Constraint-Aware Self-Supervised Learning for Edge Selection

Xinda Zheng
Department of Data Science and AI, Faculty of IT, Monash University, Australia
OPTIMA ARC Training Centre, Australia

Frits de Nijs
Department of Data Science and AI, Faculty of IT, Monash University, Australia
OPTIMA ARC Training Centre, Australia

Edward Lam
Department of Data Science and AI, Faculty of IT, Monash University, Australia
OPTIMA ARC Training Centre, Australia

Abstract

Many edge-selection problems, such as the Traveling Salesman Problem and Orienteering Problem, are NP-hard, making them expensive to solve with exact methods and challenging to address with hand-crafted heuristics. Learning-based approaches provide an efficient alternative, while self-supervised methods avoid costly solution labels. However, existing approaches often still rely on heavy post-processing or narrow problem-specific designs. We propose a reusable self-supervised framework for edge-selection optimization that learns directly from unlabeled instances. The framework uses differentiable surrogate objectives and feasibility-driven penalties to encourage the model to learn feasibility-aware solution structure during training. To support efficient inference, we introduce a lightweight graph architecture centered on a cost-attention convolution, where edge costs and feasibility information directly shape message passing. Experiments on three problem families demonstrate strong solution quality and efficient inference across diverse edge-selection settings.

2012 ACM Subject Classification Computing methodologies → Machine learning approaches

Keywords and phrases Combinatorial Optimization, Learning to optimize, Graph neural networks

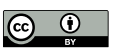
Digital Object Identifier 10.4230/LIPIcs.CP.2026.51

Funding This work was supported by the Australian Research Council under grant DE240100042 and the China Scholarship Council under grant 202508240018.

1 Introduction

Combinatorial optimization underlies many scientific and engineering applications, including path finding [27] and resource allocation [9]. Many of these problems can be formulated as graph-structured edge-selection problems, where feasible solutions must satisfy explicit combinatorial constraints such as degree, budget, connectivity, and exclusivity requirements [5, 31]. Because many such problems are NP-hard, exact methods become increasingly expensive as instance sizes grow [37]. This motivates efficient methods for generating high-quality feasible solutions in many practical settings.

Recent learning-based methods have shown that predictive models can support hard combinatorial decisions in different ways. One line of work integrates learning into optimization algorithms, for example by guiding branching or search decisions [4, 15, 38]. These approaches can improve solver efficiency, but they remain tied to the optimization pipeline, where inference cost is still dominated by the underlying optimization process. Another line of work learns constructive policies that map problem instances directly to solutions [10, 22, 32, 33, 20]. This direction is attractive for fast solution generation, but many methods rely on supervised labels generated by strong solvers, which are costly to obtain and may



limit scalability across problem classes. As a result, there is still a need for approaches that are less dependent on expensive supervision.

To reduce reliance on solver-generated labels, self-supervised and other label-efficient neural approaches have therefore emerged as a promising direction [1, 7, 36]. One line of work learns autoregressive constructive policies that generate solutions sequentially, often improving performance through rollout-based training [24], diverse trajectory generation [26], or multiple inference-time evaluations [17]. Another line of work combines learned predictions with computationally intensive downstream procedures such as beam search [28], local search [30], or iterative refinement mechanisms [21, 39] to improve solution quality.

While effective, many of these methods rely heavily on downstream procedures for feasibility enforcement and solution improvement. As a result, final solution quality can depend substantially on search, repair, or refinement rather than on the learned prediction itself. This makes these approaches less suitable for settings where efficient single-pass construction is preferred or where the quality of the neural prediction itself matters. Moreover, many existing methods are closely tied to particular problem formulations or relaxation principles [25, 35, 19], or focus on constraint-handling designs tailored to specific problems [2, 30], which can limit their reuse across broader families of edge-selection problems. These observations reveal a gap in the current literature: existing methods provide limited support for reusable learning frameworks in which learned predictions themselves offer effective structural guidance for constructing feasible, high-quality solutions efficiently, without heavy reliance on downstream optimization procedures.

To address this gap, we propose a reusable self-supervised framework for edge selection and make the following contributions:

- **Constraint-aware self-supervised framework:** We introduce a framework for edge-selection optimization that learns directly from unlabeled instances through differentiable surrogate losses. It adapts across problem classes through task-specific objectives, constraints, and construction rules within a shared learning pipeline.
- **Lightweight cost-attention graph architecture:** We propose a lightweight graph architecture centered on a cost-attention convolution, where edge costs and global feasibility information directly shape message passing. This design incorporates optimization structure without relying on deep message passing.
- **Cross-problem instantiation under explicit constraints:** We instantiate the framework on the Traveling Salesman Problem (TSP), Orienteering Problem (OP), and Team Orienteering Problem (TOP) through task-specific self-supervised losses and single-pass decoders that enforce feasibility by construction. Across multiple spatial distributions, experiments against exact, heuristic, and learned algorithms demonstrate strong quality-efficiency trade-offs across constrained routing problems.

2 Problem Definition

We study optimization problem classes with a common graph-based structure. A problem class specifies the form of the objective function and the constraints shared by all instances in that class. Each instance is represented by a finite graph $G = (V, E)$, called the instance graph, with nodes $V = \{v_1, \ldots, v_n\}$ and edges $E = \{e_1, \ldots, e_m\} \subseteq V \times V$. The instance graph is augmented with node, edge, and global features that specify the instance-dependent objective coefficients and constraint parameters. In particular, nodes and edges may carry scalar features $\phi^V(i)$ and $\phi^E(i, j)$, such as rewards and travel costs, and a global feature vector $\phi^{\mathrm{G}} \in \mathbb{R}^{d_{\mathrm{G}}}$ may provide information shared across the graph, such as a budget.

Solutions: For a given instance graph $G = (V, E)$, a feasible or infeasible edge-selection decision is represented by a binary vector $x = (x_{e_1}, \ldots, x_{e_m})^\top \in \{0,1\}^m$, where each entry x_{e_k} indicates whether edge e_k is selected. Feasible solutions are those binary vectors that satisfy the problem-specific constraints.

Objective: The quality of a solution is measured by an objective function $f(G, x)$ that evaluates a graph G together with a binary edge selection vector $x \in \{0,1\}^m$. The objective may depend on the topology of G, on its node and edge features, and on which edges are chosen. A common example is a linear objective that sums the costs of the selected edges. Let $c(G) = (c_{e_1}, \ldots, c_{e_m})^\top \in \mathbb{R}^m$ denote the edge cost vector derived from the instance. The objective then takes the form $f(G, x) = \sum_{k=1}^{m} c_{e_k} x_{e_k}$. Other tasks may use reward-based objectives or introduce nonlinear terms that couple selected edges.

Constraints: Not every binary vector $x \in \{0,1\}^m$ corresponds to a valid solution for the instance. The feasible set $\mathcal{X}(G) \subseteq \{0,1\}^m$ contains exactly those edge selections that satisfy the constraints. These requirements are determined by the topology of G, the available features, and any additional parameters specified by the instance. For clarity, we express feasibility using a collection of constraint functions. Let Γ be an index set that enumerates all constraints associated with the instance. Each constraint is represented by a function $c_\gamma(G, x) \in \mathbb{R}$, where $\gamma \in \Gamma$, and a solution is feasible if all constraints are satisfied. The feasible set can be written as $\mathcal{X}(G) = \{x \in \{0,1\}^m \mid c_\gamma(G, x) \leq 0 \text{ for all } \gamma \in \Gamma\}$. The functions $c_\gamma(G, x)$ describe the specific rules that define validity for the problem. Depending on the task, these rules may enforce node degree patterns, resource or budget limits, connectivity requirements, or the exclusion of undesired substructures.

