Improving Subtour Elimination Constraint Generation in Branch-and-Cut Algorithms for the TSP with Machine Learning

Thi Quynh Trang Vo, Mourad Baiou, Viet Hung Nguyen, Paul Weng

Improving Subtour Elimination Constraint Generation in Branch-and-Cut Algorithms for the TSP with Machine Learning

Thi Quynh Trang Vo[1], Mourad Baiou[1], Viet Hung Nguyen[1*], and Paul Weng[2]

[1] LIMOS, Université Clermont Auvergne, France
[2] Shanghai Jiao Tong University, Shanghai, China
{thi_quynh_trang.vo,mourad.baiou,viet_hung.nguyen}@uca.fr,
paul.weng@sjtu.edu.cn
[*] Corresponding author

Abstract. Branch-and-Cut is a widely-used method for solving integer programming problems exactly. In recent years, researchers have been exploring ways to use Machine Learning to improve the decision-making process of Branch-and-Cut algorithms. While much of this research focuses on selecting nodes, variables, and cuts [11,9,26], less attention has been paid to designing efficient cut generation strategies in Branch-and-Cut algorithms, despite its large impact on the algorithm performance. In this paper, we focus on improving the generation of subtour elimination constraints, a core and compulsory class of cuts in Branch-and-Cut algorithms devoted to solving the Traveling Salesman Problem, which is one of the most studied combinatorial optimization problems. Our approach takes advantage of Machine Learning to address two questions before launching the separation routine to find cuts at a node of the search tree: 1) Do violated subtour elimination constraints exist? 2) If yes, is it worth generating them? We consider the former as a binary classification problem and adopt a Graph Neural Network as a classifier. By formulating subtour elimination constraint generation as a Markov decision problem, the latter can be handled through an agent trained by reinforcement learning. Our method can leverage the underlying graph structure of fractional solutions in the search tree to enhance its decision-making. Furthermore, once trained, the proposed Machine Learning model can be applied to any graph of any size (in terms of the number of vertices and edges). Numerical results show that our approach can significantly accelerate the performance of subtour elimination constraints in Branch-and-Cut algorithms for the Traveling Salesman Problem.

Keywords: Traveling Salesman Problem · Subtour elimination constraints · Branch-and-Cut · Cut generation · Machine Learning.

1 Introduction

Branch-and-Cut (B&C) is a popular method for solving integer programming (IP) problems exactly. B&C is the combination of two methods: branch-and-bound and cutting-plane. While branch-and-bound breaks down the problem

into subproblems by a divide-and-conquer strategy, the cutting-plane method tightens these subproblems by adding valid inequalities. B&C contains a sequence of decision problems such as variable selection, node selection, and cut generation. Consequently, its performance heavily depends on decision-making strategies.

One of the critical components of B&C is the cutting-plane method that strengthens the linear programming (LP) relaxations (subproblems) of the IP problem by adding valid inequalities (a.k.a. *cuts*). More precisely, given a solution $\boldsymbol{x}^*$ obtained by solving some LP relaxations of the IP problem, we solve a separation problem, which either asserts the feasibility of $\boldsymbol{x}^*$ or generates a cut violated by $\boldsymbol{x}^*$. Adding cuts can remove a large portion of the infeasible region and improve the performance. In general, cuts are categorized into general-purpose cuts obtained by the variable's integrality conditions and combinatorial cuts arising from the underlying combinatorial structure of the problem.

Generating cuts within B&C is a delicate task [8]. One of the design challenges of using cuts is balancing the separation routine's computational cost and the benefits of generated cuts. Generating cuts in a naive way can reduce the branch-and-bound tree's size but potentially increase the overall computing time due to the time spent executing the separation routine and solving the LP relaxations in the search tree. Thus, learning a deft policy for cut generation is crucial. In spite of its importance, cut generation is less studied than other related decision-making problems in B&C. To the best of our knowledge, only a few simple heuristics [4,20] have been proposed for cut generation, and concrete work has yet to be investigated to learn a cut generation policy.

In this paper, we focus on the generation of subtour elimination constraints (SECs)—a core class of cuts—for the Traveling Salesman Problem (TSP) in B&C. SECs were proposed by Dantzig, Fulkerson, and Johnson [7] to ensure the biconnectivity of solutions. They are well-known facet-defining inequalities for the TSP polytope. Due to their exponential number, SECs are usually served as cuts in the course of B&C. The separation problem of SECs is solvable in polynomial time [20] by using the Gomory-Hu procedure [10] to find a minimum cut in a graph. Although adding SECs is able to decrease the number of branching nodes, generating all possible SECs can decelerate the B&C performance, as the separation procedure of SECs is computationally expensive, especially for large-sized instances.

