



Machine Learning-Enhanced Ant Colony Optimization for Column Generation

Hongjie Xu

School of Computing Technologies, RMIT University
Melbourne, Australia
s3880497@student.rmit.edu.au

Yuan Sun

La Trobe Business School, La Trobe University
Melbourne, Australia
yuan.sun@latrobe.edu.au

Yunzhuang Shen

University of Technology Sydney
Sydney, Australia
yunzhuang.shen@uts.edu.au

Xiaodong Li

School of Computing Technologies, RMIT University
Melbourne, Australia
xiaodong.li@rmit.edu.au

ABSTRACT

Column generation (CG) is a powerful technique for solving optimization problems that involve a large number of variables or columns. This technique begins by solving a smaller problem with a subset of columns and gradually generates additional columns as needed. However, the generation of columns often requires solving difficult subproblems repeatedly, which can be a bottleneck for CG. To address this challenge, we propose a novel method called machine learning enhanced ant colony optimization (MLACO), to efficiently generate multiple high-quality columns from a subproblem. Specifically, we train a ML model to predict the optimal solution of a subproblem, and then integrate this ML prediction into the probabilistic model of ACO to sample multiple high-quality columns. Our experimental results on the bin packing problem with conflicts show that the MLACO method significantly improves the performance of CG compared to several state-of-the-art methods. Furthermore, when our method is incorporated into a Branch-and-Price method, it leads to a significant reduction in solution time.

CCS CONCEPTS

• **Applied computing** → **Operations research**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Ant colony optimization, machine learning, column generation, combinatorial optimization

ACM Reference Format:

Hongjie Xu, Yunzhuang Shen, Yuan Sun, and Xiaodong Li. 2024. Machine Learning-Enhanced Ant Colony Optimization for Column Generation. In *Genetic and Evolutionary Computation Conference (GECCO '24)*, July 14–18, 2024, Melbourne, VIC, Australia. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3638529.3654043>



This work is licensed under a Creative Commons Attribution International 4.0 License. *GECCO '24*, July 14–18, 2024, Melbourne, VIC, Australia
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0494-9/24/07.
<https://doi.org/10.1145/3638529.3654043>

1 INTRODUCTION

Column generation (CG) is a powerful method for solving linear programs (LP) that have a large number of variables (or columns) [17]. It is commonly used to obtain tight LP bounds to accelerate the process of the branch-and-bound method in combinatorial optimization [1]. CG is especially beneficial for tackling optimization problems that have a decomposable structure, such as vehicle routing, bin packing, and graph coloring problems.

CG solves a large-scale LP in iterative steps, starting from the LP containing a subset of columns, i.e., the restricted master problem (RMP). In an iteration, CG solves the current RMP and uses its dual solution to generate new columns that can improve the current RMP. Such columns should have negative reduced costs, and finding them typically involves solving an NP-hard subproblem called pricing problem. At optimality, no column with negative reduced costs can be further generated, and existing columns with nonzero values form an optimal solution to the original large-scale LP.

Repeatedly solving pricing problems is often a bottleneck in CG [17], and researchers have devised different approaches to tackle this issue, including exact methods, heuristics, and metaheuristics. It is widely recognized that the performance of CG is heavily influenced by both the quality and quantity of generated columns [30]. This differs from solving a traditional optimization problem, where typically only a single optimal column is sought.

In this paper, we introduce a hybrid method called MLACO, which combines machine learning (ML) and ant colony optimization (ACO) to efficiently generate multiple high-quality columns, as illustrated in Figure 1. In our method, we first train a ML model using a set of solved pricing problem instances. This ML model learns how to map from a set of problem-specific features and statistical measures to optimal solutions. Given a new problem instance, we use the ML model to predict its optimal solution and then incorporate the ML prediction into the ACO probabilistic model to generate a diverse set of high-quality columns. Our MLACO method is used to address the pricing problem and generate a diverse set of columns at every iteration of CG to accelerate its progress.

Our proposed MLACO method has the following advantages compared to existing methods for generating columns. Compared to exact and heuristic approaches based on mixed integer programming, our method can efficiently generate a large number of columns with a negative reduced cost. Compared to metaheuristics such as ACO, our method learns from optimally solved pricing

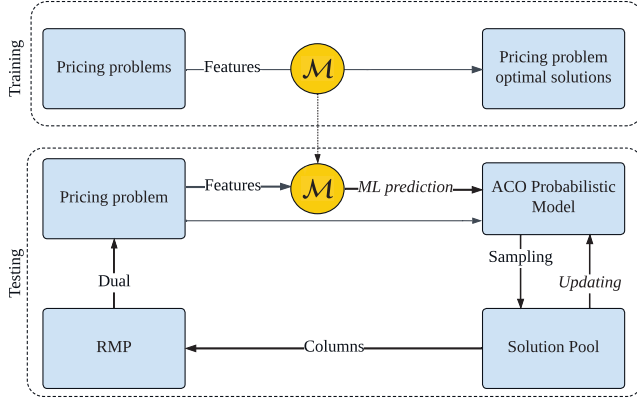


Figure 1: An illustration of our MLACO method as a pricing heuristic for CG. We train a ML model \mathcal{M} on a set of solved pricing problem instances to learn a mapping from problem features to the optimal solutions. In the testing phase, we use the offline-trained ML model \mathcal{M} to predict an optimal solution to an unseen pricing problem. The ML prediction is then incorporated into the ACO probabilistic model to sample multiple high-quality solutions, which are then iteratively improved in an online manner.

problems and can generate high-quality columns more quickly. Compared to a pure ML-based sampling method [30], our method can further improve the quality of the columns generated due to the online learning nature of ACO.

Our main contributions can be summarized as follows.

- We propose the MLACO method for solving pricing problems in the context of CG. Our method uses ML to predict optimal solutions to a pricing problem and uses the ML prediction to accelerate ACO to efficiently generate multiple high-quality columns.
- We evaluate our method on the bin packing problem with conflicts (BPPC) and show that our method significantly accelerates CG compared to several baselines. Furthermore, we show that our MLACO method can tackle problems that are larger and more difficult than those used in training.
- We also evaluate our MLACO method within branch-and-price, an exact method that integrates CG to solve integer BPPC. We show that CG with our proposed method can reduce the solution time of Branch-and-Price.

