Mathematical Programming Computation
https://doi.org/10.1007/s12532-026-00315-4

FULL LENGTH PAPER

Graph-Based modeling and decomposition of hierarchical optimization problems

David L. Cole[1] · Filippo Pecci[2,3] · Omar J. Guerra[4] · Harsha Gangammanavar[5] · Jesse D. Jenkins[6,7] · Victor M. Zavala[1,8]

Received: 5 January 2025 / Accepted: 19 March 2026
© The Author(s) 2026

Abstract

We present a graph-theoretic modeling approach for hierarchical optimization that leverages the OptiGraph abstraction implemented in the `Julia` package `Plasmo.jl`. We show that the abstraction is flexible and can effectively capture complex hierarchical connectivity that arises from decision-making over multiple spatial and temporal scales (e.g., integration of planning, scheduling, and operations in manufacturing and infrastructures). We also show that the graph abstraction facilitates the conceptualization and implementation of decomposition and approximation schemes. Specifically, we propose a graph-based Benders decomposition (gBD) framework that enables the exploitation of hierarchical (nested) structures and that uses graph aggregation/partitioning procedures to discover such structures. In addition, we provide a `Julia` implementation of gBD, which we call `PlasmoBenders.jl`. We illustrate the capabilities using examples arising in the context of energy and power systems.

This material is based on work supported by the U.S. Department of Energy under grant DE-0002722. We also acknowledge support from the National Science Foundation under award CBET-2328160. F.P also received support from the Eric & Wendy Schmidt Fund for Strategic Innovation (grant G-23-64740) and the Princeton Zero-carbon Technology Consortium, supported by unrestricted gifts from Google and Breakthrough Energy.

✉ Victor Zavala
vzavala@anl.gov

1 Department of Chemical and Biological Engineering, University of Wisconsin-Madison, Madison, WI 53706, USA

2 CMCC Foundation - Euro-Mediterranean Center on Climate Change, Lecce, Italy

3 RFF-CMCC European Institute on Economics and the Environment, Milan, Italy

4 Grid Planning and Analysis Center, National Renewable Energy Laboratory, Golden, CO 80401, USA

5 Operations Research and Engineering Management, Southern Methodist University, Dallas, TX 75205, USA

6 Andlinger Center for Energy and Environment, Princeton University, Princeton, NJ 08540, USA

7 Department of Mechanical and Aerospace Engineering, Princeton University, Princeton, NJ 08540, USA

8 Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439, USA

Published online: 25 April 2026

Keywords Optimization · Graph Theory · Hierarchical · Multi-Scale · Decomposition · Julia

Mathematics Subject Classification 90C06 · 68R10

1 Introduction

Hierarchical optimization problems often involve connectivity across different decision levels and scales (e.g., physical assets of a system are coupled across different temporal or spatial scales) [5, 13]. These types of problems are also often high-dimensional and their solution requires approximation schemes (e.g., aggregation/reduction schemes) or decomposition schemes (e.g., Benders or Lagrangian decomposition) [25, 30, 68]. Hierarchical optimization problems often arise from the integration of decisions in supply chains, planning, scheduling, and operation/control levels in industrial systems such as energy systems [21, 41, 45, 53, 92], chemical processes [1, 10, 97], agricultural systems [3, 62], water infrastructure networks [72], and healthcare systems [47].

To handle the complexity of hierarchical optimization problems, it is necessary to navigate and exploit their unique *connectivity structure*. For example, Benders Decomposition (BD) [45, 73, 77, 100], Lagrangian Decomposition (LD) [60, 100], and the Alternating Direct Method of Multipliers (ADMM) [21, 52, 87] can be used to decompose and exploit the structure of these problems. Other approaches involve computing approximate solutions by solving hierarchical layers (or components of each layer) in a sequential manner [39, 63]; this approximation strategy is used, for instance, for handling integrated unit commitment and economic dispatch problems in power systems [2, 6, 56] and for solving dynamic programs over long horizons [88]. However, to our knowledge, there is currently no modeling framework for expressing hierarchical connectivity in a unified manner. As a result, approximation/decomposition approaches are often problem-specific, and successful applications typically rely on specialized domain knowledge.

In this work, we present a general, graph-theoretic modeling approach for optimization that facilitates the navigation, discovery, and manipulation of hierarchical structures. This approach is based on the OptiGraph abstraction implemented in the `Julia` package `Plasmo.jl` [46]. Under this abstraction, the structure of an optimization problem is expressed as a graph in terms of nodes and (hyper)edges. Nodes embed optimization problems with their own variables, constraints, objective functions, and data; hyperedges contain linking constraints that couple variables of optimization problems from different nodes. Importantly, the node of an OptiGraph can be considered as embeding another OptiGraph, thus enabling the modeling of hierarchical structures.

The OptiGraph representation is abstract and can be applied to problems arising in distinct *application domains*. The OptiGraph abstraction is also agnostic to the meaning of *its modeling elements*; for example, a node can capture either a spatial domain (e.g., a geographical region), a temporal domain (e.g., a day or month), a probabilistic scenario (e.g., for stochastic programs), or a physical asset (e.g., a battery

of a power plant). Furthermore, graph structures can be manipulated via aggregation and partitioning, which can facilitate the visualization/discovery of structures and the re-casting of problem structure in a way that is amenable to tailored solution algorithms. For instance, an algorithm that requires an acyclic graph structure (e.g., nested BD requires a tree structure) could potentially be applied to a general graph by eliminating cycles via node aggregation.

We illustrate how the graph abstraction can be used to enable the implementation of approximation and decomposition schemes. Specifically, we introduce a graph-based Benders decomposition scheme (gBD) that generalizes a nested Benders decomposition algorithm [14, 75, 76] to general graph structures. In doing so, we are extending the work of Brunaud [22] who originally proposed using `Plasmo.jl`'s OptiGraph abstraction as a basis for BD. We also show how graph operations facilitate the application of BD to a variety of problem classes and structures. For example, nested BD (also known as dual dynamic programming) is traditionally applied to decompose problems over a time domain, but the graph structure enables its use in spatial and spatio-temporal domains (e.g., infrastructure networks). We also introduce a `Julia` implementation of the gBD scheme that we call `PlasmoBenders.jl`.

Several applications of graph-based modeling frameworks have been reported in the literature. For example, the work of Daoutidis and co-workers represents optimization problems as graphs [4, 32, 67, 69, 70, 90, 91] but considers a fine level of granularity where individual variables and constraints are represented as nodes/edges. Mitrai and Daoutidis have used this representation together with a network analysis approach for detecting hierarchical structures [68]. The OptiGraph abstraction presented here works at a different level of granularity, in which nodes and edges represent entire optimization subproblems. This level of representation is also similar to that of Marcucci and co-workers [65, 66, 71] in which they use nodes to represent optimization problems and edges to represent linking constraints. They solve a mixed integer program to determine a subgraph of the original graph (e.g., a path between two nodes) and solve the resulting subgraph for control applications. Their representation of the original graph could be effectively captured by the OptiGraph abstraction.

This work advances and distinguishes itself from the existing literature in several ways. i) While building on the OptiGraph abstraction introduced in [46], we formalize the mathematical notation for modeling with hierarchical graphs and manipulating their graph structure. ii) We formalize the BD algorithm for the graph-based problem (gBD). While Brunaud [22] originally introduced Benders decomposition on the nodes of `Plasmo.jl`, they did not formalize the algorithm or discuss the implications of generalizing the algorithm to the graph structure. iii) We introduce an open-source software tool for implementing gBD. While Brunaud [22] initially developed a software tool for BD with `Plasmo.jl`, this tool is not maintained. Further, it relies on an implementation of `Plasmo.jl` which is more node-centric (every node is an independent modeling object in `Julia`), whereas `Plasmo.jl` under more recent versions has been made subgraph-centric (every subgraph is an independent modeling object in `Julia`). The latter approach is more memory efficient and follows the mathematical formalization. In addition, `PlasmoBenders.jl` also differs from the `SDDP.jl` [34] package. This latter package is focused on stochastic problem instances—with special functionality for working with probabilities and scenarios—and is built more

directly on `JuMP.jl`, whereas `PlasmoBenders.jl` uses the OptiGraph abstraction and is more general in its modeling and partitioning capabilities. iv) We present many important implications of generalizing the hierarchical optimization problems to the graph structure. These include highlighting how approximation schemes can be implemented and tested (see Case Study 1) as well as how the graph structure can be flexibly exploited to improve or enhance algorithm performance. v) Finally, the implementation of gBD as outlined in this paper provides a framework for defining structure-exploiting algorithms to graph-based problems in a flexible and agnostic way. This framework could be extended to other decomposition approaches (e.g., ADMM).

The paper is structured as follows. Section 2 shows how the OptiGraph abstraction can be used to model hierarchical optimization problems and provides a tutorial overview of `Plasmo.jl` functionalities. Section 3 shows how approximation and decomposition schemes can be applied to graph structures. Section 4 provides challenging case studies, including a tri-level energy market problem showing how graphs can capture hierarchical coupling and can be used to obtain approximate solutions, as well as a capacity expansion model and a production cost model for power systems that are solved using gBD. Section 5 provides conclusions and discusses directions of future work.

2 Hierarchical optimization

Hierarchical optimization problems can be defined in several different ways [5, 9, 20, 44]. For our purposes, a hierarchical optimization problem exhibits a *single* (centralized/composite) objective function, and its constraint set has a nested algebraic structure [44]. This is not to be confused with multi-level optimization problems, where lower-level optimization problems (with their own objective) are embedded in the constraint set of a higher-level problem [20]. Multi-level problems can also be expressed as graphs, but they require specialized solution approaches (e.g., replacing low-level optimization problems with their optimality conditions) that are not considered here and that are left as future work.

2.1 OptiGraph abstraction

The OptiGraph abstraction was introduced by Jalving and co-workers [46], and we follow their notation and definitions and expand on them for representing hierarchical problems. An OptiGraph is a hierarchical hypergraph containing sets of OptiNodes $\mathcal{N}$ and OptiEdges $\mathcal{E}$. An OptiNode is a modeling object that embeds an optimization problem with its own objective, variables, constraints, and data; an OptiEdge is a modeling object that embeds constraints that link variables of multiple OptiNodes (thus generating connectivity). The OptiEdges can connect *more* than two nodes (they are hyperedges); for simplicity, we refer to hyperedges as edges. Importantly, an *OptiNode can also be thought of as embedding an entire OptiGraph*, thus providing the ability to represent hierarchical structures.

When solving the full problem, the individual objectives of the OptiNodes are combined into a single scalar composite objective function (e.g., via weighted summation). For simplicity, we will refer to OptiNodes, OptiEdges, and OptiGraphs as nodes, edges, and graphs (or subgraphs in the hierarchical case) when it is clear from context that we are referencing the OptiGraph abstraction. An OptiGraph for an optimization problem is denoted as $\mathcal{G}$ with the set of nodes and edges denoted as $\mathcal{N}(\mathcal{G})$ and $\mathcal{E}(\mathcal{G})$, respectively. We also denote edges incident to node n as $\mathcal{E}(n)$ and denote nodes connected by edge e as $\mathcal{N}(e)$.

An optimization problem modeled under the OptiGraph abstraction can be expressed in the following abstract form:

$$\min_{\{\boldsymbol{x}_n\}_{n\in\mathcal{N}(\mathcal{G})}} \; f\left(\{\boldsymbol{x}_n\}_{n\in\mathcal{N}(\mathcal{G})}\right) \qquad \text{(Objective)} \qquad (1a)$$

$$\text{s.t. } \boldsymbol{x}_n \in \mathcal{X}_n, \quad n \in \mathcal{N}(\mathcal{G}), \qquad \text{(Node Constraints)} \qquad (1b)$$

$$g_e(\{\boldsymbol{x}_n\}_{n\in\mathcal{N}(e)}) \geq 0, \quad e \in \mathcal{E}(\mathcal{G}). \qquad \text{(Link Constraints)} \qquad (1c)$$

Here, $\{\boldsymbol{x}_n\}_{n\in\mathcal{N}(\mathcal{G})}$ is the set of decision variables over all the nodes in the graph (i.e., $\boldsymbol{x}_n$ is the set of variables of node n). The constraints on the node n are represented by the set $\mathcal{X}_n$. The objective function (1a) is a scalar composite function of any variables contained on nodes in the graph. In (1), there is no restriction on variables being continuous or discrete or on objectives or constraints being linear or nonlinear; as such, from a modeling standpoint, the OptiGraph abstraction can express structures of linear programs (LPs), mixed-integer linear programs (MILPs), nonlinear programs (NLPs), and mixed-integer nonlinear programs (MINLPs). Furthermore, the linking constraints in (1c) are written as inequalities, but the abstraction is not limited to inequalities, and the software implementation in `Plasmo.jl` supports both equality and inequality constraints on the edges. In this work, we will study how the graph abstraction facilitates the development of algorithms for the solution of LPs and MILPs; however, the abstraction can also potentially be used to implement algorithms for solving nonlinear problems.

Figure 1 provides an example of the OptiGraph structure, where the optimization problem shown on the left is modeled as an OptiGraph, with variables x_1, x_2, and x_3 placed on nodes n_1, n_2, and n_3, respectively. In this case, the objective functions are separable and can be placed in graph nodes. It is not always the case that the objective function is separable among the nodes, and the OptiGraph supports non-separable objective functions. This distinction between a node objective and a graph objective is primarily important for the software implementation (which will be discussed later), as the node objectives are not part of the mathematical definition of (1).

The OptiGraph abstraction can be applied to optimization problems in a flexible manner. For instance, the coarsest graph representation of an optimization problem is obtained by placing all variables/constraints on a single node ($|\mathcal{N}(\mathcal{G})| = 1$ and thus $\mathcal{E}(\mathcal{G}) = \emptyset$), while the finest graph representation is obtained placing every variable on a separate node. A wide variety of intermediate graph structures can be obtained by partitioning/placing variables and constraints to different nodes. In building the graph, the number and placement of edges are determined automatically based on the placement of decision variables into the nodes of the graph, which determines

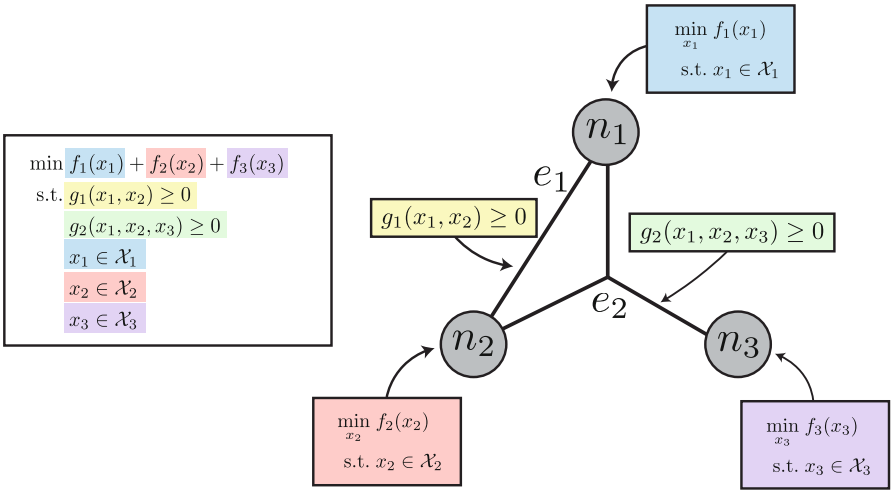


Fig. 1 Visualization of an OptiGraph where an optimization problem (left) is partitioned into the nodes and edges of a graph (right). Nodes contain their own objective, variables, constraints, and data, while edges contain constraints that link variables across nodes

whether a constraint becomes a nodal constraint, (1b), or a linking constraint (1c). Often, there may be a natural graph structure that a user may want to use (e.g., arising from physical, spatial, or temporal connectivity), but in some cases the graph structure might need to be derived from advanced partitioning or aggregation procedures [50, 83].

The OptiGraph abstraction also supports hierarchical structures. For instance, any node in an OptiGraph can be considered as embedding an OptiGraph, or one can encounter the situation in which an OptiGraph is composed of multiple OptiGraphs (subgraphs) that are linked together. In this way, nodes can belong to subgraphs that in turn belong to a higher-level OptiGraph. Here, a subgraph is defined as a subset of nodes and a subset of edges of the original OptiGraph, such that all nodes are a subset of the nodes of the original OptiGraph, and the subset of edges contains only edges connecting nodes in the subgraph. Each subgraph could likewise be further partitioned into additional subgraphs, and there are no limitations to the number of embedded subgraphs possible within an OptiGraph.

We now formalize the notation and terminology that we will use for representing hierarchy and layering. The graph containing all nested subgraphs for a problem of interest will be given by $\mathcal{G}$. We will use the term "level" to refer to how "deep" a subgraph is in a graph (i.e., how many times subgraphs are nested within other graphs). Subgraphs will be annotated by indices on $\mathcal{G}$, and the number of indices will indicate the number of levels below the main OptiGraph $\mathcal{G}$. For example, first-level graphs are denoted as $\mathcal{G}_i, i \in \{1, ..., N\}$, where N is the number of subgraphs contained in $\mathcal{G}$. Since each subgraph $\mathcal{G}_i, i \in \{1, ..., N\}$ can likewise contain nested subgraphs, these additional subgraphs are annotated by the addition of a second index on $\mathcal{G}_i$, such as $\mathcal{G}_{i,j}, i \in \{1, ..., N\}, j \in \{1, ..., N_i\}$, where N_i is the number of subgraphs contained

in $\mathcal{G}_i$. Indices can be added for additional levels with no limit on the number of levels possible.

