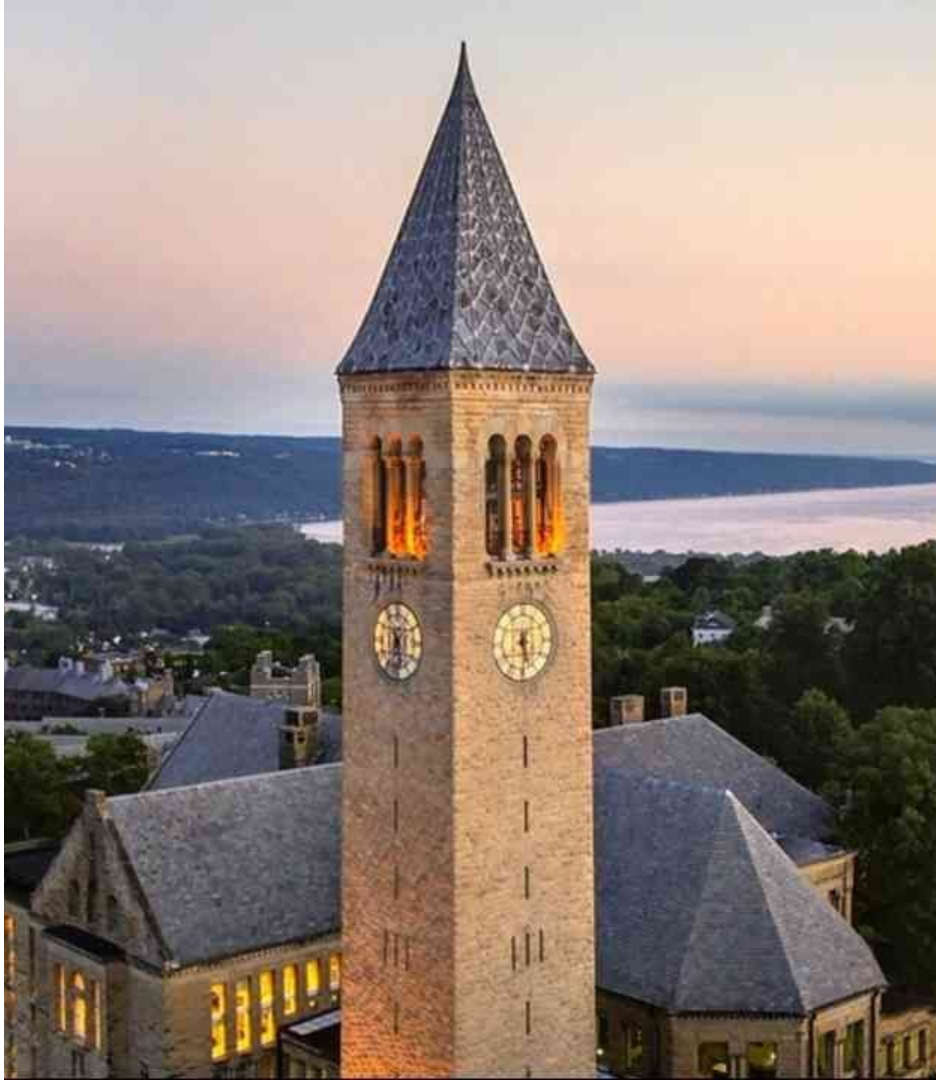
Reinforcement Learning for Integer Programming