2 BACKGROUND AND RELATED WORK

2.1 Problem Formulations

We are given a set of bins $k \in \mathcal{K}$ with a uniform capacity W , and a set of items $i \in \mathcal{V}$, each with a non-negative weight w_i . Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a conflict graph, where a vertex represents an item and an edge indicates that the corresponding vertices have conflicts. The goal of the bin packing problem with conflicts (BPPC) is to place items in the minimum number of bins within the capacity limit, such that conflicting items are not placed in the same bin. This problem has various practical applications [26], such as job scheduling, parallel computing, and database storage.

Let the binary variable $x_{i,k}$ denote if item i is assigned to bin k and binary variable y_k represent whether a bin k is used. The BPPC can be formulated as an integer programming (IP) problem:

$$\min_{\mathbf{x}, \mathbf{y}} \sum_{k \in \mathcal{K}} y_k \quad (1)$$

$$\text{s.t.} \sum_{k \in \mathcal{K}} x_{i,k} = 1, \quad i \in \mathcal{V}, \quad (2)$$

$$\sum_{i \in \mathcal{V}} w_i x_{i,k} \leq W, \quad k \in \mathcal{K}, \quad (3)$$

$$x_{i,k} + x_{j,k} \leq y_k, \quad (i, j) \in \mathcal{E}, k \in \mathcal{K}, \quad (4)$$

$$x_{i,k} \in \{0, 1\}, \quad i \in \mathcal{V}, k \in \mathcal{K}, \quad (5)$$

$$y_k \in \{0, 1\}, \quad k \in \mathcal{K}. \quad (6)$$

The objective (1) minimizes the number of bins used; constraint (2) ensures that each item must be placed in a bin; constraint (3) specifies that the total weight of items placed in a bin must be within its capacity; and constraint (4) enforces that the conflict items should not be placed in one bin. This formulation is typically referred to as the compact IP formulation because there is a polynomial number of variables and constraints. The complexity of the problem is closely related to the problem characteristics. As discussed in [28], existing methods can face significant difficulties when dealing with instances that have a higher bin capacity and/or conflict graphs without showing specific patterns.

It can be seen that the compact IP formulation for BPPC has a symmetric structure. Specifically, a configuration of item placement can correspond to multiple solutions, which can be obtained by shuffling the indices of bins. As a result, it provides a weak LP relaxation bound [18], which can be obtained by relaxing the integer constraints in the compact IP formulation and solving the resulting LP problem. This can significantly slow the branch-and-bound solution process. To address this, Dantzig-Wolfe decomposition can be adopted to obtain an alternative IP formulation of BPPC [35, 36]:

$$\min_z \sum_{P \in \mathcal{P}} z_P \quad (7)$$

$$\text{s.t.} \sum_{P \in \mathcal{P}_i} z_P \geq 1, \quad i \in \mathcal{V}, \quad (8)$$

$$z_P \in \{0, 1\}, \quad P \in \mathcal{P}. \quad (9)$$

Here, \mathcal{P} denotes the set of all possible patterns to pack items in a bin and \mathcal{P}_i denotes the set of packing patterns that include item i . P represents a specific packing pattern associated with a binary decision variable z_P , which indicates whether this packing pattern is used. The objective (7) is to minimize the number of packing patterns used, and the constraint (8) ensures that each item must be covered in at least one of the selected packing patterns.

This Dantzig-Wolfe reformulation eliminates symmetry by grouping items into packing patterns, and it yields a much stronger LP relaxation bound than the compact IP formulation. However, it typically requires an exponential number of variables (or columns) to represent all packing patterns; therefore, its LP relaxation cannot be solved directly using standard techniques, such as simplex methods. In the next subsection, we introduce CG for solving the LP relaxation of this Dantzig-Wolfe reformulation.

2.2 Column Generation for BPPC

To tackle such a large-scale LP, CG takes an iterative procedure, starting from the LP containing a subset of columns (or packing patterns), referred to as the restricted master problem (RMP). In a single iteration, CG solves the RMP and utilize its dual values (π) to formulate a pricing problem. The solution to the pricing problem with the most negative reduced costs determines the column that can be added to the RMP. Searching for such columns involves solving a pricing problem, outlined as follows.

$$\max_{\mathbf{x}} \sum_{i \in V} \pi_i x_i \quad (10)$$

$$s.t. \sum_{i \in V} w_i x_i \leq W, \quad (11)$$

$$x_i + x_j \leq 1, \quad (i, j) \in \mathcal{E}, \quad (12)$$

$$x_i \in \{0, 1\}, \quad i \in \mathcal{V}. \quad (13)$$

Here, the decision variable x_i denotes whether an item i is used to form a packing pattern. The objective function (10) aims to minimize the reduced cost ($1 - \sum_{i \in V} \pi_i x_i$), which is equivalent to maximizing $\sum_{i \in V} \pi_i x_i$. Constraints (11)-(13) correspond to the Constraints (3)-(5) in the compact IP formulation, indicating that a feasible solution to the pricing problem must be a valid packing pattern. In fact, the pricing problem is a one-dimensional knapsack problem with conflicts (1DKPC), which is NP-hard [22, 23].

If the minimum reduced cost is negative, a column representing this new packing pattern can be added to the RMP to start the next iteration. Otherwise, the non-negativity of minimum reduced cost indicates that the objective value of the RMP cannot be further improved, and hence the original large LP has been solved to optimality. At this point, the RMP solution is the optimal solution to the original large LP.

The need to repeatedly solve pricing problems is typically a bottleneck in CG [17]. Past studies have explored exact and heuristic pricing methods [13, 21, 28, 30], with extensive empirical studies showing that the performance of CG hinges on multiple evaluation criteria for a pricing method. These criteria can be summarized by the ability to produce a large and diverse set of high-quality columns efficiently. In this regard, ACO, given its ability to generate a diverse set of solutions, can be seen as a promising technique for heuristic pricing introduced in the following.

2.3 Ant Colony Optimization

ACO is a population-based metaheuristic inspired by biological ants seeking the shortest path between foods and their colony [8]. It has many applications in large-scale combinatorial optimization [7, 16, 19, 37], such as the classic traveling salesman problem.

