Enhancing Column Generation by a Machine-Learning-Based Pricing Heuristic for Graph Coloring

Yunzhuang Shen[1], Yuan Sun[2], Xiaodong Li[1], Andrew Eberhard[3], Andreas Ernst[4]
[1] School of Computing Technologies, RMIT University, Australia
[2] School of Computing and Information Systems, University of Melbourne, Australia
[3] School of Science, RMIT University, Australia
[4] School of Mathematics, Monash University, Australia
s3640365@student.rmit.edu.au, yuan.sun@unimelb.edu.au,
{xiaodong.li, andy.eberhard}@rmit.edu.au, andreas.ernst@monash.edu

Abstract

Column Generation (CG) is an effective method for solving large-scale optimization problems. CG starts by solving a subproblem with a subset of columns (i.e., variables) and gradually includes new columns that can improve the solution of the current subproblem. The new columns are generated as needed by repeatedly solving a pricing problem, which is often NP-hard and is a bottleneck of the CG approach. To tackle this, we propose a Machine-Learning-based Pricing Heuristic (MLPH) that can *generate many high-quality columns efficiently*. In each iteration of CG, our MLPH leverages an ML model to predict the optimal solution of the pricing problem, which is then used to guide a sampling method to efficiently generate multiple high-quality columns. Using the graph coloring problem, we empirically show that MLPH significantly enhances CG as compared to six state-of-the-art methods, and the improvement in CG can lead to substantially better performance of the branch-and-price exact method.

Introduction

Branch-and-price is a widely-used exact method for solving combinatorial optimization problems (Barnhart et al. 1998) in the general form of Dantzig–Wolfe reformulation (Vanderbeck 2000). This formulation often provides a much stronger Linear-Programming relaxation (LP) bound than the more compact formulations of the same problem, which may lead to a significant reduction in the problem's search space. However, solving the LP can be challenging, because it typically has an exponential number of variables (or columns) that cannot be considered all at once.

Column Generation (CG) is an iterative method for solving large-scale LPs. As illustrated in Figure 1, CG starts by solving a subproblem with a small fraction of the columns in an LP, commonly referred to as the Restricted Master Problem (RMP). Then, the optimal dual solution of the RMP is used to set up a pricing problem to search for the column with the least reduced cost. If that column has a negative reduced cost, the column is included in the RMP to further improve its solution. Otherwise, the RMP has captured all the columns with non-zero values in the optimal solution of the original LP. Since an optimal LP solution typically has only a small

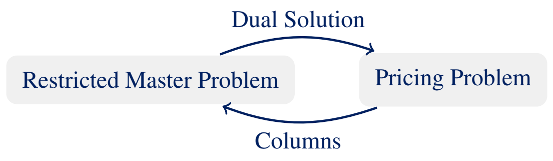

Figure 1: Illustration of the iterative process of column generation.

proportion of columns with non-zero values, CG is expected to solve the LP to optimality without the need to explicitly consider all the columns (Lübbecke 2010).

In the CG approach, repeatedly solving the pricing problem is typically a bottleneck (Lübbecke and Desrosiers 2005), because the pricing problem is often NP-hard. To attain computational advantage, heuristic methods are often preferred to trade column quality for computational efficiency. An exact method is usually used only if a heuristic method failed to generate any column with a negative reduced cost (Lübbecke 2010). Existing studies have explored a variety of pricing heuristics, such as greedy search (Mehrotra and Trick 1996; Mourgaya and Vanderbeck 2007) and metaheuristics (Taillard 1999; Malaguti, Monaci, and Toth 2011; Beheshti and Hejazi 2015). Based on past computational experience, Lübbecke (2010) notes that including multiple columns to the RMP at an iteration of CG can often speed up the progress of CG.

In this paper, we propose a novel Machine-Learning-based Pricing Heuristic (MLPH) for efficiently solving pricing problems. Specifically, we train an ML model offline using a set of solved pricing problems with known optimal solutions. For an unseen pricing problem at an iteration of CG, we use this ML model to predict the optimal solution of the pricing problem, which is then used to guide the search method to generate high-quality columns. To gain efficiency, we employ a linear Support Vector Machine (Boser, Guyon, and Vapnik 1992) for prediction and a sampling method for generating columns. As our method can potentially generate many columns, we introduce several column-selection strategies to form the new RMP to start the next iteration of CG.

By harnessing the knowledge learned from historical data, our MLPH has several advantages over existing pricing methods: (1) compared to sampling-based methods (Dorigo, Birattari, and Stützle 2006; Cai and Lin 2016), MLPH can effectively generate columns with *better* reduced costs; (2)

compared to other pricing methods (Gurobi Optimization 2018; Jiang et al. 2018; Wang, Cai, and Yin 2016), MLPH can efficiently generate *many more* columns with negative reduced costs. As our MLPH can *efficiently generate many high-quality columns*, it can help CG capture the columns in an optimal LP solution with a fewer number of iterations.

We demonstrate the efficacy of our proposed MLPH on the graph coloring problem. Our experimental study shows that MLPH can significantly accelerate the progress of CG and substantially enhance the branch-and-price exact method.

Background

In this section, we first introduce different formulations of the Graph Coloring Problem (GCP). Then, we use GCP to illustrate the solving process of CG.

Graph Coloring Problem

GCP aims to assign a minimum number of colors to vertices in a graph, such that every pair of the adjacent vertices does not share the same color (Malaguti and Toth 2010). Let $G(\mathcal{V}, \mathcal{E})$ denote a graph, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. GCP can be formulated as:

$$\min_{\boldsymbol{x},\boldsymbol{z}} \sum_{c\in\mathcal{C}} z_c, \qquad \text{(GCP-compact)} \qquad (1)$$

$$s.t. \sum_{c\in\mathcal{C}} x_{i,c} = 1, \qquad i \in \mathcal{V}, \qquad (2)$$

$$x_{i,c} + x_{j,c} \leq z_c, \qquad (i,j) \in \mathcal{E}; c \in \mathcal{C}, \qquad (3)$$

$$x_{i,c} \in \{0,1\}, \qquad i \in \mathcal{V}; c \in \mathcal{C}, \qquad (4)$$

$$z_c \in \{0,1\}, \qquad c \in \mathcal{C}. \qquad (5)$$

The binary variable z_c denotes whether a color $c \in \mathcal{C}$ is used to color the graph vertices; and $x_{i,c}$ denotes whether a certain color c is used to color the vertex indexed at i. Since this formulation has a polynomial number of variables and constraints, it is commonly referred to as the compact formulation of the GCP, i.e., GCP-compact.

