A. Experiment Details

A.1. Projection into the original variable space

In the following we look at only the first iteration of the cutting plane procedure, and we drop the iteration index t. Recall the LP relaxation of the original IP problem (1), where $A \in \mathbb{Q}^{m \times n}, b \in \mathbb{Q}^m$:

$$\begin{cases} \min c^T x \\ Ax \leq b \\ x \geq 0. \end{cases}$$

When a simplex algorithm solves the LP, the original LP is first converted to a standard form where all inequalities are transformed into equalities by introducing slack variables.

$$\begin{cases} \min c^T x \\ Ax + \mathbb{I}s = b \\ x \geq 0, s \geq 0, \end{cases} \tag{6}$$

where $\mathbb{I}$ is an identity matrix and s is the set of slack variables. The simplex method carries out iteratively operations on the tableau formed by $[A, \mathbb{I}], b$ and c. At convergence, the simplex method returns a final optimal tableau. We generate a Gomory's cut using the row of the tableau that corresponds to a fractional variable of the optimal solution x^*_{LP}. This will in general create a cutting plane of the following form

$$e^T x + r^T s \leq d \tag{7}$$

where $e, x \in \mathbb{R}^n, r, s \in \mathbb{R}^m$ and $d \in \mathbb{R}$. Though this cutting plane involves slack variables, we can get rid of the slack variables by multiplying both sides of the linear constraints in (6) by r

$$r^T Ax + r^T s = r^T b \tag{8}$$

and subtract the new cutting plane (7) by the above. This leads to an equivalent cutting plane

$$(e^T - r^T A)x \leq d - r^T b. \tag{9}$$

Note that this cutting plane only contains variables in the original variable space. For a downstream neural network that takes in the parameters of the cutting planes as inputs, we find it helpful to remove such slack variables. Slack variables do not contribute to new information regarding the polytope and we can also parameterize a network with a smaller number of parameters.

A.2. Integer programming formulations of benchmark problems

A wide range of benchmark instances can be cast into special cases of IP problems. We provide their specific formulations below. For simplicity, we only provide their general IP formulations (with $\leq, \geq, =$ constraints). It is always possible to convert original formulations into the standard formulation (1) with properly chosen A, b, c, x. Some problems are formulated within a graph $G = (V, E)$ with nodes $v \in V$ and edges $(v, u) \in E$.

Their formulations are as follows:

Max Cut. We have one variable per edge $y_{u,v}, (u, v) \in E$ and one variable per node $x_u, u \in V$. Let $w_{u,v} \geq 0$ be a set of non-negative weights per edge.

$$\begin{cases} \max \sum_{(u,v) \in E} w_{uv} y_{uv} \\ y_{uv} \leq x_u + x_v, \forall (u, v) \in E \\ y_{uv} \leq 2 - x_u - x_v, \forall (u, v) \in E \\ 0 \leq x, y \leq 1 \\ x_u, y_{uv} \in \mathbb{Z} \quad \forall u \in V, (u, v) \in E. \end{cases} \tag{10}$$

In our experiments the graphs are randomly generated. To be specific, we specify a vertex size $|V|$ and edge size $|E|$. We then sample $|E|$ edges from all the possible $|V| \cdot (|V| - 1)/2$ edges to form the final graph. The weights w_{uv} are uniformly sampled as an integer from 0 to 10. When generating the instances, we sample graphs such that $|V|, |E|$ are of a particular size. For example, for middle size problem we set $|V| = 7, |E| = 20$.

Packing. The packing problem takes the generic form of (1) while requiring that all the coefficients of A, b, c be non-negative, in order to enforce proper resource constraints.

Here the constraint coefficients a_{ij} for the jth variable and ith constraint is sampled as an integer uniformly from 0 and 5. Then the RHS coefficient b_i is sampled from $9n$ to $10n$ uniformly as an integer where n is the number of variables. Each component of c_j is uniformly sampled as an integer from 1 to 10.

Binary Packing. Binary packing augments the original packing problem by a set of binary constraints on each variable $x_i \leq 1$.