ORIE 6590

May 17th, 2021

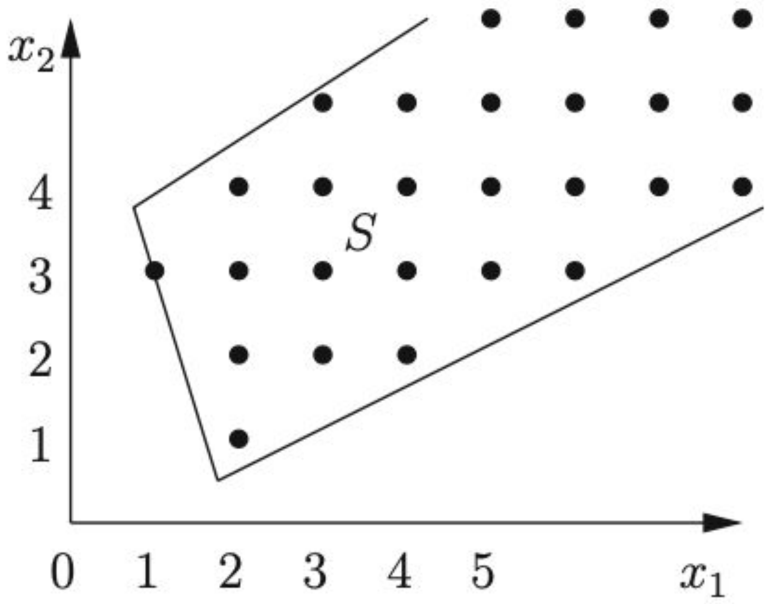
Connor Lawless & Logan Grout
Cornell University



Learning to Branch

Integer Programming

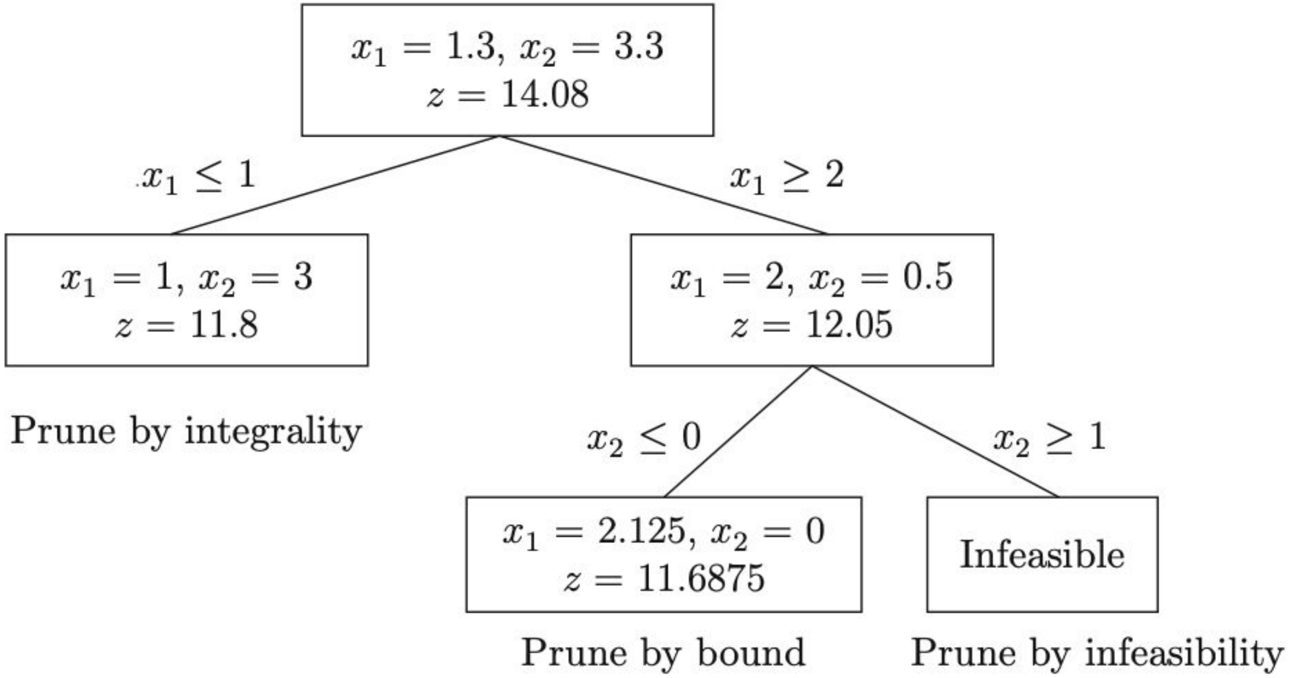
$$\begin{array}{rrcll} \text{max} & cx & & & \\ \text{subject to} & Ax & \leq & b & \\ & x & \geq & 0 & \text{integral} \end{array}$$