Given that vertices with the same color must be part of an independent set, GCP-compact can be expressed as using a minimum number of Maximal Independent Sets (MISs) to cover all the vertices in a graph such that every vertex is covered at least once (Mehrotra and Trick 1996), which can be done systematically using Dantzig–Wolfe decomposition (Vanderbeck 2000; Vanderbeck and Savelsbergh 2006). The reformulated problem is commonly referred to as the Set Covering formulation of the GCP (GCP-SC), defined as:

$$\min_{\boldsymbol{x}} \sum_{s\in\mathcal{S}} x_s, \qquad \text{(GCP-SC)} \qquad (6)$$

$$s.t. \sum_{s\in\mathcal{S}, i\in s} x_s \geq 1, \qquad i \in \mathcal{V}, \qquad (7)$$

$$x_s \in \{0,1\}, \qquad s \in \mathcal{S}. \qquad (8)$$

The binary variable x_s indicates whether a MIS s is used to cover a graph, and $\mathcal{S}$ is the set of all the possible MISs in that graph. While GCP-SC provides a much stronger LP than GCP-compact (Mehrotra and Trick 1996), it can contain an exponential number of variables (or columns) to represent all the MISs in a graph. Hence, solving the LP of such a large-scale problem is challenging.

Column Generation

Given the LP of GCP-SC, CG aims to capture the columns with non-zero values in the optimal LP solution, starting from a RMP with a tiny fraction of the columns in the original LP:

$$\min_{\boldsymbol{x_s}} \sum_{s\in\overline{\mathcal{S}}} x_s, \qquad \text{(RMP)} \qquad (9)$$

$$s.t. \sum_{s\in\overline{\mathcal{S}}, i\in s} x_s \geq 1, \qquad i \in \mathcal{V}, \qquad (10)$$

$$0 \leq x_s \leq 1, \qquad s \in \overline{\mathcal{S}}. \qquad (11)$$

Note that the integer constraints on x_s are relaxed, and only a small number of MISs is considered initially, i.e., $\overline{\mathcal{S}} \subset \mathcal{S}$.

The RMP can be efficiently solved using the simplex method or the interior point method (Dantzig 2016), and its optimal dual solution $\boldsymbol{\pi} = [\pi_1, \cdots, \pi_{|V|}]$ associated to vertices (i.e., Constraint (10)) can be used to set up a pricing problem, to search for new MISs with the least reduced cost:

$$\min_{\boldsymbol{v}} 1 - \sum_{i\in\mathcal{V}} \pi_i \cdot v_i, \qquad \text{(MWISP)} \qquad (12)$$

$$s.t.\ v_i + v_j \leq 1, \qquad (i,j) \in \mathcal{E} \qquad (13)$$

$$v_i \in \{0,1\}, \qquad i \in \mathcal{V}. \qquad (14)$$

The binary variable v_i denotes whether the vertex i is a part of the solution, i.e., a MIS according to constraints (13) and (14). Note that the pricing problem for GCP-SC is the NP-hard Maximum Weight Independent Set Problem (MWISP), where the weight of a vertex i is its dual solution π_i to RMP.

To tackle MWISP, related studies (Mehrotra and Trick 1996; Malaguti, Monaci, and Toth 2011) employ efficient heuristic methods. Only when a heuristic method fails to find any MIS with a Negative Reduced Cost (NRC), an exact method is used to solve the MWISP to optimality and so generate the MIS with the least reduced cost. If there exist NRC MISs, they are selectively included in the RMP to further improve its solution, according to a pricing scheme (Lübbecke and Desrosiers 2005). Otherwise, the RMP has captured all the columns in the optimal solution of the original LP, and hence the original LP is optimally solved.

Machine Learning Based Pricing Heuristic

Given the MWISP at a CG iteration, we employ an ML model to predict which vertices belong to the optimal MIS. This prediction is then used to guide a sampling method to generate high-quality MISs efficiently. Having many MISs, we introduce several strategies to select a subset of these to form the new RMP at the next CG iteration.

Optimal Solution Prediction

We train an ML model to predict the optimal MIS of the MWISP by solving a binary classification task. In our training data, a training example $(\boldsymbol{f}, y)$ corresponds to a vertex in an optimally solved MWISP instance, where $\boldsymbol{f}$ denotes the feature vector that summarizes the property of the corresponding vertex and y holds a binary value of 1 (or 0) indicating whether that vertex is in the optimal MIS (or not).

We make use of several features that characterize a vertex of the MWISP, including vertex weight, vertex degree, and the upper bound of a vertex (defined by the sum of weights of that vertex and the vertices that are not adjacent to it). In addition, we adopt two statistical features (Sun, Li, and Ernst 2021) to further enhance the expressiveness of the feature representation for vertices. Given a sample of randomly generated MISs ($\boldsymbol{s} \in \mathcal{S}$), the first statistical feature measures the correlation between the presence of a vertex i and the objective values of the sample MISs,

$$f_c(i) = \frac{\sum_{k=1}^{K}(\boldsymbol{s}_i^k - \overline{\boldsymbol{s}}_i)(o^k - \overline{o})}{\sum_{k=1}^{K}\sqrt{(\boldsymbol{s}_i^k - \overline{\boldsymbol{s}}_i)^2}\sqrt{\sum_{k=1}^{K}(o^k - \overline{o})^2}}, \quad (15)$$

where $\boldsymbol{s}_i^k$ is a binary value, indicating whether the vertex i is a part of the k^{th} sample; o^k denotes the objective value of that sample; $\overline{\boldsymbol{s}}_i$ and $\overline{o}$ respectively denote, the frequency of the vertex i being in a sample and the mean objective value across all samples. A vertex with a high correlation score indicates that this vertex is likely to appear in the high-quality MISs.

