GBOML: A Structure-Exploiting Optimization Modelling Language in Python

Bardhyl Miftari[1], Mathias Berger[1], Guillaume Derval[1], Quentin Louveaux[1], and Damien Ernst[1,2]

[1]University of Liège, Belgium;
[2]LTCI, Telecom Paris, Institut Polytechnique de Paris, France

ARTICLE HISTORY
Compiled September 8, 2023

ABSTRACT
Mixed-Integer Linear Programs (MILPs) have many practical applications. Most modelling tools for MILPs fall in two broad categories. Indeed, tools such as algebraic modelling languages allow practitioners to compactly encode models using syntax close to mathematical notation but usually lack support for special structures, while other tools instead provide predefined components that can be easily assembled but modifying or adding new components is difficult. In this work, we present the inner workings of the Graph-Based Optimization Modelling Language (GBOML), an open-source modelling tool implemented in Python combining the strengths of both worlds. GBOML natively supports special structures that can be encoded by a hierarchical hypergraph, offers syntax close to mathematical notation and facilitates the modular construction and reuse of time-indexed models. We detail design choices enabling these features and show that they simplify problem encoding, lead to faster instance generation times and sometimes faster solve times. We benchmark the times taken by GBOML, JuMP, Plasmo, Pyomo and AMPL to generate instances of a structured MILP. We find that GBOML outperforms Plasmo and Pyomo, is tied with JuMP but is slower than AMPL. With parallel model generation, GBOML outperforms JuMP and closes the gap with AMPL. GBOML has the smallest memory footprint.

KEYWORDS
Optimization; Linear Programming; MILP; Graph; Structure; Exploitation; Modelling; Tool;

1. Introduction

Many real-life problems can be represented and tackled as mathematical programs[12, 16, 33]. In particular, Mixed-Integer Linear Programming (MILP) often provides an exact representation of many problems or enables one to approximate some nonlinear problems via piecewise-affine approximations of nonlinear functions. Applications for which MILP is particularly suitable include energy system planning and control as well as supply chain management[4, 39, 50].

Working with mathematical programs often involves four basic steps. First, a practitioner must come up with an abstract mathematical representation of the

CONTACT: Bardhyl Miftari. Email: bmiftari@uliege.be

problem, which is often referred to as a formulation, and is composed of parameters, variables, constraints and objectives. Second, this abstract representation must be encoded in a format intelligible by a computer, which is typically done via a modelling tool. Then, the encoded problem is passed to a solver which solves it. Finally, the results of the solution provided by the solver are retrieved and post-processed.

This paper focuses on the second aforementioned step, namely modelling tools. Modelling tools are widely used by practitioners in the field of mathematical programming [45] and they have also been gaining momentum in related fields where their use was traditionally limited. For instance, in the field of constraint programming, where the problem formulation was traditionally linked directly to a solver, more generic modelling tools such as Essence[22], CMPpy[24], MiniZinc[40] and XCSP3[8] have emerged.

Most modelling tools for mathematical programming are used to perform four essential tasks: **encoding** a high-level mathematical representation of a model, transforming it into a **lower-level representation** suitable for numerical computing, **communicating with solvers** and **processing the solution**. During the **encoding**, the user writes their problem in a modelling language or framework provided by the modelling tool. The encoded problem is then converted to a **lower-level representation** that is suitable for numerical computing, usually in the form of a collection of matrices. The modelling tool then communicates this lower-level representation to a solver, along with other information such as parameters used for tuning algorithms. Finally, if a solution is found by the solver, it is retrieved and **processed** into standard file formats and output. If no solution is found, the modelling tool retrieves and returns information about the status of the model and solver.

Modern modelling tools for mathematical programming typically fall into two broad categories: Algebraic Modelling Languages (AMLs) and what we refer to as Object-Oriented Modelling Environments (OOMEs). On the one hand, AMLs are tools that enable the encoding of a broad class of mathematical problems (such as mixed-integer non-linear programs) with syntax close to the mathematical notation. They enable models to be encoded using mathematical expressions involving parameters and variables, from which the constraints and objectives of a given problem are defined. On the other hand, OOMEs adopt a more object-oriented approach by providing the users with a library of generic pre-existing components that are easy to manipulate, reuse and assemble in order to build larger models, in a fashion similar to simulation tools such as Simulink[44] or Modelica[38]. Some OOMEs rely themselves on AMLs. In the next two paragraphs, we explain each category in greater detail.

AMLs, in addition to offering syntax that is expressive and close to standard mathematical notation, can either be available as stand-alone tools or directly embedded in a programming language. Embedded AMLs usually come in the form of a package that can be imported in scripts or code that use data structures already available in the programming language. By contrast, stand-alone AMLs define their own language and grammar. Stand-alone AMLs rely on a custom lexer and parser to read the information contained in a file defining a model. Some well-known AMLs are JuMP[36], Pyomo[11], AMPL[18], GAMS[10] and Mosel[26]. JuMP[36] is an open-source embedded AML implemented in Julia[6]. It deals with linear, mixed-integer,

second-order cone, semidefinite, and nonlinear programming[1] with set programming and Mixed Complementarity Problems (MCP) modelling capabilities enabled via extensions. Pyomo[11] is also an open-source AML embedded in Python[49] that deals with linear, quadratic and nonlinear programming for both continuous and mixed-integer variables. AMPL[18] and GAMS[10] are well-established commercial stand-alone modelling tools that deal with a similar class of problems as Pyomo. AMPL also supports constraint programming[20]. The previous tools all communicate with various solvers, enabling the user to try different algorithms and solvers to find a solution to their problems. By contrast, Mosel[26] is a commercial stand-alone AML that communicates with only one solver, namely FICO Xpress[35]. It deals with MILP, convex quadratically constrained quadratic programming and second-order cone programming.

The second category is the OOMEs. OOMEs' main features are the easy and modular construction of models via the combination of predefined components or the reuse of models. Typically, the predefined components are application-specific and often contain some data (e.g., some reference or default parameter values for a system) that can be updated. Often, users do not have direct access to the model that underpins predefined components. Introducing new components or substantially modifying predefined ones is usually cumbersome. Notable OOMEs for energy systems and supply chain management are Calliope[41], PyPSA[9] and MISPT[28]. Calliope is a multi-scale energy system modelling framework. It revolves around four types of components: technologies, carriers, locations and resources. Models are defined in YAML files by providing instances of these components. PyPSA is a toolbox for power system optimization. PyPSA relies on Pyomo and defines power systems as networks made up of components such as buses, lines, transformers, generators, storage units and loads. Power system planning and operation are common use cases of PyPSA. MISPT is a MILP production planning tool. It is typically useful for mixed-time models (i.e., a combination of continuous and discrete-time models) and multistage multi-product production problems.

Both AMLs and OOMEs suffer from certain drawbacks. For a start, in their basic form, AMLs usually fail to expose and exploit the structures that naturally arise in some MILPs or allow any kind of reuse of sub-parts (or whole parts) of models. More specifically, most AMLs fail to expose the special structure that exists in many MILPs encountered in practical applications to specialized solvers. Then, OOMEs lack expressiveness and the models that underpin existing components are often difficult to access, making the modification or addition of new components complicated. In addition, OOMEs often rely on AMLs themselves, in which case they also inherit their drawbacks.

Nonetheless, there exist extensions to AMLs such as SAMPL[48] (for AMPL) or Extended Mathematical Programming (EMP)[14] that enable AMLs to capture and expose special problem structures. SAMPL is an extension designed to handle stochastic programming. It enables the encoding of structure through stages of optimization and interfaces with structure-exploiting methods. EMP is a general framework that uses specific annotations to automatically reformulate classes of problems such as stochastic programming, disjunctive programming or complementarity problems

[1] https://jump.dev/JuMP.jl/stable/

into equivalent problems and facilitate modelling and the use of specialized solvers. At present, there is only one specific EMP implementation that extends GAMS. In its stochastic reformulation, EMP supports the definition of stages similarly to SAMPL. In addition to these extensions, a few modelling tools such as Plasmo[30] and SMS++[21] also try to bridge the gap between AMLs and OOMEs by combining modularity, expressiveness and support for special problem structures. Plasmo[30] is an open-source embedded modelling framework built on top of JuMP. It uses a graph abstraction to build hierarchical models while still benefiting from the expressiveness of JuMP. It exposes the block structure of models by enabling the encoding of nodes and edges. SMS++[21] is an embedded open-source structured modelling system for optimization models, implemented in C++. Specifically, it enables the modelling of hierarchical block-structured models by using the AbstractBlock class, also exposing the block-decomposable structure to specialised solvers and algorithms. However, none of these tools natively supports the straightforward reuse of model components, which is the core feature of OOMEs.

The Graph-Based Optimization Modelling Language (GBOML) [37] is a stand-alone open-source modelling tool for MILPs implemented in Python that combines the strengths of AMLs and OOMEs. More precisely, the tool supports special structures that can be encoded by a hierarchical hypergraph, offers syntax close to mathematical notation and facilitates the modular construction and reuse of time-indexed models. It also interfaces with various commercial and open-source solvers, including structure-exploiting ones. In this paper, we present the key objectives behind the development of GBOML, the key design choices that enable the aforementioned features and the inner workings of the tool. In particular, we discuss the benefits of maintaining a symbolic representation of models, exploiting their underlying structures and resorting to delayed evaluation. We also illustrate the use of GBOML with different examples and benchmark its performance against well-established optimization modelling tools.

This paper is structured as follows. The design aims of GBOML are first summarized in Section 2. Then, in Section 3, the hierarchical hypergraph abstraction of mixed-integer linear programs that underpins GBOML is discussed. Next, the inner workings of GBOML are explained in Section 4 to better understand how it represents problems internally and enables a key set of features. Two examples are provided in Section 5. We discuss the main features in addition to a benchmark of Plasmo[30], JuMP[36], Pyomo[11], AMPL[18] and GBOML[37] on a realistic energy system planning problem[4] in Section 6. The paper ends with a conclusion and a brief discussion of future work avenues in Section 7.

2. Aims and Requirements

Modelling tools are all about ease of use and convenience[32]. Indeed, modelling tools for mathematical programming were originally created to ease the burden of implementing and solving models for practitioners [19]. With this in mind, in the following, we explain the core aims and features that drove the development of GBOML.

Mixed-Integer Linear Programming

As mentioned before, MILP is applied to tackle a large range of problems that arise in various fields such as energy system planning and supply chain management. GBOML allows users to encode any MILP using syntax resembling that of traditional AMLs. In other words, defining parameters, variables, constraints or objectives is straightforward to do and relies on syntax close to standard mathematical notation.

Stand-Alone and Lightweight

GBOML focuses on a restricted set of key features that are easy to use and maintain. Its core is as simple as possible. Specifically, GBOML is designed to deal with structured Mixed-Integer Linear Programming and the aim is to do that well rather than extending to other classes of problems. Being stand-alone has both advantages and disadvantages. On the one hand, it does not benefit from the expressiveness and functionalities of an existing AML on top of which it could be built (e.g., that already interfaces with a broad range of solvers), and developing a stand-alone tool thus requires more implementation work. On the other hand, being stand-alone gives more freedom at the design stage as well as greater control over the implementation and maintenance of the tool.

