SCIENCE CHINA
Mathematics

AI Methods for Optimization Problems
• ARTICLES •

June 2024 Vol. 67 No. 6: 1317–1340
https://doi.org/10.1007/s11425-023-2290-y

Learning to sample initial solution for solving 0-1 discrete optimization problem by local search

Xin Liu, Jianyong Sun* & Zongben Xu

School of Mathematics and Statistics, Xi'an Jiaotong University, Xi'an 710049, *China*
*Email: liuxin*17*@stu.xjtu.edu.cn, jy.sun@xjtu.edu.cn, zbxu@mail.xjtu.edu.cn*

Received July 1, 2023; accepted April 1, 2024; published online April 29, 2024

Abstract Local search methods are convenient alternatives for solving discrete optimization problems (DOPs). These easy-to-implement methods are able to find approximate optimal solutions within a tolerable time limit. It is known that the quality of the initial solution greatly affects the quality of the approximated solution found by a local search method. In this paper, we propose to take the initial solution as a random variable and learn its preferable probability distribution. The aim is to sample a good initial solution from the learned distribution so that the local search can find a high-quality solution. We develop two different deep network models to deal with DOPs established on set (the knapsack problem) and graph (the maximum clique problem), respectively. The deep neural network learns the representation of an optimization problem instance and transforms the representation to its probability vector. Experimental results show that given the initial solution sampled from the learned probability distribution, a local search method can acquire much better approximate solutions than the randomly-sampled initial solution on the synthesized knapsack instances and the Erdős-Rényi random graph instances. Furthermore, with sampled initial solutions, a classical genetic algorithm can achieve better solutions than a random initialized population in solving the maximum clique problems on DIMACS instances. Particularly, we emphasize that the developed models can generalize in dimensions and across graphs with various densities, which is an important advantage on generalizing deep-learning-based optimization algorithms.

Keywords discrete optimization, deep learning, graph convolutional network, local search

MSC(2020) 90C27, 68T07, 05C69

Citation: Liu X, Sun J Y, Xu Z B. Learning to sample initial solution for solving 0-1 discrete optimization problem by local search. Sci China Math, 2024, 67: 1317–1340, https://doi.org/10.1007/s11425-023-2290-y

1 Introduction

It is well known that many discrete optimization problems are NP-hard [34], such as knapsack problem (KP) [9], maximum clique problem (MCP) [67], and other problems. Exact approaches, such as dynamic programming [6] and branch-and-bound (B&B) algorithm [36], can find the global optimum of a discrete optimization problem (DOP) with mathematical guarantee. However, these algorithms are with extremely-high time complexities which make it impractical to solve large-scale DOPs in a tolerable time. Alternatively, approximate algorithms, such as heuristics and meta-heuristics, are able to find a high-quality solution within a reasonable time. Approximate methods are widely used in discrete optimization and can be classified into single-solution based and population-based algorithms [19, 46, 51, 56, 60].

* Corresponding author

Single-solution based approximate methods, such as local search (LS), tabu search (TS) [21] and simulated annealing (SA) [32], start from an initial solution and update it to a 'nearby' solution in its predefined neighborhood by certain rules iteratively [13]. The key components in a single-based approximate method usually include the initial solution construction, a neighborhood structure, and a rule for solution selection in the neighborhood. For example, the best candidate solution in a neighborhood is admissible to replace the current one in TS, while in SA a worse solution is allowed with some probability [32]. In the variable neighborhood search algorithm [24], the neighborhood structure is adaptively adjusted during the search process depending on the solution quality.

Population-based approximate methods, such as particle swarm optimization, differential evolution, ant colony optimization and many others, maintain a group of solutions (called population). At each iteration, new solutions are generated by exchanging information among solutions in a way inspired by natural and physical phenomena, while those yielding better objectives are to be retained with high probability to the next iteration by selection [17].

We can see that all these methods start their search processes from an (or some) initial solution(s) in the feasible region. Initial solution(s) can be constructed by either random, greedy or heuristic methods [1,68]. Random initialization means to generate the initial solutions in a random way. For example, to construct a solution $\boldsymbol{x} \in \{0,1\}^n$, we could sample from a Bernoulli distribution with parameter p for each element $x_i, 1 \leqslant i \leqslant n$. A greedy method makes the optimal choice at each step as it attempts to find the overall optimal way to solve an optimization problem. The solution found by the greedy method can yield good performance by exploiting the considered problem. However, it fails to find the global optimum because it makes decisions based only on the information it has at any one step, without regard to the overall problem [65]. The heuristic methods, on the other hand, attempt to construct initial solution(s) according to some sort of domain knowledge, mainly trying to generate feasible solutions. For example, the first items satisfying the constraint are taken as the initial solution after sorting all items in an ascending (resp. descending) order of the cost-to-size (resp. profit-to-weight) ratio for the min-knapsack (resp. max-knapsack) problems [14, 47]. For MCP, an initial clique can be derived by successively discarding the node whose degree is less than the core number of the current subgraph after rearranging the nodes in the increasing order of node degrees [27, 53, 54].

All these aforementioned algorithms require initial solution(s). It has been well acknowledged that the quality of initial solution(s) can greatly affect the performance of these algorithms. It is easy to understand since an initial solution which locates close to the global optimum can lead the algorithm to the global optimum more easily than a solution far away from the global optimum. Furthermore, for population-based algorithms, an initial population with weak diversity would suffer from stagnation, while one with low-quality could take too long to converge. Thus, it is necessary to form appropriate initial solutions for different DOPs and different heuristics. It is thus important to develop an algorithm for initial solution generation.

In this paper, different from the aforementioned initial solution construction methods, we propose to use deep-learning-based technology. The underlying idea of this technology is to use deep learning (DL) technology to extract the experiences obtained when optimizing some related optimization problems to usable knowledge. The knowledge can then be used to facilitate the optimization on new problems for improving the algorithm's efficiency and efficacy.

We focus on DOPs with binary variables, including KP and MCP on behalf of the set-based and graph-based discrete problems. In this paper, we propose to take the initial solution as a random variable, and learn a probability distribution with respect to the initial solution by using deep learning. It is expected that by sampling initial solution(s) from the learned probability distribution, the performance of existing heuristic and meta-heuristic methods can be improved significantly.

The main contributions of this work can be summarized as follows:

- We propose to learn an optimal probability distribution for the initial solution of an LS so that the quality of the final solution found by the LS could be greatly improved.
- A learning algorithm is developed to search for the optimal initial probability distribution of a given DOP instance.

• To avoid the high computational cost of the learning algorithm required for high-dimensional DOPs, a deep neural network (DNN) is used to learn the mapping between a DOP instance and its corresponding optimal initial probability vector. Two different models are applied to learn the mapping function on set-based and graph-based problems, respectively.

• We conduct extensive experiments to compare sampling initial solution(s) from the learned probability and random sampling on various KP and MCP instances with various sizes. Experimental results favor the proposed sampling against random sampling. Furthermore, it is verified that the learned distribution is able to generalize well to complex DOP instances with large sizes.

The remainder of this paper is organized as follows. In Section 2, we introduce some related pieces of work on using deep-learning-based technology for optimizing DOPs. The framework for finding the optimal probability distribution of the initial solution to a DOP is proposed in Section 3. In Section 4, we present two studies on learning the mapping between the probability vector and the problem instance taking knapsack and maximum clique problem as examples, respectively. Experimental results are summarized in Section 5. We conclude this paper in Section 6.

2 Related work

Basically, there are three ways to use deep learning to address DOPs: end-to-end learning, learning-to-configure and embedding learning [5, 61].

End-to-end learning means to use deep learning technology to find an approximate solution to a DOP directly. For example, a pointer network is proposed based on the attention mechanism [3, 64], in which a decoder (formed as a deep neural network) outputs the selection probability of a DOP's solution over the input sequence of the DOP [66]. The point network has been applied to tasks such as traveling salesman problem (TSP), convex hull problem and feature point matching problem [48, 66]. A similar model for TSP is proposed and trained by reinforcement learning (RL) [59] via the tour length, i.e., the distance of the route where each city is visited only once, taken as the utility [4]. Compared with supervised learning, RL eliminates the need for computing optimal solutions of DOPs as labels [50]. For example, a graph attention network takes the coordinates of each node in a graph as input and generates a permutation of the input nodes for routing problems [33]. In [15], the evaluation function is firstly parameterized by embedding node features and neighborhood information, then the node with the highest evaluation value is selected to construct the solution recursively. Moreover, graph neural networks (GNNs) are applied to solve DOPs on graphs, such as graph coloring and vertex cover problems, which can be encoded as propositional satisfiability problems [37, 55, 63, 70]. Besides, the graph convolutional neural network (GCN), which is a specialized class of GNN incorporating spectral graph convolution [16], also has been applied to graph-based classification tasks [23, 31].

Learning-to-configure means to extract features using machine learning and/or deep learning and to use these features for parameter configuration in optimization algorithms. Algorithmic parameters commonly exist and are important to algorithmic performance. Traditional methods to configure these parameters usually require a great deal of effort to evaluate each set of configurations, which is tedious and time-consuming. DL-based work turns the parameter configuration problem into a machine learning problem by gaining some extra information from optimizing a class of similar instances. For example, Kruber et al. [35] proposed to learn to predict whether Dantzig-Wolfe decomposition is necessary for solving mixed-integer linear programming (MILP). Ansótegui et al. [2] developed a dynamic self-adaptation of reactive tabu search which builds a functional dependency between four search parameters involved in tabu search and eleven search features regarding the runtime statistics. Mahmood et al. [43] used the generative adversarial network to learn to predict a primary dose distribution of radiation therapy, which is an inverse optimization problem.

Embedding learning means to embed a DL model within the execution procedure of an optimization algorithm. Taking the branch and bound method for MILP as an example, choosing the branching variable heuristically is computationally intensive, while constantly running heuristic searches consumes a lot of running time. To address these issues, DL can be used to carry out variable selection and node selection in B&B for avoiding unnecessary computations, e.g., searching at nodes which result in no benefit to the solution [25, 41]. Moreover, Khalil et al. [30] proposed to learn a binary classification model for inferring whether the primal heuristic could find a better solution at the current node than the best one already found so far, and designed several features scalable to different dimensions of instances to characterize the current search state. They also proposed to transfer a DOP instance into a bipartite graph representation, and adopted a GCN to learn the variable selection strategy. Furthermore, imitation learning is applied to learn to select a variable among all fractional ones [20, 26]. Li et al. [39] proposed to use GCN to predict the probability map for each node to indicate whether it belongs to the optimal solution, which can then be employed in a heuristic local search to refine the probability into the final solution to DOPs defined on graphs.

The aforementioned DL-based methods assist in providing good-quality solutions directly and indirectly. It is also promising to produce the high-quality initial solution(s) by utilizing the knowledge extracted from optimization experiences [29]. For example, Louis et al. [42] proposed to combine some learned individuals based on similar solved problems, namely case-based initialization, and other random ones as the initial population for the genetic algorithm. Li and Olafsson [38] used the historical scheduling data to induce a decision tree for capturing the features of dispatching rules. Similarly, Nasiri et al. [49] extracted rules from the optimal solutions of the training problems by data-mining and sorting operations according to the confidence value. In [58], DNN is used to adjust and control algorithmic parameters, while it is trained to be an agent to construct reproduction operators [40] in evolutionary algorithms.

In general, the study on deep-learning-based technology for discrete optimization is promising yet still in its infancy. Particularly, using DL for assisting the construction of initial solution(s) is very scarce.

3 The framework

3.1 Problem definition

In this paper, we consider the following 0-1 discrete optimization problem,

$$\max \quad f(\boldsymbol{x}) = \sum_i c_i(x_i), \quad \boldsymbol{x} \in \{0,1\}^n \quad \text{s.t.} \quad \sum_i x_i = K, \tag{3.1}$$

where $\boldsymbol{x} = (x_1, \dots, x_i, \dots, x_n)^\top$ is an n-dimensional binary vector, and $c_i(x_i)$ is the profit of choosing the i-th element x_i, which makes up the total profit $f(\boldsymbol{x})$. The goal of the problem is to find a maximizer $\boldsymbol{x}^*$ with K non-zero elements.

3.2 The proposed framework of learning the preferable initial probability

Rather than obtaining a single initial solution, learning an optimal probability vector for the initial solution is more flexible. To realize this, we take the initial solution as a random variable. In the sequel, we denote $\boldsymbol{q} = (q_1, \dots, q_n)^\top \in [0,1]^n$ as a probability vector to characterize the distribution of the promising initial solution for a local search method, i.e., $P(x_i = 1) = q_i$, s.t. $\sum_{i=1}^n q_i = 1$, where q_i denotes the probability of x_i taking 1. In this paper, we propose to learn $\boldsymbol{q}$ by maximizing the utility with respect to the final solution found by local search.