Unified Graph-structured Problem: Combining the components above, we express each instance through $\mathcal{I} = (G, \phi^V, \phi^E, \phi^G)$, together with an objective function $f(G, x)$ and a feasibility mapping $G \mapsto \mathcal{X}(G)$. The associated optimization problem is $\min_{x \in \mathcal{X}(G)} f(G, x)$. This formulation provides a common structure for problems in which decisions are made by selecting edges subject to graph-dependent constraints. Different tasks use different objectives and constraints, but they all share the same modeling pattern: making edge-selection decisions on a graph subject to problem-specific constraints.

3 Self-Supervised Graph Learning Framework

We now present a self-supervised learning framework for solving optimization problems as defined in Section 2.

3.1 Overview of the framework

Given an instance $\mathcal{I} = (G, \phi^V, \phi^E, \phi^G)$, the framework constructs a graph representation that incorporates both the topology of G and the associated features. A graph neural network (GNN) processes this representation and outputs a vector $P = P(G) \in [0,1]^m$, where P_{e_k} denotes the predicted likelihood that edge e_k should be included in the final solution. These values serve as a continuous relaxation of the discrete solution vector $x \in \{0,1\}^m$, and they enable gradient-based training even though the solution space is discrete.

Since P is not guaranteed to satisfy the constraints, the framework applies a deterministic constraint-aware decoder Dec that maps the continuous scores into a feasible solution $\hat{x} = \mathrm{Dec}(G, P) \in \mathcal{X}(G)$. The decoder uses symbolic operations such as sorting edges by

predicted probability, selecting edges while respecting constraints, and applying stitching or repair steps when necessary. This ensures that feasibility is always preserved. It does not require combinatorial search and therefore remains efficient. Additional search or refinement procedures can be incorporated on top of the decoder when higher solution quality is desired, at the cost of additional computation.

Since no ground-truth solutions are provided, the loss function, derived entirely from the optimization problem, contains two parts:

1. **Objective Term:** The continuous prediction P is evaluated through a soft objective surrogate $\mathcal{L}_{\text{obj}}(G, P)$, which encourages the predicted probabilities to yield solutions of high quality.
2. **Constraint-penalty Term with Dynamic Lagrange Multipliers:** To guide the continuous prediction P toward feasibility, we measure soft violations of each constraint and attach a learnable coefficient to it. For every constraint γ, let $c_\gamma(G, P)$ denote its violation computed directly from P. The penalty term is $\mathcal{L}_{\text{pen}}(G, P) = \sum_{\gamma \in \Gamma} \lambda_\gamma \, c_\gamma(G, P)$, where each $\lambda_\gamma \geq 0$ is a Lagrange multiplier maintained by the model. Notably, the multipliers are not static hyperparameters. After each forward pass, the average violation $\bar{v}_\gamma$ over the batch is used to update the multiplier by a dual-ascent rule, $\lambda_\gamma \leftarrow \big[\lambda_\gamma + \eta_\gamma(\bar{v}_\gamma - v^\star_\gamma)\big]_+$, where η_γ is a step size and $v^\star_\gamma$ is the target violation, typically 0. This update increases λ_γ when a constraint is consistently violated and decreases it when the model satisfies it. The mechanism mirrors the dual updates in classical augmented Lagrangian methods [3] and removes the need for manual tuning of penalty weights.

Together, they yield the self-supervised loss function $\mathcal{L}(G; P) = \mathcal{L}_{\text{obj}}(G, P) + \mathcal{L}_{\text{pen}}(G, P)$. This weighted sum allows the model to learn directly from the optimization problem, adapting the penalties during training.

3.2 Customized graph neural architecture

Each instance is represented as a complete graph on n nodes with an edge cost matrix $C \in \mathbb{R}^{n \times n}$, where $C_{ij} \geq 0$ gives the cost of moving from node i to node j. Each node has a scalar feature r_i that may represent reward or another form of utility, and the full node-feature vector is $r \in \mathbb{R}^n$. The instance may also include a global scalar feature that influences edge selection, such as a global budget $B > 0$ that limits the total accumulated edge weight in the decoded solution. Given (C, r, B), the network outputs a transition matrix $P \in [0, 1]^{n \times n}$, where P_{ij} is interpreted as the probability of selecting edge (i, j). Each row of P forms a probability distribution over outgoing edges, so P_{ij} should be interpreted as the conditional probability of choosing j given that the current node is i.

Budget-Conditioned Cost Sensitivity: The role of the edge weights depends strongly on the available budget. Under tight budgets, the model increases its sensitivity to costs and biases scores toward shorter edges. With more budget available, longer transitions may receive higher scores when they lead to more valuable regions of the graph. To incorporate this effect, the budget is transformed into a learnable scaling coefficient that controls how strongly the attention mechanism responds to edge costs.

We introduce two learnable positive coefficients, $\alpha = g_\alpha(B), \beta = g_\beta(B)$, where each mapping g_α and g_β is implemented as a two-layer multilayer perceptron (MLP) that takes the normalized budget as a one-dimensional input, applies a linear transformation, a smooth nonlinearity, and a final softplus activation to ensure positivity. The coefficient β controls how strongly the edge weights influence message passing inside the CAC layer, and α controls how much the edge weights penalize the final edge logits. Together they allow the network to

adapt its behavior under different budget regimes through the cost-attention and transition scoring mechanisms described later.

Cost-Attention Convolution (CAC): The CAC layer is the main architectural component. Its purpose is to use edge weights as attention signals that determine how information flows across the graph. Given initial embeddings $h^{(0)}$ and weight matrix C, the layer computes updated embeddings $h^{(1)}$ in two steps:

1. **Cost-based attention logits:** The layer assigns attention weights by applying a row-wise softmax to the cost matrix, $W = \mathrm{softmax}_{\mathrm{row}}(-\beta C)$, where W_{ij} denotes the attention weight from node i to node j. Lower-cost edges therefore receive larger attention weights, while the learnable coefficient β controls the cost sensitivity based on the budget. Consequently, each row of W defines a probability distribution over outgoing neighbors, inducing a cost-aware soft neighborhood structure.
2. **Cost-weighted message passing:** The attention matrix is then used to aggregate neighboring information, $m = Wr$, where $r \in \mathbb{R}^n$ denotes the scalar node features, such as rewards or other problem-specific signals. The aggregated features are then lifted into the hidden dimension through $h_i^{(1)} = \sigma(W_{\mathrm{self}} r_i + W_{\mathrm{neigh}} m_i)$, where $W_{\mathrm{self}}, W_{\mathrm{neigh}} : \mathbb{R} \to \mathbb{R}^d$ are learnable linear maps and σ is a pointwise nonlinearity. In this way, CAC jointly performs cost-weighted aggregation and feature lifting, allowing the cost structure to directly shape information propagation across the graph.

