MACHINE LEARNING FOR OPTIMIZATION

January 24, 2025

Today (session 4/5)
Overview of ML techniques for MILP solves

Monday (Lab): End lab 2 + (MI)LPs and GNNs

Exam: Feb. 7, 10 am - 12 pm
- Open book (written/printed documents allowed)
- Will consist of
  - ↳ Questions about what we saw in class
  - ↳ (Guided) questions about a short research paper related to the course

MACHINE LEARNING FOR MIXED-INTEGER LINEAR PROGRAMS

Goal:
- → Overview of the learning tasks and the techniques used to perform these tasks
- → Following review papers: details are scattered among individual papers/codes
- → Focus: Use of graph neural networks to represent LP-related learning tasks

(1) MILPs

a) Problem and branch-and-bound

$$(MILP) \quad \underset{x \in \mathbb{R}^n}{\text{minimize}} \; c^T x \quad \text{s.t.} \quad Ax \geq b$$
$$x_j \in \mathbb{Z}_{\geq 0} \quad \forall j \in A$$
$$x_j \in \{0,1\} \quad \forall j \in B$$
$$x_j \geq 0 \quad \forall j \in \mathcal{E}$$

$A \in \mathbb{Q}^{m \times n}$
$b \in \mathbb{Q}^m$
$c \in \mathbb{Q}^n$

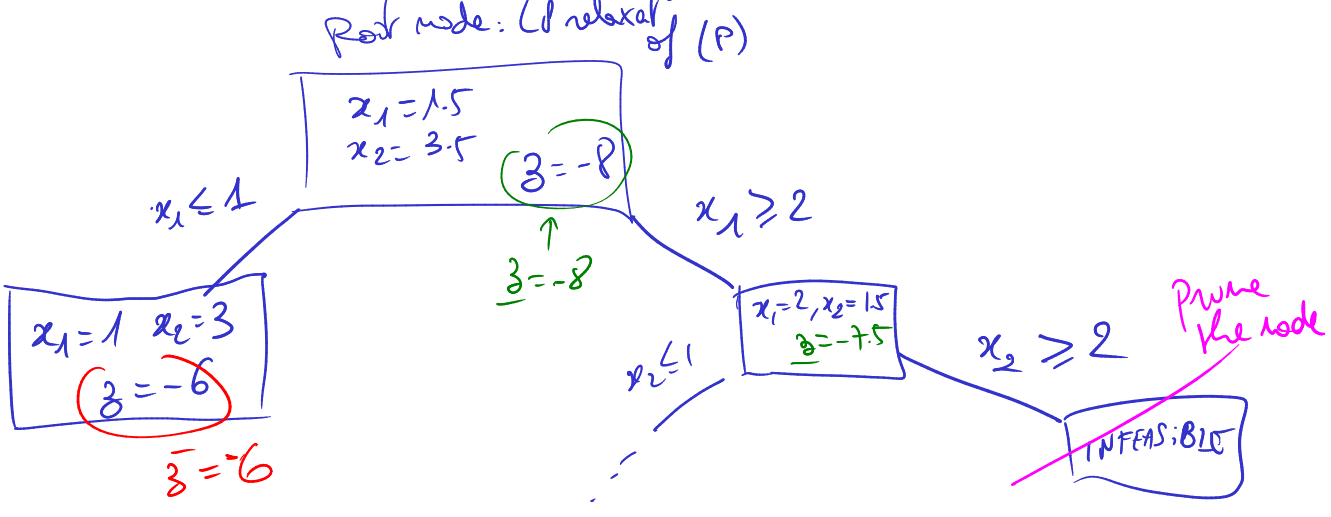
$A \cup B = I$: set of discrete (integer + binary) variables
$A, B, \mathcal{E}$ partition of $\{1, \dots, n\}$

LP relaxation: Replace $x_j \in \mathbb{Z}_{\geq 0}$ $j \in A$ $\Rightarrow$ $x_j \geq 0$ $j \in A$
$x_j \in \{0,1\}$ $j \in B$ $\Rightarrow$ $x_j \in [0,1]$ $j \in B$

⇒ Most state-of-the-art solvers for MILP are based on (variants) of the branch-and-bound technique

- Build a search tree of LP relaxations based on adding bounds on variables one at a time
- At every iteration, have
  - → an upper bound on the optimal value (from an integer feasible solution) : $\bar{z}$
  - → a lower bound for the best optimal value obtained by the LP relaxations $\underline{z}$
- Stops
  - when $\underline{z} = \bar{z}$
  - when all nodes have been processed
  - when timeout

