10-2023

Instance-specific algorithm configuration via unsupervised deep graph clustering

Wen SONG

Yi LIU

Zhiguang CAO
Singapore Management University, zgcao@smu.edu.sg

Yaoxin WU

Qiqiang LI

Citation

SONG, Wen; LIU, Yi; CAO, Zhiguang; WU, Yaoxin; and LI, Qiqiang. Instance-specific algorithm configuration via unsupervised deep graph clustering. (2023). *Engineering Applications of Artificial Intelligence*. 125, 1-13.

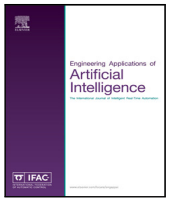
Available at: https://ink.library.smu.edu.sg/sis_research/8086

Engineering Applications of Artificial Intelligence 125 (2023) 106740

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai



Instance-specific algorithm configuration via unsupervised deep graph clustering

Wen Song [a,1], Yi Liu [b,1], Zhiguang Cao [c], Yaoxin Wu [d,*], Qiqiang Li [b,*]

[a] Institute of Marine Science and Technology, Shandong University, China
[b] School of Control Science and Engineering, Shandong University, China
[c] School of Computing and Information Systems, Singapore Management University, Singapore
[d] Faculty of Industrial Engineering and Innovation Sciences, Eindhoven University of Technology, Netherlands

ARTICLE INFO

Keywords:
Algorithm configuration
Unsupervised graph embedding
Mixed-integer programming

ABSTRACT

Instance-specific Algorithm Configuration (AC) methods are effective in automatically generating high-quality algorithm parameters for heterogeneous NP-hard problems from multiple sources. However, existing works rely on manually designed features to describe training instances, which are simple numerical attributes and cannot fully capture structural differences. Targeting at Mixed-Integer Programming (MIP) solvers, this paper proposes a novel instances-specific AC method based on end-to-end deep graph clustering. By representing an MIP instance as a bipartite graph, a random walk algorithm is designed to extract raw features with both numerical and structural information from the instance graph. Then an auto-encoder is designed to learn dense instance embeddings unsupervisedly, which facilitates clustering heterogeneous instances into homogeneous clusters for training instance-specific configurations. Experimental results on multiple benchmarks show that the proposed method can improve the solving efficiency of CPLEX on highly heterogeneous instances, and outperform existing instance specific AC methods.

1. Introduction

NP-hard problems such as Mixed-Integer Programming (MIP), Boolean Satisfiability Problem (SAT), and Constraint Programming (CP), are ubiquitous in modeling and solving practical decision-making problems (Song et al., 2022a). There are many highly parameterized algorithms (e.g., Branch-and-Cut) and solvers (e.g., CPLEX, Gurobi, OR-Tools) that can solve these problems. However, the practical solving performance of these algorithms or solvers is often largely affected by their parameters. Therefore, adjusting the parameter configuration appropriately for the algorithm/solver according to properties of the problem to be solved is of great importance. Traditionally, the parameters are set manually based on expert experience or trial-and-error. However, when the configuration space is large, which is the case for most modern solvers, finding the most appropriate configuration manually could be very tedious and even impractical.

Algorithm Configuration (AC) (Hutter et al., 2011) is an effective approach to circumvent manual parameter tuning. Instead, AC methods automatically search the configuration space to find good algorithm parameters. Depending on whether a surrogate model is included, which is used to predict the performance of configurations, existing AC methods can be roughly divided into model-free and model-based methods. Typical *model-free* methods include ParamILS (Parameter Iterated Local Search) (Hutter et al., 2009), GGA (Gender-based Genetic Algorithm) (Ansótegui et al., 2009), and irace (López-Ibáñez et al., 2016). Due to the NP-hardness, running and evaluating a configuration on instances often incurs high cost, which is hard to avoid in model-free methods. *Model-based* methods were proposed to alleviate this issue, by using a surrogate model to predict the configuration performance and updating the model through evaluation iterations. Hutter et al. (2011) proposed SMAC (Sequential Model-based Algorithm Configuration) based on the probabilistic regression model, which enables the participation of multiple instances in the configuration process. They also extended the supported parameter types to categorical and conditional parameters. Ansótegui et al. (2015) proposed GGA++ by embedding the random forest model into GGA, and optimized the selection of parents and offsprings in the underlying genetic algorithm. Wang et al. (2016) proposed REMBO (Random Embedding Bayesian Optimization), which resolves the issue of scaling Bayesian Optimization methods such as SMAC to higher dimensions.

As an alternative to AC, algorithm portfolio selects for each instance the most suitable solver from a set of available ones, so that to maximize practical solving performance (Xu et al., 2008). It is based on the

* Corresponding authors.
E-mail addresses: wensong@email.sdu.edu.cn (W. Song), 201934500@mail.sdu.edu.cn (Y. Liu), zhiguangcao@outlook.com (Z. Cao), y.wu2@tue.nl (Y. Wu), qqli@sdu.edu.cn (Q. Li).

[1] Wen Song and Yi Liu contributed equally.

https://doi.org/10.1016/j.engappai.2023.106740
Received 30 December 2022; Received in revised form 4 June 2023; Accepted 27 June 2023
Available online xxxx

intuition that no single solver performs the best on all instances. Therefore, the main advantage of algorithm portfolio is that the algorithm can be selected according to the characteristics of the specific instance, especially when the instances differ greatly (Gomes and Selman, 2001). An outstanding example is SATzilla (Xu et al., 2012), an instance-based method for automatically constructing algorithm portfolio, which employs the empirical hardness models and cost-sensitive classification models to evaluate algorithm performance. Also based on SAT solvers, Lindauer et al. (2017) proposed the automatic construction of parallel portfolios (ACPP) method, which generates parallel portfolio by combining the configuration spaces of several different solvers. Liu et al. (2019) proposed to construct ACPP by grouping instances explicitly. To improve the capability of the parallel portfolios in generalizing to instances not included in the training set, they further proposed GAST (generative adversarial solver trainer) (Liu et al., 2020), which employs an adversarially trained generator to create more training instances.

In general, conventional AC methods only generate one configuration for a target algorithm, based on the assumption that all the training instances are homogeneous from the same distribution. This is a major limitation because in reality, instances could be *heterogeneous* and come from different sources (Xu et al., 2010). In such *multi-source* cases, instances could differ greatly and it is difficult for conventional AC methods to find an overall good configuration. Algorithm portfolio methods can mitigate this issue to some extent, by selecting algorithm on a per-instance basis. However, a major drawback of portfolio methods is that they require a set of strong and uncorrelated candidate solvers, which could not be available in many cases (Xu et al., 2011). As a combination of the two types of methods, instance-specific AC preserves the advantages of both sides. In this direction, Hydra (Xu et al., 2011) is a representative AC framework. The main idea is to determine a configuration with the best overall performance, and then iteratively add useful algorithms and remove useless ones from the portfolio. Given a highly parameterized algorithm and a set of training instances, Hydra automatically generates a set of configurations that form an effective portfolio. Instead of focusing on iterative selection of algorithms, Kadioglu et al. (2010) proposed ISAC (Instance-Specific Algorithm Configuration), which first groups the instances into different clusters based on their similarities, and then apply GGA on each cluster to obtain the corresponding configurations. As shown in the experiments of Malitsky and Sellmann (2012), ISAC significantly outperforms competing methods including SATzilla and Hydra.

While various instance-specific AC methods have achieved great success in handling multi-source instances, the representation of instances is rarely studied. Such representation is important because it directly characterizes the instances and determines the similarities between them. Current instance-specific AC methods depend on manually designed features (e.g., number of variables and constraints) to represent different instances, which requires extensive domain knowledge and trial-and-error to design. Moreover, the manual features could hardly be comprehensive, and may lose important information that are crucial to achieve good performance. Recently, deep (reinforcement) learning has been shown to be effective in speeding up solving NP-hard problems such as vehicle routing (Kool et al., 2019; Xin et al., 2020; Wu et al., 2021c), scheduling (Park et al., 2021; Song et al., 2022b), satisfaction problems (Selsam et al., 2019; Song et al., 2022a), and MIP (Gasse et al., 2019; Wu et al., 2021a,b). It is well acknowledged that deep neural networks could learn high-quality features that are useful for problem solving directly from raw problem features (Bengio et al., 2021). However, most of existing works focus on learning certain components (e.g., branching heuristics) inside the target algorithm. In terms of configuring algorithm parameters from the outside, the research is rather sparse (Eggensperger et al., 2019).

In this paper, we focus on instance-specific AC for MIP, and fill the above research gap by proposing a novel deep representation learning method, so as to save tedious manual efforts and improve the performance of algorithm configuration on heterogeneous instances. Our method, named DGCAC (Deep Graph Clustering based Algorithm Configuration), is based on the framework of ISAC, one of the best instance-specific AC methods. It automatically extracts feature embeddings from training instances in an end-to-end fashion, which are then fed into an off-the-shelf clustering algorithm to divide the training instances into multiple groups, on each of which an optimized configuration is obtained by a standard AC algorithm (we use SMAC here). Specifically, we employ the graph representation of MIP, and design an auto-encoder which learns to embed the training instances into a low-dimensional feature space unsupervisedly. Such feature extraction scheme enables combining simple numerical features and complex structural features that are difficult to obtain in manual design, so as to extract rich information that can better identify and represent the MIP instances. We also discuss the effects of different clustering methods on the configuration performance, and propose to replace the original g-means algorithm in ISAC with k-means to improve the homogeneity of instances within each cluster, which empirically lead to better performance. Extensive experiments show that our DGCAC method improves the performance of ISAC in configuring the target solver CPLEX. Based on the automatically extracted features, our method can produce configurations with shorter runtime than those generated with traditional manual features. The configuration generated by our method also shows better ability in generalizing to instances of larger sizes that are unseen in training.

