GNN&GBDT-Guided Fast Optimizing Framework for Large-scale Integer Programming

Huigen Ye [1 2] **Hua Xu** [1 2] **Hongyan Wang** [1 2] **Chengming Wang** [3] **Yu Jiang** [3]

Abstract

The latest two-stage optimization framework based on graph neural network (GNN) and large neighborhood search (LNS) is the most popular framework in solving large-scale integer programs (IPs). However, the framework can not effectively use the embedding spatial information in GNN and still highly relies on large-scale solvers in LNS, resulting in the scale of IP being limited by the ability of the current solver and performance bottlenecks. To handle these issues, this paper presents a GNN&GBDT-guided fast optimizing framework for large-scale IPs that only uses a small-scale optimizer to solve large-scale IPs efficiently. Specifically, the proposed framework can be divided into three stages: Multi-task GNN Embedding to generate the embedding space, GBDT Prediction to effectively use the embedding spatial information, and Neighborhood Optimization to solve large-scale problems fast using the small-scale optimizer. Extensive experiments show that the proposed framework can solve IPs with millions of scales and surpass SCIP and Gurobi in the specified wall-clock time using only a small-scale optimizer with 30% of the problem size. It also shows that the proposed framework can save 99% of running time in achieving the same solution quality as SCIP, which verifies the effectiveness and efficiency of the proposed framework in solving large-scale IPs.

[1]State Key Laboratory of Intelligent Technology and Systems, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China [2]Beijing National Research Center for Information Science and Technology(BNRist), Beijing 100084, China [3]Meituan Inc., Block F&G, Wangjing International R&D Park, No.6 Wang Jing East Rd, Chaoyang District, Beijing, 100102, China. Correspondence to: Hua Xu <xuhua@tsinghua.edu.cn>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

1. Introduction

Integer programs (IPs) are an essential type of problem that can model a large number of combinatorial optimization problems such as routing(Shaw, 1998), network designing(Du & Pardalos, 1998), and bin packing(man Jr et al., 1996). However, solving real-world large-scale IPs is challenging because of their NP complexity (Paulus et al., 2020) and high dimensionality (Martin, 2012). Therefore, how to obtain a better feasible solution within the limited computing time has become a research hotspot in this field (Song et al., 2020a; Ye et al., 2022).

To solve IPs within a limited computing time, scholars have proposed many efficient methods in the past few decades. Currently, widely used techniques can be divided into two categories (Zhang et al., 2023): *the exact algorithm based on branch and bound* and *the approximate algorithm based on heuristics*.

The representative work of *the exact algorithm based on branch and bound* was first proposed by Land in 1960 (Land & Doig, 1960) as a classic tree search method. By dividing and conquering, it divides the search space by branching and cleverly uses the problem's relaxed boundary to delete hopeless areas from the tree, which can effectively reduce the search space and solve the problem of excessive combination space. However, solving an integer program (IP) may require millions of branching decisions, and one wrong decision may cause a doubled tree size. To improve the accuracy of branching, Bénichou first proposed pseudo-cost branching (Bénichou et al., 1971), which is efficient but unreliable in the early stage of tree search. Applegate then suggested strong branching (Applegate et al., 1995), which is more reliable but time-consuming. In order to combine the above methods' advantages, Achterberg proposed hybrid branching (Achterberg & Berthold, 2009), flexible use of pseudo-cost branching and strong branching in different branching stages, and has become the most commonly used branching method for solvers. However, the above artificially designed branching methods often manifest deficiencies in the face of increasingly complex optimization problems with large data. Using machine learning technology to "learn to branch" is an attractive improvement direction, including variable selection (Nair et al., 2020), node selection (Song et al., 2020b), and cutting plane selection (Huang et al., 2022). However, even if machine learning methods are used to assist in making branching decisions, the size of the search tree will increase exponentially, and the solution to the problem will become too complicated, when faced with large-scale problems.

Therefore, many scholars turn to *the approximate algorithm based on heuristics*, whose goal is to combine greedy algorithms and search ones to solve large-scale IPs, including classical heuristic algorithms and machine learning based ones. Classical heuristic algorithms have been widely used to solve IPs since the 1960s. Line Search Methods (Hillier, 1969) and Pivoting Methods (Balas & Martin, 1980) are the first batches of heuristic algorithms applied to IPs. To overcome the problem that the early algorithm may not find a feasible solution for a long time, Fischetti proposed the feasibility pump (Fischetti et al., 2005) to enhance the ability to find the initial solution. In order to further improve the quality of the solution, Rothberg proposed an evolutionary algorithm (Rothberg, 2007), and Pisinger proposed a large neighborhood search algorithm (Pisinger & Ropke, 2010), trying to improve the solution from the perspective of the population and the individual respectively. However, the above method has the ”cold-start” problem and does not use the information obtained from solving the same type of problems, resulting in wasting resources. Therefore, some work in recent years aims better to learn the commonality between problems through machine learning to improve the original heuristic algorithm, especially the large neighborhood search. The effect of the large neighborhood search mainly depends on the selection of the initial solution and the neighborhood. In terms of initial solution selection, Sonnerat (Sonnerat et al., 2021) introduced neural diving to obtain a better initial solution. In terms of neighborhood selection, Song (Song et al., 2020a)trained a neighborhood selection strategy through imitation learning and reinforcement learning to find a better neighborhood selection. At present, the latest framework is the two-stage framework of prediction-and-optimization based on graph neural network(GNN) and large neighborhood search. The general block diagram is depicted in Figure 1.

Although the two-stage framework in Figure 1 has achieved good results in many real-world IPs, there exist three shortcomings. Firstly, the GNN-based embedding process shares the same loss function with the prediction process, leading to poor properties of the embedding space. Secondly, the MLP-based prediction process is too simple to fully use the embedding space, generating less guidance information for the optimization stage. Finally, the large-neighborhood search-based optimization stage still searches for optimal solutions in high-dimensional space, relying on large-scale algorithms with huge complexity. Generally, the solution efficiency is poor.

To solve the shortcomings above, this paper proposes GNN&GBDT-guided fast optimizing framework for large-scale integer programming problem. The framework is divided into three stages. The stage of Multi-task GNN Embedding uses the multi-task framework on GNN (Jian et al., 2022) to make the decision variable embedding instruct the next stage. The stage of GBDT prediction introduces Gradient Boosting Decision Tree (GBDT) (Friedman, 2001) so that while producing prediction results, the loss function of the leaves and the partition of embedding space by the decision trees are instructive for the optimization stage. The stage of Neighborhood Optimization introduces neighborhood search with fixed radius and neighborhood crossover to solve large-scale problems fast using small-scale solving algorithms. The framework has been tested in four classic benchmark IPs and one real-world IP. Using small-scale algorithms to solve large-scale problems within a specified time manifests obvious advantages compared with complete large-scale solvers (e.g., Gurobi, SCIP), which has proved the effectiveness and efficiency of the proposed framework.

There are three main innovations in this paper.

- First, we introduce the multi-task GNN with half-convolutions layers and random feat policy into the embedding process. Since decision variable embedding could be used to instruct the next stage, the proposed framework can give embedding more good properties.
- Second, we introduce the GBDT into the prediction process to make it more efficient and can pass the intermediate information for the optimization stage.
- Third, we propose a Neighborhood Search with a fixed search radius and small-scale neighborhood crossover into the optimization stage. Enables solving large-scale problems fast using small-scale optimizers. The experiment shows that our method can solve IPs with millions of scales and surpass Gurobi in the specified wall-clock time using only a small-scale algorithm with 30% of the problem size.