The CAC layer is computationally lightweight in practice, although not necessarily smaller in parameter count than standard GNN layers. In dense routing problems, the main cost comes from processing pairwise interactions over the complete graph, so methods often have similar asymptotic complexity. The difference lies in how this computation is carried out. Standard message-passing GNNs and attention-based layers typically operate edge by edge, which introduces substantial overhead from irregular gather, scatter, indexing, and aggregation operations over a large number of edges. By contrast, CAC is implemented through regular dense tensor operations directly on the cost matrix, together with simple transformations of node states and aggregated messages. This makes its computation more regular and substantially cheaper in wall-clock time. As a result, CAC is particularly well matched to the edge-selection problems studied in this paper, where the full pairwise cost structure is directly available and the goal is to exploit it efficiently without introducing the extra overhead of heavy edge-wise message passing. This practical advantage is also reflected in the later ablation study in Sec. 5.3, where CAC achieves a more favorable trade-off between solution quality and computational cost than the alternative graph encoders.

Bilinear Edge Scoring: After the CAC layer, each node is represented by an embedding $h_i^{(1)} \in \mathbb{R}^d$. Let $H^{(1)} \in \mathbb{R}^{n \times d}$ denote the matrix of node embeddings stacked row-wise. To compute structural preferences between node pairs, the network applies a learned bilinear interaction $S = H^{(1)} W_{\mathrm{edge}}^{\top} (H^{(1)})^{\top} \in \mathbb{R}^{n \times n}$, where $W_{\mathrm{edge}} \in \mathbb{R}^{d \times d}$ is a learnable weight matrix. Equivalently, the score for edge (i, j) is $s_{ij} = (h_i^{(1)})^{\top} W_{\mathrm{edge}}^{\top} h_j^{(1)} = \left(W_{\mathrm{edge}} h_i^{(1)}\right)^{\top} h_j^{(1)}$. These scores provide a learned measure of how compatible it is to move from node i to node j, based on the utility and structural information encoded in the embeddings. Similar bilinear interaction mechanisms appear in prior GNNs such as BGNN [40], where pairwise relationships are modeled through learned bilinear maps.

Cost-Penalized Logits and Output Probabilities: The structural scores are adjusted by a cost penalty, and the final transition probabilities are obtained through a row-wise softmax, $P = \mathrm{softmax}_{\mathrm{row}}(S - \alpha C)$, where costly edges receive a stronger negative correction when the budget is tight. The resulting probability matrix acts as a differentiable relaxation of edge selections, reflecting how utility, cost, and budget jointly influence feasible movements.

In summary, the architecture consists of three interacting components: a node embedding module that encodes local utility, a CAC layer that propagates information through cost-sensitive attention, and a bilinear scoring module that produces edge-level logits further adjusted by budget-dependent cost penalties. Together these components form a compact and interpretable model designed for graph-structured optimization tasks.

4 Problem-specific Instantiation

We next instantiate the framework for the TSP, OP, and TOP. For each task, we specify how instances are mapped to graph features, how the generic self-supervised loss functions are specialized, and how a simple decoder recovers a feasible solution.

4.1 Mapping to the framework

TSP: The TSP asks for a minimum-cost Hamiltonian cycle on a set of n cities [6]. An instance is given by a cost matrix $C \in \mathbb{R}^{n\times n}$, where C_{ij} encodes the travel cost from city i to city j. We represent the instance as a complete graph with edge features given by C. Since there are no explicit node features, we assign each node a scalar statistic $C_i^{\mathrm{aggr}} = \frac{1}{n}\sum_{j=1}^{n} C_{ij}$, which summarizes its typical distance to the rest of the graph and serves as the node feature input to the CAC layer. Because there is no explicit budget, the cost-sensitivity parameters α and β are kept fixed to 1 across instances. The CAC produces node embeddings from C^{aggr}, and the bilinear edge scoring with a cost penalty yields a soft transition matrix $P \in [0,1]^{n\times n}$ in which self-loops are suppressed by masking $P_{ii} = 0$.

OP: The OP asks for a budget-feasible route that starts and ends at a depot, visits a subset of nodes at most once, and maximizes the collected reward [12]. An instance defines a cost matrix $C \in \mathbb{R}^{n\times n}$, a nonnegative reward vector $r \in \mathbb{R}^n$, and a travel budget $B \geq 0$. We take r_i as the scalar node feature for node i, and use C directly as the edge-cost information for CAC and the loss. The budget B is treated as the global feature that is mapped to cost-sensitivity coefficients α and β. The CAC processes the reward features and aggregates information through cost-aware attention, and the subsequent bilinear scoring with budget-dependent cost penalties produces a soft transition matrix $P \in [0,1]^{n\times n}$. In contrast to the TSP setting, we do not mask the diagonal entries; they are interpreted as the model assigning probability to not visiting a node.

TOP: The TOP extends OP to a fleet of K vehicles that share a common depot and node rewards, but have separate route budgets [12]. The instance specifies the same C and r as in OP, together with per-vehicle budgets $B_1, \ldots, B_K$ and the number of vehicles K. We again use r as node features and C as edge-cost information. A single CAC layer is applied once to the graph and produces shared node embeddings that encode both reward and cost structure. The budgets $B_1, \ldots, B_K$ are summarized through a one-layer MLP into a global scalar β that modulates the strength of cost-aware attention in CAC, while the per-vehicle cost penalties use distinct coefficients α_k obtained from their individual budgets. On top of the shared embeddings and these budget signals, vehicle-specific bilinear scoring and cost penalties

produce soft transition matrices $P^{(1)}, \ldots, P^{(K)} \in [0,1]^{n \times n}$, where $P_{ij}^{(k)}$ is interpreted as the relaxed probability that vehicle k moves from i to j. Vehicles share the same graph representation but differ through their budget-conditioned scoring, which prevents them from collapsing to identical routing patterns. As in the OP setting, the diagonal entries of each $P^{(k)}$ are interpreted as assigning probability mass to not visiting the corresponding node.

4.2 Self-supervised loss functions

The self-supervised loss couples a soft objective surrogate with penalties capturing structural and resource constraints.

TSP: For a single transition matrix $P \in [0,1]^{n \times n}$ and cost matrix $C \in \mathbb{R}^{n \times n}$, the relaxed total cost is

$$\mathcal{L}_{\text{obj}} = \sum_{i=1}^{n} \sum_{j=1}^{n} P_{ij}\, C_{ij}, \tag{1}$$

which can be interpreted as the expected path cost under the transition probabilities P. To encourage single visits, we penalize deviations of the incoming probability mass from 1. Let node 1 denote the depot. For non-depot nodes $j = 2, \ldots, n$, the visit penalty is

$$\mathcal{L}_{\text{visit}} = \sum_{j=2}^{n} \Big| \sum_{i=1}^{n} P_{ij} - 1 \Big|, \tag{2}$$

which drives the incoming mass at each non-depot node, among the $n-1$ such nodes, toward 1, thereby encouraging each node to be visited exactly once. This constraint plays the role complementary to the row-wise softmax, ensuring that the transition matrix respects the one-visit structure also adopted in [30].