To improve SEC generation in B&C for the TSP, we propose an approach based on Machine Learning (ML) to handle two questions before executing the separation routine at a node of the branch-and-bound tree: 1) Do violated SECs exist? 2) If yes, is it worth generating them? The first question is to avoid solving redundant separation problems that do not provide any SEC. We treat this question as a binary classification problem and train a Graph Neural Network (GNN) in a supervised fashion. The second one is to predict the benefit of generating SECs compared to branching. To this end, we formulate the sequential decision-making process of SEC generation as a Markov decision problem and train a policy by reinforcement learning (RL). Our GNN-RL framework can

leverage the underlying graph structure of fractional solutions to predict the SEC existence and capture the context of nodes in the search tree to make SEC generation decisions. Furthermore, it offers flexibility over instance size, namely that our model can be used for any instance (of arbitrary size) while being only trained with fixed-size graphs. Experimental results show that our trained policy for SEC generation significantly accelerates the B&C performance to solve the TSP, even on instances of different sizes from its training counterparts.

2 Related work

Most approaches in the literature for cut generation exist in heuristic forms. Padberg and Rinaldi, in their research on B&C for large-scale TSP [19], empirically discovered the tailing-off phenomenon of cuts [19, Section 4.3], which shows the cut generator's inability to produce cuts that can assist the optimal LP solution to escape the corner of the polytope where it is "trapped". To deal with the tailing-off, the authors proposed to stop generating cuts if the objective value of the relaxed LP does not improve sufficiently within a given window and switch to branching. Another approach to control cut generation introduced by Balas et al. [4] is generating cuts at every k nodes of the search tree. The number k, named "skip factor" in [4], determines the frequency of generating cuts. It can be chosen either as a fixed constant or as an adaptive value varying throughout the search tree. Another commonly used strategy is the so-called cut-and-branch which only generates cuts at the root node of the search tree. Overall, despite its importance, the question of the branching versus cutting decision has yet to receive the attention it deserves.

In contrast, a closely-related problem to cut generation, *cut selection*, has been studied extensively in the literature. While cut generation decides whether to launch separation processes to generate cuts, cut selection requires selecting cuts from a candidate set obtained by solving separation problems. Cut selection is usually considered for general-purpose cuts whose separation procedure is computationally cheap and provides many cuts. Due to its definition, cut selection can be viewed as a ranking problem where cuts are sorted and chosen based on some criteria. This point of view opened up many different approaches based on many measurements of the cut quality. Among the most popular scores are efficacy [4], objective parallelism [1], and integral support [27], to name a few. Another research line on cut selection is to use ML to learn the ranking of cuts. Most works of this approach fall into two categories: supervised learning and RL. In the former, cuts are scored (or labeled) by an expert, and a cut ranking function (usually a neural network) is trained to be able to choose the best ones [13]. For the latter, one can formulate the problem of sequentially selecting cuts as a Markov decision process. An agent can then be trained to either directly optimize the objective value (RL) [26] or mimic a look-ahead expert (imitation learning) [21].

In recent years, using ML to enhance fundamental decisions in branch-and-bound is an active research domain; we refer to [5] for a summary of this line

of work. Specific examples contain learning to branch [15,2,9], learning to select nodes [11], and learning to run primal heuristics [16,6]. Similar to cut selection, these problems can be reformulated as ranking [15,13,9], regression [2], or classification problems [16], and can then be treated correspondingly. Most of these reformulations are possible due to the existence of an expensive expert (for example, the strong branching expert for variable selection), which can be used to calculate the score, label the instances, or act as an agent to be mimicked. In the case of cut generation, such an expert is too expensive to obtain. To the best of our knowledge, our paper is the first work to build an ML framework for cut generation.

3 SEC generation in B&C for the TSP

3.1 IP formulation

Given an undirected graph $G = (V, E)$ with a cost vector $\boldsymbol{c} = (c_e)_{e \in E}$ associated with E, the TSP seeks a Hamiltonian cycle (a.k.a. tour) that minimizes the total edge cost. For all edges $e \in E$, we denote by x_e a binary variable such that $x_e = 1$ if edge e occurs in the tour and $x_e = 0$ otherwise. We denote by $\delta(S)$ the set of edges that have exactly one end-vertex in $S \subset V$; $\delta(\{v\})$ is abbreviated as $\delta(v)$ for $v \in V$. Let $x(F) = \sum_{e \in F} x_e$ for $F \subseteq E$, the TSP can be formulated as an integer program as follows:

$$\min \boldsymbol{c}^T \boldsymbol{x} \tag{1a}$$

$$\text{s.t. } x(\delta(v)) = 2 \qquad \forall\, v \in V \tag{1b}$$

$$x(\delta(S)) \geq 2 \qquad \forall\, \emptyset \neq S \subset V \tag{1c}$$

$$x_e \in \{0, 1\} \qquad \forall\, e \in E \tag{1d}$$

where $\boldsymbol{x} = (x_e)_{e \in E}$. The objective function (1a) represents the total cost of edges selected in the tour. Constraints (1b) are *degree constraints* assuring that each vertex in the tour is the end-vertex of precisely two edges. Constraints (1c) are *subtour elimination constraints*, which guarantee the non-existence of cycles that visit only a proper subset of V. Finally, (1d) are *integrality constraints*.

