Towards General Algorithm Discovery for Combinatorial Optimization: Learning Symbolic Branching Policy from Bipartite Graph

Yufei Kuang [1 2] **Jie Wang** [1 2✉] **Yuyan Zhou** [1 2] **Xijun Li** [1 2 3] **Fangzhou Zhu** [3] **Jianye Hao** [3 4] **Feng Wu** [1 2]

Abstract

Machine learning (ML) approaches have been successfully applied to accelerating exact combinatorial optimization (CO) solvers. However, many of them fail to explain what patterns they have learned that accelerate the CO algorithms due to the black-box nature of ML models like neural networks, and thus they prevent researchers from further understanding the tasks they are interested in. To tackle this problem, we propose the *first* graph-based algorithm discovery framework—namely, graph symbolic discovery for exact combinatorial optimization solver (GS4CO)—that learns interpretable branching policies directly from the *general* bipartite graph representation of CO problems. Specifically, we mainly focus on the variable selection part of the branching policy. We design a unified representation for symbolic variable selection policies with graph inputs, and then we employ a Transformer with multiple tree-structural encodings to generate symbolic trees end-to-end, which effectively reduces the cumulative error from iteratively distilling graph neural networks. Experiments show that GS4CO learned interpretable and lightweight policies outperform all the baselines on CPU machines, including both the human-designed and the learning-based. GS4CO shows an encouraging step towards general algorithm discovery on modern CO solvers. Codes are available at https://github.com/MIRALab-USTC/L2O-GS4CO.

[1]CAS Key Laboratory of Technology in GIPAS, University of Science and Technology of China [2]MoE Key Laboratory of Brain-inspired Intelligent Perception and Cognition, University of Science and Technology of China [3]Noah's Ark Lab, Huawei Technologies [4]College of Intelligence and Computing, Tianjin University. Correspondence to: Jie Wang <jiewangx@ustc.edu.cn>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

1. Introduction

Combinatorial optimization (CO) model—which is widely used to formulate real-world tasks like transportation, management, and chip design (Liu et al., 2008; Chen, 2010; Ma et al., 2019; Paschos, 2014)—is one of the most fundamental models in the field of mathematical optimization (MO). In practice, complex CO problems are usually solved with exact CO solvers like SCIP (Achterberg, 2007) and Gurobi (Gurobi Optimization, LLC, 2023) based on the branch-and-bound (B&B) framework. However, solving CO problems is usually highly time-consuming due to its NP-hard nature. To improve the efficiency of exact CO solvers, recently, researchers incorporate machine learning (ML) techniques to different components of CO solvers, e.g., branching (Khalil et al., 2016; Gasse et al., 2019), cut selection (Huang et al., 2022; Wang et al., 2023b; Tang et al., 2020), and primal heuristics (Chmiela et al., 2021; Paulus & Krause, 2023; Nair et al., 2021). These approaches consistently achieve high performance in terms of the solving efficiency on problems with chosen implicit distributions (Bengio et al., 2021; Zhang et al., 2023; Chen et al., 2022), as problems collected from similar tasks usually share similar structures.

However, due to the black-box nature of many ML models like neural networks (NNs), learning-based approaches usually fail to explain what patterns they have learned that accelerate the solving process. Based on the mixed-integer linear programming (MILP) formulation, CO problems can be modeled consistently via the general variable-constraint bipartite graph representation (Gasse et al., 2019). Thus, previous research widely incorporate graph neural networks (GNNs) to different components of exact CO solvers to learn and leverage the specific problem structures from the bipartite graphs. However, GNN-based approaches, though effective in practice, fail to help researchers further understand and improve the hard-coded CO algorithms due to their limited interpretability (Kuang et al., 2024).

In the age of data-driven scientific discovery (Wang et al., 2023a), we hope ML approaches on exact CO solvers can not only accelerate the solvers' efficiency but also provide interpretable policies to help researchers further understand what patterns they have learned. In light of this, a natural idea is to conduct automated algorithm discovery on CO

solvers. Intuitively, this is motivated from previous research, which has shown promising results in algorithms like matrix multiplication (Fawzi et al., 2022), sorting (Mankowitz et al., 2023), and NN optimizers (Chen et al., 2023).

However, compared with the research above, two distinct challenges of algorithm discovery on exact CO solvers make this topic non-trivial. **First**, how to *represent* interpretable CO policies that take graphs as inputs? Symbolic policies, which are widely studied in the field of scientific discovery due to their unreasonable effectiveness in natural science (Wigner, 1990; Petersen, 2019; Landajuela et al., 2021), is a natural solution for this challenge. However, the very recent research (Kuang et al., 2024) only consider symbolic policies that take human-designed fixed-length feature vectors as inputs. Note that compared with the general bipartite graph representation, human-designed features usually require extensive domain knowledge to extract different structural information of input CO problems for different downstream tasks, which severely limits its wide application to general CO algorithms. Thus, a general algorithm discovery framework on CO solvers need to consider the representation of symbolic policies that can directly extract structural features from graphs. **Second**, how to *generate* such interpretable symbolic policies? Previous research learns scientific laws from graph inputs by distilling symbolic functions from trained GNNs layer by layer (Cranmer et al., 2020; Shi et al., 2022). However, we observe that in CO algorithms like branching, such learning paradigm only leads to suboptimal performance due to the cumulative error from distilling GNN layers iteratively (see Section 4.1).

To tackle these challenges, we propose the *first* graph-based algorithm discovery framework—namely, graph symbolic discovery for exact combinatorial optimization solver (GS4CO)—that learns interpretable variable selection part of branching policies directly from the *general* bipartite graph representation. First, we propose a unified symbolic tree to consistently represent different functions and aggregation schemes (Hamilton, 2020b) together in the graph-based symbolic policy. Then, we employ a Transformer model (Lin et al., 2022) with multiple structural encodings for tree data to generate complex symbolic trees end-to-end, which effectively reduces the cumulative error from distilling GNN layers. Finally, we use the full strong branching (FSB) data as expert demonstrations (Gasse et al., 2019) and take the imitation learning accuracy as the fitness measure (Poli et al., 2008) for model optimization. Experiments show that GS4CO learned branching policies—which are interpretable, lightweight, and efficient for CPU-based inference—outperform all the baseline approaches on CPU machines, including both the human-designed and the learning-based ones.

We summarize our major contributions as follows. (1) General framework. As the bipartite graph representation of CO problems is widely used, GS4CO provides a general framework for algorithm discovery on modern CO solvers. (2) Interpretability. The learned symbolic policies are highly interpretable, which can help researchers further understand and optimize existing hard-coded algorithms like branching policies on solvers. (3) Performance. GS4CO learned branching policies outperform all the baseline approaches in terms of the solving efficiency on CPU machines. (4) Deployment. GS4CO is efficient for training and inference, and the learned policies are lightweight. All of these features highly benefit its wide deployment to modern CO solvers.

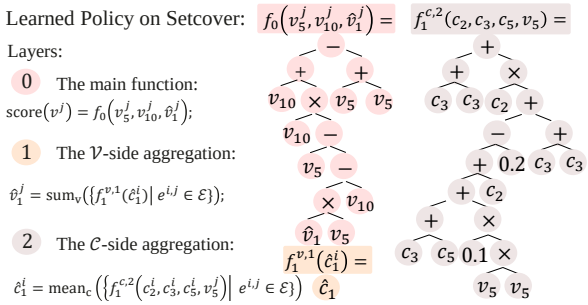

Figure 1. Visualize the symbolic branching policy with bipartite graph inputs on the Setcover benchmark. Specifically, the policy consists of three different parts: the $\mathcal{C}$-side aggregation function (Layer 2), the $\mathcal{V}$-side aggregation function (Layer 1), and the main function (Layer 0). Here $c_1^i - c_5^i$ are features of the i-th constraint and $v_1^j - v_{19}^j$ are features of the j-th variable. We select the variable v^j with highest score(v^j) for branching. See Section 4.2 for detailed descriptions about the unified representation for symbolic policies with graph inputs.

2. Related Work

Bengio et al. (2021) divides the existing research on machine learning for combinatorial optimization into two classes. One class assumes the existence of complex expert policies and attempts to replace the heavy computations with efficient approximations. For example, Gasse et al. (2019) leverages GNN to replace the strong branching policy in variable selection; He et al. (2014) employs imitation learning for optimal node selection policy; Nair et al. (2021) leverages GNN and MLP for primal heuristics and branching to achieve end-to-end solution prediction. The other class assumes insufficient expert knowledge and therefore uses machine learning to improve the heuristics that are not satisfactory yet. For example, Wang et al. (2023b) and Tang et al. (2020) employ reinforcement learning techniques to enhance the efficacy of cut selection, Chmiela et al. (2021) learn to schedule established primal heuristics within the MILP problem to diminish the primal integral. Although these NN-based approaches achieve high solving efficiency

(Chen et al., 2022; Zhang et al., 2023), they are integrated into the solver as a black box that we lack a deep understanding of the learned policies. Intuitively, these approaches can be regarded as ML-equipped CO approaches, but not the idea of data-driven CO algorithm discovery.

ML approaches can acquire implicit rules beyond human intuition and discover algorithms that outperform handcraft programs. Lion (Evo**L**ved **S**ign M**o**me**n**tum) uses evolutionary sign momentum optimization to optimize NN optimizers (Chen et al., 2023). AlphaDev leverages reinforcement learning and Monte Carlo tree search to optimize traditional sorting algorithms at the assembly level (Mankowitz et al., 2023). Though these approaches also focus on the optimization of existing manually designed algorithms, they do not leverage the prior knowledge of the input data, which is regard as the core for the acceleration of the NP-hard solving process in this research field (Bengio et al., 2021). In contrast, our approach focuses on learning interpretable policies that explicitly leverage the structural information of input CO problems. Recently, Kuang et al. (2024) applies deep symbolic regression to the scoring function in the branching module to learn interpretable policies. However, this work heavily depends on handcrafted features designed for specific downstream task, and thus it is not compatible to general CO tasks. Bipartite graph representation of CO is widely employed in many downstream tasks of CO solvers (Gasse et al., 2019; Nair et al., 2021). It provides a compact description of the problem structure and usually requires minimal expert design. Thus, algorithm discovery based on bipartite graph inputs could be more general in practice.

