Learning to Branch in Combinatorial Optimization With Graph Pointer Networks

Rui Wang, *Senior Member, IEEE*, Zhiming Zhou, Kaiwen Li, Tao Zhang, Ling Wang, Xin Xu, *Senior Member, IEEE*, and Xiangke Liao

***Abstract*—Traditional expert-designed branching rules in branch-and-bound (B&B) are static, often failing to adapt to diverse and evolving problem instances. Crafting these rules is labor-intensive, and may not scale well with complex problems. Given the frequent need to solve varied combinatorial optimization problems, leveraging statistical learning to auto-tune B&B algorithms for specific problem classes becomes attractive. This paper proposes a graph pointer network model to learn the branch rules. Graph features, global features and historical features are designated to represent the solver state. The graph neural network processes graph features, while the pointer mechanism assimilates the global and historical features to finally determine the variable on which to branch. The model is trained to imitate the expert strong branching rule by a tailored top-k Kullback-Leibler divergence loss function. Experiments on a series of benchmark problems demonstrate that the proposed approach significantly outperforms the widely used expert-designed branching rules. It also outperforms state-of-the-art machine-learning-based branch-and-bound methods in terms of solving speed and search tree size on all the test instances. In addition, the model can generalize to unseen instances and scale to larger instances.**

***Index Terms*—Branch-and-bound (B&B), combinatorial optimization, deep learning, graph neural network, imitation learning.**

Manuscript received June 14, 2023; revised August 31, 2023; accepted November 7, 2023. This work was supported by the Open Project of Xiangjiang Laboratory (22XJ02003), Scientific Project of the National University of Defense Technology (NUDT) (ZK21-07, 23-ZZCX-JDZ-28), the National Science Fund for Outstanding Young Scholars (62122093), and the National Natural Science Foundation of China (72071205). Recommended by Associate Editor Weiping Ding. *(Corresponding authors: Kaiwen Li and Tao Zhang.)*

Citation: R. Wang, Z. Zhou, K. Li, T. Zhang, L. Wang, X. Xu, and X. Liao, "Learning to branch in combinatorial optimization with graph pointer networks," *IEEE/CAA J. Autom. Sinica*, vol. 11, no. 1, pp. 157–169, Jan. 2024.

R. Wang is with the Xiangjiang Laboratory and the College of Systems Engineering, National University of Defense Technology, Changsha 410073, China (e-mail: rui_wang@nudt.edu.cn).

Z. Zhou is with the Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zhiming.zhou@ia.ac.cn).

K. Li and T. Zhang are with the College of Systems Engineering, National University of Defense Technology, Changsha 410073, and also with the Hunan Key Laboratory of Multi-Energy System Intelligent Interconnection Technology, Changsha 410073, China (e-mail: likaiwen@nudt.edu.cn; zhangtao@nudt.edu.cn).

L. Wang is with the Department of Automation, Tsinghua University, Beijing 100084, China (e-mail: wangling@mail.tsinghua.edu.cn).

X. Xu is with the College of Intelligence Science and Technology, National University of Defense Technology, Changsha 410073, China (e-mail: xinxu@nudt.edu.cn).

X. Liao is with the College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China (e-mail: xkliao@nudt.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JAS.2023.124113

I. Introduction

COMBINATORIAL optimization seeks to explore discrete decision spaces, and to find the optimal solution in an acceptable execution time. Combinatorial optimization problems arise in diverse real-world domains such as manufacturing, telecommunications, transportation and various types of planning problem [1], [2]. These kinds of problems can be immensely difficult to solve, since it is computationally impractical to find the best combination of discrete variables through exhaustive enumeration. In fact, most of the NP-hard problems in mathematical and operational research fields are typical examples of combinatorial optimization, such as the traveling salesman problem (TSP), maximum independent set [3], graph coloring [4], Boolean satisfiability [4], etc.

A large number of approaches have been proposed to tackle combinatorial optimization challenges these years [5]. Exact algorithms can always find the optimal solution to a combinatorial optimization problem [6]. A naive way is searching all possible solutions through enumeration, which results in an intractable solving time. Some advanced techniques have been proposed, such as branch-and-bound [7], to efficiently prune the searching space. Approximation algorithms are used when the optimal solution cannot be found in polynomial time [8]. They guarantee a solution within a certain ratio of the optimal one. Examples include algorithms for problems like vertex-cover or set-cover. However, such algorithms may not exist for all real-world combinatorial optimization problems. Heuristic algorithms are designed to find a good (if not optimal) solution quickly, and are useful when the problem is too large or complex for exact methods. Greedy algorithms, local search, and hill climbing are examples of heuristic methods [9]. Metaheuristic algorithms provide a higher-level, general strategy that can be applied to many different types of combinatorial optimization problems. These are often inspired by natural processes and include genetic algorithms, simulated annealing, ant colony optimization, particle swarm optimization, and more. Metaheuristic algorithms cannot guarantee finding the optimal solution, however, require less computing time than exact algorithms [10].

As exact algorithms can always solve an problem to optimality, modern optimization solvers generally employ exact algorithms to solve combinatorial optimization problems, which can be formulated as mixed-integer linear programs (MILPs). The branch-and-bound (B&B) method is a typical exact method that solves MILPs in a divide-and-conquer manner. B&B [7] recursively splits the search space of the problem into smaller regions in a tree structure, where each node represents the subproblem that searches subsets of the solution set. Subtrees can be pruned once it provably cannot produce better solutions than the current best solution; otherwise, the subtree is further partitioned into subproblems until an integral solution is found or the subproblem is infeasible. In this solving process, there are several decision-making problems that should be considered to improve performance: node selection problem, i.e., which node/subproblem should be selected to process next given a set of leaf nodes in the search tree? Variable selection problem (a.k.a. branching), i.e., which variable should branch on to partition the current node/subproblem?

Historically, decisions related to MILP instances have relied on meticulously crafted heuristics. The creation of these hard-coded expert heuristics often requires considerable design efforts and extensive trial-and-error. However, the advent of artificial intelligence has sparked increased interest in using machine learning models to learn these heuristics, rather than relying on expert design. This shift is logical, as heuristics often consist of a series of rules, which could feasibly be parameterized by models such as deep neural networks.

Recent investigations have indeed begun to explore these learning-based approaches [11]−[14]. Yet, these pioneering efforts still pose significant challenges: identifying effective features to represent the current state of the B&B process on which the branching decision is based; and developing effective models that can map the B&B state to the branching decision. These challenges underline the need for continued research and development in this promising area.

This paper presents an innovative approach to addressing the branching problem in B&B using a graph pointer network model, trained to imitate the effective yet computationally intensive "expert" strong branch heuristic. We aim to produce strategies that result in a small number of search nodes, approaching the performance of the strong branch heuristic, while maintaining a low computation cost. Though this idea is not new [13]−[15], we improve the performance of the learning model in a novel way. The contributions of this work are summarized as follows:

1) Traditional B&B methods, which depend on manually-designed branching heuristics, often struggle with adaptability and efficiency across diverse problem scenarios. In contrast, we propose using neural networks to automatically learn these branching heuristics.

2) In addition the typical graph features that are commonly explored, global and historical features are designed in this work, providing a more comprehensive and richer representation of the problem state.

3) We introduce an innovative model that combines the graph neural network with a pointer mechanism. The graph neural network processes the graph features, while the pointer mechanism assimilates the global and historical features to finally determine the variable on which to branch.

4) Our research presents a top-k Kullback-Leibler divergence loss function, specifically designed to train the model to imitate the strong branch heuristic effectively.

5) Notably, our proposed method consistently surpasses both expert-crafted branching rules and contemporary machine learning techniques across all tested problems. Once trained, our model demonstrates remarkable generalization abilities, effortlessly adapting to even unseen, larger instances.

The remainder of the paper is organized as follows. Section II reviews the recent advances of applying artificial intelligence methods for combinatorial optimization. Section III introduces the preliminaries of the work. The proposed graph pointer network model is described in Section IV. Section V outlines the imitation learning method for optimizing the model parameters. The experiment setup and numerical results are presented in Section VI-A. The last section gives the concluding remarks and future perspectives.

II. Related Work

Recent days have seen a surge of applying artificial intelligence methods for combinatorial optimization.

Vinyals *et al.* [16] developed a pointer network model for solving small scale combinatorial optimization problems like the traveling salesman problems (TSPs). It borrowed the idea of the widely used sequence-to-sequence model in the machine translation field, and used the attention mechanism to map the input sequence to the output sequence. This work inspired a number of subsequent research that involved using machine/deep learning methods for combinatorial optimization.

