Reinforcement Learning for Integer Programming: Learning to Cut

Yunhao Tang [1] **Shipra Agrawal** [1] **Yuri Faenza** [1]

Abstract

Integer programming is a general optimization framework with a wide variety of applications, e.g., in scheduling, production planning, and graph optimization. As Integer Programs (IPs) model many provably hard to solve problems, modern IP solvers rely on heuristics. These heuristics are often human-designed, and tuned over time using experience and data. The goal of this work is to show that the performance of those heuristics can be greatly enhanced using reinforcement learning (RL). In particular, we investigate a specific methodology for solving IPs, known as the cutting plane method. This method is employed as a subroutine by all modern IP solvers. We present a deep RL formulation, network architecture, and algorithms for intelligent adaptive selection of cutting planes (aka cuts). Across a wide range of IP tasks, we show that our trained RL agent significantly outperforms human-designed heuristics. Further, our experiments show that the RL agent adds meaningful cuts (e.g. resembling cover inequalities when applied to the knapsack problem), and has generalization properties across instance sizes and problem classes. The trained agent is also demonstrated to benefit the popular downstream application of cutting plane methods in Branch-and-Cut algorithm, which is the backbone of state-of-the-art commercial IP solvers.

1. Introduction

Integer Programming is a very versatile modeling tool for discrete and combinatorial optimization problems, with applications in scheduling and production planning, among others. In its most general form, an Integer Program (IP) minimizes a linear objective function over a set of integer points that satisfy a finite family of linear constraints. Classical results in polyhedral theory (see e.g. Conforti et al. (2014)) imply that *any* combinatorial optimization problem with finite feasible region can be formulated as an IP. Hence, IP is a natural model for many graph optimization problems, such as the celebrated Traveling Salesman Problem (TSP).

Due to the their generality, IPs can be very hard to solve in theory (NP-hard) and in practice. There is no polynomial time algorithm with guaranteed solutions for all IPs. It is therefore crucial to develop efficient heuristics for solving specific classes of IPs. Machine learning (ML) arises as a natural tool for tuning those heuristics. Indeed, the application of ML to discrete optimization has been a topic of significant interest in recent years, with a range of different approaches in the literature (Bengio et al., 2018)

One set of approaches focus on directly learning the mapping from an IP instance to an approximate optimal solution (Vinyals et al., 2015; Bello et al., 2017; Nowak et al., 2017; Kool and Welling, 2018). These methods implicitly learn a solution procedure for a problem instance as a function prediction. These approaches are attractive for their blackbox nature and wide applicability. At the other end of the spectrum are approaches which embed ML agents as subroutines in a problem-specific, human-designed algorithms (Dai et al., 2017; Li et al., 2018). ML is used to improve some heuristic parts of that algorithm. These approaches can benefit from algorithm design know-how for many important and difficult classes of problems, but their applicability is limited by the specific (e.g. greedy) algorithmic techniques.

In this paper, we take an approach with the potential to combine the benefits of both lines of research described above. We design a reinforcement learning (RL) agent to be used as a subroutine in a popular algorithmic framework for IP called the *cutting plane method*, thus building upon and benefiting from decades of research and understanding of this fundamental approach for solving IPs. The specific cutting plane algorithm that we focus on is Gomory's method (Gomory, 1960). Gomory's cutting plane method is guaranteed to solve *any* IP in finite time, thus our approach enjoys wide applicability. In fact, we demonstrate that our trained RL agent can even be used, in an almost blackbox manner, as a subroutine in another powerful IP method called Branch-and-Cut (B&C), to obtain significant improvements.

[1]Columbia University, New York, USA. Correspondence to: yt2541@coluymbia.edu <Yunhao>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020. Copyright 2020 by the author(s).

A recent line of work closely related to our approach includes (Khalil et al., 2016; 2017; Balcan et al., 2018), where *supervised learning* is used to improve *branching heuristics* in the Branch-and-Bound (B&B) framework for IP. To the best of our knowledge, no work on focusing on pure selection of (Gomory) cuts has appeared in the literature.

Cutting plane and B&C methods rely on the idea that every IP can be relaxed to a Linear program (LP) by dropping the integrality constraints, and efficient algorithms for solving LPs are available. Cutting plane methods iteratively add *cuts* to the LPs, which are linear constraints that can tighten the LP relaxation by eliminating some part of the feasible region, while preserving the IP optimal solution. B&C methods are based on combining B&B with cutting plane methods and other heuristics; see Section 2 for details.

Cutting plane methods have had a tremendous impact on the development of algorithms for IPs, e.g., these methods were employed to solve the first non-trivial instance of TSP (Dantzig et al., 1954). The systematic use of cutting planes has moreover been responsible for the huge speedups of IP solvers in the 90s (Balas et al., 1993; Bixby, 2017). Gomory cuts and other cutting plane methods are today widely employed in modern solvers, most commonly as a subroutine of the B&C methods that are the backbone of state-of-the-art commercial IP solvers like Gurobi and Cplex (Gurobi Optimization, 2015). However, despite the amount of research on the subject, deciding *which* cutting plane to add remains a non-trivial task. As reported in (Dey and Molinaro, 2018), "several issues need to be considered in the selection process [...] unfortunately the traditional analyses of strength of cuts offer only limited help in understanding and addressing these issues". We believe ML/RL not only can be utilized to achieve improvements towards solving IPs in real applications, but may also aid researchers in understanding effective selection of cutting planes. While modern solvers use broader classes of cuts than just Gomory's, we decided to focus on Gomory's approach because it has the nice theoretical properties seen above, it requires no further input (e.g. other human-designed cuts) and, as we will see, it leads to a well-defined and compact action space, and to clean evaluation criteria for the impact of RL.