Note that this formulation, introduced by Dantzig, Fulkerson, and Johnson [7], is widely used in most B&C algorithms for the TSP.

3.2 B&C framework for the TSP

One of the most successful approaches for exactly solving the TSP is B&C. Intuitively, B&C starts by solving a relaxation of the TSP where all SECs are omitted and the integrality constraints are relaxed to $x_e \in [0, 1]\ \forall e \in E$. At each node of the branch-and-bound tree, the LP relaxation is solved, and SECs violated by the optimal LP solution are generated as cuts through the separation routine. This principle of generating SECs is used in most B&C algorithms for the TSP, including *Concorde*—the acknowledged best exact algorithm for the TSP [3].

We denote $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ an inequality $\boldsymbol{\alpha}^T \boldsymbol{x} \leq \boldsymbol{\beta}$, $\mathcal{C}$ a set of valid inequalities for the TSP and $\langle F_0, F_1 \rangle$ an ordered pair of disjoint edge sets. Let $LP(\mathcal{C}, F_0, F_1)$ be the following LP problem:

$$
\begin{aligned}
&\min \boldsymbol{c}^T \boldsymbol{x} \\
&\text{s.t } x(\delta(v)) = 2 && \forall v \in V \\
&\quad \boldsymbol{\alpha}^T \boldsymbol{x} \leq \boldsymbol{\beta} && \forall (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathcal{C} \\
&\quad x_e = 0 && \forall e \in F_0 \\
&\quad x_e = 1 && \forall e \in F_1 \\
&\quad x_e \in [0, 1] && \forall e \in E.
\end{aligned}
$$

A basic B&C framework based on SECs is sketched as follows:

1. *Initialization.* Set $\mathcal{S} = \{\langle F_0 = \emptyset, F_1 = \emptyset \rangle\}$, $\mathcal{C} = \emptyset$, $\overline{\boldsymbol{x}} =$ NULL and $UB = +\infty$.
2. *Node selection.* If $\mathcal{S} = \emptyset$, return $\overline{\boldsymbol{x}}$ and terminate. Otherwise, select and remove an ordered pair $\langle F_0, F_1 \rangle$ from $\mathcal{S}$.
3. Solve $LP(\mathcal{C}, F_0, F_1)$. If the problem is infeasible, go to step 2; otherwise, let $\boldsymbol{x}^*$ be its optimal solution. If $\boldsymbol{c}^T \boldsymbol{x}^* \geq UB$, go to step 2.
4. *SEC verification.* If $\boldsymbol{x}^*$ is integer, verify SECs with $\boldsymbol{x}^*$. If $\boldsymbol{x}^*$ satisfies all SECs, replace $\overline{\boldsymbol{x}}$ by $\boldsymbol{x}^*$, UB by $\boldsymbol{c}^T \boldsymbol{x}^*$ and go to step 2. Otherwise, add violated SECs to $\mathcal{C}$ and go to step 3.
5. *Branching versus cut generation.* Should SECs be generated? If yes, go to step 6, else go to step 7.
6. *Cut generation.* Solve the separation problem. If violated SECs are found, add them to $\mathcal{C}$ and go to step 3.
7. *Branching.* Pick an edge e such that $0 < x_e^* < 1$. Add $\langle F_0 \cup \{e\}, F_1 \rangle$, $\langle F_0, F_1 \cup \{e\} \rangle$ to $\mathcal{S}$ and go to step 2.

When the algorithm terminates, $\overline{\boldsymbol{x}}$ is an optimal solution of the TSP. Notice that the basic B&C framework stated above simply contains fundamental steps, but it could be easily extended with additional techniques, such as the use of other valid inequality classes for cut generation, branching strategies, and primal heuristics. Any improvement for this basic B&C framework will also be valid for the extensions.