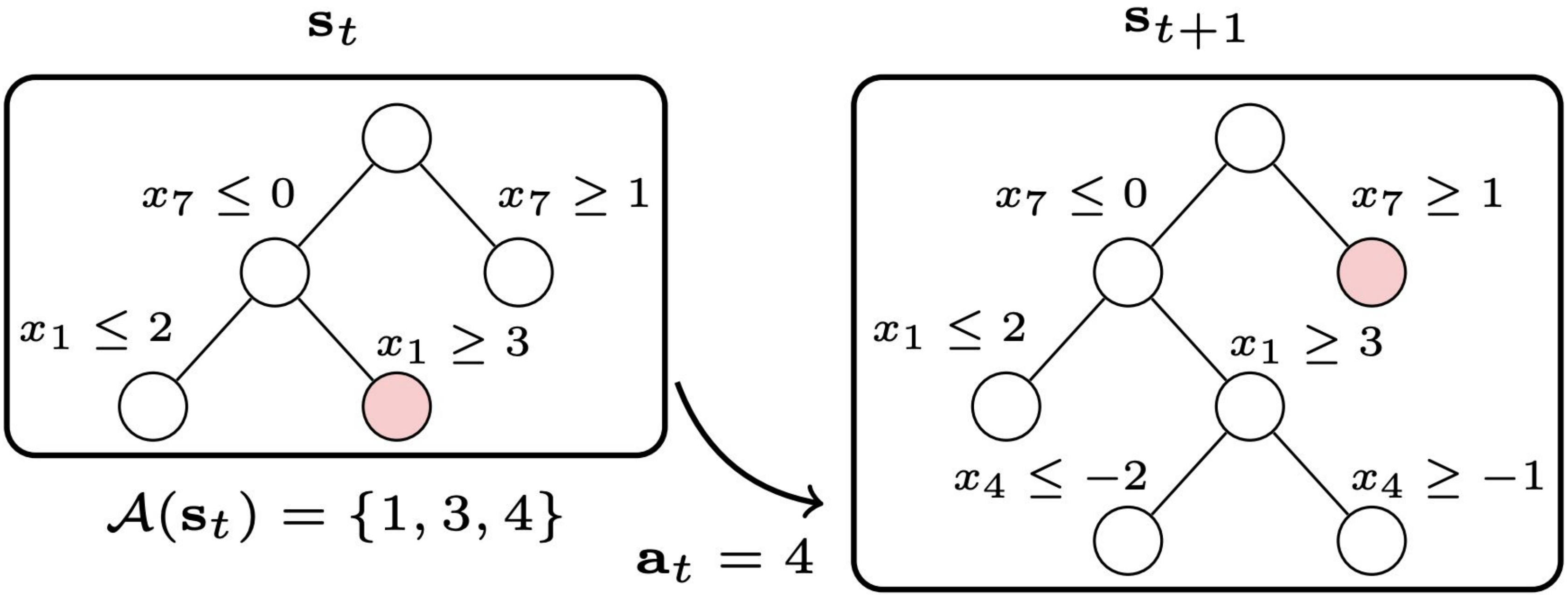
Most modern solvers are built around the idea that we can solve LPs fast (both theoretically and practically), so we solve IPs by **solving a sequence of LPs**

Branch & Bound

Key Idea: Recursively partition feasible region until we get integral solutions from linear relaxation