3. Preliminaries

3.1. Combinatorial Optimization and Mixed-Integer Linear Programming

In practice, a substantial number of CO problems can be represented as mixed-integer linear programming (MILP) problems in the following form:

$$\min_x c^T x$$
$$\text{s.t.} \quad Ax \leq b, \quad l \leq x \leq u, \quad x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$$

where $c \in \mathbb{R}^n$ is called the objective coefficient vector, $A \in \mathbb{R}^{m \times n}$ the constraint matrix, $b \in \mathbb{R}^m$ the constraint right hand side vector, $l, u \in \mathbb{R}^n$ respectively the lower and upper bounds and $p \leq n$ is the number of integer variables. Popular exact solvers like SCIP (Achterberg, 2007) and Gurobi (Gurobi Optimization, LLC, 2023) commonly use the branch-and-bound (B&B) algorithm to solve MILP. B&B algorithm recursively solves subproblems and organizes them as nodes of a search tree. The solver selects an integer variable x_i (branching variable) with fractional value x_i^* in the LP solution when exploring each node. Then, the solver adds constraints $x_i \leq \lfloor x_i^* \rfloor$ and $x_i \geq \lceil x_i^* \rceil$ to partition the feasible region and generate two new subproblems, and the solver selects a new subproblem to explore next.

3.2. Bipartite Graph Representation of MILP and Graph Neural Network

Gasse et al. (2019) introduces a natural variable-constraint bipartite graph to represent MILP problems. This graph involves two sets of nodes. One set of n variable nodes represents the decision variables, while the other set of m constraint nodes represents the linear constraints. Each edge $e^{i,j}$ in the graph with a weight a_{ij} represents that the j^{th} constraint consists of the i^{th} decision variable.

Based on the bipartite graph representation, graph neural networks (GNNs) are widely used in different downstream tasks (Gasse et al., 2019; Labassi et al.; Nair et al., 2021; Fan et al., 2023). Let $\mathcal{V}$ denote the set of n variable nodes, $\mathcal{C}$ denote the set of m constraint nodes, and $\mathcal{E}$ denote the set of edges. For integer $k > 0$, the $(2k-1)^{\text{th}}$ layer passes information from variables to constraints, and the $(2k)^{\text{th}}$ layer passes from constraints to variables. Then, the GNN message passing scheme could be written as:

$$h_{c^j}^{2k-1} = \hat{f}_c^k(\{h_{c^j}^{2k-2}, \sum_{(v^i,c^j)\in\mathcal{E}} \hat{g}_c^k(\{h_{v^i}^{2k-2}, h_{c^j}^{2k-2}\})\}), c^j \in \mathcal{C},$$
$$h_{v^i}^{2k} = \hat{f}_v^k(\{h_{v^i}^{2k-1}, \sum_{(v^i,c^j)\in\mathcal{E}} \hat{g}_v^k(\{h_{v^i}^{2k-1}, h_{c^j}^{2k-1}\})\}), v^i \in \mathcal{V},$$

where $\hat{f}_c^k, \hat{f}_v^k, \hat{g}_c^k, \hat{g}_v^k$ are multi-layer perceptrons, h_c^k and h_v^k denote the constraint's and variable's hidden embedding in the k^{th} layer, respectively.

3.3. Distilling Symbolic Functions from Trained GNNs

Cranmer et al. (2020) introduces a framework to distill interpretable symbolic functions from trained GNNs for scientific discovery. Specifically, it considers an edge model ϕ^e that maps connected nodes and edges information to message vectors, a node model ϕ^v that takes the node features and summed message vectors to compute updated node vectors, and a global model ϕ^u that aggregates all messages and updated node vectors to compute a global property. The approach first trains these NN-based models and then leverages symbolic regression (Poli et al., 2008) to *iteratively* approximate ϕ^e, ϕ^v and ϕ^u with symbolic functions.

3.4. Transformer

The Transformer model is composed of transformer layers. Each transformer layer consists of a self-attention module and a feed-forward network(Vaswani et al., 2017b). Let $H = [\hat{h}_1; \hat{h}_2; \cdots; \hat{h}_n] \in \mathbb{R}^{n \times d}$ denote the input of self-attention module, where $\hat{h}_i \in \mathbb{R}^{1 \times d}$ means the hidden representation of position i and d is the hidden dimension. The input H is multiplied by three matrices

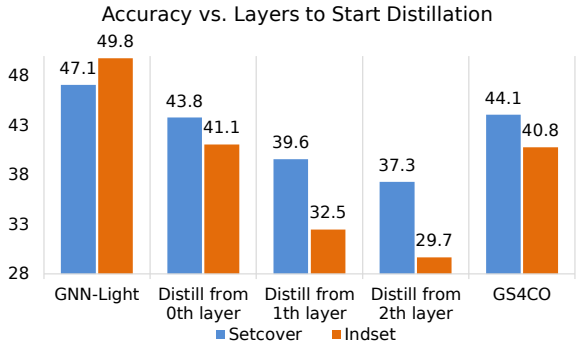

Figure 2. Learning symbolic branching policies that take the bipartite graph representation of CO problems as inputs via distilling trained two-layer GNNs (Gasse et al., 2019). We use a lightweight GNN (GNN-Light) with only eight-dimensional embedding of messages to reduce the complexity of distilling. Specifically, "Distill from 0^{th} layer" means we use the trained GNN to generate 2-hop messages and only learn a symbolic function at the final output; "Distill from 1^{th} layer" means we use the trained GNN for $\mathcal{C}$-side aggregation and then learn symbolic $\mathcal{V}$-side aggregation functions and a final output function iteratively; and "Distill from 2^{th} layer" means all the GNN layers are distilled to symbolic functions iteratively. All the distilling processes are conducted with the GPLearn package (Stephens, 2023). The results show that simply employing the distilling approach in scientific discovery (Cranmer et al., 2020) leads to suboptimal results in the branching task.

$W_Q \in \mathbb{R}^{d\times d_K}, W_K \in \mathbb{R}^{d\times d_K}$ and $W_V \in \mathbb{R}^{d\times d_V}$ to get Q(query), K(key) and V(value), respectively. Then, the self-attention can be calculated as follows:

$$Q = HW_Q, \quad K = HW_K, \quad V = HW_V,$$
$$M = \frac{QK^T}{\sqrt{d_K}}, \quad \text{Attn}(H) = \text{softmax}(M)V,$$

where M captures the similarity between queries and keys.

3.5. Symbolic Optimization

Symbolic optimization utilizes symbolic expressions to optimize targeted tasks such as symbolic regression. There are roughly two approaches for symbolic optimization: one is based on genetic programming (GP) (Bäck et al., 2018), and the other is based on deep symbolic optimization (DSO) (Petersen, 2019). Both of them represent mathematical expressions in the form of expression trees. In this tree, the leaf nodes represent variables and constants, and the internal nodes represent unary (e.g. $\log, \exp$) or binary (e.g. $+, -, \times, \div, \text{pow}$) mathematical operators. Then, GP approaches use selection, crossover, and mutation to search for symbolic expressions with high fitness. DSO leverages NN models to learn the distribution of high-quality symbolic expressions. Then, the model is trained by policy gradient methods with fitness as rewards.

4. Method

Due to the complexity of modern CO solvers[1], conducting algorithm discovery directly on the whole solver is usually impractical. However, extensive research shows that the optimization on specific components of exact CO solvers (e.g., branching, cut selection, and primal heuristics) can bring significant acceleration on the efficiency of the solver (Khalil et al., 2016; Huang et al., 2022; Chmiela et al., 2021). In this paper, we mainly focus on the branching component of the exact CO solver, as this small component plays a key role in the efficiency of the B&B framework (Achterberg, 2007) and there is extensive previous research on it (Khalil et al., 2016; Gasse et al., 2019; Gupta et al., 2020).

4.1. Problem Setup

Limitations of GNN-Based Approaches Previous research widely incorporates GNNs to tackle the bipartite graph inputs, and these approaches achieve encouraging performance in terms of the efficiency of solving CO problems (Zhang et al., 2023). However, as mentioned in Kuang et al. (2024), NN-based approaches have several general limitations when deploying to modern CO solvers: 1) Extensive training data required. Collecting training data for NN models is usually challenging due to reasons like proprietary issues (Geng et al., 2023). 2) High-end GPUs required. Servers for CO solvers are usually purely CPU-based (Gupta et al., 2020). 3) The black-box models. ML models like neural networks fail to help researchers understand and optimize the hard-coded algorithms due to the lack of interpretability (Kuang et al., 2024). These limitations severely hinder the wide application of ML approaches to modern CO solvers.

Challenges for Generating Symbolic Policies To address the above limitations, a natural solution is learning symbolic policies that takes graphs as inputs to replace the GNN models. However, in practice, we observe two distinct challenges when learning such symbolic policies:

1. Symbolic policy *representation*. Previous research employs symbolic trees to represent functions with fixed-length inputs, but in CO algorithms we need to design a representation for policies with graph inputs, i.e., representing different functions and aggregation schemes consistently in a graph-based symbolic policy.

2. Symbolic policy *generation*. Existing approaches in scientific discovery learn symbolic functions by distilling the trained GNNs layer by layer (Cranmer et al., 2020; Shi et al., 2022). However, we observe this leads

[1]For example, the open-source CO solver SCIP (Gleixner et al., 2018) contains roughly 900 thousand lines of C/C++ code in total, which is significantly larger than algorithms like matrix multiplication (Fawzi et al., 2022) and sorting (Mankowitz et al., 2023).

