MILP: How to solve your (optimization) problems

Filippo Magi

CommaLab, Univesità di Pisa, filippo.magi@phd.unipi.it

20 Febraury 2026

Mauriana Pesaresi Seminar Series

Mixed Integer Linear Programming (MILP)

Optimization problems

$$f_* : \min\{f(x) : x \in \mathbb{X}\}$$

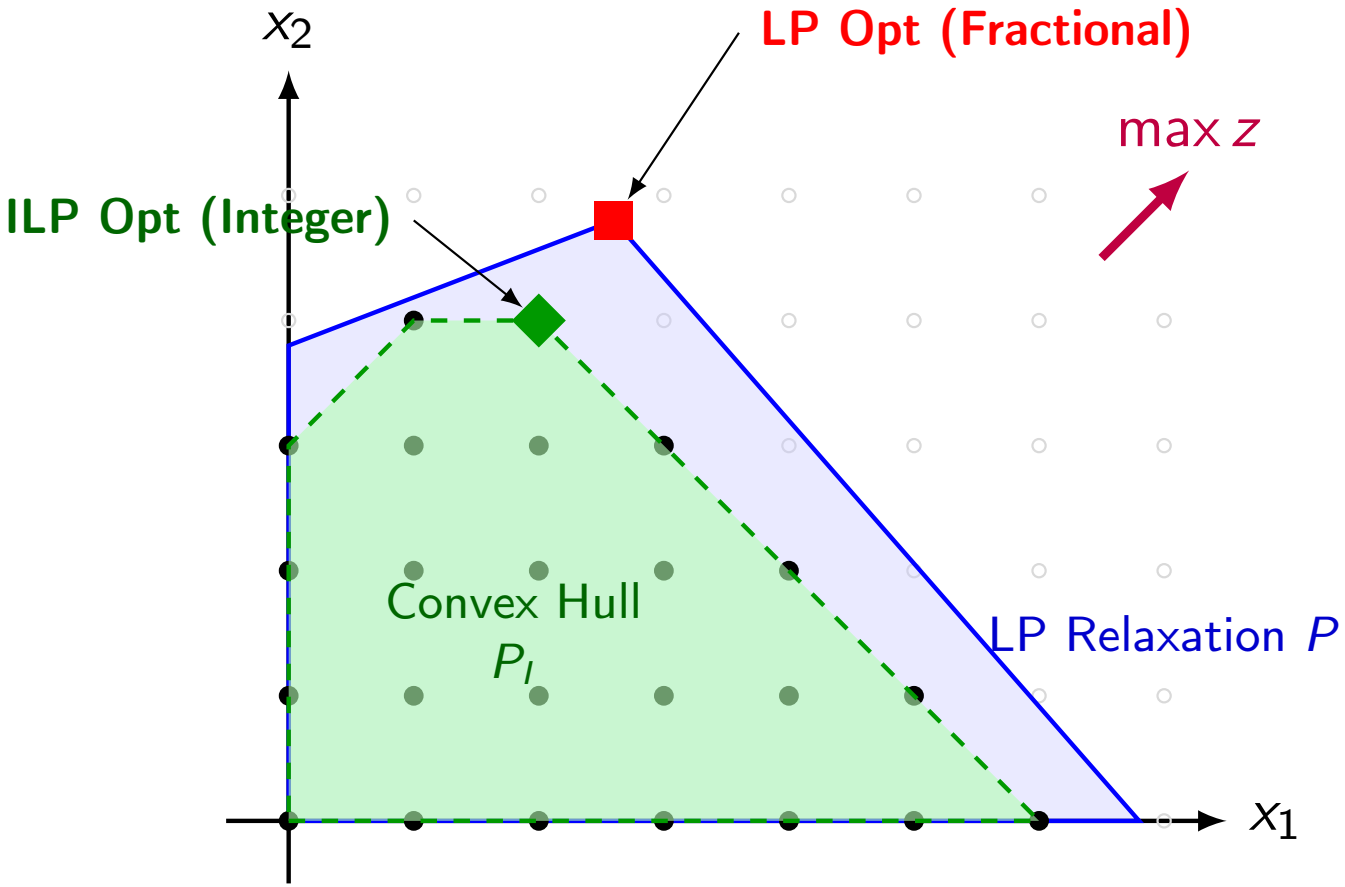
MILP

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^{n-p} \times \mathbb{Z}^p \end{aligned}$$

- Linear objective and constraints
- Integrality constraints model discrete decisions

General MILP is NP-hard

Linear Relaxation and Convex Hull



Is it always so bad?

