaluate on Random10000 and Random50000, containing 10000/50000 sets and 20000/100000 elements, respectively. Models are trained for 50 epochs with $m = 200$, and each setting is evaluated on 100 test instances. Table 8 compares Ours with the Greedy implementation from UCom2. Across these settings, Ours achieves 94.8–96.6% of Greedy's coverage using only approximately 0.01–0.33% of its reported runtime. The independent directions and top-k oracle calls support batched GPU execution even at these larger sizes.

Table 8: MC results on Random10000 and Random50000 ($m = 200$ for Ours). Coverage: higher is better; time: lower is better.

	Random10000				**Random50000**			
	$k = 200$		$k = 1000$		$k = 1000$		$k = 5000$	
Method	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)
Greedy	311016.46	1.94177	886105.48	8.10018	1556362.95	51.06446	4432881.43	252.55409
Ours	300336.75	0.00633	845122.08	0.00619	1479678.05	0.02492	4203456.14	0.02515

F.4 Effect of the Parallel Query Budget

We compare the *parallel query budget* $m \in \{10, 30, 50, 100, 200\}$ on Random1000 with $k = 20$. Each budget uses a separately trained model with 30 training epochs and the same training and test datasets. Here, the parallel query budget m is the number of query vectors $q_1, \ldots, q_m$ generated for each instance. Table 9 shows that coverage increases from 29506.38 at $m = 10$ to 29878.97 at $m = 50$. Increasing m from 50 to 200 improves coverage by only 0.43%, with similar measured inference times due to parallel processing on the GPU. We use $m = 50$ for the main comparison because it achieves coverage close to the larger budgets while generating fewer candidates. The very-large-graph experiments use $m = 200$ to provide a larger candidate set.

Table 9: Effect of the parallel query budget m on MC on Random1000 ($k = 20$).

m	Coverage ↑	Time (s) ↓
10	29506.38	0.00350
30	29683.72	0.00359
50	29878.97	0.00354
100	29939.88	0.00363
200	30008.05	0.00364

F.5 Raw Coverage Values

Table 10: MC results on Random500, Random1000, Rail, and Twitch. Normalized coverage: higher is better; time: lower is better. and indicate the best and second-best results among neural methods in each column, respectively.

	Random500				Random1000				Rail				Twitch			
	$k = 10$		$k = 50$		$k = 20$		$k = 100$		$k = 20$		$k = 50$		$k = 20$		$k = 50$	
Method	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓	Norm. cov. ↑	Time (s) ↓
Random (240s)	0.86	240.00019	0.82	240.00020	0.78	240.00009	0.79	240.00012	0.94	240.00103	0.97	240.00089	0.54	240.00028	0.54	240.00022
Gurobi (120s)	1.00	0.93991	1.00	120.07712	1.00	21.62572	1.00	120.05282	1.00	120.72583	1.00	120.62132	1.00	0.56769	1.00	0.53344
Greedy	0.99	0.00721	0.99	0.02072	0.99	0.02318	0.99	0.07883	1.00	0.07212	1.00	0.13551	1.00	0.06675	1.00	0.12559
Ucom2-short	0.98	0.45178	0.99	0.42560	0.96	0.84422	0.99	0.84090	0.98	1.42266	1.00	1.21777	1.00	8.24705	1.00	7.75699
GeoNCO	0.96	0.96187	0.91	2.23917	0.95	1.28572	0.91	3.72534	0.95	2.68297	0.97	4.04408	0.32	2.44298	0.37	3.21277
CardNN-noTTO-S	0.59	1.33321	0.73	1.32723	0.59	1.38980	0.73	1.35915	0.90	0.94137	0.94	0.99340	0.00	0.66294	0.01	0.62554
EGN-naive	0.97	53.041	0.92	120.136	0.96	120.010	0.90	120.356	0.93	120.676	0.97	121.335	0.01	108.30300	0.01	105.46050
FWNCO	0.96	0.36310	0.94	0.35790	0.95	0.39220	0.93	0.57820	0.95	1.00100	0.97	1.27860	0.32	0.68350	0.36	0.81810
Ours	0.96	0.00383	0.94	0.00353	0.95	0.00350	0.94	0.00352	0.94	0.00846	0.97	0.01191	0.39	0.00910	0.53	0.00842

Normalized coverage is the mean coverage divided by the highest mean coverage in the same column. Raw coverage values are provided in Table 11.