Separation routine for SECs. We now describe an exact separation algorithm to find violated SECs in polynomial time, proposed by Crowder and Padberg [20]. The input of the separation algorithm is the optimal solution $\boldsymbol{x}^*$ of the current LP relaxation. We then construct from $\boldsymbol{x}^*$ the so-called *support graph* $G_{\boldsymbol{x}^*} = (V, E_{\boldsymbol{x}^*})$ where $E_{\boldsymbol{x}^*} = \{e \in E \mid x_e^* > 0\}$. For each edge e in $E_{\boldsymbol{x}^*}$, we set x_e^* as its capacity. Due to the construction of $G_{\boldsymbol{x}^*}$, the value $x(\delta(S))$ for $S \subset V$ is precisely the capacity of the cut $(S, V \setminus S)$ in $G_{\boldsymbol{x}^*}$. Therefore, an SEC violated by $\boldsymbol{x}^*$ is equivalent to a cut with a capacity smaller than 2 in $G_{\boldsymbol{x}^*}$. Such a cut can be found by using the Gomory-Hu procedure [10] with $|V| - 1$ maximum flow computations. Thus, it is computationally expensive, especially for instances with large-sized graphs.

Note that when executing the separation routine for SECs, one can either build the Gomory-Hu tree completely and get all violated SECs from the tree or terminate the process as soon as a violated SEC is found. Our experimental results show that the former is more efficient than the latter in terms of overall solving time. Hence, in our implementation, we generate all violated SECs from the Gomory-Hu tree each time the separation routine is called.

3.3 SEC generation problem

One of the primary decisions to make in B&C for the TSP is to decide whether to generate SECs or to branch in Step 5, which has a tremendous impact on the B&C performance. On the one hand, generating SECs can help tighten the LP relaxations, reduce the number of nodes in the branch-and-bound tree, and significantly improve computing time. On the other hand, SEC generation can also worsen the B&C performance. One reason is the computational cost of the SEC separation routine, which can be time-consuming when the instance size is large. Furthermore, not all separation processes can produce violated SECs, and thus launching the separation routine when the optimal LP solution satisfies all SECs is wasteful. Another reason is that generating SECs is useless at some nodes of the search tree where additional SECs may not provide new information to improve the LP relaxation.

To illustrate the impact of SEC generation on the B&C performance, we consider the following experimental example. We solve the TSP on the instance rat195 from TSPLIB [24] by the commercial solver CPLEX 12.10 with three different SEC generation strategies in Step 5 of the basic B&C framework. In the first strategy (*No cut*), we do not generate any SECs; in the second one (*Every node*), SECs are generated at every node of the search tree. The last strategy, *Sample cut*, solves the separation problem exactly 100 times: at each node of the search tree, we will perform the separation routine with the probability 1/2 and stop doing so after solving the 100th (separation) problem. The CPU time limit is set to 3600 seconds. Table 1 shows the results of the strategies. *Sample 1* and *Sample 2* are two different runs of the strategy *Sample cut*. Column *"CPU time"* gives the running time in seconds of B&C, in which the time spent by the separation routine is shown in column *"Separation time"*. Column *"Nodes"* reports the number of nodes in the search tree, and column *"Cuts"* indicates the number of generated SECs. Columns *"Separations"* and *"Separations with cuts"* give the number of separation routine executions and the number of executions that can obtain violated SECs, respectively.

Table 1 shows that the SEC generation strategies may significantly affect the algorithm performance. Obviously, generating SECs is crucial, as B&C cannot solve the instance to optimality without it under the given CPU time budget. In addition, adding SECs substantially reduced the search tree size. However, solving SEC separation problems might take a major portion of computing time, and only a few separation executions obtained violated SECs. For example, with the strategy generating SECs at every tree node, B&C spent 98% of the CPU time to execute separation routines, but only 134 out of 2992 executions yielded

Table 1. The results of the SEC generation strategies on the instance rat195. The asterisk in the "CPU time" column indicates strategies that fail to solve the TSP within the time limit.

Strategy	CPU time	Separation time	Nodes	Cuts	Separations	Separations with cuts
No cut	3601.8*	0	1506514	0	0	0
Every node	2010.3	1340.5	4105	1116	2992	134
Sample 1	65.5	48.3	3834	359	100	21
Sample 2	114.5	39.5	10543	727	100	43

violated SECs. Table 1 also indicates that the effectiveness of the strategies relies not only on the number of solved separation problems but also on specific nodes where violated SECs are generated. Indeed, although the number of times the separation problem is solved is the same, the difference in nodes generating SECs makes the strategy *Sample 1* outperform *Sample 2*.

Motivated by this issue, in this paper, we study the SEC generation problem stated as follows: *"Given a fractional solution at a node of the search tree, decide whether to generate SECs or to branch".*

4 The GNN-RL framework for SEC generation

In this section, we describe our GNN-RL framework to learn an SEC generation strategy in B&C for the TSP. Our GNN-RL contains two separate components: a cut detector (i.e., a GNN) to predict the existence of violated SECs and a cut evaluator (i.e., a Q-value function) to decide whether to generate SECs or to branch when the GNN has predicted the existence of SECs.