Most the current works focus on solving the combinatorial optimization problems in an end-to-end manner. Bello *et al.* [17] first proposed the use of a deep reinforcement learning (DRL) method to optimize the pointer network model, which can output the solution sequence directly. Nazari *et al.* [18] investigated the vehicle routing problem (VRP) by modifying the pointer network and the attention mechanism. Dai *et al.* [19] developed a structure2vec graph neural network (GNN) model for combinatorial optimization. The GNN model can encode the graph feature of the problem and aid the decisions. Other works [20]−[23] explored advanced GNN models like the graph convolution networks (GCNs) and diverse training methods to solve the combinatorial optimization problems more effectively. Moreover, authors in [24], [25] improved the attention mechanism of the pointer network by leveraging the recent advances of the famous Transformer model [26] in the field of seqence-to-seqence learning. The attention model developed by Kool *et al.* [25] achieved state-of-the-art performance among the above approaches. This model can solve a number of combinatorial optimization problems, such as the TSP, VRP, the orienteering problem, etc. In addition, Li *et al.* [27] extended this line of work to a multiobjective version.

The initial efforts in using statistical learning for devising branching rules in B&B were spearheaded by Khalil *et al.* [12]. They devised a branching rule tailored for an individual instance during the B&B procedure. This approach was also explored by Alvarez *et al.* [28] and Hansknecht *et al.* [29], where they focused on learning a branching rule offline from a set of related instances. In all these studies, the aim was to learn a branching strategy by emulating the expertise of strong branching. Specifically, both Khalil *et al.* [12] and Hansknecht *et al.* [29] approached it as a ranking challenge, aimed at learning a hierarchical order of the candidates by the expert. On the other hand, Alvarez *et al.* [28] approached this as a regression task, intending to directly ascertain the strong branching scores for each candidate. Gasse *et al.* [13] pioneered the application of GNN for branching policies in MIPs. Their approach leveraged the inherent variable-constraint bipartite graph representation, utilizing imitation learning to swiftly approximate strong branching. Extending upon this foundation, Gupta *et al.* [14] introduced a hybrid framework. In their system, the GNN model operates at the root node, while a computationally efficient (albeit weaker) model governs the subsequent nodes. Moreover, Nair *et al.* [30] presented two innovative models: neural diving and neural branching, to augment the capabilities of traditional MIP solvers. The neural diving model is adept at predicting partial assignments for integer variables, facilitating the generation of more compact MIPs. Conversely, neural branching harnesses a GNN model to approximate optimal branching policies.

In a broader context, numerous researchers have developed machine learning techniques to refine exact optimization methods, even outside the scope of general MILPs. A comprehensive review on this topic was recently presented by Blum and Roli [10].

III. Preliminaries

A. Problem Definition

1) Mixed Integer Linear Program: A combinatorial optimization problem can be always modeled as a MILP, having the form

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \mathbf{l} \le \mathbf{x} \le \mathbf{u} \\ & x_i \in \mathbb{Z}, \quad i \in \mathcal{I} \\ & \mathbf{c} \in \mathbb{R}^n,\ \mathbf{A} \in \mathbb{R}^{m\times n},\ \mathbf{b} \in \mathbb{R}^m,\ \mathbf{l},\mathbf{u} \in \mathbb{R}^n. \end{aligned} \tag{1}$$

The aim is to find an optimal set of $\mathbf{x}$ to minimize the objective function with $\mathbf{c}$ as the objective coefficient vector. There are m constraints and n decision variables. A subset of the decision variables are integers. $\mathcal{I} \subseteq \{1,\dots,n\}$ is their index set, and $\mathbb{Z}$ is their decision space. $\mathbf{A},\mathbf{b}$ are the coefficient matrix and the right-hand-side vector of the constraints. $\mathbf{l},\mathbf{u}$ bound the decision variables.

2) LP Relaxation of a MILP: Given the complexity of MILP problems, where some variables are required to be integer values, they can be quite challenging and computationally expensive to solve exactly. LP relaxation simplifies these problems by ignoring the integer constraints and treating all variables as continuous. This results in a standard linear programming problem, which is significantly easier and faster to solve using well-established algorithms such as the Simplex method. This LP relaxation serves as a lower bound (in minimization problems) for the original MILP problem. This is because relaxing the integer constraints allows for a broader feasible region and hence, potentially, a smaller minimum value.

3) Branch-and-Bound: Branch-and-bound begins by solving the LP relaxation of the original MILP. The obtained solution $\mathbf{x}^*$ provides the lower bound to the problem. If the obtained solution respects all the MILP integrality constraints, it is the optimal solution to (1), and the algorithm terminates. If not, the LP relaxation is further partitioned into two subproblems by branching on an integer variable that does not respect integrality of the MILP. This is done by adding the following two constraints into the LP relaxation, respectively [13]:

$$x_i \le \lfloor x_i^\star \rfloor, x_i \ge \lceil x_i^\star \rceil, \quad \exists i \in \mathcal{I} \mid x_i^\star \notin \mathbb{Z} \tag{2}$$

where $\lfloor x_i^\star \rfloor$ refers to the maximum integer value that is smaller than $x_i^\star$, and $\lceil x_i^\star \rceil$ is the minimum integer value that is larger than $x_i^\star$. Here, i is called the branching variable.

By branching on i, two new LPs are constructed, which refer to the leaf nodes/subproblems of the search tree. The next step is to pick one leaf node, and repeat the above steps. Once a feasible solution $\hat{x}$ is found, that is, all the MILP integrality constraints are satisfied, there is an upper bound to the problem. If a solution is found with a lower objective value than $\hat{x}$, the upper bound is updated. On the other hand, if a solution is found with worse objective value than the current upper bound, this subproblem is pruned and no longer branched. The subproblem is also fathomed if the solution is an integer or the LP is infeasible. The above procedures are repeated until no subproblems remains. The incumbent solution with the best bound is returned [13].

B. Branching Strategies

In the branching variable selection decision process, an integer variable i is selected among the candidate variables $C = \{i \mid x_i^\star \notin \mathbb{Z}, i \in \mathcal{I}\}$ that do not satisfy the integer constraint. Existing methods usually score each candidate variable in C according to some handcrafted heuristics, and the variable with the largest score is selected for branching. The most commonly used scoring criterion is the change of the lower bound of the sub-problem after the variable is branched [31]. Based on this criterion, a series of branch rules are designed to improve the efficiency of B&B.

Strong branching (SB) is an effective yet expensive scoring heuristic. Empirically, it has been found that it consistently produces the smallest B&B search tree compared to other heuristics [12]. The SB rule explicitly measures the upper and lower bounds changes of the sub-problem, so as to select the best branching variable, which is computed as follows. For the LP sub-problem corresponding to the current node N, its LP solution is $\mathbf{x}^*$, and its corresponding objective value is z^*. By branching on variable i, two LP sub-problems N_i^- and N_i^+ are obtained, and the corresponding objective values are z_i^{*-} and z_i^{*+}. If N_i^- and N_i^+ have no feasible solutions, then z_i^{*-} and z_i^{*+} are set to very large values. Therefore, the change of the objective function value after branching on variable i is

$\Delta_i^- = z_i^{*-} - z^*$ and $\Delta_i^+ = z_i^{*+} - z^*$. The SB score is calculated as [12]

$$SB_i = \text{score}\left(\max\left\{\Delta_i^-, \epsilon\right\}, \max\left\{\Delta_i^+, \epsilon\right\}\right) \tag{3}$$

where the product function is usually considered as the scoring function, that is, $\text{score}(a,b) = a \times b$. ϵ is a small constant. The SB rule computes the SB scores for all the candidate variables in the candidate set C, and selects the decision variable with the largest SB score to branch on. Each branching decision requires long computational time since computing each SB score requires solving two LP sub-problems. In this case, the SB-based B&B algorithm usually suffers heavy computational burden although SB can greatly reduce the search tree size.

In view of the heavy computational burden of the SB method, calculating the pseudocost instead of the SB score is another commonly used method in the current optimization solver. Pseudocost branching (PB) estimates the score of a variable according to its historical scores during the previous search process. Instead of solving the two sub-problems by branching on i, the upwards (downwards) score of variable i is the average value of the objective value changes when upwards (downwards) branching on variable i in the previous branching process. This can greatly shorten the calculation time. Denote the upwards and downwards average scores of variable i as Ψ_j^- and Ψ_j^+, where its pseudocost score PB_i is calculated as [12]

$$PB_i = \text{score}\left(\left(x_i^* - \lfloor x_i^* \rfloor\right)\Psi_i^-, \left(\lceil x_i^* \rceil - x_i^*\right)\Psi_i^+\right) \tag{4}$$

and where $x_i^* - \lfloor x_i^* \rfloor$ and $x_i^* - \lceil x_i^* \rceil$ represent the decimal part of the variable value. The PB method can effectively reduce the computing time of each branching decision. However, the search tree is much larger than that obtained by SB, since there is no sufficient historical data in the early stage of the search to estimate the variable score, which results in incorrect branching decisions. In view of the pros and cons of SB and PB, the reliability branching (RB) method applies SB at the beginning of the search until enough historical data is accumulated, and then applies PB in the subsequent process.

It can be seen that there is a contradiction between the branching performance and the time cost by making each branching decision. This work aims to leverage the power of neural networks to emulate the SB rule, striving for comparable performance but with significantly reduced computational expenditure.