ACO maintains a probabilistic distribution that models the likelihood of decision variables taking values of 0 or 1 in high-quality solutions, i.e., whether an item is in the 1DKPC solution. It alternates between sampling solutions according to a probabilistic model and updating the models using better-quality solutions, which can be regarded as an online learning model. In the process of constructing a feasible solution, the probabilistic distribution can be defined

as follows.

$$p_j = \begin{cases} \frac{\tau_j^\alpha \eta_j^\beta}{\sum_{j \in C} \tau_j^\alpha \eta_j^\beta}, & j \in C \\ 0, & j \notin C. \end{cases} \quad (14)$$

Here, C represents the set of candidate items that can be selected without violating the constraints of the problem. η_j and τ_j , weighted by the hyperparameters α and β , are critical to the performance of ACO. More specifically, η_j is a heuristic measure that can be used to inject prior knowledge about the likelihood that an item i will be used in high-quality solutions. This value is typically set by handcrafted heuristics based on expert knowledge and remains constant throughout the process of ACO. On the other hand, τ_j denotes the desirability to choose an item j (or the amount of pheromone deposited by ants). It is typically uniformly initialized and updated dynamically as better solutions are sampled, to more accurately indicate the likelihood of the item j shown in high-quality solutions. Intuitively, if an item j is frequently selected in high-quality solutions, the corresponding pheromone value τ_j is increased to reinforce sampling that variable when constructing new samples.

We introduce a widely used policy to update pheromone values from the ant system [9]. In each iteration, after constructing a set of N newly sampled solutions, the pheromone values τ_i , where $i = 1, \dots, v$, are updated based on the sample solutions generated:

$$\tau_i = (1 - \rho)\tau_i + \sum_{n=1}^N \Delta\tau_i^n, \quad (15)$$

where $\rho > 0$ is the pheromone evaporation coefficient and $\Delta\tau_i^n$ is the amount of pheromone deposited by the n^{th} sample at the item i . Let c_n denote the objective value collected by the n^{th} sample, c_{best} represents the best objective value found so far, and $\lambda > 0$ be a constant. We can define $\Delta\tau_i^n = c_n/c_{best}/\lambda$, if the item i is selected in the current sampled solution; otherwise $\Delta\tau_i^n = 0$. The amount of pheromone deposited by an ant when it selects items is proportional to the objective value of the set. As we are solving a maximization problem, items that appear in high quality solutions are reinforced, so that these items are more likely to be selected when constructing solutions in the later iterations.

ACO represents an online learning method which can iteratively improve its optimization performance through sampling using Equation (14). ACO typically starts with an initialization of randomly generated individuals, or individuals injected with simple heuristic rules. However, we can do better by training an offline ML model using data gathered from solved problem instances before ACO's online learning phase. Such a ML-based solution prediction method can significantly accelerate ACO's optimization process.

2.4 Machine Learning for Optimization

Machine learning has been shown to be effective in improving combinatorial optimization [2], such as learning to make decisions instead of using handcrafted heuristic rules for mixed-integer programming solvers [39], metaheuristics [14, 33], and decomposition techniques [20, 30, 34].

Assuming that we have access to data of solved problem instances, then it is possible to employ ML to learn from such data and then generalize it on unseen problem instances. Sun et al. [32]

proposed MLACO that explores such a solution prediction method to warm-start ACO, substituting either the heuristic weight measure η set by some primitive heuristic rule, or the randomly initialized pheromone matrix τ . Ye et al. [38] proposed a deep neural network to achieve better quality predictions. Another study [25] presents a hybrid ACO method that integrates the problem information extracted from Graphic Neural Network with the standard pheromone and greedy information, employing a Q-learning to learn which type of information is better for solving specific problem instances. Unlike these studies that aim to find a single best solution for standalone optimization problems, we develop a method that combines ML and ACO to generate a diverse set of high-quality columns with the aim to accelerate CG. In particular, the diversity of the generated columns can be crucial to the efficiency of CG, in addition to the solution quality.

ML has also been used to improve the CG process, such as automatically learning to predict optimal dual values to stabilize CG [15], learning column selection rules [5, 20], and learning heuristic pricing methods [30]. In particular, the MLPH method proposed by Shen et al. [30] can accelerate the CG process by sampling columns based on its ML prediction of the pricing problem. However, the limitation of MLPH is that it samples columns according to a fixed probability distribution proportions to the ML prediction. This work aims to address this limitation by combining ML offline prediction with ACO online learning to further accelerate CG.

3 THE PROPOSED APPROACH

In this section, we describe the proposed heuristic pricing method that is used to address the pricing problem (i.e., 1DKPC) and generate columns at every iteration of the CG. Our hybrid method combines ML and ACO to strike a balance between efficiency, effectiveness, and solution diversity. As shown in Algorithm 1, ML is used to make a prediction of the optimal solution to the pricing problem, denoted as p , given the features of the decision variables for an unseen pricing problem instance. The ML prediction is then used to accelerate the ACO to quickly find a diverse set of good solutions.

3.1 Optimal Solution Prediction

We model the solution prediction task for 1DKPC as a binary classification problem, where the output of a decision variable denotes the likelihood of the corresponding item in the optimal solution. In our training set, a training example (f, y) corresponds to an item in an optimally solved 1DKPC problem, where f represents a set of features that are used to characterize this item and y is its optimal solution value. We describe our four problem-specific features and two statistical features as follows.

