Learning to Design Greedy Algorithm for NP-Complete Problems

Zhenxin Ding[1,3], Zihao Huang[2,3], Jingyan Sui[1,3], Ruizhi Liu[1,3], Shizhe Ding[1,3], Liming Xu[1,3], Chao Wang[1,3], Haicang Zhang[1,3], Chungong Yu[1,3], and Dongbo Bu[1,3(✉)]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
dbu@ict.ac.cn
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China

Abstract. Algorithm design is an art that heavily requires human designers' intuition, expertise, and insights into the combinatorial structure of the problems under consideration. In particular, while greedy algorithms for NP-complete problems are generally straightforward, finding an effective greedy-selection rule is consistently challenging. In the study, we introduce an approach called *artificial intelligence algorithmist for set cover problem* (AIA-SC), which leverages neural networks as greedy-selection rules to solve the minimum weighted set cover problem (WSCP), an NP-complete problem. Initially, we formulate a given WSCP as a 0–1 integer linear program (ILP), in which each variable x_i is binary: $x_i = 0$ indicating the exclusion of set s_i, and $x_i = 1$ indicating the selection of the set. Subsequently, we design a generic search framework to identify the optimal solution to the ILP. At each search step, the value of a variable is determined with the aid of neural networks. The key ingredients of our neural network involve the design of its loss function and training process: the original ILP problem and the sub-problems generated by assigning a variable x_i should satisfy the Bellman-Ford equation, which enables us to set the violation of Bellman-Ford equation as loss function of our neural network. Experimental results on representative instances indicate that the neural network-based greedy selection rule effectively identifies optimal solutions and surpasses the performance of the human-crafted Chvatal's greedy algorithm. Furthermore, the basic idea of our approach can be readily extended to design greedy algorithms for other NP-hard problems without significant modification. The source code of AIA-SC and data sets are available at https://github.com/zxding94/aia-sc.

Keywords: Algorithm Design · Greedy-selection Rules · Neural Networks · Minimum Weighted Set Cover Problem

Z. Ding and Z. Huang—Contributed equally to this study.

Y. Yin et al. (Eds.): NCTCS 2024, CCIS 2354, pp. 30–43, 2025.
https://doi.org/10.1007/978-981-96-1490-5_3

1 Introduction

NP-complete problems, the hardest ones within the NP class, are characterized by the property that the solutions to these problems can be verified in polynomial time, yet no polynomial-time algorithms have been found to solve them efficiently [1]. These problems encompass a wide range of practical applications, including strategic planning, production planning, facility locating, and various scheduling and routing problems [2]. Consequently, the quest for efficient solving algorithms for NP-complete problems remains a paramount pursuit.

The exact algorithms for NP-complete problems, like branch-and-bound and branch-and-cut, exhibit promising performance in some cases but suffer from the limitation in the size of instances that these algorithms can solve. Significant efforts have been devoted to designing heuristic and meta-heuristic approaches to efficiently find optimal or nearly optimal solutions for large instances within reasonable computing time. Prominent among these heuristic techniques are genetic algorithms, ant colony optimization, simulated annealing, and tabu search [3].

The breakthrough of deep learning techniques has led to a growing interest in utilizing these techniques to solve NP-complete problems. However, it is challenging to tackle NP-complete problems using an end-to-end neural network with supervised learning due to the following reasons: *i*) Traditional algorithms and mathematical methods possess a well-established theoretical foundation, while neural networks lack a comparable insight into the combinatorial structure of problems. *ii*) The practical instances of NP-complete problems often exhibit distinct characteristics, which may not align well with the typical data requirements of deep learning models. *iii*) Deep learning models often demand a substantial amount of labeled data under specific distributions, whereas many combinatorial optimization problems, particularly NP-complete problems, necessitate computationally expensive calculations to derive optimal solutions [4].

In the study, we present an approach (called *artificial intelligence algorithmist for set cover problem*, AIA-SC) to learn algorithm design with the aid of neural networks. The specific goal of this paper is to use machine learning to find greedy rules and then design an efficient and practical greedy algorithm for the weighted set cover problem. Experimental results on multiple datasets demonstrate that the proposed AIA approach outperforms human-designed greedy algorithms, such as the Chvatal's greedy algorithm [5], by achieving superior solutions.

2 Related Works and Background

The algorithms to solve combinatorial optimization problems can be divided into exact and approximate/heuristic algorithms. Exact algorithms are known for their capability to guarantee optimal solutions. However, these exact algorithms become computationally intractable as the size of problem instances increases, rendering them unsuitable for handling large instances. In contrast, approximate/heuristic algorithms trade the quality of solution for increased computational efficiency by applying heuristic rules [6].