To summarize, this paper makes the following contributions:

- We propose a novel deep learning based instance-specific AC method for MIP, which saves manual efforts in feature designing and improves the configuration performance.
- We design an unsupervised graph learning method to extract MIP instance feature embeddings, which enables extracting not only simple numerical information but also complex structural features.
- We propose to use k-means instead of g-means in the ISAC framework, so that the number of clusters can be adjusted appropriately for better performance.
- We verify the effectiveness of our method on heterogeneous MIP instances from various sources. Results show that the features automatically learned by DGCAC can lead to better configuration performance, and effectively generalizes to instances larger than those used in training.

This paper is organized as follows. Section 2 introduces preliminary knowledge. Section 3 describes our method in detail. Section 4 reports experiments and analysis. Section 5 concludes the paper.

2. Preliminaries

In this section, we introduce some important concepts that are closely related to our method.

2.1. The Algorithm Configuration Problem

Here we define the problem of AC following Birattari and Kacprzyk (2009). A parameterized target algorithm T is given with N configurable parameters $\theta = \{\theta_1, \ldots, \theta_N\}$. Each parameter θ_i can take any value from its domain D_i, which forms a configuration space $\Theta = \prod_{i=1}^{N} D_i$ that provides a range for all the configurable parameters in the algorithm. Depending on the function of each parameter, the domain D_i could be discrete (integer or categorical) or continuous. A parameter configuration means to select for each configurable parameter θ_i a value within their respective range D_i. At the same time, AC requires a set of training problem instances P and a cost measurement function F, which is usually set based on the runtime of solving the problem instance or the quality of the returned solution under certain time constraints. The problem of AC is to find an optimal parameter configuration θ^* that minimizes the cost measurement function F when running the target algorithm T on training instances P, i.e., $\theta^* = \text{argmin}_{\theta \in \Theta} F(\theta)$.

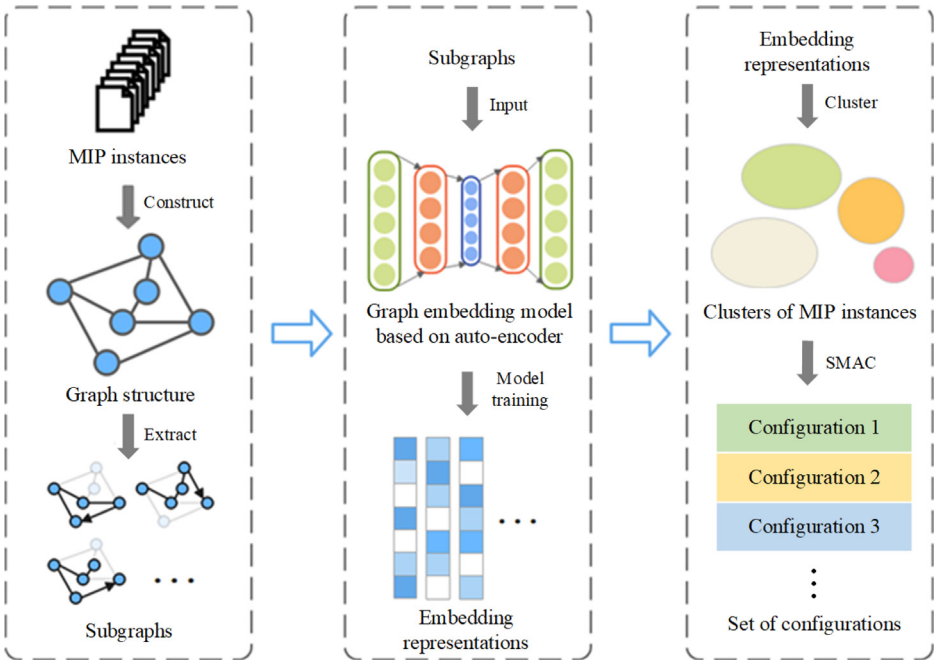


Fig. 1. Overall workflow of DGCAC.

2.2. Sequential Model-based Algorithm Configuration

Sequential Model-based Algorithm Configuration (SMAC) (Hutter et al., 2011) is a general and mainstream AC method with excellent performance. It constructs a fitting model, which replaces the random selection of configuration in model-free methods, so as to improve the configuration efficiency. Starting from the default parameter configuration of the target algorithm, SMAC iteratively performs three steps, i.e., model fitting, configuration selection and intensification, to optimize the configuration. For the fitting model, SMAC adopts random forest to produce more accurate prediction of configuration performance and quantify the uncertainty in the prediction, which is convenient for the calculation of the subsequent acquisition function. For configuration selection, SMAC adopts Expected Improvement (EI) as the acquisition function to balance the exploration and exploitation of configuration space. SMAC is equipped with an intensification mechanism to control how many times each configuration can be evaluated and when to choose a configuration to be the incumbent configuration.

2.3. Instance-Specific Algorithm Configuration

Instance-Specific Algorithm Configuration (ISAC) (Kadioglu et al., 2010) is a generic algorithm configuration method that can optimize various solvers based on instance features. Compared with instance-oblivious AC methods such as SMAC, ISAC can greatly improve the parameter configuration performance when the instances are heterogeneous. The process of ISAC is as follows. First, features of each training instance are extracted. The original ISAC uses manually designed features to represent instances. For MIP, these features include the number of variables, the number of constraints, the mean value and standard deviation of various coefficients, etc. Then, based on the extracted features, all training instances are grouped into different clusters by applying the standard clustering algorithm g-means. Finally, for each cluster which can be considered as homogeneous, GGA (Ansótegui et al., 2009) is invoked to optimize the configuration. After training, when solving new instance, ISAC finds the cluster it belongs to based on its features, and invoke the configuration of that cluster to solve it.

Note that SMAC and ISAC are designed under different assumptions. While SMAC assumes the training instances to be homogeneous, ISAC assumes a harder but more practical situation that the training instances are heterogeneous (e.g., from multiple sources). Consequently, ISAC employs a clustering mechanism to obtain multiple homogeneous instance groups, on which homogeneous AC method such as SMAC and GGA can be applied.

2.4. Clustering

The main task for clustering is to classify a set of unlabeled data into a number of groups. Data points in the same group should have similar attributes or features. In this paper, we focus on two well-known clustering algorithms, i.e., k-means and g-means. The idea of k-means algorithm is to randomly select k cluster centers, calculate the distance between each data point and each group center, and then classify all points into the group where the closest point is located. For each group, the cluster center is recalculated according to all data points in the group. The above steps were iteratively repeated to obtain the final clustering result. Another clustering algorithm, g-means (Hamerly and Elkan, 2004), is a method that can automatically adjust the number of clusters k, which resolves the problem of k-means that the k value needs to be set in advance. The main idea of g-means is that the data points of a good cluster will present a Gaussian distribution around the cluster center. The g-means algorithm considers all input data points as a cluster, then selects the current cluster in each iteration and evaluates whether the Gaussian distribution condition is satisfied with statistical Anderson–Darling test. If the Gaussian distribution condition is not satisfied, the current cluster is divided into two clusters by 2-means clustering. The above steps are iteratively repeated to obtain the final clustering results.

3. Methodology

This section presents our method DGCAC (Deep Graph Clustering based Algorithm Configuration) for heterogeneous MIP instances. Based on the framework of ISAC, our aim is to achieve end-to-end clustering directly from instance data, so as to avoid manual feature design and improve configuration quality. Fig. 1 shows the overall workflow with three stages. In the first stage, we construct a graph structure to represent the MIP instance, and extract multiple subgraphs from

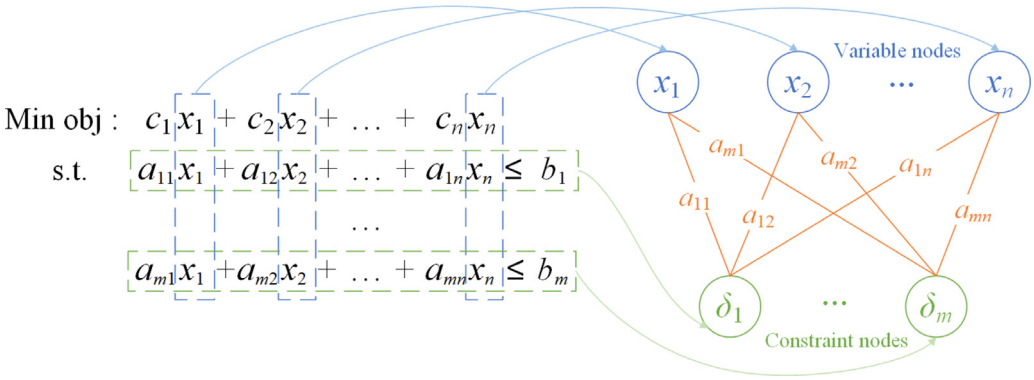


Fig. 2. Graph representation of the MIP instance.

the MIP graph by an improved random walk algorithm to retain both numerical and structural information. In the second stage, the extracted subgraphs are fed into an auto-encoder, which learns to map the graph representation of each training instance into a low dimensional space in an unsupervised manner, so as to obtain the embedding (i.e., a feature vector) of each instance. In the third stage, we apply k-means algorithm on the learned instance embeddings to cluster the instances, and train a configuration for each cluster using SMAC. In the following subsections, we will introduce each of the three stages, as well as the overall training and inference procedures.