2. Related Work

2.1. Integer Programs

Integer Programs (IPs) are a type of problem of maximizing or minimizing a linear expression subject to a number of linear constraints, where all decision variables are restricted to take integer values (Williams, 2009). Formally, an integer program can be defined as the following.

$$\begin{aligned} &\min_x c^T x, \\ &\text{subject to } Ax \leq b, l \leq x \leq u, x \in \mathbb{Z}^n, \end{aligned} \tag{1}$$

where n is the number of decision variables, with $c, l, u \in \mathbb{R}^n$ being their coefficient, lower bound and upper bound.

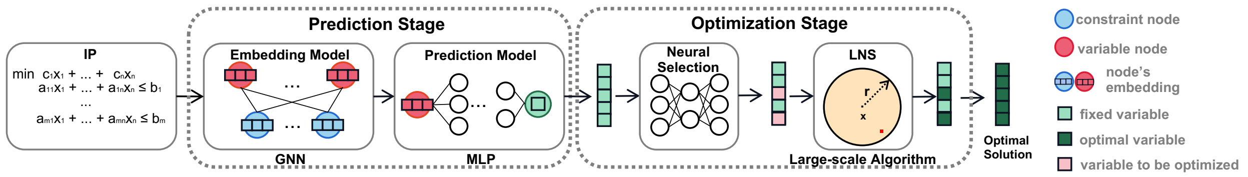

Figure 1. The two-stage framework of prediction-and-optimization. The prediction stage adopts a prediction model combining a graph neural network(GNN) based embedding model and a multi-layer perceptron(MLP) based prediction model. The optimization stage adopts a large neighborhood search strategy under the guidance of machine learning.

$A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ denote the linear constraints on x. Any element $x \in \mathbb{R}^n$ that satisfies all the constraints in formula (1) is a feasible solution of the IP, and a feasible solution attaining the minimum objective function value is called an optimal solution (Schrijver, 1998).

2.2. Bipartite Graph Representation

The Bipartite Graph Representation of IPs was proposed by Gasse in 2019 (Gasse et al., 2019), which can realize the lossless encoding of the original IP into a bipartite graph as the input of the Graph Neural Network, described in Figure 2.

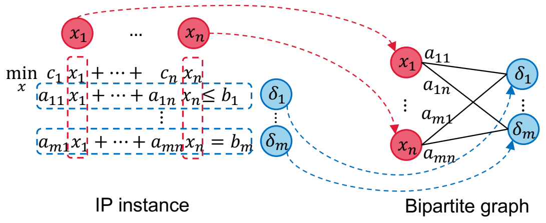
x_1
x_n
$\min_x \; c_1x_1 + \cdots + c_nx_n$
$a_{11}x_1 + \cdots + a_{1n}x_n \le b_1$
$a_{m1}x_1 + \cdots + a_{mn}x_n = b_m$
δ_1
δ_m
a_{11}
a_{1n}
a_{m1}
a_{mn}
IP instance
Bipartite graph

Figure 2. Transforming an IP instance to a bipartite graph. The set of n decision variables nodes $\{x_1, \ldots, x_n\}$ and the set of m constraint nodes $\{\delta_1, \ldots, \delta_m\}$ form the left set and right set of nodes of the bipartite graph.

The left set of n variable nodes in the bipartite graph represents the decision variables, while the right set of m constraint nodes represents the linear constraints. An edge (i, j) with edge weight a_{ij} connecting the left node i and right node j represents that the i-th decision variable appears in the j-th constraint, and the coefficient is a_{ij}. In the classic bipartite graph representation application, the feature selection of nodes and edges usually only depends on the coefficients in formula (1), such as upper bound l, lower bounds r, etc.

However, recently some studies have shown that this type of feature selection strategy may lead to a significant decline in the embedding ability of GNN in some particular IPs called "foldable" (Chen et al., 2022). In order to overcome this shortcoming, the random feat strategy is introduced into the feature selection of bipartite graph representation, which can be proved in recent papers' experiments (Chen et al., 2022).

2.3. Graph Neural Network

In IPs, a GNN is often used for model learning and neighborhood aggregation after bipartite graph representation. Formally, let $\mathcal{E}$ denote the set of edges in the bipartite graph, a k layer GNN could be written as below.

$$h_v^k = f_2^k(\{h_v^{(k-1)}, f_1^k(\{h_u^{(k-1)} : (u,v) \in \mathcal{E}\})\}), \quad (2)$$

where h_v^k denotes the hidden state of node v in the k-th layer. The function f_1^k combines the hidden value of the $(k-1)$-th layer of the neighbors to get the aggregation information, and function f_2^k combines the hidden value of the current point v and the aggregation information of its neighbors.

But for the graph structure of the bipartite graph, the GNN with two interleaved half-convolutions layers is the most popular structure now (Gasse et al., 2019; Yoon, 2022). Formally, let $\mathcal{V}_x$ denote the set of n variable nodes and $\mathcal{V}_\delta$ denote the set of m constraint nodes, a k layer half-convolutions GNN could be written as the following.

$$h_{\delta_j}^k = f_\delta^k(\{h_{\delta_j}^{(k-1)}, \sum_{(x_i,\delta_j)\in\mathcal{E}} g_\delta^k(\{h_x^{(k-1)}, h_{\delta_j}^{(k-1)}\})\}), \delta_j \in \mathcal{V}_\delta,$$
$$h_{x_i}^k = f_x^k(\{h_{x_i}^{(k-1)}, \sum_{(x_i,\delta_j)\in\mathcal{E}} g_x^k(\{h_{x_i}^{(k-1)}, h_{\delta_j}^k\})\}), x_i \in \mathcal{V}_x, \quad (3)$$

where h_x^k, h_δ^k, g_x^k and g_δ^k are all MLPs, and the activation function is ReLU.

2.4. Gradient Boosting Decision Tree

In the prediction stage, optional prediction methods include MLP, support vector machine, decision tree, etc. This paper chooses the Gradient Boosting Decision Tree (GBDT) (Friedman, 2001) based on the regression tree.

For a given data set containing n examples and m features $\mathcal{D} = \{x_i, y_i\}$ where $|D| = n$, $x_i \in \mathcal{R}^m$ and $y_i \in \mathcal{R}$, GBDT tries to use K regression trees to fit the data set. The final prediction result of GBDT is the weighted sum of T

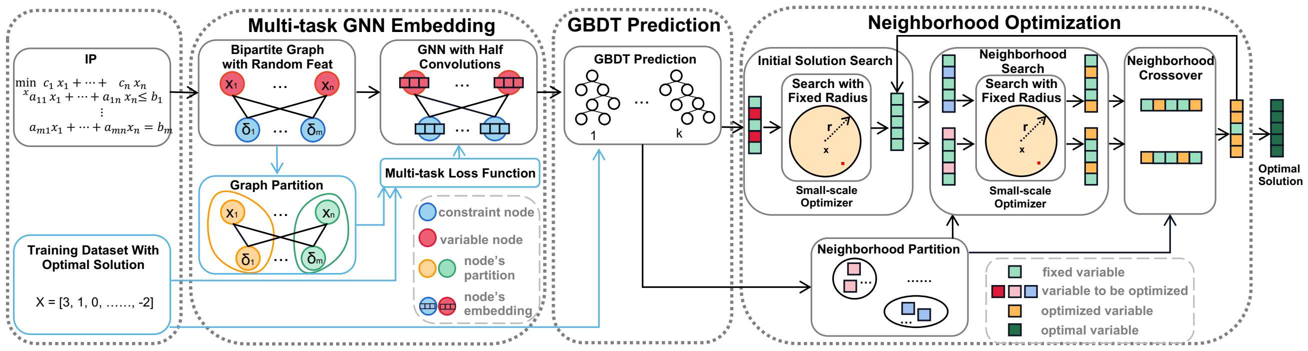

Figure 3. An overview of GNN&GBDT-Guided Fast Optimizing Framework for Large-scale Integer Programming. The blue line indicates that it is only used during training, while the black line indicates that it is used during both training and testing. In the stage of Multi-task GNN Embedding, the IP is represented as a bipartite graph, followed by employing a graph partition algorithm(FENNEL) to divide the bipartite graph into blocks. Then a multi-task GNN with half convolutions is used to learn the embedding of decision variables, where the loss function is a metric for both optimal solution and graph partition. In the stage of GBDT prediction, the GBDT is used to predict the optimal value of the decision variable in the IP through the variable embedding. In the stage of Neighborhood Optimization, some decision variables are fixed as the rounding results of the predicted values of GBDT and a search with fixed radius is used to find an initial solution. Then, under the guidance of the neighborhood partition, neighborhood search and neighborhood crossover are used iteratively to improve the current solution.