We will refer to the number of indices as the "depth" of subgraphs in the graph (e.g., $\mathcal{G}_i$ is at a depth of one and $\mathcal{G}_{i,j}$ is at a depth of two). For any subgraph on any level, we annotate the graph as

$$\mathcal{G}_\circ(\{\mathcal{G}_{\circ,i}\}_{i\in\{1,\ldots,N_\circ\}}, \mathcal{N}_{\mathcal{G}_\circ}, \mathcal{E}_{\mathcal{G}_\circ}), \tag{2}$$

where $\circ$ represents any set of indices mapping the graph, $N_\circ$ is the number of subgraphs contained on $\mathcal{G}_\circ$, $\{\mathcal{G}_{\circ,i}\}_{i\in\{1,\ldots,N_\circ\}}$ represents the set of subgraphs on $\mathcal{G}_\circ$, and $\mathcal{N}_{\mathcal{G}_\circ}$ and $\mathcal{E}_{\mathcal{G}_\circ}$ are nodes and edges contained in $\mathcal{G}_\circ$ but independent of its subgraphs $\{\mathcal{G}_{\circ,i}\}_{i\in\{1,\ldots,N_\circ\}}$. We also define notation $\mathcal{SG}(\cdot)$ such that

$$\mathcal{SG}(\mathcal{G}_\circ) := \{\mathcal{G}_{\circ,i}\}_{i\in\{1,\ldots,N_\circ\}} \tag{3}$$

is the set of subgraphs of a graph $\mathcal{G}_\circ$ on the level directly below $\circ$ (i.e., this does not directly include subgraphs nested on $\mathcal{G}_{\circ,i}$, $i \in \{1, \ldots, N_\circ\}$). Note also that subgraphs may have different numbers of nested subgraphs on them at any level; for instance, it is possible that $|\mathcal{SG}(\mathcal{G}_{\circ,i})| \neq |\mathcal{SG}(\mathcal{G}_{\circ,j})|$ for some $i \neq j$. For a graph $\mathcal{G}_\circ$ containing subgraphs, the notation $\mathcal{N}(\mathcal{G}_\circ)$ and $\mathcal{E}(\mathcal{G}_\circ)$ represents the sets of all nodes and edges, respectively, on graph $\mathcal{G}_\circ$ *including* the nodes and edges of the subgraphs. In other words, $\mathcal{N}(\mathcal{G}_\circ) = \mathcal{N}_{\mathcal{G}_\circ} \cup \{\mathcal{N}(\mathcal{G}_{\circ,i})\}_{i\in\{1,\ldots,N_\circ\}}$ and $\mathcal{E}(\mathcal{G}_\circ) = \mathcal{E}_{\mathcal{G}_\circ} \cup \{\mathcal{E}(\mathcal{G}_{\circ,i})\}_{i\in\{1,\ldots,N_\circ\}}$.

Figure 2 provides an example of an OptiGraph with a hierarchical structure and illustrates the notation used. In this example, the primary graph $\mathcal{G}(\{\mathcal{G}_1, \mathcal{G}_2\}, \emptyset, \mathcal{E}_\mathcal{G})$ (in black) contains subgraphs $\mathcal{G}_1(\{\mathcal{G}_{1,1}, \mathcal{G}_{1,2}\}, \emptyset, \mathcal{E}_{\mathcal{G}_1})$ and $\mathcal{G}_2(\{\mathcal{G}_{2,1}, \mathcal{G}_{2,2}, \mathcal{G}_{2,3}, \mathcal{G}_{2,4}\}, \emptyset, \mathcal{E}_{\mathcal{G}_2})$ (in red). These subgraphs also contain nested subgraphs (each outlined in blue); $\mathcal{G}_1$ contains two subgraphs ($\mathcal{G}_{1,1}$ and $\mathcal{G}_{1,2}$) while $\mathcal{G}_2$ contains four subgraphs ($\mathcal{G}_{2,1}$, $\mathcal{G}_{2,2}$, $\mathcal{G}_{2,3}$, and $\mathcal{G}_{2,4}$). Here, the mid-level (red) subgraphs use only one index for identification, while the lowest-level (blue) subgraphs use two indices, where the first index corresponds to the mid-level subgraph to which they belong. Each subgraph also contains a specific set of edges, which are colored in the figure based on which subgraph the edges belong. For example, the set $\mathcal{E}_\mathcal{G}$ contains the four black edges drawn from $\mathcal{G}_{1,1}$ to $\mathcal{G}_{2,1}$ and to $\mathcal{G}_{2,2}$ and from $\mathcal{G}_{1,2}$ to $\mathcal{G}_{2,3}$ and to $\mathcal{G}_{2,4}$. Likewise, $\mathcal{G}_1$ and $\mathcal{G}_2$ contain sets of edges highlighted in red (e.g., $\mathcal{E}_{\mathcal{G}_1}$ includes the red edges drawn between $\mathcal{G}_{1,1}$ and $\mathcal{G}_{1,2}$). In this case, all nodes are contained within the lowest level (blue) subgraphs, but it would be possible for there to be nodes contained within a higher level subgraph.

Because OptiGraphs can have a nested structure, they are a flexible tool for modeling hierarchical problems. There are at least two ways in which this can be done. The OptiGraph can have an inherent hierarchy based on the nested layers in the graph. For instance, in Figure 2, one could consider the graphs $\mathcal{G}_{2,1}$, $\mathcal{G}_{2,2}$, $\mathcal{G}_{2,3}$, and $\mathcal{G}_{2,4}$ as separate systems in a chemical process that are connected together. Each individual graph may contain variables and objectives corresponding to the system it represents. The mid-level graph $\mathcal{G}_2$ would then capture the higher-level connections between the separate systems, creating a hierarchy between $\mathcal{G}_2$ and its subgraphs. Another approach to capturing hierarchy in the OptiGraph abstraction is by ordering subgraphs on the same

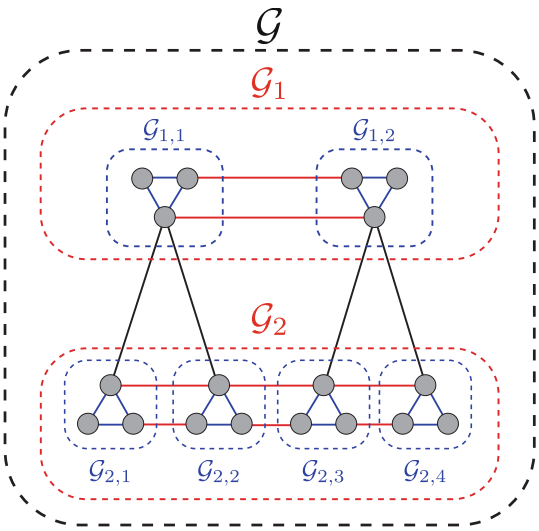


Fig. 2 Example of how OptiGraphs can capture hierarchical connectivity. Here, the red graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are subgraphs of the black $\mathcal{G}$, highlighted by the dashed borders. Both $\mathcal{G}_1$ and $\mathcal{G}_2$ contain a set of lower-level (blue) subgraphs

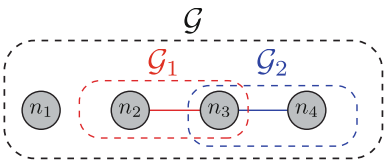


Fig. 3 Example of the hierarchical graph $\mathcal{G}(\{\mathcal{G}_1, \mathcal{G}_2\}, \{n_1\}, \emptyset)$, showing how a node can belong to multiple subgraphs, creating overlapping regions in the subgraphs

level. In Figure 2, this can be thought of as $\mathcal{G}_1$ and $\mathcal{G}_2$ representing separate hierarchical layers (even though they are on the same hierarchical level of $\mathcal{G}$). For instance, $\mathcal{G}_1$ could represent a coarse-resolution planning layer while $\mathcal{G}_2$ could represent a finer-resolution operations layer; such a structure arises frequently in applications featuring sequential strategic and tactical decisions (e.g., planning or scheduling followed by operations) [45, 63, 100].

There is no restriction that nodes/edges belong to a single graph. For instance, Figure 3 shows the graph $\mathcal{G}(\{\mathcal{G}_1, \mathcal{G}_2\}, \{n_1\}, \emptyset)$ where node n_3 belongs to *both* $\mathcal{G}_1$ and $\mathcal{G}_2$, creating overlapping subgraphs. Many problems will not have overlap, but the ability to express such overlap can have algorithmic implications (e.g., implementation of Schwarz decomposition algorithms [46, 85, 86]).

2.2 Software implementation in `Plasmo.jl`

The OptiGraph abstraction is implemented in the open-source `Julia` package `Plasmo.jl`, and this provides a user-friendly interface for constructing problems. `Plasmo.jl` leverages the modeling capabilities of `JuMP.jl` [64] for expressing node and graph subproblems. An OptiGraph is instantiated using the `Plasmo.OptiGraph()` function. Nodes are added to an OptiGraph by call-

ing the `Plasmo.@optinode` macro, which takes as arguments the OptiGraph that the node will belong to and (optionally) a name for the node. Nodes operate similarly to `JuMP.jl model` objects and can have variables, constraints, expressions, and objectives added to them using the `JuMP.jl` macros, `@variable`, `@constraint`, and `@objective`, respectively. Edges are created in the graph by calling `Plasmo.@linkconstraint`. Unlike the `@constraint` macro, which requires a `model` or `OptiNode` object as an argument, the `Plasmo.@linkconstraint` function requires an `OptiGraph` object as an argument since the edge will be "owned" by the graph rather than by a single node.

To illustrate the basic hierarchical modeling capabilities of `Plamso.jl`, we provide a small modeling example and present the code for assembling this problem. We consider a chemical process that converts a raw material X into a product Y. The price for Y fluctuates at different times, and only so much of Y can be produced or sold at a given time. This problem will consider the sizing of a storage/inventory system that can hold Y to be sold at the time periods of higher price. There is a cost associated with building the storage system based on its size, and there is a cost associated with buying raw material X. The problem will seek to maximize total profit and is written as:

$$\min \alpha \cdot s_{size} + \sum_{t=1}^{T} \left(\beta_t \cdot x_t^{buy} - \gamma_t \cdot y_t^{sell} \right) \tag{4a}$$

$$y_{t+1}^{stored} - y_t^{stored} = y_t^{save}, t = 1, ..., T-1 \tag{4b}$$

$$y_t^{save} + y_t^{sell} - \zeta x_t^{buy} = 0, t = 1, ..., T \tag{4c}$$

$$0 \leq y_t^{stored} \leq s_{size}, t = 1, ..., T \tag{4d}$$

$$0 \leq y_t^{sell} \leq \overline{d}^{sell}, t = 1, ..., T \tag{4e}$$

$$\underline{d}^{save} \leq y_t^{save} \leq \overline{d}^{save}, t = 1, ..., T \tag{4f}$$

$$y_1^{stored} = \bar{y}^{stored} \tag{4g}$$

where s_{size} is a decision variable of the maximum storage size, x_t^{buy} is the amount of raw material purchased, y_t^{save} is the amount of product sent to storage (can be negative if it is being removed from storage), y_t^{sell} is the amount of product sold, and y_t^{stored} is the amount of product in storage. The parameters α and β are costs associated with their respective variables, γ is the price of product Y, and ζ is a conversion factor of X to Y. The parameters $\overline{d}^{sell}$, $\underline{d}^{save}$, and $\overline{d}^{save}$ are upper or lower bounds on their respective variables, and $\bar{y}^{stored}$ is the initial amount in storage. Constraint (4b) is a mass balance on the storage, constraint (4c) is a mass balance on product Y, and constraint (4d) ensures that the storage cannot exceed s_{size}. This is a hierarchical problem in the sense that there is a planning/design decision in choosing storage system size and several operations decisions in choosing when to buy X or store/sell Y.

The problem in (4) can be modeled in `Plasmo.jl` using Code Snippet 1 for $T = 20$ time points. To build this problem, we will place the planning/design variable (s_{size}) on its own node, and then link this node to a set of T operations nodes containing the

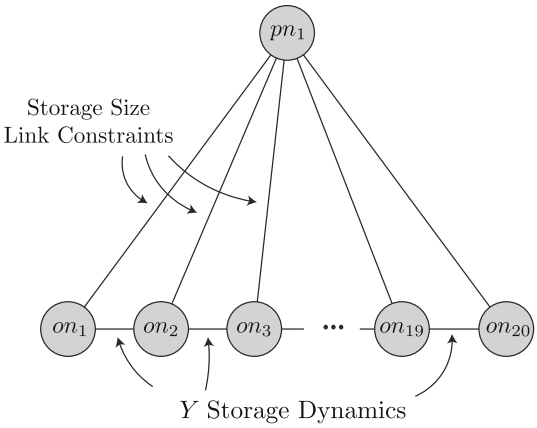


Fig. 4 Hierarchical graph structure that can be used to represent the storage problem (4) implemented in Code Snippet 1. Planning variables are placed on the top/root node pn_1, while operations variables at each time point in $T = 20$ are embedded within the children nodes on_t. The variables of the operation nodes are linked together because storage dynamics are coupled in time

operations variables x_t^{buy}, y_t^{save}, y_t^{sell}, and y_t^{stored}. The constraints (4d) and (4b) will be captured by linking constraints. Problem data is set on Lines 5 - 8. The OptiGraph is then instantiated (Line 11), a node is created for the planning variables (Line 12), and T nodes are created for the operations variables (Line 13). We then set a variable and objective on the planning node (Lines 16 and 17). We next loop through the operations nodes and add variables, a constraint, and an objective on each node (Lines 20 - 28). The initial storage value can then be set (Line 31) and constraint (4b) is enforced by adding linking constraints between each operations node (Lines 34 - 35). The planning level decision is enforced by linking constraints between the planning node and each operation node (Line 38). In this code snippet, we have set objectives directly on the nodes since our objective is separable between nodes; to set the graph objective, we will call the function `set_to_node_objectives` (Line 41), which sums each of the node objectives together to get a scalar composite objective. Finally, we set the optimizer and call `optimize!` on the graph (Lines 44 and 45). The final graph created by this modeling process is shown in Figure 4.

With an understanding of basic `Plasmo.jl` syntax, we will next show how hierarchical subgraphs can be constructed within `Plasmo.jl`, as shown in Code Snippet 2. This code creates the hierarchical graph structure shown in Figure 2. Since the example in Figure 2 is not representing a specific optimization problem, the values in this code snippet in various constraints are arbitrary, and the code snippet is intended to create a problem with a structure like that of the figure, not to represent a specific case study. For simplicity, we first define a function called `build_graph` (Lines 4 - 30) to create a basic graph structure with three nodes with edges between each node (this is the structure presented by each of the blue subgraphs shown in Figure 2). The subgraphs $\mathcal{G}$, $\mathcal{G}_1$, and $\mathcal{G}_2$ from Figure 2 are next defined as OptiGraphs named `g`, `g1`, and `g2`, respectively (Line 33). We then build $\mathcal{G}_{1,1}$, $\mathcal{G}_{1,2}$, $\mathcal{G}_{2,1}$, $\mathcal{G}_{2,2}$, $\mathcal{G}_{2,3}$, and $\mathcal{G}_{2,4}$ from Figure 2 by calling the `build_graph` function (Lines 36 - 38). Each subgraph is now created, but no hierarchical structure has been formed. Importantly, *no edges can yet be added between subgraphs since edges must be "owned" by a higher level graph*. We first add `g11` and `g12` to the higher-level OptiGraph `g1` and then add `g21`, `g22`, `g23`, and `g24` to the higher-level OptiGraph `g2` (Lines 41 - 43). Edges can now

Code Snippet 1 Code for generating the graph of storage problem (4) in `Plasmo.jl`.

```
# Load in packages
using Plasmo, HiGHS

# Set Problem data
T = 20
gamma = fill(5, T); beta = fill(20, T); alpha = 10; zeta = 2
gamma[8:10] .= 20; gamma[16:20] .= 50
d_sell = 50; d_save = 20; d_buy = 15; y_bar = 10

# Instantiate graph and nodes
graph = OptiGraph()
@optinode(graph, planning_node)
@optinode(graph, operation_nodes[1:T])

# Define planning node data which includes storage size
@variable(planning_node, storage_size >= 0)
@objective(planning_node, Min, storage_size * alpha)

# Loop through operations nodes and set variables, constraint, and objective
for (j, node) in enumerate(operation_nodes)
    @variable(node, 0 <= y_stored)
    @variable(node, 0 <= y_sell <= d_sell)
    @variable(node, -d_save <= y_save <= d_save)
    @variable(node, 0 <= x_buy <= d_buy)

    @constraint(node, y_save + y_sell - zeta * x_buy == 0)
    @objective(node, Min, x_buy * beta[j] - y_sell * gamma[j])
end

# Set initial storage level
@constraint(operation_nodes[1], operation_nodes[1][:y_stored] ==
     y_bar)

# Set mass balance on storage unit
@linkconstraint(graph, [i = 1:(T - 1)], operation_nodes[i + 1][:
     y_stored] -
                       operation_nodes[i][:y_stored] ==
                            operation_nodes[i + 1][:y_save])

# Link planning decision to operations decisiosn
@linkconstraint(graph, [i = 1:T], operation_nodes[i][:y_stored] <=
     planning_node[:storage_size])

# Set graph objective
set_to_node_objectives(graph)

# Optimizer the graph
set_optimizer(graph, HiGHS.Optimizer)
optimize!(graph)
```

be placed between the subgraphs `g11` and `g12` and between `g21`, `g22`, `g23`, and `g24` (Lines 47 - 57), represented by the red colored edges shown in Figure 2. Finally, `g1` and `g2` are added to the main OptiGraph `g` (Line 60) and edges are placed between the nodes and subgraphs contained on `g1` and `g2` (Lines 62 - 65), represented by the black colored edges shown in Figure 2. Since these edges link the mid-level subgraphs `g1` and `g2`, they must be owned by the highest-level graph `g`.