The second statistical measure uses the rank r of the sample MISs with respect to their objective values,

$$f_r(i) = \sum_{k=1}^{K}\frac{\boldsymbol{s}_i^k}{r^k}. \quad (16)$$

A vertex with a high ranking score indicates that this vertex appears frequently in the high-quality MISs.

To gain computational efficiency, we adopt Support Vector Machine with linear kernel (linear-SVM) (Boser, Guyon, and Vapnik 1992) to best separate the positive examples (i.e., vertices in the optimal MIS) and negative ones (i.e., vertices not in the optimal MIS) in the training data. For a vertex i in an unseen MWISP instance, the prediction $d_i \in \mathcal{R}$ of the trained linear-SVM is the distance of this vertex, in the feature space, from the optimal decision boundary. This indicates how confidently the linear-SVM classifies this vertex as either in the optimal solution or not according to the signed distance.

Generating Columns via Sampling

Based on the ML prediction, we can build a probabilistic model to sample multiple high-quality MISs. To generate a MIS, we start with a set containing a randomly selected vertex from the graph, and then iteratively add new vertices into the set until no new vertex can be added. We compute the probability of selecting a vertex via the ML prediction d_i, $i \sim \frac{\sigma(d_i)}{\sum_{j\in\mathcal{C}}\sigma(d_j)};\ i \in \mathcal{C}$, where $\sigma(d_i)$ denotes a logistic function to re-scale the prediction of a vertex into the range of $[0, 1]$, and $\mathcal{C}$ denotes the set of candidate vertices not already adjacent to any vertex selected. The normalized value $\sigma(d_i)$ can be interpreted as the 'likelihood' that vertex is in the optimal MIS. Note that we sample MISs starting from a random vertex, so as to increase the diversity of the generated MISs, which has an impact on the solving time of CG.

Columns Selection

MLPH can potentially generate a large number of NRC columns for unseen MWISPs, and adding all of these to the RMP at early iterations of CG can slow down the solving process of the RMP in the successive CG iterations. However, selectively adding NRC columns may increase the chances of missing out the optimal columns. Therefore, we empirically investigate several strategies to form the RMP at the next CG iteration: 1) **add-all.** Add all the newly generated NRC columns to the RMP. 2) **add-partial.** From the newly generated NRC columns, select a proportion of them to add to the RMP in increasing order of their reduced cost. 3) **replace-existing.** From all the columns, sequentially select columns for the next RMP in the increasing order of their reduced costs, while maintaining the diversity of the set of selected columns by skipping columns too similar to those already added. The algorithm is outlined in the Appendix.

At one end of the spectrum, the strategy (1) adds all the columns to the RMP, resulting in the fastest growth of the size of the RMP. On the other hand, the strategy (3) replaces some of the columns in the current RMP with newly generated columns, and it can maintain a fixed number of columns in the RMP. Due to this restriction, CG may require more iterations to capture all the optimal LP columns. The strategy (2) is somewhere in between these two extremes, resulting in relatively slow growth of the size of the RMP.

Experiment Settings

Graph benchmarks and problem instance generation. We use standard Graph Coloring Benchmarks[1]. Given a graph with n vertices, the goal is to find the columns with non-zero values in the optimal LP solution of GCP-SC, starting from a RMP initialized with $10n$ randomly generated columns. We note that reducing the number of samples can affect the performance of CG negatively. Among 136 benchmark graphs, we remove those whose initial RMPs already contain all the optimal columns and whose initial RMPs cannot be solved by an LP solver within a reasonable time. For the remaining 89 graphs, we label 81 of them as 'small' and 8 of them as 'large', according to the computational time for solving their initial RMPs. For a graph, we can generate multiple RMPs by seeding the initial set of random columns, and these RMPs can be viewed as individual problem instances because solving them can result in different optimal dual solutions and hence different subsequent MWISPs. For training, we generate 10 instances on 10 small graphs with random seed $s = 1314$. For testing, we generate 24 instances on each graph using random seeds $s \in \{1, 2, \cdots, 24\}$, resulting in a total number of 1944 small instances and 192 large instances.

Data collection and training. For each training instance, we run CG using an exact, specialized solver TSM (Jiang et al. 2018) to solve MWISPs to optimality. The MWISPs with optimal solutions are recorded every five CG iterations up to the 25th iteration of CG. In the training data, the statistical features are computed from a set of n MISs, randomly sampled uniformly, with all features normalized instance-wise (Khalil

[1]https://sites.google.com/site/graphcoloring/files

et al. 2016). The parameters for training SVM are set to the default values of (Chang and Lin 2011), except that the regularization term for misclassifying positive training examples is raised to the ratio between the negative training examples and the positive ones. For tuning the hyper-parameters in the logistic function, we employ Bayesian Optimization (BO) (Snoek, Larochelle, and Adams 2012; Nogueira 2014). Specifically, BO treats the MLPH as a black-box and attempts 300 runs of MLPH using different sets of hyper-parameters in the logistic function to minimize the reduced cost of the best-found solution of the MWISP at the first CG iteration.

Compared pricing methods.