Ex) (P) minimize $-(3x_1 + x_2)$ s.t. $-x_1 + x_2 \leq 2$
$x \in \mathbb{R}^2$ $8x_1 + x_2 \leq 19$
$x_1, x_2 \geq 0$
$x_1, x_2 \in \mathbb{Z}$



b) MILP solvers

In addition to the Branch-and-Bound algorithm, many heuristics and many degrees of freedom in designing the method

* Preprocessing

Goal: Reduce the size of problem (both in terms of constraints and variables)

→ Fundamental in practice but heuristic/empirical nature

* Branching and node selection

At every node, solve the LP relaxation. and when one integer variable is given a fractional value ($x_j^{LP} \in \mathbb{Q} \setminus \mathbb{Z}$) can branch on $x_j \leq \lfloor x_j^{LP} \rfloor$ or $x_j \geq \lceil x_j^{LP} \rceil$

⇒ Numerical studies have shown that branching strategies, but often it is possible to branch on several variables, no generic rule.

* Cutting planes

Idea: Strengthen the LP relaxation by adding new inequality constraints ("cuts") that reduce the size of the feasible set without eliminating integer feasible solutions.

Two approaches: Cuts at the root node only (standard)

Cuts at every node (Branch-and-cut)

⊖ Having too many cuts slows down the solve of the LP relaxation

⇒ Solvers have a cut management strategy

* Primal heuristics

Idea: Methods to find integer feasible solutions quickly without theoretical guarantees of success

Applying ML to these heuristic components requires:

→ To specify a learning task

→ To find data

→ To design a model to learn from the data

② Learning tasks

a) Learning paradigms

→ Supervised learning: Data $\{(x_i, y_i)\}_{i=1..N}$ with $x \in \mathcal{X}$, $y \in \mathcal{Y}$

Goal: Learn $f(\cdot, \theta) : \mathcal{X} \to \mathcal{Y}$ Model

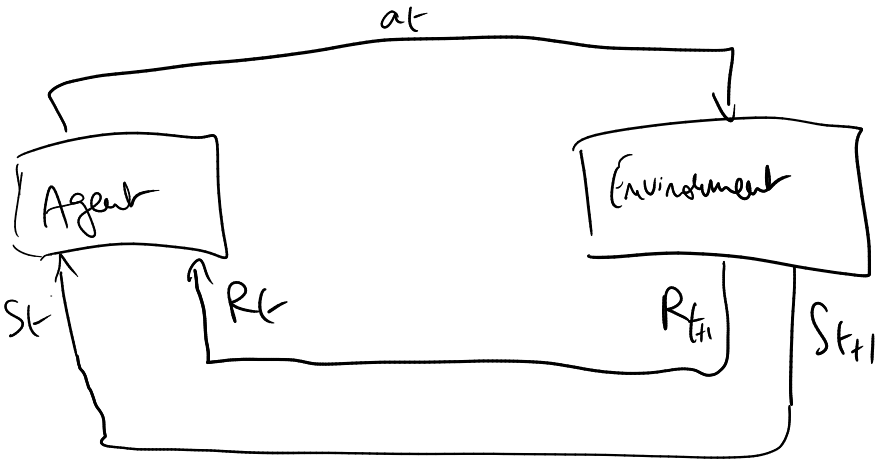
by solving $\underset{\theta \in \Theta}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i, \theta), y_i)$ (loss function)

→ Reinforcement learning

* Based on Markov Decision Processes

a_t: Action at time t

s_t: state at time t



* Goal: Learn a policy $\pi : S \rightarrow A$ that maps states to actions in the best possible way

$$\underset{\pi}{\text{maximize}} \; \mathbb{E}\left[\sum_{t=1}^{T} \gamma^{t-1} R_t\right]$$

T: time horizon

$\gamma \in [0,1]$

$$R_t = R_t(\pi, A_t, S_t, \ldots)$$

→ Standard RL setup: the agent only has access to a reward, not to the best action

→ Imitation learning: Get the best action at every step from an expert

⇒ Equivalent to having some data $\{(s_i, a_i^*)\}$

s_i: state; a_i^*: best action for that state

+ Distinction between online and offline

Offline setting: Learn once and for all (training)
Then run the solver with what you have learned (inference)

Online setting: Do training during the run
⊕ Adaptive
⊖ Cost (running time implementation)

b) Learning tasks for MILP

→ Solver configurations

Ex) SCIP has more than 2000 user-tunable parameters.

Approach: Tune one parameter at a time through learning

One example: (Bonami et al 2018) Predict whether an MIQP (Quadratic objective) should be linearized before solving with CPLEX or not

⇒ Huge impact on preprocessing
⇒ Now implemented in CPLEX

Other example: MIP 2023 challenge
Given past instances, can you learn efficient solver configurations for subsequent instances?