Our contributions. We develop an RL based method for intelligent adaptive selection of cutting planes, and use it in conjunction with Branch-and-Cut methods for efficiently solving IPs. Our main contributions are the following:

- **Efficient MDP formulation.** We introduce an efficient Markov decision process (MDP) formulation for the problem of sequentially selecting cutting planes for an IP. Several trade-offs between the size of state-space/action-space vs. generality of the method were navigated in order to arrive at the proposed formulation. For example, directly formulating the B&C method as an MDP would lead to a very large state space containing all open branches. Another example is the use of Gomory's cuts (vs. other cutting plane methods), which helped limit the number of actions (available cuts) in every round to the number of variables. For some other classes of cuts, the number of available choices can be exponentially large.
- **Deep RL solution architecture design.** We build upon state-of-the-art deep learning techniques to design an efficient and scalable deep RL architecture for learning to cut. Our design choices aim to address several unique challenges in this problem. These include slow state-transition machine (due to the complexity of solving LPs) and the resulting need for an architecture that is easy to generalize, order and size independent representation, reward shaping to handle frequent cases where the optimal solution is not reached, and handling numerical errors arising from the inherent nature of cutting plane methods.
- **Empirical evaluation.** We evaluate our approach over a range of classes of IP problems (namely, packing, binary packing, planning, and maximum cut). Our experiments demonstrate significant improvements in solution accuracy as compared to popular human designed heuristics for adding Gomory's cuts. Using our trained RL policy for adding cuts in conjunction with B&C methods leads to further significant improvements, thus illustrating the promise of our approach for improving state-of-the-art IP solvers. Further, we demonstrate the RL agent's potential to learn *meaningful* and effective cutting plane strategies through experiments on the well-studied *knapsack* problem. In particular, we show that for this problem, the RL agent adds many more cuts resembling *lifted cover inequalities* when compared to other heuristics. Those inequalities are well-studied in theory and known to work well for packing problems in practice. Moreover, the RL agent is also shown to have generalization properties across instance sizes and problem classes, in the sense that the RL agent trained on instances of one size or from one problem class is shown to perform competitively for instances of a different size and/or problem class.

2. Background on Integer Programming

Integer programming. It is well known that any Integer Program (IP) can be written in the following *canonical form*

$$\min\{c^T x \,:\, Ax \le b,\, x \ge 0,\, x \in \mathbb{Z}^n\} \tag{1}$$

where x is the set of n decision variables, $Ax \le b, x \ge 0$ with $A \in \mathbb{Q}^{m\times n}, b \in \mathbb{Q}^m$ formulates the set of constraints, and the linear objective function is $c^T x$ for some $c \in \mathbb{Q}^n$. $x \in \mathbb{Z}^n$ implies we are only interested in integer solutions. Let x^*_{IP} denote the optimal solution to the IP in (1), and z^*_{IP} the corresponding objective value.

The cutting plane method for integer programming. The cutting plane method starts with solving the LP obtained from (1) by dropping the integrality constraints $x \in \mathbb{Z}^n$. This LP is called the *Linear Relaxation* (LR) of (1). Let $\mathcal{C}^{(0)} = \{x | Ax \leq b, x \geq 0\}$ be the feasible region of this LP, $x^*_{\text{LP}}(0)$ its optimal solution, and $z^*_{\text{LP}}(0)$ its objective value. Since $\mathcal{C}^{(0)}$ contains the feasible region of (1), we have $z^*_{\text{LP}}(0) \leq z^*_{\text{IP}}$. Let us assume $x^*_{\text{LP}}(0) \notin \mathbb{Z}^n$. The cutting plane method then finds an inequality $a^T x \leq \beta$ (a *cut*) that is satisfied by all integer feasible solutions of (1), but not by $x^*_{\text{LP}}(0)$ (one can prove that such an inequality always exists). The new constraint $a^T x \leq \beta$ is added to $\mathcal{C}^{(0)}$, to obtain feasible region $\mathcal{C}^{(1)} \subseteq \mathcal{C}^{(0)}$; and then the new LP is solved, to obtain $x^*_{\text{LP}}(1)$. This procedure is iterated until $x^*_{\text{LP}}(t) \in \mathbb{Z}^n$. Since $\mathcal{C}^{(t)}$ contains the feasible region of (1), $x^*_{\text{LP}}(t)$ is an optimal solution to the integer program (1). In fact, $x^*_{\text{LP}}(t)$ is the *only* feasible solution to (1) produced throughout the algorithm.

A typical way to compare cutting plane methods is by the number of cuts added throughout the algorithm: a better method is the one that terminates after adding a smaller number of cuts. However, even for methods that are guaranteed to terminate in theory, in practice often numerical errors will prevent convergence to a feasible (optimal) solution. In this case, a typical way to evaluate the performance is the following. For an iteration t of the method, the value $g^t := z^*_{\text{IP}} - z^*_{\text{LP}}(t) \geq 0$ is called the (additive) *integrality gap* of $\mathcal{C}^{(t)}$. Since $\mathcal{C}^{(t+1)} \subseteq \mathcal{C}^{(t)}$, we have that $g^t \geq g^{t+1}$. Hence, the integrality gap decreases during the execution of the cutting plane method. A common way to measure the performance of a cutting plane method is therefore given by computing the factor of integrality gap closed between the first LR, and the iteration τ when we decide to halt the method (possibly without reaching an integer optimal solution), see e.g. Wesselmann and Stuhl (2012). Specifically, we define the Integrality Gap Closure (IGC) as the ratio

$$\frac{g^0 - g^\tau}{g^0} \in [0, 1]. \tag{2}$$

In order to measure the IGC achieved by RL agent on test instances, we need to know the optimal value z^*_{IP} for those instances, which we compute with a commercial IP solver. Importantly, note that we do not use this measure, or the optimal objective value, for training, but only for evaluation.