Table 11 provides the unnormalized coverage values used in Table 10. Each main-table coverage column is divided by its largest entry.

Table 11: Raw MC coverage and inference time on Random500, Random1000, Rail, and Twitch. Coverage: higher is better; time: lower is better.

	Random500				Random1000				Rail				Twitch			
	$k = 10$		$k = 50$		$k = 20$		$k = 100$		$k = 20$		$k = 50$		$k = 20$		$k = 50$	
Method	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)	Coverage	Time (s)
Random (240s)	13488.87	240.00019	36912.84	240.00020	24438.53	240.00009	70717.71	240.00012	5287.67	240.00103	7367.00	240.00089	13898.00	240.00028	16036.50	240.00022
Gurobi (120s)	15714.90	0.93991	44972.28	120.07712	31347.62	21.62572	89702.37	120.05282	5640.00	120.72583	7610.00	120.62132	25864.00	0.56769	30560.83	0.53344
Greedy	15633.93	0.00721	44312.81	0.02072	31132.96	0.02318	88698.89	0.07883	5617.00	0.07212	7630.00	0.13551	25855.33	0.06675	30542.17	0.12559
Ucom2-short	15471.03	0.45178	44614.61	0.42560	30016.16	0.84422	89102.86	0.84090	5525.00	1.42266	7634.00	1.21777	25833.00	8.24705	30526.00	7.75699
GeoNCO	15146.49	0.96187	40984.63	2.23917	29878.15	1.28572	81428.14	3.72534	5343.00	2.68297	7411.67	4.04408	8350.33	2.44298	11351.67	3.21277
CardNN-noTTO-S	9231.54	1.33321	33051.73	1.32723	18458.92	1.38980	65788.15	1.35915	5074.33	0.94137	7193.00	0.99340	77.00	0.66294	173.00	0.62554
EGN-naive	15262.76	53.041	41272.68	120.136	29968.04	120.010	81166.12	120.356	5234.67	120.676	7408.33	121.335	152.67	108.30300	247.17	105.46050
FWNCO	15161.20	0.36310	42087.45	0.35790	29807.93	0.39220	83343.16	0.57820	5344.67	1.00100	7393.00	1.27860	8332.00	0.68350	11042.50	0.81810
Ours	15056.48	0.00383	42165.90	0.00353	29932.85	0.00350	84039.51	0.00352	5318.33	0.00846	7393.20	0.01191	10095.40	0.00910	16276.30	0.00842

G Details and Further Experiments on Densest k-Subgraph

G.1 Experiment Setup

Training and test graphs. We train on synthetic graphs with 500 vertices and evaluate transfer to real-world networks. The training graphs are randomly generated with varying edge densities and densely connected groups of vertices. We train a single model with $k \in \{20, 50, 100, 200\}$, sampling one value of k for each training batch. Testing uses the 12 graphs from the SNAP dataset collection (Leskovec & Sosič, 2016) in Table 13, combining the graph collections used by FW-DL (Lu et al., 2025) and EP-Prox (Liu et al., 2026). They include social, collaboration, web, Internet, and road networks, up to approximately 4.85 million vertices. All tests use $k = 100$ and count each induced undirected edge once. The main table normalizes each count by the highest reported count on the same graph. Raw counts are provided in Table 14.

Graph representation. To accommodate DKS instances with millions of vertices, we use compact node-level features to encode graph information and a shared MLP to generate the query vectors. The features and query vectors are computed in chunks on the sparse graph, limiting peak GPU memory usage. This illustrates the scalability of our framework: the graph representation and neural network can be adapted to the problem size while retaining the same optimization procedure.