Figure 1 summarizes the entire workflow. For each problem instance, an algorithm, i.e., Algorithm 1, is firstly applied to find its preferable probability vector. The pair of a problem instance (denoted by Z) and its probability vector $(Z, \boldsymbol{q})$ can be regarded as a datum. By applying Algorithm 1 on a set of problems $(Z_1, \dots, Z_N)$, we can obtain a dataset $\{(Z_1, \boldsymbol{q}_1), \dots, (Z_N, \boldsymbol{q}_N)\}$. The mapping between Z and $\boldsymbol{q}$ is then approximated by DNN.

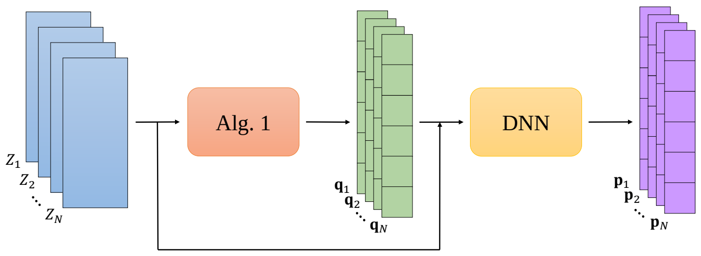


Figure 1 (Color online) The workflow of the proposed framework. Given N instances $(Z_1,\ldots,Z_N)$, an algorithm, i.e., Algorithm 1 to be introduced in the next subsection, is firstly designed to find their preferable probabilities $(\boldsymbol{q}_1,\ldots,\boldsymbol{q}_N)$. Then, a DNN is trained to predict the initial probability vector $\boldsymbol{p}$ for a DOP instance Z

3.3 Finding the optimal initial probability of each problem

Algorithm 1 The learning algorithm for achieving an optimal probability distribution for the initial solution

```
Require: the objective function $f$, the number of trajectories $N_s$, the number of repeated runs $N_e$, the momentum
    coefficient $\mu$, and the learning rate $\beta$;
Ensure: the probability distribution $\boldsymbol{q}$;
 1: Initialize $\boldsymbol{q}_0 \leftarrow \mathbf{0}$ and $\boldsymbol{V}_0 \leftarrow \mathbf{0}$;
 2: for $t = 1 \rightarrow$ maxEpoch do
 3:    ▷ Perturb the current probability vector;
 4:    for $s = 1 \rightarrow N_s$ do
 5:       Sample $\boldsymbol{\epsilon}_t^s \in \mathbb{R}^n \sim \mathcal{N}(0, \sigma^2 \boldsymbol{I})$;
 6:       $\boldsymbol{q}_t^s \leftarrow \boldsymbol{q}_{t-1} + \boldsymbol{\epsilon}_t^s$;
 7:       ▷ Evaluate the performance of the perturbed probability vector;
 8:       for $j = 1 \rightarrow N_e$ do
 9:          Sample an initial solution $\boldsymbol{x}_j \sim \boldsymbol{q}_t^s$ by roulette selection;
10:          Conduct an LS to find a best solution $\boldsymbol{x}_j^*$ and the corresponding objective value $f_j^*$ by begining with $\boldsymbol{x}_j$:
             $[\boldsymbol{x}_j^*, f_j^*] \leftarrow$ LocalSearch$(\boldsymbol{x}_j; f)$;
11:       end for
12:       Compute the utility of the perturbed probability: $r_s \leftarrow R(\boldsymbol{q}_t^s)$;
13:    end for
14:    ▷ Update $\boldsymbol{q}$;
15:    $\bar{r}_s = \frac{1}{N_s}\sum_{s=1}^{N_s} r_s$;
16:    $\Delta\boldsymbol{q}_t \leftarrow \sum_{s=1}^{N_s}(\boldsymbol{q}_t^s - \boldsymbol{q}_t)\cdot\max(\frac{r_s-\bar{r}_s}{\bar{r}_s}, 0)$;
17:    $\boldsymbol{V}_t \leftarrow \mu\boldsymbol{V}_{t-1} + \beta\Delta\boldsymbol{q}_t$;
18:    $\boldsymbol{q}_t \leftarrow \boldsymbol{q}_{t-1} + \boldsymbol{V}_t$;
19: end for
20: $\boldsymbol{q}_t \leftarrow$ softmax$(\boldsymbol{q}_t)$;
21: return $\boldsymbol{q} \leftarrow \boldsymbol{q}_t$.
```

The learning algorithm is summarized in Algorithm 1. At each iteration t, a perturbation is made to the current distribution $\boldsymbol{q}_t$ to produce multiple perturbed probability vectors (Line 6). These perturbed probabilities $\boldsymbol{q}^s, 1 \leqslant s \leqslant N_s$, are evaluated (Lines 4–13). To evaluate each perturbed probability $\boldsymbol{q}^s$, a set of initial solutions $\boldsymbol{x}_j, 1 \leqslant j \leqslant N_e$, is sampled and each $\boldsymbol{x}_j$ is deemed as an independent initial solution for an LS method, denoted as LocalSearch in Algorithm 1, which takes $\boldsymbol{x}_j$ as an argument (Lines 8–11). The LS returns the optimal solution $\boldsymbol{x}^*$ and $f^* = f(\boldsymbol{x}^*)$. The utility is then calculated according to all optimal solutions by running LS N_e times (Line 12). The utility function $R(\cdot)$ can be generally formulated as

$$R(\boldsymbol{q}) = \mathbb{E}_{\boldsymbol{x}\sim\boldsymbol{q}}[f(\boldsymbol{x}^*)] - \lambda\|\boldsymbol{q}\|_2^2, \tag{3.2}$$

where $\lambda > 0$ is a weight factor of the L_2 regularization. Note that the L_2 regularization is minimized for evolving a denser initial probability so as to explore diverse areas of the feasible region.

At the end of the epoch, $\boldsymbol{q}$ is updated by a momentum $\boldsymbol{V}$ and a measurement $\Delta\boldsymbol{q}$ to characterize the performance of the current probability (Lines 16–18). To make the learning procedure more stable, a ReLU function, i.e., ReLU$(\cdot) = \max\{0, \cdot\}$, is adopted in the increment computation, which means we only consider those probability vectors that can yield larger utility values than average (Line 16). At the end of the outer loop, the current $\boldsymbol{q}_t$ is transformed into $(0,1)$ by the softmax function (Line 20) to make sure of its legitimacy, where

$$\operatorname{softmax}(\boldsymbol{q}) = \left(\frac{e^{q_1}}{\sum_{j=1}^{n} e^{q_j}}, \ldots, \frac{e^{q_n}}{\sum_{j=1}^{n} e^{q_j}}\right)^{\top}. \tag{3.3}$$

Algorithm 2 The roulette selection

```
Require: a probability vector $\boldsymbol{q}$, the maximum cost $W$, the expected count of the selected components $K$ and the cost
    function $f_s$;
Ensure: a solution $\boldsymbol{x}$;
 1: Initialize $\boldsymbol{x} \leftarrow \mathbf{0}$ and $k \leftarrow 0$;
 2: while $k < K$ or $f_s(\boldsymbol{x}) \leqslant W$ do
 3:     set $\ell \leftarrow 0, r \leftarrow \text{rand}(0,1)$;
 4:     for $i = 1 \rightarrow n$ do
 5:         $\ell \leftarrow \ell + q_i$;
 6:         if $\ell \geqslant r$ then
 7:             $x_i \leftarrow 1$, $k \leftarrow k + 1$;
 8:             break;
 9:         end if
10:     end for
11:     if $k \geqslant n$ then
12:         Exit;
13:     end if
14: end while
15: return $\boldsymbol{x}$.
```

The sampling of an initial solution (Line 9) by the roulette selection method is given in Algorithm 2. This procedure repeats until K variables in the initial solution are assigned as 1 or the cost associated with the current option, denoted as f_s, exceeds M (Lines 2–14). In the algorithm, we firstly sample a random number r uniformly in $(0,1)$ (Line 3). Then the accumulated value of probabilities $\ell = \sum_{i=1}^{m} q_i$ is computed until there exists an index m such that ℓ is greater than r for the first time (Lines 4–10). Accordingly, the value of the m-th entry in $\boldsymbol{x}$ is set as 1, i.e., $x_m = 1$ (Line 7).

3.4 Remarks

We see that Algorithm 1 aims to find a probability vector $\boldsymbol{q}$ such that the utility over the sampled solutions is maximized. In Algorithm 1, the computational complexity mainly lies in the evaluation of each perturbed probability vector $\boldsymbol{q}_t^s$, which involves a collection of samples $\boldsymbol{x}_j, 1 \leqslant j \leqslant N_e$, specified as the starting points for local search. There are N_s perturbations per $\boldsymbol{q}_t$ and N_e initial solutions per $\boldsymbol{q}_t^s$ to be measured, which results in a total of $N_s \times N_e$ searches to be done by LS.

It is known that the more samples are measured, the more reliable the evaluation of $\boldsymbol{q}_t^s$ will be. However, the number of samples increases exponentially with the problem dimension, which makes Algorithm 1 time-consuming and impracticable to higher-dimensional problems. To alleviate the computational complexity, we propose to adopt a DNN to learn useful knowledge from past experiences about optimizing a set of low-dimensional problems, and then generalize the acquired knowledge to deal with high-dimensional problems. This procedure is problem-specific. In this paper, we take the knapsack and maximum clique problems as exemplar studies.

4 Exemplar studies

In this section, we present how to apply the proposed framework to the knapsack problem and the maximum clique problem. These two problems can be seen as representative DOPs established on set and graph, respectively.

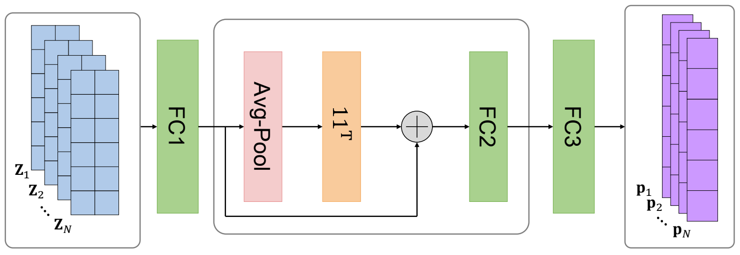


Figure 2 (Color online) The structure of Set2Pro. Set2Pro receives N knapsack instances ($\boldsymbol{Z}_1, \dots, \boldsymbol{Z}_N$) as input and outputs N corresponding probability vectors ($\boldsymbol{p}_1, \dots, \boldsymbol{p}_N$). FC1 (resp. FC2 and FC3) is a fully connected layer behind the input layer (resp. in the permutation equivariant layer and before the output layer). Avg-Pool is the average pooling operation

4.1 Model for the knapsack problem

For a KP with n candidate items denoted as $Z = \{z_1, \dots, z_n\}, 1 \leqslant i \leqslant n$, its objective function can be stated as

$$\max \quad f(\boldsymbol{x}) = \boldsymbol{v}^\top \boldsymbol{x}, \quad \boldsymbol{x} \in \{0,1\}^n, \quad \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n \quad \text{s.t.} \quad \boldsymbol{w}^\top \boldsymbol{x} \leqslant W, \tag{4.1}$$

where $x_i = 1$ means that the i-th item $z_i = (v_i, w_i)$ where v_i and w_i represents its value and weight, respectively, is selected to the knapsack which can only carry a maximal capacity of W. $x_i = 0$ means the i-th item is not selected in the knapsack.

The utility function for the KP with respect to the probability distribution $\boldsymbol{q}$ is proposed to be

$$R(\boldsymbol{q}) = \mathbb{E}_{\boldsymbol{x} \sim \boldsymbol{q}}[\boldsymbol{v}^\top \boldsymbol{x}^*] - \lambda \|\boldsymbol{q}\|_2^2, \tag{4.2}$$

where $\mathbb{E}_{\boldsymbol{x} \sim \boldsymbol{q}}[\boldsymbol{v}^\top \boldsymbol{x}^*]$ indicates the expected value over all possible $\boldsymbol{x}^*$ found by LS starting from the initial solutions sampled from $\boldsymbol{q}$.