The design of effective heuristic rules relies heavily on insights into the problem characteristics and problem-solving process. The efficacy of different heuristic rules and their associated parameters varies across different datasets and solution stages. Thus, data-driven machine learning techniques, including supervised learning and reinforcement learning, have emerged as promising strategies for designing effective heuristic rules and algorithms to solve NP-complete combinatorial optimization problems [7–9]. These efforts are briefly described below.

Supervised learning has shown promising performance in aiding the design of the solver for integer linear programming. For example, Alvarez et al. [10] employed a specialized decision tree to learn the branch variable selection strategy of the strong branch technique [11]. Khalil et al. [12] trained a support vector machine on the choice of strong branching technique collected on some sub-problems and then used this support vector machine to guide variable selection on other sub-problems. Gasse et al. [13] utilized a graph convolutional neural network to learn the choice of strong branching strategies by extracting information on variables and constraints. He et al. [14] devised a machine learning algorithm to identify the sub-problem that contains the optimal solution. To speed up solving the traveling salesman problem, Chaitanya et al. predicted the probabilities for each edge appearing in the optimal tour by using a deep graph network [15].

Unlike supervised learning, reinforcement learning does not require calculating labels in advance, which is time-consuming and practically infeasible for NP-complete problems. This property makes reinforcement learning promising for designing algorithms for NP-complete problems. For example, to solve the traveling salesman problem (TSP), Bello et al. trained a network using the total distance of a tour as reward [16]. To extract the essential information from the given TSP instance, Kool et al. incorporated a graph neural network into the reinforcement learning framework [17], whereas Xavier et al. utilized a transformer together with beam search for decoding [18].

Together, these achievements suggest that the use of machine learning techniques paves a new way to understand the combinatorial structure of the problem under consideration, and subsequently design suitable algorithms to solve the problem.

3 Method

In the study, we focus on the weighted set cover problem (WSCP) as it is a representative NP-complete problem. The basic idea of our approach to WSCP is as follows: we first transform the solving process into a multi-step decision-making process and then design a neural network to learn the optimal decision at each step. The decisions thus acquired constitute a solution to WSCP. The key components of our approach involve the training process and the loss function. The loss function exploits the recursion between consecutive decisions rather than the difference between the neural network's output and the optimal solution as finding the optimal solution itself is our objective and is usually time-consuming.

We describe the multi-step decision-making process, network architecture, loss function, and training procedure below.

3.1 Multi-step Decision-Making Process to Solve WSCP

The weighted set cover problem can be described as follows: given a set of m elements (called the universe U) and a collection of n weighted subsets of U, the goal is to identify a sub-collection of sets such that: i) the universe U should be covered, i.e., the union of the sub-selection equals U, and ii) the weight-sum of the sub-collection is minimized. We formulate the problem as the following 0-1 integer linear program:

$$\begin{aligned} \min\ & z = \boldsymbol{c}^T \boldsymbol{x} \\ s.t. & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\ & x_j = \text{0-1} \quad (j = 1, 2, \cdots, n). \end{aligned} \tag{1}$$

Here, x_j is a binary variable such that $x_j = 1$ if the j-th set is selected into the sub-collection and $x_j = 0$ otherwise. The constant c_j denotes the weight of the j-th set. Thus, the objective function $\boldsymbol{c}^T\boldsymbol{x}$ represents the sum weight of the sub-collection. In the constraints, $\boldsymbol{A}$ denotes an $m \times n$ matrix in which A_{ij} is 1 if the j-th set covers the i-th element and 0 otherwise. To guarantee that the identified collection of subsets covers all elements, we initialize $\boldsymbol{b}$ as $\boldsymbol{b} = \boldsymbol{1}$ and appropriately update it in subsequent steps.

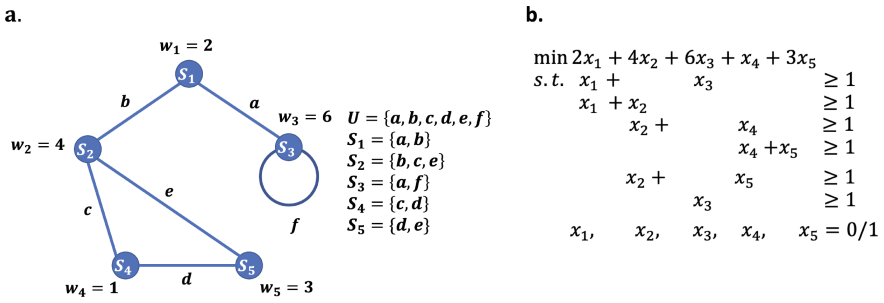

Fig. 1. Weighted set cover problem and its formulation. (a) An example with five subsets over a universe with six elements. (b) Integer linear program for this instance