Code Snippet 2 Code for generating the graph shown in Figure 2 with one high-level subgraph, two mid-level subgraphs, and six low-level subgraphs.

```
using Plasmo

# Define function for creating subgraphs with 3 nodes
function build_graph()
    # Define OptiGraph
    g = OptiGraph()

    # Add nodes
    @optinode(g, nodes[1:3])

    # Set variables on nodes
    @variable(nodes[1], x >= 0)
    @variable(nodes[2], x >= 0)
    @variable(nodes[3], x >= 0)

    # Set objective on nodes
    @objective(nodes[1], Min, 1 * nodes[1][:x])
    @objective(nodes[2], Min, 2 * nodes[2][:x])
    @objective(nodes[3], Min, 3 * nodes[3][:x])

    # Define link constraints between nodes
    @linkconstraint(g, nodes[1][:x] + nodes[2][:x] >= 1)
    @linkconstraint(g, nodes[1][:x] + nodes[3][:x] >= 1)
    @linkconstraint(g, nodes[2][:x] + nodes[3][:x] >= 1)

    set_to_node_objectives(g)

    # Return graph
    return g
end

# Define high−level and mid−level OptiGraphs
g = OptiGraph(); g1 = OptiGraph(); g2 = OptiGraph();

# Define lower−level OptiGraphs that contain nodes and edges
g11 = build_graph(); g12 = build_graph();
g21 = build_graph(); g22 = build_graph();
g23 = build_graph(); g24 = build_graph();

# Add lower−level subgraphs to mid−level
add_subgraph!(g1, g11); add_subgraph!(g1, g12);
add_subgraph!(g2, g21); add_subgraph!(g2, g22);
add_subgraph!(g2, g23); add_subgraph!(g2, g24);
```

```

# Define link constraints between lower−level subgraphs;
# these constraints are "owned" by the mid−level graphs
@linkconstraint(g1, g11[:nodes][1][:x] + g12[:nodes][1][:x] >= 1)
@linkconstraint(g1, g11[:nodes][3][:x] + g12[:nodes][2][:x] >= 1)

@linkconstraint(g2, g21[:nodes][1][:x] + g22[:nodes][1][:x] >= 1)
@linkconstraint(g2, g21[:nodes][3][:x] + g22[:nodes][1][:x] >= 1)

@linkconstraint(g2, g22[:nodes][1][:x] + g23[:nodes][1][:x] >= 1)
@linkconstraint(g2, g22[:nodes][3][:x] + g23[:nodes][1][:x] >= 1)

@linkconstraint(g2, g23[:nodes][1][:x] + g24[:nodes][1][:x] >= 1)
@linkconstraint(g2, g23[:nodes][3][:x] + g24[:nodes][1][:x] >= 1)

# Add mid−level subgraphs to high−level OptiGraph
add_subgraph!(g, g1); add_subgraph!(g, g2);

@linkconstraint(g, g11[:nodes][1][:x] + g21[:nodes][1][:x] >= 1)
@linkconstraint(g, g11[:nodes][1][:x] + g22[:nodes][1][:x] >= 1)
@linkconstraint(g, g12[:nodes][1][:x] + g23[:nodes][1][:x] >= 1)
@linkconstraint(g, g12[:nodes][1][:x] + g24[:nodes][1][:x] >= 1)

set_to_node_objectives(g)
set_to_node_objectives(g1)
set_to_node_objectives(g2)
```

An important feature of the OptiGraphs is that they can be treated as independent optimization problems. OptiGraphs can also be passed to `JuMP.jl`'s `optimize!` function, independent of other OptiGraphs. For instance, `g1` can be optimized independent of `g` or `g2`. Further, the optimization problem on `g1` would consist of all variables and constraints contained on the six nodes of `g11` and `g12` as well as the constraints stored on the red edges connecting the two subgraphs (Figure 2). Similarly, `g11` and `g12` could likewise be optimized independently of `g1` by calling `optimize!(g11, Ipopt.Optimizer)` and `optimize!(g12, Ipopt.Optimizer)`; however, the constraints on the red edges would not be enforced by these `optimize!` calls since these constraints are only enforced on `g1` (or `g` if `g` was optimized). This also means that any OptiGraph being optimized must have a solver set on them (e.g., calling `set_optimizer(g1, Ipopt.Optimizer)` does not set a solver on `g11` or `g12`). These capabilities highlight the high degree of flexibility that is available to optimize different graph components using different solvers.

It is important to note some syntax in Code Snippet 2. First, nodes, like variables, can be indexed by an array. Line 9 shows how `@optinode(g, nodes[1:3])` forms 3 OptiNodes, with indices 1, 2, and 3, which can be called directly (e.g., Lines 12 - 14). Variables on nodes can be accessed by calling the node and then the symbol for the variable (e.g., `nodes[1][:x]` for variable `x`). Similar to the symbol notation used to access variables on nodes, nodes on a graph can be accessed in a similar way (e.g., Line 47). For instance, the variable `x` on `nodes[1]` of `g11` can be accessed by calling `g11[:nodes][1][:x]`. In addition, there may be times where subgraphs or nodes do not have a global identifier defined. The API functions `local_subgraphs` and `local_nodes` can be used to find the subgraphs or nodes contained directly on an OptiGraph, while `all_subgraphs` and `all_nodes` return *all* subgraphs or nodes contained on an Optigraph and all its subgraphs. For example `local_subgraphs(g)` will return a vector containing `g1` and `g2` and `local_nodes(g)` will return an empty vector (since no nodes have defined directly on `g`). The function `all_subgraphs(g)` will instead return a vector of all subgraphs, including `g1`, `g2`, `g11`, `g12`, `g21`, `g22`, `g23`, and `g24`, while `all_nodes(g)` will return every node, including those in the lower-level subgraphs.

2.3 Graph structure manipulation

The OptiGraph provides flexibility for manipulating graph structure, including apply partitioning and aggregation schemes. These can facilitate visualization (e.g., plotting an aggregated graph with fewer nodes may be significantly easier than the original graph) and can help apply algorithms that are suitable for specific graph structures. The ideas below were first introduced in [46], but we formalize these ideas for the hierarchical setting, and we show later how they can be used for exploring and decomposing hierarchical structures. We will introduce the concepts mathematically here, and details on the software implementation are contained in the Supporting Information.

Problem structure can be manipulated by creating induced subgraphs on an existing graph, where an "induced subgraph" is one in which a subset of edges contains all edges of the original graph that connect nodes of the subset (an edge of the induced subgraph cannot connect nodes not contained in the subset) [19]. Here, we assume that a subgraph can be created from an existing graph $\mathscr{G}_\circ$ to obtain a graph with nested subgraphs that we denote as $\mathscr{G}_\circ(\{\mathscr{G}_{\circ,i}\}_{i\in 1,\ldots,N_\circ}, \mathscr{N}_{\mathscr{G}_\circ}, \mathscr{E}_{\mathscr{G}_\circ})$.

Partitioning is a common type of graph manipulation that involves placing graph nodes into subgraphs [12]. For an initial graph $\mathscr{G}_\circ(\emptyset, \mathscr{N}, \mathscr{E})$, partitioning involves splitting the node set $\mathscr{N}$ into N_{part} subsets; we define the node subsets as $\mathscr{N}_i^{part} \subseteq \mathscr{N}, i \in \{1, 2, ..., N_{part}\}$ and this satisfies $\mathscr{N}_i^{part} \cap \mathscr{N}_j^{part} = \emptyset,\ i, j \in \{1, 2, ..., N_{part}\}, i \neq j$ and $\bigcup_{i\in\{1,2,\ldots,N_{part}\}} \mathscr{N}_i^{part} = \mathscr{N}$. Each node subset is used to induce a subgraph, which allow us to express the original graph as $\mathscr{G}_\circ(\{\mathscr{G}_{\circ,i}\}\{1, \ldots, N_{part}\}, \emptyset, \mathscr{E}_{\mathscr{G}_\circ})$; here, $\mathscr{E}_{\mathscr{G}_\circ}$ is the set of all edges in $\mathscr{E}$ not contained in the subgraphs.

Aggregation is another common type of graph manipulation; we define the initial graph $\mathscr{G}_\circ(\emptyset, \mathscr{N}, \mathscr{E})$ and a set of nodes to be aggregated (denoted as $\mathscr{N}_{agg} \subseteq \mathscr{N}$). The edges connecting the aggregated nodes are given by $\mathscr{E}_{agg} = \{e : e \in \mathscr{E}, n \in \mathscr{N}_{agg},\ n \in \mathscr{N}(e)\}$. Aggregation reduces nodes in the subset $\mathscr{N}_{agg}$ into a single node, which we denote as n_{agg}. The graph after aggregation is given by $\mathscr{G}_\circ^{agg}(\emptyset, \bar{\mathscr{N}}, \bar{\mathscr{E}})$ where $\bar{\mathscr{N}} = \{n_{agg} \cup \mathscr{N}\} \setminus \mathscr{N}_{agg}$ is the set of nodes in the aggregated graph and $\bar{\mathscr{E}}$ is the set of edges in the aggregated graph. The new node n_{agg} will contain all variables and constraints of the nodes in $\mathscr{N}_{agg}$, and the objective on n_{agg} will be a summation of all objectives on nodes in $\mathscr{N}_{agg}$. All constraints contained on $\mathscr{E}_{agg}$ will also now be stored as local constraints on n_{agg}. Note that this aggregation approach does not change the optimization problem as all variables/constraints still exist (and the graph can contain its same objective function); instead, only the structure of the graph is changed.

The graph manipulation approaches are visualized in Figure 5. Here, we show how the same graph can be manipulated by inducing subgraphs and partitions or by aggregating sets of nodes. By inducing subgraphs or adding partitions, we are creating a structure through the subgraphs without changing any of the nodes. In contrast, aggregating nodes changes the nodal representation of the graph without using hierarchical subgraphs. For instance, in the figure, subgraph formation and aggregation are applied on the same set of nodes, but aggregation contains no hierarchical graphs. Instead the variables, constraints, and objectives contained in nodes n_3, n_4, and n_6, as well as the linking constraints on the edges between these nodes (i.e., the red $\mathscr{G}_1$ subgraph in the "Subgraph Formation" section of the figure), would be contained on the new node n_{agg}. In all cases, the overall optimization problem has remained the same, but the structure for representing that problem is altered. These structure manipulations result in graphs that could have different characteristics than the original graph before inducing the subgraphs or aggregating nodes. These approaches can be useful for creating graphs with different characteristics; for instance, if a graph contains a single cycle, all nodes in the cycle could be aggregated into one single node, resulting in an acyclic graph.

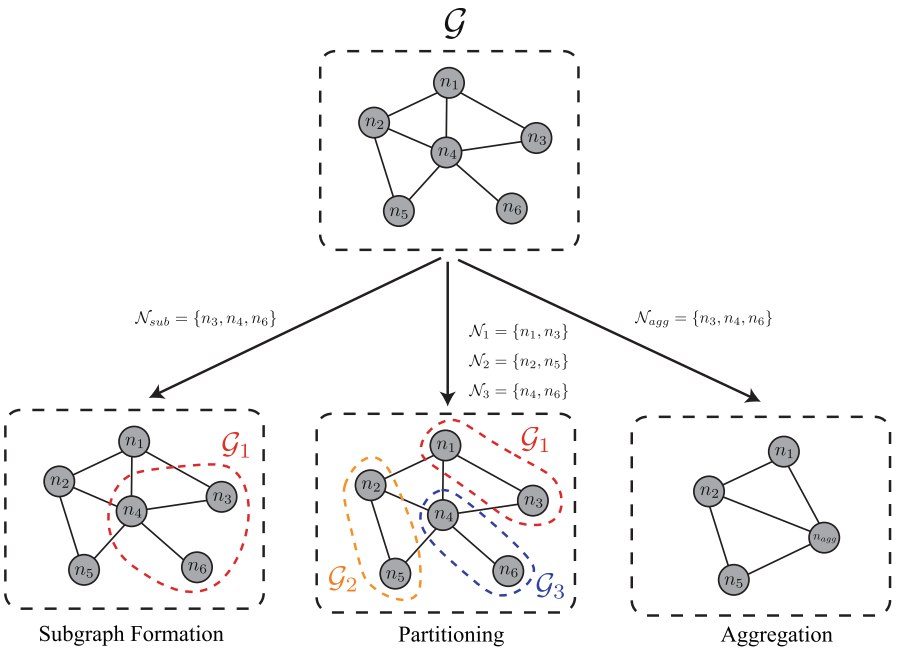


Fig. 5 Examples of how one can generate a subgraph from a graph $\mathscr{G}$ using the node set $\mathscr{N}_{sub}$, partitioning a graph $\mathscr{G}$ with three partitions using the node sets $\mathscr{N}_1$, $\mathscr{N}_2$, and $\mathscr{N}_3$, and aggregating nodes in $\mathscr{N}_{agg}$ of a graph $\mathscr{G}$ into a single new node. Each of these are manipulations of the graph or of its hierarchical structure

The ability to manipulate structure has several applications. These are useful from a modeling perspective because they can simplify the formation of subproblems. For instance, in Sections 4.2 and 4.3, a graph is modeled with no subgraphs and then partitioned into subproblem subgraphs for applying the graph-based Benders Decomposition. In 4.2, this ability allows for exploring different formations of the master/root problem, while in 4.3 it is used to form subproblems of differing temporal sizes. Both of these approaches have important implications for run time and convergence of the algorithms. In addition, these structural manipulations can allow for visualizing different structures (e.g., visualizing temporal or spatial partitionings in a graph).

3 Graph-Based decomposition

Optimization problems with hierarchical structure can often be decomposed to enable scalable solutions or the derivation of approximate solutions. The OptiGraph abstraction can be used as a basis for defining such solution schemes and can simplify a practitioner's use of such solution algorithms. In the discussion that follows, we introduce a graph-based Benders Decomposition (gBD), which is a generalization of the nested Benders Decomposition (BD) [54, 55, 75, 76, 98, 102] to handle general hierarchical graph structures. We show how manipulations of the OptiGraph

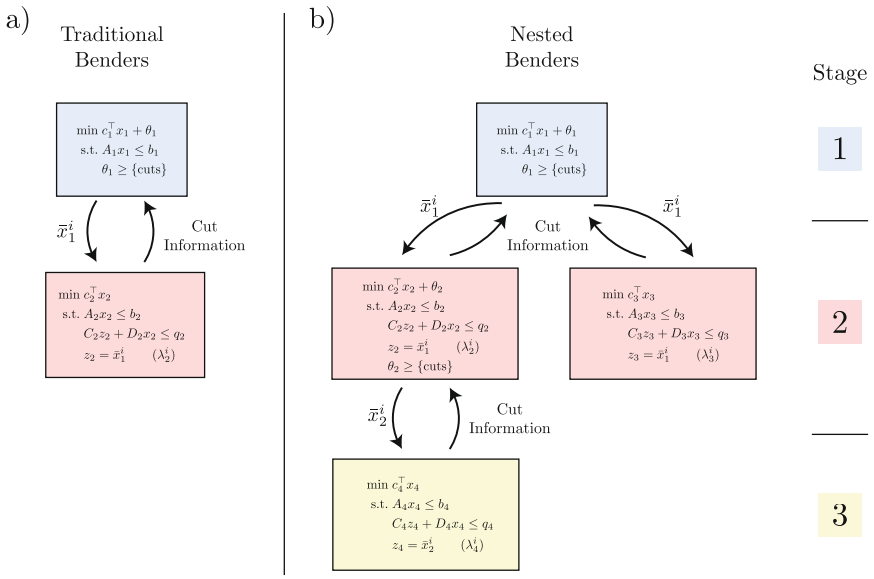


Fig. 6 Example of the Benders decomposition algorithm, with the traditional Benders case on the left and the nested Benders case on the right. Primal information is shared in a forward pass (the downward direction) and information for cuts is shared in a backward pass (the upward direction)

can be used to facilitate the derivation of gBD and introduce a `Julia` package `PlasmoBenders.jl` that implements gBD.