Recall that the 1DKPC problem (Equations (10)-(13)) aims to find a set of non-conflict items that maximize profit, subject to a capacity limit. The first feature we include is about the profit (i.e., the objective coefficient) of an item i and is normalized by the maximum and minimum profits of items in the same problem instance as follows,

$$f_1(i) = \frac{\pi_i - \min_{j \in V} \pi_j}{\max_{j \in V} \pi_j - \min_{j \in V} \pi_j}. \quad (16)$$

Algorithm 1: MLACO for CG

Input: \mathcal{M} : an offline-trained ML model;
 T : iteration number for ACO;
 (π, W, C, V, E) : problem data;
Output: New columns with negative reduced cost

- 1 Sampling a set of random samples (Alg. 2)
- 2 Compute statistical features based on random samples
- 3 ML prediction $p \leftarrow \mathcal{M}(f)$
- 4 Initialize $\eta = p$
- 5 Initialize τ uniformly
- 6 **for** $t \leftarrow 1$ to T **do**
- 7 Diversity-aware Sampling (Alg. 3)
- 8 Update τ according to Eq. (15)
- 9 **end**

Algorithm 2: Random Sampling

Input: (π, W, C, V, E) : problem data
 N : sample size;
Output: A set of randomly sampled solutions

- 1 **for** $n \leftarrow 1$ to N **do**
- 2 Initialize a candidate set
- 3 **while** *candidate set is not empty* **do**
- 4 Randomly select an item from the set of candidates
- 5 Update the candidate set
- 6 **end**
- 7 **end**

Our second feature calculates the profit per unit weight, i.e., the ratio of the profit and weight of item i ,

$$f_2(i) = \frac{\pi_i}{w_i}. \quad (17)$$

Note that this is a commonly used measure in designing greedy heuristic rules for knapsack problems. Our third feature is about degree of an item and is normalized by the maximum and minimum profits of items in the same problem instance,

$$f_3(i) = \frac{\deg(i) - \min_{j \in V} \deg(j)}{\max_{j \in V} \deg(j) - \min_{j \in V} \deg(j)}, \quad (18)$$

The fourth feature is an upper bound on the profit if item i is selected, computed by

$$f_4(i) = \pi_i + \sum_{(i,j) \notin E} \pi_j. \quad (19)$$

It calculates the sum of profits of item i and the items that are not in conflict with item i .

To better characterize decision variables in high-quality solutions, we adopt two statistical features [31] that are computed over a set of N randomly sampled solutions. Our random sampling algorithm is shown in Algorithm 2. It is obvious that the time complexity to generate one sample of 1DKPC using this method is $O(|V| + |E|)$ if we represent the conflict graph using an adjacency list, where $|E|$ represents the number of edges in the conflict graph. Hence, the total time complexity of generating N samples is $O(N(|V| + |E|))$.

The first statistical feature measures the correlation between the presence of an item i in the sample solutions and the objective

values of the samples.

$$f_c(i) = \frac{\sum_{n=1}^N (s_i^n - \bar{s}_i)(o^n - \bar{o})}{\sqrt{\sum_{n=1}^N (s_i^n - \bar{s}_i)^2} \sqrt{\sum_{n=1}^N (o^n - \bar{o})^2}}, \quad (20)$$

where s_i^n is a binary value, indicating whether the item i is a part of the n^{th} sample, o^n represents the objective value of the n^{th} sample, and $\bar{s}_i = \sum_{n=1}^N s_i^n / N$ and $\bar{o} = \sum_{n=1}^N o^n / N$ are the average values across the N samples. An item with a high correlation score indicates that this item is likely to appear in high-quality solutions of the corresponding 1DKPC instance.

The second statistical measure is based on the ranking of the N sample solutions. Let r^n denote the rank of the n^{th} sample in terms of its objective value in descending order. For an item i , this statistical feature accumulates the ranking score across the samples that contain this item:

$$f_r(i) = \sum_{n=1}^N \frac{s_i^n}{r^n}. \quad (21)$$

An item with a high ranking score indicates that this item appears more frequently in high-quality solutions.

Given a set of optimally solved 1DKPC instances, we can then construct the training set where each training example corresponds to an item in a 1DKPC instance. For each training example, we extract the six features to characterize the associated item and assign a class label based on whether the item is part of the optimal solution. We train a Support Vector Machine (SVM) [3], to distinguish items that belong to optimal solutions from those that do not. Given an unseen 1DKPC instance, the trained SVM model can then be used to predict for each item whether it belongs to an optimal solution. More specifically, we can calculate the distance from an item to the decision boundary of SVM in the feature space, which indicates the likelihood that this item belongs to an optimal solution. We employ SVM mainly because of its efficiency in making predictions.

3.2 MLACO for Column Generation

Given the offline-trained ML model, we can then use it to make predictions of the optimal solution for unseen problem instances in the iterative process of CG. An accurate prediction can then be used to accelerate ACO to quickly find high-quality solutions. Moreover, we devise diversity-aware sampling to encourage ACO to sample a diverse set of high-quality solutions.

Accelerating ACO with ML prediction. The effectiveness of ACO is heavily dependent on the parameterized probabilistic model, which contains two main parameters τ and η . The algorithm automatically learns the value of τ through searching iterations, while the initialization of η typically draws upon prior knowledge provided by experts. In the context of the 1DKPC, the profit-to-weight ratio serves as a robust greedy heuristic rule and is commonly employed in relevant research studies. For each pricing problem, η can be set to the profit-to-weight ratio: $\eta_j = \pi_i / w_i$, where π represents dual solutions retrieved from RMP. This heuristic rule evaluates the item's price per unit weight, making items with higher profit per unit weight are more likely to be selected [6]. We will use this setting for the standard ACO in our experimental comparison.

Algorithm 3: Diversity-aware Sampling

Input: (π, W, C, V, E) : problem data
 p : probabilistic model
Output: Sampled columns with negative reduced costs

```

1 for each item do
2   Add this initial item to the working solution.
3   do
4     Update candidate set
5     Update the probabilistic model according to Eq. 14
6     Sample an item from candidate set; add this item to
       the working solution
7   while the candidate set of items is not empty;
8 end
```

In contrast to initialization based on prior knowledge, we set η based on our ML predictions, aiming to guide the search toward more promising areas. As mentioned in Section 3.1, we train a binary SVM model to predict whether an item belongs to the optimal solution for a pricing problem, that is, 1DKPC. We use Platt scaling to transform the SVM prediction from the class label to a probability distribution over classes [24]. The transformation produces probability estimates given by

$$P(y = 1|x) = \frac{1}{1 + \exp(af(x) + b)}, \quad (22)$$

where a and b are two learned parameters. This expression represents a logistic transformation of the classifier scores $f(x)$ into a probability value that indicates the probability that this item belongs to an optimal solution. We use the probability values to set η in the ACO algorithm by default and will also explore and discuss other possible uses of the ML predictions to improve ACO in our experimental study.