To suppress subtours we use two complementary penalties, each targeting a different way in which the soft adjacency P can violate the single-cycle structure. The first term is a trace-based cycle penalty. We view each row of P as a distribution over the next city, so P describes a one-step Markov transition. The k-step transition probabilities are given by the matrix power P^k. In particular, the entry $(P^k)_{ii}$ is the probability that a random walk starting at city i returns to i after exactly k steps. If there is a short cycle of length k that stays inside a small subset of nodes, then some of these diagonal entries will become large. Summing the diagonal gives $\operatorname{tr}(P^k) = \sum_{i=1}^{n} (P^k)_{ii}$, which aggregates the return probabilities over all nodes. A large trace for small k therefore indicates that the mass of P is trapped in short cycles rather than spread along a long tour. We penalize this by

$$\mathcal{L}_{\text{sub}}^{\text{trace}} = \sum_{k=2}^{K} \Big(\operatorname{tr}\big(P^k\big) \Big)^{1/k}, \tag{3}$$

where the exponent $1/k$ reduces the scale difference between different powers and keeps the contribution of each k comparable. This term pushes the model away from placing high probability on short, high-frequency cycles. The upper limit K controls the trade-off between structural coverage and computational cost, and is set to 5 in our experiments for training efficiency.

The second term is a cut-based penalty that detects subsets of nodes that behave like almost closed tours. For any subset $S \subset \{1, \ldots, n\}$, the cut value $\sum_{i \in S} \sum_{j \notin S} P_{ij}$ measures the flow from S to its complement. If S behaves like a closed cycle, most transition mass

stays within S and only a small amount leaves, resulting in a small cut(S). A subtour in TSP is precisely a cycle whose nodes have no outgoing edges to the rest of the graph. With the soft relaxation, an analogous pattern arises when the cut value is small, indicating that probability mass remains trapped inside S instead of flowing outward to complete a full tour. During training we sample random subsets at each iteration. Penalizing small cut values across sampled subsets trains the model to avoid allocating transition mass in patterns that cause any subset to function as an isolated cycle. The combined penalty for all sampled subsets can be written as

$$\mathcal{L}_{\text{sub}}^{\text{cut}} = \sum_{S \in \mathcal{S}} \max\Big\{0,\ \tau - \sum_{i \in S} \sum_{j \notin S} P_{ij}\Big\}, \tag{4}$$

where $\mathcal{S}$ is the collection of subsets drawn at the current iteration and τ is a soft training threshold for the cut penalty. While the classical subtour constraint uses a lower bound of 1, we set $\tau = 1.5$ in our experiments to more strongly discourage nearly closed subsets during training. Together, the trace-based term discourages short local cycles in the transition dynamics, while the cut-based term discourages nearly closed regions with little flow leaving the subset. We obtain the **TSP loss**: $\mathcal{L}_{\text{TSP}} = \mathcal{L}_{\text{obj}} + \lambda_{\text{visit}}\mathcal{L}_{\text{visit}} + \lambda_{\text{sub}}^{\text{trace}}\mathcal{L}_{\text{sub}}^{\text{trace}} + \lambda_{\text{sub}}^{\text{cut}}\mathcal{L}_{\text{sub}}^{\text{cut}}$.

OP and TOP: For OP and TOP, recall that the diagonal entries represent the probability of not visiting a node, so these entries are masked out when computing all loss terms. Given a budget B, cost violations are discouraged through

$$\mathcal{L}_{\text{budget}} = \max\Big\{0,\ \sum_{i=1}^{n}\sum_{j=1}^{n} P_{ij}\, C_{ij} - B\Big\}, \tag{5}$$

where the double sum represents the expected travel cost, consistent with Eq. 1. Moreover, we encourage the depot to function as both the start and end of the route. Recall that node 1 denotes the depot; its soft degree is defined as the sum of incoming and outgoing probability mass with the self-loop removed. The corresponding penalty is

$$\mathcal{L}_{\text{depot}} = \Big|\sum_{j=1}^{n} P_{1j} - 1\Big| + \Big|\sum_{i=1}^{n} P_{i1} - 1\Big|, \tag{6}$$

which encourages exactly one departure and one return at the depot.

The **soft visit probability** is defined by the column sum $\sum_i P_{ij}$, which collects the probability mass entering node j. Every node may be visited at most once, so we penalize only cases in which the incoming soft visitation exceeds 1 as

$$\mathcal{L}_{\text{revisit}} = \sum_{j=2}^{n} \max\Big\{0,\ \sum_{i=1}^{n} P_{ij} - 1\Big\}. \tag{7}$$

This penalty activates only when a non-depot node, among the $n-1$ such nodes, receives more than one unit of incoming probability mass. Subsequently, we introduce the relaxed objective for OP through

$$\mathcal{L}_{\text{obj}} = -\sum_{j=1}^{n}\Big(\sum_{i=1}^{n} P_{ij}\Big) r_j, \tag{8}$$

which provides a differentiable relaxation of the discrete objective "sum of rewards over visited nodes." The main driving term of the loss is the negative soft reward. We arrive at the **OP loss**: $\mathcal{L}_{\text{OP}} = \mathcal{L}_{\text{obj}} + \lambda_{\text{budget}}\,\mathcal{L}_{\text{budget}} + \lambda_{\text{depot}}\,\mathcal{L}_{\text{depot}} + \lambda_{\text{revisit}}\,\mathcal{L}_{\text{revisit}}$.

Extending from OP to TOP, each vehicle k maintains its own transition matrix $P^{(k)}$ and budget B_k. All OP loss terms—revisit avoidance (Eq. 7), budget penalty (Eq. 5), depot start-end penalty (Eq. 6), and soft reward maximization (Eq. 8)—generalize naturally by applying the same definitions to each $P^{(k)}$ and summing over vehicles. In addition, since each customer may be visited by at most one vehicle, we penalize nodes whose aggregated soft visitation mass exceeds one. Using soft visit probabilities, the node-assignment penalty is

$$\mathcal{L}_{\text{assign}} = \sum_{j=2}^{n} \max\Big\{0,\ \sum_{k=1}^{K}\big(\sum_{i=1}^{n} P_{ij}^{(k)}\big) - 1\Big\}, \tag{9}$$

which activates only when the combined soft visit probability mass assigned to a node exceeds 1, indicating that more than one vehicle is attempting to visit the same node. Extending the OP loss to multiple vehicles and adding a node-assignment penalty yields the following **TOP loss**: $\mathcal{L}_{\text{TOP}} = \mathcal{L}_{\text{obj}} + \lambda_{\text{budget}}\mathcal{L}_{\text{budget}} + \lambda_{\text{depot}}\mathcal{L}_{\text{depot}} + \lambda_{\text{revisit}}\mathcal{L}_{\text{revisit}} + \lambda_{\text{assign}}\mathcal{L}_{\text{assign}}$.

We do not include subtour penalties for OP or TOP. Under the soft formulation, placing probability mass on small closed walks that do not contribute to reward only raises the expected travel cost and is further discouraged by the degree and revisit penalties. Explicit subtour penalties would duplicate the effects of existing terms and showed no performance gains in preliminary experiments.

4.3 Decoders

The decoders map the soft transition scores produced by the network to discrete solutions that satisfy the problem constraints. To better expose the quality of the learned solution structure, we focus on lightweight single-pass decoding without local search or repair heuristics. Accordingly, all three tasks use single-pass greedy procedures that construct feasible solutions directly from the predicted transition scores.