Total Unimodularity

- A matrix A is **totally unimodular (TUM)** if every square submatrix has determinant $0, \pm 1$
- Assume b is integral

Theorem

Every extreme point of the polyhedron

$$\{x \in \mathbb{R}^n : Ax \leq b\}$$

is integral

- LP relaxation is exact
- MILP reduces to a linear program

Examples: network flow, bipartite matching

Is it always so bad?

A **matroid** is a pair $M = (E, \mathcal{I})$, where E is a finite ground set and $\mathcal{I} \subseteq 2^E$ is a family of *independent sets* such that:

1. $\emptyset \in \mathcal{I}$
2. (*Hereditary property*) if $A \in \mathcal{I}$ and $A' \subseteq A$, then $A' \in \mathcal{I}$
3. (*Exchange property*) for all $A, B \in \mathcal{I}$ with $|A| > |B|$, there exists $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$

Key property

The greedy algorithm produces an optimal solution

Examples: spanning trees

Solving MILPs

Solving a MILP is challenging due to the combinatorial nature of integer constraints. Exhaustive enumeration of all feasible solutions is generally infeasible.
Two classical approaches:

- **Dynamic Programming**: applicable to structured problems (e.g., knapsack, sequencing), solving a smaller number of subproblems.
- **Branch and Bound**: constructs a (binary) search tree and iteratively restricts integer variables, pruning subtrees using bounds.

Dynamic Programming: main idea

Bellman's optimality principle

An optimal policy is composed of optimal sub-policies.

How we can rank policies?

Dominance

Given two sub-policies S' and S'':

- If any extension of S'' can also be appended to S' without increasing cost
- And the cost of S' is less than that of S''

then S' **dominates** S''.

Dynamic Programming: main idea

Notation:

- $c(j)$: cost/value of state j
- L: set of labels (stages), you can see as column of the matrix
- N_l: set of states in label l
- $w_{i,j}$: transition cost from state i to j

Dynamic Programming Recursion

$$\text{Base case: } c(0) = 0$$

$$\text{Recursive step: } c(j) = \min_{i \in N_{l-1}} \{c(i) + w_{i,j}\}, \quad \forall j \in N_l,\ l \geq 1$$

A DP example: binary knapsack

Binary KnapSack Problem

$$\max z : \sum_{i \in I} x_i v_i$$

$$\sum_{i \in I} x_i w_i \leq W$$

$$x_i \in \{0, 1\} \qquad \forall i \in I$$

Let's consider $I = \{1..5\}$ and $W = 15$

Object (i)	1	2	3	4	5
Weight (w_i)	2	3	4	5	8
Value (v_i)	10	15	25	30	50

A DP example: binary knapsack

The rules we used:

- **Initialization:** $z(0, 0)$
- **Extension:** $z(j, u) = \max\{z(j-1, u), z(j-1, a_j) + c_j\}$

With j last item considered and u the capacity used so far

2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0

A DP example: binary knapsack

The rules we used:

- **Initialization:** $z(0, 0)$
- **Extension:** $z(j, u) = \max\{z(j-1, u), z(j-1, a_j) + c_j\}$

With j last item considered and u the capacity used so far

2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	10	10	10	10	10	10	10	10	10	10	10	10

A DP example: binary knapsack

The rules we used:

- **Initialization:** $z(0, 0)$
- **Extension:** $z(j, u) = \max\{z(j-1, u), z(j-1, a_j) + c_j\}$

With j last item considered and u the capacity used so far

2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	15	15	25	25	25	25	25	25	25	25	25	25	25

A DP example: binary knapsack

The rules we used:

- **Initialization:** $z(0, 0)$
- **Extension:** $z(j, u) = \max\{z(j-1, u), z(j-1, a_j) + c_j\}$

With j last item considered and u the capacity used so far

2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	15	15	25	25	25	25	25	25	25	25	25	25	25
10	15	25	25	35	40	40	50	50	50	50	50	50	50

A DP example: binary knapsack

The rules we used:

- **Initialization:** $z(0, 0)$
- **Extension:** $z(j, u) = \max\{z(j-1, u), z(j-1, a_j) + c_j\}$