regression trees-based model prediction.

$$\hat{y_i} = \phi(x_i) = \sum_{t=1}^{T} f_t(x_i), \tag{4}$$

To fit the data set, GBDT uses the additive manner to minimize the following objective function. Formally, let $\hat{y_i}^{(t-1)}$ be the sum of predictions of $t-1$ regression trees for the i-th instance, the objective of the t-th regression tree can be written as the following.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)), \tag{5}$$

where l is an arbitrary metric function to represent the distance between the prediction result of GBDT and the target y_i. Applying second-order Taylor expansion approximation to formula (5) and defining $I_j = \{i|q(x_i) = j\}$ to be an instance in leaf j, the optimal weight w_j^* of leaf j can finally be written as the following.

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i}, \tag{6}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ are first-order and second-order gradient statistics of function l. The optimal value of the objective function can be also written as the following.

$$\overline{L}^{(t)} = -\frac{1}{2}\sum_{j=1}^{T} \frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i}. \tag{7}$$

For the structure of the $t-th$ regression tree, GBDT greedily adds branches from the root. Assuming I_L and I_R are the instance sets of left and right nodes after branching, let $I = I_L \cup I_R$, the objective of the branching is to maximize the loss reduction after branching.

$$L_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i} + \frac{(\sum_{i\in I_E} g_i)^2}{\sum_{i\in I_R} h_i} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i}\right]. \tag{8}$$

3. Method

The proposed GNN&GBDT-Guided Fast Optimizing Framework for Large-scale Integer Programming is divided into three stages: Multi-task GNN Embedding, GBDT prediction, and Neighborhood Optimization, which is illustrated in Figure 3.

3.1. Multi-task GNN Embedding

Given an IP to be solved, it would be represented as a *bipartite graph*. Then a *graph partition algorithm* is used to divide the bipartite graph into blocks. Further, a *multi-task GNN* with half convolutions is used to learn the embedding of decision variables, where the loss function is a metric for both optimal solution and graph partitions. Based on the above steps, we obtain the neural feature embedding of decision variables for the next stage.

Bipartite Graph Representation. On the basis of classic bipartite graph representation introduced in (Sec. 2.2), a new random feat-based (Chen et al., 2022) feature selection method is introduced to achieve better representation and embedding capabilities in GNN. Formally, let $h_x^i, h_\delta^i, h_{(i,j)}$ denotes the feature selection of the i-th variable node, j-th constraint node and edge (i,j).

$$\begin{aligned} h_x^i &= (c_i, l_i, u_i, \xi), \\ h_\delta^j &= (b_j, \mathfrak{o}_j, \xi), \\ h_{(i,j)} &= (a_{ij}), \end{aligned} \tag{9}$$

where c_i, k_i, u_i denotes the coefficient, lower bound and upper bound of the i-th decision variable, $b_j, \mathfrak{o}_j$ denotes the value and symbol of the j-th constraint, a_{ij} denotes the weight of edge (i,j) and $\xi \sim U(0,1)$ denotes the uniform random feat between 0 and 1.

Graph Partition Algorithm. In order to partition the decision variables according to the correlation, the decision variables that appear in multiple constraints together should be as close as possible. So the FENNEL-based (Tsourakakis et al., 2014) graph partition algorithm is used for the bipartite graph representation $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of the IP. The graph partition algorithm will summarize the decision variables with dense edges or high correlation into one block, while dividing the decision variables with sparse edges or low correlation into separate blocks. The result of the graph partition is transferred to GNN as a training label. FENNEL is a stream algorithm, which means that it checks each node v in the graph one by one and calculates $\delta_g(v, \mathcal{P}_i)$ with each block P_i. The calculation method is as follows.

$$\delta_g(v, \mathcal{P}_i) \leftarrow |\mathcal{P}_i \cap N(v)| - \alpha\gamma|P_i|^{\gamma-1}, \tag{10}$$

where $N(v)$ denotes the neighbor node set of v, α, γ are preset parameters related to block balancing and minimum cut. The details of the graph partitioning algorithm based on FENNEL are shown in Appendix.

Multi-task GNN. On the based GNN with half convolutions (Gasse et al., 2019; Yoon, 2022) introduced in (Sec. 2.3), the multi-task strategy is used for embedding space learning of GNN. The multi-task GNN can not only make the embedding of variables with the same optimal solution value as close as possible but also make the embedding of variables that appear in multiple constraints together as close as possible.

These two tasks are essential to make the points of the same label as close as possible while the points of different labels as far as possible in the embedding space. To achieve it, metric learning (Kulis et al., 2013) is used to re-learn the distance relationship between variable embeddings. Formally, for the training data $X = \{h_{x_i}^0, \mathcal{G}_i, \text{Opt}_i\}_{i=1}^N$, where $\mathcal{G}_i$ represents the graph partition result and Opt_i denotes the optimal solution of the decision variables, ProxyAnchorLoss (Kim et al., 2020) is used to be the loss function. Following the method of ProxyAnchorLoss, proxy set will be assigned for each class, which is initialized with a normal distribution to ensure they are uniformly distributed on the unit hypersphere, effectively representing a class's centroid. Setting the proxy sets of $\mathcal{G}$ and Opt as $\mathfrak{G}$ and $\mathfrak{O}$ respectively, the two tasks' loss functions can be written as follows.

$$\begin{aligned} loss_1(X) =& \frac{1}{|\mathfrak{G}^+|} \sum_{f \in \mathfrak{G}^+} log(1 + \sum_{x \in X_{\mathfrak{G}}^+} e^{-\alpha(s(x,f)-\delta)}) \\ &+ \frac{1}{|\mathfrak{G}|} \sum_{f \in \mathfrak{G}} log(1 + \sum_{x \in X_{\mathfrak{G}}^-} e^{\alpha(s(x,f)+\delta)}), \end{aligned} \tag{11}$$

$$\begin{aligned} loss_2(X) =& \frac{1}{|\mathfrak{O}^+|} \sum_{f \in \mathfrak{O}^+} log(1 + \sum_{x \in X_{\mathfrak{O}}^+} e^{-\alpha(s(x,f)-\delta)}) \\ &+ \frac{1}{|\mathfrak{O}|} \sum_{f \in \mathfrak{O}} log(1 + \sum_{x \in X_{\mathfrak{O}}^-} e^{\alpha(s(x,f)+\delta)}), \end{aligned} \tag{12}$$

where $\delta > 0$ denotes the margin, $\alpha > 0$ denotes the scaling factor, $s(.,.)$ represents the cosine similarity between two vectors, $\mathfrak{G}^+$ and $\mathfrak{O}^+$ are the sets of positive proxies. The final objective can be formulated as the following (Jian et al., 2022).

$$\mathcal{L} = \beta loss_1 + (1-\beta) loss_2, \tag{13}$$

where $\beta \in [0,1]$ is the hyper-parameter to control the balance.

3.2. GBDT prediction

After getting the embedding of decision variables from the previous stage, the GBDT (Friedman, 2001) introduced in (Sec. 2.4) is used to *predict the optimal value* of the decision variables in the IP through the variable embeddings, and *generating the guidance* for the initial solution search and neighborhood partition in the next stage.