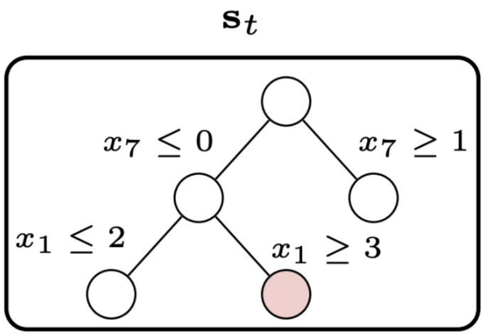


Branching Decisions



MDP Formulation

State
Current B&B Tree and node of interest



Actions
Which fractional variable to branch on

$$\mathcal{A}(\mathbf{s}_t) = \{1, 3, 4\}$$

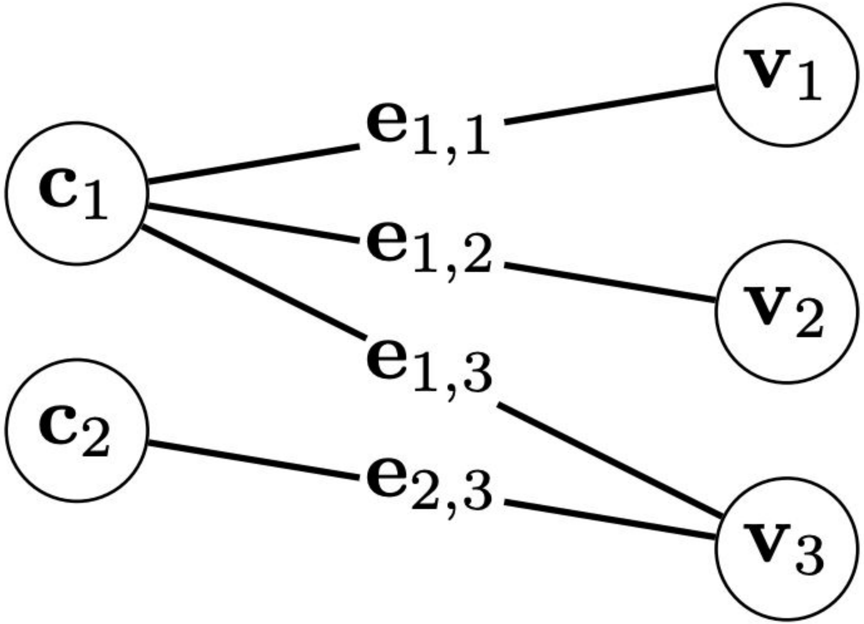
Reward
Number of nodes explored during solve process

-1 per time step

Transitions are deterministic (i.e. add nodes, follow branch + cut algo.)

State Representation

They represent the current LP node of interest as a bipartite graph with side information.