IV. Model

A graph pointer network (GPN) model is proposed to simulate the SB strategy previously mentioned. This model's input is the current state of the solver, while the output corresponds to the decision on variable selection. First, the B&B procedure is structured as a Markov decision process. At each stage, the model interprets the current state and selects an appropriate variable, causing a corresponding change in the solver's state. Then, the state of the solver is defined to encompass the graph structure feature, global feature, and historical feature. Finally, a graph pointer neural network model is developed in accordance with the state definition, enabling the perception of the solver's current state and facilitating the execution of branching decisions.

A. Markov Decision Process Modeling

B&B can be modeled as a Markov decision process [13], as shown in Fig. 1.

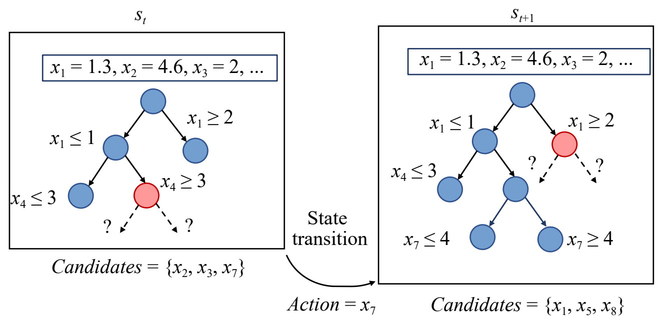


Fig. 1. Markov decision process of the branch-and-bound.

At each decision step t, the current state of the solver is $\mathbf{s}_t$, which represents the state of the current search tree. Based on the current state of the solver $\mathbf{s}_t$, the agent selects a variable $a_t = i$ from the candidate set $C = \{i \mid x_i^\star \notin \mathbb{Z}, i \in \mathcal{I}\}$ according to the strategy $\pi(a_t \mid \mathbf{s}_t)$.

The solver solves the two LP sub-problems after branching on variable i. Subsequently, the solver updates the upper and lower bounds, prunes the search tree, and selects the next leaf node to branch. At this time, the solver has been converted to a new state $\mathbf{s}_{t+1}$. Then, the solver applies the branch strategy $\pi(a_{t+1} \mid \mathbf{s}_{t+1})$ again to make the branching decision. This process is looped until all the leaf nodes are explored.

The initial state $\mathbf{s}_0$ of the Markov decision process corresponds to the root node of the B&B search tree. And the final state is the end of the optimization process, i.e., all leaf nodes cannot be branched further. Denote the branching policy as π, where the Markov decision process can be modeled as [13]

$$p_\pi(\mathbf{s}_0, \ldots, \mathbf{s}_T) = p(\mathbf{s}_0) \prod_{t=0}^{T-1} \sum_{a \in C(\mathbf{s}_t)} \pi(a \mid \mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, a). \tag{5}$$

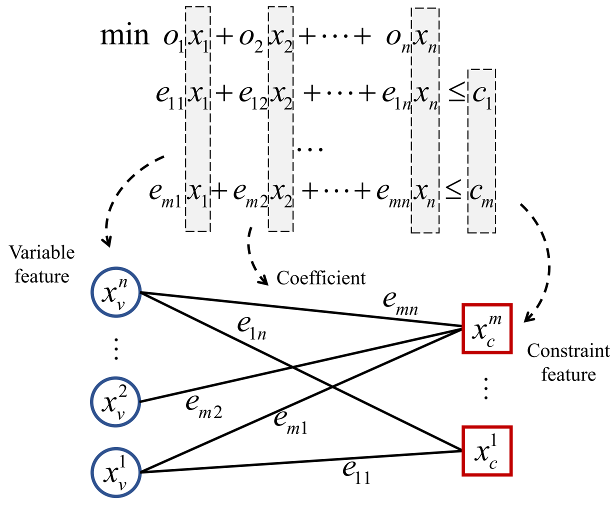
In this paper, we learn the branching policy π to imitate the SB rule, which is realized through the following steps: 1) Define the problem state $\mathbf{s}_t$. At each step of the branch decision, the branch decision needs to be made according to the current problem state. However, there is no standardized definition of the solver state. It is necessary to extract effective features to better represent the solver state, so as to make better decisions accordingly. 2) Parameterize the branch policy π via a novel model. The model should be able to map the problem state $\mathbf{s}_t$ to the branching action $\mathbf{a}_t$ correctly. The models, such as neural networks, random forests and support vector machines, need to be designed according to the characteristics of B&B. 3) Optimize the parameters of the model by an effective training algorithm. The model π can be learned through a variety of machine learning methods to minimize the size of the search tree or reduce the total run-time of the B&B algorithm.

The proposed deep learning-based B&B method consists of the above three parts introduced as follows.

B. State Definition

First, the state $\mathbf{s}_t$ of B&B at the decision-making step t is defined in this section. In addition to the graph features introduced in [13], the global features and historical features are designed, which can provide a more thorough representation of the solver state. Therefore, $\mathbf{s}_t$ is composed of variable features, constraint features, edge features, global features, and historical features, namely $\mathbf{s}_t = (V, C, E, G, H)$.

The graph features (V, C, E) of the problem is defined by the bipartite graph of the current solver state, as shown in Fig. 2. The bipartite graph is composed of m constraints and n variables. Variables $x_1, x_2, \ldots, x_n$ are on the left side of the graph. The right-hand side (constant) term of the constraint is on the right side of the graph. The edge $(i, j) \in E$ of the graph is the connection of the variable i and the constraint j, i.e., whether the constraint j includes the variable i. The weight of the edge is the coefficient of the variable i in constraint j.



$\min\ o_1x_1 + o_2x_2 + \cdots + o_nx_n$
$e_{11}x_1 + e_{12}x_2 + \cdots + e_{1n}x_n \le c_1$
$\cdots$
$e_{m1}x_1 + e_{m2}x_2 + \cdots + e_{mn}x_n \le c_m$
Variable feature
Coefficient
Constraint feature
x_v^n
x_v^2
x_v^1
x_c^m
x_c^1
e_{mn}
e_{1n}
e_{m2}
e_{m1}
e_{11}

Fig. 2. Graph structure of the MILP state.

According to the bipartite graph structure, the solver state is comprised of variable features, constraint features, edge features, global features, and historical features.

1) Variable features represent the attributes of candidate variables at branching step t, including the variable type, variable coefficient, current value of the variable, whether the current value of the variable is on the boundary, the decimal part of the solution value of the variable, etc. There are n candidate variables in total, and the feature dimension is d. Therefore, the variable feature dimension is $n \times d$. The detailed introduction of the variable features is listed Table I.

2) Constraint features represent the attributes of the LP constraint at branching step t, such as the right value of the constraint, whether the left value of the constraint exactly reaches the boundary, the similarity of the constraint coefficient and the target coefficient, etc. The current LP problem has a total of m constraints, and the feature dimension is c. Thus, the dimension of the constraint feature is $m \times c$, and the description of the constraint features can be found in Table II. Refer to [13] for a detailed description of variable and constraint

TABLE I
VARIABLE FEATURES

Feature	Numeric type
Variable type, 0: 0-1 binary; 1: Integer; and 2: Continuous	Categorical
Normalized variable coefficient in the objective function	Real
If the variable owns a upper/lower bound	Binary
If the current solution value of the variable is its upper/lower bound	Binary
Fractional part of the variable's current solution value	Real
0: The variable is at its lower bound; 1: Variable's value lies between the upper and lower bounds (basic); 2: The variable is at its upper bound; 3: Rare case	Categorical
Reduced cost, the variable's solution value can become positive if we reduce the objective coefficient of the variable by this value	Real
Number of LP iterations since the last time the variable was basic	Integer
Solution value of the variable at current node	Real
The variable's value of the best primal solution	Real
Average value of the variable in all the feasible solutions found so far	Real

TABLE II
CONSTRAINT FEATURES

Feature	Numeric type
Similarity between the left-hand-side coefficients of the constraint and the objective coefficients	Real
Normalized right-hand-side (constant) value of the constraint	Real
Number of iterations since the last time the constraint was active	Integer
Dual variable's value of the constraint	Real
If the constraint is at the bounds	Binary

features.

3) The edge feature is the coefficient of each variable in each constraint. Therefore, there are $m \times n$ edges in total, and the feature dimension is 1. The coefficient value is 0 if the constraint does not contain a certain variable.

4) Global feature G represents the global state of the solver, such as the current optimality gap of the problem, the gap between the objective value of the current node and the upper/lower bounds, the depth of the current search tree, the depth of the current node, etc. We design and extract the global features using the API interface of PySCIPOpt, which is an open source B&B solver. The detailed global features are listed in Table III.

G mainly includes two parts: a) Global features of the whole MILP, including the gap between the upper and lower bounds of the current stage of MILP, the number of feasible solutions/infeasible solutions, etc.; b) Global features of the current LP sub-problem node, including the depth of the current node, the LP objective value information of the current node, etc.