To construct the inputs used to generate m query vectors, we sample vertices $a_1, \ldots, a_m$ without replacement from pools ranked by core number and degree. For each sampled vertex a_j, its closed neighborhood $N[a_j] = \{a_j\} \cup N(a_j)$ describes the surrounding graph structure. For each vertex u in the entire graph, we compute 10 features describing its degree and core number, membership in $N[a_j]$, normalized neighborhood-overlap counts, and its core number relative to that of a_j. This gives a feature matrix $H_j \in \mathbb{R}^{n \times 10}$ for each sampled vertex a_j, where row u describes graph vertex u relative to a_j.

Neural Network Training and inference. We use one shared MLP with 10 input features, hidden-layer widths 32 and 16, and SiLU activations to produce m query vectors q_j. At training, we train for 30 epochs with 128 new graphs per epoch, batch size 4, and $m_{\text{train}} = 32$. AdamW uses learning rate 10^{-3}, weight decay 10^{-5}, and gradient clipping at norm 1. Checkpoint selection uses mean density across the four budgets on 24 synthetic validation graphs with 1000 vertices. At inference, we generate m query vectors, apply top-k, score the resulting subsets, and return the best one. We use query chunks of 16 on the full sparse graph, processing 16 independent queries at a time. Runtimes include graph/model preparation and inference.

G.2 Related Work and Baselines

We compare with three continuous-optimization approaches. FW-DL (Lu et al., 2025) applies Frank–Wolfe to a diagonally loaded quadratic relaxation. L-ADMM (Konar & Sidiropoulos, 2021) optimizes a Lovász-extension relaxation using ADMM. EP-Prox (Liu et al., 2026) uses an exact-penalty reformulation with a proximal-gradient algorithm. These methods optimize each test graph directly. Our neural network is trained on synthetic graphs and its parameters remain fixed during testing.

We use FW-DL and L-ADMM from the public DkS-Diagonal-Loading implementation provided by the authors[3], with the released demo settings of 200 and 2000 iterations, respectively. EP-Prox uses the authors' MATLAB implementation and public demo defaults. All methods are evaluated on the same prepared graph structures at $k = 100$, and objectives are reported as induced-edge counts. For Orkut, input conversion for FW-DL/L-ADMM is performed in blocks because of memory limits; this conversion time is excluded for fair comparison, with the optimizer unchanged.

G.3 Effect of the Parallel Query Budget

During training, we use a fixed query budget $m_{\text{train}} = 32$. At inference, we keep the same $m_{\text{train}} = 32$ checkpoint and evaluate $m \in \{100, 200, 500, 1000\}$. The neural network shares its parameters across all m queries.

Table 12 shows that the mean induced-edge count increases from 3016.08 at $m = 100$ to 3135.42 at $m = 1000$, while mean runtime rises from 3.5297 to 13.2186 seconds. Larger budgets are particularly useful on com-Youtube and soc-Pokec, where the counts increase from 2035 to 2397 and from 2064 to 2617, respectively. Extremely large graphs such as com-DBLP and soc-LiveJournal1 already attain the maximum possible 4950 edges at $m = 100$.

Table 12: Effect of the parallel query budget m on DKS ($k = 100$), using one model trained with $m = 32$.

Graph	$m = 100$		$m = 200$		$m = 500$		$m = 1000$	
	Edges	Time (s)	Edges	Time (s)	Edges	Time (s)	Edges	Time (s)
Facebook	4789.00	0.8004	4802.00	0.5067	4805.00	0.7214	4819.00	0.5134
web-Stanford	4149.00	0.8109	4149.00	0.7799	4143.00	1.1586	4143.00	1.5178
com-Youtube	2035.00	1.2997	2117.00	1.5544	2216.00	2.7178	2397.00	4.6347
soc-Pokec	2064.00	3.1761	2212.00	3.6994	2508.00	5.4008	2617.00	9.1007
as-Skitter	4256.00	1.9902	4256.00	2.5392	4471.00	4.2418	4479.00	7.1603
com-Orkut	3274.00	15.0157	3404.00	16.0264	3413.00	19.6053	3424.00	27.5313
ca-HepTh	1160.00	0.5066	1160.00	0.4960	1160.00	0.4866	1160.00	0.5276
ca-CondMat	1022.00	0.4532	1049.00	0.4883	1054.00	0.5487	1027.00	0.5730
com-DBLP	4950.00	0.6359	4950.00	0.7666	4950.00	1.0709	4950.00	1.5621
roadNet-CA	40.00	1.5208	43.00	2.1781	43.00	4.2050	46.00	7.4008
wiki-Talk	3504.00	1.7402	3553.00	2.5712	3541.00	5.0471	3613.00	9.0202
soc-LiveJournal1	4950.00	14.4073	4950.00	22.4427	4950.00	48.1916	4950.00	89.0817
Mean	3016.08	3.5297	3053.75	4.5041	3104.50	7.7829	3135.42	13.2186