Model Reuse and Modular Construction

GBOML implements features similar to those of OOMEs by allowing any element defined in a GBOML file to be reused elsewhere. It supports the reuse of models in parts or as a whole. More precisely, one core feature is that any block defined could be easily imported and reused in another model in a library-like manner. Combining blocks of models or full models into a larger one is of primary importance. Therefore, the definition of these entities is as self-contained as possible to enable their easy manipulation. It should be noted that no two models are usually exactly identical and there is a need for the elements and models to be slightly adapted to one particular use case by enabling, for instance, a change in the values of parameters.

Structured Models

Many real-life problems possess a time dimension and a natural block structure wherein a component (e.g., a power plant) of a large system (e.g., the power system) can be seen as a block [4]. The tool allows the representation of *block structures* that can be encoded by a hierarchical hypergraph, and its core revolves around these so-called *blocks*. Structure is defined via special language constructs and can be exploited during model generation and solving by exposing it to specialised algorithms. In addition, the tool facilitates the encoding and generation of *time-indexed models*. In Section 3, we provide a more detailed formulation of the problems that we consider.

Solvers

Interfaces to both *commercial* and *open-source* solvers are provided. GBOML communicates with a wide range of solvers and has an intermediate representation sufficiently generic so that it can be adapted to fit the format expected by various solvers. Com-

mercial solvers are the state-of-the-art, with the likes of Gurobi[25], CPLEX[29] and Xpress[35] being amongst the fastest solvers, but their price is often a barrier for a lot of users. In order to be used by everyone, free open-source alternatives also exist for solvers, such as HiGHS[27] and CBC[17]. Furthermore, in order to exploit the structure, interfaces to structure-exploiting solvers that implement methods such as the Dantzig-Wolfe and Benders algorithms have been implemented.

Open-source and Language

In order to promote transparency and to make it widely accessible, GBOML is released under the MIT Licence[46]. It is indeed paramount that both academics and industry practitioners have access to GBOML. Furthermore, releasing its source code enables practitioners to study and modify the software in order to enhance it or customize it to better suit their needs. Hence, providing full open access can also improve the tool's reliability. The tool was implemented in Python, owing to its ease of use, its object-oriented nature (which facilitates the implementation of key features of the tool) and the fact that it has a very broad user base spanning both industry and academia.

3. Structure Formalization

Let us first provide a concise reference describing the notation. We denote a scalar with a standard-font symbol a or A (uppercase for constants), a vector with a bold lowercase letter $\mathbf{a}$, a matrix with a bold uppercase letter $\mathbf{A}$, a set with a calligraphic uppercase letter $\mathcal{A}$ and a tuple or graph-related object with a sans-serif letter a or A. The i^{th} element of a vector $\mathbf{x}$ is noted $x[i]$.

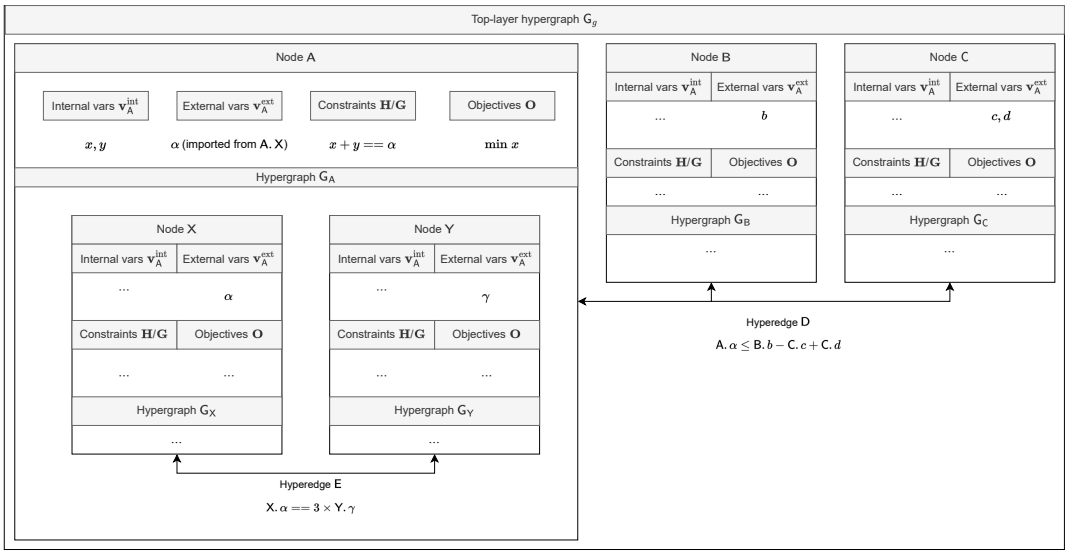


Figure 1. The hypergraph $\mathsf{G_g}$ is the top layer hypergraph made up of three nodes A, B and C and one hyperedge D. The node A contains a hypergraph $\mathsf{G_A}$ made-up of two nodes X and Y and a hyperedge E.

In this work, we focus on time-indexed mixed-integer linear problems containing a block-decomposable structure where each block has its own variables, constraints

and objectives. Each block can itself contain other blocks, in a tree-like fashion. The blocks are interconnected by constraints that link a subset of the variables of several blocks. To formalize the problems, we take a hierarchical hypergraph abstraction.

We consider a hypergraph $\mathsf{G} = (\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ denotes a set of nodes, each node represents a block that can itself contain a hypergraph and $\mathcal{E}$ the hyperedges connecting these blocks. The highest level hypergraph, the one that is not contained in a node, is also called the top-layer hypergraph G_g composed of the so-called top-layer nodes $\mathcal{N}_g$ and hyperedges $\mathcal{E}_g$. All nodes and hyperedges that do not belong to the top-layer are called subnodes and sub-hyperedges, i.e. nodes and hyperedges that belong to a hypergraph contained in another node, their definition being otherwise identical. An example of hierarchical hypergraph is given in Figure 1.

Each node n is defined as a tuple $< \mathbf{v}_\mathsf{n}^{ext}, \mathbf{v}_\mathsf{n}^{int}, \mathbf{G}_\mathsf{n}, \mathbf{H}_\mathsf{n}, \mathsf{G}_\mathsf{n}, \mathbf{O}_\mathsf{n} >$ where:

- $\mathbf{v}_\mathsf{n}^{ext}$ and $\mathbf{v}_\mathsf{n}^{int}$ respectively denote the external and internal vector variables such as their concatenation is denoted $\mathbf{v}_\mathsf{n} = \mathbf{v}_\mathsf{n}^{ext} \oplus \mathbf{v}_\mathsf{n}^{int}$,
- $\mathbf{G}_\mathsf{n} \in \mathbb{R}^{\psi_\mathsf{n} \times (1+|\mathbf{v}_\mathsf{n}|)}$ and $\mathbf{H}_\mathsf{n} \in \mathbb{R}^{\eta_\mathsf{n} \times (1+|\mathbf{v}_\mathsf{n}|)}$ denote the inequality and equality constraints where ψ_n is the number of inequality constraints in node n and η_n the number of equality constraints, such that

$$\mathbf{G}_\mathsf{n} \begin{bmatrix} 1 \\ \mathbf{v}_\mathsf{n} \end{bmatrix} \leq \mathbf{0}, \quad \mathbf{H}_\mathsf{n} \begin{bmatrix} 1 \\ \mathbf{v}_\mathsf{n} \end{bmatrix} = \mathbf{0},$$

- G_n is the hypergraph $(\mathcal{N}_\mathsf{n}, \mathcal{E}_\mathsf{n})$ contained in node n where $\mathcal{N}_\mathsf{n}$ represents the set of subnodes of the hypergraph contained in node n and $\mathcal{E}_\mathsf{n}$ the set of sub-hyperedges,
- and the matrix $\mathbf{O}_\mathsf{n} \in \mathbb{R}^{\sigma_\mathsf{n} \times (1+|\mathbf{v}_\mathsf{n}|)}$ represents the objective function to minimize with σ_n representing the number of objectives defined in node n,

$$\min \ \mathbf{1}^{1 \times \sigma_\mathsf{n}} \ \mathbf{O}_\mathsf{n} \begin{bmatrix} 1 \\ \mathbf{v}_\mathsf{n} \end{bmatrix},$$

  $\mathbf{1}^{1 \times \sigma_\mathsf{n}}$ being the matrix filled with ones of size $(1 \times \sigma_\mathsf{n})$.

It should be noted that the variables of each node are not necessarily independent from the variables of its subnodes. In other words, if we note by n_1 a parent node and n_2 its child node, there can exist a variable v such that $v = v_{\mathsf{n}_1}[i]$ and $v = v_{\mathsf{n}_2}[j]$ where i and j are not necessarily equal. We say that n_1 *imports* the variable from n_2.

The hyperedges $\mathsf{e} \in \mathcal{E}_\mathsf{g}$ and sub-hyperedges $\mathsf{e} \in \mathcal{E}_\mathsf{n}$ connect the external variables $\mathbf{v}_\mathsf{n}^{ext}$ of several nodes $\mathcal{N}_\mathsf{e}$ of a given layer. Each hyperedge is defined as a tuple $< \mathcal{N}_\mathsf{e}, \mathbf{G}_\mathsf{e}, \mathbf{H}_\mathsf{e} >$ where $\mathcal{N}_\mathsf{e}$ is the set of nodes concerned by the hyperedge e. All the nodes in $\mathcal{N}_\mathsf{e}$ belong to the same hypergraph G. For ease of writing, let us note $\mathbf{v}_\mathsf{e} = \bigoplus_{\mathsf{n} \in \mathcal{N}_\mathsf{e}} \mathbf{v}_\mathsf{n}^{ext}$. $\mathbf{G}_\mathsf{e} \in \mathbb{R}^{\psi_\mathsf{e} \times (1+|\mathbf{v}_\mathsf{e}|)}$ denotes the inequality constraints and $\mathbf{H}_\mathsf{e} \in \mathbb{R}^{\eta_\mathsf{e} \times (1+|\mathbf{v}_\mathsf{e}|)}$ the equality constraints such as

$$\mathbf{G}_\mathsf{e} \begin{bmatrix} 1 \\ \mathbf{v}_\mathsf{e} \end{bmatrix} \leq \mathbf{0}, \quad \mathbf{H}_\mathsf{e} \begin{bmatrix} 1 \\ \mathbf{v}_\mathsf{e} \end{bmatrix} = \mathbf{0}.$$

In order to state the overall problem, let us define the following recursive functions,

- the function f that takes a set of nodes $\mathcal{N}$ as input and returns the sum of the objectives of the nodes and their subnodes recursively,

$$f(\mathcal{N}) = \sum_{\mathsf{n} \in \mathcal{N}} \left(\mathbf{1}^{1 \times \sigma_{\mathsf{n}}} \, \mathbf{O}_{\mathsf{n}} \begin{bmatrix} 1 \\ \mathbf{v}_{\mathsf{n}} \end{bmatrix} + f(\mathcal{N}_{\mathsf{n}}) \right).$$

- the *Boolean-valued* function g that takes a hypergraph $\mathsf{G} = (\mathcal{N}, \mathcal{E})$ as input and returns,

$$g(\mathsf{G}) = \left[\mathbf{G}_{\mathsf{e}} \begin{bmatrix} 1 \\ \mathbf{v}_{\mathsf{e}} \end{bmatrix} \leq \mathbf{0} \; \forall \mathsf{e} \in \mathcal{E} \right] \wedge \left[\left(\mathbf{G}_{\mathsf{n}} \begin{bmatrix} 1 \\ \mathbf{v}_{\mathsf{n}} \end{bmatrix} \leq \mathbf{0} \wedge g(\mathsf{G}_{\mathsf{n}}) \right) \forall \mathsf{n} \in \mathcal{N} \right].$$