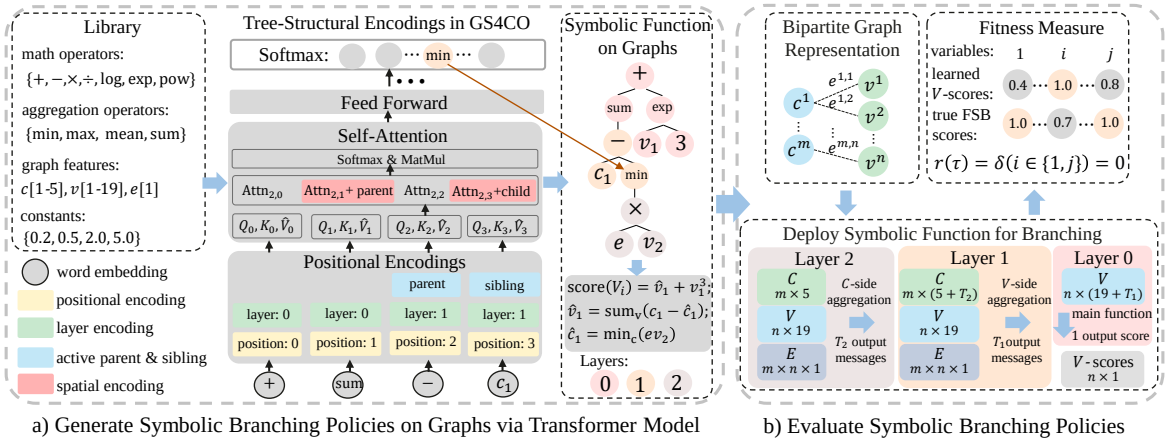

Figure 3. Illustration of the graph-based symbolic discovery framework on branching. We employ a unified symbolic representation for functions on graphs and a Transformer with tree-structural encodings to generate symbolic policies end-to-end. These policies can directly tackle bipartite graphs. Then, we use the imitation learning accuracy on expert data collected from the FSB policy as the fitness measure.

to suboptimal performance in this task due to the learning error accumulated by layers (see Figure 2).

To tackle these challenges, we propose the graph-based symbolic discovery framework for exact combinatorial optimization solver (GS4CO). We illustrate our approach in Figure 3 and the pseudo code in Algorithm 1 in Appendix.

4.2. Representing Symbolic Policy with Graph Inputs

In genetic programming (Poli et al., 2008), we represent functions as symbolic trees, where leaf nodes are features or constants and other nodes are mathematical operators. For the branching task, the symbolic functions take the variable-constraint bipartite graph representation of CO problems as inputs and output the scores for all branching variables.

Features, Operators, and Constants We use all the features from the variable-constraint bipartite graphs defined in (Gasse et al., 2019). Specifically, these features can be categorized into three groups, i.e., the variable node features (nineteen-dimensional), the constraint node features (five-dimensional), and the edge features (one-dimensional). We note that compared with the human-designed 91-dimensional features used by Gupta et al. (2020), all of these graph features are: 1) simple to design, as they are all commonly used to describe the input problems and the current status of the CO solver (Gasse et al., 2019; Labassi et al.; Fan et al., 2023); 2) cheap to obtain, as they are recorded by most CO solvers by default. We employ $\{+, -, \times, \div, \log, \exp, \text{pow}\}$ as mathematical operators and $\{0.2, 0.5, 2.0, 5.0\}$ as constants. Furthermore, we employ four unary operators for neighborhood aggregation on graphs, i.e., $\{\min, \max, \text{mean}, \text{sum}\}$. These aggregation operators play the same role as that in GNNs (Hamilton, 2020b), but now they are used in the following symbolic aggregation functions to generate structural features.

Symbolic Aggregation Function In GNNs, we extract structural information from the input graphs via neighborhood aggregation schemes (Hamilton, 2020b), which can be regarded as generalized convolution operations on graphs. Thus, it is a natural idea to generalize the aggregation schemes to symbolic aggregation functions, which are employed in the symbolic main function to extract structural information as new node features. Specifically, a symbolic aggregation function takes all features listed above (i.e., the constraint, the variable, and the edge features) as inputs to calculate a new feature (i.e., a one-dimensional message). This new feature will then be used in its parent function, which is either the main function or another aggregation function. Similar to that in GNNs, this process can also be conducted iteratively to obtain multi-hop information from input graphs. Thus, if we represent the branching policy as a symbolic tree, then the symbolic aggregation functions can be represented as subtrees on it, and the root node of each subtree is selected from the aggregation operators mentioned above to describe its aggregation scheme. We provide a toy example to illustrate it in Figure 3 Part a).

Symbolic Function with Bipartite Graph Inputs As illustrated in Figure 1 and Figure 3 Part b), we can divide the symbolic branching policy with general variable-constraint bipartite graph inputs into three different parts, i.e., the $\mathcal{C}$-side aggregation functions (at Layers $2k$, $k \geq 1$), the $\mathcal{V}$-side aggregation functions (at Layers $2k-1$), and the main function (at Layer 0). Formally, at Layer $2k$, the t^{th} $\mathcal{C}$-side aggregation function $f_t^{c,2k}$ passes the message from variables to constraints to generate a new feature $\hat{c}_{2k,t}^i$ for

all constraint node $i = 1, \cdots, m$, i.e.,

$$\hat{c}^i_{2k,t} = \text{AGGR}(\{f^{c,2k}_t(c^i_1, \cdots, c^i_5, e^{i,j}, v^j_1, \cdots, v^j_{19}, \hat{v}^j_{2k+1,1}, \cdots, \hat{v}^j_{2k+1,T_{2k+1}}) \mid e^{i,j} \in \mathcal{E}\}), \tag{1}$$

where AGGR is the aggregation operator selected from the for operators mentioned above, $\hat{v}^j_{2k+1,\cdot}$ is the new structural feature of variable j generated at Layer $2k+1$, and $T_{2k+1} = 0$ if no new features is generated. Similarly, for t^{th} $\mathcal{V}$-side aggregation function $f^{v,2k-1}_t$ at Layer $2k-1$,

$$\hat{v}^j_{2k-1,t} = \text{AGGR}(\{f^{v,2k-1}_t(c^i_1, \cdots, c^i_5, e^{i,j}, v^j_1, \cdots, v^j_{19}, \hat{c}^i_{2k,1}, \cdots, \hat{c}^i_{2k,T_{2k}}) \mid e^{i,j} \in \mathcal{E}\}). \tag{2}$$

Finally, the main function f_0 at Layer 0 takes all the variable features (both the original and the generated ones) as inputs, and it outputs the scores for all branching variables, i.e.,

$$\text{score}(v^j) = f_0(v^j_1, \cdots, v^j_{19}, \hat{v}^j_{1,1}, \cdots, \hat{v}^j_{1,T_1}), \tag{3}$$

which can be regarded as a standard symbolic regression formulation with a vector feature as input.

4.3. Generating Symbolic Branching Policies via Sequential Model

Given a fixed traversal, a symbolic function (represented by the symbolic tree) is one-to-one to a symbolic sequence as $\tau = \tau_1 \tau_2 \cdots \tau_n$, and thus sequential models are widely employed (Petersen, 2019; Landajuela et al., 2021) to generate such sequence. Intuitively, generating branching functions with graph inputs could be more complex, as we need to generate multiple symbolic aggregation sub-functions to extract structural features from graphs. Note that the search space of all the feasible τ grows *exponentially* with the complexity of the symbolic policy (Poli et al., 2008). To tackle this challenge, we employ a Transformer model (Lin et al., 2022) with multiple structural encodings to encode complex tree-structural information at each step and generate such symbolic trees end-to-end, as illustrated in Figure 3 Part a).

Sequential Formulation We formulate the symbolic tree generation task as a sequence generation task with a given depth-first traversal. Then, we output a categorical distribution over all possible tokens (as listed in Section 4.2) to sample the current token at each step. That is,

$$p_{\boldsymbol{\theta}}(\tau) = \Pi_{i=1}^{|\tau|} p_{\boldsymbol{\theta}}(\tau_i \mid \tau_{1:(i-1)}), \tag{4}$$

where $p_{\boldsymbol{\theta}}(\tau)$ is the probability of generating the expression sequence τ, $p_{\boldsymbol{\theta}}(\tau_i \mid \tau_{1:(i-1)})$ is the probability of selecting the token at i-th step , and $\boldsymbol{\theta}$ is the learnable parameters of the sequential model. In GS4CO, we employ a decoder-only Transformer as the sequential model, i.e., $p_{\boldsymbol{\theta}}(\tau_i \mid \tau_{1:(i-1)})$ is approximated via a Transformer model.

Tree-Structural Information Encodings In practice, we observe that a vanilla Transformer model achieves suboptimal performance as it ignores the tree-structural information of the symbolic trees (see Section 5.3 for comparisons). To further improve the performance of the Transformer model, we employ four different encodings to extract the tree's structural information as follows, including three types of encodings that are *specially designed* for the tree structure.

1) Positional encoding. We use the vanilla positional encoding as that in the original Transformer model (Vaswani et al., 2017a). The positional encoding of token τ_i is:

$$PE_{(\tau_i, 2j)} = \sin(i/10000^{2j/d}) \tag{5a}$$

$$PE_{(\tau_i, 2j+1)} = \cos(i/10000^{2j/d}) \tag{5b}$$

where d is the total dimensions of the word embeddings and j is the j^{th} dimension (Vaswani et al., 2017a).

2) Layer encoding. We employ learnable layer encodings to encode the layer information of the nodes on the tree:

$$LE(\tau_i) = \alpha_{l_i}, \quad l_i = 0, 1, \cdots, L_{\text{max}} \tag{6}$$

where l_i is the layer of token τ_i, L_{max} is the maximal layer, and α_{l_i} is a d-dimensional learnable vector indexed by l_i.