Diversity-aware Sampling. In CG, sampling a diverse set of columns can be crucial to its performance. To better align this objective, we propose a diversity-aware sampling method to replace the traditional sampling method in ACO. We generate columns based on the ML prediction while ensuring that each item is covered at least once, i.e., at each iteration, assuming that the sample size is equal to the number of items, we initiate the sampling process by starting from the first item and progressing to the last. This ensures that each item is treated as the first added item at least once. As shown in Algorithm 3, for every sampled solution, once the first item k_1 is selected, we choose the remaining items based on the ML prediction. More specifically, our method to generate one column includes the following steps:

- Initialize an item with the starting item k_1 .
- Generate candidate items k_j that can be visited.
- Select the next item to be included in the sampled column that does not violate the capacity and conflict constraints.

Note that we only include columns with negative reduced cost into RMP in each sampling iteration.

4 EMPIRICAL STUDIES

4.1 Experimental Setup

Bin Packing Benchmarks and Conflict Graph Generation. We consider a standard bin packing benchmark, Hard28 [29], for testing. Hard28 consists of 28 problem instances with a uniform bin capacity 1000. For a problem instance, the number of items is in $\{160, 180, 200\}$. Following previous work [28], we model the conflicts between items using randomly generated graphs with density values around 0.5. We also consider different capacity multipliers, $\{1, 2, 5, 15\}$. These settings can result in more challenging pricing problems as discussed in [28]. For training, we use a set of 20 small problem instances with 120 items from Falkenauer [10], different from the test problem instances. Each of these training problem instances has 120 items, and their weights are uniformly distributed.

Data collection and training. We use CG to solve a training problem instance. In this process, multiple pricing problems are solved to optimality, and we record the optimal solution values and the features of pricing problem variables to form the training data. Specifically, we consider pricing problems in every five iterations to increase the diversity of training data. Moreover, we start the recording from the 10^{th} iteration and continue to the 30^{th} iteration of CG, because we observe that these pricing problem instances are relatively easy to solve in the early iterations. The statistical features are calculated from a set of N randomly sampled solutions, where N is the number of items. All features are normalized on an instance-wise basis.

With the training data, we train a linear SVM using the standard software package [4]. We configure the SVM to directly output the probabilistic values between 0 and 1, and set the regularization term according to the ratio between the negative training examples and the positive ones. The regularization term controls the importance of correctly classifying positive and negative training examples, and our setting of the regularization parameter specifies that these two situations are equally important. We set the remaining hyperparameters to the default values.

Compared Methods. The following methods are compared in our empirical study: 1) **MLACO** (the proposed method)¹: We use MLACO to solve a pricing problem and add all newly generated columns with negative reduced costs to the RMP to start the next iteration of CG. The heuristic value η is set to the ML prediction, and the pheromone value τ is initialized uniformly. We set α and β to 1, and set ρ to 0.95. The iteration number is set to 10 and the population size is set to the number of items in a problem instance; 2) **Gurobi^s**: We use the Gurobi solver [12] to solve the pricing problem to optimality and add the optimal column to start the next iteration of CG. Note that Gurobi is a state-of-the-art mixed-integer programming solver; 3) **Gurobi^m**: In this setting, we use Gurobi to solve a pricing problem to optimality and add multiple columns with negative reduced costs in addition to the optimal column. We turn on the solution pool feature of Gurobi to encourage it to find multiple high-quality solutions; 4) **MLPH** [30]: A method that samples columns based on ML prediction of the optimal solution on the graph coloring problem. Compared to our method, the probabilistic model in MLPH is set according to the ML prediction and

¹Our source code is written in C++ and is available at https://github.com/hongjie-ml/MLACO_binpack

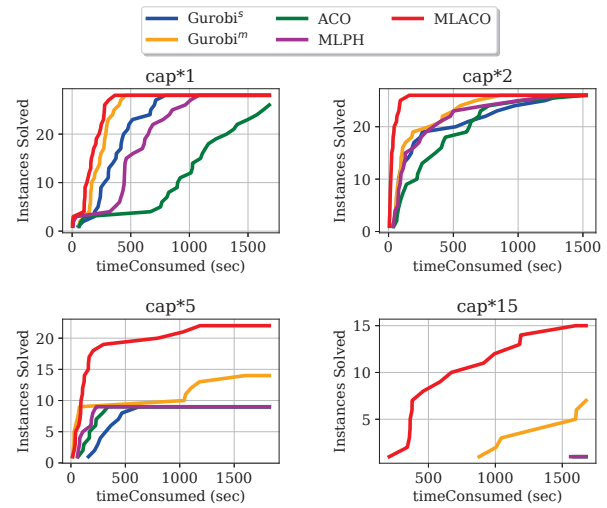


Figure 2: Number of instances solved by CG using different pricing methods. Each subfigure shows the results for a different bin capacity multiplier.

remains fixed throughout the sampling process. 5) **ACO**: A widely-used metaheuristic for combinatorial optimization as introduced in Section 2.3. The parameters in ACO are the same as MLACO. We set the heuristic value η to a widely-used heuristic rule, the profit-to-weight ratio.

We initialize the RMP with a set of randomly generated columns such that the RMP is feasible, to start the solution process of CG. The pricing problem in an iteration of CG is solved by a compared method. For MLACO, MLPH, and ACO, they are heuristic pricing methods without the guarantee of finding the optimal solution. When these methods do not find any column with a negative reduced cost, we execute the exact method Gurobi^s to find the optimal column. If the minimum reduced cost is negative, CG proceeds to the next iteration. Otherwise, CG reaches optimality and ends.

In the remainder of this section, we first compare the efficacy of different pricing methods in terms of the performance of CG to solve the LP relaxations for the BPPC. Then, we report the results for branch-and-price for solving the integer problem BPPC, where CG with a pricing method is used to derive LP relaxation bounds at every node of the branch-and-bound tree. In all experimental settings, CG is initialized using a set of randomly generated columns and is subject to a cutoff time of 1800 seconds.