The depth of the current node and the gap between the upper and lower bounds can be directly obtained by calling the PySCIPOpt interface. The number of feasible/infeasible solutions is computed by the proportion of leaf nodes that produce feasible/infeasible solutions:

TABLE III
GLOBAL FEATURES

Feature	Numeric type
Depth of the current node	Integer
Normalized number of feasible solutions	Real
Normalized number of infeasible solutions	Real
Gap between the global upper and lower bounds	Real
Gap between the current node's LP objective value and the global upper bound	Real
Gap between the current node's LP objective value and the global lower bound	Real
Relative position of the current node's LP objective value to the global upper/lower bounds	Real
Gap between the current node's LP objective value and the root node's upper bound	Real
Gap between the current upper bound and the root node's upper bound	Real

$$P_{\text{feasible}} = \frac{N_{\text{feasible}}}{\max(N_{\text{leaves}}, 0.1)} \tag{6}$$

where N_{leaves} is the number of all leaves. The gap between the current node's LP objective v_{LP} value and the global upper/lower bounds v_b is calculated by the following formula according to [15]:

$$\text{gap} = \begin{cases} 0, & \text{if } xy < 0 \\ \dfrac{|v_{LP} - v_b|}{\max\{|v_{LP}|, |v_b|, 1\times 10^{-10}\}}, & \text{else} \end{cases} \tag{7}$$

where the current node's LP objective value and the global upper/bounds are obtained from the PySCIPOpt interface.

The relative position pos of the current node's LP objective value to the global upper bound v_u and lower bound v_l is computed as $\frac{|v_{LP}-v_u|}{|v_{LP}-v_u|}$.

5) The historical feature consists of two parts. The first part is comprised of features of all past branching decisions $C_1 = a_1 \cdots a_{t-1}$ at previous steps $1 \cdots t-1$. The second part is comprised of features of variables $C_2 = x_1, x_2, \ldots$ whose values have changed when generating the current node. That is, C_2 is the set of variables whose values have changed in the solution of the new problem after adding an integer constraint to the parent problem.

Traditional approaches only considers variable features, constraint features and edge features [13]. This work further extracts global features and historical features, so as to obtain a richer representation of the environment state $\mathbf{s}_t$. The global status of the current search tree and the current node can provide more information for the agent to make branching decisions. Moreover, observing the variables whose values have changed when generating the current node, and observing the variables selected during the historical branching process, can also provide effective information for making the branching decisions. Therefore, it is expected that adding additional global features and historical features can better describe the state of the current problem.

C. Graph Pointer Network Model

In this section, a GPN that combines the graph neural network and the pointer mechanism is proposed to model the branching policy, which can map the solver state to the branching decisions effectively.

From the features extracted in the previous section, it can be seen that the solver state has a bipartite graph structure, that is, the left nodes (variables) and the right nodes (constraints) are connected by edges, as shown in Fig. 2. Graph neural network can effectively process the information of the graph structure, and has been successfully applied to various machine learning tasks with graph structure input, such as social networks and citation networks. Therefore, we encode the graph structure of the solver state by a graph neural network model.

In addition, we take the global and historical features as a query, and compute the attention value, which is then normalized as a softmax probability distribution, as a pointer to the input sequence. In this way, the variable with the largest probability is selected as the branching variable.

The proposed graph pointer neural network model is composed of two parts: 1) The graph neural network calculates the feature vector for each variable based on variable features, constraint features and edge features; 2) The pointer mechanism outputs the variable selection probabilities by computing the attention values according to variables' feature vectors and the query which is constructed by the global and historical features. The detailed process of modeling the branching policy is as follows.

1) Initial Embedding Calculation: Variable features, constraint features, edge features, and global features have different dimensions. For example, the variable feature is 13-dimensional, and the global feature is 9-dimensional. Therefore, the d_h-dimensional embeddings of the variable features $\mathbf{x}_v$, constraint features $\mathbf{x}_c$, edge features $\mathbf{x}_e$ and global features $\mathbf{x}_g$ are computed as follows:

$$\begin{aligned} \mathbf{x}_v &\leftarrow \text{EMBEDDING}(\mathbf{x}_v) \\ \mathbf{x}_c &\leftarrow \text{EMBEDDING}(\mathbf{x}_c) \\ \mathbf{x}_e &\leftarrow \text{EMBEDDING}(\mathbf{x}_e) \\ \mathbf{x}_g &\leftarrow \text{EMBEDDING}(\mathbf{x}_g) \end{aligned} \tag{8}$$

where EMBEDDING$(\cdot)$ is a two-layer fully connected neural network. The hidden dimension is d_h and the activation function between layers is LeakyRELU

$$\text{LeakyRELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 10^{-2} \times x, & \text{otherwise.} \end{cases} \tag{9}$$

2) Graph Neural Network: Next, the final variable features are computed by a graph convolution neural network similar to [13]

$$\mathbf{x}_c^i \leftarrow \mathbf{f}_C\Big(\mathbf{x}_c^i, \sum_j^{(i,j)\in E} \mathbf{g}_C\big(\mathbf{x}_c^i, \mathbf{x}_v^j, \mathbf{x}_e^{i,j}\big)\Big)$$

$$\mathbf{x}_v^j \leftarrow \mathbf{f}_V\Big(\mathbf{x}_v^j, \sum_j^{(i,j)\in E} \mathbf{g}_V\big(\mathbf{x}_v^j, \mathbf{x}_c^i, \mathbf{x}_e^{i,j}\big)\Big). \tag{10}$$

Function $\mathbf{g}(\cdot)$ is defined as

$$g\big(\mathbf{x}_c^i, \mathbf{x}_v^j, \mathbf{x}_e^{i,j}\big) = \text{FF}\big(\mathbf{x}_c^i + \mathbf{x}_v^j + \mathbf{x}_e^{i,j}\big) \tag{11}$$

where FF is a two-layer fully connected neural network with LeakyRELU activation function. Function $\mathbf{f}(\cdot)$ is also a two-layer fully connected neural network with LeakyRELU activation function. As demonstrated in (10), the graph embedding is computed by two successive convolution passes, one from variables to constraints and the next from constraints to variables. The first convolution step computes the features $\mathbf{x}_c^i$ of constraint i according to features $\mathbf{x}_v^j$ of its connected variables j, features of the edge $\mathbf{x}_e^{i,j}$ and its own features. The second step computes the embedding $\mathbf{x}_v^j$ of variable j according to the obtained features $\mathbf{x}_c^i$ of its connected constraints i, features of the edge $\mathbf{x}_e^{i,j}$ and its own features. Through the graph convolution process, the final variable features aggregate the original variable features, constraint features and coefficient features of the problem, so as to effectively contain the graph information of the MILP state.

3) Historical Feature Calculation: At branching step t, the first part of the historical features is the past branching decisions $C_1 = a_1 \cdots a_{t-1}$ at steps $1 \cdots t-1$. We compute this part of d_h-dimensional historical features as

$$\mathbf{x}_{h1}^t = \text{FF}\Big(\frac{1}{t-1}\sum_{i=1}^{t-1}\mathbf{x}_v^{a_i}\Big) \tag{12}$$

where FF is a single-layer fully connected neural network layer, and a_i is the variable selected by the solver at step i.

The second part of the historical feature is the variable set C_2 whose value changes during the process of generating the current node. The same operation is performed on C_2 to obtain the d_h-dimensional vector $\mathbf{x}_{h2}^t$. In addition, $\mathbf{x}_{h2}^t$ and $\mathbf{x}_{h1}^t$ are zero vectors if $t == 0$.

4) Pointer Mechanism: The attention value, which can be seen as a pointer to the candidate variables, is computed by a compatibility function of the query with the key. The query, which is composed of global features and historical features, represents the current state of the solver. The key represents the feature of each candidate variable. Specifically, the query vector is calculated as the weighted average of global and historical features

$$\mathbf{q_t} = w_1 \times \mathbf{x}_g^t + w_2 \times \mathbf{x}_{h1}^t + w_3 \times \mathbf{x}_{h2}^t \tag{13}$$

where w_1, w_2, w_3 are weight values to be optimized while training. Moreover, the key of variable i is defined as $k_i = W_k\mathbf{x}_v^i$, $i \in C$, which is the linear projection of the variable features. Denote the query at branching step t as $\mathbf{q}_t$ and the keys of candidate variables as $\mathbf{k}_i$, $i \in C$, where one has

$$\begin{aligned} u_i^t &= W_3\left(W_1\mathbf{k}_i + W_2\mathbf{q}_t\right), \quad & i \in (1,\ldots,n) \\ p_i^t &= \text{softmax}\left(u_i^t\right), & i \in (1,\ldots,n) \end{aligned} \tag{14}$$

where u_i^t is the attention value computed by the compatibility function. Note that other compatibility function can also be applied to compute the attention, which can refer to [26] for more details. softmax is used to normalize the attention value to the probability distribution p_i^t, representing the probability of selecting variable i at branching step t. In this case, we can choose the variable with the highest probability p_i^t as the branching variable.

In addition, it is necessary to normalize the variable features, constraint features, edge features, and global features due to their different data range. To this end, the prenorm layer is applied as introduced in [13] to normalize the variable, constraint, and edge features. We also add a prenorm layer of global features accordingly, so that the neural network model can deal with problem instances with global features of different scales.