  The left-hand side of the `and` operator $\wedge$ represents the inequality constraints in the set of hyperedges $\mathcal{E}$ whereas, the right-hand side the inequality constraints of the set of nodes $\mathcal{N}$ and its sub-hypergraph G_{n}.
- the *Boolean-valued* function h that takes a hypergraph $\mathsf{G} = (\mathcal{N}, \mathcal{E})$ as input and returns,

$$h(\mathsf{G}) = \left[\mathbf{H}_{\mathsf{e}} \begin{bmatrix} 1 \\ \mathbf{v}_{\mathsf{e}} \end{bmatrix} = \mathbf{0} \; \forall \mathsf{e} \in \mathcal{E} \right] \wedge \left[\left(\mathbf{H}_{\mathsf{n}} \begin{bmatrix} 1 \\ \mathbf{v}_{\mathsf{n}} \end{bmatrix} = \mathbf{0} \wedge h(\mathsf{G}_{\mathsf{n}}) \right) \forall \mathsf{n} \in \mathcal{N} \right].$$

A compact representation of the hierarchical hypergraph abstraction of a block decomposable problem is given as,

$$\begin{aligned} \min \quad & f(\mathcal{N}_g) \\ \text{s.t.} \quad & h(\mathsf{G}_g) \text{ is true} \\ & g(\mathsf{G}_g) \text{ is true.} \end{aligned} \tag{1}$$

4. GBOML Implementation

In this section, we discuss the implementation of GBOML, which comes as a Python package installable via `PyPI`[43] containing an executable and a Python library. The executable directly takes GBOML text files as input, while the Python library enables the handling of models encoded in GBOML files (e.g., importing blocks, combining components, redefining parameters, etc.).

In the following, we first explain the general workflow of the tool by providing a high-level view of its different components. Then, we provide a step-by-step explanation of the different representations of a model, namely as encoded in the GBOML language, as a syntax tree and as a collection of matrices, as well as the transition from one form to the other. We explain how our design goals are attained along the way.

4.1. Overall Workflow

The GBOML workflow is divided into five main steps:

- **Parsing**: The goal of this step is to convert the plain information contained in a text file written in the GBOML language into a syntax tree. To do so, the executable calls the GBOML parser upon the file it reads. In terms of parsing,

we use an LALR(1) parser[1]. We use a lexer to convert the file in a stream of predefined tokens and the LALR(1) parser to convert the stream of tokens to a syntax tree.

- **Analysis**: In the second step, the syntax tree is checked and augmented with further information. These checks ensure that all elements in the input are valid (e.g., that the constraints and objectives are indeed linear).
- **Evaluation**: Once all the information has been gathered in a structured tree, we proceed with the evaluation of the expressions in order to generate a matrix representation of the problem.
- **Solving**: The matrix representation augmented with syntax tree information (such as variable types, structure, ...) are passed to the solver, which solves the problem and sends back the solution if it exists.
- **Output**: Once the solution has been retrieved, the output is stored in a standard file format (CSV or JSON) reflecting the structure of the original model.

This division in different steps is one of the key design choices enabling crucial features discussed in Section 2. It introduces a clear separation between the model declaration and the building of the problem instance which needs to be evaluated, analyzed and solved. In this sense, GBOML is a modelling language that follows the precepts described in [19]. It generates a *symbolic* representation of the model before creating an instance. In practice, this means that all the instructions are read at once and added to the model. Then, everything is checked and the evaluation only occurs after that step. To borrow the terms used in [32], the model 'waits to be transformed into another form'. We refer to the evaluation step as *delayed*, as opposed to the immediate evaluations that occur in imperative modelling tools that are usually embedded in an imperative programming language. Figure 2 illustrates the different stages of the GBOML parser. In the following subsections, we explain the inner workings of GBOML. More precisely, we explain the different representations of the model shown in blue in Figure 2.

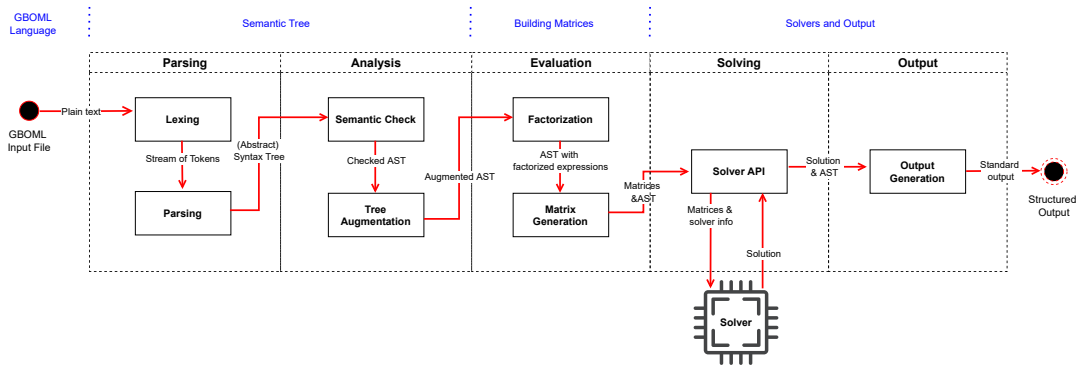


Figure 2. The inner workings of GBOML and the different representations of the model. The high-level representation of models encoded in GBOML is written in blue.

4.2. GBOML Language

A model must first be encoded in the GBOML language, which is the first representation of the model that the tool works with, as shown in Figure 2. GBOML relies

on a hypergraph abstraction for block-structured time-indexed mixed-integer linear programs. This abstraction is composed of `nodes`, `hyperedges` and a special `time` index. Therefore, these three basic constructs of the language are implemented via the `#NODE`, `#HYPEREDGE` and `#TIMEHORIZON` keywords. The basic blocks of a GBOML file are given as follows:

```
#TIMEHORIZON ...

#NODE
// node definition

#NODE
// node definition
...

#HYPEREDGE
// hyperedge definition
```

The `#TIMEHORIZON` block defines the optimization horizon `T`. In GBOML, defining the optimization horizon `T` leads to the automatic definition of a special index t that can be used throughout the file. Constraints and objectives that use t are automatically expanded for every $t \in \{0, \dots, \mathtt{T}-1\}$ (i.e., they are evaluated for each such value of t). Each `#NODE` block is composed of its own set of parameters, variables, constraints and objectives. Each `#HYPEREDGE` block is composed of its own set of parameters and constraints that link variables defined in one or several `nodes`. The basic structure of a `#NODE` is given by:

```
#NODE <node identifier>
#PARAMETERS
// parameter definitions
#VARIABLES
// variable definitions
#CONSTRAINTS
// constraint definitions
#OBJECTIVES
// objective definitions
```

and the structure of a `#HYPEREDGE` is given by:

```
#HYPEREDGE <hyperedge identifier>
#PARAMETERS
// parameter definitions
#CONSTRAINTS
// constraint definitions
```

Nodes can themselves be composed of several *subnodes* and *sub-hyperedges*, and they therefore represent *hierarchical hypergraphs*. It should be noted that nodes contains all the information related to the tuple $< \mathbf{v}_{\mathsf{n}}^{ext}, \mathbf{v}_{\mathsf{n}}^{int}, \mathbf{G}_{\mathsf{n}}, \mathbf{H}_{\mathsf{n}}, \mathsf{G}_{\mathsf{n}}, \mathbf{O}_{\mathsf{n}} >$ defined in Section 3 and augmented by parameters. The same stands for hyperedges, where the level in which a hyperedge is defined determines the nodes that can be linked by that hyperedge.

4.3. Semantic Tree

Once the model has been encoded in the GBOML language, the second step is to take the file and convert it into a syntax tree, as shown in Figure 2. The parser

and lexer read the file line by line and convert it into a semantic tree which is an in-memory representation of the problem. This subsection explains this representation and explains the typical advantages of this representation that help one to better understand the involvement of the internal representation for our goals.

The lexer[1] takes the file written in GBOML language and cuts it into a stream of `tokens`, basically tuples of `<type, value>` where the `type` corresponds to special language types such as *number* or *identifier* and `value` the value defined in the file. The parser[1] takes the stream of tokens and matches a grammar to it in order to build an abstract syntax tree.

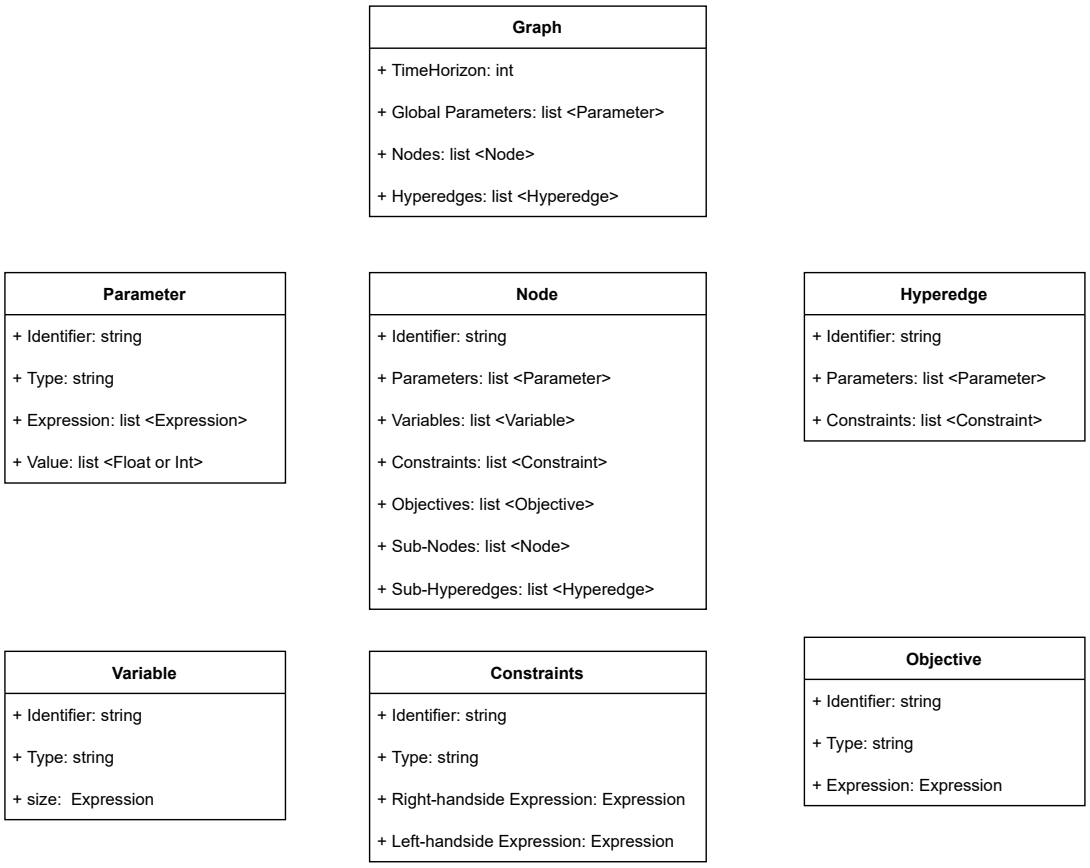


Figure 3. In-memory data structure of GBOML.