4.2 Results for CG

Figure 2 shows the number of solved problem instances for a method within a certain time limit. For instance, under the capacity multiplier 15, in the first 1,500 seconds, our MLACO method solves 15 LP instances, Gurobi^m solves 5 LP instances, and the rest of the methods do not solve any. Overall, we can observe that MLACO can solve more problem instances at a given time limit under all capacity settings. The performance gap between MLACO and the compared methods increases as the bin capacity increases, and this is because the higher bin capacity leads to more difficult pricing problems. Specifically, we can first observe that MLACO outperforms ACO

Method	cap*1	cap*2	cap*5	cap*15
Gurobi ^s	28	26	9	0
Gurobi ^m	28	26	14	7
ACO	26	26	9	1
MLPH	28	26	10	1
MLACO	28	26	23	15

Table 1: Number of instances solved by CG with various pricing methods for different capacity multipliers.

Method	cap*1	cap*2	cap*5	cap*15
Gurobi ^s	383.26	426.65	1332.62	1800.00
Gurobi ^m	236.75	320.77	1126.82	1676.12
ACO	1059.76	494.99	1283.05	1792.43
MLPH	535.66	379.06	1264.46	1791.52
MLACO	171.31	162.43	566.23	1192.31

Table 2: Solving time of CG with different pricing methods averaged across all problem instances.

by a large margin. This shows the benefit of incorporating ML techniques over handcrafted heuristics. Our proposed MLACO also outperforms MLPH significantly. This shows the efficacy of updating the probabilistic model during the sampling process, resulting in more efficient sampling of high-quality solutions. It can also be seen that Gurobi^m better accelerates CG compared to Gurobi^s, showing the benefits of generating multiple high-quality columns in an iteration of CG. Similar observations can be made in Table 1, which shows the number of optimally solved problem instances at the cutoff time. Most noticeably, MLACO still manages to solve 15 LP instances when the capacity multiplier equals 15, demonstrating its capability under more difficult pricing problem settings.

In addition, we present the runtimes of CG with various pricing methods averaged across all problem instances in Table 2. Note that if a method cannot solve a problem instance within the cutoff time, the runtime is set to the cutoff time 1800, which is used to compute the average results. We can observe that CG with our MLACO method achieves significantly shorter runtimes compared to other methods, and this observation is consistent across problem instances with all different capacity multipliers.

Ablation Study. Table 3 presents the results of an ablation study on several different ways of combining ML and ACO. Specifically, we examine the following three variants:

- **mlaco_predicted_eta**: Set the η value to the ML prediction value and initialize τ uniformly, as that in Section 3;
- **mlaco_pred_heu_eta**: Set the η value to the product of the predicted probability and profit-to-weight ratio: $\eta_i = p_i \cdot \pi_i / w_i$ and initialize τ uniformly;
- **mlaco_predicted_tau**: Set the τ value to the ML prediction value and initialize η uniformly.

We can observe that initializing η with the ML prediction value achieves the best performance across all three model variants. This is closely followed by initializing τ with the ML prediction value. The variant which sets the value of η to the product of the predicted probability and a greedy heuristic rule performs worse when the

Method	Sampling	cap*1	cap*2	cap*5	cap*15
mlaco_predicted_eta	Y	28	26	23	15
mlaco_pred_heu_eta	Y	28	26	13	5
mlaco_predicted_tau	Y	28	26	18	14
mlaco_predicted_eta	N	27	24	17	12
mlaco_pred_heu_eta	N	27	25	9	2
mlaco_predicted_tau	N	25	26	16	12

Table 3: Comparison of MLACO variants with and without the use of diversity-aware sampling.

capacity multiplier becomes larger. One possible explanation is that the greedy heuristic rule biases the search in a wrong direction, as ACO lacks information about the conflict graph.

Finally, we examine the efficacy of the proposed diversity-aware sampling method compared to the regular sampling method in ACO. Recall that our method encourages sample diversity by ensuring that each item must be covered at least once by the newly generated columns. The results of MLACO without using diversity-aware sampling are shown in the bottom half of Table 3. We can see that diverse-aware sampling significantly improves the performance of MLACO variants across all capacity settings. This result shows the importance of generating a diverse set of columns for CG.

4.3 Results for Branch-and-Price

Our experiments have shown that the proposed MLACO can efficiently generate a diverse set of high-quality columns, thereby accelerating CG to derive tight LP relaxation bounds for BPPC. In this part, we leverage CG with our pricing method for enhancing branch-and-bound, to obtain an optimal integer solution to BPPC. Branch-and-bound is a canonical method for exact combinatorial optimization. It works by decomposing the integer problem into smaller subproblems and exploring the subproblems recursively. This process can be seen as a tree search, where each node represents a subproblem. A critical component of branch-and-bound is its bounding function, which determines whether a node can be safely pruned without a further visit. Specifically, a node can be pruned if its best possible solution (e.g., the LP bound) is no better than the objective value of the incumbent solution. Here, we use CG at each node to produce tight LP bounds. Note that branch-and-bound with CG, commonly called branch-and-price, is very effective on a range of problems [1].

Our branch-and-price implementation is based on an academic mixed-integer programming solver, SCIP [11]. Most importantly, we adopt the Ryan/Foster branching [27], which is commonly preferred in branch-and-price. Specifically, a problem is decomposed into two subproblems, by adding constraints that specify that a pair of items that are either a) forced to be packed together or b) prohibited from being packed together. Note that this branching rule can play a crucial role in the solution time of branch-and-price, in addition to the bounding function.