[3] https://github.com/luqh357/DkS-Diagonal-Loading

Table 13: DKS results on 12 SNAP graphs ($k = 100$). Normalized objective: higher is better; time: lower is better. and indicate the best and second-best results, respectively.

	Gurobi (120s)		FW-DL		L-ADMM		EP-Prox		Ours ($m = 100$)		Ours ($m = 1000$)	
Dataset	Norm. obj. ↑	Time (s) ↓	Norm. obj. ↑	Time (s) ↓	Norm. obj. ↑	Time (s) ↓	Norm. obj. ↑	Time (s) ↓	Norm. obj. ↑	Time (s) ↓	Norm. obj. ↑	Time (s) ↓
Facebook	1.00	118.0970	1.00	0.5495	0.99	4.0285	0.99	0.1015	0.98	0.8004	0.99	0.5134
web-Stanford	0.00	120.0404	1.00	11.1951	0.29	148.3811	0.06	0.8216	1.00	0.8109	1.00	1.5178
com-Youtube	0.06	120.0634	1.00	21.9967	0.94	411.8932	0.50	1.0469	0.77	1.2997	0.90	4.6347
soc-Pokec	0.12	120.0375	0.52	162.3344	0.30	1880.5188	0.17	5.7141	0.79	3.1761	1.00	9.1007
as-Skitter	0.02	120.0398	1.00	52.7224	0.94	1143.8981	0.53	4.8766	0.93	1.9902	0.97	7.1603
com-Orkut	0.10	121.1402	1.00	62.4983	0.74	8644.3508	0.75	17.3336	0.84	15.0157	0.88	27.5313
ca-HepTh	1.00	118.0885	0.78	0.3836	1.00	3.0927	0.76	0.1042	1.00	0.5066	1.00	0.5276
ca-CondMat	1.00	25.9802	0.90	0.9467	0.99	7.7723	0.80	0.1322	0.81	0.4532	0.81	0.5730
com-DBLP	0.01	120.0482	1.00	6.7888	1.00	102.7746	1.00	0.4909	1.00	0.6359	1.00	1.5621
roadNet-CA	0.59	118.9115	0.19	27.9232	1.00	538.4673	0.13	2.4999	0.22	1.5208	0.25	7.4008
wiki-Talk	0.11	118.9754	1.00	33.8185	0.97	1014.6148	0.97	1.8537	0.90	1.7402	0.93	9.0202
soc-LiveJournal1	0.04	120.0782	0.63	326.1160	1.00	4861.7574	1.00	12.5898	1.00	14.4073	1.00	89.0817
Mean	0.34	111.7917	0.83	58.9394	0.85	1563.4625	0.64	3.9638	0.85	3.5297	0.89	13.2186

Normalized objective is the induced-edge count divided by the highest count in the same row. Mean averages these ratios over all 12 graphs. Raw counts are provided in Table 14. Orkut input conversion for FW-DL/L-ADMM was performed in blocks due to out-of-memory errors; conversion time is excluded for fair comparison.

G.4 Raw Results

Table 14 provides the induced-edge counts used in Table 13. Normalization divides each count by the largest count on the same graph; the main-table Mean averages the 12 resulting ratios.

Table 14: Raw DKS induced-edge counts and inference time on 12 SNAP graphs ($k = 100$). Edges: higher is better; time: lower is better.