3.1. MIP graph representation and subgraph extraction

3.1.1. MIP graph representation

The general form of MIP problem can be written as:

$$\min \quad \mathbf{c}^\top \mathbf{x} \tag{1}$$

$$\text{s.t.} \quad \mathbf{A}\mathbf{x} \le \mathbf{b}, \tag{2}$$

$$\mathbf{l} \le \mathbf{x} \le \mathbf{u}, \tag{3}$$

where $\mathbf{x} \in \mathbb{Z}^r \times \mathbb{R}^{n-r}$ is a vector of n decision variables with r integer variables and $n-r$ real variables bounded by $\mathbf{l} = [l_1, \ldots, l_n] \in \mathbb{R}^n$ and $\mathbf{u} = [u_1, \ldots, u_n] \in \mathbb{R}^n$, $\mathbf{c} = [c_1, \ldots, c_n] \in \mathbb{R}^n$ is the objective coefficient vector, and $\mathbf{A} = [a_{ji}]_{m \times n} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} = [b_1, \ldots, b_n] \in \mathbb{R}^m$ together defines m constraints.

Instance-specific AC methods describe each MIP instance using a set of features. Conventional methods rely on manually designed features (e.g., the number of variables n and constraints m, mean and standard deviation of objective and constraint coefficients). However, such manual design is tedious and time-consuming, and the resulting feature set could hardly be optimal in fully describing the instance. Instead of using manual features, we represent MIP instances based on the recent proposed bipartite graph scheme (Gasse et al., 2019). It considers variables and constraints in an MIP instance as two types of nodes, and all kinds of coefficients are constructed as edges or node attributes. The resulting graph structure could fully capture information of the MIP instance, which provides the data basis for automatically learning the internal structure and properties that is difficult to find by manual feature designing.

The bipartite graph representation of MIP is shown in Fig. 2. Given an MIP instance with n variables and m constraints, we first construct n variable nodes x_i $(1 \le i \le n)$ and m constraint nodes δ_j $(1 \le j \le m)$. The coefficient of constraint matrix can be constructed as the edge between variable and constraint node because it is related to both variables and constraints and there is a one-to-one corresponding relationship. If a variable i appears in a constraint j with nonzero coefficient a_{ji}, the corresponding variable node is connected to the constraint node to form an edge e_{ji}. For the zero values in the coefficient matrix $\mathbf{A}$, the corresponding variable node and constraint node will not form an edge, resulting in structural differences in the graph representation among different MIP instances.

Based on the above defined topology of the MIP graph structure, we further define attributes for each node and edge, so that numerical information can be injected. For each variable node x_i, its attributes consist of four parts including (1) its coefficient c_i in the objective function; (2) the type of x_i, which could be binary, integer or continuous; (3) the upper bound information of x_i, and (4) the lower bound information of x_i. For the variable type, we use one-hot encoding with three binary values z_i^B, z_i^I and z_i^C to indicate whether x_i is binary, integer or continuous, respectively. For the upper and lower bound information, since not all variables have a specified upper or lower bound in the vector $\mathbf{l}$ or $\mathbf{u}$, we use four values I_i^U, $\bar{u}_i$, I_i^L, and $\bar{l}_i$ as the attributes. Specifically, I_i^U and I_i^L are two binary values to indicate whether x_i has a specified upper or lower bound, respectively. If x_i has an upper bound, then $I_i^U = 1$ and $\bar{u}_i = u_i$, otherwise $I_i^U = 0$ and $\bar{u}_i = 0$. The logic of I_i^L and $\bar{l}_i$ is the same. For each constraint node δ_j, we use the corresponding constraint bounds b_j as its node attribute. Finally, for each edge that connects variable node x_i and constraint node δ_j, we use the corresponding coefficient a_{ji} in the constraint matrix $\mathbf{A}$ as its attribute. The attribute vectors $\mathbf{y}_{x_i}$ for each variable node, $\mathbf{y}_{\delta_j}$ for each constraint node, and $\mathbf{y}_{e_{ji}}$ for each edge are summarized as follows:

$$\mathbf{y}_{x_i} = [c_j, z_i^B, z_i^I, z_i^C, I_i^U, \bar{u}_i, I_i^L, \bar{l}_i]^\top \in \mathbb{R}^8, \forall 1 \le i \le n, \tag{4}$$

$$\mathbf{y}_{\delta_j} = [b_j] \in \mathbb{R}, \forall 1 \le j \le m, \tag{5}$$

$$\mathbf{y}_{e_{ji}} = [a_{ji}] \in \mathbb{R}, \forall a_{ji} \ne 0. \tag{6}$$

In summary, an MIP instance can be constructed as a graph $G = (V_v, V_c, E)$, where V_v, V_c and E are the set of variable nodes, constraint nodes and edges, respectively. Defining an MIP instance in this way gives a more comprehensive representation of the information it contains.

3.1.2. Subgraph extraction

After constructing the graph structure for the MIP instance, the amount of obtained data could be huge, especially for large-scale instances. How to extract information from the MIP graph to achieve efficient and comprehensive data acquisition for each instance should be carefully considered. Here we design an improved random walk method to extract subgraphs of the MIP graph structure to obtain comprehensive instance information in an efficient way. Random walk algorithm is an information extraction method for graph, which randomly selects a node in the graph as the starting point, and continuously selects an adjacent node of the current node as the next node until it reaches a preset path length. The set of nodes and edges accessed by each round of random walk on the graph form a subgraph.

However, conventional random walk has some limitations in extracting structural information of the MIP graph. Since the walking is random, it is possible to return to some part of the graph that has been

visited. These revisits cannot be detected in conventional random walk methods. Here we adopt the recently proposed SEED method (Wang et al., 2020), which improves conventional random walk by extracting the timestamp feature, which indicates the time when the node is first visited (i.e., its earliest visiting time) in the current round of walk. This feature is useful in distinguishing subgraphs with similar numerical features but different structural features which is common in the MIP graph we defined in this paper, and hence is beneficial for distinguishing different MIP instances.

Remind that the MIP graph structure we constructed in Section 3.1.1 is a bipartite graph, which contains two types of nodes with different attribute types and quantities. However, the SEED method can only handle standard graphs with homogeneous nodes. In this paper, we extend it to handle bipartite graph. According to structural characteristics of the bipartite graph, we set the starting node of each random walk to be of the same type, and set the same path length for each random walk. In this way, the extracted subgraph representations (in terms of feature vectors) of different random walks are consistent and comparable.

Our random walk algorithm is designed as follows. Given a bipartite graph $G = (V_v, V_c, E)$ created for an MIP instance p as input, we set the starting node of all random walks to be a randomly selected variable node. Then, we perform s times of random walks to extract s subgraphs, each of the same path length w (excluding the starting node) where w is set as an even number. When the time (or step) t is 0, we randomly select a variable node $x^{(0)} \in V_v$ as the starting node, and set its timestamp $y_\tau^{(0)}$ (i.e., the earliest visiting time) to 0. Then, among all neighboring nodes of $x^{(0)}$, another node is randomly sampled as the next node which is a constraint node $\delta^{(1)} \in V_c$, and the corresponding timestamp $y_\tau^{(1)}$ of the sampled node is set to 1. This process is iterated for w steps, during which if the sampled node has not been visited, then its timestamp $y_\tau^{(t)}$ is set to the current time t, otherwise its timestamp remains as its earliest visiting time. Besides the timestamp, we also extract two parts of information during each step t ($1 \le t \le w$) of the random walk, including the attribute $\mathbf{y}_x^{(t)}$ or $\mathbf{y}_\delta^{(t)}$ of the visited node (depending on whether a variable node or constraint node is visited) and the edge attribute $\mathbf{y}_e^{(t)}$ of the edge connecting the two nodes sampled in step $t-1$ and t.

The result of a random walk is a subgraph of the MIP instance graph, which can be represented as a (raw) feature vector $\mathbf{Y}$ by organizing the attribute and timestamp data obtained during the random walk. Specifically, the vector $\mathbf{Y}$ contains three parts, including the node attribute vector $\mathbf{y}_{node}$, edge attribute vector $\mathbf{y}_{edge}$ and timestamp vector $\mathbf{y}_{time}$. For the node attribute vector, the w node attribute vectors obtained in w times of sampling are concatenated to construct a β-dimension vector defined below:

$$\mathbf{y}_{node} = [\mathbf{y}_\delta^{(1)}, \mathbf{y}_x^{(2)}, \dots, \mathbf{y}_\delta^{(w-1)}, \mathbf{y}_x^{(w)}]^\top \in \mathbb{R}^\beta, \tag{7}$$

where $\beta = 8\times\frac{w}{2} + 1\times\frac{w}{2} = \frac{9w}{2}$, since w is even and the number of visited variable nodes and constraint nodes are $w/2$.