With j last item considered and u the capacity used so far

2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	15	15	25	25	25	25	25	25	25	25	25	25	25
10	15	25	25	35	40	40	50	50	50	50	50	50	50
10	15	25	30	35	40	45	55	55	65	70	70	80	80

A DP example: binary knapsack

The rules we used:

- **Initialization:** $z(0,0)$
- **Extension:** $z(j,u) = \max\{z(j-1,u), z(j-1,a_j) + c_j\}$

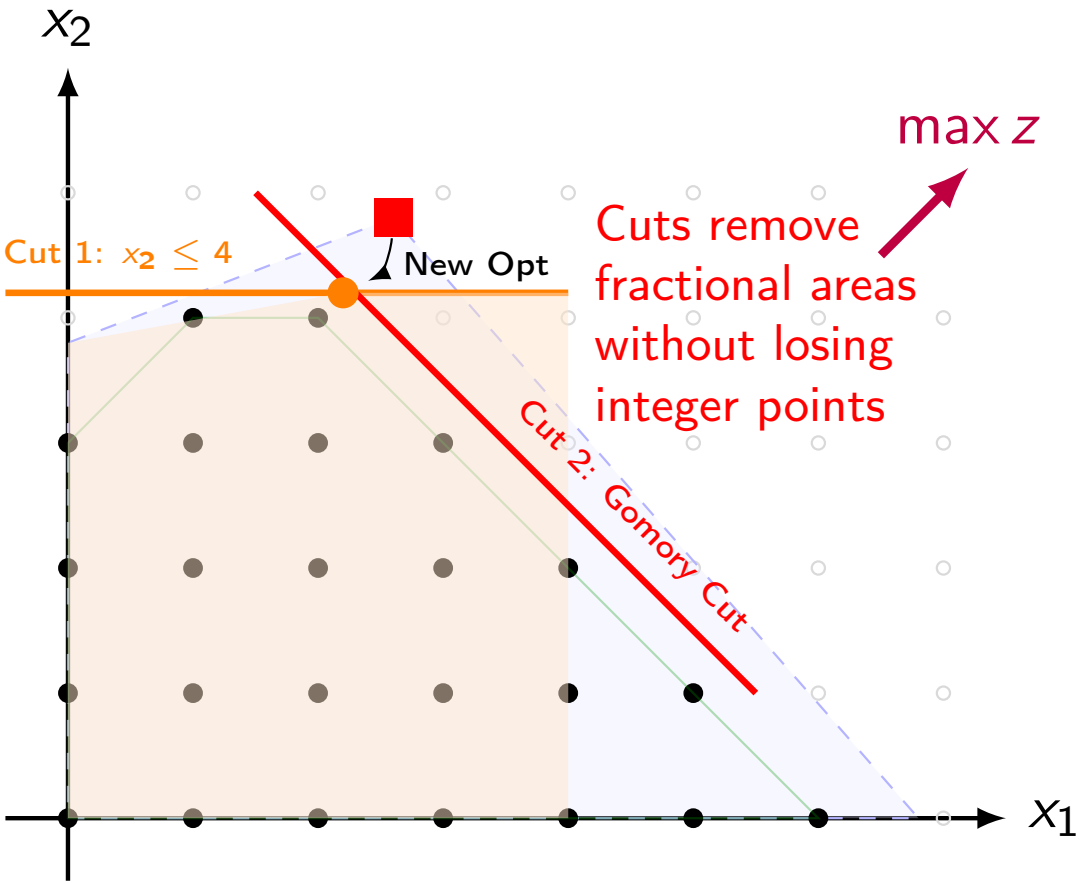
With j last item considered and u the capacity used so far

2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	10	10	10	10	10	10	10	10	10	10	10	10
10	15	15	25	25	25	25	25	25	25	25	25	25	25
10	15	25	25	35	40	40	50	50	50	50	50	50	50
10	15	25	30	35	40	45	55	55	65	70	70	80	80
10	15	25	30	35	40	50	55	60	65	75	80	85	90

Branch and Bound

- **Branch**: recursively partition the feasible region by branching on a fractional integer variable (e.g., $x_i \leq \lfloor x_i^* \rfloor$ or $x_i \geq \lceil x_i^* \rceil$), generating a search tree.
- **Bound**: for each node, compute bounds on the optimal objective value. A node is pruned if it cannot improve the best known feasible solution.
- Bounds:
  - **Primal bound**: objective value of the best feasible (integer) solution found so far, often obtained via heuristics.
  - **Dual bound**: bound obtained from solving a relaxation of the node (e.g., LP, Lagrangian), valid for all solutions in the subtree.