Gomory's Integer Cuts. Cutting plane algorithms differ in how cutting planes are constructed at each iteration. Assume that the LR of (1) with feasible region $\mathcal{C}^{(t)}$ has been solved via the simplex algorithm. At convergence, the simplex algorithm returns a so-called *tableau*, which consists of a constraint matrix $\tilde{A}$ and a constraint vector $\tilde{b}$. Let $\mathcal{I}_t$ be the set of components $[x^*_{\text{LP}}(t)]_i$ that are fractional. For each $i \in \mathcal{I}_t$, we can generate a *Gomory cut* (Gomory, 1960)

$$(-\tilde{A}_{(i)} + \lfloor \tilde{A}_{(i)} \rfloor)^T x \leq -\tilde{b}_i + \lfloor \tilde{b}_i \rfloor, \tag{3}$$

where $\tilde{A}_{(i)}$ is the ith row of matrix $\tilde{A}$ and $\lfloor \cdot \rfloor$ means component-wise rounding down. Gomory cuts can therefore be generated for any IP and, as required, are valid for all integer points from (1) but not for $x^*_{\text{LP}}(t)$. Denote the set of all candidate cuts in round t as $\mathcal{D}^{(t)}$, so that $I_t := |\mathcal{D}^{(t)}| = |\mathcal{I}_t|$.

It is shown in Gomory (1960) that a cutting plane method which at each step adds an appropriate Gomory's cut terminates in a finite number of iteration. At each iteration t, we have as many as $I_t \in [n]$ cuts to choose from. As a result, the efficiency and quality of the solutions depend highly on the sequence of generated cutting planes, which are usually chosen by heuristics (Wesselmann and Stuhl, 2012). We aim to show that the choice of Gomory's cuts, hence the quality of the solution, can be significantly improved with RL.

Branch and cut. In state-of-the-art solvers, the addition of cutting planes is alternated with a *branching phase*, which can be described as follows. Let $x^*_{\text{LP}}(t)$ be the solution to the current LR of (1), and assume that some component of $x^*_{\text{LP}}(t)$, say wlog the first, is not integer (else, $x^*_{\text{LP}}(t)$ is the optimal solution to (1)). Then (1) can be split into two *subproblems*, whose LRs are obtained from $\mathcal{C}^{(t)}$ by adding constraints $x_1 \leq \lfloor [x^*_{\text{LP}}(t)]_1 \rfloor$ and $x_1 \geq \lceil [x^*_{\text{LP}}(t)]_1 \rceil$, respectively. Note that the set of feasible integer points for (1) is the union of the set of feasible integer points for the two new subproblems. Hence, the integer solution with minimum value (for a minimization IP) among those subproblems gives the optimal solution to (1). Several heuristics are used to select which subproblem to solve next, in attempt to minimize the number of suproblems (also called *child nodes*) created. An algorithm that alternates between the cutting plane method and branching is called *Branch-and-Cut* (B&C). When all the other parameters (e.g., the number of cuts added to a subproblem) are kept constant, a typical way to evaluate a B&C method is by the number of subproblems explored before the optimal solution is found.

3. Deep RL Formulation and Solution Architecture

Here we present our formulation of the cutting plane selection problem as an RL problem, and our deep RL based solution architecture.

3.1. Formulating Cutting Plane selection as RL

The standard RL formulation starts with an MDP: at time step $t \geq 0$, an agent is in a state $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$, receives an instant reward $r_t \in \mathbb{R}$ and transitions to the next state $s_{t+1} \sim p(\cdot | s_t, a_t)$. A policy $\pi : \mathcal{S} \mapsto \mathcal{P}(\mathcal{A})$

gives a mapping from any state to a distribution over actions $\pi(\cdot|s_t)$. The objective of RL is to search for a policy that maximizes the expected cumulative rewards over a horizon T, i.e., $\max_\pi \ J(\pi) := \mathbb{E}_\pi[\sum_{t=0}^{T-1} \gamma^t r_t]$, where $\gamma \in (0,1]$ is a discount factor and the expectation is w.r.t. randomness in the policy π as well as the environment (e.g. the transition dynamics $p(\cdot|s_t, a_t)$). In practice, we consider parameterized policies π_θ and aim to find $\theta^* = \arg\max_\theta J(\pi_\theta)$. Next, we formulate the procedure of selecting cutting planes into an MDP. We specify state space $\mathcal{S}$, action space $\mathcal{A}$, reward r_t and the transition $s_{t+1} \sim p(\cdot|s_t, a_t)$.

State Space $\mathcal{S}$. At iteration t, the new LP is defined by the feasible region $\mathcal{C}^{(t)} = \{a_i^T x \leq b_i\}_{i=1}^{N_t}$ where N_t is the total number of constraints including the original linear constraints (other than non-negativity) in the IP and the cuts added so far. Solving the resulting LP produces an optimal solution $x^*_{\text{LP}}(t)$ along with the set of candidate Gomory's cuts $\mathcal{D}^{(t)}$. We set the numerical representation of the state to be $s_t = \{\mathcal{C}^{(t)}, c, x^*_{\text{LP}}(t), \mathcal{D}^{(t)}\}$. When all components of $x^*_{\text{LP}}(t)$ are integer-valued, s_t is a terminal state and $\mathcal{D}^{(t)}$ is an empty set.

Action Space $\mathcal{A}$. At iteration t, the available actions are given by $\mathcal{D}^{(t)}$, consisting of all possible Gomory's cutting planes that can be added to the LP in the next iteration. The action space is discrete because each action is a discrete choice of the cutting plane. However, each action is represented as an inequality $e_i^T x \leq d_i$, and therefore is parameterized by $e_i \in \mathbb{R}^n, d_i \in \mathbb{R}$. This is different from conventional discrete action space which can be an arbitrary unrelated set of actions.

Reward r_t. To encourage adding cutting plane aggressively, we set the instant reward in iteration t to be the gap between objective values of consecutive LP solutions, that is, $r_t = c^T x^*_{\text{LP}}(t+1) - c^T x^*_{\text{LP}}(t) \geq 0$. With a discount factor $\gamma < 1$, this encourages the agent to reduce the integrality gap and approach the integer optimal solution as fast as possible.