Given a KP instance $Z = \{z_1, \dots, z_n\}$, we hope to find a model $\mathcal{G}$ such that it outputs a probability vector $\boldsymbol{p} = \mathcal{G}(Z), \boldsymbol{p} \in [0,1]^n$ under the constraint that it is *equivariant* to the order of items in Z. That is, for any permutation π to the elements in Z, elements in $\boldsymbol{p}$ are rearranged in the order of those in Z, i.e., $\mathcal{G}(\pi(Z)) = \pi(\boldsymbol{p})$. Specifically,

$$\mathcal{G}([z_{\pi(1)}, z_{\pi(2)}, \dots, z_{\pi(n)}]) = [p_{\pi(1)}, p_{\pi(2)}, \dots, p_{\pi(n)}]. \tag{4.3}$$

Inspired by the network developed by Zaheer et al. [69], which is equivariant to permutation, we construct a deep learning model named as Set2Pro to maintain the permutation equivariant property. Values and weights of each instance make up the input $\boldsymbol{Z} = [\boldsymbol{v}, \boldsymbol{w}] \in \mathbb{R}^{n \times 2}$ to Set2Pro.

Figure 2 illustrates the structure of Set2Pro. Besides the input and output layers, it consists of two types of hidden layers, including the feature extraction layer (denoted as FC1 and FC3) and the permutation equivariant layer (in the middle block). FC1 captures the individual feature of each element. The permutation equivariant layer aims to maintain the permutation effect of the intermediate solution, followed by FC3 which combines the element features and extracts the overall features from a higher perspective. The output layer is composed of a multilayer perceptron (MLP) to map the features extracted by the permutation equivariant layer into a probability distribution vector.

Note that there is no bias in FC1 and FC3, which would not jeopardize the permutation equivariant property of Set2Pro. FC1 and FC3 serve as the embedding layers, mapping the input data into vectors in another dimension and gathering local features of each item, whereas the pooling layer in the permutation equivariant layer takes into account the global feature by integrating the weight and value of each item, on the set-wise level.

A standard neural network layer G with permutation equivariance can be formulated as

$$G(\boldsymbol{x}; \boldsymbol{W}) = \sigma(\boldsymbol{W}\boldsymbol{x}), \tag{4.4}$$

where $\boldsymbol{W} \in \mathbb{R}^{n\times n}$ is the weight parameter, $\sigma : \mathbb{R}^n \to \mathbb{R}^n$ is the point-wise nonlinear activation function, such as the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. The lemma in [69] gives the necessary and sufficient conditions for G as summarized in Theorem 4.1.

Theorem 4.1. *The function $G(\boldsymbol{x}; \boldsymbol{W}) = \sigma(\boldsymbol{W}\boldsymbol{x})$ is permutation equivariant, if and only if all the off-diagonal elements of $\boldsymbol{W}$ are tied together and all the diagonal elements are equal as well, i.e.,*

$$\boldsymbol{W} = \alpha \boldsymbol{I} + \beta(\mathbf{1}\mathbf{1}^\top), \quad \alpha, \beta \in \mathbb{R}^1, \quad \mathbf{1} = [1, 1, \ldots, 1]^\top \in \mathbb{R}^n, \tag{4.5}$$

where $\boldsymbol{I} \in \mathbb{R}^{n\times n}$ is the identity matrix.

When the weight matrix $\boldsymbol{W}$ satisfying (4.5) is applied to its input $\boldsymbol{Z}$:

$$\boldsymbol{W}\boldsymbol{Z} = \alpha \boldsymbol{I}\boldsymbol{Z} + \beta(\mathbf{1}\mathbf{1}^\top)\boldsymbol{Z}, \tag{4.6}$$

the output $\boldsymbol{W}\boldsymbol{Z}$ can be regarded as a weighted sum of the input $\boldsymbol{I}\boldsymbol{Z}$ and the input sum $(\mathbf{1}\mathbf{1}^\top)\boldsymbol{Z}$. Since the summation of these two terms is permutation invariant, this layer is thus permutation equivariant depending on Z, which means that $G(\pi(\boldsymbol{Z})) = \pi(\boldsymbol{p})$ can be satisfied. If we change the summation to another pooling function, such as the average

$$\boldsymbol{W}\boldsymbol{Z} = \alpha \boldsymbol{I}\boldsymbol{Z} + \beta \ \mathrm{meanpool}(\boldsymbol{Z})\mathbf{1},$$

where meanpool($\boldsymbol{Z}$) is an average pooling that calculates the average of elements in each column of $\boldsymbol{Z}$, this layer is still permutation equivariant. We can also replace the real-valued parameters α and β to the matrix $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{H_1\times H_2}$ respectively to obtain H_2 feature maps by H_1/H_2 (input/output) channels as follows:

$$G(\boldsymbol{Z}) = \sigma(\boldsymbol{Z}\boldsymbol{A} + \mathbf{1}\mathbf{1}^\top \boldsymbol{Z}\boldsymbol{B}), \tag{4.7}$$

where H_1 (resp. H_2) is the number of neurons used in the layer FC1 (resp. FC2). To reduce the amount of parameters, letting $\boldsymbol{Z}\boldsymbol{A} \approx \boldsymbol{Z}\boldsymbol{B} + \gamma$, $\gamma \in \mathbb{R}^d$, we have

$$G(\boldsymbol{Z}) = \sigma(\gamma + (\boldsymbol{Z} + \mathbf{1}\mathbf{1}^\top \boldsymbol{Z})\boldsymbol{B}). \tag{4.8}$$

Although both (4.7) and (4.8) represent a permutation equivariant neural network layer, (4.8) has fewer parameters which is beneficial for easy training and better generalization [69]. The stack of multiple permutation equivariant layers can still guarantee the characteristic of permutation equivariance.

4.2 Model for the maximum clique problem

Given an undirected graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ is the node set and $E \subseteq V \times V$ is the edge set, the objective of MCP can be written as

$$\max \quad f(\boldsymbol{x}) = \sum_{i=1}^{n} x_i, \quad \boldsymbol{x} \in \{0,1\}^n \qquad \text{s.t.} \quad x_i + x_j \leqslant 1, \quad \forall \{i,j\} \in \bar{E}, \tag{4.9}$$

where $x_i = 1$ (resp. $x_i = 0$) means the node i is (resp. is not) in the clique that is a node subset where every two are adjacent; $\bar{G} = (V, \bar{E})$ is the complementary graph of G, i.e., $\{i,j\} \in \bar{E}$ if $\{i,j\} \notin E$. The maximum clique of G is equivalent to the independent set of $\bar{G}$. An independent set is a node subset where every two nodes are not connected by an edge [67]. The objective is to find the maximum clique with the largest cardinality in G.

The local search method for finding a maximal clique proposed by Marchiori [44,45] is adopted, which is summarized in Algorithm 3. Note that the algorithm takes a node subset as input. We can transform a binary vector $\boldsymbol{x} \in \{0,1\}^n$ easily to a node subset $U \subseteq V$ by setting $U = \{i, \text{ for all } x_i = 1, 1 \leqslant i \leqslant n\}$. In Algorithm 3, firstly a repair procedure is developed to shrink the node subset U by removing those nodes that are not adjacent with any others so that U is a clique (Lines 2–16). The clique U is then expanded to a maximal one by adding the nodes that are connected with all nodes in U (Lines 17–25).

Algorithm 3 The local search method for finding a maximal clique

```
Require: a vector $\boldsymbol{x}$ on an undirected graph $G = (V, E)$;
Ensure: a maximal clique $U$ in $G$;
 1: Initialize $U \leftarrow \{i,$ for all $x_i = 1, 1 \leqslant i \leqslant n\}$;
 2: $\triangleright$ Repair: reduce the node subset $U$ to a clique;
 3: Copy $W \leftarrow U$;
 4: repeat
 5:    Randomly select a node $w \in W$;
 6:    if rand$(0,1) < 0.01$ then
 7:       Delete the node $w$ in $U$;
 8:    else
 9:       for $u \in U$ do
10:          if $u$ is not adjacent to $w$ in $G$ then
11:             Remove $u$ from both $U$ and $W$;
12:          end if
13:       end for
14:    end if
15:    Delete $w$ in $W$;
16: until $W = \emptyset$
17: $\triangleright$ Extend: extend a clique $U$ to a maximal clique;
18: Initialize $W \leftarrow V \setminus U$;
19: repeat
20:    Choose a node $w \in W$ randomly;
21:    if $\{w\} \cup U$ is a clique then
22:       Add $w$ to $U$;
23:    end if
24:    Delete $w$ in $W$;
25: until $W = \emptyset$;
26: return $U$.
```

The utility function for MCP can be written as

$$R(\boldsymbol{q}) = \mathbb{E}_{\boldsymbol{x}\sim\boldsymbol{q}}|U| - \lambda\|\boldsymbol{q}\|_2^2, \tag{4.10}$$

where $|U| = \sum_{i=1}^n x_i$ is the cardinality of U, i.e., the clique size of U found by Algorithm 3.

Regarding the mapping from a problem instance to its preferable probability vector, we employ a two-layer GCN to create the probability over all nodes in a graph. The GCN transforms the input matrix to a new representation by the following layer-wise forward propagation:

$$X^{l+1} = \sigma((\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2})X^l\boldsymbol{H}^l), \tag{4.11}$$

where X^l (resp. X^{l+1}) denotes the input (resp. output) of the l-th hidden layer taking $\boldsymbol{H}^l$ as the weight matrix, and $\boldsymbol{A}$ (resp. $\boldsymbol{D}$) as the adjacency (resp. degree) matrix of G. Readers can refer to [8,16,31] for the detailed mathematical formulation of GCN in the context of spectral graph theory, and the theoretical foundation about the equivariance of GNN has been analyzed in [10].

Suppose we define the input feature of a graph as $\boldsymbol{Z} \in \mathbb{R}^{n\times c}$, where $\boldsymbol{z}_i, 1 \leqslant i \leqslant n$, denotes a c-dimensional feature of each node. In this paper, we take the degree of each node, i.e., the number of the adjacent nodes, as its feature representation, which indicates $c = 1$. The output, denoted as $\boldsymbol{P} \in \mathbb{R}^{n\times 1}$, of the two-layer GCN by stacking two hidden layers can be calculated as follows:

$$\boldsymbol{P} = \hat{\boldsymbol{A}}\mathrm{ReLU}(\hat{\boldsymbol{A}}\boldsymbol{Z}\boldsymbol{H}^1)\boldsymbol{H}^2, \tag{4.12}$$

where $\hat{\boldsymbol{A}} = \boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$, $\boldsymbol{H}^1 \in \mathbb{R}^{1\times H}$ (resp. $\boldsymbol{H}^2 \in \mathbb{R}^{H\times 1}$) is the weight of the first (resp. second) hidden layer, and ReLU is the rectified linear unit activation function $\max\{0, x\}$. The softmax function is then applied to all entries of $\boldsymbol{P}$ over one graph to obtain the probability vector $\boldsymbol{p}$. Note that $\boldsymbol{p}$ is the predicted probability vector according to the feature of a graph, whereas $\boldsymbol{q}$ is the label derived by enhancing the quality of solutions found by LS in Algorithm 1.

4.3 Supervised learning

The mapping between a DOP instance and its corresponding probability distribution can be learned as follows. The preferable probability vectors collected by Algorithm 1 on some DOP instances are deemed as labels, while DOP instances can be considered as the input of the DL model. To carry out the learning, we use the classical cross entropy as the loss function:

$$\mathcal{L}(\boldsymbol{q},\boldsymbol{p}) = -\sum_i q_i \times \log(p_i). \tag{4.13}$$

5 Experimental results

5.1 Experiments on the knapsack problem

5.1.1 *Training settings*

The following hyper-parameter values are set in Algorithm 1 for KP, including $N_s = 100$, $N_e = 500$, $\sigma = 0.01$, the standard deviation of perturbation $\sigma = 0.1$, the regularization weight $\lambda = 0.04$ and $\text{maxEpoch} = 1000$. The size of the hidden layer FC1 (resp. FC2) in Set2Pro is $H_1 = 30$ (resp. $H_2 = 30$). The learning rate β decays (resp. the momentum coefficient μ increases) linearly from $\beta_0 = 30$ to the minimum value $\beta_\tau = 0.3$ (resp. from $\mu_0 = 0.5$ to the maximum value $\mu_\tau = 0.99$) during the first $\tau_\beta = 700$ (resp. $\tau_\mu = 300$) epochs, and remains the same for the rest epochs. As suggested in [22], at the t-th epoch, β shrinks and μ raises as follows:

$$\beta_t = \begin{cases} \beta_0 + \dfrac{t}{\tau_\beta}(\beta_\tau - \beta_0), & 0 \leqslant t \leqslant \tau_\beta, \\ \beta_\tau, & t > \tau_\beta \end{cases} \quad \text{and} \quad \mu_t = \begin{cases} \mu_0 + \dfrac{t}{\tau_\mu}(\mu_\tau - \mu_0), & 0 \leqslant t \leqslant \tau_\mu, \\ \mu_\tau, & t > \tau_\mu. \end{cases} \tag{5.1}$$

5.1.2 *Experimental results*

Synthetic problem. To verify the developed algorithm, we first generate a set of synthetic knapsack problems and use these problems for learning and testing. We randomly generate 10,000 knapsack problems in 15–30 dimensions, i.e., the number of items n changes from 15 to 30, where $v \in \{1, 2, \ldots, 30\}$ and $w \in \{1, 2, \ldots, 20\}$ with the maximum capacity limited to be $W = 100$.