Figure 1 provides the flowchart of GNN-RL at a node of the search tree. After obtaining an optimal solution to the LP relaxation at the node, the cut detector predicts whether the solution violates SECs. If it predicts that no SEC is violated, we skip to the branching step. Otherwise, the cut evaluator will assess the effectiveness of additional SECs to select the next action to perform.

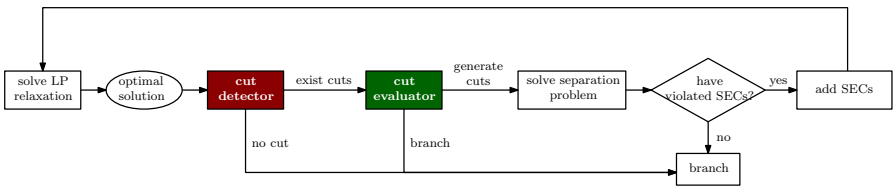

Fig. 1. The flowchart of GNN-RL

4.1 Cut detector

Given a fractional solution, the cut detector predicts whether there exists any violated SEC. Therefore, one can view the cut detector as a binary classifier that takes a fractional solution $\boldsymbol{x}$ as input and returns:

$$y = \begin{cases} 1 & \text{if there exists any SEC violated by } \boldsymbol{x}, \\ 0 & \text{otherwise.} \end{cases}$$

We adopt a GNN [25], a message-passing (MP) neural network, for this classification task to take into account the underlying graph structure of fractional solutions, which is critical for the separation problem. Furthermore, GNN possesses many properties which make it a natural choice for graph-related ML problems, such as permutation invariance or independence from the instance size.

We parameterize the cut detector as follows. Given a fractional solution $\boldsymbol{x}$, we construct from $\boldsymbol{x}$ its support graph $G_{\boldsymbol{x}} = (V, E_{\boldsymbol{x}})$ where the capacity w_e of edge e is x_e. For each node $i \in V$, we define the node feature as its degree d_i in $G_{\boldsymbol{x}}$, and embed d_i to a h-dimensional vector by a multi-layer perceptron (MLP):

$$h_i^{(0)} = \boldsymbol{W}^{(0)} d_i + b^{(0)}$$

where $\boldsymbol{W}^{(0)} \in \mathbb{R}^{h \times 1}$ and $b^{(0)} \in \mathbb{R}^h$. To update the node embeddings, we use two MP layers [18]:

$$h_i^{(l)} = \text{ReLU}\left(\boldsymbol{W}_1^{(l)} h_i^{(l-1)} + \boldsymbol{W}_2^{(l)} \sum_{j \in N(i)} w_{(i,j)} \cdot h_j^{(l-1)} \right)$$

where $h_i^{(l-1)}$ is the representation of node i in layer $l-1$, $\mathcal{N}(i)$ is the set of i's neighbors in $G_{\boldsymbol{x}}$, $\boldsymbol{W}_1^{(l)}$ and $\boldsymbol{W}_2^{(l)}$ are weight matrices in the l-th layer, and $w_{(i,j)}$ is the capacity associated with edge (i,j). To obtain a representation of the entire graph $G_{\boldsymbol{x}}$, we apply a min-cut pooling [28] layer to assign nodes into two clusters, compute the element-wise addition of all node vectors in each cluster and concatenate the two cluster vectors. Finally, an MLP with a softmax activation function is used to predict the probability of the solution's labels, i.e., $P(y = 0|\boldsymbol{x})$ and $P(y = 1|\boldsymbol{x})$.

Our training dataset $\{(\boldsymbol{x}^i, y^i)\}_{i=1}^N$ is collected by solving several separation problems on training instances generated randomly by Johnson and McGeoch's generator [14]. We train the cut detector's parameters Θ_G to minimize the cross-entropy loss:

$$L(\Theta_G) = -\sum_{i=1}^{N} \left(y^i \cdot \log P_{\Theta_G}(y^i = 1|\boldsymbol{x}^i) + (1 - y^i) \cdot \log(1 - P_{\Theta_G}(y^i = 0|\boldsymbol{x}^i)) \right).$$

4.2 Cut evaluator

We now formulate SEC generation as a Markov decision process (MDP) [22]. Considering the IP solver as the environment and the cut evaluator as the agent, we define the state space, action space, and transition and reward functions of the MDP as follows.

State space. At iteration t, a state s_t contains the information about the TSP instance and the current search tree, which comprises branching decisions, the lower and upper bounds, the LP relaxations at nodes with SECs added so far, and the considered node. A terminal state is achieved when the instance is solved to optimality.