TSP: The TSP decoder treats every entry P_{ij} as an edge score, sorts all node pairs (i, j) once, and scans them greedily. A union-find structure maintains connected components and each node is restricted to have degree at most two. For the first $n-1$ accepted edges, an edge (i, j) is added only if i and j belong to different components; the last edge is accepted only if it closes a single cycle over all nodes. Once n edges are selected, the resulting adjacency is a connected 2-regular graph that uniquely determines the tour. The overall complexity is $O(n^2 \log n)$ due to sorting all n^2 scores; the subsequent union-find scan runs in $O(n^2)$ time.

OP: The OP decoder constructs a feasible route using a single greedy pass. Starting at the depot, it repeatedly considers all unvisited nodes whose inclusion keeps the remaining-cost-to-return within budget, and moves to the node with the highest transition probability P_{ij} excluding diagonal entries among this candidate set. If no feasible candidate remains, the decoder returns to the depot and terminates. This procedure is budget-feasible by construction, visits each node at most once, and runs in $O(n^2)$ time per instance.

TOP: The TOP decoder builds up to K feasible routes in a single sequential pass over the vehicles, using the soft transition matrices $P^{(1)}, \ldots, P^{(K)}$. It maintains a global set of already visited nodes to enforce cross-vehicle exclusivity. For each vehicle k in a fixed order, the decoder performs an OP-style greedy walk with an additional mask that forbids all nodes visited by previous vehicles. At each step, it selects the highest-probability feasible

move under $P^{(k)}$ that keeps the remaining-cost-to-return within the vehicle's budget; if no such move exists, the vehicle returns to the depot and its route is finalized. The overall complexity is $O(Kn^2)$ per instance, and the resulting routes satisfy per-vehicle budget limits and one-visit-per-node exclusivity by construction.

5 Experiments

This section presents the experimental setup and reports the empirical results.

5.1 Experimental Setup

Data Generation: Synthetic instances are adopted because no unified benchmark exists across the three routing tasks, and available datasets vary in scale and constraints. This also enables controlled comparisons under consistent settings and permits systematic evaluation of distributional generalization via structured geometric shifts. We generate instances by sampling coordinates in the unit square $[0, 1]^2$. The training and validation sets are drawn only from a uniform distribution over this domain. To evaluate generalization under distribution shift, we construct four additional geometric layouts, illustrated in Figure 1. The *Clusters* distribution samples points from several Gaussian components with randomly chosen means and small isotropic variance, producing well-separated node groups. The *Ball* distribution concentrates points in a compact neighborhood around the center of the square. The *Donut* distribution places points on an annulus with mild radial perturbations. The *Waves* distribution arranges points along several approximately concentric rings centered in the plane with mild radial perturbations. Given the sampled coordinates, we compute the cost matrix using Euclidean distances. We designate the first node in the index ordering as the fixed start and end points (depot) of the route.

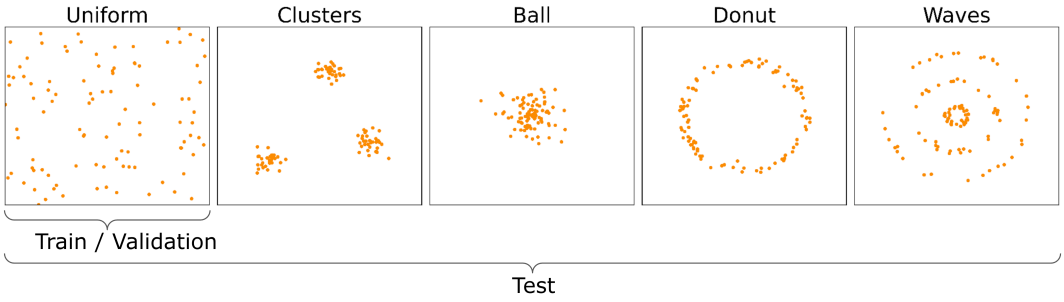


Figure 1 Spatial distributions used for evaluating generalization across geometric layouts. The training set contains uniformly sampled instances, while the test set covers both the uniform distribution and four unseen geometric patterns (Clusters, Ball, Donut, Waves) to assess robustness under distribution shift.

For OP and TOP, each instance requires a reward value at every non-depot node. We generate these rewards so that they reflect geometry-dependent variability commonly observed in routing tasks. For node i, we first compute its average incoming distance $d_i^{\text{in}} = \frac{1}{n-1} \sum_{j \neq i} C_{ji}$, as a geometry-based measure of accessibility. Node rewards are then generated by perturbing these values with additive Gaussian noise, $r_i = \max\{d_i^{\text{in}} + \epsilon_i,\ 0\}$, where $\epsilon_i \sim \mathcal{N}(0,\ 0.1^2)$. This introduces controlled variability while keeping the reward landscape aligned with the spatial structure.

Each OP and TOP instance also requires a travel budget. We aim to test on meaningful instances such that each problem does not degenerate into a pure TSP or into a regime

where only a few nodes are reachable. To obtain a geometry-dependent scale for setting the budget, we compute the sum of nearest-neighbor distances $\ell_{\text{nn}} = \sum_{i=1}^{n} \min_{j \neq i} C_{ij}$, which serves as a natural lower bound on the cost of visiting all nodes, since any tour must include at least one outgoing edge per node and no edge can be shorter than its nearest neighbor. We therefore use ℓ_{nn} for setting the budget. In practice, this bound is optimistic because chaining nearest-neighbor steps rarely produces a low-cost tour. Consequently, the budget is usually insufficient to visit every node, though many are expected to be visited. This creates a non-trivial routing problem in which the model must decide not only which nodes to visit, but also how to allocate the travel budget. For TOP, we test on $K = 5$ and each vehicle receives a per-vehicle budget obtained by scaling the total budget by $1/K$ and adding a small amount of independent noise ($\epsilon_k \, B/K$, where $\epsilon_k \sim \text{Uniform}(-0.3, 0.3)$), so that different vehicles have heterogeneous travel limits, which also helps avoid symmetric solutions.

We train the model on 10,000 unlabeled synthetic instances and use 100 validation instances for model selection. Test sets contain 500 instances (100 per distribution) for TSP and OP, and 100 instances (20 per distribution) for TOP.

Baselines: We compare against complementary baselines to evaluate solution quality, runtime, and the contribution of the learned component. These include a greedy heuristic as a lightweight constructive reference, Gurobi as an exact optimization benchmark, and for TSP, representative learning-based and specialized heuristic methods. Together, they place the proposed framework relative to simple constructive rules, exact optimization, and stronger task-specific alternatives.

For all three problem classes, we implement a standard greedy baseline. For TSP, the route begins at the depot and repeatedly moves to the nearest unvisited node under the edge cost, continuing until all nodes have been visited. For OP, the procedure ranks unvisited nodes according to the reward gained per unit of incremental cost and selects the best feasible move at each step while checking that a return to the depot remains within the remaining budget. For TOP, the same rule is applied sequentially for each vehicle, with each vehicle constructing a route until its individual budget becomes insufficient for additional visits. All three greedy variants run in $O(n^2)$ time per instance.