Here the constraint coefficients a_{ij} for the jth variable and ith constraint is sampled as an integer uniformly from 5 and 30. Then the RHS coefficient b_i is sampled from $10n$ to $20n$ uniformly as an integer where n is the number of variables. Each component of c_j is uniformly sampled as an integer from 1 to 10.

Production Planning. Consider a production planning problem (Pochet and Wolsey, 2006) with time horizon T. The decision variables are production $x_i, 1 \leq i \leq T$, along with by produce / not produce variables $y_i, 1 \leq i \leq T$ and storage variables $s_i, 0 \leq i \leq T$. Costs p_i', h_i', q_i and demands d_i are given as problem parameters. The LP formulation is as follows

$$\begin{cases} \min \sum_{i=1}^{T} p_i' x_i + \sum_{i=0}^{T} h_i' s_i + \sum_{i=0}^{T} q_i y_i \\ s_{i-1} + x_i = d_i + s_i, \forall 1 \leq i \leq T \\ x_i \leq M y_i, \forall 1 \leq i \leq T \\ s \geq 0, x \geq 0, 0 \leq y \leq 1 \\ s_0 = s_0^*, s_T = s_T^* \\ x, s, y \in \mathbb{Z}^T, \end{cases} \tag{11}$$

where M is a positive large number and s_0^*, s_T^* are also given.

The instance parameters are the initial storage $s_0^* = 0$, final storage $s_T^* = 20$ and big $M = 100$. The revenue parameter p_i', h_i', q_i are generated uniformly random as integers from 1 to 10.

Size of IP formulations. In our results, we describe the sizes of the IP instances as $n \times m$ where n is the number of columns and m is the number of rows of the constraint matrix A from the LR of (1). For a packing problem with n items and m resource constraints, the IP formulation has n variables and m constraints; for planning with period K, $n = 3K + 1$, $m = 4K + 1$; for binary packing, there are n extra binary constraints compared to the packing problem; for max-cut, the problem is defined on a graph with a vertex set V and an edge set V, and its IP formulation consists of $n = |V| + |E|$ variables and $m = 3|E| + |V|$ constraints.

A.3. Criteria for selecting Gomory cuts

Recall that I_t is the number of candidate Gomory cuts available in round t, and i_t denotes the index of cut chosen by a given baseline. The baseline heuristics we use are the following:

- Random. One cut $i_t \sim \text{Uniform}\{1, 2...I_t\}$ is chosen uniformly at random from all the candidate cuts.
- Max Violation (MV). Let $x_B^*(t)$ be the basic feasible solution of the curent LP relaxation. MV selects the cut that corresponds to the most fractional component, i.e. $i_t = \arg\max\{|[x_B^*(t)]_i - \text{round}([x_B^*(t)]_i)|\}$.
- Max Normalized Violation (MNV). Recall that $\tilde{A}$ denotes the optimal tableau obtained by the simplex algorithm upon convergence. Let $\tilde{A}_i$ be the ith row of $\tilde{A}$. Then, MNV selects cut $i_t = \arg\max\{|[x_B^*(t)]_i - \text{round}([x_B^*(t)]_i)|/\|\tilde{A}_i\|\}$.
- Lexicographic (LE): Add the cutting plane with the least index, i.e. $i_t = \arg\min\{i, [x_B^*(t)]_i \text{ is fractional}\}$.

The first three rules are common in the IP literature, see e.g. (Wesselmann and Stuhl, 2012), while the fourth is the original rule used by Gomory to prove the convergence of his method (Gomory, 1960).

A.4. Hyper-parameters

Policy architecture. The policy network is implemented with Chainer (Tokui et al., 2015). The attention embedding F_θ is a 2-layer neural network with 64 units per layer and tanh activation. The LSTM network encodes variable sized inputs into hidden vector with dimension 10.

During a forward pass, a LSTM + Attention policy will take the instance, carry out embedding into a n-d vector and then apply attention. Such architecture allows for generalization to variable sized instances (different number of variables). We apply such architecture in the generalization part of the experiments.

On the other hand, a policy network can also consist of a single attention network. This policy can only process IP instances of a fixed size (fixed number of varibles) and cannot generalize to other sizes. We apply such architecture in the IGC part of the experiments.