The construction process of edge attribute vector $\mathbf{y}_{edge}$ and timestamp vector $\mathbf{y}_{time}$ is the same as above. Since the information sampled by edge attribute and timestamp attribute in each step is a 1-bit scalar, the final vector dimension is w, as shown in Eqs. (8) and (9):

$$\mathbf{y}_{edge} = [\mathbf{y}_e^{(1)}, \mathbf{y}_e^{(2)}, \dots, \mathbf{y}_e^{(w-1)}, \mathbf{y}_e^{(w)}]^\top \in \mathbb{R}^w, \tag{8}$$

$$\mathbf{y}_{time} = [y_\tau^{(1)}, y_\tau^{(2)}, \dots, y_\tau^{(w-1)}, y_\tau^{(w)}]^\top \in \mathbb{R}^w. \tag{9}$$

Based on the above definitions, the feature vector $\mathbf{Y}$ of the subgraph after each round of walk is constructed by concatenating the node attribute vector, edge attribute vector and timestamp vector to form a $\beta + 2w$ dimensional vector, as shown in Eq. (10):

$$\mathbf{Y} = [\mathbf{y}_{node}, \mathbf{y}_{edge}, \mathbf{y}_{time}]^\top \in \mathbb{R}^{\beta+2w}. \tag{10}$$

After s rounds of random walks on the corresponding MIP instance graph, we can obtain the representation of an MIP instance p as $\mathbb{Y}_p = \{\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_s\}$, where $\mathbf{Y}_i$ is the subgraph feature vector extracted in the ith round. So far, the random walk based feature extraction of the MIP instance is completed.

Remark. Traditional manually designed features can only extract numerical information (e.g., the number of variables and constraints) from the MIP instances. However, numerical features are far from enough. When giving two instances from the same source but with different scale, for example, two combinatorial auction problems, the coefficient of one instance is much larger than that of the other. Considering only numerical features, the two instances are likely to be divided into two clusters. However, they have strong similarity in the relationship between variables and constraints, and should be put into the same cluster and solved using the same parameter configuration. Our feature extraction method avoids the above limitation since we inject the structural information of MIP instances. Specifically, the subgraphs extracted by random walk can reflect the structural differences between instances. With the support of numerical information on the nodes and edges, the extracted information is more comprehensive, hence could lead to better instance clustering.

3.2. Deep graph embedding learning based on auto-encoder

The raw features of each MIP instance graph extracted above is high-dimensional, and is difficult to cluster directly by clustering algorithm. Hence it is necessary to obtain dense low-dimensional feature representation, i.e., embedding of the MIP graph. To this end, we design a deep auto-encoder (Zhai et al., 2018), which learns to map the raw feature vector of each instance to the embedding space unsupervisedly, so as to reduce feature dimension of the MIP instance graph for better clustering.

Before fed into the auto-encoder, the raw feature vectors of the MIP instance graphs are pre-processed by the following two steps. First, the timestamp vector $\mathbf{y}_{time}$ of each subgraph is converted to one-hot encoding, which is suitable for dealing with discrete values. Since the length of each random walk is w, a state register of w-bit is used to represent the state of each timestamp attribute $y_{time}^{(t)}$ in $\mathbf{y}_{time}$. Each state has an independent register bit, and only one of the w bits is valid. Therefore, the timestamp vector of each random walk is converted from $\mathbf{y}_{time} \in \mathbb{R}^w$ to $\mathbf{y}_{time} \in \mathbb{R}^{w^2}$, and the raw feature vector of the corresponding subgraph is converted from $\mathbf{Y} \in \mathbb{R}^{\beta+2w}$ to $\mathbf{Y} \in \mathbb{R}^{\beta+w+w^2}$. Second, we normalize the raw features of each subgraph, since the coefficients and attributes of instances from different sources and different scales may be quite different in the order of magnitude. We use min–max normalization to scale the raw features into [0, 1].

The auto-encoder we designed consists of the encoder $f(\cdot)$ and decoder $g(\cdot)$, which are implemented as Multi-Layer Perceptrons (MLPs). For each extracted subgraph raw feature vector $\mathbf{Y}$, the encoding function $f(\cdot)$ transforms it to a dense low-dimensional vector $\mathbf{h} \in \mathbb{R}^{d_h}$ (i.e., the embedding vector) using MLP parameters η_e. Then, the decoding function reconstruct a vector $\hat{\mathbf{Y}}$ of the same length as $\mathbf{Y}$, using MLP parameters η_d. The above process is shown below:

$$\mathbf{h} = f(\mathbf{Y}; \eta_e), \tag{11}$$

$$\hat{\mathbf{Y}} = g(\mathbf{h}; \eta_d). \tag{12}$$

Note that the encoding and decoding MLPs have the symmetrical structure, which consists of an input layer, a hidden layer and an output layer. Dimensions of the hidden layers in both the encoder and decoder are equal, denoted as d_e. LeakyReLU is used as the activation function in the two MLPs.

Given an input $\mathbf{Y}$ and the corresponding output $\hat{\mathbf{Y}}$ of the auto-encoder, we calculate the mean square error (MSE) between them to measure the similarity between the original raw feature and the reconstructed one:

$$MSE(\mathbf{Y}, \hat{\mathbf{Y}}) = \| \hat{\mathbf{Y}} - \mathbf{Y} \|^2 . \tag{13}$$

Based on MSE, we can optimize the encoding and decoding parameters η_e and η_d using standard gradient descent algorithm, by setting the mean MSE over subgraphs extracted from all training instances as the

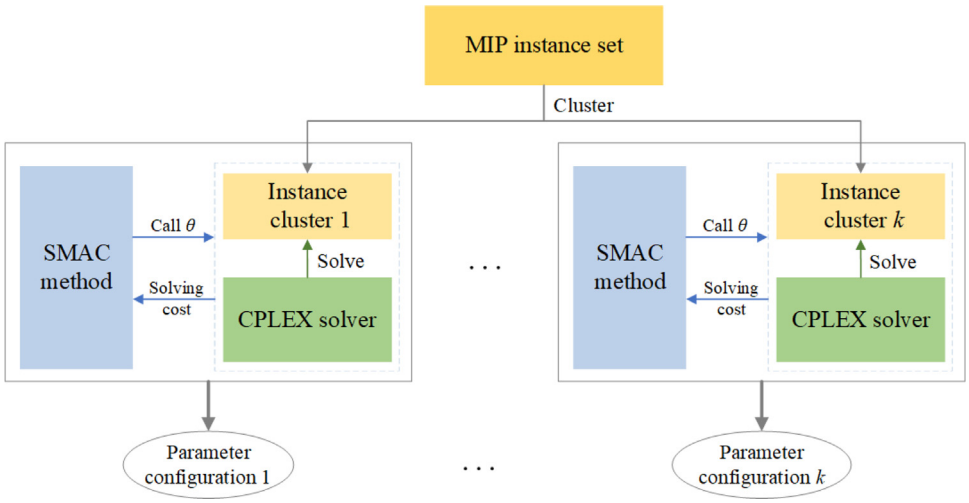


Fig. 3. Schematic diagram of algorithm configuration training.

loss function. After training, we can obtain a set of embedding vectors $\mathbb{H}_p = \{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_s\}$ of each input instance p, by applying the trained encoder to each subgraph vector $\mathbf{Y}$. By averaging all vectors in $\mathbb{H}_p$, we obtain a d_h-dimensional vector $\overline{\mathbf{h}}_p$, as the representation of the input MIP instance p in the embedding space:

$$\overline{\mathbf{h}}_p = \frac{1}{s}\sum_{i=1}^{s}\mathbf{h}_i. \tag{14}$$

3.3. Clustering and configuration

Based on the dense low-dimensional representation learned by the auto-encoder, we employ clustering algorithm to divide the multi-source instances into multiple clusters as in ISAC, so that instances with similar features can be grouped together. To measure the similarity between instances, we calculate the Euclidean distance between the corresponding embedding vectors. Taking two MIP instances p and q as an example, the corresponding instance embeddings are $\overline{\mathbf{h}}_p$ and $\overline{\mathbf{h}}_q$ of dimension d_h. The distance between the two embeddings in the embedding space is:

$$dist(\overline{\mathbf{h}}_p, \overline{\mathbf{h}}_q) = \sqrt{\sum_{i=1}^{d_h}(\overline{h}_{p_i} - \overline{h}_{q_i})^2}. \tag{15}$$

The smaller the distance between the two embeddings, the higher the similarity of the two instances, and the more inclined they are to the same cluster.

The schematic diagram for the clustering and configuration process is shown in Fig. 3. For the configuration training of each cluster, we adopt the general AC method SMAC described in Section 2.2, due to its good performance in optimizing parameters for homogeneous instances. In terms of instances clustering, as mentioned in Section 2.3, ISAC adopts g-means as the clustering method. While it does not need to preset the number of clusters k as in k-means, however, the clustering performance could be affected if the number of clusters is completely controlled by the algorithm. In g-means, the termination condition for iterative division of clusters is that the data points of the current clustering conforms to Gaussian distribution. In the instance-specific AC problem, if a large number of training instances have similar features or are evenly distributed in the feature space, g-means cannot continue to divide instances when the overall distribution has reached Gaussian distribution. This could prevent the clustering to reach a desirable degree, resulting in an undesirable small number of clusters and the instances within each cluster could be not sufficiently homogeneous to achieve satisfactory configuration performance. To resolve this issue, we propose to substitute the g-means algorithm in ISAC with k-means, which offers better control on the number of clusters, so as to obtain better configuration optimization effect. In the experiments, we will show that the algorithm configuration performance obtained by k-means is better than that of g-means.