3) Active parent & sibling encoding. Given the already selected tokens $\tau_0, \cdots, \tau_{i-1}$, the current candidate token τ_i, and the parent and the sibling τ_{p_i}, τ_{s_i} of τ_i (if τ_{p_i}, τ_{s_i} exist), we encode the information of them via:

$$APE(\tau_{p_i}) = \beta_p; \; APE(\tau_{s_i}) = \beta_s, \tag{7}$$

where β_p, β_s are d-dimensional learnable vectors.

4) Spatial encoding. We use spatial encoding to capture the tree-structural information of the symbolic tree, which is mainly motivated from the Graphormer model proposed by Ying et al. (2021). Let τ_i, τ_j denote two nodes in the symbolic tree. Then, the spatial encoding SE is:

$$SE(\tau_i, \tau_j) = \begin{cases} \gamma_p, & \text{if } \tau_j \text{ is the parent of } \tau_i; \\ \gamma_c, & \text{if } \tau_j \text{ is the child of } \tau_i; \\ \gamma_s, & \text{if } \tau_i, \tau_j \text{ are siblings;} \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

where $\boldsymbol{\gamma} = [\gamma_p, \gamma_c, \gamma_s]$ is a 3-dimensional learnable vector. We add the spatial encoding before calculating attentions to leverage the structural information of the tree:

$$M^{\text{new}}_{i,j} = M_{i,j} + SE(\tau_i, \tau_j), \tag{9}$$

where $M^{\text{new}}_{i,j}, M_{i,j}$ are used for attentions (see Section 3.4).

Constraints for Symbolic Policies on Graphs We employ the following six different types of constraints on the search

space to avoid trivial functions and accelerate the training process, including three constraints that are *specially designed* for graph inputs. 1) Aggregation function input constraints. If a symbolic aggregation function is $\mathcal{C}$-side, then it needs to contain at least one variable feature as input, and vice versa for $\mathcal{V}$-side aggregation functions. This constraint helps us avoid generating trivial aggregation functions. 2) Main function input constraints. The main function only uses the structural features extracted by aggregation functions together with the original variable features. 3) Aggregation function layer constraints. The layers (similar to the layers in GNNs) of the aggregation functions are constrained to no more than two, as two-layer GNNs are widely used in previous research (Gasse et al., 2019; Labassi et al.). 4) Length constraints. The complexity of the expression is restricted to specific ranges to avoid too simple or too complex expressions. 5) Inverse operator constraints. The child of a unary operator can not be the inverse of its parent. 6) Nontrivial constraints. The functions need to contain at least one graph feature to avoid trivial expressions.

4.4. Fitness Measure and Model Training

The Fitness Measure Due to the NP-hard nature of solving CO problems, it is extremely intractable to train with end-to-end solving time as fitness measure. Thus, we use the full strong branching (FSB) policy (Achterberg, 2007) as an expert and use imitation accuracy as the fitness measure like that in previous research (Gasse et al., 2019; Gupta et al., 2020). The collected data is $\mathcal{D} = \{(\mathbf{s}_t, u_{1:k_t})_{t=1}^{T}\}$, where $\mathbf{s}_t = (\mathcal{G}, \mathcal{C}, \mathcal{V}, \mathcal{E})$ denote the features of the bipartite graph and u_i the FSB scores for all branching candidates $i = 1, 2, \cdots, k_t$ at each node $t = 1, 2, \cdots, T$. Then, the fitness measure is

$$r(\tau) = \mathop{\mathbb{E}}_{(\mathbf{s}_t, u_{1:k}) \sim \mathcal{D}} [\mathbf{I}_{\{i | u_i \geq u_j, 1 \leq j \leq k\}} (\mathop{\mathrm{argmax}}_{1:k} \tau(\mathbf{s}))], \quad (10)$$

where $(\mathbf{s}, u_{1:k})$ is sampled from $\mathcal{D}$ as an expert demonstration, $\tau(\mathbf{s})$ returns a vector of length k that represents the branching scores calculated by the symbolic policy τ on all the branching candidates, and $\mathbf{I}_A(x)$ is the indicator function of set A that returns 1 if and only if $x \in A$, and otherwise returns 0. Then, the fitness measure is the average number of branching variables that the symbolic policy τ selects with the maximal FSB score.

The Model Training With the generator $p_{\boldsymbol{\theta}}$ and the fitness measure $r(\tau)$, we can optimize $\boldsymbol{\theta}$ to maximize the probability of τ with high fitness. We use reinforcement learning (RL) (Sutton & Barto, 2018) for optimization as the objective is non-differentiable for $\boldsymbol{\theta}$. The training task is formulated as a continuous bandit problem, the $p_{\boldsymbol{\theta}}(\tau)$ is the parameterized policy, the expression sequence τ is the action we select, and $r(\tau)$ is the reward function. Then, we employ the risk-seeking objective (Tamar et al., 2014; Petersen, 2019) to optimize the best-case performance, i.e.

$$J(\boldsymbol{\theta}; \epsilon) = \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} [r(\tau) | r(\tau) \geq r_\epsilon(\tau)], \quad (11)$$

where $r_\epsilon(\tau)$ is the $(1-\epsilon)$ quantile of current batch rewards. We employ proximal policy optimization (PPO) (Schulman et al., 2017) to maximize the objective above, and we employ both the hierarchical entropy regularizer and the soft-length regularizer as that in (Landajuela et al., 2021).

5. Experiments

We conduct extensive experiments on GS4CO to 1) illustrate it significantly outperforms existing CPU-based approaches in terms of the end-to-end performance; 2) analyse the learned graph-based policies for better interpretability and understanding; 3) conduct ablation studies on all proposed components to show their effectiveness.

5.1. Experimental Settings

Baselines There are five baselines to compare in this section to illustrate the superior performance of GS4CO. Specifically, the reliability pseudocost branching (RPB) and the full strong branching (FSB)[3] are two human-designed branching policies integrated in modern solvers like SCIP by default (Achterberg, 2007); the Trees model (Alvarez et al., 2017) is a baseline that serves as an interpretable branching policy; the GNN (Hamilton, 2020a) model proposed in Gasse et al. (2019) is a baseline that serves as an branching policy with graph inputs; the Hybrid model proposed in Gupta et al. (2020) is a *strong* baseline for purely CPU-based devices.

Benchmarks We employ four standard benchmarks used in previous research (Gasse et al., 2019) for performance evaluation. Specifically, the benchmarks include the set covering (Setcover) (Balas & Ho, 1980), the combinatorial auction (Cauctions) (Leyton-Brown et al., 2000), the capacitated facility location (Facilities) (Cornuéjols et al., 1991), and the maximum independent set (Indset) (Bergman et al., 2015). For each benchmark, we generate small instances (Easy) for training and testing and generate larger instances (Medium and Hard) to evaluate the generalization ability.

Training To train GS4CO, we generate 100 instances for training and 20 instances for validation, and we obtain $10,000$ and $2,000$ samples via the FSB policy; for training the GNN, the Hybrid, and the Trees models, we generate $10,000$ and $2,000$ instances, $100,000$ and $20,000$ samples, respectively. We use the codes provided by Gupta et al. (2020) to generate the MILP instances and the training sam-

[2]Since most of the instances have not been solved, the number of nodes expanded is small but can not reflect the real solving ability

[3]FSB and RPB do not leverage instance information from the training dataset

Table 1. Compare GS4CO with five different baselines to illustrate the superior performance of learned symbolic policies in terms of the end-to-end solving efficiency. All results are evaluated with the time limit of $3000s$ over 50 test instances, and the number after wins records the success time that the SCIP solver solve the instances to optimal within the given time limit. We report the time and the nodes with the 1-shifted geometric mean to reduce the effect of extreme values. All models are trained on the Easy instances only and then evaluated on the Easy, the Medium, and the Hard datasets. Results show that GS4CO outperforms all the baselines on CPU machines.

Indset:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	243.58	0/50	67.8	2432.30	0/28	82.7	3000.00	0/0	25.4
RPB	75.14	2/50	1690.6	357.56	2/50	8503.8	2640.00	5/28	30976.6
Trees	142.55	0/50	1515.7	1206.56	0/41	13582.2	2963.21	0/4	**15917.8** [2]
GNN	54.60	11/50	**863.9**	432.52	3/50	9712.4	2852.30	0/10	45386.0
Hybrid	61.83	4/50	2150.7	309.91	7/50	9123.1	2223.04	8/33	33852.6
GS4CO	**53.18**	**33/50**	912.5	**217.67**	**38/50**	**4812.5**	**2157.23**	**24/37**	34061.5
GNN-GPU	47.25	-/50	863.9	262.52	-/50	9712.4	2456.80	-/34	36532.4

Cauctions:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	9.31	0/50	12.7	220.58	0/50	113	2372.31	0/24	437.2
RPB	5.11	3/50	**33.2**	37.25	3/50	1308.6	331.52	15/50	13372.6
Trees	5.03	0/50	146.5	48.38	0/50	1786.0	889.28	0/42	23788.4
GNN	3.75	0/50	115.3	33.34	8/50	**1050.1**	403.59	2/50	**11489.9**
Hybrid	3.35	17/50	120.8	33.12	11/50	1270.9	351.75	12/50	15349.0
GS4CO	**3.16**	**30/50**	130.1	**30.58**	**28/50**	1314.0	**326.19**	**21/50**	14044.7
GNN-GPU	3.00	-/50	115.3	23.10	-/50	1050.1	289.62	-/50	11489.9

Setcover:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	47.60	0/50	29.3	843.61	0/50	233.8	2944.07	0/2	389.5
RPB	18.12	8/50	**174.4**	110.19	12/50	3288.8	2237.32	6/25	62706.2
Trees	19.76	0/50	384.1	224.40	0/50	5132.3	2709.57	0/15	29939.8
GNN	19.26	0/50	223.8	201.80	0/50	**2338.1**	2694.93	0/21	**20017.8**
Hybrid	13.88	18/50	279.2	105.13	17/50	2761.8	2160.35	8/29	43636.2
GS4CO	**13.29**	**24/50**	310.7	**103.74**	**21/50**	3591.3	**2100.53**	**15/29**	59915.1
GNN-GPU	12.41	-/50	223.8	79.60	-/50	2338.1	2011.74	-/50	53405.2

Facilities:	Easy			Medium			Hard		
Model	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes	Time(s)	Wins	Nodes
FSB	93.14	0/50	96.5	569.90	0/49	178.5	1452.65	0/40	86.5
RPB	78.75	10/50	**195.5**	314.90	13/50	**343.5**	1232.94	10/45	**256.0**
Trees	75.28	1/50	425.7	358.94	0/50	708.7	1306.05	0/43	542.1
GNN	79.00	0/50	328.6	363.26	0/50	563.0	1305.70	2/45	449.1
Hybrid	**71.11**	**23/50**	378.7	**284.24**	**21/50**	557.5	1219.91	11/45	412.0
GS4CO	76.27	16/50	353.3	295.82	16/50	528.1	**1018.70**	**22/45**	421.5
GNN-GPU	63.45	-/50	328.6	253.95	-/50	563.0	948.09	-/48	530.7

ples. Then, we record all the symbolic policies with top ten imitation learning accuracy on each benchmark, and we evaluate all of them to select the one with best end-to-end performance on the validation instances for testing. We terminate the training process (i.e., early stop) if there is no performance improvement for 2000 iterations. For the other baselines, we use the codes provided by Gupta et al. (2020).