3.1 Benders decomposition

BD is a powerful decomposition approach for solving linear and mixed-integer linear programs [18, 77]. The most general form of BD (also referred to as nested BD or dual dynamic programming) is applied to problems with tree graph structures. Here, the problem is partitioned into multiple, sequential "stages" (each stage has a corresponding optimization subproblem); a visualization of this algorithm is given in Figure 6. The general approach of BD is to solve each stage of a problem sequentially in a forward pass, passing the primal solution of the previous stage to the next stage, and then performing a backward pass, where information (often dual values) is passed from each stage to the previous stage to create cutting planes. Each pass of cutting planes helps to form a better lower bound on the problem. An iteration of BD consists of a forward pass and a backward pass, which produce an upper and lower bound, respectively, and the iterations continue until the upper and lower bounds converge or until another termination criterion is reached. The traditional BD algorithm consists of two stages (Figure 6a), the first containing a master problem and the second containing a subproblem. However, BD can be applied to problems with multiple stages; see Figure 6b.

BD is applied to problems that can be separated into "stages" and "subproblems". By "stages" we are referring to a set of subproblems that depend on "complicating

variables" (i.e., variables that are linked by constraints to the subproblem's variables) from the previous stage. These subproblems typically become easier to solve once those complicated variables are fixed. The form of these problems is given by

$$\min \sum_{s=1}^{S} \left(\sum_{w \in W_s} c_w^\top x_w \right) \tag{5a}$$

$$\text{s.t. } A_w x_w \leq b_w, \quad w \in W_s, s \in \{1, \ldots, S\}, \tag{5b}$$

$$C_v x_w + D_v x_v \leq q_v, \quad v \in \mathscr{C}(w), w \in W_s, s \in \{1, \ldots, S-1\} \tag{5c}$$

where $\mathscr{S} = \{1, \ldots, S\}$ is the set of stages (with S the total number of stages); W_s is the set of subproblem indices at stage $s \in \mathscr{S}$; x_w, $w \in W_s$ are the decision variables (can be continuous or integer); c_w, A_w, b_w, C_w, D_w and q_w, $w \in W_s$ are vector and matrix data; and $\mathscr{C}(w)$, $w \in W_s$, $s \in \{1, \ldots, S-1\}$ is the index set of child subproblems for a subproblem w. By child subproblems, we are referring to the subproblems connected by the linking constraints (5c) that are in the next stage $s+1$ (note that a subproblem at stage $s+1$ cannot be a child of multiple subproblems in the previous stage). In other words, $\mathscr{C}(w)$ is the set of all $v \in W_s, s \in \{2, ..., S\}$ where there are nonzero matrices C_v and D_v being multiplied by x_w and x_v, respectively. Hence, for child subproblems of a subproblem w, variables x_w can be considered the complicating variables (note that $S = 2$ for the traditional BD case). Further, we note that (5c) is written as inequality constraints, but this formulation could easily be extended to equality constraints, since any equality constraint can be written with two inequality constraints.

To solve (5), the BD algorithm partitions the problem into a root (or master) problem and subsequent subproblems across $S-1$ stages. The root (first stage) problem has the form

$$\underline{\phi}_r^i := \min c_r^\top x_r + \theta_r \tag{6a}$$

$$\text{s.t. } A_r x_r \leq b_r \tag{6b}$$

$$\theta_r \geq \{\text{cuts}\}. \tag{6c}$$

where $r \in W_1$ and W_1 is a singleton, $\underline{\phi}_r^i$ is the optimal value of (6), i is the iteration number, and θ_r is the cost-to-go variable which underestimates the optimal value of the linked subproblems in the next stage. To approximate the optimal value of the subproblem, the root problem uses cutting planes (i.e., cuts) to constrain variable θ_r. These cuts are based on information from the solution of the next stage subproblems and are said to be valid if they underestimate the optimal value of the subproblems. Note that rather than using a single cost-to-go variable, θ_r could be replaced in the objective by $\Sigma_{w \in W_2} \theta_{r,w}$, and constraint (6c) could be replaced by constraints on each individual $\theta_{r,w}$, where $\theta_{r,w}$ is a cost-to-go variable which underestimates the optimal value of individual subproblems at the subsequent stage (rather than an aggregation of subproblems at the subsequent stage). These two approaches are referred to as an aggregated cut when using θ_r and multi-cuts when using $\theta_{r,w}$, $w \in W_2$, and both

can have their own benefits [15, 16, 103]. The multi-cut approach adds cuts after each iteration of the algorithm and can be more efficient for some problems, but the aggregated approach results in fewer constraints being added to the root problem at each iteration, which can be important if there are already many constraints on the root problem or if $|W_2|$ is large.

Each subsequent subproblem $w \in W_s$, $s \in \{2, ..., S-1\}$ with $\mathscr{C}(w) \neq \emptyset$, has the form

$$\begin{aligned} \overline{\phi}_w^i(\bar{x}_{p_w}^i) := \min\ & c_w^\top x_w + \theta_w && \text{(7a)} \\ \text{s.t. } & A_w x_w \leq b_w && \text{(7b)} \\ & C_w z_w + D_w x_w \leq q_w && \text{(7c)} \\ & z_w = \bar{x}_{p_w}^i \quad (\lambda_w^i) && \text{(7d)} \\ & \theta_w \geq \{\text{cuts}\}. && \text{(7e)} \end{aligned}$$

where p_w is the index of the parent subproblem of w (i.e., $w \in \mathscr{C}(p_w)$), $\bar{x}_{p_w}$ is a feasible solution of (7), and where $\overline{\phi}_w$ is the optimal value of (7). Note that, unlike (5), which is written in terms of the parent subproblem, (7) is written in terms of the child subproblem. As with the root problem, the cuts in (7) could be of the form of aggregated cuts or multi-cuts. A feasible solution of the previous stage subproblem, $\bar{x}_{p_w}$, is also passed to (7), and the constraints linking the two subproblems, (5c), are also enforced in (7c). The final subproblems (i.e., subproblems $w \in W_s$, $s \in \{2, \ldots, S\}$, with $\mathscr{C}(w) = \emptyset$) have the form

$$\begin{aligned} \overline{\phi}_w^i(\bar{x}_{p_w}^i) := \min\ & c_w^\top x_w && \text{(8a)} \\ \text{s.t. } & A_w x_w \leq b_w && \text{(8b)} \\ & C_w z_w + D_w x_w \leq q_w && \text{(8c)} \\ & z_w = \bar{x}_{p_w}^i \quad (\lambda_w^i). && \text{(8d)} \end{aligned}$$

where (8) has a similar form to (7) but where there is no cost-to-go variable since there are no children subproblems in future stages. We also note that at any given stage, not all subproblems of that stage may have children subproblems.

With the root and subproblems defined, we can now define the upper and lower bounds. Assuming relatively complete recourse of the subproblems (this can be ensured by adding slack variables to constraints and penalizing these slacks in the objectives), the upper bound at the kth iteration is given by

$$UB_{BD}^k := \min_{i=1,2,\ldots,k} \left(\sum_{s=1}^{S} \sum_{w \in W_s} c_w^\top \bar{x}_w^i \right). \tag{9}$$

The cost-to-go variables θ_w are an underestimate of the objective value of the subsequent subproblems, and if the cuts in (6c) or (7e) are valid, then at iteration i, $\sum_{w \in W_s} \theta_w \leq \sum_{w \in W_{s+1}} \overline{\phi}_w^i$ (or for multicuts, $\theta_{w,v} \leq \overline{\phi}_v^i$, $w \in W_s$, $v \in \mathscr{C}(w)$). Thus,

since cost-to-go variables in any stage are underestimating the next stage optimal value through every stage of the problem, the lower bound is given by the optimal solution of the root-problem, $LB_{BD}^{k} := \underline{\phi}_{r}^{k}$. We will omit the subscript BD when it is obvious from context.

The BD algorithm is summarized in Algorithm 1. The maximum number of iterations, K_{max}, the tolerance ϵ_{tol}, and any cut data are set. An iteration of the algorithm consists of first solving the root subproblem, (6), which gives a lower bound on the solution. Then, each stage is sequentially solved (including all subproblems in that stage) with feasible solutions of the previous stage fixed in the current stage. The feasible solutions that are passed to the next stage are not required to be the optimal solutions of the previous stage; in some cases, other feasible solutions can be chosen, such as by using a regularization procedure [58, 73, 80], which can yield better cuts. After each subproblem is solved, the upper bound is computed. If the gap between the upper and lower bounds is below ϵ_{tol}, the solutions at the iteration that yielded the upper bound are returned. If not, new cuts are computed and added to the respective root and subproblems, and the process is repeated. Note that the initial pass through the stages is often considered a forward pass because each stage is solved sequentially. The computation of new cuts is often considered a backward pass because it can likewise be a sequential process starting at the last stage and proceeding backward to the first, adding a cut to the previous stage before resolving it. If the number of iterations reaches K_{max} without reaching the solver tolerance, the algorithm terminates.

BD is a useful decomposition approach, but it has some limitations. It is best suited for large and possibly intractable problems since BD results in smaller problems that are iteratively solved. However, BD presents a challenge in that it may have several stages that must be solved sequentially, which can limit the parallelization. One method for addressing this is to place multiple subproblems in a given stage, as each of these subproblems can be solved in parallel [45]. The choice of which variables belong in the first stage (root) problem is an important modeling decision since placing linking variables in this stage can result in more subproblems in the second stage, each of which could be solved in parallel.

There are some extensions of BD that endeavor to parallelize more of the computations for problems with many stages [28, 78, 81, 82], but the sequential nature of BD can be a shortcoming for some problem instances. Consequently, having a modeling framework that allows for exploring various subproblem structuring (e.g., which variables belong to which problem, how many stages or subproblems per stage to use) can be important for applying BD in an efficient manner. For instance, one can imagine having an intractable monolithic problem that they structure into a multi-stage BD problem with one subproblem per stage (e.g., as in the structure seen in [55, 98]). One could then further structure the problem into a two-stage BD problem (e.g., as in the structure seen in [73]) by placing all linking variables between stages into a root or master problem, making each subproblem independent and parallelizable in the second stage.

Algorithm 1 Benders Decomposition

Set $K_{max}, \epsilon_{tol}, k = 1$, and cut data
while $k \leq K_{max}$ **do**
  Solve (6) to get $\underline{\phi}_r^k$ and set $LB^k = \underline{\phi}_r^k$
  Determine root-problem iterates $\bar{x}_r^k$
  for $s \in \{2, ..., S\}$ **do**
    for $w \in W_s$ **do**
      Solve (7) and (8) for w to get $\bar{\phi}_w^k$
      If $s < S$, determine decisions $\bar{x}_w^k$ to use in next stage subproblems.
    end for
  end for
  Compute UB^k from (9)
  if $(UB^k - LB^k)/|LB^k| \leq \epsilon_{tol}$ **then**
    Set $j = \text{argmin}_{i=1,...,k} \left(\sum_{s=1}^{S} \sum_{w \in W_s} c_w^\top \bar{x}_w^i \right)$
    Return solutions $\bar{x}_w^j$, $w \in W_s, s = 1, \ldots, S$ and UB^j
    Stop
  else
    for $s \in S - 1, ..., 1$ **do**
      for $w \in W_s$ **do**
        Compute new cut(s) and add them to (6c) or (7e)
        Solve (6) or (7) for w
      end for
    end for
    $k = k + 1$
  end if
end while
if $(UB^k - LB^k)/|LB^k| > \epsilon_{tol}$ **then**
  Set termination status to `Maximum Number of Iterations Exceeded`.
end if

3.2 Graph-Based benders decomposition

To define gBD, we identify basic structures and syntax that we will use to represent the decomposed subproblems (e.g., the root and subproblems of BD). So long as subgraphs do not overlap, subgraphs and nodes can be treated as interchangeable since each encodes an optimization problem. Thus, an algorithm generalized for an OptiGraph could use either subgraphs or nodes as subproblems. We will write the graph-based algorithms in terms of subgraphs rather than nodes because subgraphs can capture hierarchy and because `Plasmo.jl` is designed to be more subgraph-centric.

gBD requires a graph of the form $\mathcal{G}(\{\mathcal{G}_i\}_{i \in \{1,...,N\}}, \emptyset, \mathcal{E}_\mathcal{G})$ where no subgraphs in $\mathcal{G}$ overlap. The graph $\mathcal{G}$ must also meet the following criteria, where we introduce the notation $\mathcal{SG}(e)$ as the set of subgraphs connected by an edge e:

1. $|\mathcal{SG}(e)| = 2, e \in \mathcal{E}_\mathcal{G}$. i.e., any edge connecting the subgraphs $\mathcal{SG}(\mathcal{G})$ only connects two subgraphs. This condition is required because $\mathcal{G}$ is a hypergraph and therefore a single edge has the capability of connecting more than two subgraphs.
2. $\bigcup_{e \in \mathcal{E}_\mathcal{G}} \mathcal{SG}(e) = \mathcal{SG}(\mathcal{G})$. i.e., each subgraph in $\mathcal{G}$ must be connected to at least one other subgraph in $\mathcal{G}$ so that there are no disconnected subgraphs.

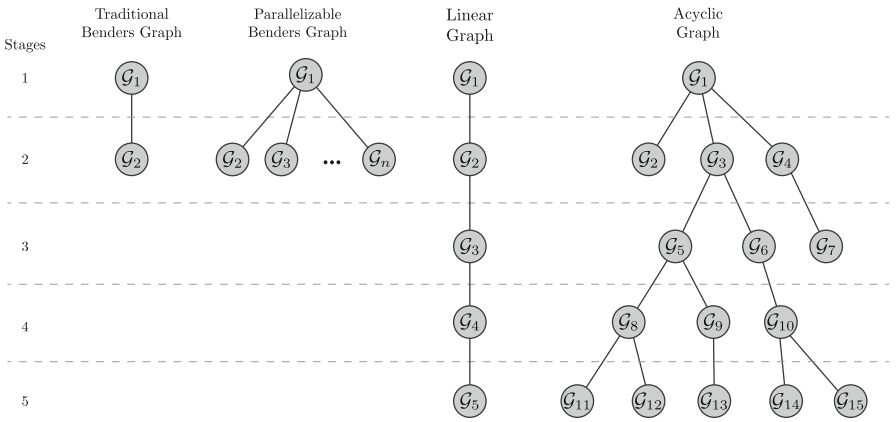


Fig. 7 Examples of the graph structure required for gBD, including a traditional BD case, a BD case with parallelizable subproblems, a linear graph, and a more complex acyclic graph. Here, subgraphs are represented by circles (nodes) but can embed more structure, nodes, and complexity. The root subgraph, $\mathcal{G}_r$ is set to be $\mathcal{G}_1$. Once the root subgraph is set, the remainder of the subgraphs are placed into stages

3. The subgraphs form a tree structure. That is, for any subgraph $g_i \in \mathcal{SG}(\mathcal{G})$, it is not possible to move along edges to another subgraph $g_j \in \mathcal{SG}(\mathcal{G})$ and return to g_i without traversing across the same set of subgraphs again. In other words, there are no cycles among any subgraphs $\mathcal{SG}(\mathcal{G})$.

Some examples of the required structure are visualized in Figure 10, where each subgraph (i.e., subproblem) is represented by a node and where each depicted edge could contain many edges between OptiNodes in the different subgraphs. The subgraph structure for each graph is an acyclic (tree) graph. The first graph shows the traditional BD case where there is a root problem ($\mathcal{G}_1$) and a subproblem ($\mathcal{G}_2$). The second graph is a BD case where the subproblem subgraphs ($\mathcal{G}_2$ - $\mathcal{G}_n$) can be solved in parallel since there are no links between the second stage subgraphs. The third graph is a linear graph; these graphs commonly arise in dynamic optimization, where each time point is constrained by the previous and next time points. The final graph is a more complex tree graph, which includes multiple subgraphs in each stage. In all three cases, we treat $\mathcal{G}_1$ as the "root" or first-stage subgraph. However, any subgraph could be chosen as the root subgraph (this could change the number of stages), and the choice of root subgraph can have important algorithmic implications. We will notate the root subgraph as $\mathcal{G}_r$.

We will also define the mathematical form of the problems defined on the subgraphs and edges. Each subgraph $g \in \mathcal{SG}(\mathcal{G})$ corresponds to the optimization problem:

$$\min\ c_g^\top x_g \tag{10a}$$

$$\text{s.t. } A_g x_g \leq b_g. \tag{10b}$$

Each edge $e \in \mathcal{E}_\mathcal{G}$ contains constraints of the form

$$\sum_{g \in \mathcal{SG}(e)} F_{g,e} x_g \leq q_e \tag{11}$$

where $|\mathscr{SG}(e)| = 2$ by the definition of the gBD structure and where $F_{g,e}$ is a matrix. Note that the optimization problem on the overall OptiGraph $\mathscr{G}$ has the form of (5), where (5a) is represented by the summation of the subgraphs objectives, (10a), the constraints (5b) are represented by the constraints (10b), and the linking constraints (5c) are represented by (11).