Algorithm 1: DGCAC-Training

Input: MIP instance set P, target algorithm T, configuration space Θ, number of subgraphs s and steps w of random walk, training epoch τ, learning rate α, number of clusters k

Output: Cluster centers C, encoder weights η_e, configurations of all clusters Q

1 $\mathcal{G} \leftarrow$ BulidGraphModel(P);
2 $\mathbb{Y} \leftarrow$ RandomWalk($\mathcal{G}, s, w$);
3 **for** $i \leftarrow 1$ **to** τ **do**
4 $\quad \mathbb{H} \leftarrow$ Encoder($\mathbb{Y}, \eta_e$);
5 $\quad \hat{\mathbb{Y}} \leftarrow$ Decoder($\mathbb{H}, \eta_d$);
6 $\quad \eta_e, \eta_d \leftarrow$ MinLoss($\mathbb{Y}, \hat{\mathbb{Y}}$) with learning rate α;
7 $\overline{\mathbf{h}} \leftarrow$ Ave-Aggregation($\mathbb{H}$);
8 $(C, S) \leftarrow$ K-Means($\overline{\mathbf{h}}, k$);
9 **for** $j \leftarrow 1$ **to** k **do**
10 $\quad Q_j \leftarrow$ SMAC(T, S_j, Θ);
11 **return** $C = \{C_1, \ldots, C_k\}$, η_e, $Q = \{Q_1, \ldots, Q_k\}$;

3.4. Training and inference procedures

The overall training process of DGCAC is shown in Algorithm 1. Given a set of training MIP instances P, we first construct the bipartite graph for each instance, collected in the set $\mathcal{G}$ (Line 1). Then, the improved random walk algorithm is performed on each instance graph, to obtain the subgraph set $\mathbb{Y}$ (Line 2). Next, we perform τ epochs of auto-encoder training to obtain the set of instance embeddings $\mathbb{H}$ (Line 3–6), based on which we get the set of aggregated instance embeddings $\overline{\mathbf{h}}$ (Line 7) and perform k-means clustering to divide the whole training set into k clusters $S = \{S_1, \ldots, S_k\}$ (Line 8). Finally, we run SMAC on each cluster S_j to obtain the optimized parameter configuration Q_j.

After training, the inference process of DGCAC for solving any new instance p is shown in Algorithm 2. We first build the bipartite graph model of instance p and perform random walk to get the subgraph representations of p, which is then processed by the trained encoder to obtain the aggregated instance embedding $\overline{\mathbf{h}}_p$ (Line 1–4). To determine which cluster the input instance p belongs to, we calculate the distance

Algorithm 2: DGCAC-Inference

Input: MIP instance p , number of subgraphs extracted s and steps w of random walk, encoder weights η_e, cluster centers C, configurations of all clusters Q , target algorithm T

Output: Configuration for testing instance Q_{min}

1 $G \leftarrow$ BulidGraphModel(p);
2 $\mathbb{Y}_p \leftarrow$ RandomWalk(G, s, w);
3 $\mathbb{H}_p \leftarrow$ Encoder($\mathbb{Y}_p, \eta_e$);
4 $\bar{\mathbf{h}}_p \leftarrow$ Ave-Aggregation($\mathbb{H}_p$);
5 **for** $j \leftarrow 1$ **to** k **do**
6 $\quad D_j \leftarrow dist(\bar{\mathbf{h}}_p, C_j)$;
7 $Q_{min} \leftarrow$ MinDistance($D = \{D_1, ..., D_k\}$);
8 $T(p, Q_{min})$;
9 **return**;

between its embedding to each of the cluster center (Line 5–6). Then, we find the cluster with the minimum distance, and retrieve the corresponding parameter configuration (Line 7), which is used by the target algorithm to solve p (Line 8).

4. Experimental evaluation

In this section, we perform experiments to validate our method. After introducing the experiment setup in Section 4.1, we first examine the performance of k-means and g-means algorithm in the proposed DGCAC in Section 4.2. Then, we compare the performance of different AC methods in Section 4.3. Finally, we examine the performance of different AC methods in generalizing the trained configuration to larger-scale unseen instance sets in Section 4.4.

4.1. Experimental setup

4.1.1. Configuration scenario

In this paper, we choose the leading commercial solver IBM ILOG CPLEX 12.10.0 as the configuration target, due to its superior performance and wide applications in MIP solving. As listed in Table A.10 in the Appendix, we optimize 24 important parameters (including 1 continuous and 23 discrete parameters) of CPLEX, and set the configuration space according to the adjustable range of parameters provided in the CPLEX user manual. Our objective is to minimize the average runtime of CPLEX.

4.1.2. Instance sets

Three instance sets are used in our experiments. All instance sets contains MIP instances from multiple sources and different scales, and is publicly available or can be generated by open-source generator.

Set1 is generated using the generator[2] in Gasse et al. (2019), which contains 4 types of MIP problems including combinatorial auction problem (Cauctions), capacitated facility location problem (Facilities), maximum independent set problem (Setcover) and maximum independent set problem (Indset). We generate three subsets of small, medium and large scale, named Set1-S, Set1-M and Set1-L. Each subset includes instances from the four types with the proportion of 1:1:1:1. The parameters we used in generating the three subsets is listed in Table 1, and the corresponding problem scale is shown in Table 2. For the small subset Set1-S, we generate 300 instances in which 200 is used for training and 100 for testing. The larger two subsets Set1-M and Set1-L contains 100 instances each, and are only used in the generalization evaluation, hence we do not perform training on these two subsets.

[2] https://github.com/ds4dm/learn2branch.

Table 1
Instance generation parameters used in creating Set1.

Source	Parameter	Small	Medium	Large
Cauctions	# items	300	400	500
	# bids	800	800	800
Setcover	# rows	1000	1500	2000
	# cols	1000	1000	1000
	Density	0.05	0.05	0.05
	Max coefficient	1000	1000	1000
Indset	# nodes	1000	1500	2000
	Affinity	4	4	4
Facilities	# customers	300	400	500
	# facilities	100	100	100
	Ratio	5	5	5

Table 2
Number of variables (n) and constraints (m) in Set1.

Source	Small		Medium		Large	
	n	m	n	m	n	m
Cauctions	800	450	800	550	800	650
Setcover	1000	1000	1000	1500	1000	2000
Indset	1000	4000	1500	6000	2000	8000
Facilities	30 000	400	40 000	500	50 000	600

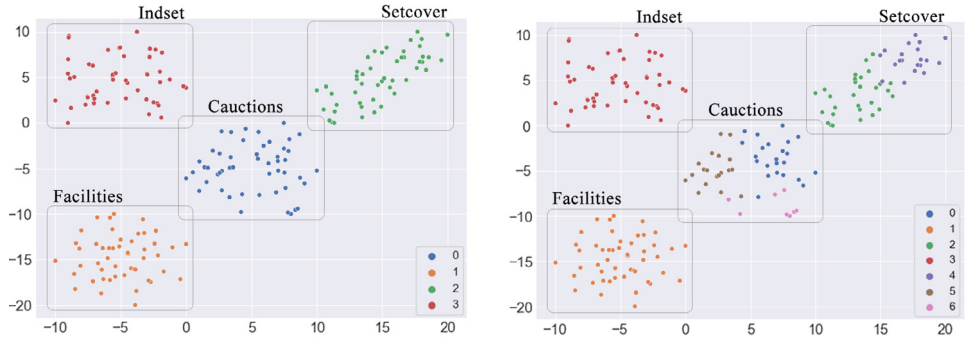
Set2 is from Alvarez et al. (2017), which is a multi-source MIP instance set with four types of constraints including: set covering (SC), multi-knapsack (MKN), bin packing (BP) and equality (EQ). Set2 contains instances generated from three constraint combinations: BP-EQ, BP-SC and MKN-SC. Here we directly use the public available instance library,[3] where for each constraint combination, there are 25 and 50 instances for training and testing, respectively, hence Set2 contains 75 training and 150 testing instances in total. The average number of variables and constraints of instances in Set2 are 200 (maximum 360) and 100 (maximum 160), respectively.

Set3 is from the well recognized and widely used benchmark MIPLIB,[4] which contains highly heterogeneous instances from a wide range of practical applications. Based on the instance scale that the proposed method proposed can handle and the difficulty of experiments, we select 60 MIPLIB instances, for which CPLEX with default configuration can solve optimally within 100 s. Details of the 60 instances are listed in Table A.9 in Appendix. The average number of variables and constraints of instances in Set3 are 8835.8 (maximum 87482) and 2334.7 (maximum 13206), respectively.

4.1.3. Hyperparameters

We empirically tune the hyperparameters of DGCAC on a small validation set. We set the path length $w = 16$ and the number of random walks $s = 100$. For the auto-encoder, we set the hidden dimension of the encoder and decoder as $d_e = 128$, and the dimension of extracted subgraph embedding (and instance embedding) to $d_h = 64$. We use the Adam optimizer to train the auto-encoder, which is one of the best-performing and widely used optimizers for neural network training (Choi et al., 2019). We perform training for $\tau = 1000$ epochs with learning rate $\alpha = 0.01$. Depending on the clustering algorithm, DGCAC is executed in two modes, namely **DGCAC-g** for g-means clustering and **DGCAC-k** for k-means. For DGCAC-g, we set the minimum number of instances in a cluster and the maximum clustering depth of g-means to 5 and 6, respectively. For k in DGCAC-k, we will discuss its impact and setting in the next subsection. As typical in previous works (e.g., Hutter et al. (2011) and Kadioglu et al. (2010)), we set a time limit of 300 s for solving each training instance (called captime) to prevent the case that the solving takes too much time under bad configurations. For instances reach the captime, we consider their solving runtime as 300 s.

[3] http://www.montefiore.ulg.ac.be/~ama/research.php.
[4] https://miplib.zib.de/.



(a) DGCAC-g (b) DGCAC-k

Fig. 4. Instance clustering visualization of DGCAC-g and DGCAC-k.