Various ways of building syntax trees exist. In our case, we wanted to keep our core structure as simple as possible while also **structuring the information to correspond to our class of problems**. The core of GBOML revolves around expressions and time indices. Each GBOML model contains a time-horizon definition, and a list of `global` parameters, `nodes` and `hyperedges` inside a structure named **Graph**. When talking about a particular computer structure, the name of the said structure is put in bold. The **Graph** outputted by the parser is a directed acyclic graph (represented by a reference tree). The **Node** objects contain a unique identifier, a list of parameters, a list of variables, a list of constraints, and a list of objectives. They can also contain a Graph in the form of a list of `sub-nodes`

and `sub-hyperedges` that are simply a list of references to other **Node** objects. It matches the definition made in Section 3 of a `node` $< \mathbf{v}_{\mathsf{n}}^{ext}, \mathbf{v}_{\mathsf{n}}^{int}, \mathbf{G}_{\mathsf{n}}, \mathbf{H}_{\mathsf{n}}, \mathsf{G}_{\mathsf{n}}, \mathbf{O}_{\mathsf{n}} >$. The **Constraint** object is a tuple of two expressions with a given comparison operator (either $\leq, =, \geq$). The **Objective** is a type (either min or max) and an expression.

The expressions are kept as **Expression** trees. The leaves are the terms (either an identifier or a number) and the internal nodes are operators. It should be noted that no evaluation is done at this stage and *that every expression, value and name is kept symbolic in an expression tree.* Figure 4 shows the expression tree representation of the expression $a * x + b * (y + z)$.

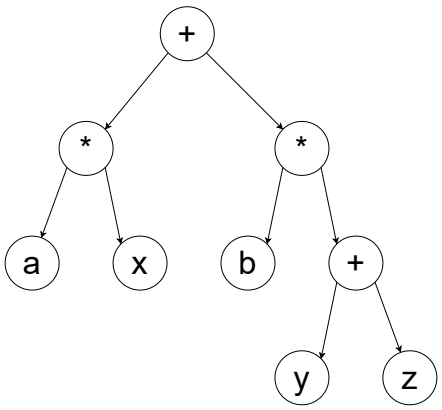


Figure 4. Expression tree of the expression $a * x + b * (y + z)$.

The **Hyperedge** is also made-up of a unique identifier, a reference to its parent, a list of parameters and a list of constraints, again corresponding to the set $< \mathcal{N}_{\mathsf{e}}, \mathbf{G}_{\mathsf{e}}, \mathbf{H}_{\mathsf{e}} >$ defined in Section 3.

Using an acyclic graph for **Graph** and trees make adding information such as nodes, constraints, objectives, hyperedges, variables or parameters very easy as only a reference needs to be added to the tree. The tree then passes a series of tests such as the linearity and definition check which also augments the information contained in the syntax tree, such as putting object references to link relevant information.

4.4. Generating Matrices

Once the semantics of the language have been checked, the third stage is the conversion of the syntax tree to its matrix equivalent as shown in Figure 2. The syntax tree is converted to its matrix equivalent for the full problem. We create a problem instance by evaluating all the expressions with the values of the parameters provided by the user. To be more precise, we build the matrices $\mathbf{O}_{\mathsf{n}}$, $\mathbf{H}_{\mathsf{n}}$ and $\mathbf{G}_{\mathsf{n}}$ for each node n and the matrices $\mathbf{H}_{\mathsf{e}}$ and $\mathbf{G}_{\mathsf{e}}$ for each hyperedge e during *the delayed evaluation.* First, we build the vector $\mathbf{x}$ of all variables in all the nodes. To do so, to each variable in every node we associate a start and an end index corresponding to its absolute index in $\mathbf{x}$.

In order to generate the matrix representation, we have to convert the tree representation of constraints and objectives to the matrices $\mathbf{O}_{\mathsf{n}}$, $\mathbf{H}_{\mathsf{n}}$, $\mathbf{G}_{\mathsf{n}}$, $\mathbf{H}_{\mathsf{e}}$ and $\mathbf{G}_{\mathsf{e}}$ by using the parameters values provided. One must recall that constraints are made-up

of a right-hand-side expression tree, a left-hand-side expression tree and a comparison operator. Often, as we work with time-indexed MILPs, an explicit index or/and a condition are declared for the extension of that constraint but in the following we only assume that the time index t is used as the other indices work in similar ways. First, we factorize constraints in tuples of four elements :

$$\text{Coefficients},\ \text{Variables},\ \text{Comparison Operator},\ \text{Independent Term}$$

By so doing, we decouple the coefficients' vector from the independent term matrix in the matrices $\mathbf{H_n}$, $\mathbf{G_n}$, $\mathbf{H_e}$ and $\mathbf{G_e}$. The coefficients can depend on the time index t, therefore, they are also expression trees. The variables are seen as tuples of the start index associated with the variable and an expression tree corresponding to the offset index. Once the constraints have been factorized, we evaluate the expression trees in the factorization and aggregate all the coefficients - offset - comparison operators and independent terms together to form the matrices.

To illustrate, let us consider the model,

```
#TIMEHORIZON T = 2;

#NODE Example
#PARAMETERS
a = import "profile1.csv";
b = import "profile2.csv";
#VARIABLES
internal: x[T];
internal: y[T];
internal: z;
#CONSTRAINTS
a[t]*x[t]+b[t]*y[t] <= 3;
```

First, absolute indexes are distributed to all the variables. In our case, the absolute indexes of $\mathbf{x}$, $\mathbf{y}$, z are distributed as follows:

variable	x[0]	x[1]	y[0]	y[1]	z
absolute index	0	1	2	3	4

Once the indexes have been fixed, the constraint $a[t] * \mathtt{x}[t] + b[t] * \mathtt{y}[t] <= 3;$ is converted to the tuples

$$\underbrace{[a[t], b[t]]}_{\text{Coefficients}},\ \underbrace{[[0,t],[2,t]]}_{\text{Variables}},\ \underbrace{<=}_{\text{Comparison operator}},\ \underbrace{3}_{\text{Independent term}} \tag{2}$$

One must recall that coefficients and the independent term are expression trees and that the tuples $[0,t]$ and $[2,t]$ respectively represent the variables $\mathtt{x}[t]$ and $\mathtt{y}[t]$ (the first element is the start index, the second the expression tree of the offset). All the expression trees need to be evaluated in order to extend this constraint to its matrix form. Hence, we have to evaluate this factorization for every index t in $[0 : T-1]$. We get,

$$[a[0], b[0]], [[0,0],[2,0]], <=, 3 \text{ for t} = 0$$
$$[a[1], b[1]], [[0,1],[2,1]], <=, 3 \text{ for t} = 1$$

Written in matrix form, we get,

$$\begin{bmatrix} a[0] & 0 & b[0] & 0 & 0 \\ 0 & a[1] & 0 & b[1] & 0 \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ y[0] \\ y[1] \\ z \end{bmatrix} \leq \begin{bmatrix} 3 \\ 3 \end{bmatrix}.$$

Both constraint matrices are kept in the sparse Coordinate List format[47] which represents matrices as a triplets of row, column and value, also called COO. In COO form, we get,

$$\underbrace{[\underbrace{0}_{\text{t=0}},\underbrace{0}_{\text{t=1}},\underbrace{1}_{\text{t=0}},\underbrace{1}_{\text{t=1}}]}_{\text{Rows}},\underbrace{[\underbrace{0}_{\text{t=0}},\underbrace{2}_{\text{t=1}},\underbrace{1}_{\text{t=0}},\underbrace{3}_{\text{t=1}}]}_{\text{Columns}},\underbrace{[\underbrace{a[0]}_{\text{t = 0}},\underbrace{b[0]}_{\text{t = 1}},\underbrace{a[1]}_{\text{t = 0}},\underbrace{b[1]}_{\text{t = 1}}]}_{\text{Coefficients}}, <=, \underbrace{[\underbrace{3}_{\text{t=0}},\underbrace{3}_{\text{t=1}}]}_{\text{Independent terms}}$$

Once all the constraints have been converted to this form, we aggregate them on a node-per-node basis in the COO format by aggregating the coefficients, rows and columns together. Then, we aggregate all the nodes and hyperedges together to create one equality constraint matrix A_1 and one inequality constraint A_2, which take the form shown in Figure 5 for a model with three nodes and two hyperedges with its Node 3 being a hierarchical node composed of three sub-nodes and two sub-hyperedges. Every sub-node can itself be of the same form recursively.

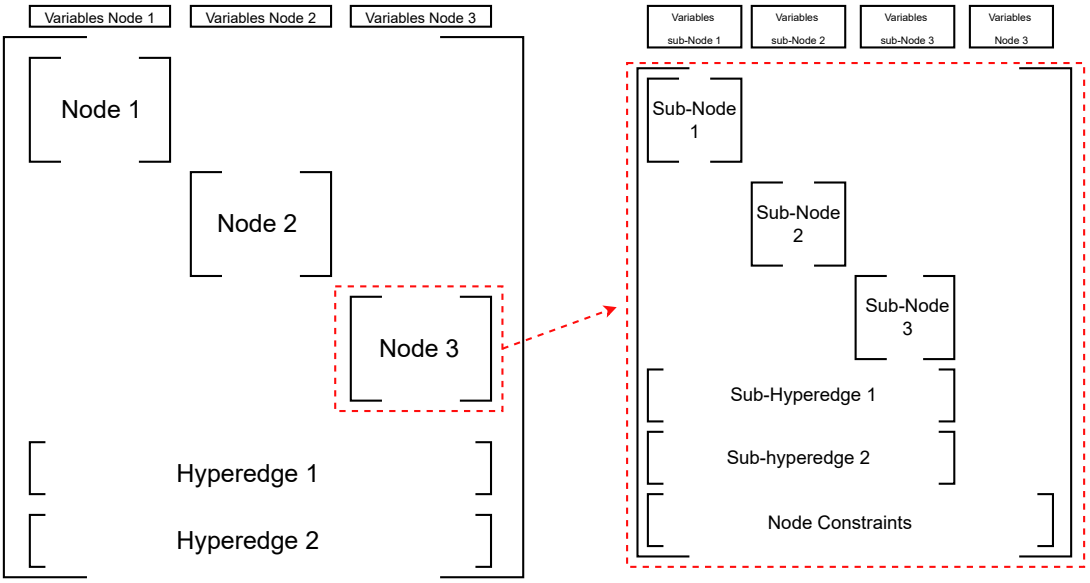


Figure 5. Matrix structure of GBOML.

A similar procedure is used to implement the objective matrix. In order to accelerate the evaluation of syntax trees, we convert all the expressions that need evaluation into Python's syntax trees.

4.5. Solvers and Output

The matrix representation of the previous step is passed to each solver with additional elements from the syntax tree to match its expected input format. This is the fourth stage shown in Figure 2. The solver then solves the problem and sends back the solution, if it exists. The solution can then be printed to CSV or JSON files, the latter also reflecting the hierarchical structure in the original input.

In terms of solvers, GBOML interfaces with the commercial solvers CPLEX[29], Gurobi[25] and Xpress[15, 35] and two open-source alternatives Highs[27] and CLP/CBC[17]. CPLEX, Gurobi, Xpress implement the primal simplex, dual simplex and barrier methods for LPs and branch-and-cut for MILPs. Highs implements the primal simplex and (parallel) dual simplex algorithms as well as an interior point method for LPs and a branch-and-price algorithm for MILPs. CLP and CBC are two related software projects under the COIN-OR umbrella. CLP implements the primal and dual simplex algorithms for LPs, while CBC implements a branch-and-cut algorithm for MILPs.