- **Gurobi,** a state-of-the-art commercial Mixed-Integer-Programming (MIP) solver (Gurobi Optimization 2018). Such a MIP solver is used as the pricing method for GCP by related work (Malaguti, Monaci, and Toth 2011). By default, Gurobi aims to solve a MWISP to optimality, hence it spends most of its computational time on improving the duality bound. In addition to this default configuration, we include another setting, Gurobi-heur, that focuses on finding feasible solutions. This is done by setting the parameters ‘PoolSearchMode’ to 2, ‘PoolSolutions’ to 10^8, and ‘Heuristics’ to 95%.
- **Ant Colony Optimization (ACO),** an efficient meta-heuristic that has been investigated for many combinatorial optimization problems (Dorigo, Birattari, and Stützle 2006). ACO maintains a probabilistic distribution during the solving process and constructs solutions by sampling from that distribution. We adopt the ACO variant as described in (Xu, Ma, and Lei 2007).
- **Specialized methods.** Since the optimal solution of a MWISP is the same as that of solving the Maximum Weight Clique Problem (MWCP) in its complementary graph, we also include three state-of-the-art MWCP solvers: 1) TSM (Jiang et al. 2018), an exact solver based on the branch-and-bound framework with domain-specific knowledge for tightening the dual bounds; 2) LSCC (Wang, Cai, and Yin 2016), a heuristic method based on Local Search; 3) Fastwclq (Cai and Lin 2016), a heuristic method that constructs solutions in a greedy fashion with respect to a benefit-estimation function.

Computational budgets, evaluation criteria, and other specifications. Table 1 shows the computational budgets for solving small and large problem instances, respectively. In addition to an overall cutoff time for CG, we also set a cutoff time for solving the MWISP at every CG iteration. In particular, the exact methods are also subject to the time limit and are evaluated as heuristic methods. Moreover, we set for each of the pricing methods an individual termination condition. For LSCC based on Local Search (LS), we terminate LSCC if it cannot find better solutions for $50n$ LS iterations. This is because, empirically, LSCC can find high-quality solutions efficiently, and providing excessive computational time can hardly improve these solutions. For MLPH, ACO, and Fastwclq, we set the number of constructed solutions to $50n$. For TSM and Gurobi, they may terminate early when solving a MWISP instance to optimality. We use ‘add-partial’ as the

Label	Total # Instances	Cutoff Time (Overall)	Cutoff Time (Pricing)	# CPUs (Paralleled)
small	1944	1800s	30s	1
large	192	8000s	150s	4

Table 1: Test instances and Computational Budget.

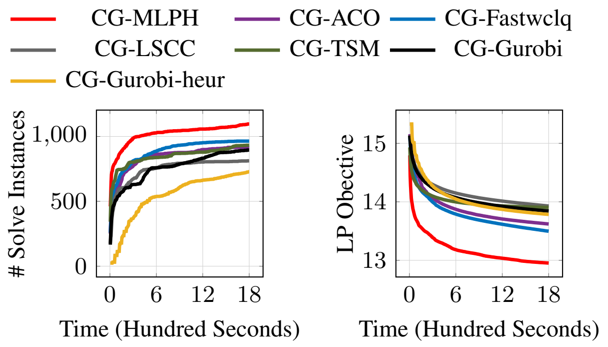

Figure 2: Results for CG with different pricing methods for solving small problem instances. **Left:** the number of solved instances. **Right:** the objective values of the RMP (the lower the better), averaged over all problem instances using the geometric mean.

default column-selection method with the column limit n. When a pricing method fails to find any NRC column, TSM is used to solve the MWISP to optimality and the optimal column is added to the RMP to start the next iteration.

We evaluate the performance of CG using a certain pricing heuristic based on two criteria, the computational time of CG for solved problem instances and the objective value of the RMP (the lower the better) for unsolved problem instances. The latter measures how close the solution of the RMP is to the optimal LP solution and so reflects the progress that CG has made. When reporting the results, we will address CG using a certain pricing method in short, e.g., CG-MLPH.

During CG the RMPs are solved by the default LP solver of Gurobi (Gurobi Optimization 2018). The experiment is conducted on a cluster with 8 nodes. Each node has 32 CPUs (AMD EPYC Processor, 2245 MHz) and 128 GB RAM. Our code is written in C/C++ and is available online[2].

Results & Analysis

Results for CG using different pricing methods. Figure 2 shows the solving statistics for small graphs. The left sub-figure shows that CG-MLPH can solve many more problem instances than CG using other pricing methods (with a given computational budget). The right sub-figure shows that CG-MLPH can make more substantial progress than the comparison methods over all test instances. Noticeably, CG-Fastwclq and CG-ACO have comparable performances, and they are better than the remaining methods we consider. To better understand the CG using different pricing methods with respect to graph characteristics, we report in Figure 3 the best method on individual graphs. Overall, we observe

[2]https://github.com/Joey-Shen/MLPH.git

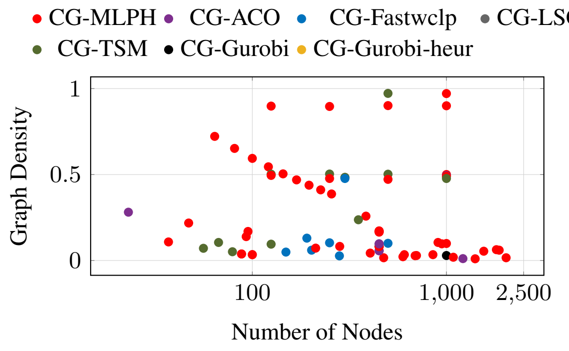

Figure 3: 81 small graphs labeled by CG with the winning pricing method. MLPH is the best pricing method for 52 graphs, followed by TSM for 12 graphs, ACO for 8 graphs, Fastwclq for 7 graphs, Gurobi for 1 graph, and LCSS for 1 graph.

Graph	# Nodes	Density	CG-MLPH	CG-ACO	CG-Gurobi	CG-Gurobi-heur	CG-Fastwclq
wap04a	5231	0.022	**44.45**	75.07	56.0	55.46	81.13
wap03a	4730	0.026	**44.87**	70.09	54.09	54.50	76.04
4-FullIns_5	4146	0.009	**6.68**	12.32	9.60	7.67	7.91
C4000.5	4000	0.500	**263.47**	264.55	304.80	304.89	286.65
wap02a	2464	0.037	**40.02** (3)	58.57	42.69	43.49	64.52
wap01a	2368	0.040	**41.0** (24)	56.21	42.71	43.34	62.13
C2000.5	2000	0.500	**137.91**	140.78	164.60	164.62	141.98
ash958GPIA	1916	0.007	**3.37**	**3.37**	3.45	3.42	3.41
Geometric Mean	-	-	**37.09**	49.20	43.08	42.15	49.49

Table 2: The mean LP objective values (the lower the better) for CG with different pricing methods for large graphs. The results are averaged over the 24 problem instances generated using the same graph. The number of solved instances (if any) is shown in brackets.

that MLPH is particularly suitable for CG on relatively larger and/or denser graphs. In contrast, Fastwclp, ACO, and TSM are only competitive on small and sparse graphs.