D. Branch and Bound Algorithm Based on GPN

The proposed GPN model is then used to select the branching variable in B&B. The GPN-based B&B is illustrated in Algorithm 1.

Algorithm 1 Branch and Bound Algorithm Based on GPN

```
Input: Root node R, representing the LP relaxation of the original MILP
Output: Optimal solution $S^*$
1: $R.lowerBound \leftarrow -\infty$   /*Initialize the lower bound of $R$*/
2: $Queue \leftarrow \{R\}$   /*Store the unexplored node into the Queue*/
3: $UpperBound \leftarrow \infty$   /*Initialize the global upper bound*/
4: $S^* \leftarrow null$
5: while $Queue$ is not empty do
6:     $N \leftarrow Queue.get()$   /*Dequeue the node*/
7:     if $N.lowerBound \geq UpperBound$ then
8:         /*If node $N$'s parent node's lower bound is greater than the global upper bound, prune this node*/
9:         continue
10:    end if
11:    $S_r \leftarrow$ solve($N$)
12:    if $S_r$ is not feasible then
13:        /*prune this node*/
14:        continue
15:    end if
16:    $O_r \leftarrow S_r.objectiveValue$
17:    if $O_r > UpperBound$ then
18:        /*If node $N$'s lower bound is greater than the global upper bound, prune this node*/
19:        continue
20:    end if
21:    if $S_r$ is feasible then
22:        $UpperBound \leftarrow O_r$
23:        $S^* \leftarrow S_r$
24:        /*Update the global upper bound and $S^*$*/
25:        continue
26:    end if
27:    Extract features of the solver state, $state = (V, C, E, G, H)$
28:    $V \leftarrow$ GPN($S_r, state$)   /*Select varibale $V$ by the GPN model*/
29:    $a \leftarrow$ floor($V.value$)
30:    $L \leftarrow$ addConstraint($N, V \leq a$)
31:    $R \leftarrow$ addConstraint($N, V \geq a$)
32:    /*Branch on $V$ and obtain the two LP sub-porblems*/
33:    $L.lowerBound \leftarrow O_r$, $R.lowerBound \leftarrow O_r$
34:    $Queue.add(L)$, $Queue.add(R)$
35: end while
36: return $S^*$
```

First, the LP relaxation of the original MILP problem is set as the root node. The queue data structure is maintained to store the sub-problem nodes to be solved. Each node defines an initial lower bound l, which represents the lower bound of its parent node. After the global upper bound is updated, if l is greater than the global upper bound, then the node will be pruned. When the node is taken out of the queue, its lower bound is compared with the global upper bound, and the node is pruned if the lower bound is greater than the global upper bound. The global upper bound is initialized to ∞, and is updated every time a better feasible solution is obtained. Variable, constraint, edge, global and historical features of candidate variables are extracted, which are subsequently input to the GPN model. The model outputs the probability distribution of the candidate variables. The one with the highest probability can be selected as the variable to branch on. Two subproblems are generated accordingly. This process loops until the queue is empty, i.e., all leaves of the search tree are explored.

With n variables and m constraints, the computational complexity of the GPN model is $O(mnd_h)$ where d_h is the dimension of hidden layers. When applying the GPN model for branching, a single forward pass through the GPN model can yield the result. Therefore, it costs much less running time than the classical SB rule, which requires solving a number of subproblems.

V. Training Method

An imitation learning method is proposed to train the proposed model. The objective is to imitate the strong branching rule. Imitation learning [32] can solve various multi-step decision-making problems. In comparison with unsupervised reinforcement learning methods, imitation learning can improve the training efficiency with the help of expert experiences. Imitation learning requires labeled training data provided by human experts $\{\chi_1,\chi_2,\ldots,\chi_m\}$, where $\chi_i=\langle \mathbf{s}_1^i,a_1^i,\mathbf{s}_2^i,a_2^i,\ldots\rangle$. $\mathbf{s}_1^i$, a_1^i represents the "state-action" pairs in a Markov decision process generated by solving an instance using the SB-based B&B. Therefore, the labeled training set can be constructed as $\mathcal{D}=\{(\mathbf{s}_1,a_1),(\mathbf{s}_2,a_2),(\mathbf{s}_3,a_3),\ldots\}$. Denote a_i as the label, the variable selection problem can be converted into a classification problem. The objective is to minimize the difference between the expert actions and the predicted actions.

Specifically, the SB-based B&B is conducted on randomly generated combinatorial optimization instances. The "state-action" pairs are recorded to form a training set $\mathcal{D}=\{(\mathbf{s}_i,\mathbf{a}_i^\star)\}_{i=1}^N$. Denote the expert actions as $\mathbf{a}^\star$ and the predicted actions as $\tau_\theta(\mathbf{s})$, where the model parameters θ are optimized by minimizing

$$\mathcal{L}(\theta)=\frac{1}{N}\sum_{(\mathbf{s},\mathbf{a}^*)\in\mathcal{D}}\text{loss}\left(\tau_\theta(\mathbf{s}),\mathbf{a}^\star\right) \tag{15}$$

where loss($*$) is a function that defines the difference between the true value and the predicted value. For classification problems, there are a number of loss($*$) functions such as the accuracy and cross entropy.

However, in B&B, SB scores of different variables might be the same or pretty close. It is equivalent to select these variables. But only one variable is selected when constructing the labeled dataset. By applying loss functions like cross entropy, the similarities of SB scores between different variables cannot be leveraged. In this case, we choose to imitate the distribution of the SB scores instead of the branching actions. To this end, the SB scores of all the candidate variables are recorded to construct the training set $\mathcal{D}=\{(\mathbf{s}_i,score_i^\star)\}_{i=1}^N$. And Kullback-Leibler (KL) divergence is used as a measure of the difference between the SB score distribution and the predicted probability distribution. By minimizing the KL divergence, the model can work better for the above situation where multiple variables own the same or similar SB scores. It can better help the model imitate the SB scores.

Denote P as the true distribution of the data and Q as the predicted distribution of the model to fit P, KL divergence is defined as

$$D_{\text{KL}}(P\|Q)=\sum_{x\in\mathcal{X}}P(x)\log\left(\frac{P(x)}{Q(x)}\right). \tag{16}$$

Therefore, given the probabilities $\pi_\theta(\mathbf{s})$ of the candidate variables output by the model, the model parameters θ are optimized by minimizing

$$\mathcal{L}(\theta)=D_{\text{KL}}(score^\star\|\pi_\theta(\mathbf{s}))=\sum_{(\mathbf{s},score^\star)\in\mathcal{D}}score^\star\log\left(\frac{score^\star}{\pi_\theta(\mathbf{s})}\right). \tag{17}$$

In addition, we only care about the variables with high SB scores. The probability distribution of other variables has no effect on the branching variable selection. Thereby, we emphasize the similarity loss of variables with high SB scores in the training phase. Specifically, the variables are sorted according to their probabilities output by the model. More attention should be paid to the first few variables. To this end, the KL divergence of the top-k variables is added to the loss item. In specific, the probabilities $\pi_\theta(\mathbf{s})$ output by the model are sorted, and the first k variables $\mathcal{I}_k$ are selected. The KL divergence value of variables $\mathcal{I}_k$ is computed by (16) as $D_{\text{KL}}(score_{\mathcal{I}_k}^\star\|\pi_\theta(\mathbf{s})_{\mathcal{I}_k})$. And the loss for training the model is defined as

$$\mathcal{L}(\theta)=D_{\text{KL}}(score^\star\|\pi_\theta(\mathbf{s}))+D_{\text{KL}}(score_{\mathcal{I}_k}^\star\|\pi_\theta(\mathbf{s})_{\mathcal{I}_k}). \tag{18}$$

The first term of the loss can make the overall predicted distribution similar to the distribution of the SB scores, while the second term makes the model pay more attention to the variables of large probabilities and weakens the distribution of irrelevant variables for selecting the branching variables. This can alleviate the situation where a large amount of training time is cost to fit the distribution of irrelevant variables.

VI. Experimental Results and Discussion

A. Experiment Settings

1) Comparison Algorithm: The proposed approach is compared against the following approaches.

a) First, the proposed approach is compared against the classic B&B algorithm. The branching rule of reliability branching (RB), strong branching (SB) and pseudocost branching

(PB) are compared respectively. They are all implemented in the well-known SCIP solver. The cutting plane is only allowed at the root node. Other heuristics are disabled during the branching process for fair comparison. Our method is also implemented in the SCIP solver, and uses the same set of parameters as the competitor methods.

b) Next, the proposed approach is compared with the state-of-the-art machine learning-based B&B algorithms, often regarded as standard comparisons in the literature. The algorithms under consideration are: branching method based on ExtraTrees [33] model [28] (TREES); branching method [12] (SVMRANK) and [29] (LMART) based on SVMrank [34] and LambdaMART [35] model; branching method based on graph neural network [13] (GNN).