Linear Relaxation and Cutting Planes



Lagrangian relaxation and problem decomposition

Original Problem (Generalized Assignment)

$$\min \quad \sum_{i=1}^{M} \sum_{j=1}^{N} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{i=1}^{M} x_{ij} = 1 \quad \forall j = 1, \ldots, N \quad \text{(Assignment Constraints)}$$

$$\sum_{j=1}^{N} a_{ij} x_{ij} \leq b_i \quad \forall i = 1, \ldots, M \quad \text{(Capacity Constraints)}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j$$

Lagrangian relaxation and problem decomposition

Lagrangian Relaxation 1

Penalty terms for constraints $\sum_{i=1}^{M} x_{ij} = 1$ take the form $\sum_{j=1}^{N} \lambda_j (1 - \sum_{i=1}^{M} x_{ij})$. We obtain:

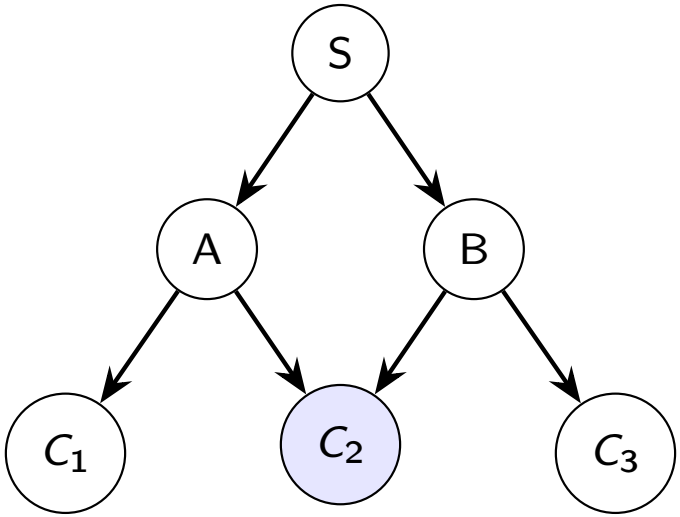
$$\min z_{LR}(\lambda) = \sum_{i=1}^{M} \sum_{j=1}^{N} (c_{ij} - \lambda_j) x_{ij} + \sum_{j=1}^{N} \lambda_j$$

$$\text{s.t.} \quad \sum_{j=1}^{N} a_{ij} x_{ij} \leq b_i \quad \forall i = 1, \ldots, M$$

$$x \in \{0,1\}^{M \times N}$$

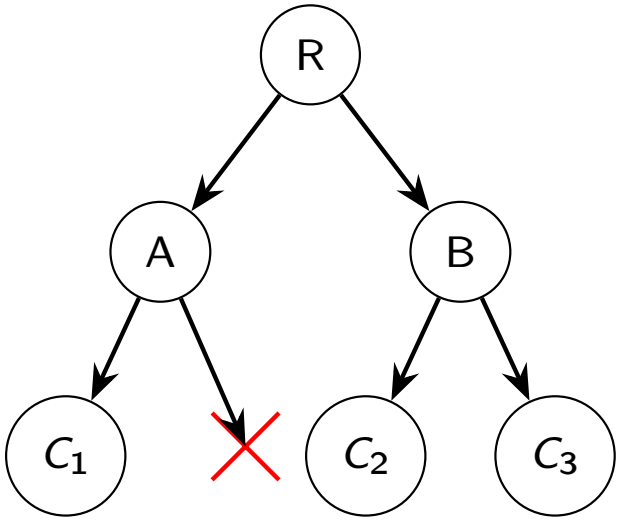
DP vs. Branch and Bound: State Space Structure

Dynamic Programming



Overlapping subproblems:
Directed Acyclic Graph (DAG)

Branch and Bound



Unique decision paths:
Search Tree (with Pruning)

Future Trends in MILP Solving

- **Massive Parallelization**
  - Concurrent exploration of multiple search tree nodes.
  - Simultaneous computation of diverse bounds (e.g., running different Lagrangian relaxations in parallel).
- **Machine Learning (ML) Integration**
  - *Learning to Branch*: Using ML models to predict the most promising branching variables.
  - *Adaptive Relaxations*: Identifying which constraints to relax or which cuts to generate based on instance features.
- **Quantum-Classical Hybrids**