Table 2 shows the results on large graphs for CG. Note that the cutoff time is extended to $8,000$ seconds, and the compared pricing methods are parallelized on 4 CPUs. It can be seen that CG-MLPH achieves the best performance on every individual graph, and it outperforms the other methods substantially in most cases. In contrast, the other methods are only competitive for certain graphs. Notably, CG-MLPH can optimally solve some problem instances, while other methods cannot solve any instance to optimality.

MLPH as a competitive pricing method. For the set of 8 large graphs, Table 3 compares the performances of different pricing methods for solving the pricing problems in the initial CG iteration (to ensure the results used for compared methods are from solving the same set of MWISPs). Comparing MLPH with other pricing methods, MLPH can find many more NRC columns. Furthermore, the quality of the best-found column by MLPH is highly competitive. Subsequently, CG-MLPH achieves the best performance overall (Table 2). In addition, it can be noted that CG-MLPH solves all problem instances on the graph 'wap01a' to optimality using an average number of 36.5 iterations, while CG methods based on other pricing methods cannot optimally solve any of these, having at a minimum, an average number of 43.7 iterations. For other pricing methods, we can also observe that the good performance of CG with a certain pricing method

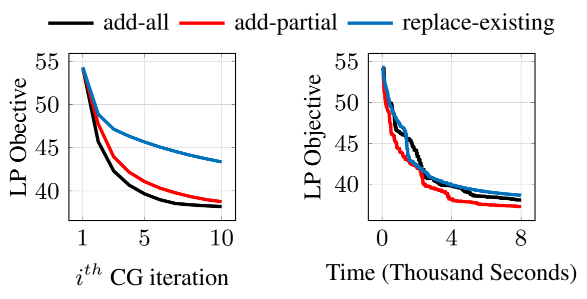

Figure 4: CG-MLPH using different column-selection strategies for large problem instances (on the left, the x-axis is the number of CG iterations; on the right, the x-axis is the wall clock time).

is often accompanied by finding many high-quality columns by that pricing method, such as ACO for dense graphs (e.g., 'C4000.5') and Fastwclq for sparse graphs (e.g., 4-FullIns_5). Similar observations can be also made from the results on small graphs as shown in the Appendix.

The results indicate that finding a large number of high-quality columns can accelerate the progress of CG. In particular, our proposed MLPH can find a large number of high-quality NRC columns, thereby helping CG obtain much better LP objective values for unsolved problem instances or spending many fewer CG iterations for solved problem instances.

Efficiency and effectiveness trade-off in column selection. As shown in the left of Figure 4, adding all the NRC columns generated at every CG iteration (i.e., 'add-all') results in the faster convergence of the LP objective. On the other hand keeping a fixed number of columns in the RMP, by replacing the existing columns already in the RMP with newly generated NRC columns (i.e., 'replace-existing'), tends to slow down the progress of CG. This shows that adding more columns can increase the chance of capturing the optimal LP columns. When measuring the progress of CG in wall-clock time as shown on the right, we observe that 'add-all' and 'replace-existing' are comparable because 'add-all' increases the computational burden for solving the fast-growing RMP. Compared to these two methods, adding a proportion (n in our case) of the best NRC columns ('add-partial') better balances the trade-off between efficiency and effectiveness.

Branch-and-price with MLPH

In this part, we use CG-MLPH to enhance Branch-and-Price (B&P), an exact method that solves a GCP to optimality by recursively decomposing the original problem (root node) into subproblems (child nodes). During the solving process, CG is used at every node to compute their LP bounds (lower bounds), and a node can be safely pruned without further expansion if its lower bound is no better than the current best-found solution.

Setup

We use the B&P code from an open-source MIP solver, SCIP (Gamrath et al. 2020). This B&P implementation incorporates specialized techniques for solving GCPs from pre-

Graph	# Nodes	Density	# Columns with Negative Reduced Costs					Minimum Reduced Cost				
			MLPH	ACO	Gurobi	Gurobi-heur	Fastwclq	MLPH	ACO	Gurobi	Gurobi-heur	Fastwclq
wap04a	5231	0.022	191494.5	277.8	5.5	23.0	0.0	-2.48	-0.37	-3.11	-2.94	N/A
wap03a	4730	0.026	234610.1	277.8	5.8	42.1	0.0	-2.44	-0.35	-3.05	-3.04	N/A
4-FullIns_5	4146	0.009	28273.2	26.3	3.0	4820.1	676.8	-4.37	-0.55	-4.37	-4.38	-3.93
C4000.5	4000	0.500	185611.9	140979.4	2.4	1.1	89.8	-0.49	-0.39	-0.24	-0.15	-0.30
wap02a	2464	0.037	123309.2	202.7	10.8	1064.8	0.0	-1.76	-0.33	-2.30	-2.29	N/A
wap01a	2368	0.040	118508.7	243.0	10.6	1132.1	0.0	-1.79	-0.36	-2.32	-2.33	N/A
C2000.5	2000	0.500	89512.1	91193.0	1.9	1.7	253.5	-0.50	-0.42	-0.23	-0.23	-0.38
ash958GPIA	1916	0.007	95888.6	1962.9	29.1	1507.0	58.0	-0.12	-0.05	-0.20	-0.20	-0.06

Table 3: Results for different pricing methods solving the MWISP at the initial CG iteration for large problem instances. The first statistic shows the number of columns with negative reduced costs that a pricing method can find. The second statistic shows the reduced cost of the best column that a pricing method can find. Both statistics are averaged over 24 problem instances generated using the same graph.

vious studies (Mehrotra and Trick 1996; Malaguti, Monaci, and Toth 2011). By default, an efficient greedy search is used as the pricing heuristic for tackling MWISPs during the CG process, and an exact method called t-clique is used to optimally solve the MWISP only when that greedy search fails to generate any NRC column. In the latter case, the optimal solution to the MWISP is used to compute the Lagrangian lower bound, which is then used to compare with the objective value of the current RMP to detect early termination of CG with optimality guarantee (Malaguti, Monaci, and Toth 2011). Once the LP at the current node is solved, the node is branched into child nodes, and the columns generated at this node are passed into the child nodes.