The optimization problem on each subgraph in (10) will be updated to create the subproblem structures given by (6), (7), and (8). An important task in gBD is choosing the root subgraph, and any $g \in \mathscr{SG}(\mathscr{G})$ can be chosen. The root subgraph is updated to have the form

$$\underline{\phi}_{\mathscr{G}_r} := \min\ c_{\mathscr{G}_r}^\top x_{\mathscr{G}_r} + \theta_{\mathscr{G}_r} \tag{12a}$$

$$\text{s.t. } A_{\mathscr{G}_r} x_{\mathscr{G}_r} \leq b_{\mathscr{G}_r} \tag{12b}$$

$$\theta_{\mathscr{G}_r} \geq \{\text{cuts}\} \tag{12c}$$

which matches the form of (6). The subsequent subgraphs $g \in \mathscr{SG}(\mathscr{G}) \setminus \mathscr{G}_r$ with $\mathscr{C}(g) \neq \emptyset$ (where $\mathscr{C}(g)$ are the child subgraphs or the subgraphs connected by an edge to g in the next stage) take the form

$$\overline{\phi}_g^i := \min\ c_g^\top x_g + \theta_g \tag{13a}$$

$$\text{s.t. } A_g x_g \leq b_g \tag{13b}$$

$$C_{p_g,e} z_g + C_{g,e} x_g \leq q_e, \quad e \in \{e : e \in \mathscr{E}_\mathscr{G}, g, p_g \in \mathscr{SG}(e)\} \tag{13c}$$

$$z_g = \bar{x}_{p_g}^i \qquad (\lambda_g^i) \tag{13d}$$

$$\theta_g \geq \{\text{cuts}\} \tag{13e}$$

which matches the form of (7) and where p_g is the parent graph of g (the graph connected by an edge to g and which is in the previous stage). Finally, the subgraphs with a parent subgraph but no children subgraphs take the form

$$\overline{\phi}_g^i := \min\ c_g^\top x_g \tag{14a}$$

$$\text{s.t. } A_g x_g \leq b_g \tag{14b}$$

$$C_{p_g,e} z_g + C_{g,e} x_g \leq q_e, \quad e \in \{e : e \in \mathscr{E}_\mathscr{G}, g, p_g \in \mathscr{SG}(e)\} \tag{14c}$$

$$z_g = \bar{x}_{p_g}^i \qquad (\lambda_g^i) \tag{14d}$$

which matches the form of equation (8). In both (13) and (14), the edge constraints (11) are enforced as (13c) and (14c) in the subgraph that is downstream from the root subgraph. The upper bound at the kth iteration of gBD is given by

$$UB_{gBD}^k := \min_{i=1,2,\ldots,k} \sum_{g \in \mathscr{SG}(\mathscr{G})} c_g^\top \bar{x}_w^i \tag{15}$$

where we will omit the subscript gBD when it is obvious from context. As with BD, the lower bound is given by $LB_{gBD}^k := \underline{\phi}_{\mathscr{G}_r}^k$.

The gBD algorithm is summarized in Algorithm 2. A root subgraph, $\mathscr{G}_r$ is defined for the graph $\mathscr{G}$. The root subgraph is updated to take the form of (12). The graph then must be traveled through (traversed) to update each subgraph and note its stage. This is done by moving from the root subgraph to each stage of the new graph using two new sets: a queue set, $\mathscr{SG}_{queue}$ that records the subgraphs that need to be explored next, and a visited set, $\mathscr{SG}_{visited}$, which includes a set of all subgraphs that have been visited (i.e., subgraphs that have been updated to take the form of (12), (13), or (14)). This is effectively a breadth-first search [23, 57]. We also define a function f_{stage} that maps a subgraph to its stage. After the subgraphs have been updated, the maximum number of iterations, K_{max}, the tolerance, ϵ_{tol}, and cut data are set, as well as the total number of stages. The graph is then sequentially solved, starting at the root subgraph (which also yields the lower bound), and then progressing through each stage, where the solutions of the previous stage are passed to the next stage and those solutions fixed as in (13d) and (14d). Note that each set of subgraphs at a given stage can be solved in parallel. After each subgraph is solved, the upper bound is computed. If the relative gap between the upper and lower bounds is below the tolerance, the algorithm terminates, and the solutions of the iteration that yielded the best lower bound are returned. If the relative gap is not below the tolerance, new cuts are added to their respective subproblems, and another iteration is run.

Several variants of nested BD exist, and Algorithm 2 can be easily extended in several ways, many of which are implemented in our software. For instance, Algorithm 2 assumes an aggregated cut approach, but this can be easily extended to multi-cuts. In addition, the most common nested BD approach for the backwards pass is outlined in this algorithm, where the backwards pass sequentially solves stages in reverse order, starting at stage $S-1$ and proceeding backwards to stage 1, passing a cut each time. In doing so, it allows information to propagate backwards each iteration. However, other variants of this approach exist, and the variant used in the third case study below is one in which the "backwards pass" is done in parallel, meaning cuts are not added from stage $i+1$ to i before solving stage i at each iteration. This variant has an added benefit for LPs because the backwards pass has no computational cost since dual information is available from the forward pass. This approach has been taken in literature (see [54] or [7]), and both implementations of the backwards pass are supported in `PlasmoBenders.jl` beginning in version 0.2.0.

The requirement that there be no cycles among subgraphs for gBD may seem restrictive. However, the flexibility of the OptiGraph abstraction can allow the user to obtain the required structure from any graph via aggregation/partitioning procedures. For example, Figure 8a shows a subgraph structure containing a cycle (among $\mathscr{G}_2$, $\mathscr{G}_3$, and $\mathscr{G}_4$). However, if the graph is partitioned with the cycle being placed on a single subgraph, it results in a subgraph structure without cycles. This can be thought of as aggregating the cycle into a single node or subgraph. More complex structures can similarly be partitioned into the required structure, as shown in Figure 8b, where we give two examples. First, we can arbitrarily partition any set of subgraphs into two separate subgraphs, forming the traditional BD case. In fact, any optimization problem with linking constraints can be partitioned into this traditional BD structure, since we can arbitrarily choose to place variables into one of two subgraphs (in this

Algorithm 2 Graph Benders Decomposition

Set OptiGraph $\mathcal{G}(\{\mathcal{G}_i\}_{i\in\{1,\ldots,N\}}, \emptyset, \mathcal{E}_{\mathcal{G}})$ and root subgraph $\mathcal{G}_r \in \mathcal{SG}(\mathcal{G})$
Ensure: $\mathcal{G}$ meets the three criteria for graph structure for gBD
Define root subgraph $\mathcal{G}_r$
Define a new set for subgraphs in queue, $\mathcal{SG}_{queue} := \mathcal{C}(\mathcal{G}_r)$
Define a new set for subgraphs that have been visited, $\mathcal{SG}_{visited} := \{\mathcal{G}_r\}$
Define a function, f_{stage}, that maps a sugraph to its stage.
Set $f_{stage}(\mathcal{G}_r) = 1$
while $|\mathcal{SG}_{visited}| \neq |\mathcal{SG}(\mathcal{G})|$ **do**
  for $g \in \mathcal{SG}_{queue}$ **do**
    if $\mathcal{C}(g) \neq \emptyset$ **then**
      Formulate problem (13) for g
      Set $\mathcal{SG}_{queue} \leftarrow \mathcal{SG}_{queue} \cup \mathcal{C}(g)$
    else
      Formulate problem (14) for g
    end if
    Set $f_{stage}(g) = f_{stage}(p_g) + 1$
    Set $\mathcal{SG}_{queue} \leftarrow \mathcal{SG}_{queue} \setminus g$
    Set $\mathcal{SG}_{visited} \leftarrow \mathcal{SG}_{visited} \cup \{g\}$
  end for
end while
Set K_{max}, ϵ_{tol}, $k = 1$, and cut data, and set $S = \max_{g\in\mathcal{SG}(\mathcal{G})} f_{stage}(g)$
while $k \leq K_{max}$ **do**
  Solve root subgraph to compute $\underline{\phi}^k_{\mathcal{G}_r}$ and set $LB^k = \underline{\phi}^k_{\mathcal{G}_r}$
  Determine root-problem decisions $\bar{x}^k_{\mathcal{G}_r}$
  for $s \in \{2, \ldots, S\}$ **do**
    for $g \in \{g : f_{stage}(g) = s, g \in \mathcal{SG}(\mathcal{G})\}$ **do**
      Solve the corresponding subproblem and compute $\overline{\phi}^k_g$
      Determine iterates to use in next stage subproblems, $\bar{x}^k_g$
    end for
  end for
  Compute UB^k from (15)
  if $(UB^k - LB^k)/|LB^k| \leq \epsilon_{tol}$ **then**
    Set $j = \text{argmin}_{i=1,2,\ldots,k} \sum_{g\in\mathcal{SG}(\mathcal{G})} c_g^\top \bar{x}^i_g$
    Return solutions $\bar{x}^j_g$, $g \in \mathcal{SG}(\mathcal{G})$ and UB^k
    Stop.
  else
    for do $s \in \{S-1, \ldots, 1\}$
      for $g \in \{g : f_{stage}(g) = s, g \in \mathcal{SG}(\mathcal{G})\}$ **do**
        Compute new cut(s) and add them to (12c) or (13e)
        Solve (12) or (13) for g
      end for
    end for
    $k = k + 1$
  end if
end while
if $(UB^k - LB^k)/|LB^k| > \epsilon_{tol}$ **then**
  Set termination status to `Maximum Number of Iterations Exceeded`.
end if

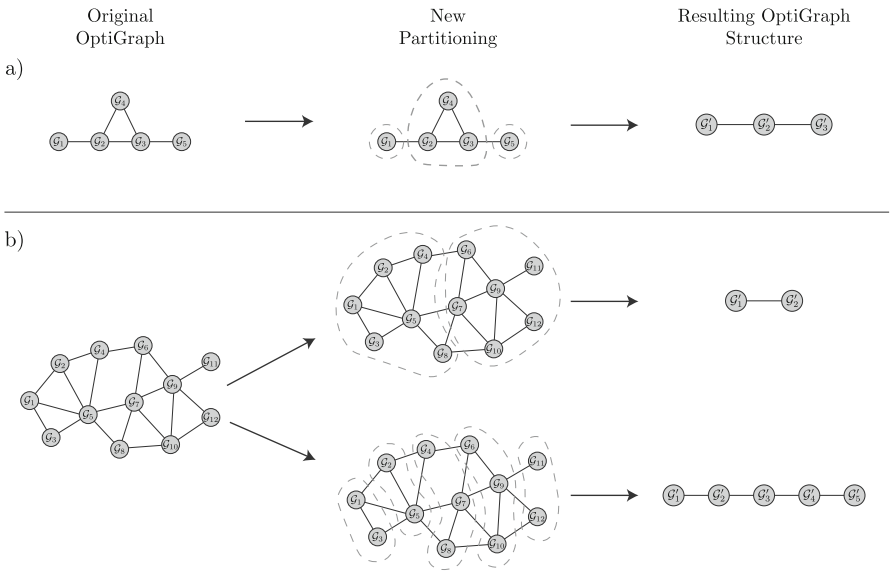


Fig. 8 Examples of cycle removal by aggregation. a) Shows a simple structure with a cycle where the cycle is destroyed by placing the cycle on its own partition. b) Shows a more complex structure that can be partitioned into subgraphs for the traditional BD approach or partitioning into a longer linear graph

sense, *the gBD algorithm is broadly applicable to any graph structure*). Second, we can set partitions in such a way that we create an acyclic structure. In this case, we set five partitions and create a linear graph structure (such as that seen in dynamic programming). If we consider the original graph to be a spatial structure, we can see that we can recover a linear graph structure that is commonly seen in temporal problems (this is the structure required by dual dynamic programming), and to which we can apply gBD.

While partitioning and aggregation functionalities allow for removing cycles, structures can also be specified directly by the modeler. Figure 9 shows an example of a graph with three subgraphs that form a cycle. The cycle can be removed by creating dummy variables on one of the subgraphs such that a linking constraint can be removed (this is a lifting procedure). In this case, a variable $z_{\mathcal{G}_2}$ is added to $\mathcal{G}_2$ that is constrained to be equal to the variables on $\mathcal{G}_1$ (i.e., $z_{\mathcal{G}_2} = x_{\mathcal{G}_1}$). Then, the constraint $c_3(x_{\mathcal{G}_1}, x_{\mathcal{G}_3}) = 0$ can be placed instead between $\mathcal{G}_2$ and $\mathcal{G}_3$ by the constraint $c_3(z_{\mathcal{G}_2}, x_{\mathcal{G}_3})$ which forms a linear graph. This changes the structure of the problem but results in an optimization problem that will have the same solution as before the lifting procedure. This same procedure could be applied to more complex graphs with many cycles.

The gBD algorithm (Algorithm 2) also provides flexibility in how this is applied based on the choice of the root subgraph. Any subgraph in the graph can be set as the root subgraph, so long as the subgraphs form the required structure noted above. The choice of root subgraph could have algorithmic and convergence implications. For instance, Figure 10 shows the stages for three different root subgraphs for a given

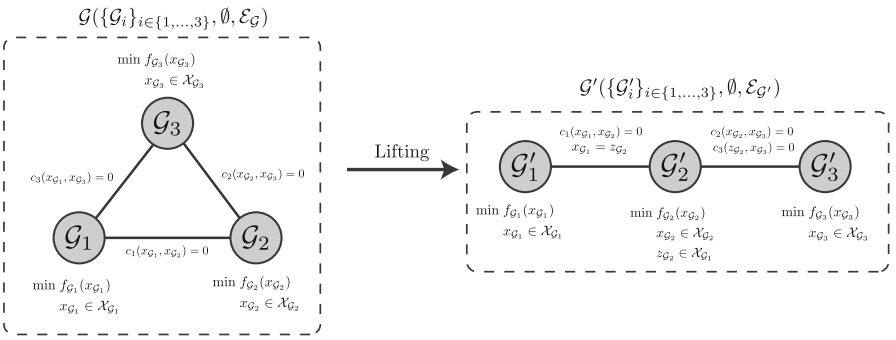


Fig. 9 Visualization showing how lifting of an optimization problem changes the graph structure. The left graph, $\mathcal{G}$, contains a cycle among its subgraphs. However, dummy variables $z_{\mathcal{G}_2}$ can be added to subgraph $\mathcal{G}_2$ and set equal to the variables $x_{\mathcal{G}_1}$ and then the constraint $c_3(x_{\mathcal{G}_1}, x_{\mathcal{G}_3})$ can be replaced by $c_3(z_{\mathcal{G}_2}, x_{\mathcal{G}_3})$ which eliminates the connection between $\mathcal{G}_1$ and $\mathcal{G}_3$ and results in the graph $\mathcal{G}'$ which has no cycles

OptiGraph. Each decision results in a different number of stages. Further, the number of connections to the next stage can also be different and could result in different cuts (or number of cuts). With $\mathcal{G}_r = \mathcal{G}_2$, the subgraph $\mathcal{G}_3$ has two child subgraphs, but $\mathcal{G}_r = \mathcal{G}_3$ results in $\mathcal{G}_3$ having three child subgraphs. As a further example, linear graphs can arise from temporal problems and can be solved using a nested BD approach (i.e., dual dynamic programming) with the root problem being the subgraph with the first time point. However, we could instead apply this same algorithm starting at the middle time point, resulting in half as many stages. With the flexibility provided by the partitioning and aggregation capabilities of the OptiGraph, many different root subgraphs could be explored for the same problem, such as exploring different partitionings of integer and continuous variables into subgraphs or choosing a root subgraph with a larger number of integer variables. This ability to define any subgraph as the root subgraph in gBD is an important and flexible ability of the algorithm.

Cutting planes are added to (6c), (7e), (12c), or (13e) in part based on problem structure. During the gBD algorithm, the graph is spanned in the forward direction, starting at the root subgraph and moving across stages. For instance, in Figure 7, the forward pass will start in Stage one and sequentially move to the next stages. After the forward pass, cuts can be added in a backward pass. Often, this will include starting at the final stage and spanning the graph backwards. This is often the case when subgraphs contain integer variables, since the forward pass requires solving an MILP, the backward pass can require solving an LP relaxation, depending on the types of cuts chosen.