Algorithm 1 is applied on each problem Z_j to obtain the corresponding probability vector $\boldsymbol{q}_j, 1 \leqslant j \leqslant 10,000$. All the pairs $\{(Z_j, \boldsymbol{q}_j), 1 \leqslant j \leqslant 10,000\}$ constitute the data set for the supervised learning. We take the first 9,000 pairs as the training set and the remaining 1,000 pairs as the test set, respectively. The following score is used to measure the quality of the learned probability vector

$$\text{score}(\boldsymbol{q}) = \mathbb{E}_{\boldsymbol{x}\sim\boldsymbol{q}} \frac{\boldsymbol{v}^\top \boldsymbol{x}^*}{\boldsymbol{v}^\top \boldsymbol{x}^{\text{best}}}, \tag{5.2}$$

where $\boldsymbol{x}^{\text{best}}$ is the global optimal solution of the knapsack problem, which is acquired by an exact dynamic programming algorithm [62]. Clearly, the score can measure the proximity between the objective value of the global optimal solution $\boldsymbol{x}^{\text{best}}$ and the best one found by the LS with the initial solution sampled from the preferable probability $\boldsymbol{q}$. The closer the score is to 1, the more likely the LS method is to find the optimal solution.

We apply Algorithm 1 to obtain the optimal probability vectors on the generated knapsack problems. For each probability vector, we sample 1,000 initial solutions for hill climbing1) to evaluate the performance in terms of the score defined in (5.2).

1) Hill climbing [12] is a kind of local search algorithm which searches along the neighborhood of an initial solution and returns the best solution in this neighborhood. In this experiment, the neighborhood of a solution $\boldsymbol{x}$ is defined as $\mathcal{N}(\boldsymbol{x}_j) = \{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x}_j\|_H = 1\}$, where $\|\cdot\|_H$ denotes the Hamming distance, i.e., the number of occurrences of different values at the same position between two Boolean vectors.

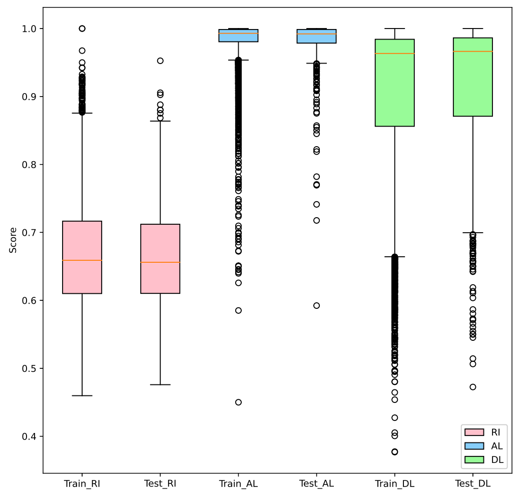


Figure 3 (Color online) The scores obtained by random initialization (RI), AL initialization and DL initialization on training and test problems, respectively

In the following, we name the initialization method by Algorithm 1 as AL, the deep-learning-based initialization method as DL and the random initialization as RI. Random initialization (RI) is compared with the proposed AL-based and DL-based initialization methods. Figure 3 summarizes the results. In Figure 3, the first two (resp. the third, fourth and the last two) columns present the results obtained by RI (resp. AL and DL) on the training and test problems. We see that the average scores obtained by AL and DL are much greater than those obtained by RI. Furthermore, the average score on the training set obtained by Set2Pro is 0.91 (the fifth column), and that on the test set is 0.92 (the sixth column) which is similar to the score on the test set obtained by AL (the fourth column). However, Set2Pro requires much less computational power to obtain the probability vector than AL since it does not need to run many simulations as it does in Algorithm 1.

We may conclude that (1) the proposed learning algorithm (Algorithm 1) can find a preferable probability distribution for initialization; (2) the mapping learned by the proposed deep neural network Set2Pro is able to well approximate the learned probability vectors for initialization on knapsack problems in 15–30 dimensions, which can greatly save computational resources.

Large-scale benchmark. To assess the performance of the learned DNN model, Set2Pro trained above is tested to predict the optimal initial probability on a large-scale 0-1 KP benchmark dataset. This benchmark comprises three types of KPs, including uncorrelated data instances, weakly correlated instances and strongly correlated instances, in $\{100, 200, 500, 1000, 2000, 5000, 10000\}$ dimensions [11] [2]. The weakly and strongly correlated instances amount to most situations in real life, which are hard to solve [52]. The setting of each instance in the large-scale benchmark is shown in Table 1.

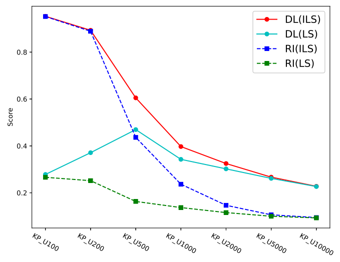
We again use the iterated local search (ILS) [57] algorithm as the local search method in the following experiments. The score of each instance obtained by applying the learned probability and the uniformly sampled initial probability on the large-scale benchmark is depicted in Figure 4.

As we can see from Figure 4, due to the superior performance of Set2Pro on the weakly and strongly correlated instances, the effectiveness of the learned probability in finding the optimal solution is clearer on harder and higher-dimensional KPs.

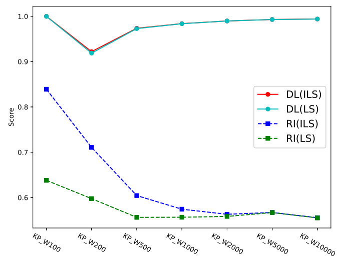
[2] The dataset is available to download from http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/.

Table 1 Large-scale 0-1 knapsack problem benchmark

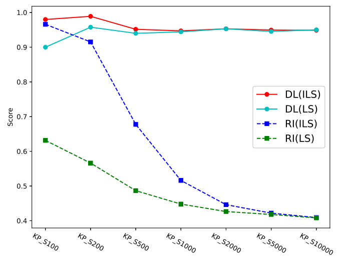
Problem	ID	n	W	f^*
Uncorrelated data instances	KP_U100	100	995	9147
	KP_U200	200	1008	11238
	KP_U500	500	2543	28857
	KP_U1000	1000	5002	54503
	KP_U2000	2000	10011	110625
	KP_U5000	5000	25016	276457
	KP_U10000	10000	49877	563647
Weakly correlated instances	KP_W100	100	995	1514
	KP_W200	200	1008	1634
	KP_W500	500	2543	4566
	KP_W1000	1000	5002	9052
	KP_W2000	2000	10011	18051
	KP_W5000	5000	25016	44356
	KP_W10000	10000	49877	90204
Strongly correlated instances	KP_S100	100	997	2397
	KP_S200	200	997	2697
	KP_S500	500	2517	7117
	KP_S1000	1000	4990	14390
	KP_S2000	2000	9819	28919
	KP_S5000	5000	24805	72505
	KP_S10000	10000	49519	146919



(a) Uncorrelated data instances



(b) Weakly correlated instances



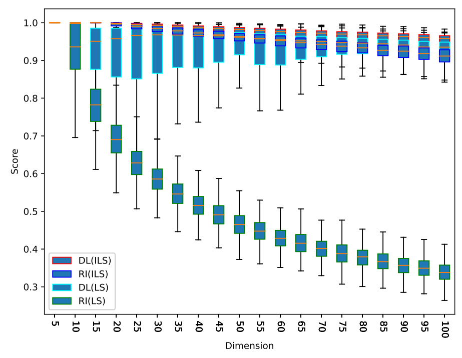
(c) Strongly correlated instances

Figure 4 (Color online) The scores of the random initial probability and the learned probability by the learned Set2Pro on the large-scale 0-1 knapsack problem benchmark. Specifically, the score of uniformly sampled (resp. learned) probability by hill climbing is denoted as 'RI(LS)' in green (resp. 'DL(LS)' in cyan) and by ILS as 'RI(ILS)' in blue (resp. 'DL(ILS)' in red)

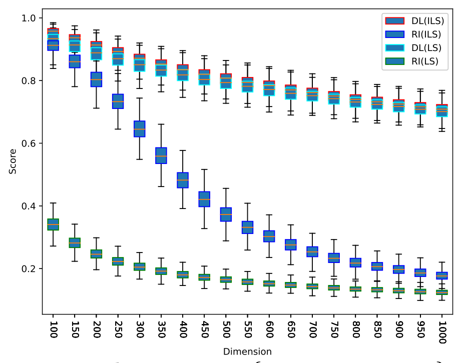
5.1.3 *Generalization*

Firstly, we investigate the generalization performance of the proposed deep learning model, Set2Pro, in dimension. To this end, we generate 1,000 knapsack problem instances in $\{5, 10, \ldots, 100, 150, \ldots, 1000\}$ dimensions, then employ the learned Set2Pro, trained on 15–30 dimensional KPs in Subsection 5.1.2, to predict corresponding preferable probability vectors. The boxplot (resp. average) of the scores obtained by hill climbing and ILS using DL initialization and RI are shown in Figure 5(a) and Figure 5(b) (resp. Figure 5(c) and Figure 5(d)), respectively.

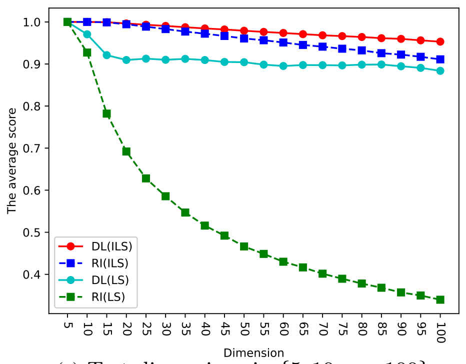
From Figure 5(a) and Figure 5(c), we observe that the average scores of the learned Set2Pro for hill climbing and ILS are both greater than those of the uniform sampling on KP instances up to 100 dimensions, respectively. As the scale increases up to 1,000 dimensions, in Figure 5(b) and Figure 5(d), the scores of the initial solution by RI for hill climbing and ILS are both less than 0.2, due to the sharp drop of ILS by randomness. However, it is worth noting that scores of Set2Pro with hill climbing and ILS stay around 0.8, and they just decline slightly within a narrow range over the dimensions.



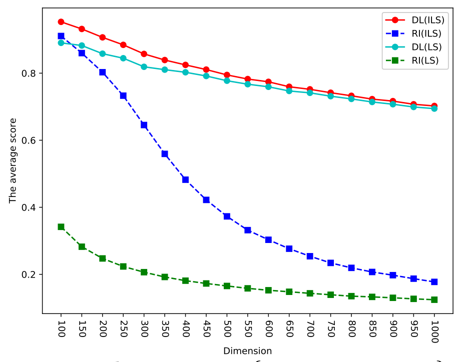
(a) Test dimensions in $\{5, 10, \dots, 100\}$



(b) Test dimensions in $\{100, 150, \dots, 1000\}$



(c) Test dimensions in $\{5, 10, \dots, 100\}$



(d) Test dimensions in $\{100, 150, \dots, 1000\}$

Figure 5 (Color online) All scores on KPs in $\{5, 10, \dots, 100, 150, \dots, 1000\}$ dimensions are shown in boxplots ((a) and (b)) and the average scores ((c) and (d)). The scores of the initial probabilities acquired by Set2Pro (resp. by the uniform distribution) and then used in ILS and in the simple hill climbing are designated as 'DL(ILS)' in red (resp. 'RI(ILS)' in blue) and 'DL(LS)' in cyan (resp. 'RI(LS)' in green), respectively. Boxplots of DL(ILS) in red and DL(LS) in cyan are overlapped and indistinguishable in (a) and (b)

We may conclude that the initial vectors sampled from the probability vector obtained by Set2Pro are promising in high-dimensional knapsack problems. This also indicates that learning from a less-capable LS is also beneficial for a more-capable LS.