We refer to the default setup of B&P as B&P-def, and compare it with B&P-MLPH that replaces the greedy search in B&P-def with the MLPH for solving MWISPs. Although the greedy search has negligible computational cost, it is less effective as it can only construct a single column at a CG iteration. In contrast, our MLPH can sample many high-quality columns effectively. Empirically, we examine the sample size λ of MLPH in $\{10n, n, 0.1n\}$, and observe that no single sample size can fit all graph benchmarks. To best contrast our B&P-MLPH with B&P-def, we report the results when $\lambda = 10n$. From newly generated NRC columns, we add at most θ columns into the RMP in the increasing order of their reduced costs, $\theta = n$ for the root node and $\theta = 0.1n$ for child nodes. We observe that setting a smaller θ in column selection for child nodes can reduce the memory required for storing all the columns generated during the B&P process without sacrificing the performance. This is because child nodes often have a sufficient number of quality columns inherited from their parents and their initial RMPs are already close to the optimum.

For each method, a total number of 1584 seeded runs are performed to solve the GCPs on a set of 66 graphs in the Graph Coloring Benchmarks. The excluded graphs are either too easy (both methods can solve them within 10 seconds) or too hard to solve (both methods cannot solve the LP at the root node) under the cutoff time of 8000 seconds. When the LP at the root node is solved to optimality, we report optimality gap, defined as $Gap = 100\% \times \frac{upper\_bound - global\_lower\_bound}{upper\_bound}$. The upper bound is the objective value of the best-found solution and the global lower bound is determined by the smallest lower bound amongst the remaining open tree nodes.

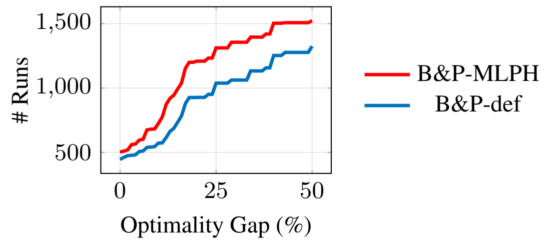

Figure 5: The number of runs where a GCP instance can be solved within a certain optimality gap threshold.

Instance	# Nodes	Density	Solving Time in Seconds	
			B&P-MLPH	B&P-def
r125.5	109	0.565	**12**	13
le450_25b	294	0.29	**14**	16
school1	355	0.603	**51**	1966
ash331GPIA	661	0.038	**53**	307
qg.order30	900	0.129	**58**	60
will199GPIA	660	0.054	**70**	128
flat300_20_0	300	0.953	**84**	343
DSJR500.1c	311	0.972	**99**	109
flat300_26_0	300	0.965	**223**	1910
le450_25a	264	0.336	14	**13**
DSJC125.9	125	0.898	32	**29**
qg.order100	10000	0.04	6259	**6127**

Table 4: Results for graphs solved by both B&P methods in all runs.

Results

Figure 5 shows the number of runs for which B&P-def and B&P-MLPH can obtain optimality gap within a certain threshold value. We can observe that 1) B&P-MLPH (red) solves GCP within a certain optimality gap in more runs than B&P-def (blue); 2) B&P-MLPH solves GCP to optimality ($Gap = 0\%$) in 482 runs, better than 444 runs by B&P-def; 3) B&P-MLPH obtains optimality gap (i.e., the LP at the root node is solved) on GCPs in 1524 runs, whereas B&P-def obtains optimality gap in 1325 runs.

Next, we report the numerical results on 36 benchmark graphs where the performance of the two compared methods are significantly different (according to the student's t-test with a significance level of 0.05). The results are grouped into Tables 4-6 based on their comparative performances. Table 4

Instance	# Nodes	Density	Gap (# Root Solved) B&P-MLPH	B&P-def
queen16_16	256	0.387	**11.1** (**24**)	N/A (0)
queen15_15	225	0.411	**11.8** (**24**)	N/A (0)
le450_25d	433	0.366	**10.7** (**24**)	N/A (0)
le450_15b	410	0.187	**6.2** (**24**)	N/A (0)
le450_25c	435	0.362	**10.7** (**24**)	N/A (0)
le450_15a	407	0.189	**6.2** (**24**)	N/A (0)
DSJC250.9	250	0.896	**2.3** (24)	4.1 (24)
wap06a	703	0.288	**4.8** (**24**)	N/A (0)
DSJC1000.9	1000	0.9	11.5 (**24**)	11.5 (17)
myciel6	95	0.338	**31.0** (24)	42.9 (24)
qg.order40	1600	0.098	**2.4** (**24**)	N/A (0)
myciel5	47	0.437	**22.9** (24)	32.6 (24)
1-Insertions_5	202	0.121	43.8 (24)	**33.3** (24)
2-Insertions_5	597	0.044	50.0 (1)	50.0 (**24**)
3-Insertions_5	1406	0.02	N/A (0)	**50.0** (**24**)
4-Insertions_4	475	0.032	40.0 (15)	40.0 (**24**)
r1000.5	966	0.989	7.8 (24)	**1.2** (24)

Table 5: Results for graphs not solved by either of the two methods.