Transition. Given state $s_t = \{\mathcal{C}^{(t)}, c, x^*_{\text{LP}}(t), \mathcal{D}^{(t)}\}$, on taking action a_t (i.e., on adding a chosen cutting plane $e_i^T x \leq d_i$), the new state s_{t+1} is determined as follows. Consider the new constraint set $\mathcal{C}^{(t+1)} = \mathcal{C}^{(t)} \cup \{e_i^T x \leq d_i\}$. The augmented set of constraints $\mathcal{C}^{(t+1)}$ form a new LP, which can be efficiently solved using the simplex method to get $x^*_{\text{LP}}(t+1)$. The new set of Gomory's cutting planes $\mathcal{D}^{(t+1)}$ can then be computed from the simplex tableau. Then, the new state $s_{t+1} = \{\mathcal{C}^{(t+1)}, c, x^*_{\text{LP}}(t+1), \mathcal{D}^{(t+1)}\}$.

3.2. Policy Network Architecture

We now describe the policy network architecture for $\pi_\theta(a_t|s_t)$. Recall from the last section we have in the state s_t a set of inequalities $\mathcal{C}^{(t)} = \{a_i^T x \leq b_i\}_{i=1}^{N_t}$, and as available actions, another set $\mathcal{D}^{(t)} = \{e_i^T x \leq d_i\}_{i=1}^{I_t}$. Given state s_t, a policy π_θ specifies a distribution over $\mathcal{D}^{(t)}$, via the following architecture.

Attention network for order-agnostic cut selection. Given current LP constraints in $\mathcal{C}^{(t)}$, when computing distributions over the I_t candidate constraints in $\mathcal{D}^{(t)}$, it is desirable that the architecture is agnostic to the ordering among the constraints (both in $\mathcal{C}^{(t)}$ and $\mathcal{D}^{(t)}$), because the ordering does not reflect the geometry of the feasible set. To achieve this, we adopt ideas from the attention network (Vaswani et al., 2017). We use a parametric function $F_\theta : \mathbb{R}^{n+1} \mapsto \mathbb{R}^k$ for some given k (encoded by a network with parameter θ). This function is used to compute projections $h_i = F_\theta([a_i, b_i]), i \in [N_t]$ and $g_j = F_\theta([e_j, d_j]), j \in [I_t]$ for each inequality in $\mathcal{C}^{(t)}$ and $\mathcal{D}^{(t)}$, respectively. Here $[\cdot,\cdot]$ denotes concatenation. The score S_j for every candidate cut $j \in [I_t]$ is computed as

$$S_j = \tfrac{1}{N_t} \sum_{i=1}^{N_t} g_j^T h_i \tag{4}$$

Intuitively, when assigning these scores to the candidate cuts, (4) accounts for each candidate's interaction with all the constraints already in the LP through the inner products $g_j^T h_i$. We then define probabilities $p_1, \ldots, p_{I_t}$ by a softmax function $\text{softmax}(S_1, \ldots, S_{I_t})$. The resulting I_t-way categorical distribution is the distribution over actions given by policy $\pi_\theta(\cdot|s_t)$ in the current state s_t.

LSTM network for variable sized inputs. We want our RL agent to be able to handle IP instances of different sizes (number of decision variables and constraints). Note that the number of constraints can vary over different iterations of a cutting plane method even for a fixed IP instance. But this variation is not a concern since the attention network described above handles that variability in a natural way. To be able to use the same policy network for instances with different number of variables , we embed each constraint using a LSTM network LSTM_θ (Hochreiter and Schmidhuber, 1997) with hidden state of size $n+1$ for a fixed n. In particular, for a general constraint $\tilde{a}_i^T \tilde{x} \leq \tilde{b}_i$ with $\tilde{a}_i \in \mathbb{R}^{\tilde{n}}$ with $\tilde{n} \neq n$, we carry out the embedding $\tilde{h}_i = \text{LSTM}_\theta([\tilde{a}_i, \tilde{b}_i])$ where $\tilde{h}_i \in \mathbb{R}^{n+1}$ is the last hidden state of the LSTM network. This hidden state $\tilde{h}_i$ can be used in place of $[\tilde{a}_i, \tilde{b}_i]$ in the attention network. The idea is that the hidden state $\tilde{h}_i$ can properly encode all information in the original inequalities $[\tilde{a}_i, \tilde{b}_i]$ if the LSTM network is powerful enough.

Policy rollout. To put everything together, in Algorithm 1, we lay out the steps involved in rolling out a policy, i.e., executing a policy on a given IP instance.

3.3. Training: Evolutionary Strategies

We train the RL agent using evolution strategies (ES) (Salimans et al., 2017). The core idea is to flatten the RL problem into a blackbox optimization problem where the input is a policy parameter θ and the output is a noisy esti-

Algorithm 1 Rollout of the Policy

Input: policy network parameter θ, IP instance parameterized by c, A, b, number of iterations T.
Initialize iteration counter $t = 0$.
Initialize minimization LP with constraints $\mathcal{C}^{(0)} = \{Ax \leq b\}$ and cost vector c. Solve to obtain $x^*_{\text{LP}}(0)$. Generate set of candidate cuts $\mathcal{D}^{(0)}$.
while $x^*_{\text{LP}}(t)$ not all integer-valued and $t \leq T$ **do**
  Construct state $s_t = \{\mathcal{C}^{(t)}, c, x^*_{\text{LP}}(t), \mathcal{D}^{(t)}\}$.
  Sample an action using the distribution over candidate cuts given by policy π_θ, as $a_t \sim \pi_\theta(\cdot|s_t)$. Here the action a_t corresponds to a cut $\{e^T x \leq d\} \in \mathcal{D}^{(t)}$.
  Append the cut to the constraint set, $\mathcal{C}^{(t+1)} = \mathcal{C}^{(t)} \cup \{e^T x \leq d\}$. Solve for $x^*_{\text{LP}}(t+1)$. Generate $\mathcal{D}^{(t+1)}$.
  Compute reward r_t.
  $t \leftarrow t + 1$.
end while