Gurobi (version 12.0.3) [13] serves as the reference solver. Each of the three problem classes is formulated as a mixed-integer program and solved directly. To keep the formulations practical at our instance scales, we apply light structural preprocessing while preserving the original optimization objectives. For TSP, we adopt a standard degree-2 formulation and eliminate subtours through DFJ-type lazy constraints [8], which add cutting planes only when a candidate solution contains a subtour. For OP, we use MTZ [29] ordering variables to rule out subtours. For TOP, we employ a multi-vehicle orienteering formulation with per-vehicle budget limits and single-commodity flow constraints, combined with preprocessed budget-aware arc pruning and a greedy warm start to improve solve time. All TSP and OP instances are solved without a time limit and to proven optimality. TOP is more difficult to solve, therefore we enforce a 1% MIP gap to keep solve times manageable.

For TSP, UTSP [30] and AGFN [39] are the two learned heuristic baselines most comparable to our approach: UTSP learns a permutation policy, and AGFN uses an adversarially trained network; both methods produce a model output that is then converted into a feasible solution by a construction procedure. However, their full published pipelines typically include substantial post-model search or sampling stages, which can significantly affect both final quality and runtime beyond the contribution of the learned policy itself. To preserve logical comparability and focus the evaluation on the learning component, we therefore also evaluate

the single-pass variants of UTSP and AGFN, denoted by UTSP_1 and AGFN_1, under a unified protocol: after model output, we apply only a single-pass construction step chosen to match the semantics of the model output described in the respective papers, without additional search, multi-start sampling, or iterative refinement. This setting provides a cleaner comparison of the learned component across methods. In addition, LKH3 [16] is a widely used heuristic solver for TSP. For OP and TOP, we compare only against Gurobi and the greedy heuristic, as we are not aware of directly comparable learned baselines under the same evaluation setting.

Evaluation Metrics: We evaluate performance for each optimization problem using the following metrics:

1. **Average Relative Gap:** For each instance, let $f_i^\star$ denote the best bound returned by Gurobi (optimal for TSP and OP, and within a 1% MIP gap tolerance for TOP), and f_i the value produced by the evaluated method. We define the relative optimality gap as $\text{gap}_i = \frac{|f_i - f_i^\star|}{f_i^\star}$. The average optimality gap is computed by taking the mean of gap_i over the test set.
2. **Worst-Case Relative Gap:** To assess robustness, we report the maximum deviation over all test instances, $\text{gap}_{\max} = \max_i \text{gap}_i$, which captures the most challenging cases under each method.
3. **Solve Time:** The wall-clock time required to produce feasible solutions for the full test set, which evaluates the computational efficiency of the method.

Implementation Details: Training runs for 100 epochs using Adam [23] with a learning rate of 10^{-4} and batch size 64, and the checkpoint with the best validation average gap is selected. The architecture uses a single CAC layer, and all MLP components have hidden dimension 16. Dynamic Lagrange coefficients are updated by dual ascent with a fixed step size of 10^{-3} to drive average constraint violations toward zero. All experiments are implemented in Python 3.12.9 with PyTorch 2.10.0 and executed on an Apple M3 Pro with 18 GB of main memory.

5.2 Experimental Results

Table 1 reports performance on both in-distribution and out-of-distribution test sets across four optimization tasks. Overall, our method consistently achieves the best solution quality among the fast single-pass learned and heuristic baselines, while remaining much faster than exact or search-heavy solvers. On TSP, the most relevant learned comparisons are the single-pass variants UTSP_1 and AGFN_1, which exclude additional search or refinement and therefore better reflect the quality of the learned component itself. Under this setting, our method outperforms both baselines on TSP-100 and TSP-200 in both average and worst-case gap, for both in-distribution and out-of-distribution instances. On TSP-100, for example, the average in-distribution gap decreases from 19.91% with AGFN_1 and 16.69% with UTSP_1 to 7.02%, and on the out-of-distribution set it decreases from 24.05% and 18.73% to 8.53%, respectively. Similar margins are observed on TSP-200. These results suggest that the proposed framework learns stronger solution representations, rather than depending on downstream search to recover solution quality, and that these representations transfer more reliably under distribution shift. The gap to LKH3 remains substantial, especially on larger TSP instances, showing that a single-pass learned constructor has not yet matched the best specialized search-based solver in raw solution quality.

Table 1 Performance comparison on four optimization tasks. The table reports average and worst-case relative gaps on in-distribution (ID) and out-of-distribution (OOD) test sets, inference time for machine learning methods, and feasible solution search runtime.

Task	Method	% ID Gap		% OOD Gap		Total runtime	
		Avg	Worst	Avg	Worst	Inference (s) +	Search (h:m:s)
TSP-100	Gurobi	–	–	–	–	–	3:20
	LKH3	0.00	0.00	0.00	0.00	–	4:20
	Greedy	24.08	43.90	26.91	53.62	–	0.23
	UTSP	0.02	0.43	0.04	0.54	0.40 +	52.45
	$UTSP_1$	16.69	30.78	18.73	33.54	0.38 +	0.72
	AGFN	9.04	17.02	14.74	25.93	2.07 +	7.50
	$AGFN_1$	19.91	39.75	24.05	47.91	1.91 +	0.93
	Ours	7.02	12.62	8.53	16.75	0.06 +	0.71
TSP-200	Gurobi	–	–	–	–	–	30:25
	LKH3	0.00	0.02	0.00	0.00	–	23:53
	Greedy	25.48	48.35	28.36	51.37	–	0.95
	UTSP	0.16	0.96	0.27	1.30	0.89 +	12:00.76
	$UTSP_1$	19.52	30.99	36.78	64.61	1.01 +	2.93
	AGFN	10.57	14.79	16.64	42.71	3.58 +	27.84
	$AGFN_1$	16.46	28.26	24.71	50.78	3.81 +	2.91
	Ours	8.24	13.54	9.08	15.78	0.14 +	2.82
OP-100	Gurobi	–	–	–	–	–	25:57:00
	Greedy	29.78	41.63	28.89	46.97	–	0.54
	Ours	3.06	6.40	6.05	20.69	0.12 +	1.04
TOP-100	Gurobi	–	–	–	–	–	38:37:00
	Greedy	22.47	45.09	22.06	56.41	–	0.15
	Ours	6.49	16.42	8.47	43.01	0.04 +	0.49

The results on OP-100 and TOP-100 show a similar pattern. Compared with greedy, our method reduces the average in-distribution gap from 29.78% to 3.06% on OP-100 and from 22.47% to 6.49% on TOP-100, with corresponding improvements also on out-of-distribution instances. In terms of efficiency, total solve times remain within a few seconds across all tasks, while the neural network forward pass itself takes only a fraction of a second. Overall, Table 1 shows that the proposed learning framework achieves a strong quality-efficiency trade-off: it learns stronger solution representations than simple constructive heuristics and comparable single-pass learned baselines, while remaining far faster than exact optimization and search-heavy methods.

Across all four layouts (Fig. 2), the learned transition structure closely aligns with the geometry of the optimal tour. The model behaves consistently across distributions, suggesting that it has learned geometric principles that transfer across layouts rather than memorizing features tied to any single distribution. Notably, in Clusters and Donut, the model avoids placing probability mass on edges that cut across empty regions, indicating that it correctly learns large-scale geometric constraints. In Ball and Waves, the predicted transitions align closely with the main traversal routes of the optimal tour, concentrating probability on the same high-utility edges. Overall, the strong agreement between high-probability edges and the optimal tour structure suggests that the learned policy, once decoded, is likely to find near-optimal solutions.