We further formulate the solving process of the linear program as a multi-step decision-making process: at the k-th step, we attempt to decide whether $x_k = 0$ or 1 with the variables $x_1, \cdots, x_{k-1}$ already determined at previous steps. The problem $\mathcal{P}$ to be solved at the k-th step is:

$$\begin{aligned} \min\ & z = \boldsymbol{c}_k^T \boldsymbol{x}_k \\ s.t. & \boldsymbol{A}_k \boldsymbol{x} \geq \boldsymbol{b}_k \\ & x_j = \text{0-1}, \quad j = k, k+1 \cdots, n. \end{aligned} \tag{2}$$

Here, $\boldsymbol{x}_k = [x_k\ x_{k+1} \cdots x_n]$ denote the variables, the parameters include $\boldsymbol{c}_k = [c_k\ c_{k+1} \cdots c_n]^T$, $\boldsymbol{b}_k = \boldsymbol{b} - x_1\alpha_1 - \cdots - x_{k-1}\alpha_{k-1}$, and $\boldsymbol{A}_k = [\alpha_k\ \alpha_{k+1} \cdots \alpha_n]$ in which α_j denotes the j-th column of the matrix $\boldsymbol{A}$. Hereafter, $\mathcal{P}$ and $(\boldsymbol{A}_k, \boldsymbol{b}_k, \boldsymbol{c}_k)$ will be interchangeably used to represent the problem. The optimal objective value of this problem is denoted as $\mathrm{OPT}(\boldsymbol{A}_k, \boldsymbol{b}_k, \boldsymbol{c}_k)$ or $\mathrm{OPT}(\mathcal{P})$ (Fig. 2).

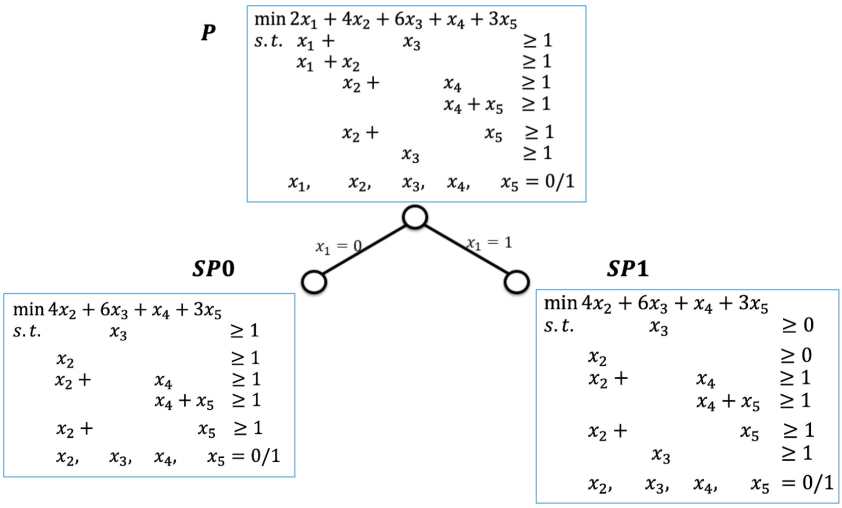

Fig. 2. Multi-step decision-making process to solve the weighted set cover problem. Here, we show the first decision $x_1 = 1$ (selecting subset S_1) and $x_1 = 0$ (abandoning subset S_1). Each option corresponds to a sub-problem of the original problem

The decision $x_k = 1$ will derive a sub-problem of $\mathcal{P} = (\boldsymbol{A}_k, \boldsymbol{b}_k, \boldsymbol{c}_k)$ to be solved at the $k+1$-th step, denoted as $\mathcal{SP}_1 = (\boldsymbol{A}_{k+1}, \boldsymbol{b}_k - \alpha_k, \boldsymbol{c}_{k+1})$. Similarly, the decision $x_k = 0$ will derive another sub-problem denoted as $\mathcal{SP}_0 = (\boldsymbol{A}_{k+1}, \boldsymbol{b}_k, \boldsymbol{c}_{k+1})$. The key observation is that the optimal objective values of a problem $\mathcal{P}$ and its two sub-problems $\mathcal{SP}_0, \mathcal{SP}_1$ satisfy the following recursion:

$$\mathrm{OPT}(\mathcal{P}) = \min\{\mathrm{OPT}(\mathcal{SP}_0), c_k + \mathrm{OPT}(\mathcal{SP}_1)\}. \tag{3}$$

Reversely, we can make correct decision on x_k if we know the optimal solutions $\mathrm{OPT}(\mathcal{SP}_0)$ and $\mathrm{OPT}(\mathcal{SP}_1)$, i.e.,

$$x_k = \begin{cases} 0 & \text{if } \mathrm{OPT}(\mathcal{SP}_0) \leq c_k + \mathrm{OPT}(\mathcal{SP}_1), \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

3.2 AIA-SC Approach

Equation 4 implies an approach to solving WSCP: we sequentially apply this equation on $x_1, x_2, ..., x_n$, thus acquiring the correct decisions on these variables

that constitute the complete solution. This approach is feasible if can exactly calculate, or accurately estimate, the optimal objective values $\mathrm{OPT}(\mathcal{SP}_0)$ and $\mathrm{OPT}(\mathcal{SP}_1)$. For this aim, we design a neural network (denoted as g_θ with θ representing the parameters) to estimate the optimal objective value. The key ingredients of our approach involve the construction of training data, and the design of loss function suitable for learning the objective values for NP-complete problems, which are described in more detail below.