2) Test Problems: Effectiveness of the proposed method is evaluated on the following three benchmark combinatorial optimization problems.

a) Set covering problem [36]: The set covering instances contain 1000 columns. The model is trained on instances with 500 rows, and is evaluated on instances with 500 and 1000 rows, respectively.

b) Capacitated facility location problem [37]: The instances are generated with 100 facilities. The model is trained on instances with 100 customers, and is evaluated on instances with 100 and 200 customers, respectively.

c) Maximum independent set problem [38]: The instances are generated following the process in [38]. The model is trained on instances of 500 nodes, and is evaluated on instances with 500 and 1000 nodes, respectively.

3) Experimental Parameter Settings: All compared algorithms are implemented by Python on the SCIP solver. SCIP uses its default parameters. The hidden dimensions of the models are set to $d_h = 64$. The Adam optimizer is used for training with learning rate of 0.001. $k = 10$ is set for the top-k imitation learning. The learning rate decreases 80% if the loss does not decrease for 10 epochs. The training is terminated if the loss does not decrease for 20 epochs.

4) Training Data Generation: The SCIP solver with default settings is used to collect training samples offline. Random instances are generated and solved using the SCIP. During the collecting procedure, the branching rule of RB is adopted with a probability of 95%, and the branching rule of SB is adopted with a probability of 5%. Only the samples generated by SB are collected. The data of variable, constraint, edge, global and historical features, candidate variable sets, and SB scores of the variables is collected.

Instances are randomly generated and solved until 140 000 samples are collected. 100 000 samples are used as the training set, 2000 samples are used as the validation set, and 2000 samples are used as the test set.

5) Evaluation: First, the capability of the GPN model in imitating the SB rule is examined. Since multiple variables may have the same or similar SB scores, the following indices are used to evaluate the model accuracy [13]: a) The percentage of times the output of the model is exactly the variable with the highest SB score (acc@1); b) The percentage of times the output of the model is one of the five variables with the highest SB scores (acc@5); c) The percentage of times the output of the model is one of the ten variables with the highest SB scores (acc@10). Moreover, the total solving time of the GPN-based B&B in comparison with benchmark methods is evaluated.

B. Results

Figs. 3−5 present the training performances of the proposed GPN model in comparison to the classic GNN model across three test problems. The convergence of the loss and model accuracy on the validation set are both compared.

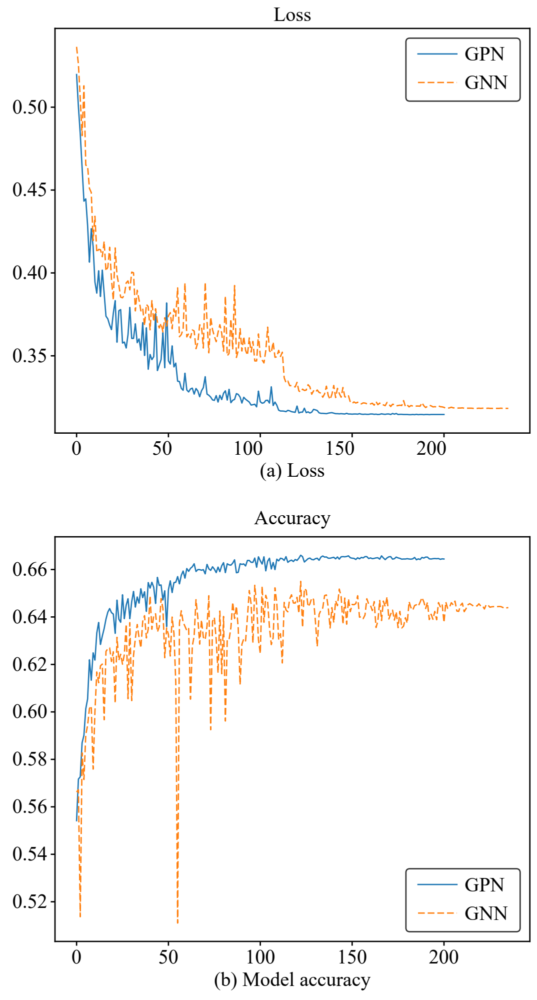


Fig. 3. Training performances of the models on the set covering.

Results indicate that the proposed GPN model surpasses the conventional GNN in both convergence speed and overall convergence performance during training. Additionally, GPN consistently outshines GNN in model accuracy for all three problems on the validation set. The superiority of GPN is particularly evident on the location problem and the maximum independent set problem; here, GPN converges to values of 0.66 and 0.003, respectively, while GNN reaches only 0.72 and 0.0047.

The superior convergence performance of the GPN over the GNN model stems from its advanced modeling capabilities and informative features that represent the problem state. By

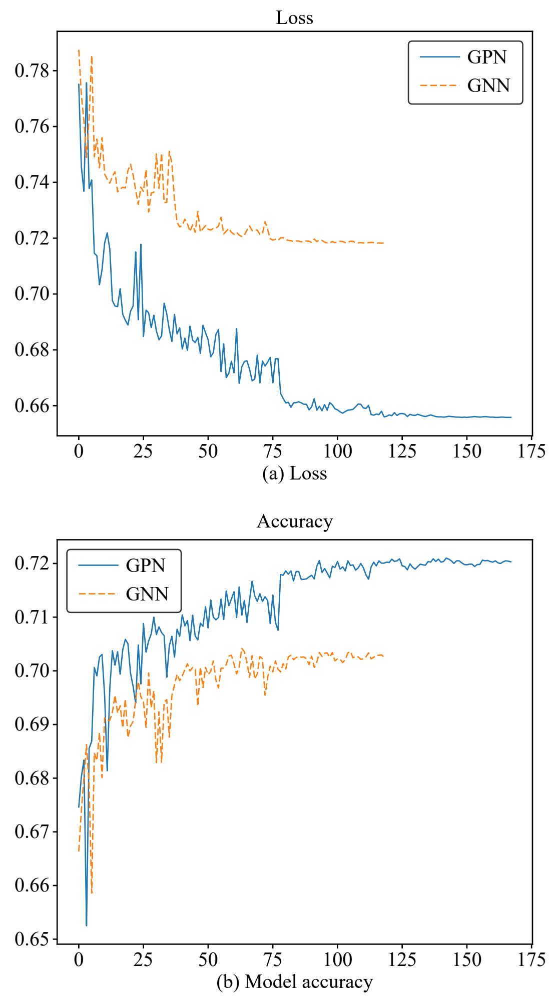


Fig. 4. Training performances of the models on the capacitated facility location.

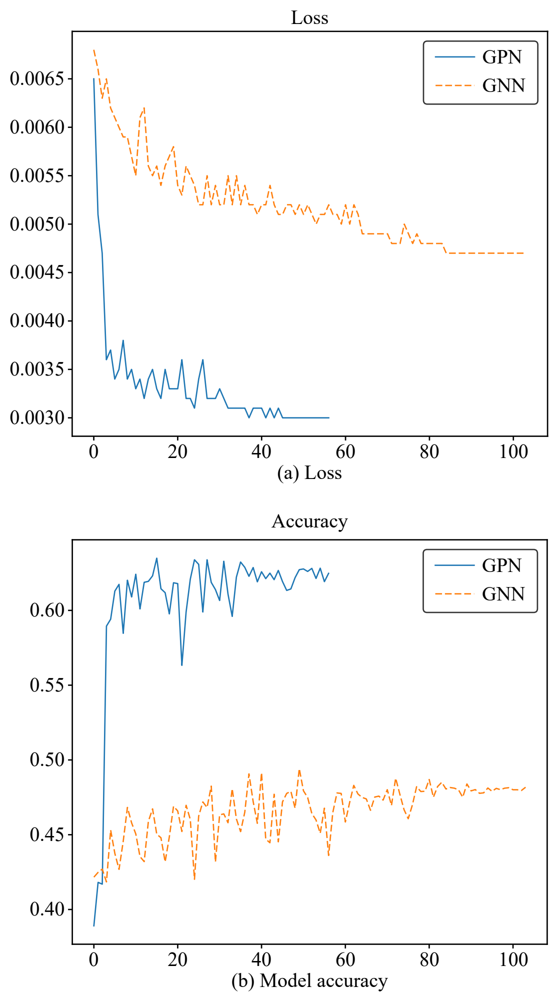


Fig. 5. Training performances of the models on the maximum independent set.

incorporating an attention-based pointer mechanism, the GPN model comprehends the graph, global and historical characteristics of the problem more effectively, thereby facilitating more accurate decision-making. Therefore, when compared to the GNN model, which solely employs a graph convolution network to process constraint and variable features, the GPN model exhibits a more expedient convergence rate and lesser validation loss. This highlights the advantage of the GPN model in fostering computational efficiency while maintaining high-quality performance.

Tables IV−VI present the model accuracy of GPN, TREES, SVMRANK, LMART, and GNN methods on the test set. Results of TREES, SVMRANK and LMART are from [13]. Results of acc@1, acc@5 and acc@10 are listed respectively.