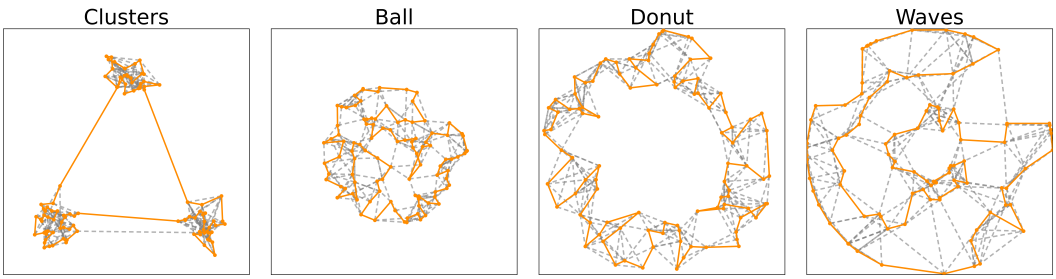


Figure 2 Comparison between optimal TSP tours and model predictions on instances belonging to each layout. The optimal Hamiltonian cycles obtained by Gurobi are shown as solid edges, while the top-3 outgoing edges predicted for each node are shown as dashed edges.

Overall, the results show that the proposed framework achieves a favorable balance between quality and efficiency, consistently outperforming greedy heuristics, generalizing beyond the training distribution, and delivering near real-time inference without substantial search or post-processing.

5.3 Ablation Study

This section presents the ablation study for the proposed CAC layer and the constraint-aware learning components.

CAC Ablation: Table 2 evaluates the contribution of the CAC encoder under a fixed training budget on TSP-100. We keep all non-architectural factors unchanged, including the data, loss function, and decoding procedure, so the only difference across variants is the graph convolutional layer, namely CAC, MLP, SAGEConv [14], GINEConv [18], or GATConv [34], implemented using PyTorch Geometric [11]. For a fair comparison, every variant is trained for exactly 10 epochs, which gives all methods the same number of parameter update steps. Under this controlled setting, differences in training time mainly reflect the computational cost of the architecture itself. We report average and worst-case relative gaps to measure solution quality, together with training time and inference latency to assess efficiency, which allows us to compare both effectiveness and computational cost in a unified way.

Table 2 CAC ablation on TSP-100 with solution quality and efficiency metrics. Lower is better for gap, training time, and inference latency.

Variant	Avg (%)	Worst (%)	Training Time (s)	Inference Time (ms)
CAC	8.21	12.53	41.27	0.0605
MLP	459.08	569.17	35.63	0.0323
SAGEConv	10.34	18.41	204.91	0.7677
GINEConv	14.48	19.32	350.04	1.5089
GATConv	8.43	13.13	453.77	1.7314

The results in Table 2 show that CAC provides the best overall trade-off between solution quality and efficiency. The proposed model achieves the lowest average gap among all replacement architectures under the same 10-epoch training budget, while maintaining substantially lower training and inference cost than most graph convolution alternatives. In contrast, replacing CAC with a MLP leads to a dramatic degradation, with both average and

worst-case gaps increasing sharply, which suggests that simple node-wise transformations are insufficient and that explicitly modeling graph structure is necessary for high-quality construction. Among the graph-based alternatives, GATConv is the closest to CAC in solution quality, but it is much more expensive, requiring the longest training time and the highest inference latency. SAGEConv and GINEConv also remain clearly worse than CAC in both average and worst-case performance, while still introducing substantial computational overhead. These results suggest that the gain from CAC does not come from using a heavier encoder, but from a more suitable inductive bias for this task.

Constraint-Aware Components Ablation: For Table 3, we keep the backbone architecture and decoder fixed. We only ablate constraint-aware components: selected loss terms and, for OP/TOP, budget-conditioned CAC (implemented by setting $\alpha = \beta = 1$). We train every variant for exactly 10 epochs, so all methods use the same number of parameter-update steps. This gives a fair optimization budget and highlights learning efficiency in early training. It also makes each component's effect on convergence speed and stability easier to compare.

Table 3 Ablation on constraint-aware components in the self-supervised framework.

Ablation variant	TSP-100		OP-100		TOP-100	
	Avg (%)	Worst (%)	Avg (%)	Worst (%)	Avg (%)	Worst (%)
Full loss (objective + all constraints)	8.21	12.53	6.56	13.48	11.17	24.96
Objective only (remove all constraint terms)	50.65	71.72	18.84	25.10	59.36	65.79
w/o subtour terms ($\mathcal{L}_{\text{sub}}^{\text{trace}}, \mathcal{L}_{\text{sub}}^{\text{cut}}$)	9.82	17.57	–	–	–	–
w/o budget term ($\mathcal{L}_{\text{budget}}$)	–	–	8.53	18.34	30.69	55.57
w/o budget-conditioned CAC ($\alpha = \beta = 1$)	–	–	7.48	19.19	60.48	72.55
w/o visit-related terms ($\mathcal{L}_{\text{depot}}, \mathcal{L}_{\text{revisit}}, \mathcal{L}_{\text{assign}}^{TOP}$)	–	–	34.32	44.46	29.13	45.83

Under the fixed 10-epoch budget, the full constraint-aware setup is best overall on TSP-100, OP-100, and TOP-100. Removing all constraint terms causes large degradations, especially on TSP-100 (average gap: 8.21% to 50.65%) and TOP-100 (11.17% to 59.36%), which shows that objective-only training is insufficient. Task-specific constraints also matter: removing TSP subtour terms worsens both average and worst-case gaps, and removing OP visit-related terms leads to a major collapse (6.56% to 34.32%). For budget-constrained tasks, the key factor differs by complexity. In OP-100, removing budget terms causes moderate drops, while in TOP-100 removing budget-conditioned CAC is severe (11.17% to 60.48%), indicating that budget information should be injected into representation learning, not only into the loss. Overall, Table 3 shows that explicit constraints improve both solution quality and robustness, and that the most critical component depends on the task structure.

6 Conclusion

This paper presented a reusable self-supervised framework for edge-selection optimization, supported by a lightweight cost-attention graph architecture. Across TSP, OP, and TOP, the framework achieved strong solution quality and efficient inference under lightweight single-pass decoding. The results suggest that directly learning feasibility-aware solution structure can provide an effective alternative to search-heavy optimization pipelines for edge-selection problems. Future work includes extending the framework toward more general settings with less task-specific design.

References

1 Sen Bai, Chunqi Yang, Xin Bai, Xin Zhang, and Zhengang Jiang. Deep k-grouping: An unsupervised learning framework for combinatorial optimization on graphs and hypergraphs, 2025. URL: https://arxiv.org/abs/2505.20972, arXiv:2505.20972.

2 Jieyi Bi, Yining Ma, Jianan Zhou, Wen Song, Zhiguang Cao, Yaoxin Wu, and Jie Zhang. Learning to handle complex constraints for vehicle routing problems. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 93479–93509. Curran Associates, Inc., 2024. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/a9d2a5fd12d34250c21b5e4fa8d906b0-Paper-Conference.pdf, doi:10.52202/079017-2964.

3 E. G. Birgin and J. M. Martínez. *Practical Augmented Lagrangian Methods for Constrained Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2014. doi:10.1137/1.9781611973365.