Loss Function: The general loss function, say mean squared error, considers the difference between the neural network prediction $g_\theta(\mathcal{P})$ and the ground-truth value of the objective function $\mathrm{OPT}(\mathcal{P})$. Although widely used, this design strategy of loss function is infeasible as it relies on the ground-truth objective values that are difficult to calculate for NP-complete problems. In addition, it has been reported that a small change of $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}$ might lead to a considerable shift in objective value [19], thus making it challenging to use these values for training.

To overcome this challenge, we resort to the recursive relationship stated in Eq. 3 with the inspiration from Yang et al. [20]. The underlying rationale is that the neural network predictions $g_\theta(\mathcal{P})$ will also satisfy the recursion relationship if they perfectly approximate the ground-truth values $\mathrm{OPT}(\mathcal{P})$, i.e.,

$$g_\theta(\mathcal{P}) = \min\{g_\theta(\mathcal{SP}_0), c_k + g_\theta(\mathcal{SP}_1)\}. \tag{5}$$

This fact enables us to use the violation of the recursion as loss function:

$$\mathcal{L}_\theta = (g_\theta(\mathcal{P}) - \min\{g_\theta(\mathcal{SP}_0), c_k + g_\theta(\mathcal{SP}_1)\})^2. \tag{6}$$

Intuitively, this loss function uses $\min\{g_\theta(\mathcal{SP}_0), c_k + g_\theta(\mathcal{SP}_1)\}\}$ instead of ground-truth value $\mathrm{OPT}(\mathcal{P})$, therefore completely avoiding calculating this value, which is practically infeasible for NP-complete problems.

It should be noted that Eq. 3 does not hold if one of the two sub-problems is infeasible. To amend this deficiency, we pose an extra restriction that $g_\theta(\mathcal{SP}_0)$ is sufficiently large if $\mathcal{SP}_0$ is infeasible (and so is $g\theta(\mathcal{SP}_0)$). We implement this restriction by augmenting the loss function with a ReLU activation function. The augmented loss function is:

$$\mathcal{L}_\theta^{aug} = \begin{cases} (g_\theta(\mathcal{P}) - c_k - g_\theta(\mathcal{SP}_1))^2 + \mathrm{ReLU}(g_\theta(\mathcal{SP}_1) + c_k - g_\theta(\mathcal{SP}_0)) & \text{if } \mathcal{SP}_0 \text{ is infeasible} \\ (g_\theta(\mathcal{P}) - g_\theta(\mathcal{SP}_0))^2 + \mathrm{ReLU}(g_\theta(\mathcal{SP}_0) - g_\theta(\mathcal{SP}_1) - c_k) & \text{if } \mathcal{SP}_1 \text{ is infeasible} \\ (g_\theta(\mathcal{P}) - \min\{g_\theta(\mathcal{SP}_0), c_k + g_\theta(\mathcal{SP}_1)\})^2 & \text{otherwise.} \end{cases}$$

Here, ReLU nuron is used to force an infeasible sub-problem's predicted objective value to be larger than feasible sub-problems. In this study, a sub-problem is considered infeasible if one or more of its constraints are violated.

Network Architecture: We employ a two-layer fully-connected neural network to predict the optimal function value $g_\theta(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ of a WSCP instance $\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}$.

The input layer consists of $m \times n + m + n$ input nodes while the hidden layer consists of $m + n$ nodes.

Note that some sub-problems might have negative $\boldsymbol{b}_k$ as $\boldsymbol{b}_k = \boldsymbol{b} - x_1\alpha_1 - x_2\alpha_2 \cdots - x_{k-1}\alpha_{k-1}$ is calculated through subtracting certain α_i from $\boldsymbol{b}$. These negative $\boldsymbol{b}_k$ make the sub-problem differ from the canonical form in which all b_i might be 0 or 1 and thus incur difficulties in predicting the optimal function value for this problem. To circumvent these difficulties, we transform the negative b_i into 0 by applying a ReLU layer onto b_i. It should be noted that this transformation does not change the optimal function value of the problem.

Training Process: To train the neural network used by AIA-SC, we first prepare an instance-specific training dataset constructed specifically from the given instance $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ with network parameters set as θ. The dataset consists of a collection of sub-problem triples with the form $(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1)$, in which $\mathcal{SP}_0$ and $\mathcal{SP}_1$ are sub-problems of $\mathcal{P}$ (Fig. 3).