ES optimization. Across all experiments, we apply Adam optimizer (Kingma and Ba, 2014) with learning rate $\alpha = 0.01$ to optimize the policy network. The perturbation standard deviation σ is selected from $\{0.002, 0.02, 0.2\}$. By default, we apply $N = 10$ perturbations to construct the policy gradient for each iteration, though we find that $N = 1$ could also work as well. For all problem types except planning, we find that $\sigma = 0.2$ generally works properly except for planning, where we apply $\sigma = 0.02$ and generate $N = 5$ trajectory per instance per iteration. Empirically, we observe that the training is stable for both policy architectures and the training performance converges in ≤ 500 weight updates.

Distributed setup. For training, we use a Linux machine with 60 virtual CPUs. To fully utilize the compute power of the machine, the trajectory collection is distributed across multiple workers, which run in parallel.

B. Branch-and-Cut Details

As mentioned in the introduction, Branch-and-Cut (B&C) is an algorithmic procedure used for solving IP problems. The choice of which variable to branch on, as well as which node of the branching tree to explore next, is the subject of much research. In our experiments, we implemented a B&C with very simple rules, as explained below. This is motivated by the fact that our goal is to evaluate the quality of the cutting planes added by the RL rather than obtaining a fast B&C method. Hence, sophisticated and computationally expensive branching rules could have overshadowed the impact of cutting planes. Instead, simple rules (applied both to the RL and to the other techniques) highlight the impact of cutting planes for this important downstream application.

We list next several critical elements of our implementation of B&C.

Branching rule. At each node, we branch on the most fractional variable of the corresponding LP optimal solution (0.5 being the most fractional).

Priority queue. We adopt a FIFO queue (Breath first search). FIFO queue allows the B&C procedure to improve the lower bound.

Termination condition. Let $z_0 = c^T x^*_{\text{LP}}(0)$ be the objective of the initial LP relaxation. As B&C proceeds, the procedure finds an increasing set of feasible integer solutions $\mathcal{X}_F$, and an upper bound on the optimal objective $z^* = c^T x^*_{\text{IP}}$ is $z_{\text{upper}} = \min_{x \in \mathcal{X}_F} c^T x$. Hence, z_{upper} monotonically decreases.

Along with B&C, cutting planes can iteratively improve the lower bound z_{lower} of the optimal objective z^*. Let z_i be the objective of the LP solution at node i and denote $\mathcal{N}$ as the set of unpruned nodes with unexpanded child nodes. The lower bound is computed as $z_{\text{lower}} = \min_{i \in \mathcal{N}} z_i$ and monotonically increases as the B&C procedure proceeds.

This produces a ratio statistic

$$r = \frac{z_{\text{upper}} - z_{\text{lower}}}{z_{\text{upper}} - z^*_{\text{LP}}} > 0$$

Note that since $z_{\text{lower}} \geq z^*_{\text{LP}}$, z_{lower} monotonically increases, and z_{upper} monotonically decreases, r monotonically decreases. The B&C terminates when r is below some threshold which we set to be 0.0001.

C. Test Time Considerations

Stopping criterion. Though at training time we guide the agent to generate aggressive cuts that tighten the LP relaxation as much as possible, the agent can exploit the defects in the simulation environment - numerical errors, and generate invalid cuts which cut off the optimal solution.

This is undesirable in practice. In certain cases at test time, when we execute the trained policy, we adopt a *stopping criterion* which automatically determines if the agent should stop adding cuts, in order to prevent from invalid cuts. In particular, at each iteration let $r_t = |c^T x_{\text{LP}^*(t)} - c^T x_{\text{LP}^*(t+1)}|$ be the objective gap achieved by adding the most recent cut. We maintain a cumulative ratio statistics such that

$$s_t = \frac{r_t}{\sum_{t' \leq t} r_t}.$$

We terminate the cutting plane procedure once the average s_t over a fixed window of size H is lower than certain threshold η. In practice, we set $H = 5, \eta = 0.001$ and find this work effectively for all problems, eliminating all the numerical errors observed in reported tasks. Intuitively, this approach dictates that we terminate the cutting plane procedure once the newly added cuts do not generate significant improvements for a period of H steps.

To analyze the effect of η and H, we note that when H is too small or η is too large, we have very conservative cutting plane procedure. On the other hand when H is large while η is small, the cutting plane procedure becomes more aggressive.