The cuts added to (6c), (7e), (12c), or (13e) can take several forms [55, 102]. Below, we will detail Benders cuts, Lagrangian cuts, and strengthened Benders cuts based on the work of [54, 55, 102]. We note that [102] applied their work to the case of binary state variables (i.e., binary complicating variables), but [55] extended their work to allow for both binary and continuous variables, which comes at the expense of losing the finite convergence property due to the potential of a duality gap. For this reason, we did not specify any variables being continuous or integer in the above algorithm

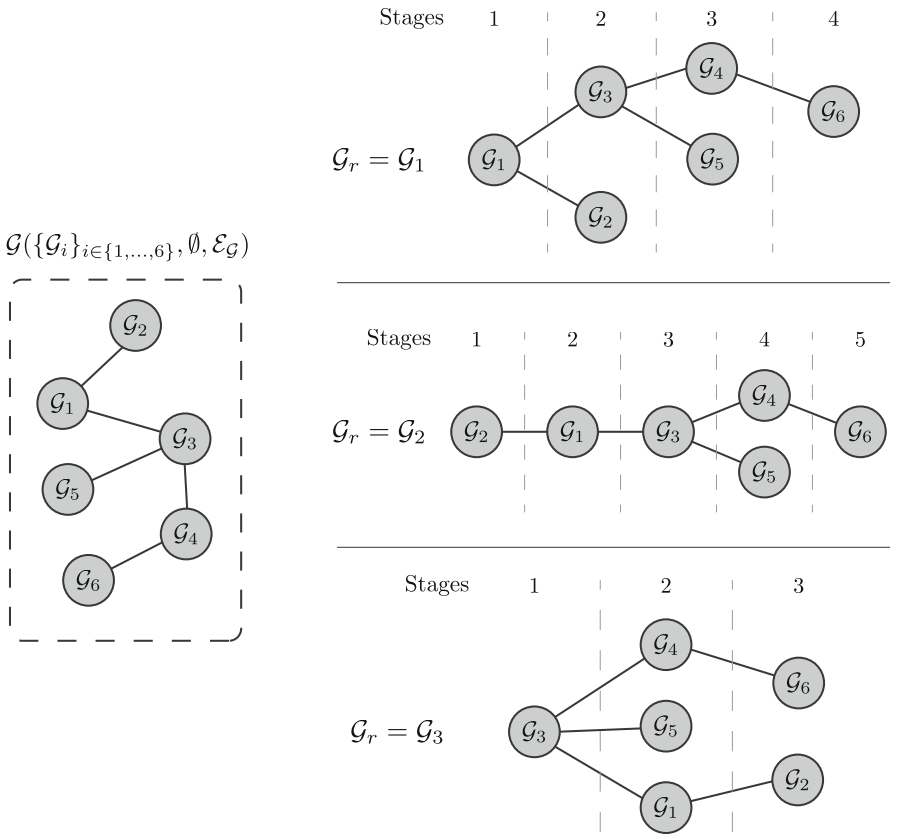


Fig. 10 Visualization of how the choice of root subgraph influences how the gBD algorithm is applied. The graph, $\mathcal{G}$, on the left contains six subgraphs but has the required structure for gBD. On the right are the resulting stages for setting $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_3$ as the root subgraph. Each results in a different number of stages and could result in different cuts being added to the subproblems. For instance, when using multi-cuts, using $\mathcal{G}_1$ or $\mathcal{G}_2$ as the root subgraph results in two sets of cuts to $\mathcal{G}_3$ at each iteration, but using $\mathcal{G}_3$ as the root subgraph results in three sets of cuts being added to $\mathcal{G}_3$ at each iteration

definitions. The cuts we discuss below will have the general form:

$$\Psi_w^i(\Phi_w^i, \pi_w^i, \bar{x}_{p_w}) := \Phi_w^i + \pi_w^{i\top}(x_{p_w} - \bar{x}_{p_w}^i) \tag{16}$$

where $\Psi_w^i(\Phi_w^i, \mu_w^i, \bar{x}_{p_w})$ is the lower bound on the subproblem w, Φ_w^i and π_w^i are information coming the child subproblems, and x_{p_w} are the complicating variables that come from the parent subproblem, p_w (with $\bar{x}_{p_w}$ being the fixed feasible solution; see (7d), (8d), (13d), and (14d)). Here, the set of aggregated cuts at iteration k is given by $\theta_{p_w} \geq \sum_{w\in\mathscr{C}(p_w)} \Psi^i(\bar{\Phi}_w^i, \pi_w^i, \bar{x}_{p_w}),\ i = 0, 1, ..., k-1$ and the multi-cut is given by $\theta_{p_w,w} \geq \Psi^i(\bar{\Phi}_w^i, \pi_w^i, \bar{x}_{p_w}),\ w \in \mathscr{C}(p_w),\ i = 0, 1, ..., k-1$. Each algorithm given above allows the user to set initial cut data (i.e., $i = 0$), where additional cuts are added after each iteration.

Benders cuts come from LP duality properties [54]. If (7) or (8) are LPs, the Benders cut for decision x_{p_w} at iteration $i+1$ for the wth subproblem can be given by $\Psi^i(\bar{\phi}^i_w, \bar{\lambda}^i_w, \bar{x}_{p_w})$. This cut comes from the fact that (7) and (8) can be written in dual form with the objective $\overline{\phi}^{i+1}_w(\bar{x}^{i+1}_{p_w}) = \lambda^{i+1\top}_w x_{p_w} + \nu^{i+1}$, where ν^{i+1} denotes the cost component of constraints (7b), (7c), (8b), and (8c). The cost to go at iteration $i+1$ can then be approximated by using information from the previous iteration, where we replace λ^{i+1}_w with $\bar{\lambda}^i_w$ and replace ν^{i+1} with $\bar{\phi}^i_w(\bar{x}^i_{p_w}) - \bar{\lambda}^{i\top}_w \bar{x}^i_{p_w}$, where $\bar{\phi}^i_w$ is the objective value of (7) or (8) at iteration i. When (7) or (8) are a MILP, rather than using $\bar{\phi}^i_w$, the Benders cut uses the solution of their LP relaxation. The above cuts are also easily extended to the graph-based case.

Lagrangian cuts come from a relaxation of the constraint (7d) or (8d). This relaxation for iteration i is given by:

$$\phi^{LR,i}_w(\bar{x}^i_{p_w}, \mu_w) := \min\ c^\top_w y_w - \mu^\top_w(z_w - \bar{x}^i_{p_w}) \tag{17a}$$

$$\text{s.t. } B_w y_w \leq b_w \tag{17b}$$

$$C_w z_w + D_w y_w \leq q_w \tag{17c}$$

for some Lagrange multiplier, μ_w. Optimal Lagrange multipliers $\bar{\mu}^i_w$ are found by solving

$$\Phi^{LR,i}_w(\bar{x}^i_{p_w}) = \max_{\mu_w} \phi^{LR,i}_w(\bar{x}^i_{p_w}, \mu_w). \tag{18}$$

Once the optimal lagrange multipliers, $\bar{\mu}^i_w$, have been determined, the lagrange cut takes the form $\Psi^i_w(\Phi^{LR,i}_w(\bar{x}^i_{p_w}), \bar{\mu}^i_w, \bar{x}_{p_w})$.

Benders cuts can be quick to compute but tend to be weak, while Lagrangian cuts can be expensive to compute because the solution of (18) can be expensive. Strengthened Benders cuts were introduced by [102] and seek to mitigate performance issues. Rather than solving (18), they solve $\phi^{LR,i}_w(\bar{x}^i_{p_w}, \bar{\lambda}^i_w)$, where $\bar{\lambda}^i_w$ comes from (7d) or (8d) at iteration i (or the LP relaxation if it is a MILP). Thus, the strengthened cut has the form $\Psi^i_w(\bar{\phi}^{LR,i}_w(\bar{x}^i_{p_w}, \bar{\lambda}^i_w), \bar{\lambda}^i_w, \bar{x}^i_{p_w})$.

3.3 Software implementation of gBD – `PlasmoBenders.jl`

To highlight how the graph-based abstraction provides a framework for implementing decomposition approaches, we now introduce the Julia package `PlasmoBenders.jl` which implements gBD as outlined above. Using the OptiGraph abstraction, we can generalize the decomposition scheme which enables the user to build a problem as a graph and then pass it to the generalized, graph-based solver. Additional solution options and algorithmic approaches (e.g., different types of cuts, regularization schemes) can likewise be generalized to the graph-based approach and implemented in the solver as we will show below. In this case, we use `PlasmoBenders.jl` to highlight these capabilities, but many of the principles described in this manuscript could be applied to other decomposition approaches.

The primary object in `PlasmoBenders.jl` is the `BendersAlgorithm` object, which is motivated by the gBD algorithm (Algorithm 2). This algorithm is applied to a graph containing subgraphs with edges between subgraphs, and its implementation is dependent on setting a root subgraph. Similarly, after a user has defined an OptiGraph object in `Plasmo.jl`, they can create a `BendersAlgorithm` object by passing that OptiGraph object, $\mathscr{G}$, as well as a root subgraph, $\mathscr{G}_r \in \mathscr{SG}(\mathscr{G})$. After setting the graph and root subgraph, the `BendersAlgorithm` constructor will make the changes to the subgraphs noted in Algorithm 2 (e.g., adding cost-to-go variables or enforcing the linking constraints). The `BendersAlgorithm` object also includes additional data structures for implementing gBD (e.g., best upper bounds, lower bounds, solver tolerance).

`PlasmoBenders.jl` gives the user access to several additional options and functionalities. More details on the package operation can be found in the documentation at https://github.com/plasmo-dev/PlasmoAlgorithms.jl/tree/main/lib/PlasmoBenders and in the Supporting Information, but we will highlight some of these here. Note that some of these topics (like regularization) are able to be generalized to the graph structure, simplifying their implementation. Available options include:

- Solver information such as K_{max} or ϵ_{tol}
- Whether to use strengthened Benders cuts rather than regular Benders cuts (if the problem is mixed-integer)
- Whether to use aggregated or multi-cuts
- Whether to parallelize certain computations, such as solutions of Benders subproblems when $S = 2$ and $|W_2| > 1$
- Whether to add slack variables to constraints (7c) or (8c) to help ensure recourse.
- Whether to warm-start the next iteration with the current best solution
- Whether to use a regularization scheme to get the variables $\bar{x}_g^i$ that are fixed in each subproblem at iteration i (see [73] and Supporting Information).
- Beginning in version 0.1.0, whether to use feasibility cuts if subproblems are infeasible (only supported for two stage problems)
- Beginning in version 0.2.0, whether to use a sequential backwards pass, where the stages are resolved in reverse order after cuts are passed to the previous stages.

As `PlasmoBenders.jl` is used in some of the case studies below, we give an example in Code Snippet 3 to show how it can be applied to an OptiGraph defined in `Plasmo.jl`. First, we define a graph with three nodes and two edges (Lines 4 - 18), where one edge is between node one and node two and one edge is between node two and node three (i.e., this is a linear graph). The graph is then partitioned so that each node is on its own subgraph (Lines 21 - 23). We then define a solver for the subgraphs (Line 26) which we will pass to the constructor to set on each of the subgraphs internally. Next, we can build the `BendersAlgorithm` object by calling the function `BendersAlgorithm` (Lines 29 - 40). This function takes the overall OptiGraph object `graph` as well as the root subgraph, which we use here as the first subgraph (containing node one). We can also set other information, including the maximum number of iterations, the solver tolerance, to use strengthened cuts, not to use regularization, not to add slack variables, not to use multicuts, and what to use for the subproblem solver. We can then call `run_algorithm!` on the

`BendersAlgorithm` object (line 43) which applies gBD to solve the problem. Solution information can be queried from the `BendersAlgorithm` object using `PlasmoBenders.jl` functions and extended `JuMP.jl` functions (Lines 46 - 48).

Code Snippet 3 Code for applying graph-based Benders and Nested Benders decompositions to an OptiGraph defined in `Plasmo.jl`. The package `PlasmoBenders.jl` is able to apply these decomposition schemes based on the user-defined graph and root subgraph.

```
using Plasmo, PlasmoBenders, HiGHS

# Define a graph with three nodes and two edges
graph = OptiGraph()
@optinode(graph, nodes[1:3])

# Set local variable, constraint, and objective information
for node in nodes
    @variable(node, x, Bin)
    @variable(node, y >= 0)
    @constraint(node, x + y >= 1.3)
    @objective(node, Min, x + 2 * y)
end

# Add linking constriants
for i in 1:2
    @linkconstraint(graph, nodes[i][:x] + nodes[i + 1][:y] >= i)
end

# Create subgraphs with one node on each subgraph
part_vector = [1, 2, 3]
partition = Partition(graph, part_vector)
apply_partition!(graph, partition)

# Define a solver to use for the subproblems
solver = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" =>
     false)

# Construct the BendersAlgorithm object
benders_alg = BendersAlgorithm(
    graph, # Set the overall graph
    local_subgraphs(graph)[2]; # Set the root subgraph
    max_iters = 20, # Maximum iterations
    tol = 1e-7, # Solver tolerance
    parallelize_benders = true, # Whether to parallelize the second stage solutions
    strengthened = true, # Whether to use strengthened cuts
    regularize = false, # Whether to use a regularization scheme
    add_slacks = false, # Whether to add slack variables for recourse
    multicut = true, # Whether to use multi-cuts rather than aggregated
    solver = solver # Set the solver on the subgraphs
)

# Solve the problem using gBD
run_algorithm!(benders_alg)

# Query Solutions
println("Objective value is ", JuMP.objective_value(benders_alg))
println("Relative gap is ", PlasmoBenders.relative_gap(benders_alg))
println("Value of x on node 1 is ", [JuMP.value(benders_alg, nodes
     [1][:x])])
```

The generalization of the BD algorithm to graphs and its implementation in `PlasmoBenders.jl` allows for a simple user-interface for applying what otherwise might be complex algorithms. In Code Snippet 3, it requires less than fifteen lines of code and only two function calls to solve a problem using gBD. Because the algorithm is generalized to a graph, the problem can be modeled as a `Plasmo.jl` OptiGraph, and the structure of that OptiGraph, along with the definition of the root

subgraph, determines the complicating variables and subproblem sizes. It also gives the user flexibility in how they will apply these algorithms such as allowing them to use strengthened cuts or regularization. Further, the graph above is a linear graph with three subgraphs. In Code Snippet 3, we solve using gBD and set the root subgraph to be `local_subgraphs(graph)[2]` (the second subgraph) which forms two stages, with two subgraphs in the second stage. We could instead use almost the same code but set the root subgraph to be `local_subgraphs(graph)[1]` (the first subgraph) which would have three stages. If we use the first subgraph as the root subgraph, the `multicut` option would have no influence because there is only one subpgraph in each stage. The `parallelize_benders` option also would throw an error since this parallelization is only implemented for two-stage problems. Moving between these algorithm options is simple to do in practice because it can be done by changing a few arguments in the `BendersAlgorithm` constructor function.

3.4 Other decomposition approaches

While this section has focused on gBD as a solution approach, other decomposition approaches for graphs are possible and these are not restricted to LP/MILP problems. Other approaches have been applied to graph-based optimization problems, either on the modeling level (using graphs to form subproblems) or on the linear algebra level (exploiting the structure formed by the graph for different computations). Jalving and co-workers applied an overlapping Schwarz decomposition scheme for LP and NLP graph structured problems [46, 85, 89]. A Schur decomposition has also been applied to graphs in two-stage stochastic NLPs [27], and an ADMM approach was applied to the graph structure of a NLP power system problem [87]. In addition, a Hansknecht et al. [42] showed how the linear algebra resulting from tree graph structures can be exploited or decomposed for applications in non-linear programming or sequential quadratic programming. These or other decomposition schemes could be generalized to graph structure as well, and there are instances of approaches such as ADMM being applied to problems that we believe could be readily modeled by the OptiGraph abstraction; for example, multiple studies introduce different distributed control strategies where ADMM is used as a solution approach for coordinating distributed subsystems, forming a hierarchical structure [8, 21, 95].

Hierarchical structures are also often leveraged for obtaining *approximate solutions*, such as partitioning problems into subproblems and solving those subproblems sequentially; this is often done when the problem is computationally intractable or when it is impractical to solve the full problem to optimality. For instance, doing a single iteration of gBD results in a feasible solution (assuming complete recourse) from the forward pass through the graph and could be used as a solution to the problem. This sequential solution approach is known as receding-horizon decomposition and is widely used in dynamic programming and model predictive control (for linear or tree graphs). Sequential decomposition approaches are used in many applications, including to hierarchical optimization problems (where decisions flow from one layer to the next), multi-scale problems (where a coarse problem is solved to create targets in a higher resolution problem), and to clustered problems, where clusters are iteratively

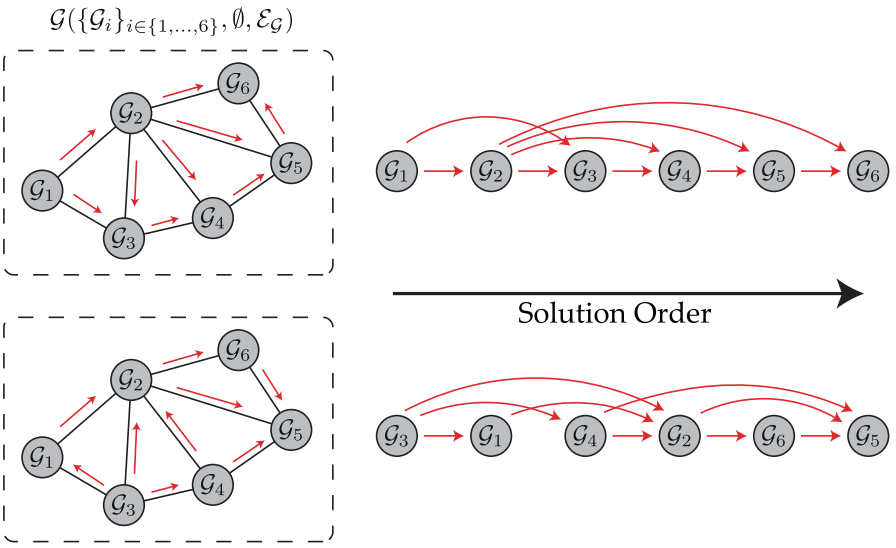


Fig. 11 Visualization showing how an OptiGraph can be solved sequentially to obtain a feasible solution, with two different solution orders shown. Subgraphs are solved in a specific order and solutions are passed to the next subgraph in order. Red arrows represent the direction in which solutions are shared. Linking constraints on the edges are enforced on the "downstream" subgraph. Solution order can be arbitrary, and a tree structure is not necessarily required

solved and solutions shared between clusters; further, this approach is not restricted to LPs, MILPs, NLPs, or MINLPs. A visualization of how a sequential solution approach can be applied to a graph is shown in Figure 11. Note that the choice of solution order, like the choice of root subgraph in gBD, is a choice made by the user. In addition, the sequential solution approach is not necessarily restricted to tree structures. In this case, each subgraph is solved in a set order and solutions (represented by the red arrows) are shared to the next subproblem. The linking constraints contained on the edges would be enforced on the downstream subgraph. Assuming recourse, this sequential approach yields an upper bound on the optimal solution. In addition, a lower bound can be obtained by solving each subgraph in parallel, ignoring the linking constraints between subgraphs.