Figure 3 shows the number of instances solved within a particular optimality gap by the branch-and-price algorithm with different pricing methods. The optimality gap measures the difference between the objective value of the best-found solution and the worst LP objective among all leaf nodes in the branch-and-bound tree. The

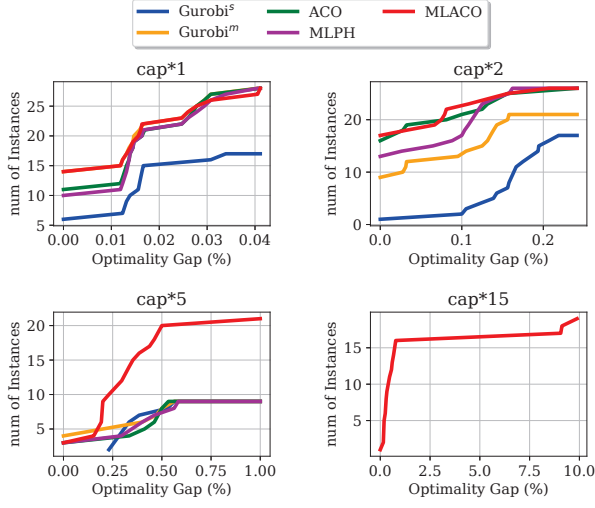


Figure 3: Number of instances solved within an optimality gap by branch-and-price with various pricing methods.

optimality gap reduces to 0 when a problem is solved to optimality. It can be seen that MLACO achieves very competitive performance under small capacity multipliers 1 and 2. With high capacity multipliers 5 and 15, branch-and-price with MLACO solves much more problem instances within a certain optimality gap. For instance, with the capacity multiplier 5, MLACO solves 20 out of 28 instances to less than 0.5% of the optimality gap, while the compared methods achieve this result for less than 10 problem instances. The main reason for such a significant difference in performance is that our pricing heuristic, MLACO, can still solve many LP relaxations at the root node under large capacity multiplier settings. In contrast, the compared methods struggle to solve the root LP; therefore, the optimality gap does not exist. Table 4 reports the number of problem instances optimally solved by each method (i.e., the optimality gap is 0) under different capacity multipliers.

In Table 5, we report the solution time of branch-and-price with various pricing methods for the problem instances with default bin capacity (multiplier equal to 1). MLACO obtains the shortest solution time for 10 problem instances, and the compared methods achieve the shortest solution time in 8 problem instances in total. Specifically, MLACO outperforms ACO and MLPH in a majority of the problem instances, because it inherits the benefits from both of-line learning (i.e., ML) and online learning (i.e., ACO). It can be seen that the performance of MLACO and Gurobi^m tends to differ substantially among the test problem instances. This should be understandable because Gurobi is based on mixed-integer-programming techniques and is very different from the nature of MLACO.

5 CONCLUSION

In this paper, we introduced a hybrid approach called MLACO that combines machine learning (ML) with ant colony optimization (ACO) to boost Column Generation (CG). We trained an offline ML model based on historical data to predict the optimal solution to a pricing problem in CG and integrated the ML prediction with the probabilistic model of ACO to generate multiple high-quality

Method	cap*1	cap*2	cap*5	cap*15
Gurobi ^s	6	1	0	0
Gurobi ^m	14	9	4	0
ACO	11	16	3	0
MLPH	10	13	3	0
MLACO	14	17	3	1

Table 4: Number of instances solved to optimality by branch-and-price with various pricing methods.

Instance	Gurobi ^s	Gurobi ^m	ACO	MLPH	MLACO
BPP832	-	1730.8	-	-	-
BPP485	-	1398.3	-	-	-
BPP181	1331.4	518.7	-	-	945.8
BPP47	648.5	243.7	78.6	312.8	127.3
BPP640	1112.7	606.1	44.5	133.9	64.0
BPP60	-	1531.3	1745.9	1143.3	-
BPP14	-	1542.7	1018.2	1009.3	-
BPP814	1138.8	758.9	1627.9	374.2	679.4
BPP13	-	-	-	-	1441.5
BPP709	-	-	-	-	1320.2
BPP40	-	-	1654.2	-	1223.6
BPP766	-	-	1527.2	1463.7	1169.2
BPP645	-	1714.5	-	-	892.1
BPP716	-	1565.7	-	-	674.6
BPP531	-	798.3	583.8	532.3	432.1
BPP360	832.7	961.6	539.8	496.0	411.8
BPP359	-	699.7	524.9	610.3	351.3
BPP742	1646.5	620.0	371.5	581.9	118.5

Table 5: Solution time for problem instances solved by at least one method. ‘-’ denotes that a method does not solve the problem instance within the cutoff time.

columns. We explored different ways of combining ML predictions with ACO and proposed a diversity-aware sampling method to improve the diversity of generated columns, which is crucial for CG. We evaluated our method on the bin packing problem with conflicts and showed that our MLACO method significantly improved the performance of CG compared to several state-of-the-art methods, especially when pricing problems are difficult. We also showed that our method significantly reduced the solution time of a Branch-and-Price algorithm for generating optimal integer solutions.

There are several potential avenues for further research. First, our current method relies on manual feature extraction, and thus using graph neural networks could offer a promising strategy for automatic feature extraction. Second, it would be interesting to extend our method to solve other combinatorial optimization problems, such as vehicle routing problems, where pricing problems are shortest-path problems. Finally, exploring the use of reinforcement learning to learn an intelligent policy for generating columns in solving pricing problems would be another interesting direction for future investigation.