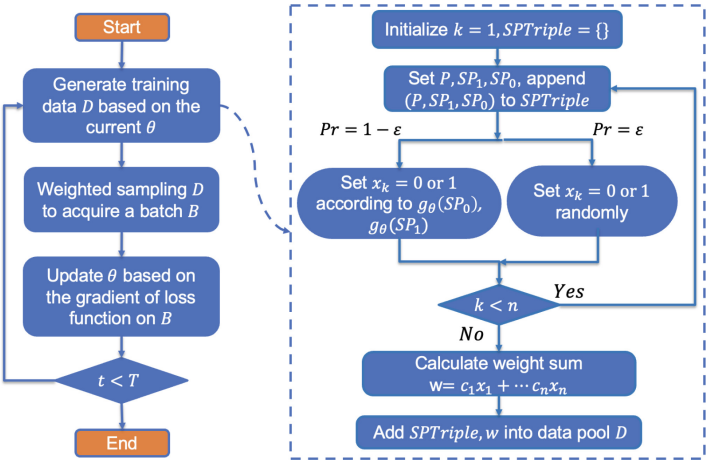

Fig. 3. Training process used by AIA-SC. A collection of sub-problem triples $(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1)$ are first generated as training data, and each triple is assigned with a weight

Specifically, we start from $\mathcal{P} = (\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$, enumerate its two sub-problems $\mathcal{SP}_0$ and $\mathcal{SP}_1$, and insert the triple $(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1)$ into the training dataset. Next, we fix x_1 by distinguishing three cases: *i*) we fix $x_1 = 1$ if $\mathcal{SP}_0$ is infeasible, *ii*) we fix $x_1 = 0$ if $\mathcal{SP}_1$ is infeasible, and *iii*) when both $\mathcal{SP}_0$ and $\mathcal{SP}_1$ are feasible, we fix $x_1 = 0$ if $g_\theta(\mathcal{SP}_0) < c_1 + g_\theta(\mathcal{SP}_1)$ and $x_1 = 1$ otherwise. We further update $\mathcal{P}$ according to the value of x_1. Similarly, we fix $x_2, x_3, \cdots, x_n$ and acquire $n-1$ more triples accordingly. Each triple $(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1)$ is associated with a weight $w = c_1x_1 + c_2x_2 + \cdots + c_nx_n$. To increase the diversity of training data and explore

more combinations of variables, we introduce randomness into data preparation, i.e., with probability ϵ we fix $x_i = 1$ and 0 uniformly at random.

Then, we sample the training dataset to acquire a batch B of triples. Here, we employ weighted sampling to emphasize the triples with high weight as these triples are more critical to train the neural network. We calculate the gradient of the loss function $\nabla \mathcal{L}_\theta^{aug}$ for all sampled triples and then use it to optimize the network parameters θ. The pseudocode of the training process is presented in Algorithm 1.

Algorithm 1. Training procedure of the neural network for AIA-SC

function TrainingNetwork($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, n$)
  Initialize network parameters θ^0 with random values, data pool $D = \emptyset$, learning rate η;
  for $t = 1$ to T **do**
    $SPTriples, w \leftarrow$ GeneratesubproblemTriplesAndWeight($\theta^{t-1}, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, n$);
    Append D with all sub-problem triples extracted from $SPTriples$ and associate each triple with a weight w;
    Generate a batch B of training data through weighted sampling D;
    for each $(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1) \in B$ **do**
      $\theta^t \leftarrow \theta^{t-1} - \eta \nabla \mathcal{L}_{\theta^{t-1}}^{aug}(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1)$;
    end for
  end for
  return θ^T;
end function
function GeneratesubproblemTriplesWithWeight($\theta, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, n$)
  Initialize $SPTriples = \emptyset$;
  for $k = 1$ to n **do**
    Set $\mathcal{P} = (\boldsymbol{A}_k, \boldsymbol{b}_k, \boldsymbol{c}_k)$;
    Set $\mathcal{SP}_0 = (\boldsymbol{A}_{k+1}, \boldsymbol{b}_k, \boldsymbol{c}_{k+1})$;
    Set $\mathcal{SP}_1 = (\boldsymbol{A}_{k+1}, \boldsymbol{b}_k - \alpha_k, \boldsymbol{c}_{k+1})$;
    Append $SPTriples$ with the triple $(\mathcal{P}, \mathcal{SP}_0, \mathcal{SP}_1)$;
    With probability ϵ set $x_k = 0$ or 1 randomly;
    With probability $1 - \epsilon$
      set $x_k = 1$ if $\mathcal{SP}_0$ is infeasible;
      set $x_k = 0$ if $\mathcal{SP}_1$ is infeasible;
      set $x_k = 0$ if $g_\theta(\mathcal{SP}_0) < c_k + g_\theta(\mathcal{SP}_1)$ and 1 otherwise;
  end for
  Calculate the weight sum $w = c_1 x_1 + \cdots + c_n x_n$;
  return $SPTriples$, w
end function

Once we have trained the neural network g_θ, we finally decide each variable based on the network predictions for the corresponding sub-problems, i.e., we decide $x_k = 1$ if $g_\theta(\mathcal{SP}_0) < c_1 + g_\theta(\mathcal{SP}_1)$ and $x_k = 0$ otherwise. The pseudo-code is presented in Algorithm 2.