Results outlined in Tables IV−VI indicate that for the set covering, capacitated facility location, and maximum independent set problems respectively, the GPN method consistently outperforms other approaches. This superiority is evident across all metrics: acc@1, acc@5, and acc@10. This empirical evidence solidifies the GPN's position as the leading method in these scenarios. Its performance is distinctly superior in the maximum independent set problem. This observation aligns seamlessly with the performance of the model during the training and validation process, indicating a consistent model behaviour. This empirical evidence affirms our premise

TABLE IV
RESULTS OF MODEL ACCURACY ON SET COVERING

	acc@1	acc@5	acc@10
TREES [28]	51.8	80.5	91.4
SVMRANK [12]	57.6	84.7	94
LMART [29]	57.4	84.5	93.8
GNN [13]	65.5	92.4	**98.2**
GPN	**66.5**	**92.7**	**98.2**

TABLE V
RESULTS OF MODEL ACCURACY ON CAPACITATED FACILITY LOCATION

	acc@1	acc@5	acc@10
TREES [28]	63	97.3	**99.9**
SVMRANK [12]	67.8	98.1	**99.9**
LMART [29]	68	98	**99.9**
GNN [13]	71.2	98.6	**99.9**
GPN	**72.2**	**98.7**	**99.9**

TABLE VI
RESULTS OF MODEL ACCURACY ON MAXIMUM INDEPENDENT SET

	acc@1	acc@5	acc@10
TREES [28]	51.8	80.5	91.4
SVMRANK [12]	57.6	84.7	94
LMART [29]	57.4	84.5	93.8
GNN [13]	65.5	92.4	**98.2**
GPN	**66.5**	**92.7**	**98.2**

that the GPN method is adept at imitating the SB rule, surpassing the alternative methods examined.

In addition, the running time of the approaches is evaluated, since the aim of the branching models is to reduce the overall solving time of B&B. The solving time is determined by the size of the search tree, that is, the number of explored nodes. It is also determined by the time consumed making the branch decisions. Therefore, a ood branching model can reduce the size of the search tree while making fast branching decisions.

Tables VII–IX list the results of solving time and the number of explored nodes when using GPN and the compared approaches for solving the three test problems. The "TimeB" index represents the duration required to make a single branching decision. Results are obtained by solving 100 randomly generated problems and taking the average.

TABLE VII
RESULTS OF RUNNING TIME ON SET COVERING

	500 rows			1000 rows		
Methods	Time	Nodes	TimeB	Time	Nodes	TimeB
SB	7.02	**12.5**	0.562	173.9	**227.5**	0.764
PB	2.88	98.6	0.029	19.8	2211.2	0.009
RB	3.73	18.8	0.198	22.1	1192.5	0.019
GNN [13]	2.07	43.9	0.047	14.2	900.6	0.016
GPN	**2.04**	42.3	0.048	**13.9**	891.2	0.016

Table VII shows that, in comparison with the PB and RB rule, the proposed GPN method achieves at least 40% increase in the solution speed when solving the 500-row and 1000-row set covering instances. In terms of the number of explored nodes, GPN outperforms all of the compared methods except for the SB rule on the set cover instances. SB can always get the smallest search tree. But its total solving time has no advantage over our method due to its long computation time of making branching decisions. It is obvious that the GPN method outperforms all the compared machine learning methods in terms of solving speed and the ability to reduce the search tree on the set covering instances.

It can be seen from Table VIII that the GPN method shows greater advantages in solving the 100-customer and 200-customer capacitated facility location instances compared to other methods. In specific, GPN runs twice as fast as the PB and RB method. Compared with the machine learning methods, GPN has the fastest solving speed and the fewest number of nodes.

On maximum independent set instances, GPN achieves nearly 10% improvement in the solution speed and 20% reduction in the number of nodes as seen in Table IX. The solving time is reduced nearly twice when using the GPN compared with the PB and RB methods.

TABLE VIII
RESULTS OF RUNNING TIME ON CAPACITATED FACILITY LOCATION

	100 customers			200 customers		
Methods	Time	Nodes	TimeB	Time	Nodes	TimeB
SB	157.4	**116.5**	1.351	1163.3	**158.7**	7.330
PB	82.8	541.9	0.153	510.7	614.2	0.831
RB	96.7	264.7	0.365	598.9	303.5	1.973
GNN [13]	37.4	467.4	0.080	145.6	529.6	0.275
GPN	**35.2**	428.9	0.082	**140.3**	516.2	0.272

TABLE IX
RESULTS OF RUNNING TIME ON MAXIMUM INDEPENDENT SET

	500 nodes			1000 nodes		
Methods	Time	Nodes	TimeB	Time	Nodes	TimeB
SB	87.1	**35.41**	2.460	2844.4	**164.5**	17.291
PB	14.6	1937.8	0.008	2002.9	17 213	0.116
RB	11.4	92.7	0.123	210.2	6717	0.031
GNN [13]	5.01	61.7	0.081	222.5	14 862	0.015
GPN	**4.63**	45.3	0.102	**198.6**	12 587	0.016

Upon analysis of the results, it is evident that the proposed GPN method is capable of delivering branching performance commensurate with the SB rule, but with markedly less computational time. As a result, the solving speed of the B&B method is significantly enhanced. This illustrates the efficiency of the GPN approach.

Note that, the test instances are generated randomly, and are different from the training set. Once the model is trained, it can be generalized to unseen instances, and scale to larger instances. Although the RB heuristic is carefully handcrafted by experts, it is still defeated by the proposed GPN method, which can learn heuristics from the data. Experiments validate the novelty and efficiency of the GPN method.

The goal of B&B is to solve the combinatorial optimization problem as fast as possible, so the branch strategy should be a trade-off between the quality of the decision and the time spent on each decision. An extreme example is the SB branch rule: by calculating the SB score for variable selection, the final solution can be obtained with a small number of searches, but each decision step is very time-consuming, so that the overall running time is very long. From the results of the 500-row set covering problem in Table VII, the SB rule takes 0.562 s in average to make a single branching decision, while our GPN method takes just 0.048 s. Despite GPN processing more nodes (42.3) than SB (12.5), its total solving time is shorter. Similarly, PB, although faster in decision-making than GPN, processes more nodes (98.6) leading to a longer overall solving time. Comparatively, while SB has the best branching decisions, it's the slowest. PB is the quickest but lacks in efficiency. GPN presents an optimal balance of performance and speed, offering the most effective solution time for B&B. This observation remains consistent across all other instances, validating the fact that the proposed method can achieve a better balance between the decision quality and decision time, therefore reducing the overall solving time of B&B.

VII. Conclusion

This paper has presented a novel method of modeling the variable selection strategy in B&B using a deep neural network model. We made use of graph features and introduced global and historical features to represent the solver state. The architecture combines a graph neural network with a pointer mechanism, enabling effective variable selection. Our experimental results on benchmark problems show that our approach surpasses traditional expert-designed branching rules and also outperforms state-of-the-art machine-learning-based B&B methods.

Looking ahead, there are several promising avenues for extending and refining our work.

1) The process of constructing labeled datasets for imitation learning can be computationally intensive. To mitigate this, a potential solution would be to integrate reinforcement learning techniques. By using reinforcement learning, we could harness unsupervised training strategies that could alleviate the need for labor-intensive labeled datasets, paving the way for more scalable and efficient learning.

2) While our model has demonstrated its effectiveness on benchmark problems, it is essential to test it on a wider spectrum of practical problems. Exploring its applicability in diverse domains will provide insights into the generalizability and robustness of our approach.

3) The current model leverages graph, global, and historical features. Future work can explore the inclusion of other types of features, potentially capturing more nuanced aspects of the problem space, leading to even more informed decisions during the branching process.

4) One exciting direction would be the introduction of mechanisms allowing the neural network model to adapt in real-time based on feedback during the B&B process. This could lead to models that fine-tune their strategies on-the-fly, adjusting to the unique characteristics of the problem being solved.

5) As deep learning models can be computationally intensive, especially in real-time scenarios, future work can also explore optimizations tailored to specific hardware platforms, ensuring efficient execution and reduced computational overhead.

In conclusion, the present work lays a foundation upon which numerous exciting advancements can be built. We believe that the interplay of optimization and deep learning has much more to offer, and we are enthusiastic about the potential breakthroughs the future may bring.

References

[1] P. Festa, "A brief introduction to exact, approximation, and heuristic algorithms for solving hard combinatorial optimization problems," in *Proc. 16th Int. Conf. Transparent Optical Networks*, Graz, Austria, 2014, pp. 1–20.

[2] A. Schrijver, "On the history of combinatorial optimization (till 1960)," *Handb. Oper. Res. Manage. Sci.*, vol. 12, pp. 1–68, 2005.

[3] K. Abe, I. Sato, and M. Sugiyama, "Solving NP-hard problems on graphs by reinforcement learning without domain knowledge," arXiv preprint arXiv: 1905.11623, 2020

[4] E. Yolcu and B. Póczos, "Learning local search heuristics for Boolean satisfiability," in *Proc. 33rd Int. Conf. Neural Information Processing Systems*, Vancouver, Canada, 2019, pp. 7992–8003.