REFERENCES

- [1] Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. 1998. Branch-and-price: Column generation for solving huge integer programs. *Operations research* 46, 3 (1998), 316–329.
- [2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2021. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research* 290, 2 (2021), 405–421.
- [3] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory* (Pittsburgh, Pennsylvania, USA) (COLT '92). Association for Computing Machinery, New York, NY, USA, 144–152. <https://doi.org/10.1145/130385.130401>
- [4] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Issue 3. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [5] Cheng Chi, Amine Aboussalah, Elias Khalil, Juyoung Wang, and Zoha Sherkat-Masoumi. 2022. A deep reinforcement learning framework for column generation. *Advances in Neural Information Processing Systems* 35 (2022), 9633–9644.
- [6] Stefano Coniglio, Fabio Furini, and Pablo San Segundo. 2021. A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. *European Journal of Operational Research* 289, 2 (2021), 435–455.
- [7] Marco Dorigo. 2007. Ant colony optimization. *Scholarpedia* 2, 3 (2007), 1461.
- [8] M. Dorigo, V. Maniezzo, and A. Colomi. 1996. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26, 1 (1996), 29–41. <https://doi.org/10.1109/3477.484436>
- [9] M. Dorigo, V. Maniezzo, and A. Colomi. 1996. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26, 1 (1996), 29–41. <https://doi.org/10.1109/3477.484436>
- [10] Emanuel Falkenauer. 1996. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics* 2 (1996), 5–30.
- [11] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. 2018. *The SCIP Optimization Suite 6.0*. Technical Report. Optimization Online.
- [12] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. <https://www.gurobi.com>
- [13] Mhand Hifi and Mustapha Michrafy. 2006. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society* 57, 6 (2006), 718–726.
- [14] Maryam Karimi-Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. 2022. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research* 296, 2 (2022), 393–422.
- [15] Sebastian Kraul, Markus Seizinger, and Jens O Brunner. 2023. Machine learning-supported prediction of dual variables for the cutting stock problem with an application in stabilized column generation. *INFORMS Journal on Computing* (2023).
- [16] John Levine and Frederick Ducatelle. 2004. Ant colony optimization and local search for bin packing and cutting stock problems. *Journal of the Operational Research society* 55, 7 (2004), 705–716.
- [17] Marco E Lübbecke and Jacques Desrosiers. 2005. Selected topics in column generation. *Operations research* 53, 6 (2005), 1007–1023.
- [18] François Margot. 2009. Symmetry in integer linear programming. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art* (2009), 647–686.
- [19] Michalis Mavrovouniotis, Felipe M Müller, and Shengxiang Yang. 2016. Ant colony optimization with local search for dynamic traveling salesman problems. *IEEE transactions on cybernetics* 47, 7 (2016), 1743–1756.
- [20] Mouad Morabit, Guy Desautniers, and Andrea Lodi. 2021. Machine-learning-based column selection for column generation. *Transportation Science* 55, 4 (2021), 815–831.
- [21] Albert E Fernandes Muritiba, Manuel Iori, Enrico Malaguti, and Paolo Toth. 2010. Algorithms for the bin packing problem with conflicts. *Inform Journal on computing* 22, 3 (2010), 401–415.
- [22] Ulrich Pferschy and Joachim Schauer. 2009. The knapsack problem with conflict graphs. *J. Graph Algorithms Appl.* 13, 2 (2009), 233–249.
- [23] Ulrich Pferschy and Joachim Schauer. 2017. Approximation of knapsack problems with conflict and forcing graphs. *Journal of Combinatorial Optimization* 33, 4 (2017), 1300–1323.
- [24] John Platt. 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10, 3 (1999), 61–74.
- [25] Jairo Enrique Ramírez Sánchez, Camilo Chacón Sartori, and Christian Blum. 2023. Q-Learning Ant Colony Optimization supported by Deep Learning for Target Set Selection. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) (GECCO '23). Association for Computing Machinery, New York, NY, USA, 357–366. <https://doi.org/10.1145/3583131.3590396>
- [26] Marc P Renault, Adi Rosén, and Rob van Stee. 2015. Online algorithms with advice for bin packing and scheduling problems. *Theoretical Computer Science* 600 (2015), 155–170.
- [27] David M Ryan and Brian A Foster. 1981. An integer programming approach to scheduling. *Computer scheduling of public transport urban passenger vehicle and crew scheduling* (1981), 269–280.
- [28] Ruslan Sadykov and François Vanderbeck. 2013. Bin Packing with Conflicts: A Generic Branch-and-Price Algorithm. *INFORMS Journal on Computing* 25, 2 (2013), 244–255. <https://doi.org/10.1287/ijoc.1120.0499> arXiv:<https://doi.org/10.1287/ijoc.1120.0499>
- [29] Jon E Schoenfeld. 2002. Fast, exact solution of open bin packing problems without linear programming. *Draft, US Army Space and Missile Defense Command, Huntsville, Alabama, USA* (2002).
- [30] Yunzhuang Shen, Yuan Sun, Xiaodong Li, Andrew Eberhard, and Andreas Ernst. 2023. Enhancing column generation by a machine-learning-based pricing heuristic for graph coloring. *Proceedings of the AAAI Conference on Artificial Intelligence*, 9.
- [31] Yuan Sun, Xiaodong Li, and Andreas Ernst. 2021. Using Statistical Measures and Machine Learning for Graph Reduction to Solve Maximum Weight Clique Problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 5 (2021), 1746–1760. <https://doi.org/10.1109/TPAMI.2019.2954827>
- [32] Yuan Sun, Sheng Wang, Yunzhuang Shen, Xiaodong Li, Andreas T. Ernst, and Michael Kirley. 2022. Boosting ant colony optimization via solution prediction and machine learning. *Computers & Operations Research* 143 (2022), 105769. <https://doi.org/10.1016/j.cor.2022.105769>
- [33] El-Ghazali Talbi. 2021. Machine learning into metaheuristics: A survey and taxonomy. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–32.
- [34] Roman Václavík, Antonín Novák, Přemysl Šůcha, and Zdeněk Hanzálek. 2018. Accelerating the Branch-and-Price Algorithm Using Machine Learning. *European Journal of Operational Research* 271, 3 (Dec. 2018), 1055–1069. <https://doi.org/10.1016/j.ejor.2018.05.046>
- [35] François Vanderbeck. 2000. On Dantzig-Wolfe Decomposition in Integer Programming and ways to Perform Branching in a Branch-and-Price Algorithm. *Operations Research* 48, 1 (February 2000), 111–128. <https://doi.org/10.1287/opre.48.1.111.124>
- [36] François Vanderbeck and Martin WP Savelsbergh. 2006. A generic view of Dantzig-Wolfe decomposition in mixed integer programming. *Operations Research Letters* 34, 3 (2006), 296–306.
- [37] Xiaoshu Xiang, Ye Tian, Xingyi Zhang, Jianhua Xiao, and Yaochu Jin. 2021. A pairwise proximity learning-based ant colony algorithm for dynamic vehicle routing problems. *IEEE transactions on intelligent transportation systems* 23, 6 (2021), 5275–5286.
- [38] Haoran Ye, Jiarui Wang, Zhiguang Cao, Helan Liang, and Yong Li. 2023. DeepACO: Neural-enhanced Ant Systems for Combinatorial Optimization. In *Advances in Neural Information Processing Systems*.
- [39] Jiayi Zhang, Chang Liu, Xijun Li, Hui-Ling Zhen, Mingxuan Yuan, Yawen Li, and Junchi Yan. 2023. A survey for solving mixed integer programming via machine learning. *Neurocomputing* 519 (2023), 205–217.