→ Branching strategies

Well-known: strong branching

At node i, suppose the optimal value of the LP relaxation is z^{LP_i}. and that by branching on x_h, get two other optimal values $z^{LP_i^-}$ and $z^{LP_i^+}$

Scores of x_h : $z^{LP_i^+} - z^{LP_i}$ and $z^{LP_i^-} - z^{LP_i}$

⊕ Can drastically reduce the size of B&B tree
⊖ Requires to solve several LPs.

Approaches * Learn to predict strong branching scores
offline (past instances)
or online (as you run B&B)

→ Can learn the scores themselves or just the ranking of the variables according to their scores

* Learn (by imitation) branching rules from scratch

→ Node selection

Classical: Best first search (use best known lower bound)

→ Research from the mathematical programming has been scarce for the past decades
→ New effort due to ML (typically supervised learning / offline)

→ Cutting planes

In a solver: given an LP relaxation, generate a number of cuts of different kinds, select some of them and add them to the relaxation so that the fractional solution of the relaxation becomes infeasible

Approach: Learn cut selection from imitation learning or (standard) RL

(One-cut selection seems easier to learn than multiple-cut selection)

→ Primal heuristics

4 tasks:

offline
- → Given a family of heuristics: learn how to schedule them for a specific instance
  - ↓ which heuristics to run, and for how long
- → Predict a reference solution, i.e. a partial assignment of the integer variables (e.g. by supervised learning on known feasible sets)

(also by supervised learning on known optimal values)

(2) Datasets / Software

Recent developments:
- Ecole → ML (RL) + SCIP solver (≃ Gym for RL)
- MIPLearn

→ Need instance generators and instance distributions
→ Highly dependent on the task
(ex) Learning branching rules should be done in instances for which proving optimality is difficult

→ In construction!

(3) Graph neural networks for (MI)LP

a) Graph Neural Network (GNN)

→ $G = (V, E)$ undirected graph
$\forall$ vertex $v \in V$, $N(v)$: set of neighbors of v

→ Graph embedding $\xi : (G, v) \mapsto \xi(G, v) \in \mathbb{R}^d$
($v \in V$)

$\Rightarrow$ Represents some "features" of v within G

Definition A GNN takes as inputs a graph G and an embedding ξ^0 on G, then defines a recursive embedding by

$$\forall v \in V, \forall t \in \mathbb{N}, \quad \xi^{t+1}(G,v) = \text{comb}\left(\xi^t(G,v), \sum_{u \in N(v)} \xi^t(G,u)\right)$$

comb: combination function, e.g. sum, a layer of a neural network

"Message passing"

Variant based on edge and vertex information

$\rightarrow$ Suppose we have in addition to ξ^0 (embedding on vertices) an embedding $\bar{\xi}$ on edges $\bar{\xi}: (G, e \in E) \mapsto \bar{\xi}(G,e) \in \mathbb{R}^d$

$\rightarrow$ Recursive embedding becomes

$$\xi^{t+1}(G,v) = \text{comb}\left(\xi^t(G,v), \sum_{u \in N(v)} \text{aggr}\left(\xi^t(G,u), \bar{\xi}(G,(u,v))\right)\right)$$

aggr: aggregation function (sum, NN layer, etc.)

Goal: Use GNNs to process MILP data

Challenges: should represent and process data while respecting invariants

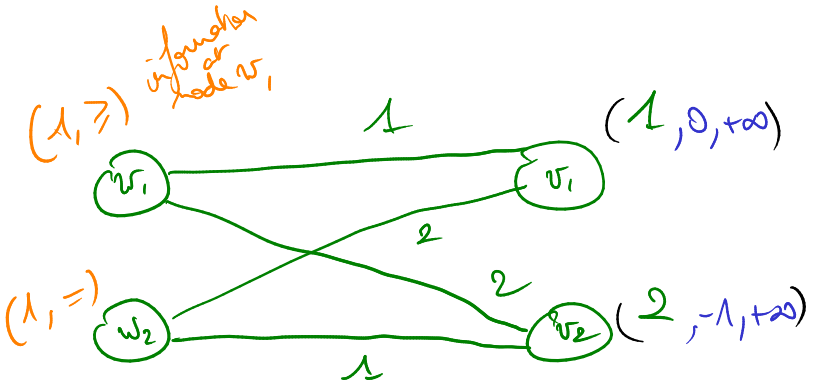
Ex) Permutation of variables / constraints

b) Representation and GNNs for (continuous) linear programs

Key idea: LPs can be represented as bipartite graphs with weighted edges and node information

$$\begin{aligned} \text{minimize}_{x \in \mathbb{R}^2} \quad & 1x_1 + 2x_2 \quad \text{s.t.} \quad x_1 + 2x_2 \geq 1 \\ & 2x_1 + x_2 = 1 \\ & x_1 \geq 0 \\ & x_2 \geq -1 \end{aligned}$$