Greedy action. The policy network defines a stochastic policy, i.e. a categorical distribution over candidate cuts. At test time, we find taking the greedy action $i^* = \arg\max p_i$ to be more effective in certain cases, where p_i is the categorical distribution over candidate cuts. The justification for this practice is that: the ES optimization procedure can be interpreted as searching for a parameter θ such that the induced distribution over trajectories has large concentration on those high return trajectories. Given a trained model, to decode the *most likely* trajectory of horizon T generated by the policy, we need to run a full tree search of depth T, which is infeasible in practice. Taking the greedy action is equivalent to applying a greedy strategy in decoding the most likely trajectory.

This approach is highly related to beam search in sequence modeling (Sutskever et al., 2014) where the goal is to decode the prediction that the model assigns the most likelihood to. The greedy action selection above corresponds to a beam search with 1-step lookahead.

D. Details on the Interpretation of Cutts

One interesting aspect of studying the RL approach to generating cuts, is to investigate if we can interpret cuts generated by RL. For a particular class of IP problems, certain cuts might be considered as generally 'better' than other cuts. For example, these cuts might be more effective in terms of closing the objective gap, according to domain knowledge studied in prior literature. Ideally, we would like to find out what RL has learned, whether it has learned to select these more 'effective' cuts with features identified by prior works. Here, we focus on *Knapsack problems*.

Problem instances. Consider the knapsack problems

$$\begin{cases} \max \sum_i^n c_i x_i \\ \sum_{i=1}^n a_i x_i \leq \beta := \sum_{i=1}^n a_i/2 \\ x_i \in \{0,1\}, \end{cases} \tag{12}$$

where a_i are generated independently and uniformly in $[1, 30]$ as integers, and the c_i are generated independently and uniformly in $[1, 10]$. We consider $n = 10$ in our experiments. Knapsack problems are fundamental in IP, see e.g. (Kellerer et al., 2003). The intuition of the problem is that we attempt to pack as many items as possible into the knapsack, as to maximize the profit of the selected items. Polytopes as (12) are also used to prove strong (i.e., quadratic) lower bounds on the Chvátal-Gomory rank of polytopes with $0/1$ vertices (Rothvoß and Sanità, 2017).

Evaluation scores. For knapsack problems, one effective class of cuts is given by *cover inequalities*, and their strengthening through *lifting* (Conforti et al., 2014; Kellerer et al., 2003). The cover inequality associated to a set $S \subseteq \{1, \ldots, n\}$ with $\sum_{i \in S} a_i > \beta$ and $|S| = k$ is given by

$$\sum_{i \in S} x_i \leq k - 1.$$

Note that cover inequalities are valid for (12). The inequality can be strengthened (while maintaining validity) by replacing the 0 coefficients of variables x_i for $i \in \{1, \ldots, n\} \setminus S$ with appropriate positive coefficients, leading to the *lifted cover inequality* below:

$$\sum_{i \in S} x_i + \sum_{\notin S} \alpha_i x_i \leq k - 1. \tag{13}$$

with all $\alpha_i \geq 0$. There are in general exponentially many ways to generate lifted cover inequalities from a single cover inequality. In practice, further strengthenings are possible, for instance, by perturbing the right-hand side or the coefficients of x_i for $i \in S$. We provide three criteria for identifying (strengthening of) lifted cover inequalities, each capturing certain features of the inequalities (below, RHS denotes the right-hand side of a given inequality).

1. There exists an integer p such that (1) the RHS is an integer multiple of p and (2) p times (number of variable with coefficient exactly p) $>$ RHS.

Criterion 1 is satisfied by all lifted cover inequalities as in (13). The scaling by p is due to the fact that an inequality may be scaled by a positive factor, without changing the set of points satisfying it.

2. There exists an integer p such that (1) holds and (2') p times (number of variables with coefficients between p and $p + 2$) $>$ RHS.
3. There exists an integer p such that (1) holds and (2") p times (number of variables with coefficients at least p) $>$ RHS.

A lifted cover inequality can often by strengthened by increasing the coefficients of variables in S, after the lifting has been performed. We capture this by criteria 2 and 3 above, where 2 is a stricter criterion, as we only allow those variables to have their coefficients increased by a small amount.