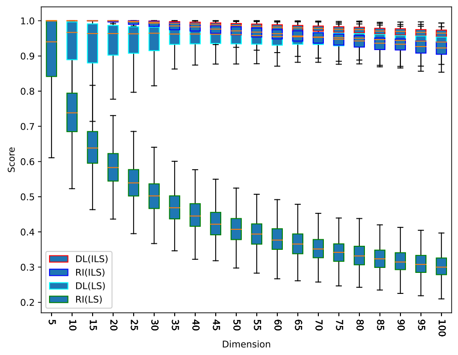
Secondly, we investigate how well Set2Pro generalizes to various KP instances with larger coefficients. We generate 1,000 knapsack problems with larger values $v \in \{1, 2, \dots, 60\}$ and larger weights $w \in \{1, 2, \dots, 40\}$ than those in the training KP instances in $\{5, 10, \dots, 100\}$ dimensions. The left (resp. right) plot of Figure 6 shows the boxplot of scores (resp. the average score) on KP instances with larger values and weights in each dimension.

From Figure 6, we see that Set2Pro with ILS exhibits the best performance on new instances across dimensions, which implies it can generalize well to KP instances with larger values and weights, which have not been seen in the training set.

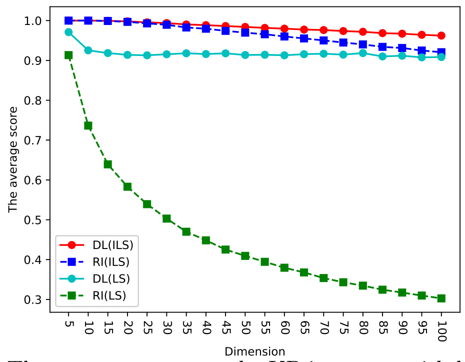
5.1.4 *Component analysis*

We carry out component analysis on the permutation equivariant layer in the Set2Pro network and the regularization term in the loss function.

Permutation equivariance. Here we randomly generate 10 knapsack problems per dimension in $\{5, 10, \dots, 100\}$, and then shuffle the order of all elements in each problem 100 times. The scores of the permuted instances on each dimension are shown in Figure 7. We can see that on the same problem with different permutations, Set2Pro can still achieve a preferable probability distribution than random

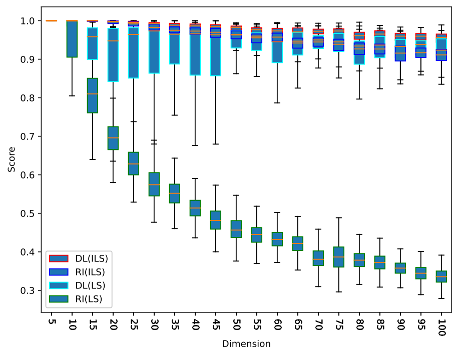


(a) The box plots of scores on the KP instances with larger values and weights

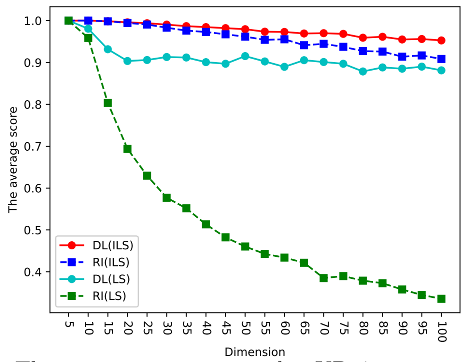


(b) The average scores on the KP instances with larger values and weights

Figure 6 (Color online) All scores by Set2Pro with the legend 'DL' (ILS in red and LS in cyan) and the uniformly distributed initial vector with the legend 'RI' (ILS in blue and LS in green) on the knapsack instances with various values and weights are summarized in boxplot (a) and the average scores (b)



(a) The box plots of scores on the KP instances after permuting the order of items



(b) The average scores on the KP instances after permuting the order of items

Figure 7 (Color online) All scores by Set2Pro with the legend 'DL' (ILS in red and LS in cyan) and the uniformly distributed initial vector with legend 'RI' (ILS in blue and LS in green) on the knapsack instances with various permutations are represented in boxplot (a) and summarized by the average scores (b)

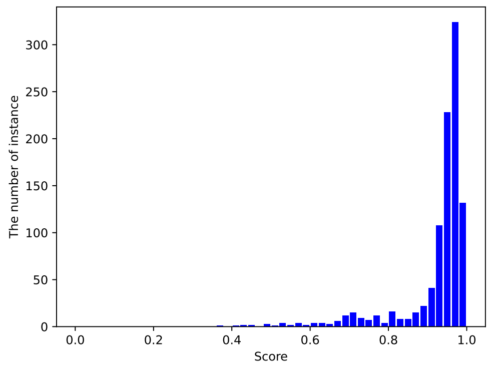
initialization in all dimensions in terms of a higher score. Furthermore, we can observe a significant improvement against hill climbing by the learned model.

Regularization. To validate the effectiveness of the regularization term, i.e., $\lambda\|\boldsymbol{q}\|_2^2$, on generating preferable $\boldsymbol{q}$ and improving the robustness of the training process, we compare it with the utility function without the regularization term, denoted as R_0.

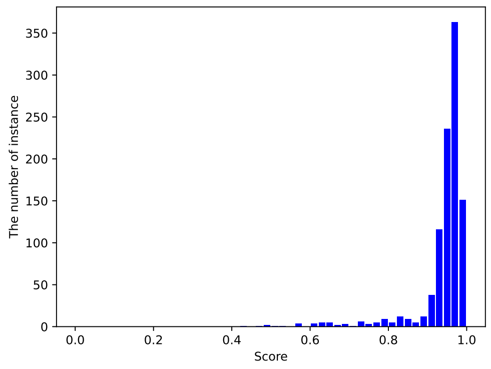
We generated 1,000 knapsack problems with dimension $n = 50$. For each problem instance, the values $\{v_i\}$ are random integers in $\{1, 2, \dots, 100\}$, and the weights $\{w_i\}$ are randomly selected in $\{1, 2, \dots, 20\}$. The final scores obtained by taking R_0 and R (see 4.10) as the utility function in Algorithm 1 are shown in Figure 8, which provides the number of KP instances as the score varies from 0 to 1 with the interval of 0.02. Table 2 provides the mean scores and the standard deviations by using R_0 and R in Algorithm 1.

From Figure 8, we can see that the majority of the scores on the KP instances fall in $[0.8, 1.0]$, which implies that the solution found by the initial probability generated by Algorithm 1 is fairly close to the optimal solution. From Table 2, we can also see that the initial solutions obtained by maximizing R have a higher mean score and a lower variance than that by R_0.

The statistics of the probability distribution obtained by running Algorithm 1 taking R_0 and R as



(a) The scores of the initial probability by R_0.



(b) The scores of the initial probability by R.

Figure 8 (Color online) The scores of the initial probability obtained by Algorithm 1 with R_0 (a) and R (b) are summarized in histogram. The x-axis is the acquired score from 0 to 1 with an interval of 0.2 and the y-axis represents the number of the KP instances

Table 2 The means and standard deviations of scores by Algorithm 1 with the utility functions R_0 and R

	Mean	Std. Dev.
R_0	0.9237	0.096
R	0.9408	0.073

Table 3 The statistics of the optimal initial probability distribution by Algorithm1 with the utility functions R_0 and R

	Median	Q3	> 0.8	> 0.9
R_0	6.40E−07	3.45E−04	356	221
R	2.70E−06	2.67E−03	2	2

reward, respectively, including the median and the 75-th percentile of all probabilities, and the number of items whose probability value is larger than 0.8 and 0.9, are summarized in Table 3.

From Table 3, we can see that there are many items with probability values larger than 0.9 obtained by Algorithm 1 running on R_0, which means that the probability vector moves and traps in some specific locations in the search space. In contrast, the utility function R, which includes a regularization term, can form a more uniform probability distribution. This indicates that such a utility function is able to capture more information and diverse features in various search areas. Therefore, by using such probability vectors as labels to train the DNN, it is more possible that a preferable probability distribution can be learned. In light of these observations, in the following, we take R as the utility function for the sake of robustness.

5.2 Experiments on the maximum clique problem

5.2.1 *Training settings*

The size of the hidden layer in GCN is $H = 64$, and the learning rate is 0.0001. Analogical to (5.2), the score in MCP is the ratio between the clique size found by Algorithm 3 beginning with the initial probability and the size of the largest clique found:

$$\text{score}_{\text{MCP}}(\boldsymbol{q}) = \mathbb{E}_{\boldsymbol{x}\sim\boldsymbol{q}} \frac{\|\boldsymbol{x}^*\|_1}{\|\boldsymbol{x}^{\text{best}}\|_1}, \tag{5.3}$$

where $\boldsymbol{x}^{\text{best}}$ is the maximum clique size by the backtracking tree and the branch and bound algorithm as proposed in [7], and $\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$.

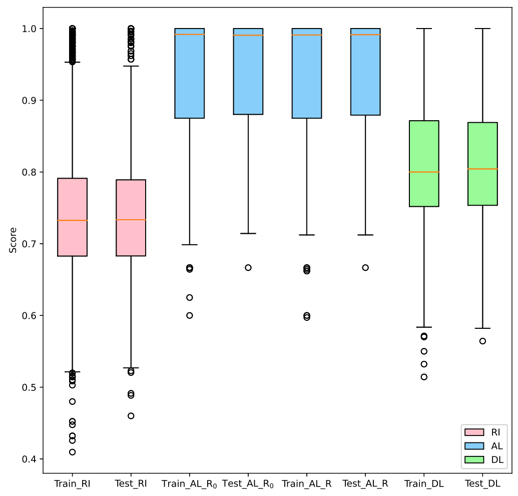


Figure 9 (Color online) The scores of the initial probabilities obtained by the uniform sampling designated as "rand_p" (resp. "rand_p_test") in the first (resp. second) column, by R_0 in Algorithm 1 as "train_p_R$_0$" (resp. "test_p_R$_0$") in the third (resp. forth) column, by R in Algorithm 1 as "train_p_R" (resp. "test_p_R") in the fifth (resp. sixth) column and by a two-layer GCN as "train set" (resp. "test set") in the seventh (resp. eighth) column on the training (resp. test) set

5.2.2 *Experimental results*

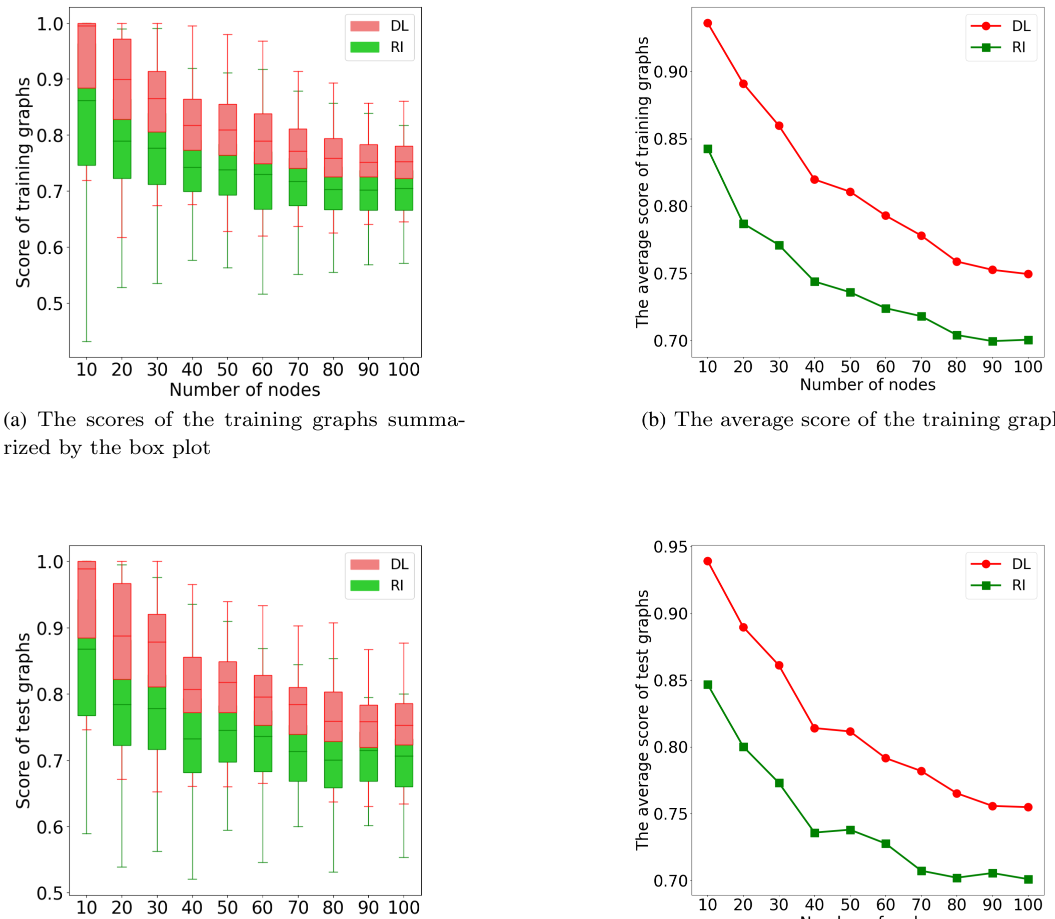
Synthetic graphs. To verify the proposed framework is adaptable to the graph-based DOPs, the proposed algorithms are examined for finding the maximum clique on the Erdős-Rényi (ER) random graph [18], which is a commonly-used graph generator [15]. Here we choose the $G(n,p), p \in [0,1]$ model without the restriction to the number of edges for gaining random graphs with diverse density. $G(n,p)$ is an undirected graph with n nodes, where every two nodes are joined by an edge with the probability p independently. The larger p is, the denser $G(n,p)$ becomes because more edges emerge.