Table 3
Runtime of DGCAC-k under different k values (unit: s)

k	Train	Test	Difference
5	7.01 ± 0.92%	6.71 ± 0.91%	−4.28%
6	7.00 ± 0.97%	7.14 ± 1.02%	2.00%
7	**6.65** ± 0.94%	**6.47** ± 0.85%	−2.71%
8	6.90 ± 1.19%	6.81 ± 0.98%	−1.30%
9	7.27 ± 0.97%	7.28 ± 0.92%	0.14%

Table 4
Results of DGCAC under two clustering algorithms ("Difference" is calculated based on the runtime of train and test).

Method	Train		Test		Difference
	Runtime (s)	Wins	Runtime (s)	Wins	
DGCAC-g	6.85 ± 1.03%	95/200	6.71 ± 0.91%	32/100	−2.04%
DGCAC-k	**6.65** ± 0.94%	**105**/200	**6.47** ± 0.85%	**68**/100	−2.71%

4.1.4. Baselines

We compare our approach with the following baselines: **(1)** CPLEX with the **Default** configuration; **(2)** one of the most commonly used homogeneous AC method **SMAC**; **(3)** state-of-the-art instance-specific AC method ISAC, for which we substitute the GGA configurator with the more advanced SMAC for fair comparison, denoted as $\mathbf{ISAC_{SMAC}}$. In Table A.11 in the Appendix, the manually designed MIP features used by $\text{ISAC}_{\text{SMAC}}$ are listed. Note that the only difference between $\text{ISAC}_{\text{SMAC}}$ and DGCAC-g is that the features in our method are learned end-to-end instead of manually designed. For fair comparison, we give all methods (including ours) 10 h total training time. Our implementation is based on Python, and the experimental environment is a Linux machine (Ubuntu 18.04) with Intel(R) Core(TM) i9-9900k(3.60 GHz) CPU and 32G RAM.

4.2. Experiment 1: Analysis of clustering algorithms

In this subsection, we empirically compare the performance of DGCAC with different clustering algorithms using Set1-S. We first discuss the impact of k in DGCAC-k, which determines the number of instance clusters and further affects the following configuration performance. We set k to 5, 6, 7, 8 and 9 respectively, and conduct 10 h configuration training for each k, that is, the configuration training time allocated to each cluster is about 2 h, 1.67 h, 1.43 h, 1.25 h and 1.11 h respectively. In Table 3, we present the average runtime on the 200 training and 100 testing instances in Set1-S. We can see that the value of k does have an impact on the configuration performance, and the best k for Set1-S is 7 which will be used in the following experiments.

Table 5
Results of different methods on Set1-S.

Method	Train		Test	
	Runtime (s)	Wins	Runtime (s)	Wins
Default	8.32 ± 1.01%	11/200	7.97 ± 1.05%	10/100
SMAC	8.54 ± 0.93%	26/200	7.62 ± 0.84%	17/100
$\text{ISAC}_{\text{SMAC}}$	7.54 ± 1.10%	43/200	7.82 ± 1.09%	17/100
DGCAC-g	6.85 ± 1.03%	57/200	6.71 ± 0.91%	15/100
DGCAC-k	**6.65** ± 0.94%	**63**/200	**6.47** ± 0.85%	**41**/100

Next, we compare the two versions of our method, DGCAC-g and DGCAC-k ($k = 7$). For each version, we list the average runtime (in seconds) and the number of instances on which it achieves smaller runtime (the column "Wins") in Table 4. We can see that DGCAC-k consistently outperforms DGCAC-g, showing that k-means clustering is better than g-means for the instance-specific AC problem. To understand why, we visualize the clustering results of the two algorithms using the t-SNE method in Fig. 4. We can see that the number of clusters obtained by g-means is less than that of k-means, due to the mechanism of g-means. After the current cluster is gaussified, g-means stops on this part of instances and does not continue to cluster downward. However, such mechanism may result in clusters with relatively wide in-cluster instance distribution, which are still not homogeneous enough and cannot be effectively solved by one configuration. While k-means avoids the above limitation, the k value needs to be set appropriately. Under the same total training time, if k is too large, then the time spent in configuration training for each cluster is reduced, which could affect the configuration performance. If k is too small, then it cannot overcome the limitation of g-means. To sum up, with a suitable k, DGCAC with k-means can lead to better overall configuration performance.

4.3. Experiment 2: Performance comparison

In this subsection, we compare our method with the baselines using Set1-S, Set2 and Set3. We give all methods the same 10 h total training time, and the results are summarized in Table 5. We also give the boxplot for the distribution of instance solving time in Fig. 5. From the results, we can observe that the two versions of our method significantly outperform the baselines. SMAC performs relatively poorly, and its runtime is even larger than that of the default CPLEX in the training set. This shows that it is difficult to find a single configuration for highly heterogeneous instances from multiple sources. The instance-specific method $\text{ISAC}_{\text{SMAC}}$ shows better performance, at least in the training set on which it shortens the runtime of default CPLEX by 9.4%. With the same clustering algorithm, our DGCAC-g significantly improves $\text{ISAC}_{\text{SMAC}}$ and shortens the CPLEX default runtime by 17.7% and 15.8%

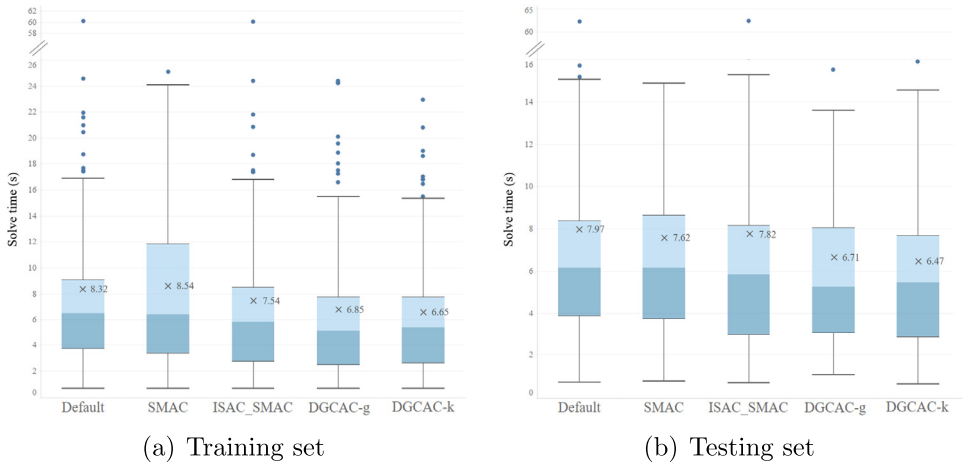


Fig. 5. Boxplots of runtime on Set1-S.

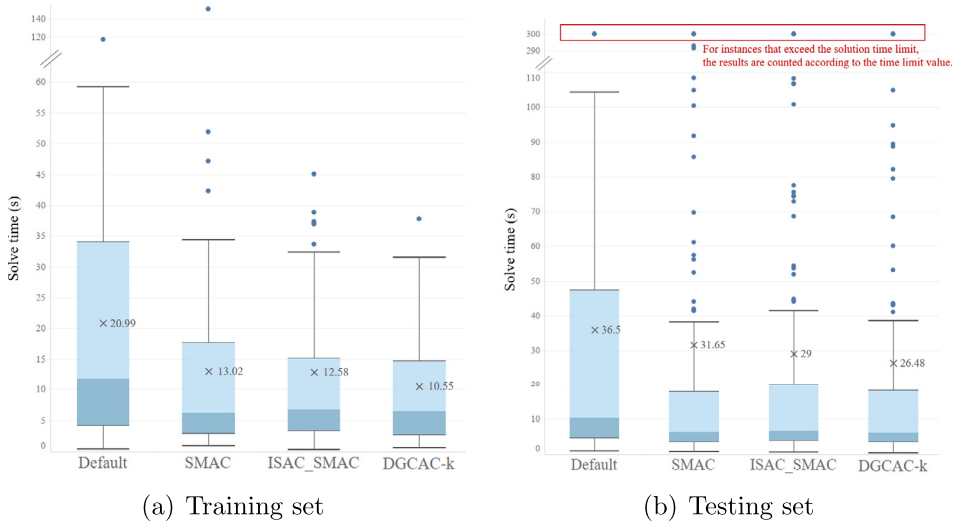


Fig. 6. Boxplots of runtime on Set2.

Table 6
Results of different methods on Set2.

Method	Train		Test	
	Runtime (s)	Wins	Runtime (s)	Wins
Default	20.99 ± 1.11%	4/75	36.50 ± 1.68%	4/145
SMAC	13.02 ± 1.51%	17/75	31.65 ± 2.14%	33/145
$ISAC_{SMAC}$	12.58 ± 1.21%	16/75	29.00 ± 2.10%	28/145
DGCAC-k	**10.55** ± 1.21%	**35**/75	**26.48** ± 2.24%	**81**/145

on the training and testing set, respectively. This verifies the advantage of our graph-based unsupervised representation learning method over using the traditional manually designed numerical features. By substituting the g-means clustering algorithm with k-means, DGCAC-k further boosts the performance and reduces the default CPLEX runtime by 20.1% and 18.8% on the training and testing set respectively, with more winning instances. Due to its better performance, we will use DGCAC-k as the representative of our method in the following experiments.