	Gurobi (120s)		FW-DL		L-ADMM		EP-Prox		Ours ($m = 100$)		Ours ($m = 1000$)	
Dataset	Edges	Time (s)	Edges	Time (s)	Edges	Time (s)	Edges	Time (s)	Edges	Time (s)	Edges	Time (s)
Facebook	4872	118.0970	4871	0.5495	4824	4.0285	4837	0.1015	4789	0.8004	4819	0.5134
web-Stanford	0	120.0404	4151	11.1951	1190	148.3811	256	0.8216	4149	0.8109	4143	1.5178
com-Youtube	156	120.0634	2659	21.9967	2496	411.8932	1334	1.0469	2035	1.2997	2397	4.6347
soc-Pokec	308	120.0375	1363	162.3344	796	1880.5188	436	5.7141	2064	3.1761	2617	9.1007
as-Skitter	106	120.0398	4595	52.7224	4341	1143.8981	2449	4.8766	4256	1.9902	4479	7.1603
com-Orkut	373	121.1402	3889	62.4983	2884	8644.3508	2929	17.3336	3274	15.0157	3424	27.5313
ca-HepTh	1164	118.0885	908	0.3836	1159	3.0927	888	0.1042	1160	0.5066	1160	0.5276
ca-CondMat	1267	25.9802	1135	0.9467	1251	7.7723	1014	0.1322	1022	0.4532	1027	0.5730
com-DBLP	57	120.0482	4950	6.7888	4950	102.7746	4950	0.4909	4950	0.6359	4950	1.5621
roadNet-CA	106	118.9115	34	27.9232	181	538.4673	23	2.4999	40	1.5208	46	7.4008
wiki-Talk	426	118.9754	3899	33.8185	3800	1014.6148	3768	1.8537	3504	1.7402	3613	9.0202
soc-LiveJournal1	187	120.0782	3126	326.1160	4948	4861.7574	4944	12.5898	4950	14.4073	4950	89.0817
Mean	751.83	111.7917	2965.00	58.9394	2735.00	1563.4625	2319.00	3.9638	3016.08	3.5297	3135.42	13.2186

Orkut input conversion for FW-DL/L-ADMM was performed in blocks due to out-of-memory errors; conversion time is excluded for fair comparison. The original baseline implementations are unchanged.

G.5 Ablation on Training

Since the input features are a strong signal of the underlying structure, we investigate the impact of training on the model's performance. We compare the trained checkpoint with its saved initialization, retaining the same features, fixed terms in q_j, sampled vertices, top-k oracle, and best-candidate selection. The initial neural residual is zero, so the untrained model uses the two fixed terms in q_j. Trained and untrained models use identical sampled vertices within each repetition. We report results at $m = 200$ and $m = 1000$.

Table 15 shows that training increases the mean edge count from 2324.83 to 3043.86 at $m = 200$, and from 2484.22 to 3127.00 at $m = 1000$, corresponding to improvements of 30.9% and 25.9%. Large gains on com-Youtube, as-Skitter, com-Orkut, and wiki-Talk demonstrate that learning improves upon vertex sampling and the fixed features.

Table 15: Effect of training on Ours for DKS ($k = 100$, training $m = 32$). Mean induced edges over three repetitions; higher is better.

	Test $m = 200$		Test $m = 1000$	
Dataset	Untrained	Trained	Untrained	Trained
Facebook	4229.00	4804.67	4425.33	4820.67
web-Stanford	4134.67	4143.67	4143.00	4144.33
com-Youtube	841.33	2184.33	1067.67	2341.67
soc-Pokec	1409.33	2132.00	1806.67	2577.33
as-Skitter	2692.33	4275.67	2866.67	4477.00
com-Orkut	2080.33	3356.67	2296.67	3425.33
ca-HepTh	577.33	1160.33	622.67	1161.33
ca-CondMat	780.33	1032.00	912.33	1070.00
com-DBLP	4950.00	4950.00	4950.00	4950.00
roadNet-CA	114.00	45.00	115.33	49.67
wiki-Talk	1139.33	3492.00	1654.33	3556.67
soc-LiveJournal1	4950.00	4950.00	4950.00	4950.00
Mean (12 graphs)	2324.83	3043.86	2484.22	3127.00