Evaluation All evaluations are conducted over 50 test instances. We report the 1-shifted geometric mean (Achterberg, 2007) of all the results, which is widely used in previous work (Gasse et al., 2019; Gupta et al., 2020) as the evaluation metric. We report the end-to-end running time, the B&B tree nodes, and the wins for comparison. See Appendix B for more details about the implementations.

5.2. Comparative Evaluation

End-to-End Performance We compare GS4CO to all the baselines in Table 1. Results show that GS4CO outperforms all the other baselines on CPU machines, including both human-designed and learning-based ones[4]. It also generalizes well to larger datasets. The results show that GS4CO extracted structural features from bipartite graphs are effective for the downstream task. We further compare the imitation learning accuracy of all the approaches in Table 2. Results show that GS4CO learned symbolic policies significantly outperform the interpretable Tree model.

Strengths for Deployment We show the strengths of

[4]RPB achieves minimum number of nodes since in SCIP it employs multiple strategies for probing and cutting off child nodes.

Table 2. The imitation learning accuracy of different approaches on test sets. Results show that: 1) GS4CO learned symbolic policies significantly outperform the interpretable Tree model; 2) GS4CO significantly improves the learning accuracy for symbolic approaches via avoiding distilling GNN layers; 3) the Transformer model, the tree-structural encodings, and the constraints for graph inputs all effectively improve the performance of GS4CO.

Type	Model	Setcover(%)	Facilities(%)	Indset(%)	Cauctions(%)
White-box:	Tree	40.43	63.39	24.61	38.72
Black-box:	Hybrid	48.75	66.83	51.40	44.30
	GNN	54.18	69.33	56.42	47.50
Symbolic:	Distilling-GNN	37.28	66.15	29.70	36.70
	LSTM	40.05	66.30	36.45	31.61
	No-Encodings	35.73	66.78	36.78	38.40
	No-Constraints	41.85	66.78	39.50	38.76
	GS4CO	44.13	66.78	40.78	41.13

Table 3. Compare the model size and the inference time on CPU machines for different ML models on Setcover. Results show that GS4CO learned graph-based policies are highly lightweight and efficient compared with GNN models. These strengths significantly benefit the wide deployment of GS4CO to modern CO solvers.

ML Model	Tree	GNN	Hybrid	GS4CO
Model Size (KB)	$3.04e6$	266	1331	**0.49**
Inference Time (ms)	27.45	33.88	7.92	**1.84**

GS4CO for deployment. 1) Lightness. We compare the size of learned models in Table 3. Results show that GS4CO learned policies are roughly 500 times lighter than GNNs. 2) Inference efficiency. We compare the inference efficiency on CPU in Table 3. Results show that GS4CO are roughly 20 times faster than GNNs for inference on CPU machines. 3) Sample efficiency. As reported in Section 5.1, we use only 1% CO instances and 10% FSB samples compared with GNNs. All of these features significantly benefit the wide deployment of GS4CO to modern CO solvers.

Interpretability We visualize the symbolic tree representation of the branching policy on Setcover in Figure 1 and report all symbolic policies in Table B5 in Appendix. Since the learned policies are highly interpretable, they can help researchers further understand what patterns they have learned on the branching component. Specifically, we observe that: 1) all the learned symbolic policies use two-hop structural information from the bipartite graph, which is consistent to the implementation in previous GNN-based approach (Gasse et al., 2019); 2) all the policies ignore the edge feature and only a small subset of node features are used, which indicates that the effective information in the bipartite graph representation is relatively sparse. We believe these results can further help researchers to optimize both the hard-coded and the learning-based algorithms on this component.

5.3. Ablation Study

We conduct experiments to show the effectiveness of the unified representation, the Transformer model, the structural encodings, and the constraints for graph inputs proposed in Section 4. The results are shown in Table 2. Specifically, we compare the model employed in GS4CO to: 1) distilling symbolic functions iteratively from trained lightweight GNNs (Distilling-GNN) as that widely employed in scientific discovery approaches (Cranmer et al., 2020; Shi et al., 2022); 2) using the long short-term memory (LSTM) network as the sequential model as that in DSO (Petersen, 2019; Landajuela et al., 2021); 3) using the vanilla Transformer model without the three specially designed tree-structural encodings (No-Encodings); 4) only using the constraint proposed by Petersen (2019) without the three constraints designed for graph inputs (No-Constraints). Results show that all the components proposed in Section 4 effectively improve the learning accuracy of GS4CO.

6. Conclusion

In this paper, we propose a graph-based symbolic discovery framework for exact combinatorial optimization solver (GS4CO) to learn interpretable branching policies directly from the *general* bipartite graph representation of CO problems. Experiments show the superior end-to-end performance of GS4CO learned symbolic policies on purely CPU-based machines and their strengths for the wide deployment to modern CO solvers. Applying GS4CO to more components in modern CO solvers like primal heuristics (Nair et al., 2021), node selection (Labassi et al.), and initial basis (Fan et al., 2023) are exciting avenues for further work.

Acknowledgements

The authors would like to thank all the anonymous reviewers for their insightful comments and valuable suggestions. This work was supported by National Key R&D Program of China under contract 2022ZD0119801 and National Nature Science Foundations of China grants U23A20388, U19B2026, U19B2044, and 62021001.

Impact Statement

Our work shows an encouraging step towards general algorithm discovery on modern CO solvers. Thus, it has a potential positive impact on the development of modern CO solvers for both the open-source and the commercial ones. Our work also provides a new paradigm for the research on the field of machine learning for combinatorial optimization. We hope our work can motivate more new research on this field and improve the performance of modern CO solvers. One limitation of GS4CO is that, though much less domain knowledge is required to design states and features compared to previous research, it still requires manually designed node features for the general bipartite graph representation of CO problems for different downstream tasks.

References

Achterberg, T. *Constraint integer programming*. PhD thesis, Berlin Institute of Technology, 2007. URL http://opus.kobv.de/tuberlin/volltexte/2007/1611/.

Alvarez, A. M., Louveaux, Q., and Wehenkel, L. A machine learning-based approximation of strong branching. *INFORMS J. Comput.*, 29(1):185–195, 2017. doi: 10.1287/ijoc.2016.0723. URL https://doi.org/10.1287/ijoc.2016.0723.

Bäck, T., Fogel, D. B., and Michalewicz, Z. *Evolutionary computation 1: Basic algorithms and operators*. CRC press, 2018.

Balas, E. and Ho, A. *Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study*. Springer, 1980.

Bengio, Y., Lodi, A., and Prouvost, A. Machine learning for combinatorial optimization: A methodological tour d'horizon. *Eur. J. Oper. Res.*, 290(2):405–421, 2021. doi: 10.1016/j.ejor.2020.07.063.

Bergman, D., Ciré, A. A., van Hoeve, W. J., and Hooker, J. Decision diagrams for optimization. *Constraints*, 20:494 – 495, 2015.

Chen, T., Chen, X., Chen, W., Heaton, H., Liu, J., Wang, Z., and Yin, W. Learning to optimize: A primer and a benchmark. *Journal of Machine Learning Research*, 23(189):1–59, 2022. URL http://jmlr.org/papers/v23/21-0308.html.

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Liu, Y., Pham, H., Dong, X., Luong, T., Hsieh, C.-J., Lu, Y., and Le, Q. V. Symbolic discovery of optimization algorithms. *ArXiv*, abs/2302.06675, 2023.

Chen, Z.-L. Integrated production and outbound distribution scheduling: review and extensions. *Operations research*, 58(1):130–148, 2010.

Chmiela, A., Khalil, E. B., Gleixner, A. M., Lodi, A., and Pokutta, S. Learning to schedule heuristics in branch and bound. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 24235–24246, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/cb7c403aa312160380010ee3dd4bfc53-Abstract.html.

Cornuéjols, G., Sridharan, R., and Thizy, J.-M. A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, 50:280–297, 1991.

Cranmer, M., Sanchez Gonzalez, A., Battaglia, P., Xu, R., Cranmer, K., Spergel, D., and Ho, S. Discovering symbolic models from deep learning with inductive biases. *Advances in Neural Information Processing Systems*, 33: 17429–17442, 2020.

Fan, Z., Wang, X., Yakovenko, O., Sivas, A. A., Ren, O., Zhang, Y., and Zhou, Z. Smart initial basis selection for linear programs. In *International Conference on Machine Learning*, pp. 9650–9664. PMLR, 2023.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., Ruiz, F. J. R., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., and Kohli, P. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nat.*, 610(7930):47–53, 2022. doi: 10.1038/S41586-022-05172-4. URL https://doi.org/10.1038/s41586-022-05172-4.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. Exact combinatorial optimization with graph convolutional neural networks. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 15554–15566, 2019.