mate of the agent's performance under the corresponding policy. ES apply random sensing to approximate the policy gradient $\hat{g}_\theta \approx \nabla_\theta J(\pi_\theta)$ and then carry out the iteratively update $\theta \leftarrow \theta + \hat{g}_\theta$ for some $\alpha > 0$. The gradient estimator takes the following form

$$\hat{g}_\theta = \tfrac{1}{N} \textstyle\sum_{i=1}^{N} J(\pi_{\theta'_i}) \tfrac{\epsilon_i}{\sigma}, \quad (5)$$

where $\epsilon_i \sim \mathcal{N}(0, \mathbb{I})$ is a sample from a multivariate Gaussian, $\theta'_i = \theta + \sigma\epsilon_i$ and $\sigma > 0$ is a fixed constant. Here the return $J(\pi_{\theta'})$ can be estimated as $\sum_{t=0}^{T-1} r_t \gamma^t$ using a single trajectory (or average over multiple trajectories) generated on executing the policy $\pi_{\theta'}$, as in Algorithm 1. To train the policy on M distinct IP instances, we average the ES gradient estimators over all instances. Optimizing the policy with ES comes with several advantages, e.g., simplicity of communication protocol between workers when compared to some other actor-learner based distributed algorithms (Espeholt et al., 2018; Kapturowski et al., 2018), and simple parameter updates. Further discussions are in the appendix.

3.4. Testing

We test the performance of a trained policy π_θ by rolling out (as in Algorithm 1) on a set of test instances, and measuring the IGC. One important design consideration is that a cutting plane method can potentially cut off the optimal integer solution due to the LP solver's numerical errors. Invalid cutting planes generated by numerical errors is a well-known phenomenon in integer programming (Cornuéjols et al., 2013). Further, learning can amplify this problem. This is because an RL policy trained to decrease the cost of the LP solution might learn to aggressively add cuts in order to tighten the LP constraints. When no countermeasures were taken, we observed that the RL agent could cut the optimal solution in as many as 20% of the instances for some problems! To remedy this, we have added a simple *stopping criterion* at test time. The idea is to maintain a running statistics that measures the relative progress made by newly added cuts during execution. When a certain number of consecutive cuts have little effect on the LP objective value, we simply terminate the episode. This prevents the agent from adding cuts that are likely to induce numerical errors. Indeed, our experiments show this modification is enough to completely remove the generation of invalid cutting planes. We postpone the details to the appendix.

4. Experiments

We evaluate our approach with a variety of experiments, designed to examine the *quality* of the cutting planes selected by RL. Specifically, we conduct five sets of experiments to evaluate our approach from the different aspects:

1. **Efficiency of cuts.** Can the RL agent solve an IP problem using fewer number of Gomory cuts?
2. **Integrality gap closed.** In cases where cutting planes alone are unlikely to solve the problem to optimality, can the RL agent close the integrality gap effectively?
3. **Generalization properties.**
   - (size) Can an RL agent trained on smaller instances be applied to 10X larger instances to yield performance comparable to an agent trained on the larger instances?
   - (structure) Can an RL agent trained on instances from one class of IPs be applied to a very different class of IPs to yield performance comparable to an agent trained on the latter class?
4. **Impact on the efficiency of B&C.** Will the RL agent trained as a cutting plane method be effective as a subroutine within a B&C method?
5. **Interpretability of cuts: the knapsack problem.** Does RL have the potential to provide insights into effective and meaningful cutting plane strategies for specific problems? Specifically, for the knapsack problem, do the cuts learned by RL resemble lifted cover inequalities?

IP instances used for training and testing. We consider four classes of IPs: Packing, Production Planning, Binary Packing and Max-Cut. These represent a wide collection of well-studied IPs ranging from resource allocation to graph optimization. The IP formulations of these problems are provided in the appendix. Let n, m denote the number of variables and constraints (other than nonnegativity) in the IP formulation, so that $n \times m$ denotes the size of the IP instances (see tables below). The mapping from specific problem parameters (like number of nodes and edges in

Table 1: Number of cuts it takes to reach optimality. We show mean $\pm$ std across all test instances.

Tasks	Packing	Planning	Binary	Max Cut
Size	10×5	13×20	10×20	10×22
RANDOM	48 ± 36	44 ± 37	81 ± 32	69 ± 34
MV	62 ± 40	48 ± 29	87 ± 27	64 ± 36
MNV	53 ± 39	60 ± 34	85 ± 29	47 ± 34
LE	34 ± 17	310 ± 60	89 ± 26	59 ± 35
RL	$\mathbf{14 \pm 11}$	$\mathbf{10 \pm 12}$	$\mathbf{22 \pm 27}$	$\mathbf{13 \pm 4}$

Table 2: IGC for test instances of size roughly 1000. We show mean $\pm$ std of IGC achieved on adding $T = 50$ cuts.