Bounds



$v_1 \equiv x_1$
$v_2 \equiv x_2$
$w_1 \equiv x_1 + 2x_2 \geq 1$
$w_2 \equiv 2x_1 + x_2 = 1$

Generally

Graph $G = (V \cup W, E)$ bipartite
$|V| = n$ $|W| = m$

Every node is equipped with a feature vector

$$\forall i \in \{1, \dots, n\}, \quad h_i^V \in \underbrace{\mathbb{R}}_{\text{weight of variable } i \text{ in the objective}} \times \underbrace{(\mathbb{R} \cup \{-\infty\}) \times (\mathbb{R} \cup \{+\infty\})}_{\text{bounds on variable } i}$$

$$\forall j \in \{1, \dots, m\}, \quad h_j^W \in \underbrace{\mathbb{R}}_{\text{right-hand side}} \times \underbrace{\{\leq, \geq, =\}}_{\text{nature of the constraint}}$$

↳ Based on this representation, can define a GNN using learnable embedding functions

$\xi^{0,V}: R_i^V \mapsto \xi_i^{0,V} \in \mathbb{R}^{d_0^V}$ $\xi^{0,W}: h_j^W \mapsto \xi_j^{0,W} \in \mathbb{R}^{d_0^W}$

learnable = parameterized functions on which you can apply learning techniques (e.g. NN layers)

= will be learned by training

Every layer l involves learnable functions

comb_l^V, comb_l^W,

and computes

$$h_i^{l,V} = \text{comb}_l^V\left(h_i^{l-1,V}, \sum_{j\in N(i)} E_{ij}\, \text{comb}_l^W\left(h_j^{l-1,W}\right)\right)$$

(E_{ij}: weight of edge (i,j))

$$h_j^{l,W} = \text{comb}_l^W\left(h_j^{l-1,W}, \sum_{i\in N(j)} E_{ij}\, \text{comb}_l^V\left(h_i^{l-1,V}\right)\right)$$

Two sorts of outputs (after L layers)

- Single output $f_{out}\left(\sum_{i=1}^m h_i^{L,V}, \sum_{j=1}^m h_i^{L,W}\right)$
- One output per vertex in V ($\equiv$ variable)

$$\left[f_{out}\left(\sum_{i=1}^m h_i^{L,V}, \sum_{j=1}^m h_j^{L,W}, h_k^{L,W}\right)\right]_{k=1\ldots m}$$

One use case

Want a GNN that given an instance outputs
0 if the problem is infeasible
1 otherwise

in reality, we have a continuous output $\in [0,1]$

Theorem (2023, informal)

Under some assumptions, there exists a GNN such that, if $F(G) \in [0,1]$ is the output of the GNN, then $F(G)$ $\begin{cases} > 1/2 & \text{if the problem is feasible} \\ \leq 1/2 & \text{otherwise} \end{cases}$

c) More advanced use of GNNs in B&B

(Gasse et al '19)

Goal: Learn (strong) branching rules from an expert

LP model $\Leftrightarrow$ LP relaxation at the current node

NB: If the solver only add cuts at the root node, then all the other LP relaxations have same structure

Imitation learning: State: state of the solver: current node and LP relaxation, $\bar{x}, \bar{z}, \ldots$

Actions: strong branching

GNN: 3 layers, mix of convolution + fully connected + ReLU

Output: $\pi \in \mathbb{R}^n$, n number of variables

π probability vector π_i: proba of branching on variable i

Training: Done with GPUs

=> One drawback of GNNs: computational cost

BIBLIOGRAPHY

Y. Bengio, A. Lodi, A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon (2021)

P. Bonami, A. Lodi, G. Zarpellon. Learning a classification of mixed-integer quadratic programming problems (2018)

Z. Chen, J. Liu, X. Wang, W. Yin. On representing linear programs by graph neural networks (2023)

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, A. Lodi. Exact combinatorial optimization with graph convolutional neural networks (2019)

L. Scavuzzo, K. Aardal, A. Lodi, N. Yorke-Smith
Machine learning augmented branch-and-bound for mixed integer linear programming (2024)

For more in the future

→ Papers in ML conferences: NeurIPS, ICLR

→ Papers in optimization/OR journals:
- Mathematical Programming
- European Journal of Operations Research
- INFORMS Journals on { Computing / Optimization