Due to the search tree complexity, we represent a state s_t as a collection of the following elements:

- *An optimal solution $\boldsymbol{x}_t$ to the LP relaxation of the considered node.* It is used to provide information about the separation problem for the agent. We represent this solution as its corresponding support graph $G_{\boldsymbol{x}_t} = (V, E_{\boldsymbol{x}_t})$ with edge capacities $\boldsymbol{x}_t$.
- *The TSP instance.* Encoding the TSP instance in the state representation is essential, as SEC generation is naturally instance-dependent. Recall that the TSP instance is an undirected graph $G = (V, E)$ with edge costs $\boldsymbol{c}$. We define the node features as the node degrees in G. The edge features contain the edge costs and information about the variables representing edges at the considered node (values in the optimal LP solution, lower and upper bounds), which is used to encode the context of the considered node in the search tree.
- *Features of the search tree.* To enrich the information about the search tree in the state representation, we design 11 tree features based on our experimental observations and inspired by hand-crafted input features for branching variable selection proposed in [29]. The features are shown in Table 2. The top four features correspond to the incumbent existence, the IP relative gap (i.e., $|L - U|/U$ where L, U are respectively the lower and upper bounds), and the portions of processed and unprocessed nodes, which help to capture the state of the search tree. The remaining features are extracted at the considered node to describe its context through depth, objective value, optimal solution, and fixed variables. Each feature is normalized to the range $[0, 1]$.

Action space. Given a non-terminal state s_t, the RL agent selects an action a_t from the action space $\mathcal{A} = \{\textit{generate SECs, branch}\}$.

Transition. After selecting an action a_t, the new state s_{t+1} is determined as follows. If a_t is to branch, the solver selects a branching variable to create two child nodes, picks the next node to explore, and solves the corresponding LP

Table 2. The features extracted from the search tree

Feature group	Feature	Description	Ref.
Tree (4)	has_incumbent	1 if an integer feasible solution is found and 0 otherwise	
	IP_rel_gap	(upper bound - lower bound) / upper bound	[29]
	processed_nodes	the number of processed nodes / the total nodes in the current search tree	[29]
	unprocessed_nodes	the number of unprocessed nodes / the total nodes in the current search tree	[29]
Node (7)	node_depth	max(1, the node depth / |V|)	[29]
	obj_quality	objective value / upper bound	
	vars_1	the number of variables equal to 1 in the solution / |V|	
	fixed_vars	the number of fixed variables / |E|	
	unfixed_vars	the number of unfixed variables / |E|	
	vars_fixed_1	the number of variables fixed to 1 / |V|	
	vars_fixed_0	the number of variables fixed to 0 / |E|	

relaxation to get an optimal solution $\boldsymbol{x}_{t+1}$. Otherwise, if a_t is to generate SECs, the solver launches the separation routine to yield SECs violated by $\boldsymbol{x}_t$, adds them to the formulation, and solves the LP relaxation again with the new cuts to obtain $\boldsymbol{x}_{t+1}$. If no cut is found, the next state s_{t+1} is determined in the same way as when performing the branching action.

Reward function. Since we want to solve the instance as fast as possible, we consider the reduction of the IP relative gap to define the reward function. The faster the IP relative gap drops, the faster the instance is solved. Formally, let γ_t be the IP relative gap at iteration t, the reward at iteration t is defined as

$$r_t = r(s_t, a_t, s_{t+1}) = \gamma_t - \gamma_{t+1} \,. \tag{3}$$

An issue of this reward function is its sparsity, namely that most rewards are 0; thus, it rarely gives feedback to guide the agent. To deal with this issue, we add additional rewards, a.k.a *reward shaping*, to provide more frequent feedback for training the agent. In particular, to encourage the solver to terminate as soon as possible, we set penalties for each additional iteration, and each solved redundant separation problem in the cases where the cut detector predicts incorrectly. Moreover, we also give a bonus for each SEC found by the separation routine. Details of the additional rewards are shown in Table 3.

Policy parametrization. We parameterize the cut evaluator (i.e., a Q-value function) as a neural network consisting of two parts: one to embed a state into a vector and one to approximate the Q-value of actions. In the first part, we use three separate models to encode the state components, i.e., a GNN for the optimal LP solution, another GNN for the TSP instance, and an MLP for

Table 3. The additional rewards for SEC generation

Additional reward	Value
Penalty for each additional iteration	-0.01
Penalty for solving redundant separation problem	-0.10
Bonus for an SEC	0.01

the tree features. The state embedding is the concatenation of the outputs of these three models. We then pass this embedding to a 3-layer perceptron to get the Q-value approximation of actions. Figure 2 illustrates the cut evaluator architecture.

Training. To train the cut evaluator, we use the Deep Q-Network algorithm [17], i.e., the parameters of the cut evaluator are updated to minimize an L2 loss defined with a target network using data sampled from a replay buffer filled with transitions generated during online interactions with the environment. For simplicity, an ϵ-greedy policy is used for exploration.