Geng, Z., Li, X., Wang, J., Li, X., Zhang, Y., and Wu, F. A deep instance generative framework for milp solvers under limited data availability. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Gleixner, A., Bastubbe, M., Eifler, L., Gally, T., Gamrath, G., Gottwald, R. L., Hendel, G., Hojny, C., Koch, T., Lübbecke, M. E., et al. The scip optimization suite 6.0. technical report. *Optimization Online*, 2018.

Gupta, P., Gasse, M., Khalil, E. B., Kumar, M. P., Lodi, A., and Bengio, Y. Hybrid models for learning to branch. 2020.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

Hamilton, W. L. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2020a. doi: 10.2200/S01045ED1V01Y202009AIM046. URL https://doi.org/10.2200/S01045ED1V01Y202009AIM046.

Hamilton, W. L. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2020b. ISBN 978-3-031-00460-5. doi: 10.2200/S01045ED1V01Y202009AIM046. URL https://doi.org/10.2200/S01045ED1V01Y202009AIM046.

He, H., Daume III, H., and Eisner, J. M. Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 27, 2014.

Huang, Z., Wang, K., Liu, F., Zhen, H.-L., Zhang, W., Yuan, M., Hao, J., Yu, Y., and Wang, J. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123:108353, 2022.

Khalil, E. B., Bodic, P. L., Song, L., Nemhauser, G. L., and Dilkina, B. Learning to branch in mixed integer programming. In Schuurmans, D. and Wellman, M. P. (eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pp. 724–731. AAAI Press, 2016. doi: 10.1609/aaai.v30i1.10080. URL https://doi.org/10.1609/aaai.v30i1.10080.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Kuang, Y., Wang, J., Liu, H., Zhu, F., Li, X., Zeng, J., HAO, J., Li, B., and Wu, F. Rethinking branching on exact combinatorial optimization solver: The first deep symbolic discovery framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=jKhNBulNMh.

Labassi, A. G., Chételat, D., and Lodi, A. Learning to compare nodes in branch and bound with graph neural networks. In *Advances in Neural Information Processing Systems*.

Landajuela, M., Petersen, B. K., Kim, S., Santiago, C. P., Glatt, R., Mundhenk, N., Pettit, J. F., and Faissol, D. Discovering symbolic policies with deep reinforcement learning. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 5979–5989. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/landajuela21a.html.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Leyton-Brown, K., Pearson, M., and Shoham, Y. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pp. 66–76, 2000.

Lin, T., Wang, Y., Liu, X., and Qiu, X. A survey of transformers. *AI Open*, 3:111–132, 2022. doi: 10.1016/J.AIOPEN.2022.10.001. URL https://doi.org/10.1016/j.aiopen.2022.10.001.

Liu, S., Pinto, J. M., and Papageorgiou, L. G. A tsp-based milp model for medium-term planning of single-stage continuous multiproduct plants. *Industrial & Engineering Chemistry Research*, 47(20):7733–7743, 2008.

Ma, K., Xiao, L., Zhang, J., and Li, T. Accelerating an fpga-based sat solver by software and hardware co-design. *Chinese Journal of Electronics*, 28(5):953–961, 2019.

Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J.-B., Ahern, A., Koppe, T., Millikin, K., Gaffney, S., Elster, S., Broshear, J., Gamble, C., Milan, K., Tung, R., Hwang, M., Cemgil, T., Barekatain, M., Li, Y., Mandhane, A., Hubert, T., Schrittwieser, J., Hassabis, D., Kohli, P., Riedmiller, M., Vinyals, O., and Silver, D. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023. doi: 10.1038/s41586-023-06004-9.

Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., Addanki, R., Hapuarachchi, T., Keck, T., Keeling, J., Kohli, P., Ktena, I., Li, Y., Vinyals, O., and Zwols, Y. Solving mixed integer programs using neural networks, 2021.

Paschos, V. T. *Applications of combinatorial optimization*, volume 3. John Wiley & Sons, 2014.

Paulus, M. B. and Krause, A. Learning to dive in branch and bound. *arXiv preprint arXiv:2301.09943*, 2023.

Petersen, B. K. Deep symbolic regression: Recovering mathematical expressions from data via policy gradients. *CoRR*, abs/1912.04871, 2019. URL http://arxiv.org/abs/1912.04871.

Poli, R., Langdon, W. B., and McPhee, N. F. *A Field Guide to Genetic Programming*. lulu.com, 2008. ISBN 978-1-4092-0073-4. URL http://www.gp-field-guide.org.uk/.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Shi, H., Ding, J., Cao, Y., Liu, L., Li, Y., et al. Learning symbolic models for graph-structured physical mechanism. In *The Eleventh International Conference on Learning Representations*, 2022.

Stephens, T. Gplearn: Genetic programming for symbolic regression in python, 2023. URL https://github.com/trevorstephens/gplearn.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Tamar, A., Glassner, Y., and Mannor, S. Policy gradients beyond expectations: Conditional value-at-risk. *CoRR*, abs/1404.3862, 2014. URL http://arxiv.org/abs/1404.3862.

Tang, Y., Agrawal, S., and Faenza, Y. Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning*, pp. 9367–9376. PMLR, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017a. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017b.

Wang, H., Fu, T., Du, Y., Gao, W., Huang, K., Liu, Z., Chandak, P., Liu, S., Katwyk, P. V., Deac, A., Anandkumar, A., Bergen, K., Gomes, C. P., Ho, S., Kohli, P., Lasenby, J., Leskovec, J., Liu, T., Manrai, A., Marks, D. S., Ramsundar, B., Song, L., Sun, J., Tang, J., Velickovic, P., Welling, M., Zhang, L., Coley, C. W., Bengio, Y., and Zitnik, M. Scientific discovery in the age of artificial intelligence. *Nat.*, 620(7972):47–60, 2023a. doi: 10.1038/S41586-023-06221-2. URL https://doi.org/10.1038/s41586-023-06221-2.

Wang, Z., Li, X., Wang, J., Kuang, Y., Yuan, M., Zeng, J., Zhang, Y., and Wu, F. Learning cut selection for mixed-integer linear programming via hierarchical sequence model. *arXiv preprint arXiv:2302.00244*, 2023b.

Wigner, E. P. The unreasonable effectiveness of mathematics in the natural sciences. In *Mathematics and science*, pp. 291–306. World Scientific, 1990.

Ying, C., Cai, T., Luo, S., Zheng, S., Ke, G., He, D., Shen, Y., and Liu, T.-Y. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021.

Zhang, J., Liu, C., Li, X., Zhen, H., Yuan, M., Li, Y., and Yan, J. A survey for solving mixed integer programming via machine learning. *Neurocomputing*, 519:205–217, 2023. doi: 10.1016/j.neucom.2022.11.024. URL https://doi.org/10.1016/j.neucom.2022.11.024.

Table B4. The features of constraints, edges and variables used for symbolic functions, following previous research (Gasse et al., 2019)

Features	Name	Description
c[1]	Cosine Similarity	Cosine similarity between constraint coefficients and objective coefficients vector
c[2]	Bias	Right hand side value normalized with constraint coefficients
c[3]	Tight Indicator	Indicate whether the constraint is tight
c[4]	Dual Value	Normalized dual solution value
c[5]	Age	LP age normalized by the total number of LP iterations
e[1]	Edge Coefficient	Coefficient of the variables in the constraint, normalized per constraint
v[1-4]	Type	Type of the variables (Binary, Integer, Continuous, Implied Integer)
v[5]	Variable Coefficient	Normalized variable coefficient in the objective
v[6-7]	Specified Bounds	Whether the variable has a lower bound (upper bound)
v[8-9]	Solution Bounds	Whether the solution value of the variable equals lower bound (upper bound)
v[10]	Fractionality	Variable solution value fractionality
v[11-14]	Basis	One of four basis classes (lower, basic, upper, zero)
v[15]	Reduced Cost	Normalized reduced cost
v[16]	Age	LP age normalized by the total number of LP iterations
v[17]	Solution Value	Variable value in the LP solution
v[18]	Incumbent Value	Value of the incumbent
v[19]	Average Primal Value	Average value of the variable of all feasible solutions

Table B5. The learned graph-based symbolic policies on all benchmarks. Here AGGR_c represents the $\mathcal{C}$-side aggregation and AGGR_v represents the $\mathcal{V}$-side aggregation, and AGGR is selected from the four aggregation operators mentioned in Section 4.2. We found that the symbolic policy learned on the benchmark Cautions generalizes well to the benchmark Indset. Thus, we simply use the same policy here.

Setcover	$((((((\text{sum}_v(\text{mean}_c((((c_2 * ((((((v_5 * (0.1 * v_5)) + (c_5 + c_3)) + c_2) - 0.2) + c_3) + c_3)) + c_3) + c_3))) * v_5) - v_{10}) - v_5) * v_{10}) - 2v_5) + v_{10})$
Facilities	$\text{mean}_v((((((\text{sum}_c(v_{14}) - \text{mean}_c(v_6)) * \text{sum}_c(v_{10})) + \text{mean}_c(v_6)) * \text{sum}_c(v_4)) * \text{mean}_c(v_{10})))$
Indset&Cauctions	$(((\text{sum}_v((\text{mean}_c(((v_{13} * ((((((((((((c_4 - v_8) * v_{11}) - v_5) * c_3) * v_{17}) + 2.0) * v_6) - v_4) - v_{13}) + v_{17}) * v_{15}) + v_{17}) + c_2) - v_7)) + v_6)) - v_{10})) + v_{17}) + 2.0) * v_{10})$

A. Features and Learned Policies

Input Features We use the 5-dimension constraint features, 1-dimension edge features and 19-dimension variable features based on the previous research (Gasse et al., 2019). See Table B for detailed descriptions for all the features.