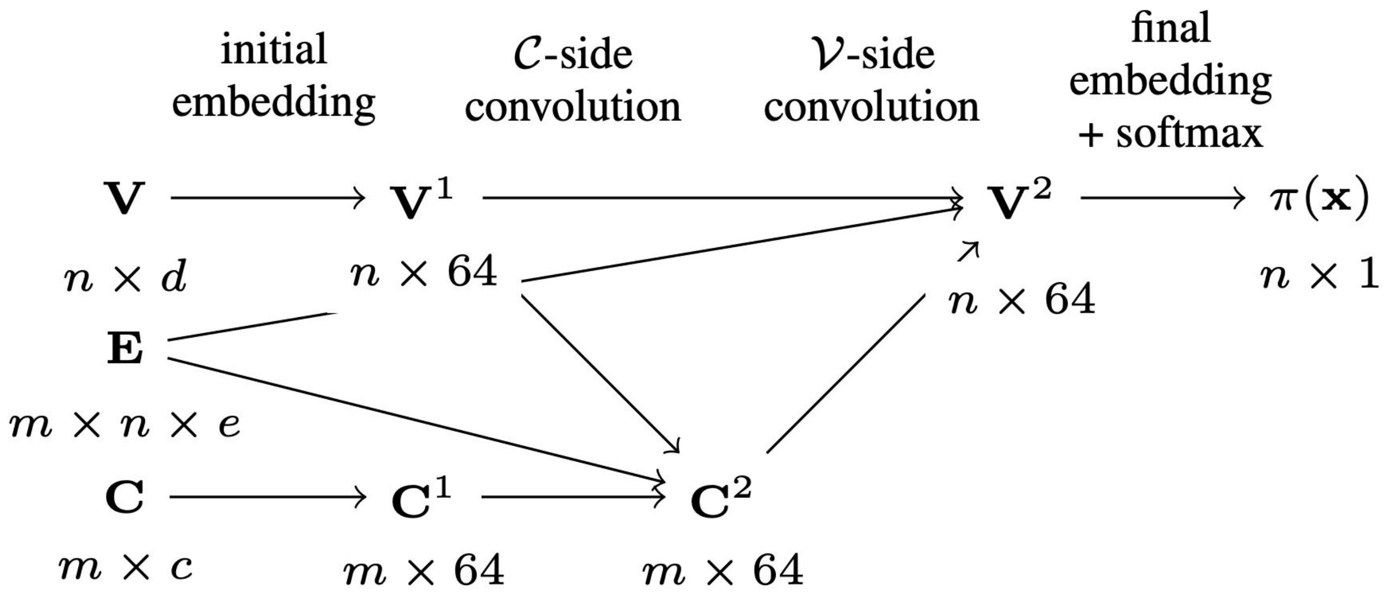


State Representation

Tensor	Feature	Description
C	obj_cos_sim	Cosine similarity with objective.
	bias	Bias value, normalized with constraint coefficients.
	is_tight	Tightness indicator in LP solution.
	dualsol_val	Dual solution value, normalized.
	age	LP age, normalized with total number of LPs.
E	coef	Constraint coefficient, normalized per constraint.
V	type	Type (binary, integer, impl. integer, continuous) as a one-hot encoding.
	coef	Objective coefficient, normalized.
	has_lb	Lower bound indicator.
	has_ub	Upper bound indicator.
	sol_is_at_lb	Solution value equals lower bound.
	sol_is_at_ub	Solution value equals upper bound.
	sol_frac	Solution value fractionality.
	basis_status	Simplex basis status (lower, basic, upper, zero) as a one-hot encoding.
	reduced_cost	Reduced cost, normalized.
	age	LP age, normalized.
	sol_val	Solution value.
	inc_val	Value in incumbent.
	avg_inc_val	Average value in incumbents.

Our Approach

GCNN for Parametric Policy



$$\mathbf{c}_i \leftarrow \mathbf{f}_{\mathcal{C}}\left(\mathbf{c}_i, \sum_{j}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{C}}\left(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}\right)\right), \qquad \mathbf{v}_j \leftarrow \mathbf{f}_{\mathcal{V}}\left(\mathbf{v}_j, \sum_{i}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{V}}\left(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}\right)\right)$$

Warm Start: Imitation Learning

Generate expert (full strong branching) training samples $\mathcal{D} = \{(\mathbf{s}_i, \mathbf{a}_i^\star)\}_{i=1}^N$.

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{s},\mathbf{a}^*)\in\mathcal{D}} \log \pi_\theta(\mathbf{a}^* \mid \mathbf{s})$$

Gasse et al. (2019): Exact Combinatorial Optimization with Graph Convolutional Neural Networks. *NeurIPs*.