There are many reported applications of sequential approximation approaches. Power systems studies frequently solve a series of separate optimization problems, such as first solving an MILP unit commitment problem, then solving a LP economic dispatch problem and/or LP or NLP system dynamics problems [29, 56, 96]. This sequential solution approach is often required to ensure tractability or because certain information, such as future demand, is not yet realized. Supply chain problems often include multiple echelons or decision layers [61, 99] which can be solved sequentially, such as is done by [84] for biorefineries, where multiple MILPs for flexibility and structural evaluation are solved sequentially. Supply chains often result in complex structures in addition to hierarchical layers, and in [39], they solve coupled shale gas and water supply chains by clustering the problem spatially (rather than by echelons) and sequentially solving and sharing solutions between clusters. Similarly, problems

in health care (often routing and scheduling problems) have been addressed using sequential strategies, such as by solving two problems in series (sharing solutions of the first problem with the second [79]) or by solving initial clustered problems and using the solutions to inform a secondary problem [11, 35]. In heating, ventilation, and air conditioning systems, [63] proposed a hierarchical control approach for coordinating temperature control of multiple rooms, where a coarse representation of the full system is solved, and the solution is then passed to lower-level controllers which operate independently. These different sequential approaches can be implemented within the OptiGraph abstraction by using subgraphs to capture the different subproblems. In addition, the freedom provided by the OptiGraph structuring capabilities (e.g., partitioning and aggregation) allows a user to explore additional representations of the same problem, which could lead to other applications of these sequential solution approaches. For instance, exploring different partitioning approaches and solving the resulting subgraphs sequentially could potentially yield multiple upper bounds on the optimal solution (one for each partitioning), while solving these same subgraphs in parallel (ignoring linking constraints between them) would provide different lower bounds.

4 Case studies

In this section, we use challenging case studies that show how the graph approach can be used for modeling hierarchical structures and obtain approximate solutions (case study I) and how gBD can be applied to solve hierarchical problems (case studies II and III). Scripts for reproducing all case studies, as well as a script for creating some of the visuals in case study 3, are available at https://github.com/zavalab/PlasmoBendersExamples.

4.1 Case Study 1:Hierarchical electricity markets

This problem is based on the work of Atakan and co-workers [6] who presented a tri-level hierarchy of market operations (Figure 12). This included three hierarchical layers that could each have different time resolutions and time horizons: a day-ahead unit commitment layer (DA-UC), a short-term unit commitment layer (ST-UC), and an hour-ahead economic dispatch layer (HA-ED). Atakan and co-workers [6] applied this framework to a 118-bus system [74], which we will likewise follow below. They treated this as a stochastic problem, but we will only use the deterministic form of the problem for simplicity. We will give a general overview of this problem and show how we model and solve it using OptiGraphs. Additional mathematical details can be found in the Supporting Information.

An overview of the three hierarchical layers are described below and in Table 1. Each hierarchical layer consists of different numbers of subproblems which each have different horizons and degrees of overlap between subproblems. The DA-UC subproblem is an MILP that spans a 25-hour period, is solved once per day, and runs with a coarse resolution. This layer commits an initial subset of generators while considering

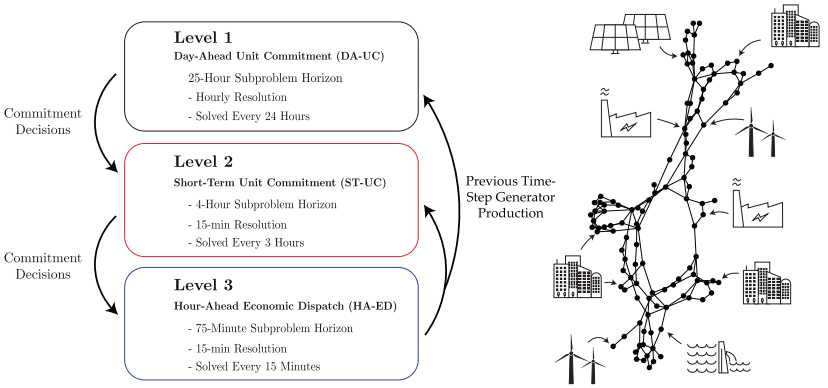


Fig. 12 The tri-level hierarchical architecture of Atakan and co-workers [6] of an electricity market

operation of these generators to meet a demand profile (e.g., it considers ramp up/-down constraints and minimum up/down time constraints). The ST-UC subproblems are MILPs that span 4-hour periods at a 15 minute resolution and which are solved every 3 hours; they use the commitment decisions of the DA-UC layer and also commit an additional subset of generators. The ST-UC layer considers operation of *both* the DA-UC generators and ST-UC generators, but maintains the commitment decisions of the DA-UC generators from the previous layer. The commitment decisions of the DA-UC and ST-UC layers are passed to the HA-ED layer, which include LP subproblems over a 75-minute time span and which are solved every 15 minutes. Each layer also includes power provided by renewable generators as well as demand data. The demand data can vary between layers (e.g., the realized demand of the HA-ED layer may differ from the projected demand of the DA-UC layer; see Supporting Information). Operation of the power systems in each layer was also subject to transmission constraints using DC approximations of the power flow. For this case study, we used the 118-bus system [74] used by Atakan and co-workers [6].

This problem is constructed as a graph in `Plasmo.jl`. An overall, monolithic graph, representing a single day, contains a subgraph for each subproblem (i.e., the overall graph contains 1 DA-UC subgraph, 8 ST-UC subgraphs, and 96 HA-ED subgraphs). Each subproblem subgraph contains nested sugraphs for any given time point. Each time point subgraph is made up of nodes for each bus in the system and nodes for each transmission line in the system. Each of these nodes contains variables corresponding to their objects (e.g., bus nodes contain variables for generators and line nodes contain variables for power flow). Edges are placed between the time point subgraphs in the same subproblem and between bus and line nodes (e.g., for up/down time constraints on generators or transmission constraints on lines). Additional edges are also placed between subproblem subgraphs which link solutions of different layers. A visualization of the resulting graph structure is shown in Figure 13. Here, the subproblem subgraphs are highlighted by red bubbles, and each time point is represented by a node (which encodes the 118-bus system OptiGraph). Edges are placed that link solutions of each layer. There are also additional edges between the DA-

Table 1 Details for the DA-UC, ST-UC, and HA-ED layers of the tri-level hierarchical optimization problem of Atakan et al. [6]

	DA-UC	ST-UC	HA-ED
Subproblem Horizon (hr)	25	4	1.25
Discretization Size (min)	60	15	15
Subproblems per Day	1	8	96
Variables per Subproblem	31,725	19,008	4,770
Binary Variables per Subproblem	11,775	3,744	0
Constraints per Subproblem	87,525	63,440	18,265
Renewable Generators	31	31	31
Conventional Generators	157	235	235
Conventional Generators Committed	157	78	0
Total Generators	188	266	266

UC and HA-ED layers that are not shown which enforce the DA-UC commitment decisions in the HA-ED subgraphs. In addition, each of the subproblem subgraphs on this figure exists on an overall, monolithic graph. The system we consider here has 32 days of data, so there are 32 different monolithic graphs that will be created, with the solutions of the previous day problems being fixed in the subsequent day graph (note that we only keep 3 graphs in memory at a time to reduce overall memory requirements). Each monolithic graph contained 641,709 total variables (41,727 binary), highlighting the inherent high dimensionality of the problem. Further details on the graph representation can be found in the Supporting Information.

We solve this problem using two different approaches. The first approach is to solve the overall "monolithic" graph as a single problem for each day. The second is a "receding horizon" approximation approach, where each subproblem subgraph is solved sequentially. In the receding horizon approach, we first solve the DA-UC subgraph, then the first ST-UC subgraph, then the 12 HA-ED subgraphs linked to the first ST-UC subgraph, then the second ST-UC subgraph, and so forth. This approach is represented in Figure 14, where each subgraph is represented by a node, and red arrows highlight information being shared to the next subproblem. These two approaches were repeated over all 32 days for which we have data, and both approaches used the same demand profiles in each layer for comparison. This highlights how the graph representation allows us to explore different ways of computing exact (when possible) or approximate solutions. We also note that the implementation of these approaches required slight modeling differences since, in the receding horizon approach, the linking constraints like those between layers had to be enforced on the subgraphs as well.

These approaches result in different overall solutions and solution times. We solved these problems with Gurobi with access to 20 threads on a 40-core Intel(R) Xeon(R) CPU E5-2698 v4 processor running @2.2 Ghz. The overall cost of economic dispatch (based on the first time point of each HA-ED subproblem; this is the realized performance) was $ 199.6 million and $ 222.1 million for the monolithic and receding horizon approaches, respectively (a difference of 10.8%). However, the monolithic

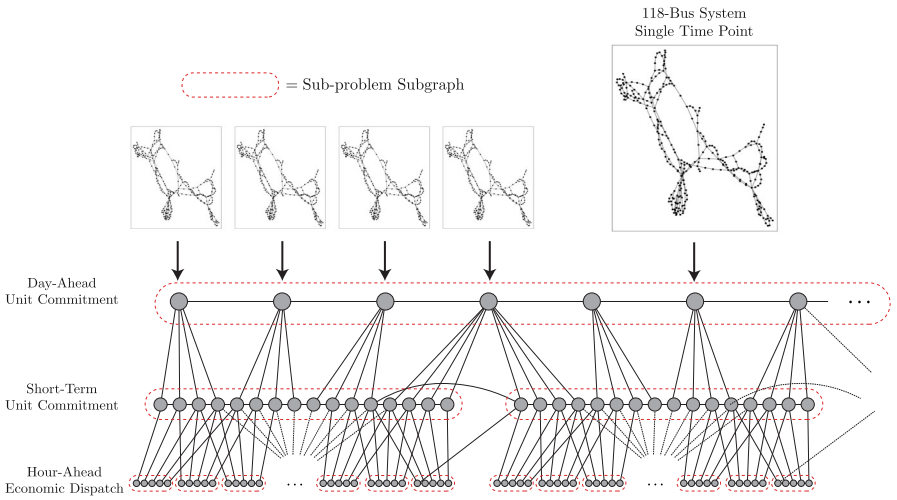


Fig. 13 Visualization of the hierarchical graph structure that is formed by the tri-level markets problem presented by Atakan and co-workers [6]. Here, subproblem subgraphs are formed for each subproblem in the DA-UC, ST-UC, and HA-ED layers and nested on an overall monolithic graph for each day of operation. Each "node" in this figure is a nested subgraph representing a single time point; this subgraph encodes the spatial structure of the 118-bus system as an OptiGraph, which buses and transmission lines represented by nodes. Edges are placed between the different subgraphs and across hierarhical layers and subproblems

approach took 48.7 hours to solve, compared to 7.2 hours for the receding horizon approximation. In addition, we used a tighter gap for the receding horizon DA-UC and ST-UC layers (0.5% or less, with a maximum time limitation) compared to a 1% gap on the monolithic problem (though using an even larger gap could reduce runtime). Further details on the solutions are included in the Supporting Information.

This case study highlights how the graphs facilitate hierarchical model building and enable different solution approaches. Often, the receding-horizon approach above is used because the monolithic problem is too large to solve in a reasonable time, because some information is not yet know (e.g., realized demand 12 hours from now), or a combination of both. The OptiGraph abstraction provides a framework in which to test and apply different approaches that operate within these types of constraints; for instance, a receding-horizon approach could be applied at an even higher resolution, where each time point subgraph is solved in series. Alternatively, problems could be partitioned in space (rather than their current time partitioning), or different numbers of hierarchical layers or different time scales could be applied, and the OptiGraph abstraction could help facilitate these analyses.

4.2 Case Study 2: Integration of planning and operations in power grid expansion

This problem highlights how gBD can be applied to a capacity expansion model (CEM) with both a planning level (long-term decisions) and an operations level (short-term decisions). CEMs are models of power systems that seek to minimize total system cost by considering investment and retirement (planning-level) decisions for power

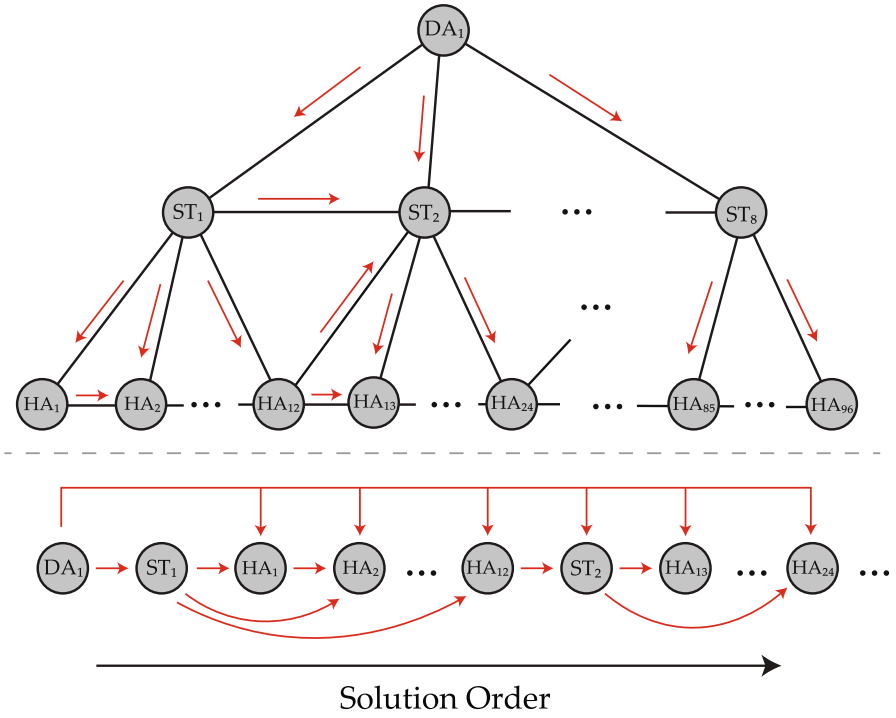


Fig. 14 Visualization of the receding horizon decomposition approach for the hierarchical energy markets case study. Each subproblem subgraph of Figure 13 are represented by a node, with DA_i, ST_i, and HA_i representing the i subproblem of the DA-UC, ST-UC, and HA-ED layers, respectively. Red arrows show the direction in which solutions are shared

systems. CEMs also often model higher resolution operation, such as regular (e.g., hourly) operation of generators (unit commitment and dispatch decisions), energy storage, power transmission, and flexible demands. This allows the CEM to capture the impact of planning level decisions on the day-to-day operation of the system and the resultant approximation of likely total system cost, permitting co-optimization of both fixed and variable costs. The problem we consider is implemented using the open-source software package, `GenX.jl` [48]. This implementation includes variables at the planning-level (including policy decisions and investment decisions) and variables at the operations-level, where the operations-level is split into multiple sub-periods, each constituting a series of linked operational steps (e.g. a n-day long periods of hourly sequential operations).

A compact form of the CEM (adpated from [45] and [73]) is given by

$$\min\ c_p^\top x_p + \sum_{w \in W_o} c_w^\top x_w \tag{19a}$$

$$\text{s.t. } A_w x_w + B_w x_p \leq b_w, \quad w \in W_o \tag{19b}$$

$$\sum_{w \in W_o} Q_w x_w \leq d \tag{19c}$$

$$x_w \geq 0, \quad w \in W_o \tag{19d}$$

$$x_p \in \mathcal{X}_p \tag{19e}$$

where W_o is the set of operations-level sub-periods, x_p is the set of variables on the planning level, x_w is the set of variables on the operations-level for the wth sub-period, $\mathcal{X}_p$ is a feasible region for the planning variables, and c_p, c_w, A_w, B_w, b_w, Q_w, and d are problem data (vectors and matrices). The objective (19a) defines total system cost, where c_p represent fixed costs (fixed operation costs and investment costs) and c_w represent variable costs (e.g., fuel prices, variable operation costs, penalties for non-served energy or constraint violations). The constraints (19b) include operational constraints that do not link sub-periods, such as constraints on power generation resources, transmission and ramping limits, and start-up/shut-down constraints. The constraints (19c) include policy constraints, such as CO_2 emission limits or renewable portfolio standards which couple operational subproblems.

We apply this formulation to an application in `GenX.jl` example systems. The specific application includes 11 operational sub-periods, each 168 hours long with hourly resolution. The problem models three zones in the northeastern United States with transmission allowed between zones. Each zone includes thermal generators, variable renewable energy generators (e.g., wind, solar PV), battery storage, and demands. The problem also includes policy constraints on CO_2 production, capacity reserve requirements, energy share requirements, and minimum and maximum capacity requirements. The final problem was an LP and included 138,972 variables and 166,612 constraints. This problem is not especially large but serves to highlight how gBD can be applied. Further, this model can scale rapidly, such as by considering additional operations sub-periods (e.g., weather years) or increasing the spatial resolution, and this same approach could be readily applied to a larger, scaled-up version of the problem where conventional monolithic solution methods struggle.