Variable Value Prediction. In the decision variables predicting problem, the GBDT is used to predict the optimal value of the decision variables. A training data set $D = \{h_{x_i}^K, \text{Opt}_i\}_{i=1}^N$ is used to construct GBDT. The trained GBDT can predict the variable's value in the optimal solution through the weighted accumulation of the prediction results of each regression tree that can be rewritten as follows according to formula (4).

$$\hat{y}_i = \phi(h_{x_i}^K) = \sum_{t=1}^T f_t(h_{x_i}^K), \tag{14}$$

And the prediction loss and embedding space partition of each regression tree will be used to guide the initial solution search and neighborhood partition in the next stage.

Guidance Generation. Two kinds of guidance information can be used to guide the optimization work in the next stage.

On one hand, because the essence of regression tree-based GBDT prediction is to partition the embedding space, the partition result will be used to guide the neighborhood partition. Multiple decision trees are used to ensure that GBDT can have good performance, each with its weight. In the process of neighborhood partition in each round, a decision tree is randomly selected based on its weight as the partition criterion for this round, using its partition result to guide the neighborhood partition. The larger the weight of the decision tree, the more likely it is to be selected. This way, it uses decision tree partitioning as the neighborhood partitioning criterion and retains some randomness to prevent getting stuck in local optima.

On the other hand, when the loss function of GBDT is defined as the squared error, the value of the loss function essentially represents the square of the physical distance. In this case, for multiple decision variables of a training instance, the sum of the loss function can measure the distance to the optimal solution and has physical significance. Therefore, the weighted accumulation of each regression tree's prediction loss will guide which decision variables will be fixed in the initial solution search. Formally, let $\mathcal{I}_j = I_1, \ldots, I_T$ represent the leaf nodes in the T regression trees that constitute the prediction results of the decision variable x_j. According to formula (7), the weighted accumulation of the prediction loss can be written as follow.

$$\mathcal{P}_j = \sum_{t=1}^{T} \beta_t \overline{L}_{i_t}^{(t)} = \sum_{t=1}^{T} -\frac{1}{2}\beta_t \frac{(\sum_{i\in I_t} g_i)^2}{\sum_{i\in I_t} h_i} \tag{15}$$

3.3. Neighborhood Optimization

In the stage of neighborhood optimization, for solving a large-scale IP with n decision variables, a variable proportion $\alpha \in (0, 1)$ needs to be defined first, which means that an optimizer that can solve a small-scale IP with αn decision variables can be used to solve the corresponding large-scale IP. In order to realize this stage, a search with a fixed radius is used to *search an initial solution* first. Then under the guidance of *neighborhood partition*, *neighborhood search* and *neighborhood crossover* are used iteratively to improve the current solution. Finally, when the predetermined time is reached, the current solution is output as the optimization result.

Initial Solution Search. Given the predicted value $\hat{y}_i$ and the prediction loss $\mathcal{P}_i$ of each decision variable, the decision variables are sorted according to the prediction loss from small to large. After that, the first $(1-\alpha)n$ decision variables are fixed, while the remaining variables are searched with a fixed radius. The details are shown in Algorithm 1.

In Algorithm 1, $\eta \in (0, 1)$ is a reduction coefficient used to expand the fixed proportion, REPAIR () is a function used to cancel the fixation of illegal variables that will be shown in Appendix, SEARCH () is a search with a fixed radius that needs to use the small-scale optimizer. The formal expression of the search with a fixed radius is as follows.

Algorithm 1 Initial Solution Search

```
Input: The number of decision variables $n$, predicted value $\hat{y}$, prediction loss $\mathcal{P}$, variable proportion $\alpha$
Init: Initial Solution $\mathcal{X} = \{\}$
$\mathcal{X} \leftarrow \hat{y}$
Sort the decision variables in ascending order of $\mathcal{P}$
$\alpha_{set} = \alpha$
repeat
    $\mathcal{F} \leftarrow$ The first $(1-\alpha_{set})n$ decision variables    ▷Fixed
    $\mathcal{U} \leftarrow$ The last $\alpha_{set}$ decision variables    ▷Unfixed
    $\mathcal{F}', \mathcal{U}' \leftarrow$ REPAIR($\mathcal{F}, \mathcal{U}, \mathcal{X}$)
    if $|\mathcal{U}'| > \alpha n$ then
        $\alpha_{set} = \eta * \alpha_{set}$
    end if
until $|\mathcal{U}'| \leq \alpha n$
$\mathcal{X} \leftarrow$ SEARCH($\mathcal{F}', \mathcal{U}', \mathcal{X}$)
Return: $\mathcal{X}$
```

$$\begin{aligned} &\min_{x\in \text{Unfixed}} \quad c^T x \\ &\text{subject to } Ax \leq b, l \leq x \leq u, x \in \mathbb{Z}^n, \\ &\qquad\qquad x_i = \hat{x}_i, \forall x_i \in \text{Fixed}, \end{aligned} \tag{16}$$

where $\hat{x}_i$ denotes the value in current solution of the i-th decision variable.

Neighborhood Partition. Every iteration of neighborhood search and neighborhood crossover requires a new neighborhood partition. Since the embedding of variables with a strong correlation are close, the embedding space partition result of the regression tree in GBDT is directly used as the neighborhood partition. Figure 4 is an example of the neighborhood partition.

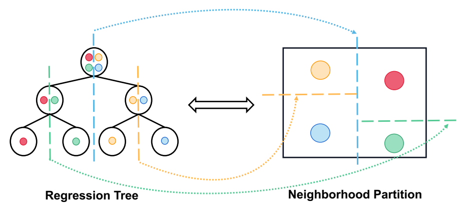

Figure 4. Using the partition result of the regression tree as the neighborhood partition. Different branches on the left regression tree correspond to the partitions of the right embedding space.

Neighborhood Search. Based on the neighborhood partition results, the current solution is used to explore neighborhoods in parallel. Specifically, for the i-th neighborhood N_i, the details in neighborhood search are shown in Algorithm 2.

Algorithm 2 Neighborhood Search

```
**Input:** The set of decision variables $X$, the number of decision variables $n$, predicted value $\hat{y}$, prediction loss $\mathcal{P}$, variable proportion $\alpha$, neighborhood $N_{now}$, current solution $\mathcal{X}$
**Init:** Neighborhood search solution $\mathcal{X}' = \{\}$
Sort the decision variables in $N_{now}$ in descending order of $\mathcal{P}_i * |\phi_i - \hat{y}_i|$
$\mathcal{N} \leftarrow$ The first $\alpha n$ decision variables in $N_{now}$
$\mathcal{F} \leftarrow \{x \| x \in X \wedge x \notin N\}$    ▷Fixed
$\mathcal{U} \leftarrow \{x \| x \in X \wedge x \in N\}$    ▷Unfixed
$\mathcal{X}' \leftarrow$ SEARCH($\mathcal{F}$,$\mathcal{U}$, $\mathcal{X}$)
**Return:** $\mathcal{X}'$
```

In Algorithm 2, in order to keep the neighborhood within the search range of the optimizer, a new evaluation index is proposed as shown below to select the decision variables that are most likely to modify the value.

$$\mathcal{Q}_i = \mathcal{P}_i * |\mathcal{X}_i - \hat{y}_i|, \tag{17}$$

where $\mathcal{P}_i$, $\mathcal{X}_i$ and $\hat{y}_i$ denotes the prediction loss, the value in current solution and the predicted value of the i-th decision variable respectively. Here $\mathcal{Q}_i$ is used to assess the confidence level of each decision variable's current solution, i.e., the larger $\mathcal{Q}_i$, the lower the confidence level of that decision variable, and the more likely it is to be set as an unfixed decision variable to be optimized.