4 Results

4.1 Data Set and Experiment Setting

Following the approach used by Balas and Ho [21], we randomly generated weighted set coverage problems as datasets in two scales, including:

(1) WSCP20: WSCP20 consists of 100 instances and each instance has a total of 20 subsets over a universe U ($|U| = 20$). Here, the elements of each subset were randomly selected from U. Following the convention [21], we set the density as 0.1, i.e., the total number of elements of these subsets is $0.1 \times 20 \times 20 = 40$.
(2) WSCP50: WSCP50 consists of 10 instances and each instance has a total of 50 subsets over a universe U ($|U| = 50$). The density is also set as 0.1.

Algorithm 2. AIA-SC algorithm

function AIA-SC($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, n$)
  $\theta =$TrainingNetwork($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, n$)
  for $k = 1 \rightarrow n$ **do**
    Set $\boldsymbol{A}_{k+1} = [\alpha_{k+1} \cdots \alpha_n]$;
    Set $\boldsymbol{b}_k = \boldsymbol{b} - x_1\alpha_1 \cdots - x_{k-1}\alpha_{k-1}$;
    Set $\boldsymbol{c}_{k+1} = [c_{k+1} \cdots c_n]^T$;
    Set $\mathcal{SP}_0 = (\boldsymbol{A}_{k+1}, \boldsymbol{b}_k, \boldsymbol{c}_{k+1})$;
    Set $\mathcal{SP}_1 = (\boldsymbol{A}_{k+1}, \boldsymbol{b}_k - \alpha_k, \boldsymbol{c}_{k+1})$;
    if $g_\theta(\mathcal{SP}_0) < c_k + g_\theta(\mathcal{SP}_1)$ **then**
      $x_k = 0$;
    else
      $x_k = 1$;
    end if
  end for
  return the solution $x_1, x_2, ..., x_n$ together with the weight sum $\sum_{i=1}^{n} c_i x_i$.
end function

All experiments were performed on a machine with an Intel CPU i7-8700K and an NVIDIA GeForce GTX 1080Ti GPU card.

4.2 Evaluating the Solution Quality of AIA-SC

We evaluated the solution quality of AIA-SC, and compared it with the established Chvatal's greedy algorithm [5]. Chvatal's greedy algorithm was designed purely based on human experience and heuristics: it repeatedly selects the subset with the highest performance/cost ratio, i.e., the ratio of the number of newly covered elements by the subset over its weight, until the selected subsets cover all elements.

AIA-SC found the optimal solutions on 87 out of the 100 WSCP20 instances; in contrast, Chvatal's greedy algorithm found the optimal solutions for only 11 instances. Furthermore, compared with Chvatal's greedy algorithm, AIA-SC found better solutions for 87 instances and inferior solutions for only 3 instances.

Table 1 shows the performance of the two algorithms for 10 WSCP20 instances. AIA-SC failed to find the optimal solution for only two instances, whereas Chvatal's greedy algorithm failed for seven instances. The gap between the solution by AIA-SC and the optimal solution is much lower than that by the Chvatal's greedy algorithm, e.g., 1.82% vs. 21.05% for WSCP20-3.

Table 1. Performance of AIA-SC and Chvatal's greedy algorithm on WSCP20 and WSCP50 instances

Instance	OPT	Chvatal's Greedy		AIA-SC	
		Objective value	Gap	Objective value	Gap
WSCP20-1	416	416	0	416	0
WSCP20-2	621	676	8.86%	**621**	**0**
WSCP20-3	494	598	21.05%	**503**	**1.82%**
WSCP20-4	451	490	10.87%	**451**	**0**
WSCP20-5	542	542	0	542	0
WSCP20-6	520	553	6.35%	**520**	**0**
WSCP20-7	570	678	18.95%	**570**	**0**
WSCP20-8	602	702	16.61%	**602**	**0**
WSCP20-9	529	549	3.78%	**529**	**0**
WSCP20-10	427	**427**	**0**	432	1.17%
WSCP50-1	1559	1742	11.73%	**1725**	**10.65%**
WSCP50-2	2206	2452	11.15%	**2331**	**5.67%**
WSCP50-3	1221	1899	55.53%	**1760**	**44.14%**
WSCP50-4	2548	3203	25.71%	**3067**	**20.37%**
WSCP50-5	2315	2417	4.41%	**2408**	**4.02%**
WSCP50-6	1965	**2294**	**16.74%**	2298	16.95%
WSCP50-7	1933	2128	9.83%	**2030**	**5.02%**
WSCP50-8	1975	2101	6.28%	**2013**	**1.92%**
WSCP50-9	2105	**2439**	**15.87%**	2525	19.95%
WSCP50-10	1630	1888	15.83%	**1630**	**0**