GBOML also interfaces with Dantzig-Wolfe decomposition via DSP[34]. DSP is a meta-solver which relies on Gurobi, CPLEX and SCIP[5] for implementing Benders Decomposition[3], Dantzig-Wolfe decomposition[13] and Dual decomposition, three structure-exploiting methods. GBOML also interfaces with Benders decomposition via CPLEX. The partitions passed to structure-exploiting algorithms are the ones provided by the user in the input file.

5. Examples

In this section, we provide two examples, one of a unit commitment problem and a more generic investment problem, both defined in GBOML. The first one highlights the reuse functionality of the language. The second one defines a generic investment node that could for instance be used to replace two components in an energy systems planning model.

Unit Commitment

In this example, we consider a manufacturer and some demand, and the goal is to supply the demand while minimizing the overall production cost. First, let us define a generic production machine. The `PRODUCTION_MACHINE` node can, up to a certain capacity, produce a number of goods for a marginal cost per unit produced, knowing that a fixed cost is linked to its use. Let us define this node in a file named `"production_node.txt"`,

```
#NODE PRODUCTION_MACHINE
#PARAMETERS
marginal_cost = 10; // marginal cost per good produced
fixed_cost = 100; // fixed cost
maximum_capacity = 100; // maximum output if turned on
minimum_capacity = 1; // minimum output if turned on
#VARIABLES
external : produced_goods[T];
internal binary: is_used[T];
```

```
#CONSTRAINTS
produced_goods[t] >= minimum_capacity*is_used[t];
produced_goods[t] <= maximum_capacity*is_used[t];
#OBJECTIVES
min: marginal_cost*produced_goods[t]+fixed_cost*is_used[t];
```

Second, let us define some demand in a file named "demand.txt":

```
#NODE DEMAND
#PARAMETERS
demand_profile = import "demand.csv";
#VARIABLES
external : demand[t];
#CONSTRAINTS
demand[t] == demand_profile[t];
```

Let us now consider a case where the manufacturer possesses three machines, named M1, M2 and M3, respectively. Based on their forecast of the demand, the manufacturer wants to know when to start producing with each machine. M1 has a fixed cost of 100 and a marginal cost of 10. M2 has a fixed cost of 200 but a marginal cost of 8. Finally, M3 has a fixed cost of 500 but a marginal cost of 3. We consider that all three machines have the same minimum and maximum capacity. In GBOML, one could write the problem as follows:

```
#TIMEHORIZON T=100;

#NODE Producer
#NODE M1 = import PRODUCTION_MACHINE from "production.txt";
\\ no modification

#NODE M2 = import PRODUCTION_MACHINE from "production.txt" where
marginal_cost = 6;
fixed_cost = 200;

#NODE M3 = import PRODUCTION_MACHINE from "production.txt" where
marginal_cost = 2;
fixed_cost = 500;

#VARIABLES
internal : machine_1_production[T]<-M1.produced_goods[T];
internal : machine_2_production[T]<-M2.produced_goods[T];
internal : machine_3_production[T]<-M3.produced_goods[T];
external : total_production[T];

#CONSTRAINTS
total_production[t] == machine_1_production[t]+
   machine_2_production[t] + machine_3_production[t];

#NODE Demand = import DEMAND from "demand.txt";
demand_profile = import "forecast.csv";

#HYPEREDGE Market
#CONSTRAINTS
Demand.demand[t] <= Producer.total_production[t];
```

Adapting and importing the node PRODUCTION_MACHINE several times is made easy by GBOML reuse feature, and the time horizon is adapted automatically as well. The results of the problem are shown in Appendix A.1.

Generic Node

In this example, we consider a very generic investment node, defined in GBOML in a file named `"generic_investment.txt"` as,

```
#NODE Investment
#PARAMETERS
cost_per_unit_capacity = 600;
max_production_per_unit_installed = import "prod_profile.csv";
cost_per_unit_produced = 10;
max_capacity = 100;
#VARIABLES
internal: capacity;
external: production[T];
#CONSTRAINTS
capacity >= 0;
capacity <= max_capacity;
production[t] >= 0;
production[t] <= max_production_per_unit_installed[t] * capacity
    ;
#OBJECTIVES
min investment: capex * capacity;
min cost_per_timesteps: cost_per_unit_produced*production[t];
```

This `Investment` node can be used in modelling many investment problems and is totally independent of the use case. For example, in Berger et al.[4], this node could be directly used to replace the `WIND` and `PV` plants in the model, as shown in Appendix A.2.

6. Discussion and Benchmarking

As shown in Section 4, GBOML relies on four basic concepts:

- **Delayed Evaluation**: The step of reading the model and the step of evaluating the model are completely separated.
- **Symbolic Representation**: All the information read in a GBOML file is kept symbolically.
- **Structured Semantic Tree**: The data structure supports the definition of structured MILPs.
- **Special Time Index**: GBOML is particularly well-suited for time-indexed MILPs. It makes a strong hypothesis about the shape of the constraints and objectives that are considered.

Some key features of GBOML are **reuse**, **vectorization**, **parallelization** and **interfacing with structure-exploiting methods**. In this section, we discuss these features of GBOML and how they rely on these four basic concepts by illustrating them on a realistic instance of an energy system planning and sizing problem, the so-called remote renewable energy hub planning problem[4] and a problem from the MIPLIB[7]. Second, we propose a benchmark of GBOML with respect to Plasmo[30] and JuMP[36], two Julia AMLs, Pyomo[11], a Python AML, and AMPL, a stand-alone commercial AML on the aforementioned energy planning problem.

6.1. Reuse

Reuse in itself is straightforward in GBOML. The parser reads a given file and builds its symbolic structured semantic tree. As all the information is kept symbolically in this tree without any evaluation, any component of the tree can be copied and reused in another tree. The semantic tree is mostly made up of references to other objects, therefore, reusing an object from a first semantic tree is as easy as copying its reference into another semantic tree. Embedded AMLs are usually unable to do such things as they usually suffer from the imperative nature of the language they are built on.

The language supports such an operation by writing,

```
#NODE new_identifier = import old_identifier from "filename";
```

To illustrate, let us consider the `DEMAND` node in a file named `"file1.txt"`,

```
#TIMEHORIZON T=10;

#NODE DEMAND
#PARAMETERS
demand = import "demand.csv";
#VARIABLES
external: consumption[T];
#CONSTRAINTS
consumption[t] == demand[mod(t, 24)];
```

In GBOML, reuse can simply be done by writing,

```
#TIMEHORIZON T = 24;

#NODE DEMAND = import DEMAND from "file1.txt";
```

in any other GBOML file. The full syntax tree of `"file1.txt"` is cached in memory and the node `DEMAND` is copied into the appropriate location in the importing syntax tree. Having a special time index enables models to automatically adapt to the `T` value that has been defined in the new file as the indices are also kept symbolically. Very general modifications can be performed on the syntax tree by manipulating the Python objects. The main modifications one might want, such as parameter redefinition, are supported by the language. This kind of ease in reuse and automatic adaptation of the horizon is not possible in non-symbolic languages that do not possess a delayed evaluation.

6.2. Vectorization

In this work, vectorization refers to the process of performing an operation on vectors of values in a single instruction rather than several instructions on scalar values inside a loop. More precisely, modern CPUs have larger registers (i.e. larger than the bit representation of floats - int - etc.) on which they are able to perform a single instruction. The idea is therefore to stack several adjacent values in these registers[31]. To illustrate let us consider the following small chunk of code,

```
for i in range(n):
    x[i] = a[i] * b[i]
```

It means that first `a[i]`, `b[i]` and the operator $*$ are loaded with most register space remaining idle and the result is put to another register `x[i]`. This operation

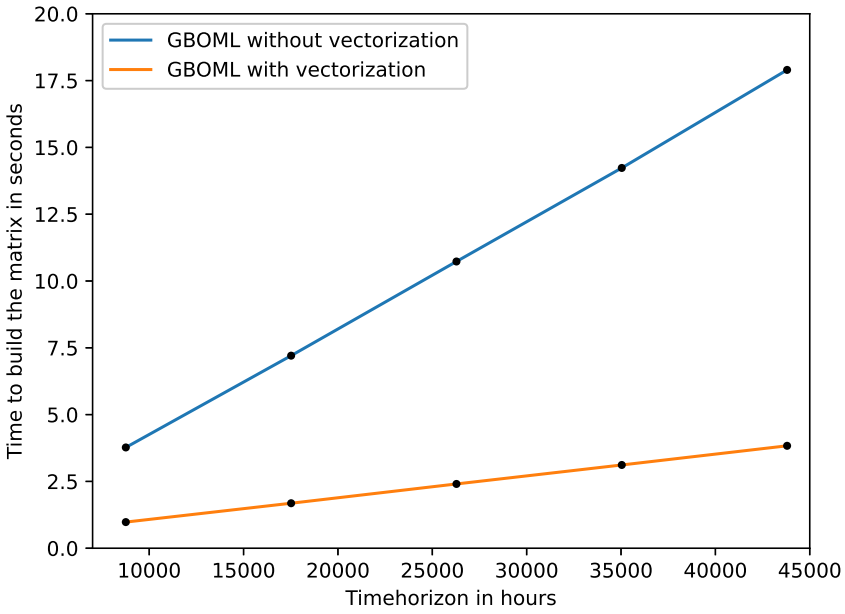


Figure 6. Time needed to generate the model using GBOML with vectorization vs GBOML without vectorization on an instance of the remote renewable energy hub as the time horizon parameter `T` grows.

is repeated n times. The vectorized operation would put several adjacent values of `a` and `b` to fill the register size and compute the corresponding `x` values inside one register.

GBOML is designed to support time-indexed MILPs. Building a matrix for time-indexed MILPs is typically the type of problems where vectorization reduces the computation time. Figure 6 and Table 1 show the time needed to generate the matrix representation of an instance of the remote renewable energy hub from Berger et al[4]. Vectorization is enabled in GBOML thanks to its symbolic representation and is particularly useful to construct time-indexed MILPs.

Table 1. Time taken to generate an instance of the remote hub model with and without vectorization as the time horizon parameter `T` grows, along with the equivalent speedup in percentage.