Neighborhood Crossover Since the size of the neighborhood is limited to no more than αn, it is easy to fall into the local optima because the radius of the neighborhood search is too small. So neighborhood crossover is crucial. Based on the result of neighborhood partition, neighborhood crossover is carried out step by step, as shown in Figure 5.

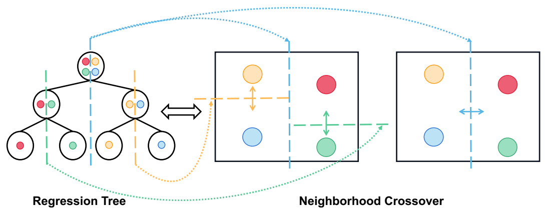

Figure 5. Carrying out neighborhood crossover step by step. First, the neighborhood partition corresponding to the second layer of branching of the regression tree is crossed by neighborhood crossover. Then the crossover over the neighborhood partition corresponding to the first layer of branching is similar to the second layer.

For the two neighborhoods N_1, N_2 to be crossed, the details are shown in Algorithm 3.

Algorithm 3 Neighborhood Crossover

```
**Input:** The set of decision variables $X$, the number of decision variables $n$, neighborhood $N_1, N_2$, neighborhood search solution $\mathcal{X}'_1, \mathcal{X}'_2$
**Init:** Neighborhood crossover solution $\mathfrak{X} = \{\}$
$\mathcal{X}'' \leftarrow \{\}$
**for** $i = 1$ **to** $n$ **do**
  **if** The $i$-th decision variables in $N_1$ **then**
    $\mathcal{X}''[i] \leftarrow \mathcal{X}'_1[i]$
  **else**
    $\mathcal{X}''[i] \leftarrow \mathcal{X}'_2[i]$
  **end if**
**end for**
$\mathcal{F} \leftarrow X$    ▷Fixed
$\mathcal{U} \leftarrow \emptyset$    ▷Unfixed
$\mathcal{F}'$,$\mathcal{U}' \leftarrow$ REPAIR($\mathcal{F}$,$\mathcal{U}$, $\mathcal{X}''$)
**if** $|\mathcal{U}'| \leq \alpha n$ **then**
  $\mathfrak{X} \leftarrow$ SEARCH($\mathcal{F}'$,$\mathcal{U}'$, $\mathcal{X}''$)
**end if**
**Return:** $\mathfrak{X}$
```

4. Experiments

Experiments are performed on one real-world large-scale IP in the internet domain and four widely used NP-hard benchmark IPs: Combinatorial Auction (CA) (De Vries & Vohra, 2003), Maximum Independent Set (MIS) (Tarjan & Trojanowski, 1977), Minimum Vertex Covering (MVC) (Dinur & Safra, 2005) and Set Covering (SC) (Caprara et al., 2000). The detailed dataset and environment introduction are shown in Appendix. The state-of-the-art IP solvers SCIP (Achterberg, 2009) and Gurobi (Achterberg, 2019) are used as the baseline, and their scale-constrained versions are used as the small-scale optimizer for neighborhood optimization.

In order to show the benefits of the proposed GNN&GBDT-Guided Fast Optimizing Framework for Large-scale Integer Programming, a comprehensive computational study is carried out as follows. First of all, we compare the optimization results between the proposed framework and baseline in the same wall-clock time (Sec. 4.1). Moreover, we compare the running time of the proposed framework with the baseline under the same optimization result (Sec. 4.2). All experiments are repeated five times and the metric average is recorded. Finally, we analyze the convergence performance of the proposed framework (Sec. 4.3). Code for reproducing all the experiments can be found at https://github.com/thuiar/GNN-GBDT-Guided-Fast-Optimizing-Framework.

4.1. Comparison of Objective Value

In order to verify the ability of the proposed framework to solve different types of large-scale IPs, this paper has

Table 1. Comparison of objective value results with SCIP and Gurobi under the same running time on different benchmark IPs. Ours-20%S means the scale-limited versions of SCIP which limit the variable proportion α to 20%. Ours-20%G means the scale-limited versions of Gurobi which limit the variable proportion α to 20%. ↑ means the result is better than the baseline. - means that no feasible solution is found.

	IP_1	IP_2	CA_2	CA_3	MIS_2	MIS_3	MVC_2	MVC_3	SC_2	SC_3
SCIP	-	901099.8	11235.7	115277.7	18552.6	9086.6	31180.4	491444.9	-	272189.6
Ours-20%S	232179.9↑	924740.0↑	13160.4↑	131994.1↑	21865.5↑	220045.8↑	27776.3↑	282199.6↑	17870.6↑	291778.9
Ours-30%S	232594.6↑	**940877.9**↑	13663.6↑	137110.9↑	22294.2↑	223342.0↑	27419.6↑	276258.6↑	17174.8↑	**225681.4**↑
Ours-50%S	**232967.7**↑	901099.8↑	**14017.6**↑	**139939.4**↑	**22728.4**↑	**228148.4**↑	**27015.7**↑	**272189.6**↑	**16724.7**↑	244132.8↑
Gurobi	226336.4	907252.1	12944.4	130008.6	21780.3	216565.9	28088.8	283904.9	17953.6	320305.7
Ours-20%G	232254.7↑	933815.5↑	12788.9	131466.9↑	21867.6↑	215494.4	28021.6↑	287434.6	17415.7↑	231356.8↑
Ours-30%G	232700.6↑	943328.7↑	**13426.8**↑	134685.9↑	22158.4↑	218109.7↑	27737.4↑	280458.9↑	17052.6↑	229642.5↑
Ours-50%G	**241030.8**↑	**951097.3**↑	12466.0↑	**138835.0**↑	**22626.7**↑	**225088.5**↑	**27215.6**↑	**275596.0**↑	**16882.7**↑	**219448.8**↑
Time	60s	3000s	2000s	30000s	2000s	8000s	2000s	8000s	2000s	12000s

Table 2. Comparsion of running time with SCIP and Gurobi under the same optimization solution results on different benchmark IPs.Ours-20%S means the scale-limited versions of SCIP which limit the variable proportion α to 20%. Ours-20%G means the scale-limited versions of Gurobi which limit the variable proportion α to 20%. >20000s indicates the inability to achieve the target objective function within the 20000s.

	IP_1	IP_2	CA_2	CA_3	MIS_2	MIS_3	MVC_2	MVC_3	SC_2	SC_3
SCIP	65.0s	3149.6s	>20000s	>60000s	>20000s	>60000s	>20000s	>60000s	>20000s	7332.0s
Ours-20%S	131.4s	3456.4s	>20000s	>60000s	>20000s	>60000s	>20000s	>60000s	>20000s	25010.2s
Ours-30%S	49.7s	**2152.2s**	1925.8s	29973.1s	969.8S	7594.7s	1045.3s	**7749.4s**	2345.0s	3942.2s
Ours-50%S	**44.2s**	3444.2s	**211.4s**	**26645.6s**	**81.2s**	**5983.4s**	**33.3s**	10485.0s	**250.8s**	**459.2s**
Target	232594.6	940877.9	13663.6	137110.9	22294.2	223342.0	27419.6	276258.6	17174.8	225681.4
Gurobi	67.1s	>6000s	>20000s	>60000s	>20000s	>60000s	>20000s	>60000s	15213.0s	>60000s
Ours-20%G	99.1s	>6000s	>20000s	>60000s	>20000s	28373.9s	>20000s	>60000s	10915.3s	26052.8s
Ours-30%G	50.2s	**1371.8s**	**2900.5s**	29360.4s	1121.8s	**5817.4s**	1183.0s	7905.7s	3814.1s	11362.7s
Ours-50%G	**39.1s**	1912.6s	>20000s	**28242.4s**	**389.9s**	6542.5s	**383.1s**	**4087.4s**	**1134.8s**	**2329.3s**
Target	232700.6	943328.7	13426.8	134685.9	22158.4	218109.7	27737.4	280458.9	17052.6	229642.5

carried out a comparative experiment between the proposed framework with different proportions α and the large-scale baseline solver SCIP and Gurobi in the same wall-clock time. The experimental results are shown in Table 1. This paper compares scale-limited versions of SCIP and Gurobi which limit the variable proportion α to 20%, 30% and 50% to compare with the large-scale version. Generally, when the small-scale optimizer limits the variable proportion $\alpha = 20\%$, the optimization result of the proposed framework has almost exceeded the large-scale version of the baseline in the same wall-clock time. Moreover, when the small-scale optimizer limits the variable proportion $\alpha = 30\%$, the proposed framework is far ultra the large-scale version of the baseline on all benchmark IPs.

