Searching Large Neighborhoods for Integer Linear Programs with Contrastive Learning

Taoan Huang, Aaron Ferber, Yuandong Tian, Bistra Dilkina, Benoit Steiner

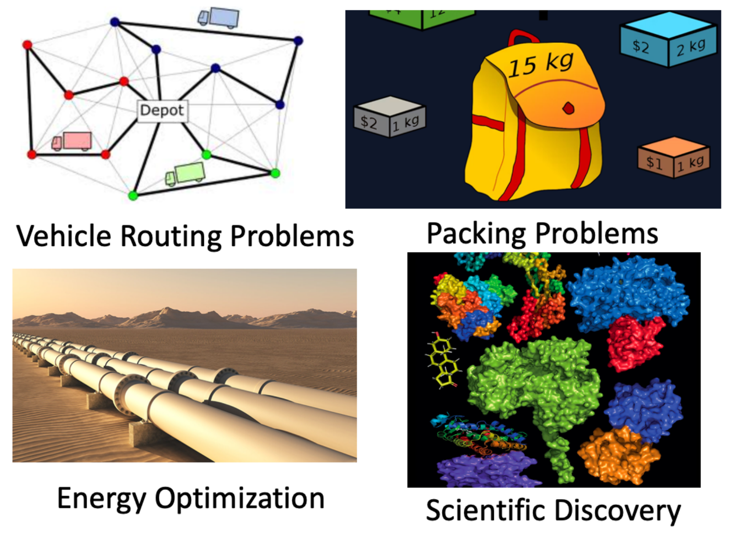
USC University of Southern California   Meta AI

1 Introduction

Integer Linear Programs (ILP)

$$\begin{aligned} \min \quad & \mathbf{c}^{\mathsf{T}}\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \mathbf{x} \in \{0,1\}^n \end{aligned}$$