[5] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'Horizon," *Eur. J. Oper. Res.*, vol. 290, no. 2, pp. 405–421, Apr. 2021.

[6] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optim.*, vol. 19, pp. 79–102, Feb. 2016.

[7] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," in *50 Years of Integer Programming 1958–2008*, M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, Eds. Berlin, Germany: Springer, 2010, pp. 105–132.

[8] N. Vesselinova, R. Steinert, D. F. Perez-Ramirez, and M. Boman, "Learning combinatorial optimization on graphs: A survey with applications to networking," *IEEE Access*, vol. 8, pp. 120388–120416, Jun. 2020.

[9] E. A. Silver, "An overview of heuristic solution methods," *J. Oper. Res. Soc.*, vol. 55, no. 9, pp. 936–956, May 2004.

[10] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sept. 2003.

[11] H. He, H. Daume III, and J. Eisner, "Learning to search in branch and bound algorithms," in *Proc. 27th Int. Conf. Neural Information Processing Systems*, Montreal, Canada, 2014, pp. 3293–3301.

[12] E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Proc. 30th AAAI Conf. Artificial Intelligence*, Phoenix, Arizona, 2016, pp. 724–731.

[13] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," in *Proc. 33rd Int. Conf. Neural Information Processing Systems*, Vancouver, Canada, 2019, p. 1396.

[14] P. Gupta, M. Gasse, E. B. Khalil, M. P. Kumar, A. Lodi, and Y. Bengio, "Hybrid models for learning to branch," in *Proc. 34th Int. Conf. Neural Information Processing Systems*, Vancouver, Canada, 2020, p. 1518.

[15] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio, "Parameterizing branch-and-bound search trees to learn branching policies," in *Proc. AAAI Conf. Artificial Intelligence*, 2021, pp. 3931–3939.

[16] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. 28th Int. Conf. Neural Information Processing Systems*, Montreal, Canada, 2015, pp. 2692–2700.

[17] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proc. 5th Int. Conf. Learning Representations*, Toulon, France, 2016.

[18] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takáč, "Deep reinforcement learning for solving the vehicle routing problem," arXiv preprint arXiv: 1802.04240, 2018.

[19] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. 31st Int. Conf. Neural Information Processing Systems*, Long Beach, USA, 2017, pp. 6351–6361.

[20] A. Mittal, A. Dhawan, S. Manchanda, S. Medya, S. Ranu, and A. Singh, "Learning heuristics over large graphs via deep reinforcement learning," arXiv preprint arXiv: 1903.03332, 2019.

[21] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A note on learning algorithms for quadratic assignment with graph neural networks," arXiv preprint arXiv: 1706.07450, 2017.

[22] C. K. Joshi, T. Laurent, and X. Bresson, "An efficient graph convolutional network technique for the travelling salesman problem," arXiv preprint arXiv: 1906.01227, 2019.

[23] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Proc. 32nd Int. Conf. Neural Information Processing Systems*, Montréal, Canada, 2018, pp. 537–546.

[24] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the TSP by policy gradient," in *Proc. 15th Int. Conf. Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Delft, The Netherlands, 2018, pp. 170–181.

[25] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems," in *Proc. 7th Int. Conf. Learning Representations*, New Orleans, USA, 2019.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. 31st Int. Conf. Neural Information Processing Systems*, Long Beach, USA, 2017, pp. 5998–6008.

[27] K. Li, T. Zhang, and R. Wang, "Deep reinforcement learning for multiobjective optimization," *IEEE Trans. Cyber.*, vol. 51, no. 6, pp. 3103–3114, Jun. 2021.

[28] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS J. Comput.*, vol. 29, no. 1, pp. 185–195, Jan. 2017.

[29] C. Hansknecht, I. Joormann, and S. Stiller, "Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem," arXiv preprint arXiv: 1805.01415, 2018.

[30] V. Nair, S. Bartunov, F. Gimeno, I. Von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. J. Li, O. Vinyals, and Y. Zwols, "Solving mixed integer programs using neural networks," arXiv preprint arXiv: 2012.13349, 2020.

[31] E. Mitchell, "Branch-and-cut algorithms for combinatorial optimization problems," *Handbook of Applied Optimization*, vol. 1, no. 1, pp. 65–77, 2002.

[32] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Comput. Surv.*, vol. 50, no. 2, p. 21, Mar. 2018.

[33] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Mar. 2006.

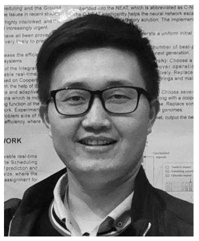
[34] T. Joachims, "Optimizing search engines using clickthrough data," in *Proc. 8th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, Edmonton, Canada, 2002, pp. 133–142.

[35] C. J. C. Burges, "From RankNet to LambdaRank to LambdaMART: An overview," Microsoft Corp., Seattle, USA, Microsoft Research Technical Report MSR-TR-2010-82, 2010.

[36] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study," *Combinatorial Optimization I*, M. W. Padberg, Ed. Berlin, Germany: Springer, 1980, pp. 37–60.

[37] G. Cornuejols, R. Sridharan, and J. M. Thizy, "A comparison of heuristics and relaxations for the capacitated plant location problem," *Eur. J. Oper. Res.*, vol. 50, no. 3, pp. 280–297, Feb. 1991.

[38] D. Bergman, A. A. Cire, W.-J. Van Hoeve, and J. Hooker, *Decision Diagrams for Optimization*. Cham, Germany: Springer, 2016.



Rui Wang (Senior Member, IEEE) received the B.S. degree in system engineering from the National University of Defense Technology (NUDT) in 2008, and the Ph.D. degree in system engineering from the University of Sheffield, UK in 2013. Currently, he is with the National University of Defense Technology. His current research interests include evolutionary computation, multiobjective optimization and the development of algorithms applicable in practice.

Dr. Wang has authored more than 40 referred papers including those published in *IEEE Transactions on Evolutionary Computation*, *IEEE Transactions on Cybernetics*, and *Information Sciences*. He serves as an Associate Editor of the *IEEE Transacions on Evolutionary Computation*, *Swarm and Evolutionary Computation*, *Expert System with Applications*, etc. He is the recipients of The Operational Research Society Ph.D. Prize in 2014, of the Funds for Distinguished Young Scientists from the Natural Science Foundation of Hunan province at 2016, of the Wu Wen-Jun Artificial Intelligence Outstanding Young Scholar at 2017, of the National Science Fund for Outstanding Young Scholars at 2021.

Zhiming Zhou received the B.S. and Ph.D. degrees in control science and engineering from Beijing Institute of Technology in 2015 and 2021, respectively. He is a Assistant Research Fellow with the Institute of Automation, Chinese Academy of Sciences. His research interests include reinforcement learning, optimal control and flight control and decisions making.

Kaiwen Li received the B.S., M.S. and Ph.D. degrees in management science and engineering from National University of Defense Technology (NUDT), in 2016, 2018 and 2022, respectively. He is a Lecturer with the College of Systems Engineering, NUDT. His research interests include prediction technique, multiobjective optimization, reinforcement learning, data mining, and optimization methods on energy Internet.

Tao Zhang received the B.S., M.S, and Ph.D. degrees in management science and engineering from National University of Defense Technology (NUDT) in 1998, 2001 and 2004, respectively. He is a Professor with the College of Systems Engineering, NUDT. His research interests include multicriteria decision making, optimal scheduling, data mining, and optimization methods on energy Internet network.

Ling Wang received the B.Sc. degree in automation, and the Ph.D. degree in control theory and control engineering from Tsinghua University in 1995 and 1999, respectively. Since 1999, he has been with the Department of Automation, Tsinghua University, where he became a Full Professor in 2008. His current research interests include intelligent optimization and production scheduling.

He was the recipient of the National Natural Science Fund for Distinguished Young Scholars of China, the National Natural Science Award (second place) in 2014, the Science and Technology Award of Beijing City in 2008, and the Natural Science Award (first place in 2003, and second place in 2007) nominated by the Ministry of Education of China.

Xin Xu (Senior Member, IEEE) received the B.S. and the Ph.D. degrees in control science and engineering from National University of Defense Technology (NUDT) in 1996 and 2022, respectively. He is currently a Full Professor with the Institute of Unmanned Systems, College of Intelligence Science and Technology, NUDT. His research interests include intelligent control, reinforcement learning, approximate dynamic programming, machine learning, robotics, and autonomous vehicles. He received the National Science Fund for Outstanding Youth in China and the second-class National Natural Science Award of China.

Xiangke Liao received the B.S. degree in computer science and technology from Tsinghua University in 1985, and the M.S. degree in computer science and technology from the National University of Defense Technology (NUDT) in 1988, both in computer science. He is currently a Professor with the College of Computer Science and Technology, NUDT. His research interests include high-performance computing systems, operating systems, and parallel and distributed computing. He is the Principle Investigator and Chief Designer of Tianhe-2 supercomputer.