Learned Policies We report the learned symbolic branching policies on all the benchmarks in Table B5. These policies directly take the general variable-constraint bipartite graph representation as input and output the scores of all the branching candidates. We observe that: 1) all these policies use the two-hop information of the bipartite graphs, which is consistent to the implementation of previous GNN-based approach (Gasse et al., 2019); 2) all the policies ignore the edge feature and only a small subset of node features appear in them, indicating that the useful information in the bipartite graph is sparse.

B. Implementation Details

Pseudo Code for GS4CO We provide the pseudo code of GS4CO in Algorithm 1. The algorithm includes two phase, i.e., the data collection phase and the model training phase. We execute FSB policy to collect expert demonstrations during the first phase, which is just the same as many previous approaches (Gasse et al., 2019; Gupta et al., 2020). Then, we employ PPO with the risk-seeking objective (Tamar et al., 2014) to train the sequential model at the second phase. This process is similar to previous research for symbolic regression tasks (Petersen, 2019; Landajuela et al., 2021), but we use the imitation learning accuracy as the fitness measure instead.

Model Architecture and Hyperparameters GS4CO leverages a decoder-only transformer to generate symbolic branching policies. Specifically, we report the network structure and the training hyperparameters in Table B7, in which we report both

Algorithm 1 Graph Symbolic Discovery for Exact Combinatorial Optimization Solver (GS4CO)

Input: the Transformer-based sequential model p_θ, the library of all the tokens $\mathcal{L}$ (features, operators, and constants), and the distribution of CO problems $\mathcal{I}$ from which we can generate new instances.
// Collect expert demonstration with strong branching:
Initial expert demonstration buffer: $\mathcal{D} \leftarrow \emptyset$.
while $|\mathcal{D}| < S$ **do**
  Sample CO problem $I \sim \mathcal{I}$, and solve I with full strong branching.
  Collect variable features and SB scores: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, u_{1:k_t})_{t=1}^{T}\}$.
end while
// Train the sequential model p_θ:
while not early stop **do**
  Sample J symbolic functions using tokens in $\mathcal{L}$: $\tau_{1:J} \sim p_\theta$.
  // Calculate the fitness of each symbolic function:
  for $j = 1, 2, \cdots, J$ **do**
    $r_j(\tau) = \mathop{\mathbb{E}}_{(\mathbf{s}_t, u_{1:k}) \sim \mathcal{D}} [\mathbf{I}_{\{i|u_i \geq u_j, 1 \leq j \leq k\}}(\mathop{\mathrm{argmax}}_{1:k} \tau(\mathbf{s}))]$
  end for
  Train the sequential model p_θ via PPO by optimizing $J(\boldsymbol{\theta}; \epsilon) = \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)}[r(\tau) | r(\tau) \geq r_\epsilon(\tau)]$.
end while
Output: Symbolic function τ_{best} with highest $r(\cdot)$ on the validation dataset.

Table B6. Instance generation algorithms and the detailed hyperparameters.

Benchmark	Algorithms	Hyperparameters
Set covering	(Balas & Ho, 1980)	Easy: 500 rows 1000 columns Medium: 1000 rows 1000 columns Hard: 2000 rows 1000 columns
Combinatorial auction	(Leyton-Brown et al., 2000)	Easy: 100 items for 500 bids Medium: 200 items for 1000 bids Hard: 300 items 1500 bids
Capacitated facility location	(Cornuéjols et al., 1991)	Easy: 100 facilities with 100 customers Medium: 100 facilities with 200 customers Hard: 100 facilities with 400 customers
Maximum independent set	(Bergman et al., 2015)	Easy: 750 nodes with affinity 4 Medium: 1000 nodes with affinity 4 Hard: 1500 nodes with affinity 4

the LSTM-based and the Transformer-based implementation of our GS4CO. All the other baselines are executed via the codes provided in Gupta et al. (2020), we use the official implementations with the default hyperparameter settings, and the detailed parameters can be referred in the papers (Alvarez et al., 2017; Gasse et al., 2019; Gupta et al., 2020). We execute all experiments on the machine with Intel Xeon Platinum CPUs @ 2.50GHz and NVIDIA Tesla V100 GPUs.

Symbolic Policy Deployment As pointing out by previous research (Gupta et al., 2020), extracting the variable-constraint bipartite graph state $\mathbf{s}$ (Gasse et al., 2019) is usually significantly faster than generating the human-designed fixed-length feature vector (Khalil et al., 2016). Thus, we directly use the graph feature extraction codes implemented in previous research (Gupta et al., 2020) and the learned Python expression for deployment with the official Python interface of SCIP (i.e., the PySCIPOpt package). However, we note that GS4CO learned policies are very similar to human-designed branching policies like pseudocost branching. Thus, it can also be implemented via C/C++ codes when considering the software distribution. We deploy the graph-based symbolic policies at the first 25 layers of the B&B tree and switch the branching policy to RPB when depth is larger than that, as in deep layers the RPB policy is both efficient and precise (Achterberg, 2007). This deployment method has limited effect for Easy and Medium datasets, since many B&B trees of these problems have a depth less than that, but it consistently improves the end-to-end performance on Hard datasets. All the instances are

solved with the SCIP seed 0 to reduce randomness from the solver.

Distilling Symbolic Functions from GNN Layers We distilling lightweight GNNs with only eight-dimensional embedding of messages to reduce the complexity of distilling. Specifically, we follow the methods proposed by Cranmer et al. (2020). That is, we firstly train a sparse GNN network, and then use the genetic programming package (Poli et al., 2008) to distill the message functions and the node embedding functions iteratively. We report the hyperparameters in GPLearn in Table B7.

Data Generation We follow the dataset generation process in (Gasse et al., 2019), using four CO problem benchmarks including set covering (Setcover), capacitated facility (Facilities), combinatorial auction (Cauctions) and maximum independent set (Indset). We set three levels (i.e. easy, medium and hard) of difficulty by increasing problem scales for each benchmark. For GS4CO, we generate ten training instances and four validation instances. To obtain the datasets of state-action pairs, we first leverage SCIP to solve the training instances. Then, we record new states and FSB decisions in the B&B process. We get 1,000 training samples with 400 for validation. For other ML-based approaches, we generate 10,000 instances for training and 2,000 instances for validation. Then, we attain 100,000 training and 20,000 validation state-action pairs on all benchmarks. We evaluate all the methods on 240 instances with a 3,000s time limit (80 easy, 80 medium and 80 hard instances). We list the instance generation algorithms with hyperparameters for each benchmark in Table B6.

Description of the Optimization Problem

- **Set covering(Setcover)**: Given a set of elements $\{1, 2, \ldots, n\}$ (called the universe) and a collection $\mathbf{S}$ of m subsets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of $\mathbf{S}$ whose union equals the universe.
- **Capacitated facility location (Facilities)**: The capacitated facility location problem involves determining the optimal placement of a set of facilities to serve multiple sites efficiently at the lowest cost. Each site has specific demand requirements, while each facility has capacity constraints. The cost associated with operating a facility includes a fixed opening fee and transportation costs incurred for serving each site from the facility.
- **Combinatorial auction (Cauctions)**: A combinatorial auction problem involves allocating multiple items to bidders where bids can be made on combinations of items rather than individual ones, aiming to maximize overall value while considering complex interactions between different item combinations and bids.
- **Maximum independent set (Indset)**: The Maximum Independent Set problem involves finding the largest set of vertices in a graph such that no two vertices are adjacent (i.e., there is no edge between them). In mathematical terms, it aims to find a subset $\mathbf{S}$ of the vertices in a graph $G = (V, E)$ such that for any two vertices $(u, v) \in \mathbf{S}$, the edge $(u, v) \notin E$, and $|\mathbf{S}|$ is maximized.

1-shifted Geometric Mean Formula We use the 1-shifted geometric mean for the experiment which is the same as (Gasse et al., 2019). Specifically, the formula is

$$G(x_1, x_2, x_3, ..., x_N) = (\prod_{k=1}^{N}(1 + x_k))^{\frac{1}{N}}$$

C. Further Discussions

The Comparison of Kuang et al. (2024) and GS4CO Compared with Kuang et al. (2024), the novelty of GS4CO lies in the following aspects.

1. The motivation. Our approach is the first symbolic approach that takes the graph representation as feature inputs. In contrast, Kuang et al. (2024) only handles human-designed fixed length features. In the variable selection task, Kuang et al. (2024) employs more than ninety different human-designed branching features, while a rich set of these features require extensive domain knowledge for their design and understand. For example, Kuang et al. (2024) uses eight features to describe the statistics over the ratios of a variable's coefficient to the sum over all other variables'. For non-experts in the field of OR, these features are complex for implementation and challenging to understand.

[5] Soft length prior is used in soft length regularizer (Landajuela et al., 2021)

2. The approach. Our approach aims to learn symbolic policies with graph inputs, while the cumulative error from iteratively distilling graph neural networks makes all the existing approaches fail. Thus, we propose a unified representation for symbolic graph policies and a novel model to generate symbolic graph policies end-to-end.

3. The impact. Our approach is a more significant step towards general algorithm discovery for combinatorial optimization (CO), as representing CO problems via graphs is a more general way for learning-based approaches.

In conclusion, compared with previous research, e.g.,Symb4CO (Kuang et al., 2024), Lion (Chen et al., 2023), AlphaDev (Mankowitz et al., 2023), the core challenge in this new research field is that we have to discover new algorithms and paradigms directly from graphs, which is a highly challenging and non-trivial research topic.

Analysis on What GS4CO Learned We emphasize that without enough expert knowledge from the research field of combinatorial optimization, quantifying the learned symbolic policies is challenging for researchers in the machine learning community. Thus, the analyses we provided in Section 4.2 are mainly high-level observations. Based on the insightful question of the reviewer, we further interpret the learned policy on Setcover (in Figure 1) as follows.

1. What is an important constraint? From the right part of Figure 1, a constraint is important if it is tight (feature c_3) currently and if it relates to variables that have a large objective coefficient (feature v_5) in average (the $mean$ aggregation operator).