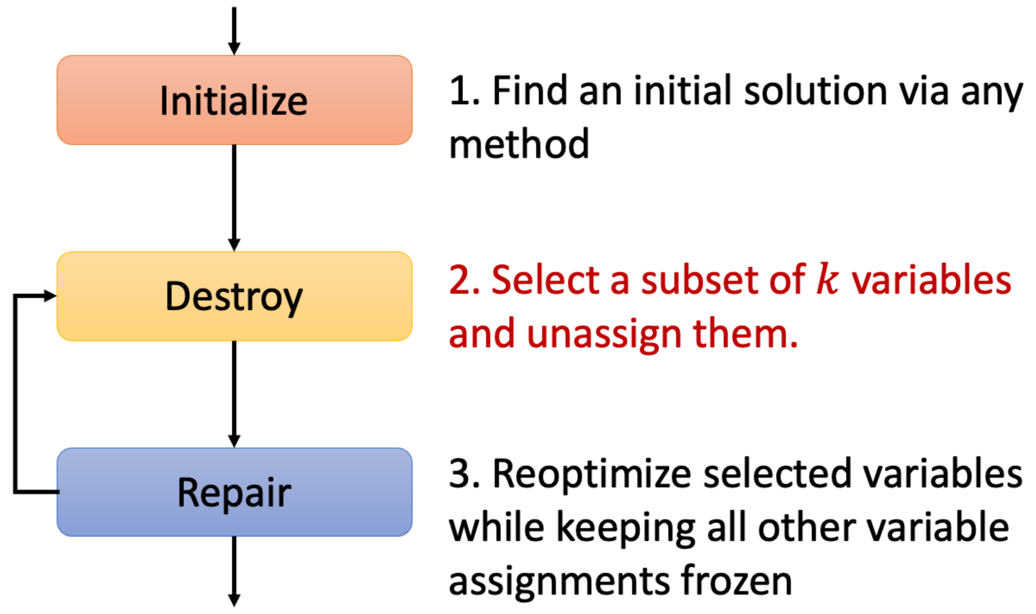
Real-World Applications



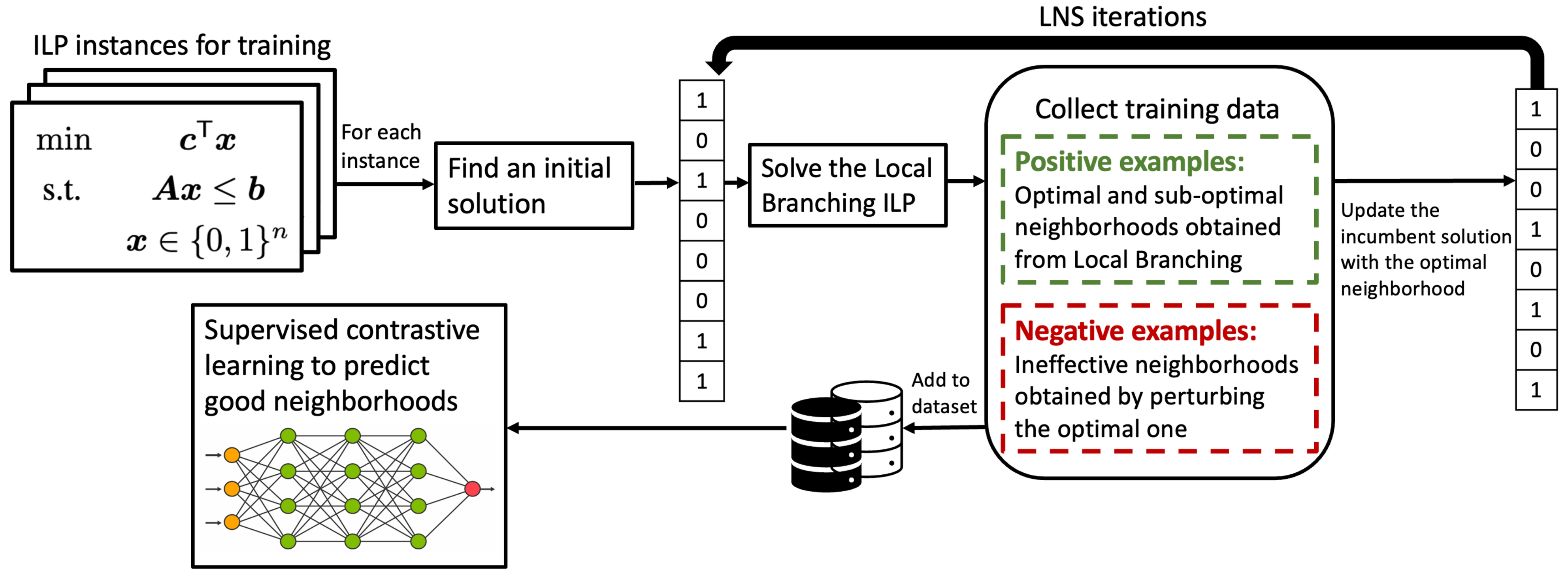
Large Neighborhood Search (LNS)

A metaheuristic that could solve large-scale combinatorial optimizations efficiently.

The **Destroy** step requires heuristic designs where machine learning can offer helpful guidance.



3 Contrastive Learning for LNS (CL-LNS)



Training and data collection overview: For each ILP instance for training, we run several LNS iterations with LB. In each iteration, we collect both positive and negative neighborhood samples and add them to the training dataset, which is used in downstream supervised contrastive learning for neighborhood selections.

Training

Network Architecture: Bipartite Graph $\boldsymbol{x}$ → Embedding layers → Graph attention network (GAT) with two rounds of message passing → MLP → Sigmoid → $[0,1]$ score per variable $\pi(\boldsymbol{x})$.

Loss Function: InfoNCE Loss (θ: network parameters; $\mathcal{D}$: training dataset; τ: temperature parameter):

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},\mathcal{S}_{pos},\mathcal{S}_{neg})\in\mathcal{D}} \frac{-1}{|\mathcal{S}_{pos}|} \sum_{\boldsymbol{a}\in\mathcal{S}_{pos}} \log \frac{\exp(\boldsymbol{a}^{\mathsf{T}}\pi(\boldsymbol{x})/\tau)}{\sum_{\boldsymbol{a}'\in\mathcal{S}_{neg}\cup\{\boldsymbol{a}\}} \exp(\boldsymbol{a}'^{\mathsf{T}}\pi(\boldsymbol{x})/\tau)}$$

Testing

- Use the learned policy to predict scores for variables and **greedily** choose the top k.
- Adaptively adjust neighborhood size k.

2 Background

Local Branching (LB)

A heuristic that models the problem of finding the optimal subset to destroy.

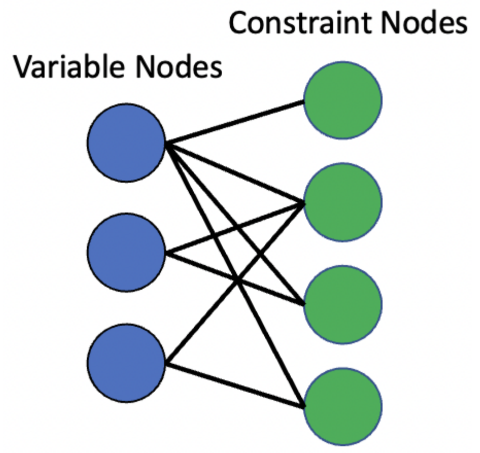
Given the current best solution x' to the ILP, it adds an extra constraint to **find** the optimal k variables to reoptimize that lead to the most improvement:

$$\begin{aligned} \min \quad & \mathbf{c}^{\mathsf{T}}\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \mathbf{x} \in \{0,1\}^n \\ & \textstyle\sum_{i:x'_i=0} x_i + \sum_{i:x'_i=1}(1-x_i) \le k. \end{aligned}$$

LB is slow to solve but could be useful for data collection for learning.