For each cut c_j generated by the baseline (e.g. RL), we evaluate if this cut satisfies the aforementioned conditions. For one particular condition, if satisfied, the cut is given a score $s(c_j) = 1$ or else $s(c_j) = 0$. On any particular instance, the overall score is computed as an average across the m cuts that are generated to solve the problem with the cutting plane method:

$$\frac{1}{m} \sum_{j=1}^{m} s(c_j) \in [0, 1].$$

Evaluation setup. We train a RL agent on 100 knapsack instances and evaluate the scores on another independently generated set of 20 instances. Please see the main text for the evaluation results.

E. Additional results on Large-scale Instances

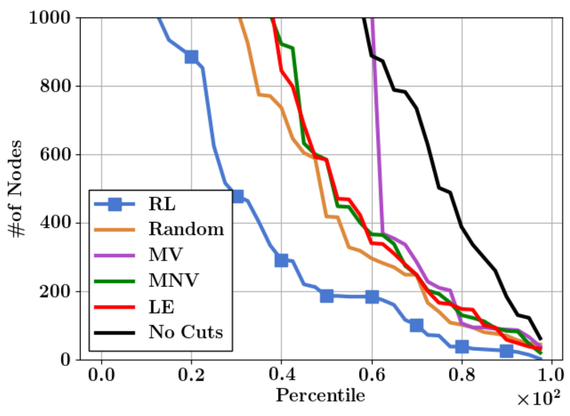
We provide additional results on large-scale instances in Figure 7, in the context of B&C. Experimental setups and details are similar to those of the main text: we set the threshold limit to be 1000 nodes for all problem classes. The results show the percentile plots of the number of nodes required to achieve a certain level of IGC during the B&C with the use of cutting plane heuristics, where the percentile is calculated across instances. Baseline results for each baseline are shown via curves in different colors. When certain curves do not show up in the plot, this implies that these heuristics do not achieve the specified level of IGC within the node budgets. The IGC level is set to be 95% as in the main text, except for the random packing problem where it is set to be 25%.

The IGC of the random packing is set at a relatively low level because random packing problems are significantly more difficult to solve when instances are large-scaled. This is consistent with the observations in the main text.

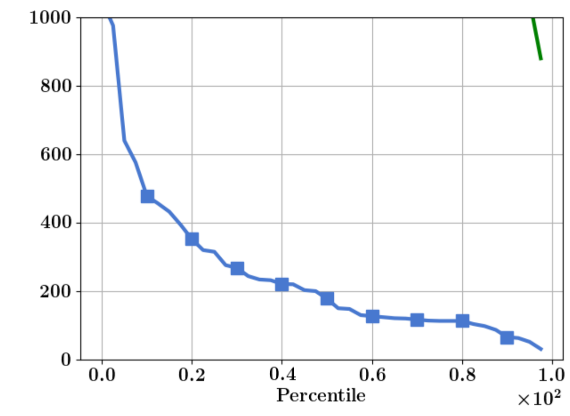
Overall, we find that the performance of RL agent significantly exceeds that of the other baseline heuristics. For example, on the planning problem, other heuristics barely achieve the IGC within the node budgets. There are also cases where RL does similarly to certain heuristics, such as to MNV on the Max Cut problems.

F. Comparison of Distributed Agent Interface

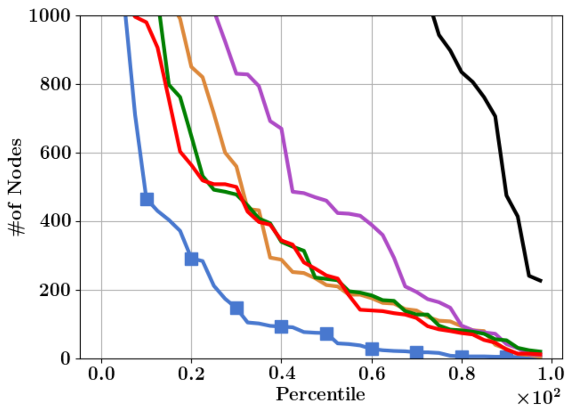
To scale RL training to powerful computational architecture, it is imperative that the agent becomes distributed. Indeed, recent years have witnessed an increasing attention on the design and implementation of distributed algorithms (Mnih et al.,