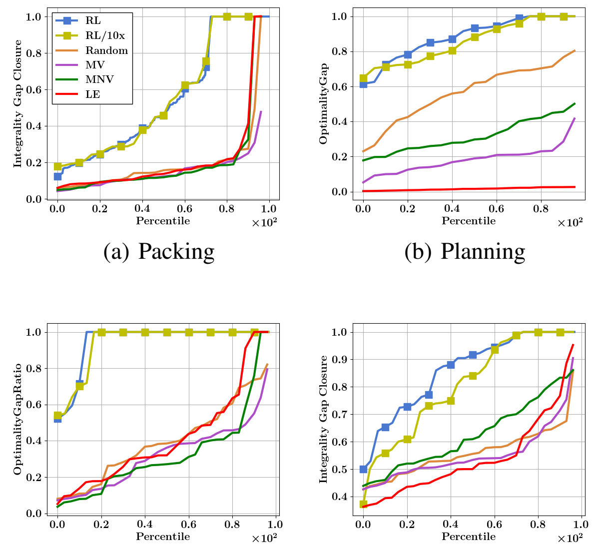
Tasks	Packing	Planning	Binary	Max Cut
Size	30×30	61×84	33×66	27×67
RAND	0.18±0.17	0.56±0.16	0.39±0.21	0.56±0.09
MV	0.14±0.08	0.18±0.08	0.32±0.18	0.55±0.10
MNV	0.19±0.23	0.31±0.09	0.32±0.24	0.62±0.12
LE	0.20±0.22	0.01±0.01	0.41±0.27	0.54±0.15
RL	$\mathbf{0.55 \pm 0.32}$	$\mathbf{0.88 \pm 0.12}$	$\mathbf{0.95 \pm 0.14}$	$\mathbf{0.86 \pm 0.14}$

maximum-cut) to n, m depends on the IP formulation used for each problem. We use randomly generated problem instances for training and testing the RL agent for each IP problem class. For the small ($n \times m \approx 200$) and medium ($n \times m \approx 1000$) sized problems we used 30 training instances and 20 test instances. These numbers were doubled for larger problems ($n \times m \approx 5000$). Importantly, note that we do not need "solved" (aka labeled) instances for training. RL only requires repeated rollouts on training instances.

Baselines. We compare the performance of the RL agent with the following commonly used human-designed heuristics for choosing (Gomory) cuts (Wesselmann and Stuhl, 2012): Random, Max Violation (MV), Max Normalized Violation (MNV) and Lexicographical Rule (LE), with LE being the original rule used in Gomory's method, for which a theoretical convergence in finite time is guaranteed. Precise descriptions of these heuristics are in the appendix.

Implementation details. We implement the MDP simulation environment for RL using Gurobi (Gurobi Optimization, 2015) as the LP solver. The C interface of Gurobi entails efficient addition of new constraints (i.e., the cut chosen by RL agent) to the current LP and solve the modified LP. The number of cuts added (i.e., the horizon T in rollout of a policy) depend on the problem size. We sample actions from the categorical distribution $\{p_i\}$ during training; but during testing, we take actions greedily as $i^* = \arg\max_i p_i$. Further implementation details, along with hyper-parameter settings for the RL method are provided in the appendix.

Experiment #1: Efficiency of cuts (small-sized instances). For small-sized IP instances, cutting planes alone can potentially solve an IP problem to optimality. For such instances, we compare different cutting plane methods on the total number of cuts it takes to find an optimal integer solution. Table 1 shows that the RL agent achieves close to several factors of improvement in the number of cuts required, when compared to the baselines. Here, for each class of IP problems, the second row of the table gives the size of the IP formulation of the instances used for training and testing.



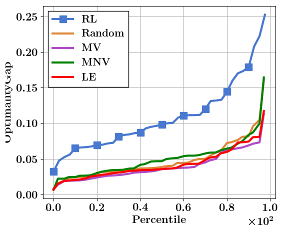
(a) Packing (b) Planning (c) Binary Packing (d) Max Cut

Figure 2: Percentile plots of IGC for test instances of size roughly 1000. X-axis shows the percentile of instances and y-axis shows the IGC achieved on adding $T = 50$ cuts. Across all test instances, RL achieves significantly higher IGC than the baselines.

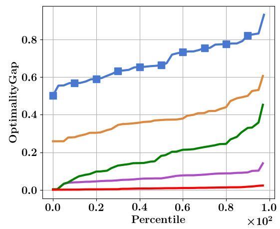
Experiment #2: Integrality gap closure for large-sized instances. Next, we train and test the RL agent on significantly larger problem instances compared to the previous experiment. In the first set of experiments (Table 2 and Figure 2), we consider instances of size ($n \times m$) close to 1000. In Table 3 and Figure 3, we report results for even larger scale problems, with instances of size close to 5000. We add $T = 50$ cuts for the first set of instances, and $T = 250$ cuts for the second set of instances. However, for these instances, the cutting plane methods is unable to reach optimality. Therefore, we compare different cutting plane methods on integrality gap closed using the IGC metric defined in (2), Section 2. Table 2, 3 show that on average RL agent was able to close a significantly higher fraction of gap compared to the other methods. Figure 2, 3 provide a more detailed comparison, by showing a percentile plot – here the instances are sorted in the ascending order of IGC and then

Table 3: IGC for test instances of size roughly 5000. We show mean $\pm$ std of IGC achieved on adding $T = 250$ cuts.

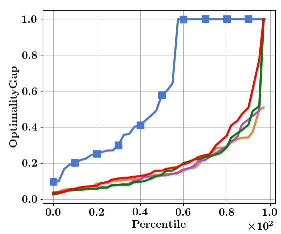
Tasks	**Packing**	**Planning**	**Binary**	**Max Cut**
Size	60×60	121×168	66×132	54×134
RANDOM	0.05 ± 0.03	0.38 ± 0.08	0.17 ± 0.12	0.50 ± 0.10
MV	0.04 ± 0.02	0.07 ± 0.03	0.19 ± 0.18	0.50 ± 0.06
MNV	0.05 ± 0.03	0.17 ± 0.10	0.19 ± 0.18	0.56 ± 0.11
LE	0.04 ± 0.02	0.01 ± 0.01	0.23 ± 0.20	0.45 ± 0.08
RL	$\mathbf{0.11\pm0.05}$	$\mathbf{0.68\pm0.10}$	$\mathbf{0.61\pm0.35}$	$\mathbf{0.57\pm0.10}$



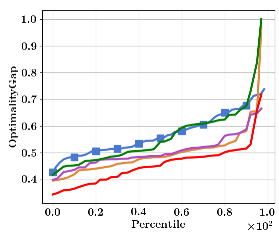
(a) Packing



(b) Planning



(c) Binary Packing



(d) Max Cut

Figure 3: Percentile plots of IGC for test instances of size roughly 5000, $T = 250$ cuts. Same set up as Figure 2 but on even larger-size instances.

plotted in order; the y-axis shows the IGC and the x-axis shows the percentile of instances achieving that IGC. The blue curve with square markers shows the performance of our RL agent. In Figure 2, very close to the blue curve is the yellow curve (also with square marker). This yellow curve is for RL/10X, which is an RL agent trained on 10X smaller instances in order to evaluate generalization properties, as we describe next.