Instance	# Nodes	Density	# Optimally Solved Runs B&P-MLPH	B&P-def
le450_5d	450	0.193	**21**	10
le450_5c	450	0.194	**21**	1
ash608GPIA	1215	0.021	**24**	0
2-Insertions_3	37	0.216	**17**	9
DSJR500.5	486	0.972	**17**	15
1-FullIns_4	38	0.364	18	**23**
queen9_9	81	0.652	0	**2**

Table 6: Results for graphs solved by the two methods in some runs.

shows the results for graphs that can be optimally solved by both methods in all runs. Here, B&P-MLPH uses less solving time than B&P-def on 9 graphs, and the speed-up is substantial on graphs such as 'school1' (38×) and 'flat300_26_0' (8×). In contrast, B&P-def performs slightly better than B&P-MLPH only on 3 graphs. Table 5 shows the results for hard graphs not solved by any method. B&P-MLPH can still solve the LP at the root node for most graphs and runs. However, B&P-def fails to solve the LP at the root node for many graphs, resulting in no optimality gap for those graphs. Table 6 shows the results for the remaining graphs. B&P-MLPH can solve GCP to optimality on more graphs and in more runs, compared to B&P-def. Overall, B&P-MLPH significantly outperforms B&P-def on 26 out of 36 graphs.

Apart from these promising results, our studies also show that MLPH can be better integrated into B&P based on certain conditions. Firstly, if the MWISP can be solved by an exact method efficiently on a graph, then it is not necessary to use MLPH (or any other pricing heuristic). In particular, on the set of 10 graphs where B&P-MLPH does not outperform B&P-def, the performance of B&P without using any pricing heuristic is on par with B&P-def's. Secondly, if the improvement of the RMP becomes very slow, i.e., the tailing-off effect (Gilmore and Gomory 1961), an exact method can be used occasionally to solve MWISP to optimality even if MLPH can still find NRC columns. This is because MLPH is likely to keep finding NRC columns, which prevents the execution of the exact method from finding the optimal column and computing the Lagrangian lower bound. This reduces the chance of an early termination of CG. When applying this condition, we observe improved results of B&P-MLPH on 12 graphs. The detailed results can be found in the Appendix.

Related Work

Machine Learning for combinatorial optimization has received a lot of attention in recent years (Bengio, Lodi, and Prouvost 2021). Existing studies have applied machine learning in a variety of ways, such as learning variable selection methods (Khalil et al. 2016; Gasse et al. 2019; Liu et al. 2020; Furian et al. 2021) or node selection methods (He, III, and Eisner 2014; Furian et al. 2021) for exact branch-and-bound solvers; learning to select the best algorithm among its alternatives based on the problem characteristics (Liberto et al. 2016; Khalil et al. 2017b); learning to determine whether to perform problem reformulation (Kruber, Lübbecke, and Parmentier 2017; Bonami, Lodi, and Zarpellon 2018) or problem reduction (Sun, Li, and Ernst 2021; Ding et al. 2020); learning primal heuristics aiming to construct an optimal solution directly (Khalil et al. 2017a; Kool, van Hoof, and Welling 2019); and learning to select columns for column generation (Morabit, Desaulniers, and Lodi 2021).

Our work is in line with the recent studies in predicting optimal solutions for combinatorial optimization problems (Li, Chen, and Koltun 2018; Sun, Li, and Ernst 2021; Sun et al. 2021; Ding et al. 2020). However, we are the first to leverage machine learning to develop a pricing heuristic for CG. More specifically, the existing machine-learning-based heuristic methods focus on *effectively finding a single best solution* (hopefully an optimal one) for a single problem instance, while our pricing heuristic aims to *efficiently generate many high-quality solutions* by solving a series of pricing problems.

Conclusion & Future Work

This paper presents a Machine-Learning-based Pricing Heuristic (MLPH) for tackling NP-hard pricing problems repeatedly encountered in the process of Column Generation (CG). Specifically, we employ Support Vector Machine with linear kernel to fast predict 'the optimal solution' for an NP-hard pricing problem, which is then adopted by a sampling-based method to *construct many high-quality columns efficiently*.

On the graph coloring problem, we demonstrate the efficacy of MLPH for solving its pricing problem - the maximum weight independent set problem. We demonstrate that MLPH can generate many more high-quality columns efficiently than existing state-of-the-art exact and heuristic methods. As a result, MLPH can significantly reduce the CG's computational time and enhance the branch-and-price exact method.

In future work, we would like to extend our MLPH method to other combinatorial optimization problems such as vehicle routing problems. Our overarching aim is to develop a *generic* ML-based pricing heuristic to speed up CG and branch-and-price for solving the Dantzig-Wolfe reformulation of combinatorial optimization problems.

References

Barnhart, C.; Johnson, E. L.; Nemhauser, G. L.; Savelsbergh, M. W. P.; and Vance, P. H. 1998. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Oper. Res.*, 46(3): 316–329.

Beheshti, A. K.; and Hejazi, S. R. 2015. A novel hybrid column generation-metaheuristic approach for the vehicle routing problem with general soft time window. *Inf. Sci.*, 316: 598–615.

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

Bonami, P.; Lodi, A.; and Zarpellon, G. 2018. Learning a Classification of Mixed-Integer Quadratic Programming Problems. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, Delft, The Netherlands, June 26-29, 2018, Proceedings*, volume 10848 of *Lecture Notes in Computer Science*, 595–604. Springer.

Boser, B. E.; Guyon, I.; and Vapnik, V. 1992. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, Pittsburgh, PA, USA, July 27-29, 1992*, 144–152. ACM.

Cai, S.; and Lin, J. 2016. Fast Solving Maximum Weight Clique Problem in Massive Graphs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, New York, NY, USA, 9-15 July 2016*, 568–574. IJCAI/AAAI Press.

Chang, C.; and Lin, C. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3): 27:1–27:27.

Dantzig, G. 2016. *Linear programming and extensions*. Princeton university press.

Ding, J.; Zhang, C.; Shen, L.; Li, S.; Wang, B.; Xu, Y.; and Song, L. 2020. Accelerating Primal Solution Findings for Mixed Integer Programs Based on Solution Prediction. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, New York, NY, USA, February 7-12, 2020*, 1452–1459. AAAI Press.