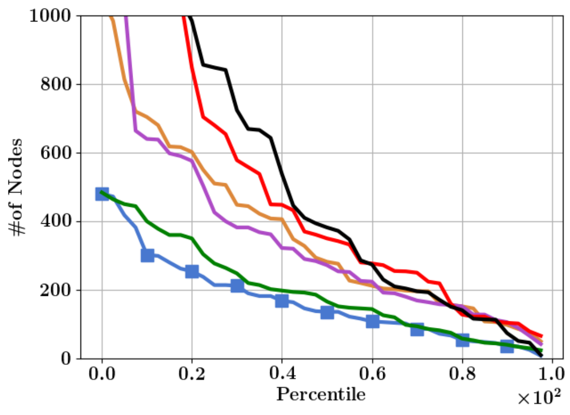
(a) Packing (25% IGC, 1000 nodes)



(b) Planning (95% IGC, 1000 nodes)



(c) Binary Packing (95% IGC, 1000 nodes)



(d) Max Cut (95% IGC, 1000 nodes)

Figure 7: Percentile plots of number of B&C nodes expanded for large-scale instances. The same setup as Figure 5 but for even larger instances.

2016; Salimans et al., 2017; Espeholt et al., 2018; Kapturowski et al., 2018).

General distributed algorithms adopt a learner-actor architecture, i.e. one central learner and multiple distributed actors. Actors collect data and send partial trajectories to the learner. The learner takes data from all actors and generates updates to the central parameter. The general interface requires a function $\pi_\theta(a|s)$ parameterized by θ, which takes a state s and outputs an action a (or a distribution over actions). In a gradient-based algorithm (e.g. (Espeholt et al., 2018)), the actor executes such an interface with the forward mode and generates trajectory tuple (s, a); the learner executes the interface with the backward mode to compute gradients and update θ. Below we list several practical considerations why ES is a potentially better distributed alternative to such gradient-based distributed algorithms in this specific context, where state/action spaces are irregular.

- **Communication.** The data communication between learner-actor is more complex for general gradient-based algorithms. Indeed, actors need to send partial trajectories $\{(s_i, a_i, r_i)\}_{i=1}^{\tau}$ to the learner, which requires careful adaptations to cases where the state/action space are irregular. On the other hand, ES only require sending returns over trajectories $\sum_{i=0}^{T} r_i$, which greatly simplifies the interface from an engineering perspective.
- **Updates.** Gradient-based updates require both forward/backward mode of the agent interface. Further, the backward mode function needs to be updated such that batched processing is efficient to allow for fast updates. For irregular state/action space, this requires heavier engineering because of e.g. arrays of variable sizes are not straightforward to be

Table 5: IGC in B&C with large-scale instances. We adopt the same setup as Table 2

Tasks	**Packing**	**Planning**	**Binary**	**Max Cut**
Size	60×60	122×168	66×132	54×134
No cut	0.26 ± 0.09	0.25 ± 0.04	0.74 ± 0.24	0.95 ± 0.09
Random	0.31 ± 0.08	0.65 ± 0.10	0.94 ± 0.10	0.99 ± 0.04
MV	0.23 ± 0.08	0.27 ± 0.07	0.92 ± 0.12	0.98 ± 0.06
MNV	0.27 ± 0.08	0.33 ± 0.15	0.93 ± 0.10	1.0 ± 0.0
LE	0.28 ± 0.08	0.25 ± 0.04	0.95 ± 0.08	0.95 ± 0.09
RL	$\mathbf{0.36 \pm 0.10}$	$\mathbf{0.99 \pm 0.02}$	$\mathbf{0.96 \pm 0.08}$	1.0 ± 0.0

batched. On the other hand, ES only requires forward mode computations required by CPU actors.

G. Considerations on CPU Runtime

In practice, instead of the number of cuts, a more meaningful budget constraint on solvers is the CPU runtime, i.e. practitioners typically set a runtime constraint on the solver and expect the solver to return the best possible solution within this constraint. Below, we report runtime results for training/test time. We will show that even under runtime constraints, the RL policy achieves significant performance gains.