Time Horizon `T`	8 760	17 520	26 280	35 040	43 800
$\text{GBOML}_{no-vectorization}$	3.771s	7.206s	10.730s	14.231s	17.900s
$\text{GBOML}_{vectorization}$	0.977s	1.683s	2.405s	3.115s	3.832s
Speedup	74.07%	76.63%	77.58%	78.11%	78.58%

6.3. Parallel Model Generation

Parallel model generation has already been discussed and motivated in Andreas and Feng[2]. For very large models, the time to build the matrix representation can be significant (several minutes); minimizing this duration can lead to significant speedups, especially if the model must be generated and solved repeatedly. For example, when performing sensitivity analyses, a model must be run several times with different parameters and sometimes constraints, which typically requires the matrix representation

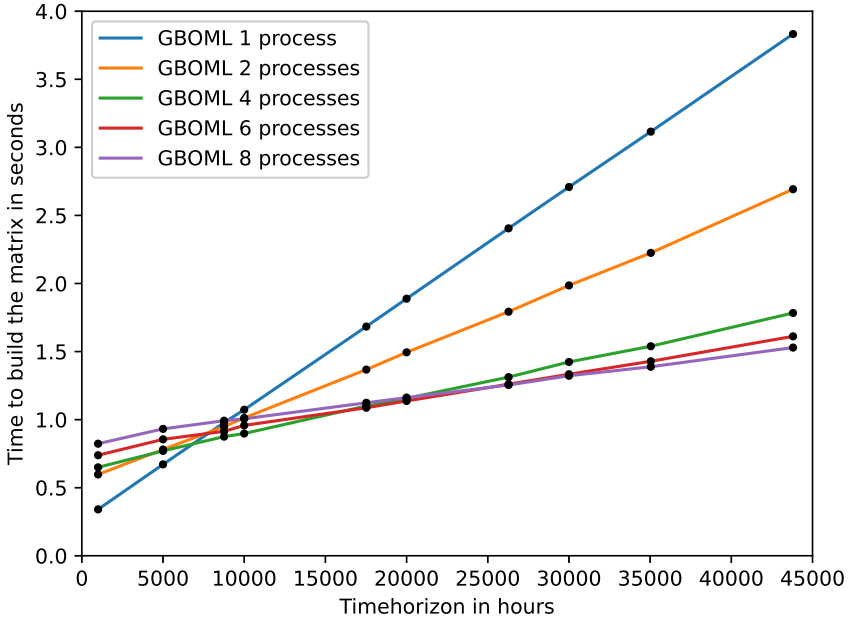


Figure 7. Time to build the model for a given number of processes on an instance of the remote renewable energy hub for a growing time horizon.

to be generated from scratch. GBOML enables the user to build models in parallel thanks to its symbolic representation, the internal structure and its delayed evaluation. Indeed, the constraints and objectives are kept symbolically and need to be converted to their matrix form. On a per-node/per-hyperedge basis, the constraints and objectives are independent from one another, given the variable indexes and parameters values and can therefore be extended in an embarrassingly parallel fashion. This parallelization is typically interesting in order to generate the matrix representation faster, as shown in Figure 7 for the remote renewable energy hub defined in Berger et al.[4]. For smaller models, we can see that there is a fixed cost associated with dealing with a high number of processes. For larger models, on the other hand, the more processes the better as the time to build the model is always reduced by using parallelization.

6.4. Structure-Exploiting Methods

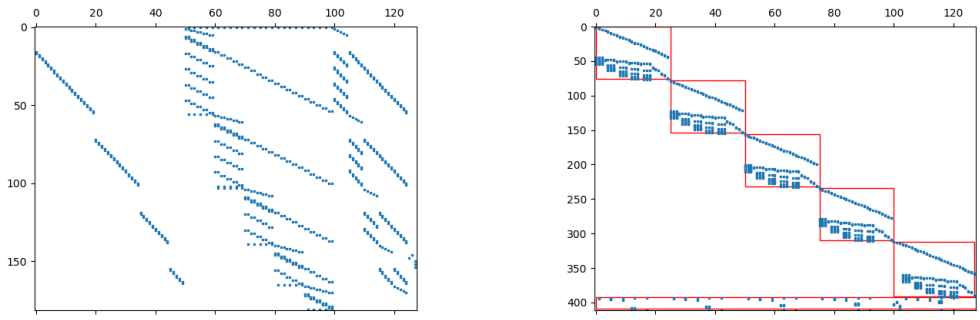
Thanks to the internal representation that captures some structure encoded in the model, GBOML is able to interface with structure-exploiting solvers such as DSP[34] and communicate the encoded structure directly. To illustrate the structure-exploiting methods in GBOML, we consider the MIPLIB noswot problem[7]. The problem possesses a block-decomposable structure that can be exploited by the Dantzig-Wolfe algorithm of DSP. The noswot problem was rewritten in GBOML in a file named `"noswot_gboml.txt"`. It is solved via Gurobi by writing,

```
1 gboml noswot_gboml.txt --gurobi
```

and the solving takes 25.15 seconds. To use DSP's Dantzig-Wolfe algorithm, one can write,

```
gboml noswot_gboml.txt --dsp_dw
```

and the solving takes 2.277 seconds. Figure 8 shows the original and GBOML representation of the constraint matrix of the noswot problem. In GBOML, the noswot problem was defined with five nodes and a hyperedge as shown in Figure 8b, with the structure directly encoded in the GBOML file.



(a) Original Representation

(b) GBOML representation

Figure 8. Constraint matrix of the noswot problem from MIPLIB.

In its current state, GBOML does not perform the partitioning but passes the partition given by the user. As explained in [32], providing access to multiple solvers enables the users to find the one algorithm or implementation that works best for their problem. Allowing users to exploit their knowledge of the topology of a problem and communicate it to specialized solvers could typically enable faster solve times.

6.5. Benchmarking

This section is divided in two. Subsection 6.5.1 provides a detailed explanation of the experimental protocol followed in order to benchmark Pyomo, JuMP, Plasmo, AMPL and GBOML. In Subsection 6.5.2, the results of the benchmarking are provided. The aforementioned tools were benchmarked in terms of Maximum Resident Set Size, time to generate the model and solve time with Gurobi.

6.5.1. Experimental Protocol

We benchmark our tool on an instance of a realistic problem taken from an energy systems application, namely the remote renewable energy hub planning problem analysed by Berger et al.[4] (we consider the version with a weighted average cost of capital of 7%). We implemented this model in each of the five tools. This problem is a structured time-indexed problem whose size grows with the optimization horizon parameter `T`. To be more precise, the number of variables v_n and the number of constraints c_n are given by

$$v_n = 22 + 45 \cdot \mathtt{T} \qquad\qquad c_n \approx 8 + 74 \cdot \mathtt{T}$$

We consider an optimization horizon `T` going from one to five years on an hourly scale (i.e., with $365 * 24 = 8760$ time periods per year). The exact problem size with respect to the time horizon is given in Table 2.

Table 2. Size of instances of the remote hub model, as the time horizon parameter `T` grows.

Time Horizon `T`	8 760	17 520	26 280	35 040	43 800
Number of constraints	648 234	1 296 474	1 944 714	2 592 954	3 241 194
Number of variables	394 222	788 422	1 182 622	1 576 822	1 971 022
Number of non-zeros	1 577 512	3 155 162	4 732 830	6 310 467	7 888 142

We benchmark the five tools on three points:

- Time taken to generate the model: time taken to build a model that can be passed to a solver (without solving it).
- Maximum Resident Set Size (MRSS): the maximum amount of physical memory assigned to the tool at any point in time for generating the model (without solving the model).
- Time taken to solve the model: time taken by the solver to solve the model.

Benchmarking different tools across two programming languages and against stand-alone executable is cumbersome. The goal of our benchmark is to be as fair as possible to the different tools.

For the tools implemented in Julia (Plasmo and Jump), as Julia is precompiled, we followed the guidelines provided by Julia[2]. We first ran the Julia prompt, we then included the file containing the model, and we then timed the function that generates the model once. We repeated this approach ten times and reported the average model generation time on these ten runs for each value of `T`. We also computed the total time taken to generate the model, which is included in our results as the so-called compilation cost. In order to obtain the Maximum Resident Set Size, we used `GNU`'s time command[23] and averaged over ten runs for each `T`. As Julia is precompiled, all the packages included are also precompiled and kept in memory. In order to remove this from the Maximum Resident Set Size (MRSS), we computed the MRSS including only the packages without an instance of the model and obtained a maximum fixed set size of 589 536 kB for JuMP and 591 232 kB for Plasmo over ten runs, which is subtracted from our original estimate.

For Pyomo and GBOML, we used Python's *time* function[42] to benchmark the time taken to generate the model and `GNU`'s *time* command for the MRSS. We repeated the experiment ten times and reported the average for each `T`. We implemented a ConcreteModel for Pyomo without Pyomo parameters.

To the best of our knowledge, AMPL does not provide any feature that allows practitioners to only generate an instance of a model without writing it in a file. Hence, our protocol works as follows. First, AMPL's `times` option is set to 1, and we then read the model and `display` an element of the model, which presumably leads AMPL to generate the full instance. The MRSS is given by `GNU`'s *time* command.

To obtain solve times, we ran Gurobi 9.5.1 on each model and reported its solve time, also averaged over ten runs. The tests were made on a MacBook pro 2021 with 32GB of RAM 10 M1-cores with Python 3.9 and Julia 1.7.2. AMPL interfaces with

[2] `https://docs.julialang.org/en/v1/manual/performance-tips/\#Measure-performance-with-[@time](@ref)-and-pay-attention-to-memory-allocation`

a `mac[rosetta2]` version of Gurobi. Therefore, to enable a fair comparison to other tools that run on `mac[ARM]`, we generated an MPS file from the AMPL model and solved it on the `mac[ARM]` version of Gurobi. The versions of Pyomo, GBOML, Julia, Plasmo and AMPL are respectively `v6.4.1`, `v0.1.1`, `v0.22.3`, `v0.4.3` and `20230228`.

6.5.2. Results

Table 3 and Figure 9 show the time taken by the various tools to generate the model. It should come as no surprise that AMPL is the fastest tool since it is implemented in a low-level, high-performance language. JuMP and GBOML then follow and are virtually tied. Next comes Plasmo and Pyomo is the slowest tool. We are interested in the slopes of the different tools to understand how they evolve when the model grows. AMPL, Pyomo, Plasmo, JuMP and GBOML take, respectively, approximately 0.15s per 8760 time steps, 5s per 8760 time steps, 2s per 8760 time steps, 0.68s per 8760 time steps and 0.7s per 8760 time steps. When parallel model generation is used in GBOML, it always outperforms all other tools except AMPL, even with as few as two processes, both in terms of time taken for the largest models and in terms of slope. In particular, GBOML with 8 processes has a slope of 0.17s per 8760 time steps and closes the gap with AMPL. It should be noted that JuMP and Plasmo pay a fixed start-up cost of 8s and 10s, respectively, which is not negligible in the case of a model that is only run once.

Table 3. Time taken to generate an instance of the remote hub model as the time horizon parameter `T` grows. GBOML_2, GBOML_4 and GBOML_8 respectively stand for GBOML with 2, 4 and 8 processes.

	Time horizon T					
Tool	8 760	17 520	26 280	35 040	43 800	Compilation cost
Plasmo	2.19s	4.06s	6.02s	8.18s	9.80s	$\approx$ 10s
JuMP	0.91s	1.63s	2.31s	3.06s	3.72s	$\approx$ 8s
Pyomo	5.06s	10.04s	15.11s	19.71s	24.70s	0s
AMPL	0.44s	0.60s	0.77s	0.90s	1.06s	0s
GBOML	0.98s	1.69s	2.41s	3.11s	3.83s	0s
GBOML_2	0.95s	1.37s	1.79s	2.22s	2.69	0s
GBOML_4	0.87s	1.10s	1.31s	1.54s	1.78s	0s
GBOML_8	0.91s	1.09s	1.25s	1.43s	1.61s	0s

In terms of Maximum Resident Set Size, GBOML has the overall smallest footprint. For the one-year model, JuMP slightly outperforms GBOML. For larger models, GBOML consistently outperforms the four other tools with JuMP being second for smaller models and AMPL being second for larger ones. The slope of GBOML is the smallest, as can be seen in Figure 10 and Table 4. As Plasmo is an abstraction layer built on top of JuMP, the fact that it takes more memory to implement models should come as no surprise. Pyomo and Plasmo have similar slopes and similar values when it comes to RAM usage. As mentioned before, Plasmo and JuMP pay a fixed MRSS cost respectively estimated at 591 232 kB and 589 536 kB, which is subtracted in this benchmark.