Next, we discuss the performance on Set2. Results of all methods are listed in Table 6, and the corresponding boxplot is shown in Fig. 6. We can see that SMAC can effectively improve the default CPLEX configuration, probability because Set2 is not that heterogeneous than Set1. Nevertheless, the instance-specific methods $ISAC_{SMAC}$ and DGCAC-k still outperforms SMAC, with our method being the best. Specifically, DGCAC-k reduces the runtime of default configuration by 49.7% and 27.5% on the training and testing set, respectively. This shows that even for homogeneous instances, it could be better to divide them into multiple clusters such that more focused configurations can be trained. Note that in the testing set, each method has five instances that cannot be solved optimally within the 300 s limited time, for which 300 s is used in calculating the average runtime in Table 6, which explains why the average runtime of the testing set is greater than that of the training set.

For Set3, since the MIPLIB instances are highly heterogeneous, we conduct three rounds of experiments, during each we randomly split the 60 instances in the ratio of 3:1 as the training and testing set, so as to avoid the coincidence that good results are caused by a particular data split. Results of the three rounds of experiments are summarized in Table 7. It is interesting to see that in all the three rounds, the performance of SMAC and $ISAC_{SMAC}$ are almost the same as the default configuration. In fact, the configurations trained by the two methods are almost the same as the default one, showing that they are not capable in handling instances that are so different from each other. For SMAC, it is simply because it is designed for homogeneous instances only. For $ISAC_{SMAC}$, the manually designed numerical features are

Table 7
Results of different methods on Set3.

Round	Method	Train	Test
1	Default	16.94 ± 1.28%	13.20 ± 1.66%
	SMAC	17.06 ± 1.28%	13.21 ± 1.66%
	$ISAC_{SMAC}$	17.11 ± 1.28%	13.20 ± 1.68%
	DGCAC-k	**15.64** ± 1.23%	**10.18** ± 1.47%
2	Default	16.63 ± 1.38%	21.2 ± 1.34%
	SMAC	16.65 ± 1.38%	21.13 ± 1.34%
	$ISAC_{SMAC}$	16.49 ± 1.38%	**21.04** ± 1.34%
	DGCAC-k	**14.86** ± 1.35%	34.50 ± 2.19%
3	Default	19.27 ± 1.35%	12.91 ± 1.30%
	SMAC	19.20 ± 1.35%	12.89 ± 1.30%
	$ISAC_{SMAC}$	19.21 ± 1.35%	13.00 ± 1.30%
	DGCAC-k	**15.81** ± 1.37%	**11.89** ± 1.39%

Table 8
Results of the generalization experiments on Set1.

Scale	Method	Runtime (s)	Wins
Small	Default	7.97 ± 1.05%	10/100
	SMAC	7.62 ± 0.84%	17/100
	$ISAC_{SMAC}$	7.82 ± 1.09%	17/100
	DGCAC-k	**6.47** ± 0.85%	**41**/100
Medium	Default	40.06 ± 1.09%	12/100
	SMAC	38.86 ± 1.09%	28/100
	$ISAC_{SMAC}$	37.10 ± 1.10%	29/100
	DGCAC-k	**36.45** ± 1.20%	**31**/100
Large	Default	97.63 ± 0.76%	17/100
	SMAC	99.71 ± 0.79%	19/100
	$ISAC_{SMAC}$	94.59 ± 0.69%	10/100
	DGCAC-k	**90.22** ± 0.83%	**54**/100

not informative enough to generate meaningful instance clusters. In contrast, our method can still effectively optimize the configurations in all the three rounds of training, except on the testing set of round 2. This is because one instance reaches the 300 s solving time limit, resulting in an excessive value.

To sum up, on the three benchmarks with different characteristics, our method almost consistently outperforms baselines, showing its strong ability and good robustness in configuring MIP solvers for heterogeneous instances.

4.4. Experiment 3: Generalization performance analysis

Finally, we evaluate the ability of each method in generalizing the trained configuration to large instances unseen in training, which is a desired property for practical usage. For this purpose, we apply the configurations trained by each method on Set1-S, which has the smallest scale, to solve the two subsets Set1-M and Set1-L that are much larger. Results are summarized in Table 8 (results on Set1-S are copied from Table 5). We can see that both $ISAC_{SMAC}$ and DGCAC-k exhibit the generalization ability on the instances of three scales. In terms of the average runtime, $ISAC_{SMAC}$ improves that of default CPLEX configuration by 1.9%, 7.4%, and 3.1% on the small, medium and large instances, respectively. For our method DGCAC-k, the corresponding improvement is 15.8%, 9.0%, and 7.6%. We can see that for both methods, the improvement over default configuration drops with the increase of problem scale, which is common for machine learning models since the performance usually degrade with the increase of distribution shift. Nevertheless, the generalization ability of our method is still better than that of $ISAC_{SMAC}$.

5. Conclusions and future work

While being effective in training configurations for heterogeneous instances, existing instances-specific algorithm configuration methods rely on simple numerical features that are manually designed, which relies on human experience and could limit the configuration performance. This paper proposes DGCAC, a novel instance-specific AC method for MIP problem, which overcomes this limitation by learning instance representation directly from instance data. DGCAC reformulates each MIP as a graph, extracts both numerical and structural information through random walk. Then, it learns instance representation unsupervisedly based on auto-encoder, and performs k-means clustering to formulate homogeneous instance clusters. The above process is automated, without the need of manual feature engineering. Experiments on three representative benchmarks with different degrees of heterogeneity well validate the effectiveness of our method. Results show that our method can generate appropriate clusters for heterogeneous instances, leading to better AC performance than the well-known instance-oblivious method SMAC. Moreover, the features automatically extracted by the unsupervised graph learning can lead to better configuration performance than that of the traditional manually designed features used in the original ISAC. Since our target solver CPLEX is a powerful commercial solver widely used in many industries (e.g., manufacturing, logistics, maritime affairs), our method can potentially be applied in speeding up MIP solving in a wide range of practical applications. In the future, an interesting direction is to include clustering in the unsupervised representation learning process, so as to enhance the feature extraction ability. We also plan to extend our method to support other types of NP-hard problems such as SAT and CP.

CRediT authorship contribution statement

Wen Song: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Yi Liu:** Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Zhiguang Cao:** Methodology, Software, Validation, Writing – review & editing. **Yaoxin Wu:** Conceptualization, Methodology, Writing – original draft. **Qiqiang Li:** Conceptualization, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This study is supported by the National Natural Science Foundation of China under Grant 62102228, and in part by Shandong Provincial Natural Science Foundation, China under Grant ZR2021QF063.

Appendix

The MIPLIB instances considered in this paper is listed in Table A.9. The optimized CPLEX parameters and their descriptions are listed in Table A.10. The manual features used in ISAC are listed in A.11.

Table A.9
The MIPLIB instances and the number of variables (n) and constraints (m).

Instance name	n	m	Instance name	n	m
roll3000	1 166	2 295	mc11	3 040	1 920
fast0507	63 009	507	neos5	63	63
beasleyC3	2 500	1 750	p200x1188c	2 376	1 388
swath1	6 805	884	nursesched-sprint02	10 250	3 522
neos-2978193-inde	20 800	396	drayage-25-23	11 090	4 630
n2seq36q	22 480	2 565	rail507	63 019	509
enlight_hard	200	100	neos-1445765	20 617	2 147
supportcase18	13 410	240	ns1952667	13 264	41
exp-1-500-5-5	990	550	dano3_3	13 873	3 202
mik-250-20-75-4	270	195	gmu-35-40	1 205	424
roi2alpha3n4	6 816	1 251	30n20b8	18 380	576
neos-5075914-elvire	5 003	3 720	pg	2 700	125
fastxgemm-n2r6s0t2	784	5 998	neos-5107597-kakapo	3 114	6 498
peg-solitaire-a3	4 552	4 587	timtab1	397	171
neos-860300	1 385	850	neos-662469	18 235	1 085
neos-1171737	2 340	4 179	pg5_34	2 600	225
ns1208400	2 883	4 289	nu25-pr12	5 868	2 313
neos-911970	888	107	neos17	535	486
pk1	86	45	nw04	87 482	36
rmatr100-p10	7 359	7 260	tr12-30	1 080	750
ns1830653	1 629	2 932	rocI-4-11	6 839	10 883
qap10	4 150	1 820	hypothyroid-k1	2 602	5 195
graph20-20-1rand	2 183	5 587	eil33-2	4 516	32
markshare_4_0	34	4	sp150x300d	600	450
drayage-100-23	11 090	4 630	n5-3	2 550	1 062
irp	20 315	39	cod105	1 024	1 024
mcsched	1 747	2 107	neos-3381206-awhea	2 375	479
neos-3083819-nubu	8 644	4 725	neos-1456979	4 605	6 770
sct2	5 885	2 151	seymour1	1 372	4 944
neos859080	160	164	neos-1171448	4 914	13 206

Table A.10
CPLEX parameters considered in this paper.