Dorigo, M.; Birattari, M.; and Stützle, T. 2006. Ant colony optimization. *IEEE Comput. Intell. Mag.*, 1(4): 28–39.

Furian, N.; O'Sullivan, M. J.; Walker, C. G.; and Çela, E. 2021. A machine learning-based branch and price algorithm for a sampled vehicle routing problem. *OR Spectr.*, 43(3): 693–732.

Gamrath, G.; Anderson, D.; Bestuzheva, K.; Chen, W.-K.; Eifler, L.; Gasse, M.; Gemander, P.; Gleixner, A.; Gottwald, L.; Halbig, K.; et al. 2020. The scip optimization suite 7.0.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 32, December 8-14, 2019, Vancouver, BC, Canada*, 15554–15566.

Gilmore, P. C.; and Gomory, R. E. 1961. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6): 849–859.

Gurobi Optimization, I. 2018. Gurobi optimizer reference manual.

He, H.; III, H. D.; and Eisner, J. 2014. Learning to Search in Branch and Bound Algorithms. In *Advances in Neural Information Processing Systems 27, December 8-13 2014, Montreal, Quebec, Canada*, 3293–3301.

Jiang, H.; Li, C.; Liu, Y.; and Manyà, F. 2018. A Two-Stage MaxSAT Reasoning Approach for the Maximum Weight Clique Problem. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 1338–1346. AAAI Press.

Khalil, E. B.; Bodic, P. L.; Song, L.; Nemhauser, G. L.; and Dilkina, B. 2016. Learning to Branch in Mixed Integer Programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, 724–731. AAAI Press.

Khalil, E. B.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017a. Learning Combinatorial Optimization Algorithms over Graphs. In *Advances in Neural Information Processing Systems 30, December 4-9, 2017, Long Beach, CA, USA*, 6348–6358.

Khalil, E. B.; Dilkina, B.; Nemhauser, G. L.; Ahmed, S.; and Shao, Y. 2017b. Learning to Run Heuristics in Tree Search. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, Melbourne, Australia, August 19-25, 2017*, 659–666. ijcai.org.

Kool, W.; van Hoof, H.; and Welling, M. 2019. Attention, Learn to Solve Routing Problems! In *7th International Conference on Learning Representations, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Kruber, M.; Lübbecke, M. E.; and Parmentier, A. 2017. Learning When to Use a Decomposition. In *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, 202–210. Springer.

Li, Z.; Chen, Q.; and Koltun, V. 2018. Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. In *Advances in Neural Information Processing Systems 31, December 3-8, 2018, Montréal, Canada*, 537–546.

Liberto, G. D.; Kadioglu, S.; Leo, K.; and Malitsky, Y. 2016. DASH: Dynamic Approach for Switching Heuristics. *Eur. J. Oper. Res.*, 248(3): 943–953.

Liu, Y.; Li, C.; Jiang, H.; and He, K. 2020. A Learning Based Branch and Bound for Maximum Common Subgraph Related Problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, New York, NY, USA, February 7-12, 2020*, 2392–2399. AAAI Press.

Lübbecke, M. E. 2010. Column generation. *Wiley encyclopedia of operations research and management science*.

Lübbecke, M. E.; and Desrosiers, J. 2005. Selected Topics in Column Generation. *Oper. Res.*, 53(6): 1007–1023.

Malaguti, E.; Monaci, M.; and Toth, P. 2011. An exact approach for the Vertex Coloring Problem. *Discret. Optim.*, 8(2): 174–190.

Malaguti, E.; and Toth, P. 2010. A survey on vertex coloring problems. *Int. Trans. Oper. Res.*, 17(1): 1–34.

Mehrotra, A.; and Trick, M. A. 1996. A Column Generation Approach for Graph Coloring. *INFORMS J. Comput.*, 8(4): 344–354.

Morabit, M.; Desaulniers, G.; and Lodi, A. 2021. Machine-Learning-Based Column Selection for Column Generation. *Transp. Sci.*, 55(4): 815–831.

Mourgaya, M.; and Vanderbeck, F. 2007. Column generation based heuristic for tactical planning in multi-period vehicle routing. *Eur. J. Oper. Res.*, 183(3): 1028–1041.

Nogueira, F. 2014. Bayesian Optimization: Open source constrained global optimization tool for Python. *URL https://github. com/fmfn/BayesianOptimization*.

Snoek, J.; Larochelle, H.; and Adams, R. P. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems 25, December 3-6, 2012, Lake Tahoe, Nevada, United States*, 2960–2968.

Sun, Y.; Ernst, A. T.; Li, X.; and Weiner, J. 2021. Generalization of machine learning for problem reduction: a case study on travelling salesman problems. *OR Spectr.*, 43(3): 607–633.

Sun, Y.; Li, X.; and Ernst, A. T. 2021. Using Statistical Measures and Machine Learning for Graph Reduction to Solve Maximum Weight Clique Problems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 43(5): 1746–1760.

Taillard, É. D. 1999. A heuristic column generation method for the heterogeneous fleet VRP. *RAIRO Oper. Res.*, 33(1): 1–14.

Vanderbeck, F. 2000. On Dantzig-Wolfe Decomposition in Integer Programming and ways to Perform Branching in a Branch-and-Price Algorithm. *Oper. Res.*, 48(1): 111–128.

Vanderbeck, F.; and Savelsbergh, M. W. P. 2006. A generic view of Dantzig-Wolfe decomposition in mixed integer programming. *Oper. Res. Lett.*, 34(3): 296–306.

Wang, Y.; Cai, S.; and Yin, M. 2016. Two Efficient Local Search Algorithms for Maximum Weight Clique Problem. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, 805–811. AAAI Press.

Xu, X.; Ma, J.; and Lei, J. 2007. An Improved Ant Colony Optimization for the Maximum Clique Problem. In Lei, J.; Yao, J.; and Zhang, Q., eds., *Third International Conference on Natural Computation, Haikou, Hainan, China, 24-27 August 2007, Volume 4*, 766–770. IEEE Computer Society.