In terms of time taken to solve the model, we solved the remote hub model for a time horizon of one year (8760 time periods) with Gurobi. Solve times are shown in Table 5. The five tools work with different representations of the problem and these do not seem to affect the solver as solve times are very similar. Differences are indeed too small to have any real significance, which is worth stressing, as one wouldn't want the representation to negatively impact solve times. As AMPL uses a different version

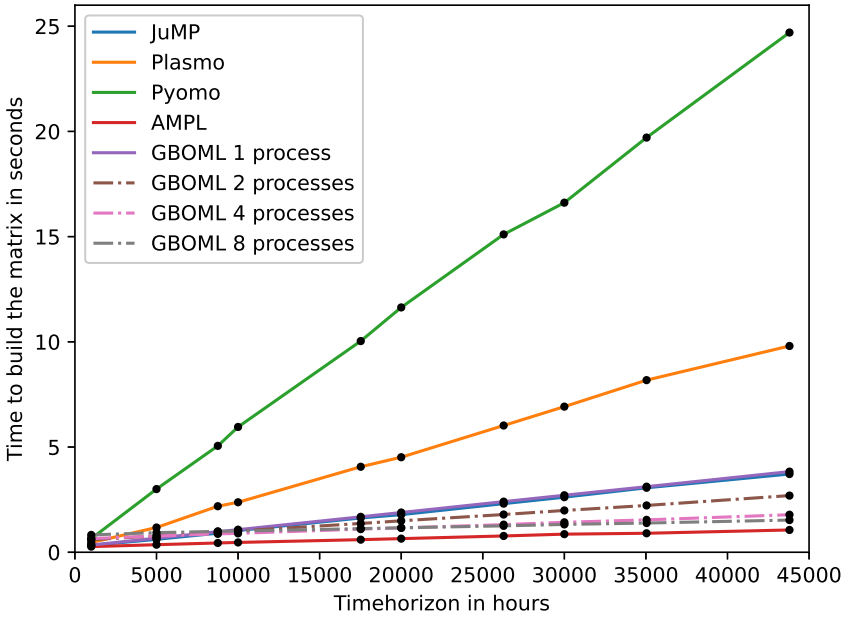


Figure 9. Time taken to generate an instance of the remote hub model with different tools as the time horizon parameter `T` grows, including both sequential and parallel model generation with GBOML.

Table 4. Maximum Resident Set Size (MRSS, in kB) for each tool when generating an instance of the remote hub model as the time horizon parameter `T` grows. The *library footprint* is already removed from the figures given in the table.

	Time horizon T					
Tool	8 760	17 520	26 280	35 040	43 800	Library footprint
Plasmo	304 328	803 987	1 315 387	1 762 550	2 294 387	591 232
JuMP	145 736	360 486	579 785	822 032	1 290 654	589 536
Pyomo	519 571	983 292	1 499 801	1 927 784	2 512 660	neglected
AMPL	223 244	424 110	584 593	828 694	999 060	neglected
GBOML	173 238	288 376	430 643	557 144	677 020	neglected

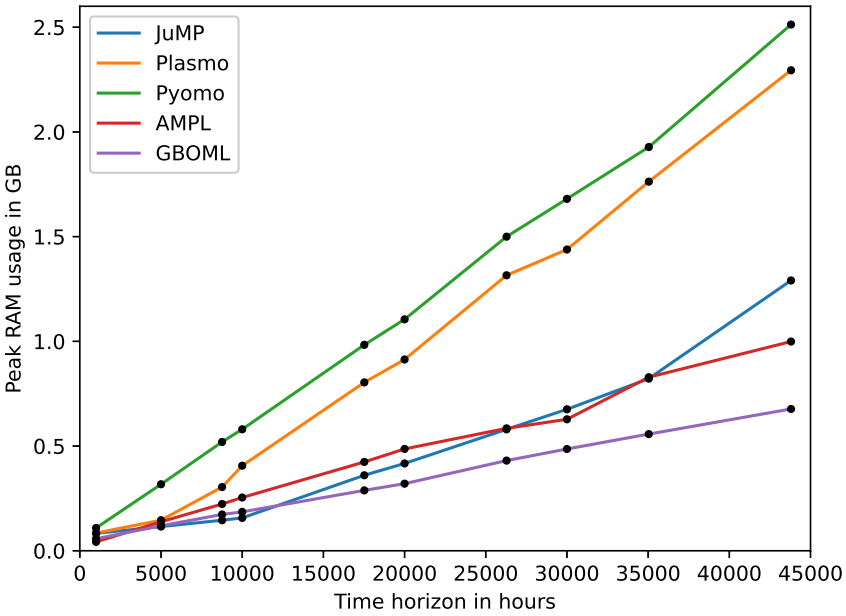


Figure 10. MRSS resulting from the generation of an instance of the remote hub model for all five tools as the time horizon parameter `T` grows. The library memory footprint is subtracted for JuMP and Plasmo.

of Gurobi, it should be noted that we were forced to generate an equivalent MPS of its representation.

Table 5. Time taken to solve an instance of the remote hub model with a one-year horizon generated by each tool (8760 time periods) using Gurobi as solver.

Plasmo	JuMP	Pyomo	AMPL	GBOML
32.698s	33.156 s	30.733s	32,746s	32.286s

In summary, JuMP is ahead of GBOML by the thinnest of margins in terms of time required to generate the model. However, JuMP has a high fixed cost and when parallel model generation is used in GBOML, the latter significantly outperforms former, especially when using a high number of processes (more than twice quicker for the largest model with eight processes). JuMP and GBOML significantly outperform Plasmo and Pyomo with Pyomo being the slowest. In terms of MRSS, GBOML outperforms the four other tools without having any fixed cost and without impacting the solve time. JuMP and AMPL outperform Plasmo and Pyomo when it comes to MRSS. AMPL has a smaller slope than JuMP and outperforms it for larger models. The difference in MRSS performance between Plasmo and Pyomo is difficult to evaluate due to Plasmo's fixed library cost.

7. Conclusion and Future Work

In this paper, we detail the inner workings of GBOML, an open-source modelling language and tool for MILPs implemented in Python. We first provide a full overview

of the design aims of GBOML, which include supporting MILPs with special structures that can be encoded by a hierarchical hypergraph, offering syntax close to standard mathematical notation and facilitating the modular construction, reuse and generation of time-indexed models. Then, we present the four basic concepts on which GBOML relies that enable these features, namely delayed evaluation, symbolic representation, structured semantic trees and the special time index, and discuss their implementation. We provide a unit commitment example and a generic investment model to illustrate the use of the language and tool. We also show that exploiting problem structure can significantly reduce instance generation times as well as solve times for some problems. Specifically, we benchmark the times taken by GBOML, JuMP, Plasmo, Pyomo and AMPL to generate instances of a structured MILP taken from an energy systems planning application. Results show that GBOML outperforms both Plasmo and Pyomo, takes about the same time as JuMP but is slower than AMPL. With parallel model generation, GBOML outperforms JuMP and closes the gap with AMPL. Finally, GBOML has a smaller MRSS than the four other tools. In the future, we would like to focus on the problem of identifying good partitions that can then be exploited by specialised solvers in order to improve solve times.

Acknowledgements

The authors would like to thank Hatim Djelassi for his work on a previous version of GBOML, his feedback and overall guidance and Benoit Legat for taking the time to answer questions about the inner workings of JuMP.

Disclosure statement

The authors report there are no competing interests to declare.

Funding

The authors gratefully acknowledge the support of the Federal Government of Belgium through its Energy Transition Fund and the INTEGRATION project.

References

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] G. Andreas and Q. Feng, Psmg-a parallel structured model generator for mathematical programming, Optimization Online (2014).

[3] J. Benders, Partitioning procedures for solving mixed-variables programming problems., Numerische Mathematik 4 (1962/63), pp. 238–252. Available at `http://eudml.org/doc/131533`.

[4] M. Berger, D. Radu, G. Detienne, T. Deschuyteneer, A. Richel, and D. Ernst, Remote renewable hubs for carbon-neutral synthetic fuel production, Frontiers in Energy Research 9 (2021). Available at `https://doi.org/10.3389\%2Ffenrg.2021.671279`.

[5] K. Bestuzheva, M. Besançon, W.K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S.J. Maher, F. Matter, E. Mühmer, B. Müller, M.E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, and J. Witzig, The SCIP Optimization Suite 8.0, ZIB-Report 21-41, Zuse Institute Berlin, 2021. Available at http://nbn-resolving.de/urn:nbn:de:0297-zib-85309.

[6] J. Bezanson, A. Edelman, S. Karpinski, and V.B. Shah, Julia: A fresh approach to numerical computing, SIAM Review 59 (2017), pp. 65–98.

[7] R.E. Bixby, E.A. Boyd, and R.R. Indovina, MIPLIB: A test set of mixed integer programming problems, SIAM News 25 (1992), p. 16.

[8] F. Boussemart, C. Lecoutre, and C. Piette, XCSP3: an integrated format for benchmarking combinatorial constrained problems, CoRR abs/1611.03398 (2016). Available at http://arxiv.org/abs/1611.03398.

[9] T. Brown, J. Horsch, and D. Schlachtberger, Pypsa: Python for power system analysis, Journal of Open Research Software 6 (2018).

[10] M.R. Bussieck and A. Meeraus, General Algebraic Modeling System (GAMS), Springer US, Boston, MA, 2004, Available at https://doi.org/10.1007/978-1-4613-0215-5_8.

[11] M.L. Bynum, G.A. Hackebeil, W.E. Hart, C.D. Laird, B.L. Nicholson, J.D. Siirola, J.P. Watson, and D.L. Woodruff, Pyomo–optimization modeling in python, 3rd ed., Vol. 67, Springer Science & Business Media, 2021.

[12] E. Castillo, A. Conejo, P. Pedregal, R. García, and N. Alguacil, Building and Solving Mathematical Programming Models in Engineering and Science, Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts, Wiley, 2001, Available at https://books.google.be/books?id=FdhyQgAACAAJ.

[13] G.B. Dantzig and P. Wolfe, Decomposition principle for linear programs, Operations Research 8 (1960), pp. 101–111. Available at https://doi.org/10.1287/opre.8.1.101.

[14] M.C. Ferris, S.P. Dirkse, J.H. Jagla, and A. Meeraus, An extended mathematical programming framework, Computers & Chemical Engineering 33 (2009), pp. 1973–1982. Available at https://www.sciencedirect.com/science/article/pii/S0098135409001653, FOCAPO 2008 – Selected Papers from the Fifth International Conference on Foundations of Computer-Aided Process Operations.

[15] FICO Xpress-Optimizer, Reference manual, http://www.fico.com/xpress. Accessed: 2022-08-30.

[16] C. Floudas and P. Pardalos, Optimization in Computational Chemistry and Molecular Biology: Local and Global Approaches, Vol. 40, Springer New York, NY, 2000 01.

[17] J. Forrest, T. Ralphs, H.G. Santos, S. Vigerske, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, and et al., coin-or/cbc: Release releases/2.10.8 (2022).