Both algorithms performed worse for WSCP50 instances than WSCP20; however, AIA-SC still outperforms Chvatal's greedy algorithm. As shown in Table 1, AIA-SC found the optimal solution for one instance WSCP50-10 while Chvatal's greedy algorithm failed for all the ten instances. In addition, AIA-SC generates better solutions for eight instances.

4.3 Evaluating the Decisions Made by AIA-SC

The key for AIA-SC to generate high-quality solutions is its ability to make correct decisions, i.e., $x_k = 1$ or $x_k = 0$, during the solving process. Figure 4 shows how AIA-SC makes decisions for the problem presented in Fig. 1. In the first step, AIA-SC made the correct decision $x_1 = 0$, i.e., abandoning the subset S_1, as $g_\theta(\mathcal{SP}_0) = 16.86$ is less than $c_1 + g_\theta(\mathcal{SP}_1) = 17.43$. Next, AIA-SC selected S_2 and S_3 as they are the only choices to cover the element a. In the fourth step, AIA-SC selected S_4 as $g_\theta(\mathcal{SP}_0) = 1.23$ is greater than $c_4 + g_\theta(\mathcal{SP}_1) = -1.65$. In the fifth step, AIA-SC abandoned S_5 as $g_\theta(\mathcal{SP}_0) = -2.73$ is less than $c_5 + g_\theta(\mathcal{SP}_1) = 0.27$. Finally, AIA-SC selected S_2, S_3, S_4 whose sum of the weights is 11.

Table 2. Confusion matrix of the decisions made by AIA-SC when solving (a) WSCP20 instances and (b) WSCP50 instances. Here, the true values denote whether $OPT(\mathcal{SP}_0) < c_k + OPT(\mathcal{SP}_1)$ or not, and the predictions denote whether $g_\theta(\mathcal{SP}_0) < c_k + g_\theta(\mathcal{SP}_1)$ or not

		True value	
		$x_k = 0$	$x_k = 1$
Predictions	$x_k = 0$	281	33
	$x_k = 1$	28	982

(a)

		True value	
		$x_k = 0$	$x_k = 1$
Predictions	$x_k = 0$	217	139
	$x_k = 1$	128	1494

(b)

In contrast, Chvatal's greedy algorithm selected S_4 as it has the highest performance-cost ratio of $\frac{2}{1}$. Subsequently, it selected S_1, S_3, S_5 and obtained the collection whose sum of the weights is 12. It should be noted that the selection of S_1 is not optimal although S_1 has the highest performance-cost ratio of $\frac{2}{2}$ at the second step. This result demonstrates the drawbacks of human-crafted heuristics.

It is worth pointing out that the relative magnitude between $g_\theta(\mathcal{SP}_0)$ and $c_k + g_\theta(\mathcal{SP}_1)$ rather than their actual values is essential to making correct decisions. As shown in Table 2, AIA-SC made a total of 1324 decisions when solving WSCP20 instances, out of which 1263 decisions are correct, thus achieving an accuracy of 95.39%. Table 2 shows that AIA-SC made 1711 correct decisions when solving WSCP50 instances, also achieving a high accuracy of 86.50% (1978 decisions in total). The high accuracy of correct decisions empowers AIA-SC to generate nearly optimal solutions.

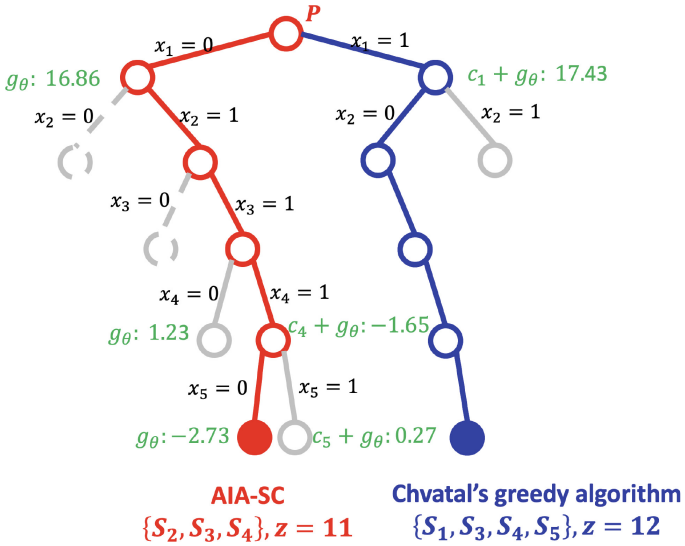

Fig. 4. The solving process of AIA-SC (in red) and Chvatal's greedy algorithm (in blue) for the instance shown in Fig. 1. AIA-SC finds the optimal solution while Chvatal's greedy algorithm fails. (Color figure online)