Training time. During training time, it is not straightforward to explicitly maintain a constraint on the runtime, because it is very sensitive to hardware conditions (e.g. number of available processors). Indeed, prior works (Khalil et al., 2016; Dai et al., 2017) do not apply runtime constraint during training time, though runtime constraint is an important measure at test time.

The absolute training time depends on specific hardware architecture. In our experiments we train with a single server with 64 virtual CPUs. Recall that each update consists in collecting trajectories across training instances and generating one single gradient update. We observe that typically the convergence takes place in ≤ 500 weight updates (iterations).

Test time. To account for the practical effect of runtime, we need to account for the following trade-off: though RL based policy produces higher-quality cutting planes in general, running the policy at test time could be costly. To characterize the trade-offs, we address the following question: **(1)** When adding a fixed number of cuts, does RL lead to higher runtime? **(2)** When solving a particular problem, does RL lead to performance gains in terms of runtime?

To address **(1)**, we reuse the experiments in Experiment #2, i.e. adding a fixed number of cuts $T = 50$ on middle sized problems. The runtime results are presented in Table 6, where we show that RL cutting plane selection does not increase the runtime significantly compared to other 'fast' heuristics. Indeed, RL increases the average runtime in some cases while decreases in others. Intuitively, we expect the runtime gains to come from the fact that RL requires a smaller number of cuts - leading to fewer iterations of the algorithm. However, this is rare in Experiment #2, where for most instances optimal solution is not reached in maximum number of cuts, so all heuristics and RL add same number of cuts ($T = 50$). We expect such advantages to become more significant with the increase of the size of the problem, as the computational gain of adding good cuts becomes more relevant. We confirm such intuitions from the following.

To address **(2)**, we reuse the results from Experiment #4, where we solve more difficult instances with B&C, we report the runtime results in Table 7. In these cases, the benefits of high-quality cuts are magnified by a decreased number of iterations (i.e. expanded nodes) - indeed, for RL policy, the advantages resulting from decreased iterations significantly overweight the potentially slight drawbacks of per-iteration runtime. In Table 7, we see that RL generally requires much smaller runtime than other heuristics, mainly due to a much smaller number of B&C iterations. Note that these results are consistent with Figure 5. Again, for large-scale problems, this is an important advantage in terms of usage of memory and overall performance of the system.

Table 6: CPU runtime for adding cutting planes (units are seconds). Here we present the results from Experiment #2 from the main text, where we fix the number of added cuts $T = 50$. Note that though RL might increase runtime in certain cases, it achieves much larger IGC within the cut budgets. Note that these results are consistent with Table 2.

Tasks	**Packing**	**Planning**	**Binary**	**Max Cut**
Size	30×30	61×84	33×66	27×67
RANDOM	0.06 ± 0.01	0.09 ± 0.01	0.088 ± 0.003	0.08 ± 0.01
MV	0.9 ± 0.01	0.100 ± 0.004	0.10 ± 0.01	0.11 ± 0.01
MNV	0.10 ± 0.02	0.100 ± 0.004	0.12 ± 0.02	$0.12 \pm 0.01\%$
RL	0.10 ± 0.02	0.14 ± 0.03	0.07 ± 0.04	0.08 ± 0.02

Table 7: CPU runtime in B&C with large-scale instances. The measures are normalized with respect to RL so that the RL runtime is always measured as 100%. Here, we measure the runtime as the time it takes to reach a certain level of IGC. We only measure the runtime on test instances where the IGC level is reached within the node budgets. When the IGC is not reached for most test instances (as in the case of the planning problem for most baselines), the runtime measure is 'N/A'. Note that the results here are consistent with Table 3 and Figure 7.

Tasks	**Packing**	**Planning**	**Binary**	**Max Cut**
Size	60×60	122×168	66×132	54×134
RANDOM	146%	N/A	190%	250%
MV	256%	N/A	340%	210%
MNV	238%	N/A	370%	$\mathbf{95}\%$
LE	120%	N/A	370%	120%
RL	$\mathbf{100}\%$	$\mathbf{100}\%$	$\mathbf{100}\%$	100%