We randomly synthesize totally 5,000 ER random graphs, with 100 graphs per $G(n,p)$, where the number of nodes $n \in \{10,20,30,40,50,60,70,80,90,100\}$ and the probability $p \in \{0.2,0.4,0.5,0.6,0.8\}$. In this subsection, the training (resp. test) set consists of 4,000 (resp. 1,000) graphs with 80 (resp. 20) graphs per $G(n,p)$. Compared with the uniform initial solution, all scores of the training and test sets obtained by R_0 and R in Algorithm 1 in terms of (5.3) are depicted in Figure 9, and the detailed scores in each size are shown in Figure 10.

From Figure 9, we can see that the median scores of the initial probabilities derived from Algorithm 1 by R_0 and R on the training and test sets are all very close to 1. They both perform the best. The score of the initial vector by R is slightly better than R_0, since the minimum of the boxplot by R on the training set is higher than that by R_0. Consequently, we use R in the following experiments. We can also see that the median scores by GCN are around 0.8 on both the training and test sets, which is better than the uniform initial probability.

Moreover, it is observed in Figure 10 that GCN achieves superior performance on both training and test graphs in all graph sizes, which reveals that the proposed method remains effective in gaining a preferable probability distribution, from which an initial solution is sampled for LS, to find a high-quality solution.

Large-scale graph. To evaluate the performance of the learned 2-layer GCN, which is trained above, on large-scale MCP instances, we randomly synthesize a total of 360 ER random graphs with 20 graphs per $G(n_{\text{test}}, p_{\text{test}})$, where $n_{\text{test}} \in \{200,300,400,500,600,700,800,900,1000\}$ and $p_{\text{test}} \in \{0.2,0.4\}$ as the test graph dataset. The scores on the large-scale MCP are shown in Figure 11 in terms of boxplot and the mean clique size.



(a) The scores of the training graphs summarized by the box plot

(b) The average score of the training graphs

(c) The scores of the test graphs summarized by the box plot

(d) The average score of the test graphs

Figure 10 (Color online) The scores of the probability by GCN to sample the initial solution (in red) and the scores of the uniform initial solution (in green) for MCP in terms of the boxplot ((a) and (c)) and the average ((b) and (d)), when the training and test graphs have the same number of node and the same probability

From Figure 11, we can see that, when the node of the graph increases to 1,000, the clique found by the learned GCN is slightly larger than the random uniform initialization in each test dimension.

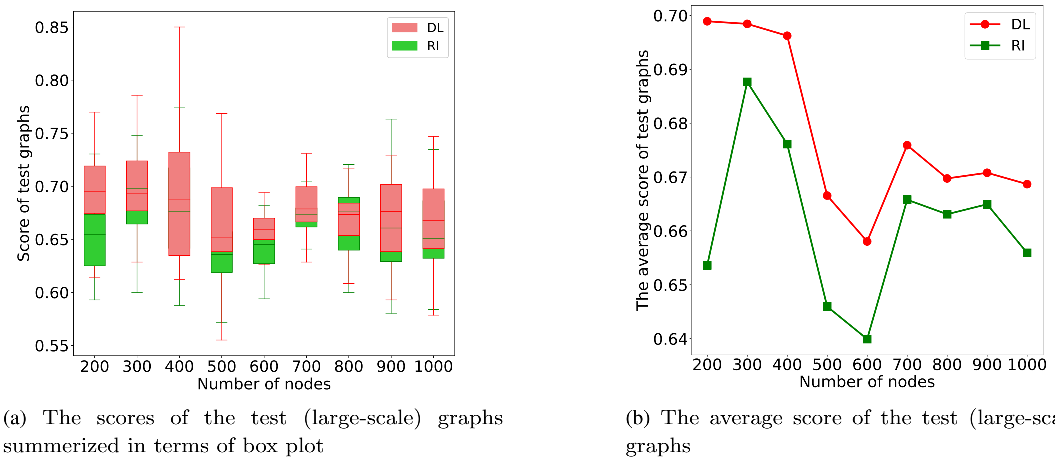
5.2.3 *Generalization*

As the network parameters, i.e., $\boldsymbol{H}^1$ and $\boldsymbol{H}^2$ in (4.12), are irrelevant to the graph size, the GCN can be trained and tested on graphs with different scales. Here we assess the generalization ability of GCN on (1) larger-size ER graphs, (2) denser graphs, and (3) larger as well as denser graphs, respectively. Accordingly, in this subsection, we generate 80 (resp. 20) ER random graphs per $G(n_{\text{train}}, p_{\text{train}})$ (resp. $G(n_{\text{test}}, p_{\text{test}})$) to compose the training (resp. test) set, which is set up as follows:

(1) The training and test graph sizes differs as $n_{\text{train}} \in \{10, 20, 30, 40, 50\}, n_{\text{test}} \in \{60, 70, 80, 90, 100\}$, while the probability values remain the same $p_{\text{train}}, p_{\text{test}} \in \{0.2, 0.4, 0.5, 0.6, 0.8\}$. There are 2,000 training graphs and 500 test graphs in total.

(2) The graph sizes are the same $n_{\text{train}}, n_{\text{test}} \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$, while the probability values varies as $p_{\text{train}} \in \{0.2, 0.4, 0.5\}$, $p_{\text{test}} \in \{0.6, 0.8\}$. There are a total of 2,400 training graphs and 400 test graphs.

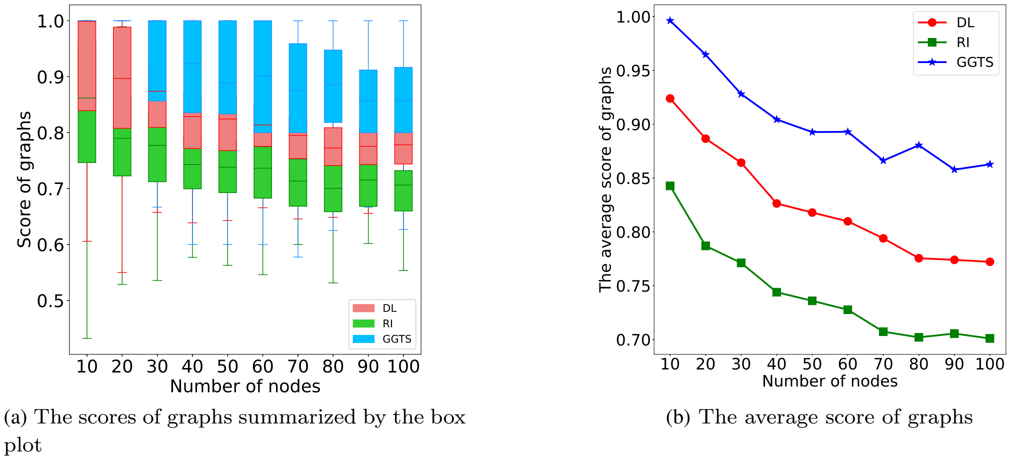
(3) Both the graph size and the probability change in the training and test sets as: $n_{\text{train}} \in$



(a) The scores of the test (large-scale) graphs summerized in terms of box plot

(b) The average score of the test (large-scale) graphs

Figure 11 (Color online) The scores of the probabilities predicted by GCN which are used to sample the initial solution (in red) and the scores of the uniform initial solution (in green) for MCP in terms of box plot (a) and the mean clique size (b), when the test graphs have a large size up to 1,000 nodes ($n_{\text{test}} \in \{200, 300, 400, 500, 600, 700, 800, 900, 1000\}$)



(a) The scores of graphs summarized by the box plot

(b) The average score of graphs

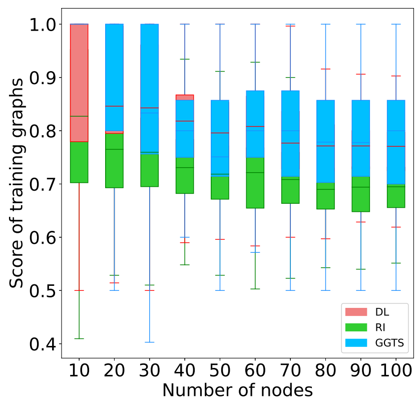
Figure 12 (Color online) The scores of the probability by GCN to sample the initial solution (in red), of the node subset by GGTS (in blue) and of the uniform initial solution (in green) for MCP in terms of the boxplot (a) and the average (b), when the test graphs have the larger sizes than the training graphs and the same probability as the training graphs ($n_{\text{test}} \in \{60, 70, 80, 90, 100\}$)

$\{10, 20, 30, 40, 50\}, p_{\text{train}} \in \{0.2, 0.4, 0.5\}; n_{\text{test}} \in \{60, 70, 80, 90, 100\}, p_{\text{test}} \in \{0.6, 0.8\}$, totally 1,200 training graphs and 200 test graphs.

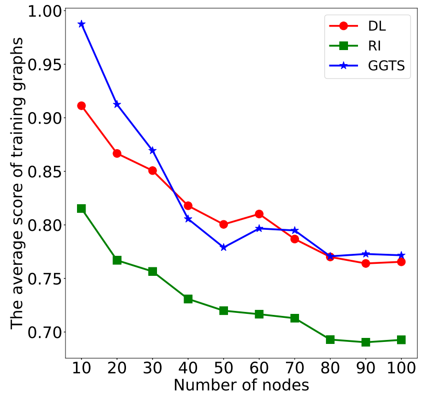
Since there is a paucity of research about the DL-based initialization for DOPs, we compare a DL-based combinatorial optimization method referred to as GGTS [39], which can directly provide a solution for graph-based DOPs, to evaluate the generalization ability of the proposed framework. When computing the score of GGTS, the solution obtained by GGTS is taken as the initial node subset and then we use Algorithm 3 to refine it to a maximal clique.