Bipartite Graph Representation for ILPs

- Two sets of nodes: variables and constraints.
- An edge between a variable and a constraint if the variable appears in the constraint.



3 Related Work

ML-Guided LNS for ILPs

- Decomposition-based LNS [3]: First ML-guided LNS work that uses both reinforcement learning (RL) and imitation learning (IL) to learn a decomposition-based LNS.
- RL-LNS [5]: the state-of-the-art RL approach.
- IL-LNS [4]: the state-of-the-art IL approach that imitates LB.

LB Variants in LNS for ILPs

- ML-tuned LB [2]: Use ML to tune the time limit and neighborhood size for LB.
- LB-RELAX [1]: Use the LP relaxation of LB to select variables to destroy.

References

[1] Taoan Huang et al. Local branching relaxation heuristics for integer linear programs. In *CPAIOR*, 2023.

[2] Defeng Liu, Matteo Fischetti, and Andrea Lodi. Learning to search in local branching. In *AAAI*, 2022.

[3] Jialin Song et al. A general large neighborhood search framework for solving integer linear programs. *NeurIPS*, 2020.

[4] Nicolas Sonnerat et al. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

[5] Yaoxin Wu et al. Learning large neighborhood search policy for integer programming. *NeurIPS*, 2021.

4 Empirical Evaluation

Experimental Setup

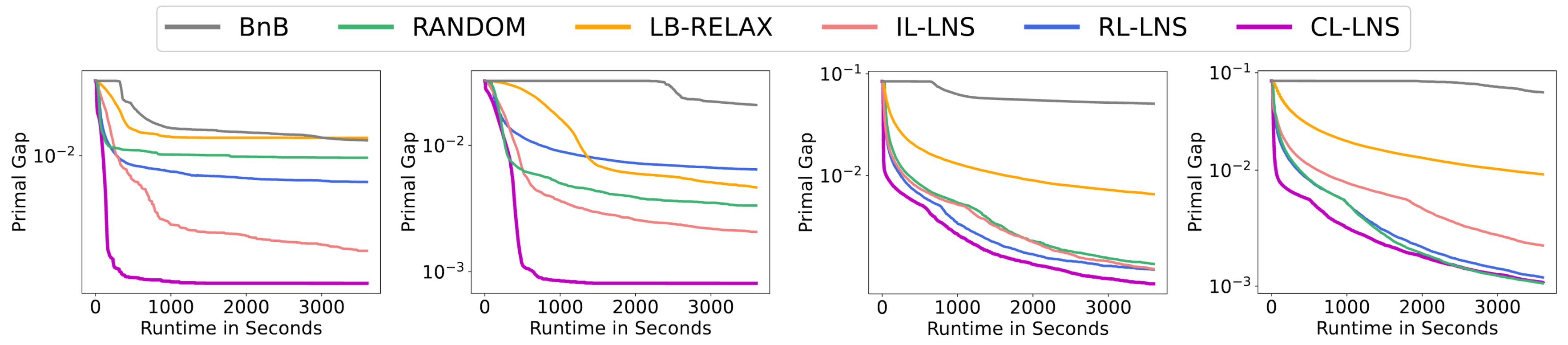
Baselines:

BnB: using SCIP as the solver; **RANDOM**: LNS with a randomized destroy heuristic; **LB-RELAX** [1]; **IL-LNS** [4]; **RL-LNS** [5].

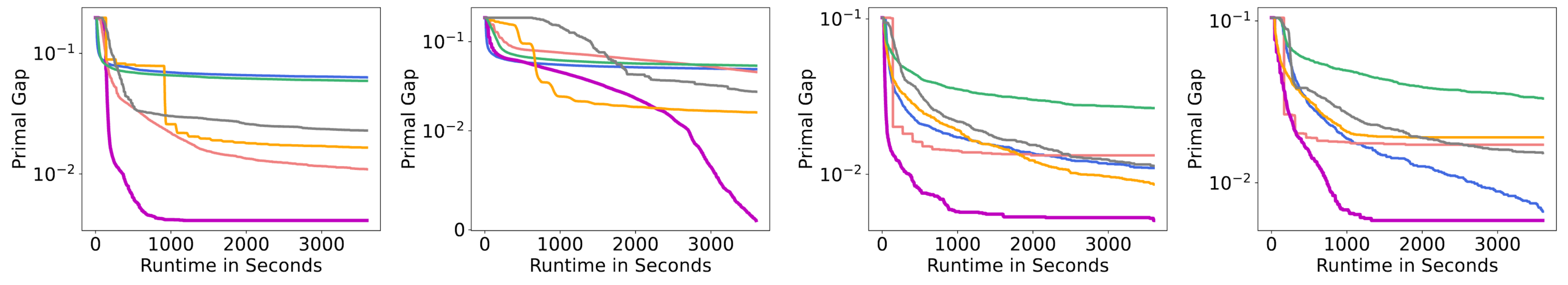
Benchmark:

	Small Instances (for **training** and **testing**)				Large Instances (for **testing only**)			
Name	MVC-S	MIS-S	CA-S	SC-S	MVC-L	MIS-L	CA-L	SC-L
#Variables	1,000	6,000	4,000	4,000	2,000	12,000	8,000	8,000
#Constraints	65,100	23,977	2,675	5,000	135,100	48,027	5,353	5,000

Results

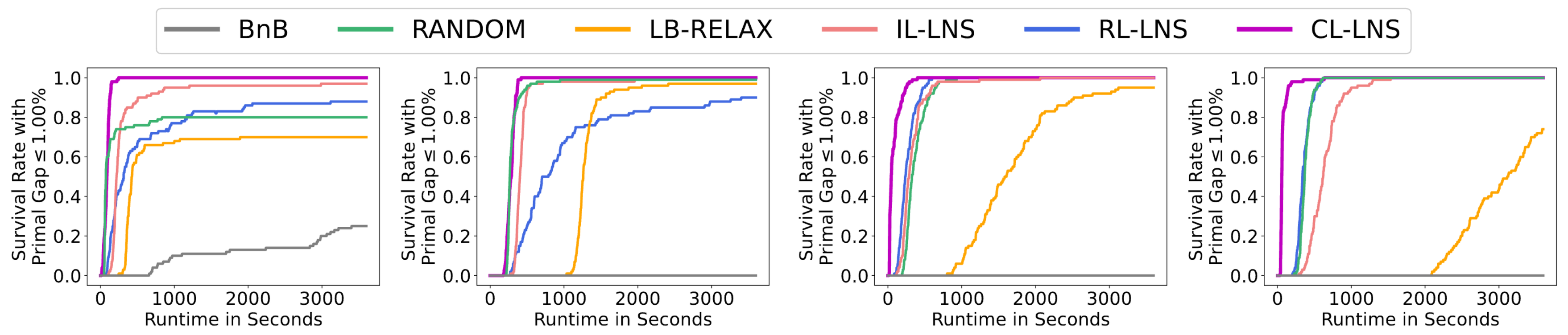


(a) MVC-S (left) and MVC-L (right). (b) MIS-S (left) and MIS-L (right).

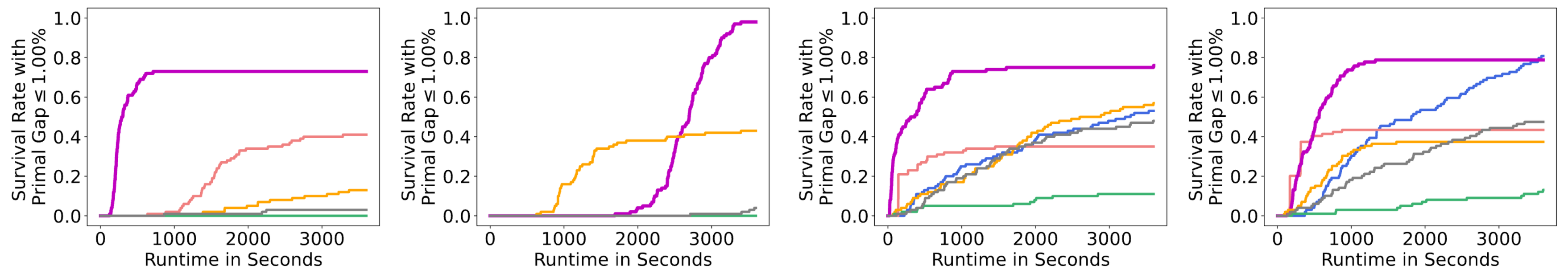


(c) CA-S (left) and CA-L (right). (d) SC-S (left) and SC-L (right).

The **primal gap** as a function of runtime, averaged over 100 instances. The **primal gap** is the normalized difference between the objective value and a best known objective.



(a) MVC-S (left) and MVC-L (right). (b) MIS-S (left) and MIS-L (right).



(c) CA-S (left) and CA-L (right). (d) SC-S (left) and SC-L (right).

The **survival rate** (the fraction of instances with primal gaps below 1.00%) as a function of runtime over 100 instances.