Experiment #3: Generalization. In Figure 2, we also demonstrate the ability of the RL agent to generalize across different sizes of the IP instances. This is illustrated through the extremely competitive performance of the RL/10X agent, which is trained on 10X smaller size instances than the test instances. (Exact sizes used in the training of RL/10X agent were were $10 \times 10, 32 \times 22, 10 \times 20, 20 \times 10$, respectively, for the four types of IP problems.) Furthermore, we test generalizability across IP classes by training an RL agent on

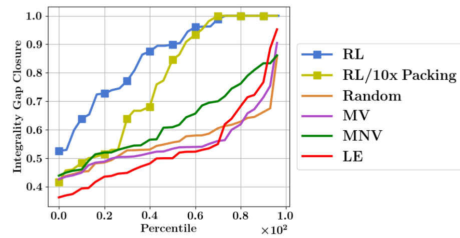
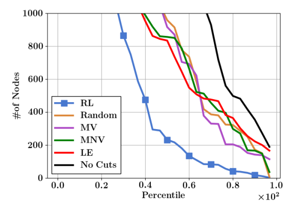


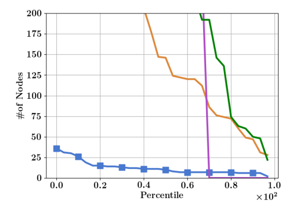
Figure 4: Percentile plots of Integrality Gap Closure. 'RL/10X packing' trained on instances of a completely different IP problem (packing) performs competitively on the maximum-cut instances.

Table 4: IGC in B&C. We show mean $\pm$ std across test instances.

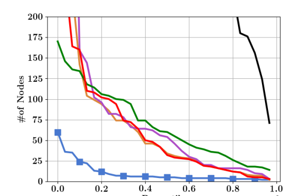
Tasks	**Packing**	**Planning**	**Binary**	**Max Cut**
Size	30×30	61×84	33×66	27×67
NO CUT	0.57 ± 0.34	0.35 ± 0.08	0.60 ± 0.24	1.0 ± 0.0
RANDOM	0.79 ± 0.25	0.88 ± 0.16	0.97 ± 0.09	1.0 ± 0.0
MV	0.67 ± 0.38	0.64 ± 0.27	0.97 ± 0.09	0.97 ± 0.18
MNV	0.83 ± 0.23	0.74 ± 0.22	1.0 ± 0.0	1.0 ± 0.0
LE	0.80 ± 0.26	0.35 ± 0.08	0.97 ± 0.08	1.0 ± 0.0
RL	$\mathbf{0.88\pm0.23}$	$\mathbf{1.0\pm0.0}$	1.0 ± 0.0	1.0 ± 0.0



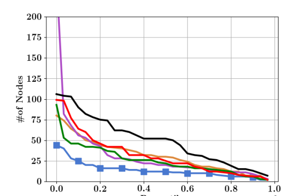
(a) Packing (1000 nodes)



(b) Planning (200 node)



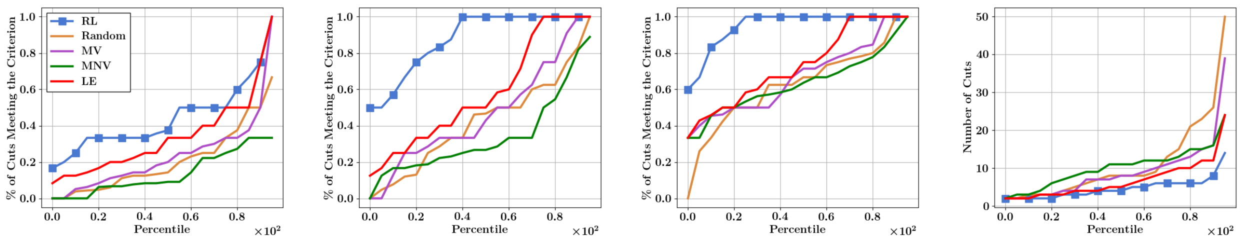
(c) Binary (200 nodes)



(d) Max Cut (200 nodes)

Figure 5: Percentile plots of number of B&C nodes expanded. X-axis shows the percentile of instances and y-axis shows the number of expanded nodes to close 95% of the integrality gap.

small sized instances of the *packing problem*, and applying it to add cuts to $10X$ larger instances of the *maximum-cut problem*. The latter, being a graph optimization problem, has intuitively a very different structure from the former. Figure 4 shows that the RL/10X agent trained on packing (yellow curve) achieve a performance on larger maximum-cut instances that is comparable to the performance of agent



(a) Criterion 1 (b) Criterion 2 (c) Criterion 3 (d) Number of cuts

Figure 6: Percentage of cuts meeting the designed criteria and number of cuts on Knapsack problems. We train the RL agent on 80 instances. All baselines are tested on 20 instances. As seen above, RL produces consistently more 'high-quality' cuts.

trained on the latter class (blue curve).

Experiment #4: Impact on the efficiency of B&C. In practice, cutting planes alone are not sufficient to solve large problems. In state-of-the-art solvers, the iterative addition of cutting planes is alternated with a branching procedure, leading to Branch-and-Cut (B&C). To demonstrate the full potential of RL, we implement a comprehensive B&C procedure but without all the additional heuristics that appear in the standard solvers. Our B&C procedure has two hyper-parameters: number of child nodes (suproblems) to expand N_{exp} and number of cutting planes added to each node N_{cuts}. In addition, B&C is determined by the implementation of the *Branching Rule*, *Priority Queue* and *Termination Condition*. Further details are in the appendix.