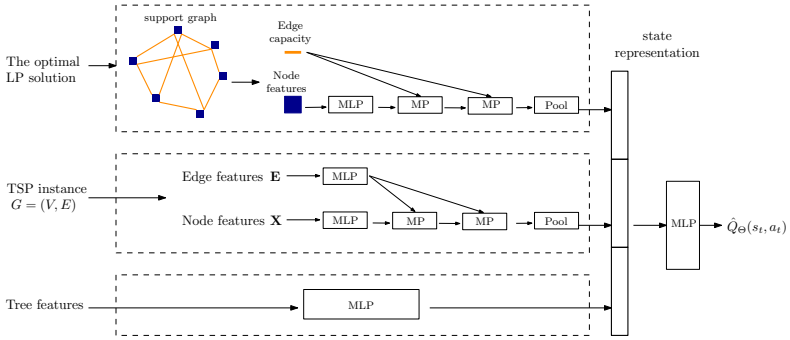

Fig. 2. The neural network architecture for the cut evaluator

5 Experiments

In this section, we demonstrate the effectiveness of the GNN-RL framework in controlling SEC generation for the TSP in B&C algorithms.

5.1 Setup

All experiments are conducted on a computing server with AMD EPYC 7742 64-core CPU, 504GB of RAM, and an RTX A4000 GPU card with 48GB graphic memory.

B&C solver. We use the commercial solver CPLEX 12.10 as a backend solver, and CPLEX UserCutCallback to generate SECs in the tree nodes and integrate our method into the solver. We keep the CPLEX's default settings which are expertly tuned. However, to focus on evaluating the benefit of SECs, we switch off the CPLEX's cuts. The solver time limit is 3600 seconds per instance.

Benchmarks. We train and evaluate our method on random TSP instances generated following Johnson and McGeoch's generator used for DIMACS TSP Challenge [14]. These instances are complete graphs. In particular, we train on 200 instances with graphs of 200 vertices and evaluate on three instance groups: instances with graphs of 200 (small), 300 (medium), and 500 vertices (large) and 100 instances per group. Furthermore, we also assess the proposed method on 30 instances with graphs of 200 to 1000 vertices from TSPLIB [24], a well-known library of sample instances for the TSP.

Neural network architecture. We describe here the model architectures for encoding the state components. For the optimal LP solution, we use a GNN with the same architecture as the cut detector without the last MLP layer. For the TSP instance, since the edge features are 4-dimensional vectors, we use an MLP layer to embed them into the same space of the node embeddings and the modified GIN architecture introduced in [12] to integrate the edge features into updating the node embeddings. Furthermore, since the used TSP instances are complete graphs, we update the embedding of a node by its 10 nearest neighbors in terms of edge costs. For the tree features, we use a two-layer perceptron model. For all architectures, the feature dimension is 64.

Training. We train the cut detector and cut evaluator separately. For the training phase of the cut detector, we generate 96000 labeled fractional solutions from 200 random instances. We train the cut detector within 100 epochs, and the learning rate is 0.0001. For the cut evaluator, we train the Q-learning network on the 200 training instances with one million steps using the package *stable-baselines3* [23].

Baselines. We compare the performance of GNN-RL with the fixed and automatic strategies proposed in [4], which generate cuts for every k nodes. For the fixed strategies, we use $k = 1$ (FS-1) as the default strategy and $k = 8$ (FS-8), which gave the best results in [4] and is one of the best skip factors in our own experiments. For the automatic strategy (AS) where the skip factor is chosen based on the instance to be solved, k is computed as follows:

$$k = \min \left\{ \text{KMAX}, \left\lceil \frac{f}{cd \log_{10} p} \right\rceil \right\}$$

where f is the number of cuts generated at the root node, d is the average distance cutoff of these cuts (a distance cutoff of a cut is the Euclidean distance

between the optimal solution and the cut [4]), p is the number of variables in the formulation, and KMAX,c are constants. In our implementation, we set KMAX= 32 as in [4] and $c = 100$ since the average distance cutoff of SECs is small. Moreover, we experimentally observe that SECs are very efficient at the root node and inefficient at the late stage of the computation. Thus, we always generate SECs at the root node and stop generating SECs when the IP relative gap is less than 1%, regardless of the strategies used.

5.2 Results

Table 4 shows the results of GNN-RL and the three baselines on both the random and TSPLIB instances. For each instance group, we report the number of instances that can be solved to optimality within the CPU time limit over the total instances (column *"Solved"*), the average CPU time in seconds (including also the running times of instances that cannot be solved to optimality within the CPU time limit) (column *"Time"*), the average number of nodes in the search tree (column *"Nodes"*), and the average number of generated SECs (column *"Cuts"*). Recall that our goal in this paper is to accelerate the B&C algorithm; thus, the main criterion for comparison is the CPU running time.