Based on the above experimental results, the proposed framework can use a small-scale optimizer to achieve good results on large-scale IPs. In addition, through experimental comparison, this paper also found some interesting conclusions. On one hand, not all experiments get the best results when proportion $\alpha = 50\%$ which is speculated that the final result should be similar to a unimodal function with the change of the proportion α. On the other hand, even though SCIP is significantly inferior to Gurobi in solving large-scale IPs, SCIP as a small-scale solver in the proposed framework can surpass Gurobi in most problems, which is speculated that SCIP has better optimization ability in small-scale problems that closes to the original intention of using small-scale solvers to solve large-scale IPs in the proposed framework.

4.2. Comparison of Running Time

In order to further explore the optimization solution capability of the proposed framework, this paper also compares the running time with the baseline solvers SCIP and Gurobi under the same optimization solution results, and the results are summarized in Table 2. This paper compares scale-limited versions of SCIP and Gurobi which limit the variable proportion α to 20%, 30% and 50% to compare with the large-scale version of the baseline. As shown in the table, under the same optimization results, the proposed framework still significantly reduces the running time compared with the baseline solver in all IPs.

It is worth noting that in the MIS_2 and MVC_2, although the optimization results of the proposed framework and the baseline solver in Table 1 are not much different, the time spent by the SCIP is more than 200 times that of the proposed framework while the time spent by the Gurobi is more than 50 times that of the proposed framework under the same optimization results.

4.3. Convergence Performance Analysis

Convergence is a very important attribute in the optimization solution framework. It can often be measured by using the figure of objective value progressions of variants. In order to deeply analyze the convergence performance of the proposed framework in solving large-scale IPs, this paper compares the statistics of convergence in the process of solving large-scale optimization problems with SCIP under different variable proportion α in IPs which is in Figure 6. It can be seen from the figure that the proposed framework combined with the small-scale optimizer has good convergence performance when solving large-scale IP, on the premise of obtaining high-quality solutions.

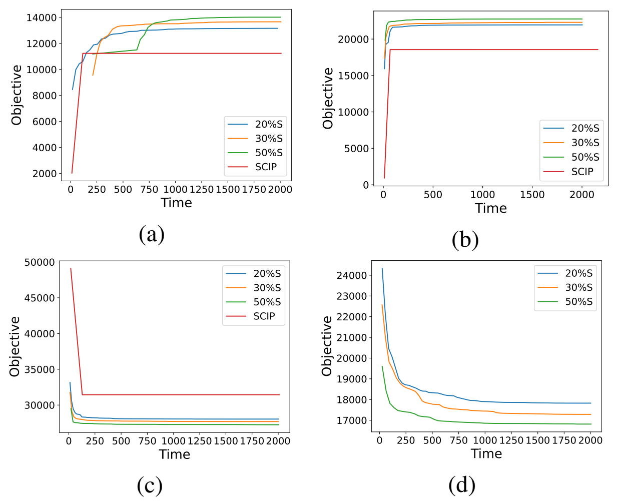

Figure 6. Time-objective figure for benchmark IPs. (a) is CAT. (b) is MIS. (c) is MVC. (d) is SC where no feasible solution is found in SCIP.

5. Conclusion

This paper proposes a GNN&GBDT-based fast optimizing framework for solving large-scale IPs. By using multi-task GNN with half-convolutions layers and the GBDT, our framework can effectively use the embedding spatial information and generate intermediate information for the optimization stage. On top of it, by using a neighborhood search with a fixed search radius and small-scale neighborhood crossover, our framework can just use a small-scale optimizer to solve large-scale IPs efficiently. Experiments show that our framework can solve large-scale IPs and surpass SCIP and Gurobi in the specified wall-clock time using only a small-scale optimizer with 30% of the problem size, and can only use one percent of the time to achieve the same solution quality as SCIP. However, the proposed framework is currently tailored for efficient solving of IPs. In the future, we will continue optimizing the proposed framework and try to make breakthroughs in the aspects of ultra-large-scale, multi-objective and mixed integer.

6. Acknowledgements

This research was supported by Meituan.

References

Achterberg, T. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

Achterberg, T. What's new in gurobi 9.0. *Webinar Talk url: https://www.gurobi.com/wp-content/uploads/2019/12/Gurobi-90-Overview-Webinar-Slides-1.pdf*, 2019.

Achterberg, T. and Berthold, T. Hybrid branching. In *Proceedings of International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 309–311, 2009.

Applegate, D., Bixby, R., Chvátal, V., and Cook, W. *Finding cuts in the TSP (A preliminary report)*. Citeseer, 1995.

Balas, E. and Martin, C. H. Pivot and complement–a heuristic for 0-1 programming. *Management science*, 26(1): 86–96, 1980.

Bénichou, M., Gauthier, J.-M., Girodet, P., Hentges, G., Ribière, G., and Vincent, O. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

Caprara, A., Toth, P., and Fischetti, M. Algorithms for the set covering problem. *Annals of Operations Research*, 98 (1-4):353–371, 2000.

Chen, Z., Liu, J., Wang, X., Lu, J., and Yin, W. On representing mixed-integer linear programs by graph neural networks. *arXiv preprint arXiv:2210.10759*, 2022.

De Vries, S. and Vohra, R. V. Combinatorial auctions: A survey. *INFORMS Journal on computing*, 15(3):284–309, 2003.

Dinur, I. and Safra, S. On the hardness of approximating minimum vertex cover. *Annals of mathematics*, pp. 439–485, 2005.

Du, D. and Pardalos, P. M. *Handbook of combinatorial optimization*. Springer Science & Business Media, 1998.

Fischetti, M., Glover, F., and Lodi, A. The feasibility pump. *Mathematical Programming*, 104:91–104, 2005.

Friedman, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pp. 1189–1232, 2001.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

Hillier, F. S. Efficient heuristic procedures for integer linear programming with an interior. *Operations Research*, 17 (4):600–637, 1969.

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.

Jian, T., Yang, F., Zuo, Z., Wang, W., Momma, M., Zhao, T., Dong, C., Gao, Y., and Sun, Y. Multi-task gnn for substitute identification. In *Proceedings of Companion Proceedings of the Web Conference 2022*, pp. 228–231, 2022.

Kim, S., Kim, D., Cho, M., and Kwak, S. Proxy anchor loss for deep metric learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3238–3247, 2020.

Kulis, B. et al. Metric learning: A survey. *Foundations and Trends® in Machine Learning*, 5(4):287–364, 2013.

Land, A. and Doig, A. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pp. 497–520, 1960.

man Jr, E. C., Garey, M., and Johnson, D. Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems*, pp. 46–93, 1996.

Martin, R. K. *Large scale linear and integer optimization: a unified approach*. Springer Science & Business Media, 2012.

Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

Paulus, A., Rolinek, M., Musil, V., Amos, B., and Martius, G. Fit the right np-hard problem: End-to-end learning of integer programming constraints. In *Proceedings of Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

Pisinger, D. and Ropke, S. Large neighborhood search. In *Handbook of metaheuristics*, pp. 399–419. 2010.

Rothberg, E. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing*, 19(4):534–541, 2007.

Schrijver, A. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

Shaw, P. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of International conference on principles and practice of constraint programming*, pp. 417–431, 1998.

Song, J., lanka, r., Yue, Y., and Dilkina, B. A general large neighborhood search framework for solving integer linear programs. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Proceedings of Advances in Neural Information Processing Systems*, pp. 20012–20023, 2020a.

Song, J., Lanka, R., Yue, Y., and Ono, M. Co-training for policy learning. In *Proceedings of Uncertainty in Artificial Intelligence*, pp. 1191–1201, 2020b.

Sonnerat, N., Wang, P., Ktena, I., Bartunov, S., and Nair, V. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

Tarjan, R. E. and Trojanowski, A. E. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

Tsourakakis, C., Gkantsidis, C., Radunovic, B., and Vojnovic, M. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pp. 333–342, 2014.

Williams, H. P. Integer programming. In *Logic and Integer Programming*, pp. 25–70. 2009.

Ye, H., Wang, H., Xu, H., Wang, C., and Jiang, Y. Adaptive constraint partition based optimization framework for large-scale integer linear programming (student abstract). *arXiv preprint arXiv:2211.11564*, 2022.

Yoon, T. Confidence threshold neural diving. *arXiv preprint arXiv:2202.07506*, 2022.

Zhang, J., Liu, C., Li, X., Zhen, H.-L., Yuan, M., Li, Y., and Yan, J. A survey for solving mixed integer programming via machine learning. *Neurocomputing*, 519:205–217, 2023.

A. Calculation Process Of GBDT

For a given data set containing n examples and m features $\mathcal{D} = \{x_i, y_i\}$ where $|D| = n$, $x_i \in \mathcal{R}^m$ and $y_i \in \mathcal{R}$, GBDT tries to use K regression trees to fit the data set. The final prediction result of GBDT is the weighted sum of K regression trees-based model prediction.

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \tag{18}$$

where $f_k(x_i)$ denots the prediction of the k-th regression tree. To fit the data set, GBDT minimizes the following objective function.

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(\hat{y_i}, y_i), \tag{19}$$

where l is an arbitrary metric function to represent the distance between the prediction of GBDT $\hat{y_i}$ and the target y_i. GBDT uses the additive manner to minimize fumula (5). Formally, let $\hat{y_i}^{(t-1)}$ be the sum of predictions of $t-1$ regression trees for the i-th instance, the objective of the t-th regression tree can be written as the following.

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)). \tag{20}$$

Applying second-order Taylor expansion approximation to formula (20), it can obtain the following.

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_i(x_i) + \frac{1}{2} h_i f_t^2(x_i)], \tag{21}$$

where $g_i = \partial_{\hat{y}^(t-1)} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2_{\hat{y}^(t-1)} l(y_i, \hat{y}^{(t-1)})$ are first-order and second-order gradient statistics of function l.

It can remove the constant term to obtain the following simplification objective of the t-th regression tree.

$$\overline{L}^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)]. \tag{22}$$

Defining $I_j = \{i | q(x_i) = j\}$ to be an instance in leaf j, it can rewrite formula (22) as the following.

$$\overline{L}^{(t)} = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} ((\sum_{i \in I_j} h_i) w_j^2]. \tag{23}$$

Therefore, corresponding to the fixed regression tree structure, we can calculate the optimal weight w_j^* of leaf j.

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i}, \tag{24}$$

and can calculate the optimal value of the objective function.

$$\overline{L}^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i}. \tag{25}$$

For the structure of the regression tree, GBDT greedily adds branches from the root. Assuming I_L and I_R are the instance sets of left and right nodes after branching, let $I = I_L \cup I_R$, the objective of the branching is to maximize the loss reduction after branching.

$$L_{split} = \frac{1}{2} [\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i} + \frac{(\sum_{i \in I_E} g_i)^2}{\sum_{i \in I_R} h_i} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i}]. \tag{26}$$

B. FENNEL Graph Partition Algorithm

For million-level graphs, the streaming graph partitioning algorithm will be used generally. There are two types of heuristics for streaming graph partitioning. On one hand, place the newly arrived vertex in the cluster with the largest number of neighbors. On the other hand, place the newly arrived vertex in the cluster with the least number of non-neighbors. FENNEL then proposed a new solution to combine the two heuristics.

Formally, for the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n vertexes and m edges to be partitioned into k blocks, FENNEL will summarize the vertexes with dense edges into one block, while dividing the vertexes with sparse edges into separate blocks. Specifically. it checks each newly arrived verte v in the graph one by one and calculates $\delta_g(v, \mathcal{P}_i)$ with each block P_i which satisfies $|P_i| < \nu \frac{n}{k}$ as the following.

$$\delta_g(v, \mathcal{P}_i) \leftarrow |\mathcal{P}_i \cap N(v)| - \alpha\gamma|P_i|^{\gamma-1}, \tag{27}$$

where $N(v)$ denotes the neighbor node set of v, ν, γ are preset parameters related to block balancing and minimum cut, and $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$ denotes the balance of two types of heuristics. The details are shown in Algorithm 4.

Algorithm 4 FENNEL-based Graph Partition Algorithm

```
**Input:** The number of blocks $k$, graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, parameters $\mu, \gamma$
**Init:** Label $\mathcal{F} = \{\}$, block $\mathcal{P} = \{\{\}, \dots, \{\}\}$
$n \leftarrow |\mathcal{V}|$
$m \leftarrow |\mathcal{E}|$
$\alpha \leftarrow \sqrt{k}\frac{m}{n^{3/2}}$
load_limit$\leftarrow \nu\frac{n}{k}$
**for** $v = 1$ **to** $n$ **do**
  **for** $i = 1$ **to** $k$ **do**
    **if** $|P_i| <$ load_limit **then**
      $N(v) \leftarrow \{u, (u, v) \in \mathcal{E}\}$
      $\delta_g(v, \mathcal{P}_i) \leftarrow |\mathcal{P}_i \cap N(v)| - \alpha\gamma|P_i|^{\gamma-1}$
    **end if**
  **end for**
  ind $\leftarrow$ argmax$_i \delta_g(v, \mathcal{P}_i)$
  Add $v$ into $\mathcal{P}_i$
  $\mathcal{F}[v] \leftarrow$ ind
**end for**
**Return:** $\mathcal{F}$
```

C. REPAIR Algorithm

The search with a fixed radius for IP is widely used in the proposed framework to improve the current solution, which can be written as the following.

$$\begin{aligned} &\min_{x \notin \mathcal{F}} c^T x \\ &\text{subject to } Ax \le b, l \le x \le u, x \in \mathbb{Z}^n, \\ &x_i = \hat{x}_i, \forall x_i \in \mathcal{F}, \end{aligned} \tag{28}$$

where $\hat{x_i}$ denotes the value in the current solution of the i-th decision variable and $\mathcal{F}$ denotes the set of decision variables that are fixed as the value in the current solution. However, the IP corresponding to formula (28) may be infeasible, leading to the current solution's failure. Therefore, a REPAIR Algorithm is proposed to check and repair the constraints that are doomed to be infeasible by canceling the fixation of some illegal variables corresponding to the infeasible constraints.

Specifically, for a given IP and a set of fixed variables $\mathcal{F}$, the REPAIR algorithm sequentially traverses every constraint in the IP. For a constraint, the algorithm will calculate whether the constraint is doomed to be infeasible through the upper and lower bounds of unfixed variables. If this constraint is doomed to be infeasible, the algorithm will cancel the fixation of some decision variables in the constraint and try to make this constraint feasible again. The details are shown in Algorithm 5.