Figure 5 gives percentile plots for the number of child nodes (suproblems) N_{exp} until termination of B&C. Here, $N_{\text{cuts}} = 10$ cuts were added to each node, using either RL or one of the baseline heuristics. We also include as a comparator, the B&C method without any cuts, i.e., the branch and bound method. The trained RL agent and the test instances used here are same as those in Table 2 and Figure 2. Perhaps surprisingly, the RL agent, though not designed to be used in combination with branching, shows substantial improvements in the efficiency of B&C. In the appendix, we have also included experimental results showing improvements for the instances used in Table 3 and Figure 3.

Experiment #5: Interpretability of cuts. A *knapsack* problem is a binary packing problem with only one constraint. Although simple to state, these problems are NP-Hard, and have been a testbed for many algorithmic techniques, see e.g. the books (Kellerer et al., 2003; Martello and Toth, 1990). A prominent class of valid inequalities for knapsack is that of *cover inequalities*, that can be strengthened through the classical *lifting operation* (see the appendix for definitions). Those inequalities are well-studied in theory and also known to be effective in practice, see e.g. (Crowder et al., 1983; Conforti et al., 2014; Kellerer et al., 2003). Our last set of experiments gives a "reinforcement learning validation" of those cuts. We show in fact that RL, with the same reward scheme as in Experiment #2, produces many more cuts that "almost look like" lifted cover inequalities than the baselines. More precisely, we define three increasingly looser criteria for deciding when a cut is "close" to a lifted cover inequality (the plurality of criteria is due to the fact that lifted cover inequalities can be strengthened in many ways). We then check which percentage of the inequalities produced by the RL (resp. the other baselines) satisfy each of these criteria. This is reported in the first three figures in Figure 6, together with the number of cuts added before the optimal solution is reached (rightmost figure in Figure 6). More details on the experiments and a description of the three criteria are reported in the appendix. These experiments suggest that our approach could be useful to aid researchers in the discovery of strong family of cuts for IPs, and provide yet another empirical evaluation of known ones.

Runtime. A legitimate question is whether the improvement provided by the RL agent in terms of solution accuracy comes at the cost of a large runtime. The training time for RL can indeed be significant, especially when trained on large instances. However, there is no way to compare that with the human-designed heuristics. In testing, we observe no significant differences in time required by an RL policy to choose a cut vs. time taken to execute a heuristic rule. We detail the runtime comparison in the appendix.

5. Conclusions

We presented a deep RL approach to automatically learn an effective cutting plane strategy for solving IP instances. The RL algorithm learns by trying to solve a pool of (randomly generated) training instances again and again, without having access to any solved instances. The variety of tasks across which the RL agent is demonstrated to generalize without being trained for, provides evidence that it is able to learn an intelligent algorithm for selecting cutting planes. We believe our empirical results are a convincing step forward towards the integration of ML techniques in IP solvers. This may lead to a functional answer to the "Hamletic question Branch-and-cut designers often have to face: to cut or not to cut?" (Dey and Molinaro, 2018).

Acknowledgements. Author Shipra Agrawal acknowledges support from an Amazon Faculty Research Award.

References

Balas, E., Ceria, S., and Cornuéjols, G. (1993). A lift-and-project cutting plane algorithm for mixed 0–1 programs. *Mathematical programming*, 58(1-3):295–324.

Balcan, M.-F., Dick, T., Sandholm, T., and Vitercik, E. (2018). Learning to branch. *arXiv preprint arXiv:1803.10150*.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural combinatorial optimization.

Bengio, Y., Lodi, A., and Prouvost, A. (2018). Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*.

Bixby, B. (2017). Optimization: past, present, future. Plenary talk at INFORMS Annual Meeting.

Conforti, M., Cornuéjols, G., and Zambelli, G. (2014). *Integer programming*, volume 271. Springer.

Cornuéjols, G., Margot, F., and Nannicini, G. (2013). On the safety of gomory cut generators. *Math. Program. Comput.*, 5(4):345–395.

Crowder, H., Johnson, E. L., and Padberg, M. (1983). Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*.

Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410.

Dey, S. S. and Molinaro, M. (2018). Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170(1):237–266.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*.

Gomory, R. (1960). An algorithm for the mixed integer problem. Technical report, RAND CORP SANTA MONICA CA.

Gurobi Optimization, I. (2015). Gurobi optimizer reference manual. *URL http://www. gurobi. com*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., and Dabney, W. (2018). Recurrent experience replay in distributed reinforcement learning.

Kellerer, H., Pferschy, U., and Pisinger, D. (2003). Knapsack problems. 2004.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358.

Khalil, E. B., Le Bodic, P., Song, L., Nemhauser, G. L., and Dilkina, B. N. (2016). Learning to branch in mixed integer programming. In *AAAI*, pages 724–731.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kool, W. and Welling, M. (2018). Attention solves your tsp. *arXiv preprint arXiv:1803.08475*.

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*, pages 539–548.

Martello, S. and Toth, P. (1990). Knapsack problems: algorithms and computer implementations. *Wiley-Interscience series in discrete mathematics and optimiza tion*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

Nowak, A., Villar, S., Bandeira, A. S., and Bruna, J. (2017). A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint arXiv:1706.07450*.

Pochet, Y. and Wolsey, L. A. (2006). *Production planning by mixed integer programming*. Springer Science & Business Media.

Rothvoß, T. and Sanità, L. (2017). 0/1 polytopes with quadratic chvátal rank. *Operations Research*, 65(1):212–220.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700.

Wesselmann, F. and Stuhl, U. (2012). Implementing cutting plane management and selection techniques. Technical report, Tech. rep., University of Paderborn.