Training: Evolution Strategies

$$\epsilon_i \sim \mathcal{N}(0, \mathbb{I})$$

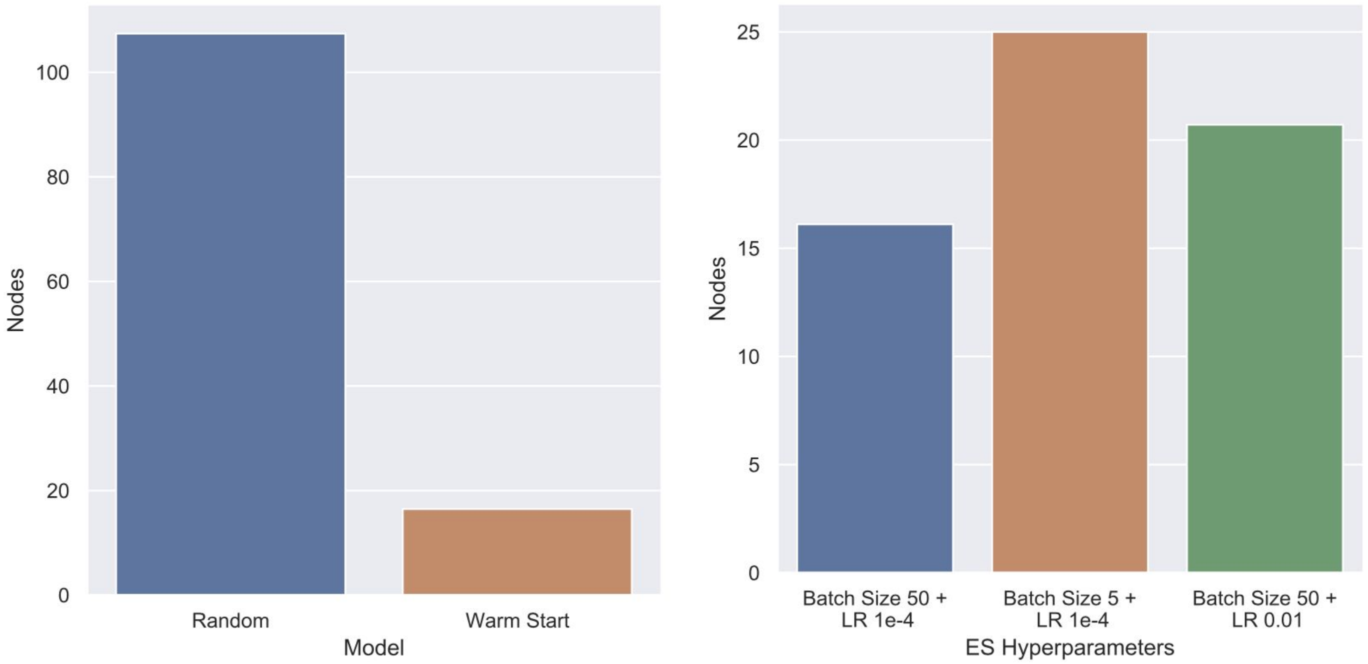
$$\boxed{\hat{g}_\theta = \frac{1}{N} \sum_{i=1}^{N} J(\pi_{\theta'_i}) \frac{\epsilon_i}{\sigma}}$$

$$\sum_{t=0}^{T-1} r_t \gamma^t$$

$$\theta'_i = \theta + \sigma \epsilon_i$$

Implementation Details

Warm starting, a small learning rate, and a larger batch size for ES were the winning combination.



Empirical Performance

Our **RL approach (GCNN + ES) is able to *modestly* outperform the GCNN** architecture alone, with the caveat that the GCNN was trained on our *limited* computing power.

Table 1: Policy evaluation on test set cover instances

Algorithm	Time	Wins	Nodes
Full Strong Branching	6.16 (11.1%)	0/100	**11.44 (11.1%)**
Pseudocost Branching	2.66 (15.2%)	23/100	20.27 (15.2%)
GCNN	2.07 (14.8%)	29/100	16.48 (16.6%)
GCNN + ES	**2.04 (13.7%)**	**48/100**	16.11 (14.7%)