Algorithm 5 REPAIR Algorithm

```
Input: The set of fixed variables $\mathcal{F}$, the set of unfixed variables $\mathcal{U}$, the current solution $\mathcal{X}$
$\{A, b, l, u\} \leftarrow$ The coefficient of the given IP
$n \leftarrow$ the number of decision variables
$m \leftarrow$ the number of constrains
for $i = 1$ to $m$ do
  $\mathcal{N} \leftarrow 0$
  for $j = 1$ to $n$ do
    if The $j$-th decision variable $\in \mathcal{F}$ then
      $\mathcal{N} \leftarrow \mathcal{N} + \mathcal{X}_j * A_{i,j}$
    else
      if $A_{ij} > 0$ then
        $\mathcal{N} \leftarrow \mathcal{N} + l_j * A_{i,j}$
      end if
      if $A_{ij} < 0$ then
        $\mathcal{N} \leftarrow \mathcal{N} + u_j * A_{i,j}$
      end if
    end if
  end for
  if $\mathcal{N} > b_i$ then
    for $j = 1$ to $n$ do
      if The $j$-th decision variable $\in \mathcal{F}$ then
        Remove the $j$-th decision variable from $\mathcal{F}$
        Append the $j$-th decision variable into $\mathcal{U}$
        $\mathcal{N} \leftarrow \mathcal{N} - \mathcal{X}_j * A_{i,j}$
        if $A_{ij} > 0$ then
          $\mathcal{N} \leftarrow \mathcal{N} + l_j * A_{i,j}$
        end if
        if $A_{ij} < 0$ then
          $\mathcal{N} \leftarrow \mathcal{N} + u_j * A_{i,j}$
        end if
        if $\mathcal{N} \le b_i$ then
          BREAK
        end if
      end if
    end for
  end if
end for
Return: $\mathcal{F}, \mathcal{U}$
```

D. Experiments Details

D.1. Experimental Setting

All experiments are run on a machine with two Intel Xeon Platinum 8375C @ 2.90GHz CPU and four NVIDIA TESLA V100(32G) GPU. The decision variables and constraint scale of the one real-world large-scale IP in the internet domain and four widely used NP-hard benchmark IPs are shown in Table 3.

In addition, on all IPs, we set that the running time of a round of the search with fixed radius cannot exceed 20% of the total time. In particular, for the CA problem, due to the huge search space of the initial solution, it is easy to exceed the memory limit of the experimental machine. In the experiments with proportion $\alpha = 50\%$, our initial solution is still carried out using proportion $\alpha = 30\%$ of the weakened version. Theoretically, machines with larger memory will get better results.

Table 3. The size of one real-world large-scale IP in the internet domain and four widely used NP-hard benchmark IPs. IP denotes the real-world large-scale IP. CA denotes the Combinatorial Auction problem. MIS denotes the Maximum Independent Set problem. MVC denotes the Minimum Vertex Covering problem. SC denots the Set Covering problem.

Problem	**Scale**	**Number of Variables**	**Number of Constraints**
Read-world IP (Maximize)	IP_1	500000	25000
	IP_2	1920000	100003
CA (Maximize)	CA_1	10000	10000
	CA_2	100000	100000
	CA_3	1000000	1000000
MIS (Maximize)	MIS_1	10000	30000
	MIS_2	100000	300000
	MIS_3	1000000	3000000
MVC (Minimize)	MVC_1	10000	30000
	MVC_2	100000	300000
	MVC_3	1000000	3000000
SC (Maximize)	SC_1	20000	20000
	SC_2	200000	200000
	SC_3	2000000	2000000

D.2. Baseline

In this paper, the state-of-the-art IP solvers SCIP and Gurobi(10.0) are used as the baseline, and their scale-constrained versions are used as the small-scale optimizer for neighborhood optimization. For the latest two-stage optimization framework based on graph neural network (GNN) and large neighborhood search (LNS) in https://github.com/deepmind/neural_lns, its code is not fully open source. So we did not include it in the baseline in this paper.

D.3. Dataset

For the four widely used NP-hard benchmark IPs, the existing data set cannot meet such large-scale data requirements, so we use data generators to generate training and test data sets. Specifically, for the Maximum Independent Set problem (MIS) or Minimum Vertex Covering problem (MVC) with n decision variables and m constraints, we generate a random graph with n nodes and m edges to correspond to an IP problem that meets the scale requirements. For the Combinatorial Auction problem (CA) with n decision variables and m constraints, we generate a random problem with n items and m bids where each bid includes 5 items. For the Set Covering problem (SC) with n decision variables and m constraints, we generate a random problem with n items and m sets where each set bid includes 4 items. For the optimal solution in the training data set, we use Gurobi to run for 8 hours to find the approximate optimal solution.

E. Additional Experiments

In order to verify the ability of the proposed framework to solve small-scale IPs, this paper has carried out an additional comparative experiment between the proposed framework with different proportions α and the large-scale baseline solver SCIP and Gurobi in the same wall-clock time. The experimental results are shown in Table 4. To further explore the optimization solution capability of the proposed framework in small-scale IPs, this paper also compares the running time with the baseline solvers SCIP and Gurobi under the same optimization solution results, and the results are summarized in Table 5.

The experimental results show that the proposed framework can still surpass the baseline in small-scale problems and has good generalization.

Table 4. Comparison of objective value results with SCIP and Gurobi under the same running time on small-scale benchmark IPs. Ours-20%S means the scale-limited versions of SCIP which limit the variable proportion α to 20%. Ours-20%G means the scale-limited versions of Gurobi which limit the variable proportion α to 20%. $\uparrow$ means the result is better than the baseline. - means that no feasible solution is found.

	CA_1	MIS_1	MVC_1	SC_1
SCIP	1107.4	1857.3	3125.7	-
Ours-20%S	1283.3$\uparrow$	2180.5$\uparrow$	2833.6$\uparrow$	1800.9$\uparrow$
Ours-30%S	1336.5$\uparrow$	2208.7$\uparrow$	2792.2$\uparrow$	1771.7$\uparrow$
Ours-50%S	**1367.8**$\uparrow$	**2264.6**$\uparrow$	**2722.1**$\uparrow$	**1680.8**$\uparrow$
Gurobi	1286.3	2196.3	2792.6	1818.2
Ours-20%G	1177.7	2188.2	2827.8	1787.5$\uparrow$
Ours-30%G	1274.7	2218.4$\uparrow$	2759.5$\uparrow$	1713.2$\uparrow$
Ours-50%G	**1302.8**$\uparrow$	**2266.7**$\uparrow$	**2714.0**$\uparrow$	**1669.0**$\uparrow$
Time	60s	40s	30s	60s

Table 5. Comparsion of running time with SCIP and Gurobi under the same optimization solution results on small-scale benchmark IPs. Ours-20%S means the scale-limited versions of SCIP which limit the variable proportion α to 20%. Ours-20%G means the scale-limited versions of Gurobi which limit the variable proportion α to 20%.

	CA_1	MIS_1	MVC_1	SC_1
SCIP	>600s	>400s	>300s	600.0s
Ours-20%S	>600s	>400s	>300s	72.9s
Ours-30%S	57.9s	38.7s	26.7s	59.7s
Ours-50%S	**25.3s**	**4.5s**	**5.5s**	**18.5s**
Target	1336.5	2208.7	2792.2	1771.7
Gurobi	45.0s	61.0s	66.0s	73.0s
Ours-20%G	165.8s	>400s	>300s	>600s
Ours-30%G	59.24s	37.7s	30.0s	57.3s
Ours-50%G	**39.9s**	**5.7s**	**5.5s**	**22.5s**
Target	1274.7	2218.4	2759.5	1713.2