5 Conclusion and Discussion

The results presented here have highlighted the special features of AIA-SC in designing algorithms aided by deep learning. The abilities of AIA-SC have been clearly demonstrated by the high quality of the generated solutions and the high accuracy of decisions for set cover problems. Experimental results suggested that our AIA-SC approach consistently yields superior solutions than Chvatal's greedy algorithm which was designed using the human experience.

Notably, the fundamental concept of our approach is highly versatile and can be easily extended, requiring minimal modifications, to design efficient greedy algorithms for addressing other NP-hard problems.

AIA-SC approach uses an instance-specific training strategy, thus leading to longer running time than commercial solvers. In addition, the time required for training varies substantially even for WSCP instances of the same scale. Training a general neural network suitable for all WSCP instances remains one of the future studies. The use of transformer instead of the current full connection layers might greatly improve AIA-SC.

This paper represents a pioneering effort in leveraging deep learning technology to aid in algorithm design. Unlike traditional approaches relying on human-crafted heuristics, our approach learns heuristic rules autonomously. This intelligent search algorithm has the potential to surpass the limitations imposed by human experience. We firmly believe that data-driven algorithm design will

emerge as a compelling and burgeoning area of research, presenting new opportunities for exploration and advancements in the field.

Acknowledgements. This study was funded by the National Key Research and Development Program of China (2020YFA0907000), the National Natural Science Foundation of China (32370657, 32270657, 32271297) and Youth Innovation Promotion Association, Chinese Academy of Sciences. We appreciate the ComputeX center, ICT, CAS for providing computation service.

References

1. Sipser, M.: Introduction to the theory of computation. ACM SIGACT News **27**(1), 27–29 (1996)
2. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations, pp. 85–103. Springer, Heidelberg (1972). https://doi.org/10.1007/978-1-4684-2001-2_9
3. Glover, F.: Tabu search-part I. ORSA J. Comput. **1**(3), 190–206 (1989)
4. Russakovsky, O., et al.: Imagenet large scale visual recognition challenge. Int. J. Comput. Vision **115**(3), 211–252 (2015)
5. Chvatal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979)
6. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Courier Corporation, Chelmsford (2013)
7. Garmendia, A.I., Ceberio, J., Mendiburu, A.: Neural combinatorial optimization: a new player in the field. arXiv preprint arXiv:2205.01356 (2022)
8. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. Adv. Neural Inf. Process. Syst. **28** (2015)
9. Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. Adv. Neural Inf. Process. Syst. **30** (2017)
10. Alvarez, A.M., Louveaux, Q., Wehenkel, L.: A supervised machine learning approach to variable branching in branch-and-bound. In: IN ECML (2014)
11. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Oper. Res. Lett. **33**(1), 42–54 (2005)
12. Khalil, E.B., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Thirtieth AAAI Conference on Artificial Intelligence (2016)
13. Gasse, M., Chételat, D., Ferroni, N., Charlin, L., Lodi, A.: Exact combinatorial optimization with graph convolutional neural networks. Adv. Neural Inf. Process. Syst., 15580–15592 (2019)
14. He, H., Daume III, H., Eisner, J.M.: Learning to search in branch and bound algorithms. Adv. Neural Inf. Process. Syst., 3293–3301 (2014)
15. Joshi, C.K., Laurent, T., Bresson, X.: An efficient graph convolutional network technique for the travelling salesman problem. arXiv preprint arXiv:1906.01227 (2019)

16. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940 (2016)
17. Kool, W., Van Hoof, H., Welling, M.: Attention, learn to solve routing problems. arXiv preprint arXiv:1803.08475 (2018)
18. Bresson, X., Laurent, T.: The transformer network for the traveling salesman problem. arXiv preprint arXiv:2103.03012 (2021)
19. Jensen, R.E.: Sensitivity analysis and integer linear programming. Account. Rev. **43**(3), 425–446 (1968)
20. Yang, F., Jin, T., Liu, T.Y., Sun, X., Zhang, J.: Boosting dynamic programming with neural networks for solving NP-hard problems. In: Asian Conference on Machine Learning, pp. 726–739. PMLR (2018)
21. Balas, E., Ho, A.: Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In: Combinatorial Optimization, pp. 37–60 (1980)