Name	Type	Range	Default	Description
emphasis.mip	int	0,1,2,3,4	0	Controls trade-offs between speed, feasibility, optimality, and moving bounds in MIP.
preprocessing.repeatpresolve	int	−1,0,1,2,3	−1	Specifies whether to re-apply presolve, with or without cuts, to an MIP model after processing at the root is otherwise complete.
preprocessing.relax	int	−1,0,1	−1	Decides whether LP presolve is applied to the root relaxation in an MIP.
solutiontype	int	0,1,2	0	Specifies type of solution (basic or non basic) that CPLEX produces for a linear program (LP) or quadratic program (QP).
preprocessing.boundstrength	int	−1,0,1	−1	Decides whether to apply bound strengthening in mixed integer programs (MIPs).
mip.strategy.startalgorithm	int	0,1,2,3,4,5,6	0	Sets which continuous optimizer will be used to solve the initial relaxation of an MIP.
mip.strategy.subalgorithm	int	0,1,2,3,4,5	0	Decides which continuous optimizer will be used to solve the subproblems in an MIP, after the initial relaxation.
mip.strategy.variableselect	int	−1,0,1,2,3,4	0	Sets the rule for selecting the branching variable at the node selected for branching.
mip.strategy.bbinterval	int	0,1,2,3,4,5,6,7	7	Sets the best bound interval for MIP strategy.
mip.strategy.branch	int	−1,0,1	0	Decides which branch, the up or the down branch, should be taken first at each node.
mip.strategy.backtrack	float	[0,1]	0.9999	Controls how often backtracking is done during the branching process.
mip.strategy.dive	int	0,1,2,3	0	Controls the MIP dive strategy.
mip.strategy.lbheur	int	0,1	0	Controls whether CPLEX applies a local branching heuristic to try to improve new incumbents found during an MIP search.
mip.strategy.nodeselect	int	0,1,2,3	1	Used to set the rule for selecting the next node to process when backtracking.
mip.strategy.presolvenode	int	−1,0,1,2,3	0	Decides whether node presolve should be performed at the nodes of a mixed integer programming (MIP) solution.
mip.strategy.probe	int	−1,0,1,2,3	0	Sets the amount of probing on variables to be performed before MIP branching.
mip.limits.aggforcut	int	1,2,3,4,5	3	Limits the number of constraints that can be aggregated for generating flow cover and mixed integer rounding (MIR) cuts.
mip.cuts.cliques	int	−1,0,1,2,3	0	Decides whether or not clique cuts should be generated for the problem.

(*continued on next page*)

Table A.10 (*continued*).

Name	Type	Range	Default	Description
mip.cuts.covers	int	−1,0,1,2,3	0	Decides whether or not cover cuts should be generated for the problem.
mip.cuts.disjunctive	int	−1,0,1,2,3	0	Decides whether or not disjunctive cuts should be generated for the problem.
mip.cuts.flowcovers	int	−1,0,1,2	0	Decides whether or not to generate flow cover cuts for the problem.
mip.cuts.pathcut	int	−1,0,1,2	0	Decides whether or not flow path cuts should be generated for the problem.
mip.cuts.gomory	int	−1,0,1,2	0	Decides whether or not Gomory fractional cuts should be generated for the problem.
mip.cuts.gubcovers	int	−1,0,1,2	0	Decides whether or not to generate GUB cuts for the problem.

Table A.11
Manually designed features used in ISAC.

ID	Feature	Type	ID	Feature	Type
1	Number of variables	int	13	Mean value of the vector of RHS of the constraints	float
2	Number of constraints	int	14	Min value of the vector of RHS of the constraints	float
3	Percentage of integer variables	float	15	Max value of the vector of RHS of the constraints	float
4	Percentage of continuous variables	float	16	Standard deviation of the vector of RHS of the constraints	float
5	Percentage of $\leq$ constraints	float	17	Mean value of the vector of number of variables (all, integer or continuous) per constraint	float
6	Percentage of $\geq$ constraints	float	18	Min value of the vector of number of variables (all, integer or continuous) per constraint	int
7	Percentage of = constraints	float	19	Max value of the vector of number of variables (all, integer or continuous) per constraint	int
8	Percentage of variables with non zero coefficients in the objective function	float	20	Standard deviation of the vector of number of variables (all, integer or continuous) per constraint	float
9	Mean value of the vector of coefficients of the objective function	float	21	Mean value of the vector of the coefficients of variables (all, integer, or continuous) per constraint	float
10	Min value of the vector of coefficients of the objective function	float	22	Min value of the vector of the coefficients of variables (all, integer, or continuous) per constraint	float
11	Max value of the vector of coefficients of the objective function	float	23	Max value of the vector of the coefficients of variables (all, integer, or continuous) per constraint	float
12	Standard deviation of the vector of coefficients of the objective function	float	24	Standard deviation of the vector of the coefficients of variables (all, integer, or continuous) per constraint	float

References

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

Ansótegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K., 2015. Model-based genetic algorithms for algorithm configuration. In: Proceedings of the 24th International Conference on Artificial Intelligence. IJCAI '15, AAAI Press, pp. 733–739.

Ansótegui, C., Sellmann, M., Tierney, K., 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (Ed.), Principles and Practice of Constraint Programming - CP 2009. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 142–157.

Bengio, Y., Lodi, A., Prouvost, A., 2021. Machine learning for combinatorial optimization: a methodological tour d'horizon. European J. Oper. Res. 290 (2), 405–421.

Birattari, M., Kacprzyk, J., 2009. Tuning Metaheuristics: A Machine Learning Perspective, Vol. 197. Springer.

Choi, D., Shallue, C.J., Nado, Z., Lee, J., Maddison, C.J., Dahl, G.E., 2019. On empirical comparisons of optimizers for deep learning. arXiv preprint arXiv:1910.05446.

Eggensperger, K., Lindauer, M., Hutter, F., 2019. Pitfalls and best practices in algorithm configuration. J. Artificial Intelligence Res. 64, 861–893.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., Lodi, A., 2019. Exact combinatorial optimization with graph convolutional neural networks. Adv. Neural Inf. Process. Syst. 32.

Gomes, C.P., Selman, B., 2001. Algorithm portfolios. Artificial Intelligence 126 (1), 43–62, Tradeoffs under Bounded Resources.

Hamerly, G., Elkan, C., 2004. Learning the K in K-Means. Adv. Neural Inf. Process. Syst. 17.

Hutter, F., Hoos, H.H., Leyton-Brown, K., 2011. Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (Ed.), Learning and Intelligent Optimization. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 507–523.

Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T., 2009. ParamILS: An automatic algorithm configuration framework. J. Artif. Intell. Res. 36 (1), 267–306.

Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K., 2010. ISAC –Instance-specific algorithm configuration. In: Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence. IOS Press, NLD, pp. 751–756.

Kool, W., van Hoof, H., Welling, M., 2019. Attention, learn to solve routing problems!. In: International Conference on Learning Representations.

Lindauer, M., Hoos, H., Leyton-Brown, K., Schaub, T., 2017. Automatic construction of parallel portfolios via algorithm configuration. Artificial Intelligence 244, 272–290.

Liu, S., Tang, K., Yao, X., 2019. Automatic construction of parallel portfolios via explicit instance grouping. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33. pp. 1560–1567.

Liu, S., Tang, K., Yao, X., 2020. Generative adversarial construction of parallel portfolios. IEEE Trans. Cybern. 1–12. http://dx.doi.org/10.1109/TCYB.2020.2984546.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., Stützle, T., 2016. The irace package: Iterated racing for automatic algorithm configuration. Oper. Res. Perspect. 3, 43–58.

Malitsky, Y., Sellmann, M., 2012. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. Springer, pp. 244–259.

Park, J., Chun, J., Kim, S.H., Kim, Y., Park, J., 2021. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. Int. J. Prod. Res. 59 (11), 3360–3377.

Selsam, D., Lamm, M., Benedikt, B., Liang, P., de Moura, L., Dill, D.L., et al., 2019. Learning a SAT solver from single-bit supervision. In: International Conference on Learning Representations.

Song, W., Cao, Z., Zhang, J., Xu, C., Lim, A., 2022a. Learning variable ordering heuristics for solving Constraint Satisfaction Problems. Eng. Appl. Artif. Intell. 109, 104603.

Song, W., Chen, X., Li, Q., Cao, Z., 2022b. Flexible job-shop scheduling via graph neural network and deep reinforcement learning. IEEE Trans. Ind. Inform. 19 (2), 1600–1610.

Wang, Z., Hutter, F., Zoghi, M., Matheson, D., De Freitas, N., 2016. Bayesian optimization in a billion dimensions via random embeddings. J. Artif. Intell. Res. 55 (1), 361–387.

Wang, L., Zong, B., Ma, Q., Cheng, W., Ni, J., Yu, W., Liu, Y., Song, D., Chen, H., Fu, Y., 2020. Inductive and unsupervised representation learning on graph structured objects. In: International Conference on Learning Representations.

Wu, Y., Song, W., Cao, Z., Zhang, J., 2021a. Learning large neighborhood search policy for integer programming. Adv. Neural Inf. Process. Syst. 34.

Wu, Y., Song, W., Cao, Z., Zhang, J., 2021b. Learning scenario representation for solving two-stage stochastic integer programs. In: International Conference on Learning Representations.

Wu, Y., Song, W., Cao, Z., Zhang, J., Lim, A., 2021c. Learning improvement heuristics for solving routing problems... IEEE Trans. Neural Netw. Learn. Syst..

Xin, L., Song, W., Cao, Z., Zhang, J., 2020. Step-wise deep learning models for solving routing problems. IEEE Trans. Ind. Inform. 17 (7), 4861–4871.

Xu, L., Hoos, H.H., Leyton-Brown, K., 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. AAAI '10, AAAI Press, pp. 210–216.

Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K., 2008. SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. 32 (1), 565–606.

Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K., 2011. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In: RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion At the International Joint Conference on Artificial Intelligence (IJCAI). pp. 16–30.

Xu, L., Hutter, F., Shen, J., Hoos, H., Leyton-Brown, K., 2012. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. In: Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions. pp. 55–58.

Zhai, J., Zhang, S., Chen, J., He, Q., 2018. Autoencoder and its various variants. In: 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC). pp. 415–419. http://dx.doi.org/10.1109/SMC.2018.00080.