In comparison with the uniformly sampled initial solution and GGTS, the scores of the generated probability by GCN within the same duration of training time in the three cases above are shown in Figures 12–14, respectively. From these figures, we can see the following:

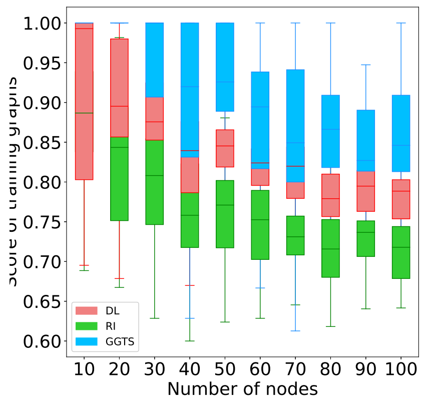
(1) generally the probabilities produced by GCN have higher scores than RI in all dimensions, even on



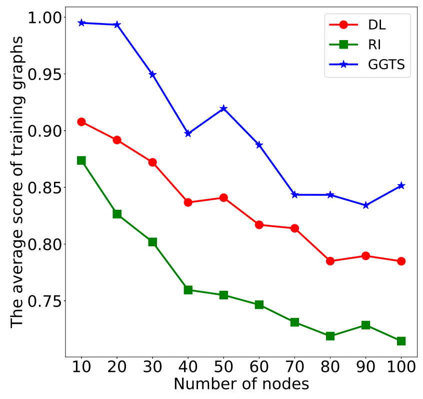
(a) The scores of the training graphs summarized by the box plot



(b) The average score of the training graphs



(c) The scores of the test (denser) graphs summarized by the box plot



(d) The average score of the test (denser) graphs

Figure 13 (Color online) The scores of the probability by GCN to sample the initial solution (in red), of the node subset by GGTS (in blue) and of the uniform initial solution (in green) for MCP in terms of the boxplot ((a) and (c)) and the average ((b) and (d)), when the test graphs are denser than the training graphs and they are of the same graph sizes ($p_{\text{test}} \in \{0.6, 0.8\}$)

the graphs with the larger scale and density, on the training and test graphs;

(2) when the test graph is denser than the training graph, there is no significant difference between DL and GGTS on the training graphs as the scale of the graph increases;

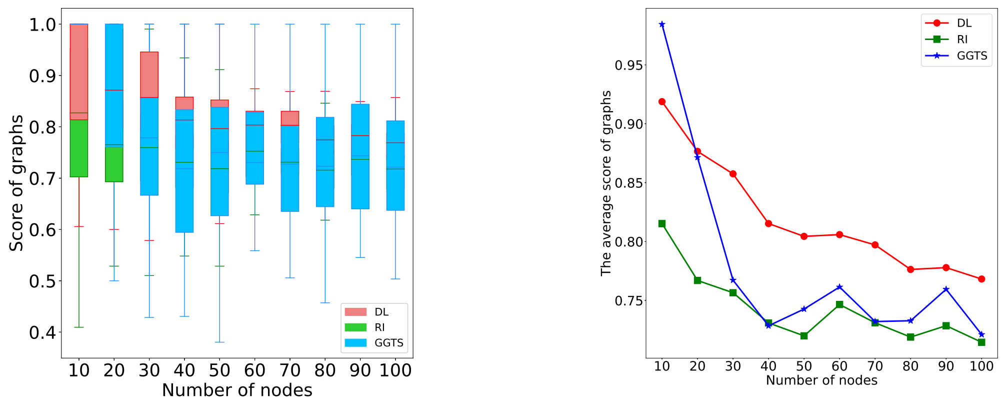
(3) when the test graphs have both larger sizes and densities, the proposed DL exhibits superior performance than both GGTS and RI on both training and test graphs in all graph sizes, except $n = 10$.

This indicates that the proposed model has a good generalization ability to the larger and denser graphs for the MCP.

5.2.4 *Deep-learning-based initialization for population-based heuristic*

In this subsection, we examine the effect of the learned probability vector on sampling the initial population for a population-based algorithm. Here we use a classical genetic algorithm, HGA [44] for solving the MCP. HGA requires a population of solutions as input, together with some algorithmic parameters. Here we use the same set of parameters as in the original reference.

To conduct the investigation, we divide the generation of the initial population into two parts. With a certain proportion, the initial population is generated by DL, while the rest is generated by RI. The



(a) The scores of graphs summarized by the box plot

(b) The average score of graphs

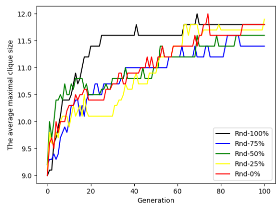
Figure 14 (Color online) The scores of the probability by GCN to sample the initial solution (in red), of the node subset by GGTS (in blue) and of the uniform initial solution (in green) for MCP in terms of the boxplot (a) and the average (b), when the test graphs are both larger and denser than the training graphs ($n_{\text{test}} \in \{60, 70, 80, 90, 100\}, p_{\text{test}} \in \{0.6, 0.8\}$)

Table 4 The average (with standard deviation) and the largest clique size by HGA with different percentages of random initialization. Mean(std): the average (standard deviation) value of the clique size in ten runs. Best: the size of the largest clique found in ten runs

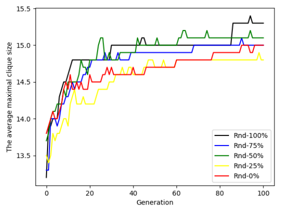
Graph	100%		75%		50%		25%		0%	
	Mean(std)	Best	Mean(std)	Best	Mean(std)	Best	Mean(std)	Best	Mean(std)	Best
brock200_2	11.8(0.6)	12.0	11.4(0.9)	12.0	11.6(0.8)	12.0	11.9(0.3)	12.0	11.8(0.6)	12.0
brock200_4	15.3(1.0)	17.0	15.0(0.4)	16.0	15.1(0.9)	17.0	14.8(0.6)	16.0	15.0(0.6)	16.0
brock400_2	22.0(0.6)	23.0	22.1(0.5)	23.0	23.1(0.3)	24.0	22.7(0.8)	24.0	22.4(0.7)	23.0
brock400_4	22.8(0.6)	24.0	22.3(0.5)	23.0	22.6(0.7)	24.0	22.4(0.9)	24.0	22.4(0.8)	24.0
C125.9	33.6(0.7)	34.0	34.0(0.0)	34.0	34.0(0.0)	34.0	33.7(0.5)	34.0	33.8(0.4)	34.0
C250.9	42.6(1.1)	44.0	41.9(0.9)	43.0	42.0(0.9)	44.0	41.5(0.8)	43.0	41.8(1.0)	43.0
C500.9	49.7(1.2)	52.0	51.0(1.1)	53.0	50.2(1.7)	53.0	50.2(1.8)	53.0	50.2(1.5)	52.0
DSJC500_5	11.9(0.5)	13.0	11.8(0.4)	12.0	12.0(0.0)	12.0	12.1(0.3)	13.0	12.1(0.8)	13.0
gen200_p0.9_44	40.2(3.0)	44.0	40.7(2.6)	44.0	39.8(2.2)	44.0	39.7(2.9)	44.0	40.1(2.3)	44.0
gen200_p0.9_55	55.0(0.0)	55.0	55.0(0.0)	55.0	55.0(0.0)	55.0	55.0(0.0)	55.0	55.0(0.0)	55.0
gen400_p0.9_55	47.4(1.3)	50.0	47.4(1.2)	49.0	47.0(1.2)	48.0	47.1(1.2)	50.0	47.0(0.9)	49.0
gen400_p0.9_65	49.5(5.3)	65.0	49.4(5.2)	65.0	54.6(8.7)	65.0	50.8(5.5)	65.0	53.3(7.2)	65.0
gen400_p0.9_75	69.9(10.2)	75.0	72.6(7.2)	75.0	67.3(11.8)	75.0	74.7(0.9)	75.0	66.0(11.8)	75.0
hamming8-4	16.0(0.0)	16.0	16.0(0.0)	16.0	16.0(0.0)	16.0	16.0(0.0)	16.0	16.0(0.0)	16.0
keller4	11.0(0.0)	11.0	11.0(0.0)	11.0	11.0(0.0)	11.0	11.0(0.0)	11.0	11.0(0.0)	11.0
MANN_a27	125.0(0.0)	125.0	125.0(0.0)	125.0	125.0(0.0)	125.0	125.0(0.0)	125.0	125.0(0.0)	125.0
p_hat300-1	7.8(0.4)	8.0	7.9(0.3)	8.0	7.8(0.4)	8.0	8.0(0.0)	8.0	8.0(0.0)	8.0
p_hat300-2	24.8(0.6)	25.0	25.0(0.0)	25.0	24.8(0.6)	25.0	25.0(0.0)	25.0	24.7(0.5)	25.0
p_hat300-3	34.2(0.7)	36.0	34.0(0.9)	36.0	34.1(0.8)	36.0	34.2(1.2)	36.0	33.7(0.6)	35.0

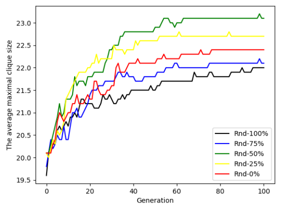
average (with standard deviation) size and the maximum size of the largest clique found in ten runs by using different numbers of random initial individuals in HGA are summarized in Table 4. In Table 4, the first (resp. second, third, fourth and fifth) set of columns reports the results in the case of all (resp. 75%, 50%, 25% and none of) random individuals in the initial population.

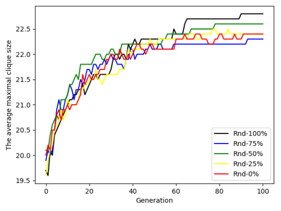
Figure 15 shows the size of the maximal clique found at each generation on 19 DIMACS graphs [28] by HGA with different proportions of random initial individuals. In the figure, the number following 'Rnd' refers to the percentage of the random initialization. From Table 4 and Figure 15, it is seen that the mean size of the maximal clique found by HGA with DL initialization is larger than that with RI on 9 out of 19 DIMACS graphs, including brock400_2, C125.9, C500.9, DSJC500_5, gen200_p0.9_44, gen400_p0.9_65, gen400_p0.9_75, p_hat300-1, and p_hat300-2. As for the other 10 graphs, there is no significant difference between the clique size found by HGA with DL and RI. We may conclude that the learned probability vector is beneficial for improving the quality of the solution in terms of the clique size by population-based algorithm on solving MCP.

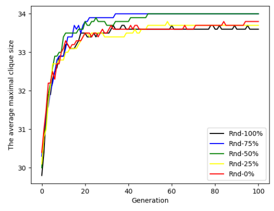


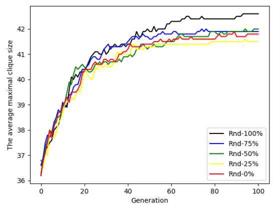
(a) brock200_2

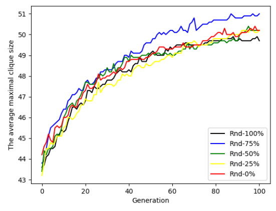

(b) brock200_4

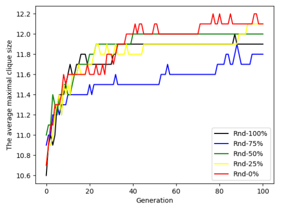

(c) brock400_2

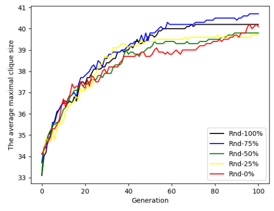

(d) brock400_4

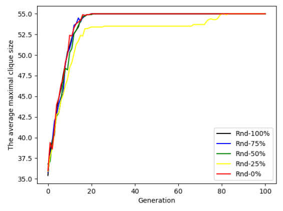

(e) C125.9

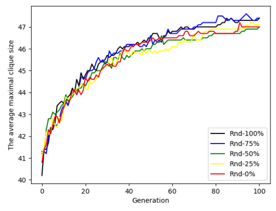

(f) C250.9

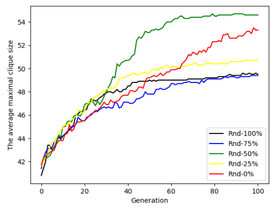

(g) C500.9

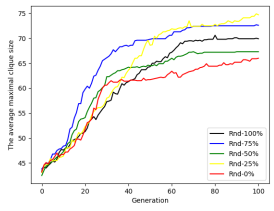

(h) DSJC500_5

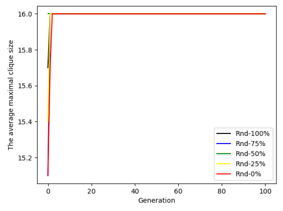

(i) gen200_p0.9_44

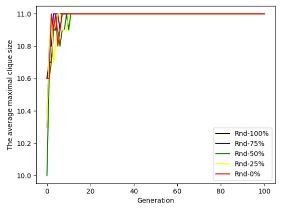

(j) gen200_p0.9_55

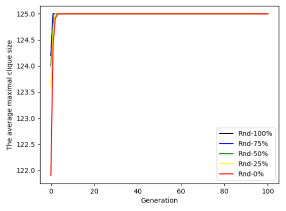

(k) gen400_p0.9_55

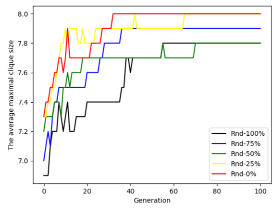

(l) gen400_p0.9_65

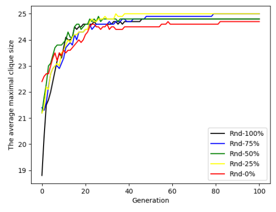

(m) gen400_p0.9_75


(n) hamming8-4


(o) keller4


(p) MANN_a27


(q) p_hat300-1


(r) p_hat300-2

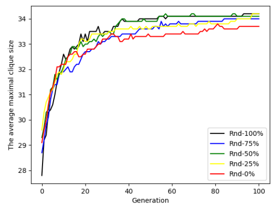

(s) p_hat300-3

Figure 15 (Color online) The size of the maximal clique found at each generation on DIMACS graphs by HGA, when some initial individuals are sampled from the learned probability and others are random ones. The number following 'Rnd' refers to the proportion of the random individuals in the initial population

6 Conclusion

In this paper, we proposed a framework to improve the performance of local search algorithms on 0-1 discrete optimization problems, by learning a preferable initial probability distribution and then sampling the initial solution from the learned probability. We first developed an algorithm to find a preferable probability distribution for a specific DOP instance. Then, a deep neural network model was employed to learn the mapping between the problem instance and the preferable distribution. The proposed framework was applied to solve two kinds of discrete problems established on set and graph: the knapsack problem and the maximum clique problem. We constructed a network called Set2Pro, which preserves equivariance for set-based discrete problems, for the knapsack problem and utilized a graph convolution network for the maximum clique problem, respectively, to learn the mapping. Experimental results showed that our framework yields a better solution than the uniformly sampled initial solution for both KP and MCP. We also observed that our framework can be well generalized to problems in different types and scales. Furthermore, the learned probability distribution is also beneficial to find a larger clique for a population-based method. In the future, we intend to learn the search method and enhance the capability for much larger and more complex graphs, as well as other discrete optimization problems over graphs and real-world problems.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 11991023 and 62076197) and Key Research and Development Project of Shaanxi Province (Grant No. 2022GXLH-01-15).