4 Junyang Cai, Taoan Huang, and Bistra Dilkina. Learning backdoors for mixed integer linear programs with contrastive learning. In *Proceedings of the ECAI 2024*. IOS Press, 2024. doi:10.3233/FAIA240768.

5 Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023. URL: http://jmlr.org/papers/v24/21-0449.html.

6 Vašek Chvátal, William Cook, George B. Dantzig, Delbert R. Fulkerson, and Selmer M. Johnson. *Solution of a Large-Scale Traveling-Salesman Problem*, pages 7–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-540-68279-0_1.

7 Andrea Corsini, Angelo Porrello, Simone Calderara, and Mauro Dell'Amico. Self-labeling the job shop scheduling problem. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

8 G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954. URL: http://www.jstor.org/stable/166695.

9 Frits de Nijs and Peter J. Stuckey. Risk-aware conditional replanning for globally constrained multi-agent sequential decision making. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 303–311, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems.

10 Jian-Ya Ding, Chao Zhang, Lei Shen, Shengyin Li, Bing Wang, Yinghui Xu, and Le Song. Accelerating primal solution findings for mixed integer programs based on solution prediction. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1452–1459, Apr. 2020. doi:10.1609/aaai.v34i02.5503.

11 Matthias Fey, Jinu Sunil, Akihiro Nitta, Rishi Puri, Manan Shah, Blaž Stojanović, Ramona Bendias, Alexandria Barghi, Vid Kocijan, Zecheng Zhang, Xinwei He, Jan Eric Lenssen, and Jure Leskovec. Pyg 2.0: Scalable learning on real world graphs. *Temporal Graph Learning Workshop @ KDD*, 2025.

12 Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, 255(2):315–332, 2016. doi:10.1016/j.ejor.2016.04.059.

13 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL: https://www.gurobi.com.

14 William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

15 Qingyu Han, Linxin Yang, Qian Chen, Xiang Zhou, Dong Zhang, Akang Wang, Ruoyu Sun, and Xiaodong Luo. A gnn-guided predict-and-search framework for mixed-integer linear programming, 2023. URL: https://arxiv.org/abs/2302.05636, arXiv:2302.05636.

16 Keld Helsgaun. *An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems: Technical report*. Roskilde Universitet, December 2017.

17 André Hottung, Mridul Mahajan, and Kevin Tierney. Polynet: Learning diverse solution strategies for neural combinatorial optimization, 2025. URL: `https://arxiv.org/abs/2402.14048`, `arXiv:2402.14048`.

18 Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks, 2020. URL: `https://arxiv.org/abs/1905.12265`, `arXiv:1905.12265`.

19 Yuma Ichikawa. Controlling continuous relaxation for combinatorial optimization. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

20 Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem, 2019. URL: `https://arxiv.org/abs/1906.01227`, `arXiv:1906.01227`.

21 Nikolaos Karalias and Andreas Loukas. Erdős goes neural: an unsupervised learning framework for combinatorial optimization on graphs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

22 Elias B. Khalil, Christopher Morris, and Andrea Lodi. Mip-gnn: A data-driven framework for guiding combinatorial solvers, 2022. URL: `https://arxiv.org/abs/2205.14210`, `arXiv:2205.14210`.

23 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL: `https://arxiv.org/abs/1412.6980`, `arXiv:1412.6980`.

24 Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems!, 2019. URL: `https://arxiv.org/abs/1803.08475`, `arXiv:1803.08475`.

25 Olga Krylova and Frank Phillipson. Unsupervised learning with gnns for qubo-based combinatorial optimization. *EURO Journal on Computational Optimization*, 13:100116, 2025. `doi:10.1016/j.ejco.2025.100116`.

26 Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 21188–21198. Curran Associates, Inc., 2020. URL: `https://proceedings.neurips.cc/paper_files/paper/2020/file/f231f2107df69eab0a3862d50018a9b2-Paper.pdf`.

27 Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter J. Stuckey. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research*, 144:105809, 2022. `doi:10.1016/j.cor.2022.105809`.

28 Zongtao Liu, Wei Dong, Chaoliang Wang, Haoqingzi Shen, Gang Sun, Qun jiang, Quanjin Tao, and Yang Yang. Boosting graph search with attention network for solving the general orienteering problem. *AI Open*, 5:46–54, 2024. URL: `https://www.sciencedirect.com/science/article/pii/S266665102400007X`, `doi:10.1016/j.aiopen.2024.01.006`.

29 C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, October 1960. `doi:10.1145/321043.321046`.

30 Yimeng Min, Yiwei Bai, and Carla P. Gomes. Unsupervised learning for solving the travelling salesman problem. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

31 Yun Peng, Byron Choi, and Jianliang Xu. Graph Learning for Combinatorial Optimization: A Survey of State-of-the-Art. *Data Science and Engineering*, 6(2):119–141, June 2021. `doi:10.1007/s41019-021-00155-3`.

32 Yunzhuang Shen, Yuan Sun, Xiaodong Li, Andrew Eberhard, and Andreas Ernst. Adaptive solution prediction for combinatorial optimization. *European Journal of Operational Research*, 309(3):1392–1408, 2023. doi:10.1016/j.ejor.2023.01.054.

33 Petr Stodola and Radomír Ščurek. Using machine learning in combinatorial optimization: Extraction of graph features for travelling salesman problem. *Knowledge-Based Systems*, 314:113216, 2025. doi:10.1016/j.knosys.2025.113216.

34 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018. URL: https://arxiv.org/abs/1710.10903, arXiv:1710.10903.

35 Haoyu Wang, Nan Wu, Hang Yang, Cong Hao, and Pan Li. Unsupervised learning for combinatorial optimization with principled objective relaxation. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

36 Frederik Wenkel, Semih Cantürk, Stefan Horoi, Michael Perlmutter, and Guy Wolf. Towards a general recipe for combinatorial optimization with multi-filter gnns. In Guy Wolf and Smita Krishnaswamy, editors, *Proceedings of the Third Learning on Graphs Conference*, volume 269 of *Proceedings of Machine Learning Research*, pages 3:1–3:20. PMLR, 26–29 Nov 2025. URL: https://proceedings.mlr.press/v269/wenkel25a.html.

37 Gerhard J. Woeginger. Exact algorithms for NP-hard problems: A survey. In Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, editors, *Combinatorial Optimization — Eureka, You Shrink!: Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001 Revised Papers*, pages 185–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/3-540-36478-1_17.

38 Fulin Yan, François Clautiaux, Aurélien Froger, and Boris Albar. Generic machine-learning-augmented beam search for resource-constrained shortest path reformulations of combinatorial optimization problems. *Computers & Operations Research*, 187:107339, 2026. doi:10.1016/j.cor.2025.107339.

39 Ni Zhang, Jingfeng Yang, Zhiguang Cao, and Xu Chi. Adversarial generative flow network for solving vehicle routing problems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 71463–71477, 2025. URL: https://proceedings.iclr.cc/paper_files/paper/2025/file/b210c387381713a14a4f5a607aff3520-Paper-Conference.pdf.

40 Hongmin Zhu, Fuli Feng, Xiangnan He, Xiang Wang, Yan Li, Kai Zheng, and Yongdong Zhang. Bilinear graph neural network with neighbor interactions. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20, 2021.