

Matrix representations for MathOptInterface

INFORMS 2021

Benoît Legat

October, 2021



MathOptInterface (MOI)

MOI in a nutshell:

- `add_variable(model)`.
- `add_constraint(model, func, set)`, e.g. $2x + 3y = 1 \rightarrow (2*x + 3*y)\text{-in-EqualTo}(1.0)$.
- `set`, `get` attributes, e.g., `ObjectiveSense`, `ObjectiveFunction`.

Extensible framework:

- **Generic** on attribute, function and set types. New ones can be defined **independently**.
- Solver-**specific** features easily exposed to JuMP/MOI users through **custom** attributes.
- Expose **specialized** problem structure easily through **custom** functions, sets.

Genericity and performance

What is the **cost** of this **genericity** ?

Can MOI be both **extensible** and **performant** ?

Matrix representation

Many **LP** solvers use the following form:

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{R}^N} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to:} \quad & \mathbf{l} \leq \mathbf{A}\mathbf{x} \leq \mathbf{u} \\ & \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u \end{aligned}$$

Many **conic** solvers use the following form:

$$\begin{array}{ll} \min_{\mathbf{x} \in \mathbb{R}^N} & \mathbf{c}^\top \mathbf{x} \\ \text{subject to:} & \mathbf{A}\mathbf{x} + \mathbf{b} \in \mathcal{K} \\ & \mathbf{x} \text{ free.} \end{array} \qquad \begin{array}{ll} \max_{\mathbf{x} \in \mathbb{R}^N} & -\mathbf{b}^\top \mathbf{y} \\ \text{subject to:} & \mathbf{A}^\top \mathbf{y} = \mathbf{c} \\ & \mathbf{y} \in \mathcal{K}, \end{array}$$

Sparse matrices

A is a **sparse** $m \times n$ matrix of k entries. Common formats:

- COO: 3 k -vectors: row indices, columns indices, entries (may be duplicates).
- CSC: 2 k -vectors: row indices, entries. 1 $(n + 1)$ -vector: column pointers.
- CSR: 2 k -vectors: column indices, entries. 1 $(n + 1)$ -vector: row pointers.

Need for **conversion**:

- `MOI.VectorAffineFunction`: COO
- `Multiple MOI.ScalarAffineFunction`: CSR-like (not sorted, with duplicate)
- Most solvers: CSC

COO $\rightarrow$ CSC

1. **Sort** and merge entries $\mathcal{O}(k \log k)$ (may improve with bucket sort)
2. **Allocate** vectors $\mathcal{O}(k + n)$
3. **Fill** vectors $\mathcal{O}(k + n)$

CSR $\rightarrow$ CSC (or CSC transpose)

1. First **pass**: **Count** number of entries per columns $\mathcal{O}(k)$
2. **Allocate** vectors $\mathcal{O}(k + n)$
3. Second **pass**: **Fill** vectors $\mathcal{O}(k + n)$

Canonical functions

- `is_canonical`: $\mathcal{O}(k)$ time, 0 allocations.
- `canonical`: $\mathcal{O}(k \log k)$ time, $\mathcal{O}(k)$ allocations.
- First and second pass need **canonical** MOI functions, i.e., sorted and without duplicates.
- JuMP creates **canonical** functions.
- Bridges may **break** canonicalization.
- `MOI.modify` may **break** canonicalization.
- `MOI.Utilities.Model` need to **copy** function given by user: might as well canonicalize them as well.
- So `MOI.Utilities.Model` functions are **canonical** unless `MOI.modify`'d
- `CanonicalConstraintFunction` check `is_canonical` and canonicalize only if needed.

Copy without bridges

VectorOfConstraints to MatrixOfConstraints

1. First pass: Collect functions with `CanonicalConstraintFunction` and `count` number of entries by column: no allocation if `modify` not called. Function not mapped with `IndexMap` to avoid allocation.
2. `Allocate` CSC representation.
3. Second pass: Go through collected function and `fill` CSC representation.
 - Canonicalization done `once` as functions are collected.
 - Indices are mapped `twice` (once per pass) but mapped function not allocated and mapping is `cheap` (`CleverDicts` is a `Vector` if no variable were deleted).

Copy with bridges

- All constraint types not bridged passed **through** bridge layers with `pass_nonvariable_constraints` to avoid **allocating** mapped functions.
- Bridged constraints add function with **mapped** indices that need to be canonicalized.
- **Mix** of functions with indices mapped or not mapped.
- The bridge layer still **allows** performance optimization.
- The bridging of some constraints types does **not affect** performance of other constraint types.

Only two copies

The number of nonzero entries k might be **large**: we don't want to allocate too many **copies**.

- CachingOptimizer: One copy of **user** model in VectorOfConstraints form.
- LazyBridgeOptimizer: Bridged layer.
- CachingOptimizer: One copy of **solver** model in MatrixOfConstraints form (0-based indexed if needed).
- Solver.

Copy of solver model is solver **specific**: depends on supported constraints, 0-based vs 1-based, splitting of matrices between constraint types...