References

1 Abdel-Basset M, Abdel-Fatah L, Sangaiah A K. Metaheuristic algorithms: A comprehensive review. In: Sangaiah A K, Zhang Z, Sheng M, eds. Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications. New York: Academic Press, 2018, 185–231
2 Ansótegui C, Heymann, B, Pon J, et al. Hyper-reactive tabu search for MaxSAT. In: Proceedings of the International Conference on Learning and Intelligent Optimization. Berlin-Heidelberg: Springer, 2019, 309–325
3 Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. In: Proceedings of the 3rd International Conference on Learning Representations. New Orleans: OpenReview.net, 2015, 1–15
4 Bello I, Pham H, Le Q V, et al. Neural combinatorial optimization with reinforcement learning. In: Proceedings of the 5th International Conference on Learning Representations. New Orleans: OpenReview.net, 2017, 1–15
5 Bengio Y, Lodi A, Prouvost A. Machine learning for combinatorial optimization: A methodological tour d'horizon. Euro J Oper Res, 2021, 290: 405–421
6 Bertsekas D P. Dynamic Programming and Optimal Control: Volume I. Cambridge: Athena Scientific, 2012
7 Bron C, Kerbosch J. Algorithm 457: Finding all cliques of an undirected graph. Commun ACM, 1973, 16: 575–577
8 Bruna J, Zaremba W, Szlam A, et al. Spectral networks and locally connected networks on graphs. In: Proceedings of the 2nd International Conference on Learning Representations. New Orleans: OpenReview.net, 2014, 1–14
9 Cacchiani V, Iori M, Locatelli A, et al. Knapsack problems - an overview of recent advances. part I: Single knapsack problems. Comput Oper Res, 2022, 143: 105692
10 Chen Z, Liu J, Wang X, et al. On representing linear programs by graph neural networks. In: Proceedings of the 11th International Conference on Learning Representations. New Orleans: OpenReview.net, 2023, 1–29
11 Cobos C, Dulcey H, Ortega J, et al. A binary fisherman search procedure for the 0/1 knapsack problem. In: Proceedings of the Advances in Artificial Intelligence. Berlin-Heidelberg: Springer, 2016, 447–457
12 Cortez P. Local Search. Cham: Springerg, 2021
13 Crama Y, Kolen A W J, Pesch E J. Local Search in Combinatorial Optimization. Berlin-Heidelberg: Springer, 1995
14 Csirik J, Frenk J, Labbé M, et al. Heuristics for the 0-1 min-knapsack problem. Acta Cybernet, 1991, 10: 15–20
15 Dai H, Khalil E B, Zhang Y, et al. Learning combinatorial optimization algorithms over graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. San Francisco: Curran Associates, 2017, 6351–6361
16 Defferrard M, Bresson X, Vandergheynst P. Convolutional neural networks on graphs with fast localized spectral filtering. In: Proceedings of the 30th International Conference on Neural Information Processing Systems. San Francisco: Curran Associates, 2016, 3844–3852
17 Doerr B, Neumann F. A survey on recent progress in the theory of evolutionary algorithms for discrete optimization. ACM Trans Evo Learn Optim, 2021, 1: 1–43

18 Erdős P, Rényi A. On the Evolution of Random Graphs. Princeton: Princeton Univ Press, 2006

19 Ezugwu A E, Shukla A K, Nath R, et al. Metaheuristics: A comprehensive overview and classification along with bibliometric analysis. Art Intell Rev, 2021, 54: 4237–4316

20 Gasse M, Chételat D, Ferroni N, et al. Exact combinatorial optimization with graph convolutional neural networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. San Francisco: Curran Associates, 2019, 15554–15566

21 Glover F. Future paths for integer programming and links to artificial intelligence. Comput Oper Res, 1986, 13: 533–549

22 Goodfellow I, Bengio Y, Courville A. Optimization for Training Deep Models. Cambridge: MIT Press, 2016

23 Haghir Chehreghani M. Half a decade of graph convolutional networks. Nat Mach Intell, 2021, 4: 192–193

24 Hansen P, Mladenović N. Variable Neighborhood Search. Cham: Springer, 2018

25 Hottung A, Tanaka S, Tierney K. Deep learning assisted heuristic tree search for the container pre-marshalling problem. Comput Oper Res, 2020, 113: 104781

26 Hussein A, Gaber M M, Elyan E, et al. Imitation learning: A survey of learning methods. ACM Comput Sur, 2017, 50: 1–35

27 Jiang H, Li C-M, Manyà F. Combining efficient preprocessing and incremental MaxSAT reasoning for maxclique in large graphs. In: Proceedings of the Twenty-Second European Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2016, 939–947

28 Johnson D J, Trick M A. Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. Providence: Amer Math Soce, 1996

29 Karimi-Mamaghan M, Mohammadi M, Meyer P, et al. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. Euro J Oper Res, 2022, 296: 393–422

30 Khalil E B, Dilkina B, Nemhauser G L, et al. Learning to run heuristics in tree search. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2017, 659–666

31 Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. In: Proceedings of the 5th International Conference on Learning Representations. New Orleans: OpenReview.net, 2017, 1–14

32 Kirkpatrick S, Gelatt C D, Vecchi M P. Optimization by simulated annealing. Science, 1983, 220: 671–680

33 Kool W, van Hoof H, Welling M. Attention, learn to solve routing problems! In: Proceedings of the 7th International Conference on Learning Representations. New Orleans: OpenReview.net, 2019, 1–25

34 Korte B, Vygen J. Combinatorial Optimization. Berlin-Heidelberg: Springer, 2018

35 Kruber M, Lübbecke M E, Parmentier A. Learning when to use a decomposition. In: Proceedings of the Integration of AI and OR Techniques in Constraint Programming. Berlin-Heidelberg: Springer, 2017, 202–210

36 Land A H, Doig A G. An automatic method of solving discrete programming problems. Econometrica, 1960, 28: 497–520

37 Lemos H, Prates M, Avelar P, et al. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In: Proceedings of the 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI). San Francisco: IEEE, 2019, 879–885

38 Li X, Olafsson S. Discovering dispatching rules using data mining. J Schedul, 2005, 8: 515–527

39 Li Z, Chen Q, Koltun V. Combinatorial optimization with graph convolutional networks and guided tree search. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. San Francisco: Curran Associates, 2018, 537–546

40 Liu X, Sun J, Zhang Q, et al. Learning to learn evolutionary algorithm: A learnable differential evolution. IEEE Trans Emerg Top Comput Intell, 2023, 7: 1605–1620

41 Lodi A, Zarpellon G. On learning and branching: A survey. TOP, 2017, 25: 207–236

42 Louis S, McDonnell J. Learning with case-injected genetic algorithms. IEEE Trans Evol Comput, 2004, 8: 316–328

43 Mahmood R, Babier A, McNiven A, et al. Automated treatment planning in radiation therapy using generative adversarial networks. In: Proceedings of the 3rd Machine Learning for Healthcare Conference. Ann Arbor: PMLR, 2018, 484–499

44 Marchiori E. A simple heuristic based genetic algorithm for the maximum clique problem. In: Proceedings of the 1998 ACM Symposium on Applied Computing. New York: Association for Computing Machinery, 1998, 366–373

45 Marchiori E. Genetic, iterated and multistart local search for the maximum clique problem. In: Proceedings of the Applications of Evolutionary Computing. Berlin-Heidelberg: Springer, 2002, 112–121

46 Martí R, Reinelt G. Exact and Heuristic Methods in Combinatorial Optimization. Berlin: Springer, 2022

47 Mencarelli L, D'Ambrosio C, Di Zio A, et al. Heuristics for the general multiple non-linear knapsack problem. Electron Note Discret Math, 2016, 55: 69–72

48 Milan A, Rezatofighi S H, Garg R, et al. Data-driven approximations to np-hard problems. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2017, 1453–1459

49 Nasiri M M, Salesi S, Rahbari A, et al. A data mining approach for population-based methods to solve the JSSP. Soft Comput, 2019, 23: 11107–11122

50 Nazari M, Oroojlooy A, Takáč M, et al. Reinforcement learning for solving the vehicle routing problem. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. San Francisco: Curran Associates, 2018, 9861–9871

51 Peres F, Castelli M. Combinatorial optimization problems and metaheuristics: Review, challenges, design, and development. Appl Sci, 2021, 11: 1–39

52 Pisinger D. Where are the hard knapsack problems? Comput Oper Res, 2005, 32: 2271–2284

53 Rossi R A, Gleich D F, Gebremedhin A H. Parallel maximum clique algorithms with applications to network analysis. SIAM J Sci Comput, 2015, 37: C589–C616

54 San Segundo P, Lopez A, Pardalos P M. A new exact maximum clique algorithm for large and massive sparse graphs. Comput Oper Res, 2016, 66: 81–94

55 Selsam D, Lamm M, Bünz B, et al. Learning a SAT solver from single-bit supervision. In: Proceedings of the 7th International Conference on Learning Representations. New Orleans: OpenReview.net, 2019, 1–11

56 Sergienko I V, Shylo V P. Problems of discrete optimization: Challenges and main approaches to solve them. Cybernet Syst Anal, 2006, 42: 465–482

57 Stützle T, Ruiz R. Iterated Local Search. New York-Berlin: Springer, 2018

58 Sun J, Liu X, Bäck T, et al. Learning adaptive differential evolution algorithm from optimization experiences by policy gradient. IEEE Trans Evo Comput, 2021, 25: 666–680

59 Sutton R S, Barto A G. Reinforcement Learning: An Introduction. Cambridge: MIT press, 2018

60 Tahami H, Fakhravar H. A literature review on combining heuristics and exact algorithms in combinatorial optimization. Euro J Inform Tech Comput Sci, 2022, 2: 6–12

61 Talbi E-G. Machine learning into metaheuristics: A survey and taxonomy. ACM Comput Surv, 2021, 54: 1–32

62 Toth P. Dynamic programming algorithms for the zero-one knapsack problem. Computing, 1980, 25: 29–45

63 Toyer S, Trevizan F W, Thiébaux S, et al. Action schema networks: Generalised policies with deep learning. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence. Palo Alto: AAAI Press, 2018, 6294–6301

64 Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. New York: Association for Computing Machinery, 2017, 6000–6010

65 Vince A. A framework for the greedy algorithm. Discrete Appl Math, 2002, 121: 247–260

66 Vinyals O, Fortunato M, Jaitly N. Pointer networks. In: Proceedings of the 28th International Conference on Neural Information Processing Systems. New York: Association for Computing Machinery, 2015, 2692–2700

67 Wu Q, Hao J-K. A review on algorithms for maximum clique problems. Euro J Oper Res, 2015, 242: 693–709

68 Yang X-S. Nature-Inspired Optimization Algorithms. New York: Academic Press, 2021

69 Zaheer M, Kottur S, Ravanbakhsh S, et al. Deep sets. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. New York: Association for Computing Machinery, 2017, 3391–3401

70 Zhou J, Cui G, Hu S, et al. Graph neural networks: A review of methods and applications. AI Open, 2020, 1: 57–81