[18] R. Fourer, D. Gay, and B. Kernighan, AMPL: A Modeling Language for Mathematical Programming, Scientific Press series, Thomson/Brooks/Cole, 2003, Available at https://books.google.be/books?id=Ij8ZAQAAIAAJ.

[19] R. Fourer, Modeling languages versus matrix generators for linear programming, ACM Trans. Math. Softw. 9 (1983), p. 143–183. Available at https://doi.org/10.1145/357456.357457.

[20] R. Fourer and D.M. Gay, Extending an algebraic modeling language to support constraint programming, INFORMS Journal on Computing 14 (2002), pp. 322–344. Available at https://doi.org/10.1287/ijoc.14.4.322.2825.

[21] A. Frangioni, N. Iardella, and R.D. Lobato, The sms++ project: A structured modelling system for mathematical models (2021). Available at https://smspp.gitlab.io/.

[22] A.M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel, Essence: A constraint language for specifying combinatorial problems, Constraints 13 (2008), p. 268–306. Available at https://doi.org/10.1007/s10601-008-9047-y.

[23] GNU Time, GNU Time (2022). Available at https://www.gnu.org/software/time.

[24] T. Guns, Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example, in Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019), Vol. 19. 2019.

[25] Gurobi Optimization, LLC, Gurobi Optimizer Reference Manual (2022). Available at https://www.gurobi.com.

[26] S. Heipcke and Y. Colombani, Xpress mosel: Modeling and programming features for optimization projects, in Operations Research Proceedings 2019, J.S. Neufeld, U. Buscher, R. Lasch, D. Möst, and J. Schönberger, eds., Operations Research Proceedings, Springer, 2020, pp. 677–683.

[27] Q. Huangfu and J.A.J. Hall, Parallelizing the dual revised simplex method, Mathematical Programming Computation 10 (2018), pp. 119–142.

[28] M. Hästbacka, J. Westerlund, and T. Westerlund, Mispt: a user friendly milp mixed-time based production planning tool, in 17th European Symposium on Computer Aided Process Engineering, V. Pleşu and P. Şerban Agachi, eds., Computer Aided Chemical Engineering Vol. 24, Elsevier, 2007, pp. 637–642. Available at https://www.sciencedirect.com/science/article/pii/S1570794607801295.

[29] IBM ILOG Cplex, V12. 1: User's manual for cplex, International Business Machines Corporation 46 (2009).

[30] J. Jalving, S. Shin, and V.M. Zavala, A graph-based modeling abstraction for optimization: Concepts and implementation in plasmo.jl (2020).

[31] J. Jeffers, J. Reinders, and A. Sodani, Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition, 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[32] J. Kallrath, Modeling Languages in Mathematical Optimization, 1st ed., no. 978-1-4613-0215-5 in Applied Optimization, Springer, 2004, Available at https://link.springer.com/book/10.1007/978-1-4613-0215-5.

[33] J. Kallrath, Business Optimization Using Mathematical Programming, no. 978-3-030-73237-0 in International Series in Operations Research and Management Science, Springer, 2021 June, Available at https://ideas.repec.org/b/spr/isorms/978-3-030-73237-0.html.

[34] K. Kibaek, Z. Victor, T. Christian, Z. Yingqiu, B. Geunyeong, and N. Hideaki, Dsp (2022). Available at https://github.com/Argonne-National-Laboratory/DSP, Accessed: 2022-08-12.

[35] R. Laundy, M. Perregaard, G. Tavares, H. Tipi, and A. Vazacopoulos, Solving hard mixed-integer programming problems with xpress-mp: A miplib 2003 case study, INFORMS Journal on Computing 21 (2009), pp. 304–313.

[36] M. Lubin and I. Dunning, Computing in operations research using julia, INFORMS Journal on Computing 27 (2015), pp. 238–248.

[37] B. Miftari, M. Berger, H. Djelassi, and D. Ernst, Gboml: Graph-based optimization modeling language, Journal of Open Source Software 7 (2022), p. 4158. Available at https://doi.org/10.21105/joss.04158.

[38] Modelica Association, Modelica - a unified object-oriented language for physical systems modeling. Tutorial (2000). Available at http://www.modelica.org/documents/ModelicaTutorial14.pdf.

[39] L. Moretti, M. Milani, G.G. Lozza, and G. Manzolini, A detailed milp formulation for the optimal design of advanced biofuel supply chains, Renewable Energy 171 (2021), pp. 159–175. Available at https://www.sciencedirect.com/science/article/pii/S0960148121002111.

[40] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack, MiniZinc: Towards a Standard CP Modelling Language, in Principles and Practice of Constraint Programming – CP 2007, C. Bessière, ed., Berlin, Heidelberg. Springer Berlin Heidelberg, 2007, pp. 529–543.

[41] S. Pfenninger and B. Pickering, Calliope: A multi-scale energy systems modelling framework, Journal of Open Source Software 3 (2018), p. 825.

[42] Python Software Foundation, The Python Standard Library (2022). Available at https://docs.python.org/3/library/time.html.
[43] Python Software Fundation, PyPI (2022). Available at https://pypi.org/.
[44] Simulink Documentation, Simulation and model-based design (2020). Available at https://www.mathworks.com/products/simulink.html.
[45] J.M.G. Sánchez, Modelling in Mathematical Programming, Springer Cham, 2020.
[46] The Open Source Initiative, MIT License (2022). Available at https://opensource.org/licenses/MIT.
[47] The SciPy community, COO format in Scipy (2022). Available at https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html.
[48] C. Valente, G. Mitra, M. Sadki, and R. Fourer, Extending algebraic modelling languages for stochastic programming, INFORMS Journal on Computing 21 (2009), pp. 107–122. Available at https://doi.org/10.1287/ijoc.1080.0282.
[49] G. Van Rossum and F.L. Drake, Python 3 Reference Manual, CreateSpace, Scotts Valley, CA, 2009.
[50] S. Wang and Q. Meng, Robust bunker management for liner shipping networks, European Journal of Operational Research 243 (2015), pp. 789–797. Available at https://www.sciencedirect.com/science/article/pii/S0377221714010674.

Appendix A. Complementary Information on Examples

In this appendix, we provide complementary information concerning the two examples given in Section 5. In appendix A.1, we provide the results of the unit commitment example. In appendix A.1, we show how the generic nature of the `Investment` node can be used to replace two nodes from a practical example.

A.1. Example: Unit commitment

In this appendix, we provide the results of the unit commitment example showed in Section 5. This example is trivial. In Figure A1, we plot the cost of production for each machine, as the demand grows. We can see the results in the lower subplot. With the demand between 1 and 24, M1 should be used. From 24 to 74, using M2 minimizes the cost. And from 74 to 100, using M3 minimizes the cost.

A.2. Example: Generic Investment Node

In this appendix, we show how the generic nature of the `Investment` node can be used to replace two nodes from a practical example. In Berger et al.[4], the `WIND` and `PV` nodes are defined as,

```
#NODE SOLAR_PV_PLANTS
#PARAMETERS
    full_capex = 380.0;
    lifetime = 25.0;
    annualised_capex = full_capex * global.wacc * (1 + global.wacc)**lifetime / ((1 + global.wacc)**lifetime - 1); // MEur
    fom = 7.25; // MEur/year
    vom = 0.0;
    capacity_factor_PV = import "pv_capacity_factors.csv"; // Dimensionless
    max_capacity = 500.0; // GW
```

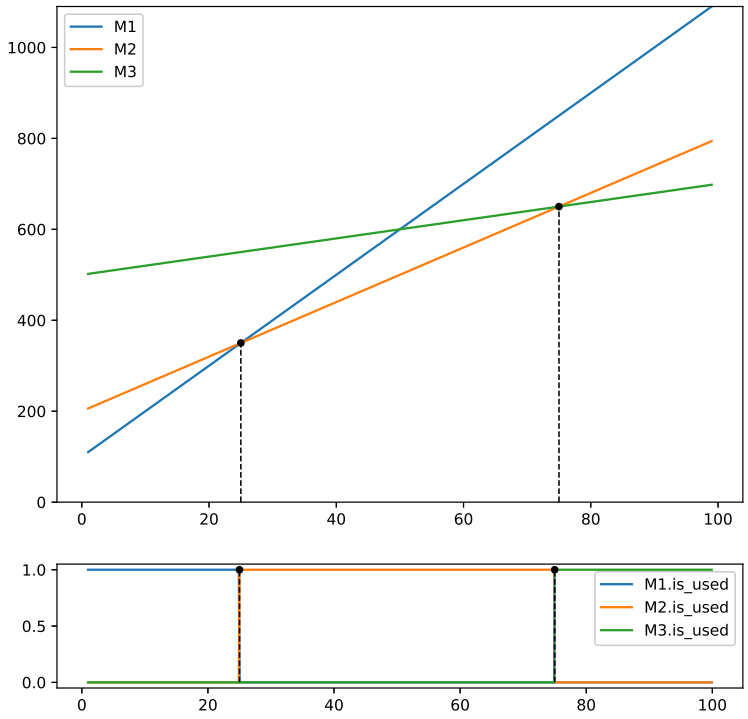


Figure A1. (Top) The evolution of the price of production for each machine with respect to the demand. (Bottom) Value of the variable `is_used` for each machine with respect to the demand.

```
#VARIABLES
    internal: capacity;
    external: electricity[T];
#CONSTRAINTS
    electricity[t] <= capacity_factor_PV[t] * capacity;
    capacity <= max_capacity;
    capacity >= 0;
    electricity[t] >= 0;
#OBJECTIVES
    min: global.number_years_horizon * (annualised_capex + fom)
   * capacity;
    min: vom * electricity[t];

#NODE WIND_PLANTS
#PARAMETERS
    full_capex = 1040.0;
    lifetime = 30.0;
    annualised_capex = full_capex * global.wacc * (1 + global.
   wacc)**lifetime / ((1 + global.wacc)**lifetime - 1); // MEur
    fom = 12.6; // MEur/year
    vom = 0.00135; // MEur/GWh
    capacity_factor_wind = import "wind_capacity_factors.csv";
   // Dimensionless
    max_capacity = 500.0; // GW
#VARIABLES
    internal: capacity;
    external: electricity[T];
#CONSTRAINTS
    electricity[t] <= capacity_factor_wind[t] * capacity;
```

```
    capacity <= max_capacity;
    capacity >= 0;
    electricity[t] >= 0;
#OBJECTIVES
    min: global.number_years_horizon * (annualised_capex + fom)
   * capacity;
    min: vom * electricity[t];
```

Both nodes could be rewritten using the `Investment` node as follows,

```
#NODE SOLAR_PV_PLANTS = import Investment from "
   generic_investment.txt" where
cost_per_unit_produced = 0.0; // Value of vom
cost_per_unit_capacity = global.number_years_horizon * (380*
   global.wacc*(1+global.wacc)**25/((1 + global.wacc)**25 - 1)
   +7.25);
max_capacity = 500.0;
max_production_per_unit = import "pv_capacity_factors.csv";


#NODE WIND_PLANTS = import Investment from "generic_investment.
   txt" where
cost_per_unit_produced = 0.00135; // Value of vom
cost_per_unit_capacity = global.number_years_horizon * (1040*
   global.wacc*(1+global.wacc)**30/((1 + global.wacc)**30 - 1)
   +12.6);
max_capacity = 500;
max_production_per_unit = import "wind_capacity_factors.csv";
```