2. What is a preferred variable for branching? From the left part of Figure 1, a preferred variable is usually related to many important constraints ($\hat{v}_1$) in total (the sum aggregation operator), and it usually has a larger variable fractionality (v_{10}) and objective coefficient (feature v_5).

Note that these observations are very consistent with our intuition and experience on the variable selection task. However, rather than simple human intuition, the learned symbolic policies give a more quantitative description of these correlations. These observations are still simple due to our lack of domain knowledge of CO solver design. However, we highly believe these policies can help the developers of CO solvers to further optimize the branching policies.

The Trade-Off between A Broader and A Narrower Search Paradigm Symbolic regression based algorithms do often encounter the dilemma of interpretability and prediction accuracy. Intuitively, neural networks can be viewed as a particular solution to the symbolic regression search space, which achieves high accuracy at the expense of good interpretability, while greatly increasing the cost of training and inference. In this paper, we choose to limit the upper bound on the prediction accuracy of the symbolic model to obtain better interpretability and ease of deployment. However, to further improve the performance of GS4CO, using more complex symbolic expressions is not the only way. Generally, there are roughly two types of symbolic discovery methods:

1. One type is based on purely mathematical expressions, usually in the form of symbolic regressions, to learn a symbolic mapping of functions, e.g. the DSR (Petersen, 2019).

2. The other type is a more general program expression that uses function logics like Python programs. For example, Lion (Chen et al., 2023) learns the flow of function execution to train neural networks.

Intuitively, the first type can be seen as a special case of the second. At this stage, the policy learned by GS4CO is more like a data-driven version of the human-designed pseudocost branching (PB) (Achterberg, 2007) policy integrated in modern CO solvers. However, in human-designed branching policies, experts have greatly improved the accuracy of branching strategies by expanding the simple scoring strategy PB into reliability pseudocost branching policy (RPB) with complex execution logic to combine the advantages of PB scoring and FSB scoring functions. Consequently, an exciting avenue for future work is to extend the symbolic policy into a more complex function flows like RPB via adding more function logics into the choice of primitives (as asked by Reviewer HrSu). Generally, this is a highly challenging task, as the searching space goes exponentially with the complexity of the learning objective, and branching policies like RPB is very complex (more than 1000 lines of C codes in SCIP). However, the rapid development of large language models gives us an exciting opportunity to employ prior knowledge to reduce the searching complexity for algorithm discovery.

D. Additional Experimental Results

Ablation on the Aggregation Layers vs. the Performance To show that GS4CO effectively leverages the variable-constraint graph structure, we conduct an ablation study to compare the imitation learning accuracy of GS4CO with different layers of neighborhood aggregations from zero to two. Specifically, this can be easily done via changing the aggregation function layer constraint (constraint 3 in Section 3.3). We report the results in the table below, which indicates that GS4CO effectively leverages the variable-constraint graph structure as expected. We report the results in Table D8, which indicates that GS4CO effectively leverages the variable-constraint graph structure as expected.

Reducing the Size of GNN and Hybrid to Improve the Inference Speed For GNN, we trained neural networks with embedding sizes $64, 32, 8$. We report the learning accuracy and the end-to-end solving time (seconds) on Setcover in Table D9. Results show that small GNNs tend to be more efficient on purely CPU-based devices, but they still fail to outperform the extremely lightweight graph symbolic policies. The main reason is that even with reduced hidden sizes, two-layer GNNs are still inefficient for inference compared with symbolic models. For Hybrid, we record the different parts of the inference time (in seconds) of Hybrid model on Setcover. That is, the time for feature extraction and the time for model inference. Results in Table D10 show that the main time cost for the inference of Hybrid approach is from feature extraction, rather than the model inference. More specifically, these results show that even if we consider a smaller Hybrid model that takes little time for model inference, its end-to-end performance still fails to outperform GS4CO.

Sensitivity to Different Sizes To further evaluate the sensitivity to problems generated with different parameters, we generate multiple datasets with a larger variance of generating hyperparameters. Specifically, we generate Setcover and Indset problems, and we report the performance of reliability pseudocost branching (RPB) policy in terms of time or gap, the relative performance of GS4CO, and the relative performance of GNNs on purely CPU-based devices. Table D11 includes results on Setcover (with rows and densities change) and Indset (with nodes and affinities change). Each entry of the table is evaluated on 20 instances, and the results are reported in the form of "RPB performance/GS4CO relative performance/GNN relative performance". Results show that symbolic policies are relatively insensitive to different problem sizes and difficulty.

Comparison between GS4CO and Symb4CO (Kuang et al., 2024) We implement both a Python version and a C version of the learned symbolic policies provided of Symb4CO (Kuang et al., 2024). For the Python version, the implementation is mainly based on the codes implemented by Gupta et al. (2020). Note that our approach is purely Python implemented. We report the accuracy and the end-to-end performance, and we conclude them as follows.

1. Results in Table D12 show that directly learning symbolic policies from the graph structure (GS4CO) achieves slightly lower accuracy than Symb4CO. The core reason accounting for this result is that the human-designed features employ highly complex rules to extract features that describe the structure of input problems. For example, a rich set of dynamic features in Gupta et al. (2020) are generated via complex logics to traverse the whole bipartite graph (see Lines 4631-4789 in scip.pyx). These can be regarded as human-designed two-layer aggregation functions with complex logics. In contrast, GS4CO extract graph structural features via simpler mathematical functions, and it is purely learning-based. However, we emphasise that GS4CO is more general and compatible to different downstream tasks. It requires only limited domain knowledge, which significantly reduces the challenges of feature design for researchers from the machine learning community.

2. The results in Table D13 show that Python implemented symbolic policies with graph inputs achieve slightly lower performance than the C++ implemented symbolic policies, in which the features requires extensive domain knowledge and human efforts for design. However, our graph symbolic policies outperform the Python implementation of Symb4CO (Kuang et al., 2024). The core reason is that extracting the whole set of fix-length features (i.e., the Khalil features) via the Python interface is very time-consuming. These results are consistent with the observations reported by Gupta et al. (2020).

Table B7. The hyperparameters for GS4CO.

Parameter	Value
GS4CO-Transformer (ours):	
Transformer embedding dimension	32
Transformer attention heads	4
Transformer feed-forward model dimension	128
Transformer number of layers	4
GS4CO-LSTM (for ablation):	
LSTM layers	2
LSTM hidden size	128
Distilling-GNN (for ablation):	
Embedding size of messages	8
population size	512
initial depth	4-6
tournament size	10
stopping criteria	0.05
training iterations	10000
GS4CO shared parameters:	
Batch size	500
Expression minimal length	4
Expression maximum length	64
Soft length prior λ	32
Soft length prior[5] σ^2	16
Maximum layers of aggregation functions	2
RL parameters:	
Risk factor	0.2
Number of training CO problems	100
Number of training FSB samples	$10,000$
Number of validation CO problems	20
Number of validation FSB samples	$2,000$
Training data batch size	1000
Hierarchical entropy regularizer γ	0.95
PPO learning rate	5e-5
PPO entropy coefficient	0.2
PPO epochs at each iteration	10
Optimizer	Adam (Kingma & Ba, 2015)
Number of iterations before early stop	2000

Table D8. The ablation study to compare the imitation learning accuracy of GS4CO with different layers of neighborhood aggregations from zero to two, which indicates that GS4CO effectively leverages the variable-constraint graph structure as expected.

Depth	Cauctions	Indset	Setcover	Facilities
0	37.12	26.75	33.65	65.86
1	40.25	39.25	41.40	66.50
2	41.13	40.78	44.13	66.78

Table D9. The learning accuracy and the end-to-end solving time (seconds) on Setcover with different embedding sizes. Results show that small GNNs tend to be more efficient on purely CPU-based devices, but they still fail to outperform the extremely lightweight graph symbolic policies.

Method	Accuracy	Time (s)
RPB	-	18.1
GNN-64(Default)	54.2	19.3
GNN-32(Default)	52.3	17.4
GNN-8(Default)	46.3	15.7
GS4CO	44.1	13.3

Table D10. Record the different parts of the inference time (in seconds) of Hybrid model on Setcover. Results show that the main time cost for the inference of Hybrid approach is from feature extraction, indicating even if we consider a smaller Hybrid model that takes little time for model inference, its end-to-end performance still fails to outperform GS4CO.

Hybrid				GS4CO
Total	Feature Extraction	Model Inference	(Total) − (Model inference)	Total
13.88	1.95	0.42	13.46	13.29

Table D11. Evaluate 1) the performance of reliability pseudocost branching (RPB) policy; 2) the relative performance of GS4CO; and 3) the relative performance of GNNs on purely CPU-based devices. Results show that symbolic policies are relatively insensitive to disturbances from different problem sizes and difficulty.

Row\Density	0.03	0.05	0.07
300	5.8/0.83/0.85	8.0/0.77/0.88	7.8/0.79/0.99
500	18.2/0.67/0.81	15.2/0.78/1.00	18.4/0.80/1.21
700	82.9/1.31/1.64	65.1/0.94/1.59	40.5/0.93/1.96

Nodes\Affinity	3	4	5
500 (time)	113.6/0.78/4.54	109.2/0.74/3.06	100.8/0.76/5.57
750 (time)	2537.0/0.73/0.97	2547.4/0.75/0.81	2230.0/0.80/1.11
1000 (gap)	5.14/0.81/1.05	5.37/0.86/1.12	5.42/0.81/1.08

Table D12. Compare the learning accuracy between Symb4CO and GS4CO.

Method	Cauctions	Indset	Setcover	Facilities
Symb4CO	43.42	45.31	45.37	66.50
GS4CO	41.13	40.78	44.13	66.78

Table D13. Compare the end-to-end performance between Symb4CO and GS4CO.

Method	Cauctions	Indset	Setcover	Facilities
RPB	5.11	75.14	18.12	78.75
Symb4CO-Python	3.31	60.47	14.92	73.32
Symb4CO-C	**2.87**	55.43	13.45	**60.26**
GS4CO(-Python)	3.16	**53.18**	**13.29**	76.27