Table 4. The numerical results of the SEC generation strategies

	Strategy	Solved	CPU Time	Nodes	Cuts
SMALL	FS-1	100/100	109.9	**2769.0**	506.9
	FS-8	100/100	56.8	3090.1	493.8
	AS	100/100	48.1	3521.1	439.3
	GNN-RL	100/100	**34.4**	3185.7	**423.7**
MEDIUM	FS-1	96/100	511.1	**11969.1**	956.8
	FS-8	98/100	424.6	15983.4	970.2
	AS	96/100	441.0	26759.5	861.7
	GNN-RL	**99/100**	**288.5**	17390.6	**726.1**
LARGE	FS-1	32/100	2998.0	**37698.9**	2330.7
	FS-8	35/100	2916.4	55882.8	2425.0
	AS	33/100	2922.4	71455.1	2235.9
	GNN-RL	**37/100**	**2889.7**	72160.1	**1965.9**
TSPLIB	FS-1	**15/30**	2062.3	**15114.4**	**2412.9**
	FS-8	14/30	2056.7	19797.6	2694.7
	AS	13/30	2087.7	23202.5	2965.1
	GNN-RL	**15/30**	**1890.1**	30995.7	2622.4

As shown in Table 4, our method outperforms all the baselines on all instance groups. Indeed, GNN-RL solves more instances to optimality within a smaller average CPU time. Compared to FS-8, GNN-RL is faster by 5% on average over all random instances, i.e., is 39%, 32%, and 1% faster for small, medium, and large instances, respectively. For the TSPLIB instances, GNN-RL is faster by 9%, 8%, and 8% compared to AS, FS-8, and FS-1.

As predicted, FS-1 has the smallest tree size on average over all instances, but its running time is the highest due to the extra time spent on generating SECs.

On the other hand, too few cuts might be detrimental to the B&C performance. It can be seen in the comparison between FS-8 and AS strategies on large and medium instances. Indeed, AS requires more computing time than FS-8 despite generating fewer SECs. The numerical results give evidence that GNN-RL can balance the separation cost and the benefit of generated SECs.

6 Conclusion

In this paper, we proposed a GNN-RL framework, the first ML-based approach, to improve SEC generation in B&C for the TSP. Experimental results showed that the policy learned by GNN-RL outperformed the previously proposed heuristics for cut generation. Most importantly, the GNN-RL architecture allows the trained policy to generalize to unseen instances with larger-sized graphs. Our future work will extend this framework to other valid inequality classes for the TSP, such as the comb and 2-matching inequalities. We will also integrate GNN-RL into B&C algorithms for other combinatorial optimization problems, for example, the max-cut and vehicle routing problems.

Acknowledgements

This work has been supported in part by the program of National Natural Science Foundation of China (No. 62176154).

References

1. Tobias Achterberg. *Constraint integer programming.* PhD thesis, 2007.
2. Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.
3. David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. Concorde TSP solver, 2006.
4. Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework, 1996.
5. Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2021.
6. Antonia Chmiela, Elias Khalil, Ambros Gleixner, Andrea Lodi, and Sebastian Pokutta. Learning to schedule heuristics in branch and bound. *NeurIPS*, 34:24235–24246, 2021.
7. George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 1954.
8. Santanu S Dey and Marco Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170:237–266, 2018.
9. Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 2019.

10. Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.
11. He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 2014.
12. Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*, 2019.
13. Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123:108353, 2022.
14. DS Johnson and LA McGeoch. Benchmark code and instance generation codes. `http://dimacs.rutgers.edu/archive/Challenges/TSP/download.html`, 2002.
15. Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *AAAI*, 2016.
16. Elias B Khalil, Bistra Dilkina, George L Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Ijcai*, pages 659–666, 2017.
17. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.
18. Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI*, volume 33, pages 4602–4609, 2019.
19. Manfred Padberg and Giovanni Rinaldi. Branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 1991.
20. Manfred W Padberg and Saman Hong. *On the symmetric travelling salesman problem: a computational study*. Springer, 1980.
21. Max B Paulus, Giulia Zarpellon, Andreas Krause, Laurent Charlin, and Chris Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *ICML*, 2022.
22. Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. Wiley, 1994.
23. Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.
24. Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.
25. Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.
26. Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *ICML*. PMLR, 2020.
27. Franz Wesselmann and U Stuhl. Implementing cutting plane management and selection techniques. In *Technical Report*. University of Paderborn, 2012.
28. Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *NeurIPS*, 31, 2018.
29. Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *AAAI*, volume 35, pages 3931–3939, 2021.