This problem can be represented as a graph by using a subgraph for the planning-level containing the variables x_p and $|W_o|$ subgraphs for the operations-level (one for each operations subperiod), each with variables x_w, $w \in W_o$. This approach results in (19b) being stored on edges connecting each operations-level subgraph and the planning-level subgraph. Importantly, the policy constraints (19c) form a hyper edge between all the operations-level subgraphs, meaning we do not form the tree structure required for gBD. To address this, budgeting variables $q_{p,w}$, $w \in W_o$ are added to the planning-level to capture the policy constraints for each operations sub-period; then, (19c) is replaced by $\sum_{w \in W_o} q_{p,w} \leq d$ which is stored on the planning-level subgraph, and the constraint $Q_w x_w \leq q_{p,w}$, $w \in W_o$ are added to the edges between each operations-level subgraph and the planning-level subgraph. This approach is similar to the lifting procedure shown in Figure 9 and was implemented in previous works with `GenX.jl` [45, 73]. The resulting graph structure can be seen in Figure 15, where the planning-level subgraph, $\mathcal{G}_1$, contained 86 variables (the complicating variables) and each operations subgraph contained 12,626 variables. Here, we will set the planning-level subgraph $\mathcal{G}_1$ as the root subgraph $\mathcal{G}_r$, as this results in a two-level

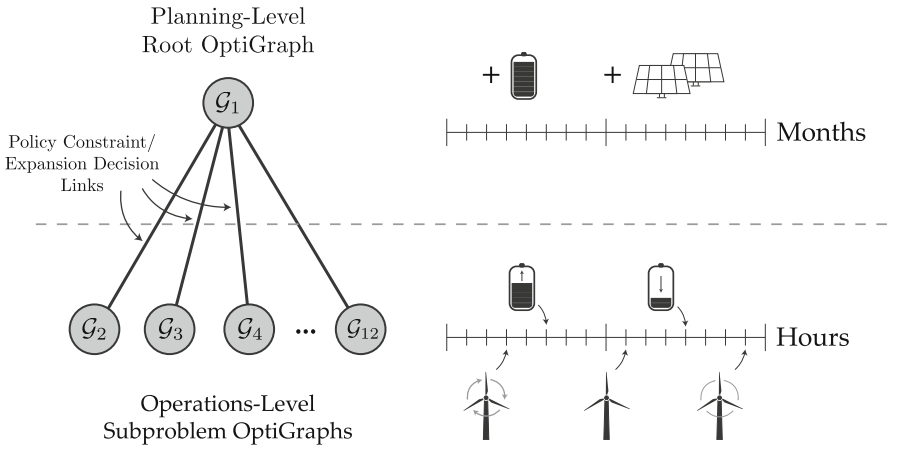


Fig. 15 Graph representation of CEM obtained from `GenX.jl`. The problem includes a planning-level containing expansion and retirement decisions and policy constraints which are contained in subgraph $\mathcal{G}_1$. The operations-level includes different subperiods containing operations variables and constraints. The planning level decisions can be considered as operating over a longer time scale (order of months) than the operations level (order of hours)

tree where the second stage is fully parallelizable (see [45]). Note however that, for the gBD algorithm, we could set any of the subgraphs as the root subgraph.

The problem modeled as a graph was solved in `PlasmoBenders.jl` using the gBD algorithm outlined in Algorithm 2. The planning level and each operational subproblem were each represented as a node and initially placed on separate subgraphs. We set $\mathcal{G}_1$ (containing the planning problem node) as the root subgraph (i.e., $\mathcal{G}_r = \mathcal{G}_1$) which results in two stages with multiple subproblems in the second stage, allowing for the use of the multi-cut approach. The convergence tolerance was set to 1×10^{-4}. We also solved using a regularization scheme outlined by [73] and which is implemented generally for the graph-based problem within `PlasmoBenders.jl` (see Supporting Information). In addition, `PlasmoBenders.jl` supports parallelizing the solution of each of the operations sub-periods to speed up overall solve time, and this functionality was used for this problem. Finally, as a third option, we solved the problem by placing two (the first and the last) of the operational subproblem nodes into a subgraph with the planning level node as shown in Figure 16. This results in a larger root problem, but one which is "better informed" by knowledge of some subproblems [31]. The results of these three approaches can be seen in Figure 17.

We note as well a challenge that arose in this problem in that the exact number of iterations was not always consistent between different runs. This inconsistency is likely due to degenerate solutions in the root problem due to policy constraints (e.g., the CO_2 budgeting variables for the maximum allowable emissions in a subproblem do not show up in the root problem objective function, meaning different CO_2 budgets can be chosen with the same root problem objective but different subproblem performance at an early iteration). Consequently, we ran each problem 10 times and recorded the number of iterations used, with the results in Figure 17 being the run with the fifth fewest iterations. These results are all included in the repository, but the three different

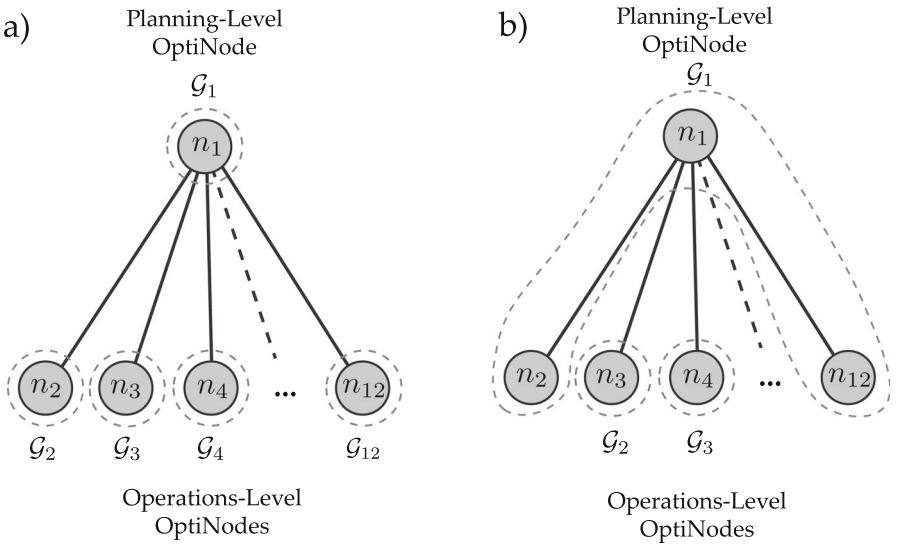


Fig. 16 The two different partitioning approaches used for the capacity expansion model. a) shows the original partitioning with a single OptiNode on each subgraph, where there is one planning level optinode and 11 operations level OptiNodes, each representing a different subperiod. b) shows an alternative partitioning where two of the operational OptiNodes are placed with the planning-level OptiNode on the root subgraph

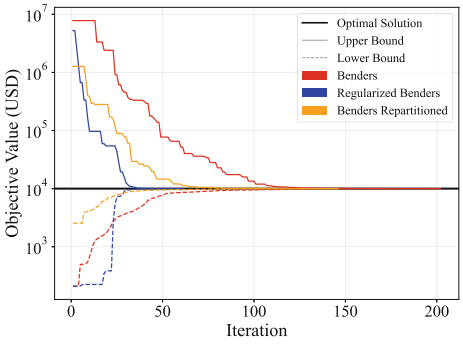


Fig. 17 Solution of (19) with gBD using `PlasmoBenders.jl`. The problem is solved without a regularization scheme and each subperiod on its own subproblem subgraph ("Benders"), with a regularization scheme and each subperiod on its own subproblem subgraph ("Regularized Benders"), and with two subperiods included in the root/master problem and without a regularization scheme ("Benders (repartitioned)")

versions (without regularization, with regularization, and the repartitioned problem with two subperiods included in the root subgraph without regularization) took on average 202, 61, and 139 iterations, respectively, with overall ranges of 149-271, 57-65, and 109-158 iterations.

Overall, `PlasmoBenders.jl` was able to solve (19) using the gBD algorithm (Algorithm 2) and reached the required tolerance between the upper and lower bounds, serving as a proof-of-concept for the graph-based approach. In addition,

`PlasmoBenders.jl` made it trivial to toggle on/off the use of a regularization strategy (by changing a single settings parameter), which substantially reduced the required number of iterations for convergence. This regularization strategy is implemented in a general manner in `PlasmoBenders.jl`, such that it can easily be applied to other BD problems. `Plasmo.jl` also readily supports altering the graph structure, such as moving operational subproblems into the root problem. Doing so for this case study resulted in overall fewer iterations, though the root problem solve time did increase. Under a bespoke implementation of BD, making an alteration to the root problem like this would be non-trivial, but it can be done with just a few lines of code because of `Plasmo.jl`'s flexibility and the gBD algorithm being based solely off of graph structrue. This highlights how the graph approach provides a framework for developing and implementing general algorithms and simplifies modeling and implementation for users.

While this case study has focused on a CEM, many problems can have a similar structure to the problem above. BD has been used in many applications, and the hierarchical or multi-scale nature of this problem like that seen in Figure 15 could arise in a variety of areas (see for instance [47, 53, 62, 94, 100]). Thus, the approach used herein could therefore be applied to other problems or disciplines as well.

4.3 Case Study 3: Production cost modeling

This study illustrates how gBD can be used to solve an electricity system production cost model (PCM). PCMs are routinely used in power systems to determine the cost of operating a power grid over a given timeframe (typically the next 24 hours) [49, 56] by considering the dispatch and operation of a set of generators (e.g., fossil fuel, wind, solar) to meet demand in a region. These models also frequently include a degree of "look-ahead" which allows the model to consider expected future demand and avoid myopic decisions (e.g., the model may choose to leave some generators turned on to meet expected demand arising after the next 24 hour period). PCMs will often then "step-forward" in time and simulate multiple consecutive timeframes (e.g., a model with a 24 hr day-ahead period and a 24 hr look-ahead period will step forward 365 times to simulate yearly operation). These dispatch and operation decisions are subject to several constraints, such as up/down time and ramping constraints.

The hierarchy in this problem arises due to the inclusion of a long-duration energy storage (LDES) device in the system. LDES devices are important for high-renewable grids [17, 33, 38], but they complicate the operation of the grid (i.e., the solution of PCMs) because they play the role of shifting power across larger time scales so capturing their true value can require looking farther into the future (see Figure 18a). Unfortunately, adding longer look-ahead horizons to PCMs can increase required computational time and resources. For instance, many PCMs operate with a total horizon of 48 hours (a 24-hour day ahead period with an additional 24 hours of "look-ahead"), but PCMs with LDES can require using look-ahead horizons of 3 or more days which can lead to an exponential increase in required solution time [40]. In this way, the presence of the LDES device creates a hierarchy in decision time scales: the PCM operation on the order of hours and the LDES operation on the

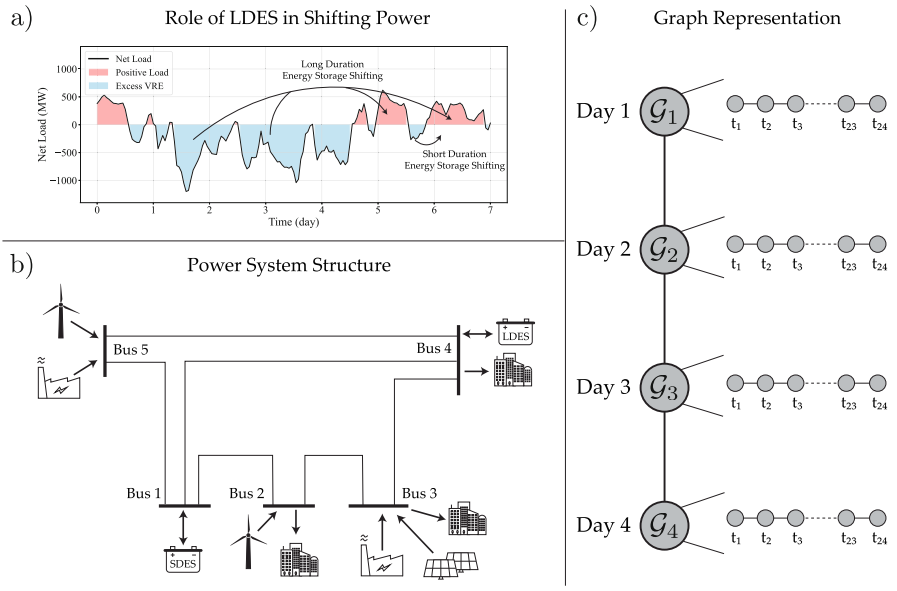


Fig. 18 A visualization highlighting PCMs and how they can be represented as a graph. LDES devices complicate PCM operation because they play the role of shifting power across longer time scales (a) for a power system like the one shown in (b). We can represent these problems as a graph (c), where each subgraph represents a period of time (forming a coarse time resolution among the subgraphs). Each time period is effectively a hierarchical decision that impacts the subsequent time period, resulting in the linear graph structure shown. Adapted from [26]

order of days. We capture this hierarchy by representing a series of time points as subgraphs (e.g., representing a sequence of hours or days as a single subproblem) that are connected to the next series of time points, with the battery state of charge being linked across the subgraphs; in other words, one could consider that the subgraphs are representing the time scale on which the LDES operates, and each subgraph contains a finer resolution problem on which the PCM operates. This creates the linear graph hierarchical structure seen in Figure 7. An example of this hierarchical structure is presented in Figure 18c, which is applied to a system such as that of Figure 18b.

The problems we consider are implemented in the `Julia` package `PowerSimulations.jl` [56] and converted into a `Plasmo.jl` OptiGraph. `PowerSimulations.jl` constructs an optimization model for a power system given by the user. Based on the power system data and components (e.g., generators, loads, transmission lines), `PowerSimulations.jl` defines the needed objective function, decision variables, and constraints to represent the PCM of the system. `PowerSimulations.jl` also provides various options for constraints and variables used for modeling components (for instance, a thermal generator can be modeled in several ways). Further details of `PowerSimulations.jl` can be found in the package documentation and in [56].

The general formulation for the PCMs we used in this case study is described below. As the purpose of this work is to highlight how graphs can be used for repre-

senting and decomposing hierarchical problems, we will not introduce every variable and constraint and instead only give a high-level overview. The PCMs we consider are MILPs that seek to minimize cost, including production and start-up/shut-down costs, while also considering penalties on slack variables (which ensure feasibility, such as by allowing load shedding). Here, production costs apply to thermal generators (e.g., coal, gas, nuclear) and consider the cost of producing energy with any given generator. Discrete startup-shutdown decisions apply only to thermal generators and incur costs associated with turning on or shutting down a generator. The PCMs are subject to constraints including generator ramp up/down, generator minimum up/down time constraints after start-up/shut-down, storage operating constraints, reserve requirements, transmission constraints, and overall power balance constraints. Transmission constraints included constraints on power flow throughout the system using a linearized DC approximation of power flow. There are also constraints on the battery operation, with the state of charge of the battery linked across every time point. The full sets of constraints are dependent in part on the system data; for instance, `PowerSimulations.jl` is built on the `PowerSystems.jl` data structure, which supports different types of thermal generators, including standard generators and multi-start generators, so the exact problem formulation is dependent on the system defined.

The examples below include two different systems: a 5-bus system and the 73-bus Reliability Test System (RTS; adapted from [59] and [37], respectively). Details of these systems can be found at https://github.com/NatLabRockies/LDES-Sizing-and-siting-model. The 5-bus system included five buses, six transmission lines, two thermal generators, one solar generator, two wind generators, and three loads. The RTS included 73 buses, 120 transmission lines and tap transformers, 54 thermal generators, 58 solar generators, five wind generators, and 51 loads. Each system included a short-duration (3.6 hr) and long-duration (13.5 hr) energy storage device. The presence of LDES in the system requires using longer look-ahead horizons in the PCM simulation.

The `PowerSimulations.jl` model can be represented as a graph by using nodes to represent buses and arcs at each time point (arcs are `PowerSystems.jl` objects that can contain lines or tap transformers; for our purposes, they are containers to hold transmission lines). Each generator, storage device, and load are contained on a specific bus, and each transmission line is contained on an arc. Thus, each node corresponding to a bus will contain variables for any generator, storage device, or load that are contained on that bus (e.g., active power variables, on variables). Each node corresponding to an arc will contain variables for any transmission line on that arc. As with Case Study 1, each node corresponds to a single point in time, such that for a system with N_B buses, N_A arcs, and N_T time points, there will be $(N_B + N_A) \times N_T$ total nodes. Edges are placed between nodes based on constraints in the `PowerSimulations.jl` model. For instance, edges are placed between nodes for an arc and two buses if that arc has a line that transports power between those two nodes. Edges are also placed across time, such that nodes at different time points are connected; for instance, a node representing a bus containing a generator with up-time limits will be linked to next and previous time points (and if the up-time limit is greater than 1, it will be linked to multiple nodes in both forward and backward directions).

The 5-bus system and RTS used in this case study can both be represented as a graph as explained above, and visualizations of these systems are given in Figures 19 and 20. Figures 19a and 20a show the basic system structure with each bus represented by a node and arcs (containing transmission lines) represented by edges (this is the most intuitive graph representation and is not the OptiGraph). Figures 19b and 20b show the OptiGraph representation for a single time point using the methods outlined above. For the single time point graph, the edges are linking constraints which capture flow constraints as well as overall system energy balance constraints (which includes all nodes containing generator variables; each bus also contains a local balance for power flowing in and out of the node). Figures 19c, 20c, and 20d show the system expanded across time, where the linking constraints now also include the state of charge of the battery, ramping constraints, and up- and down-time constraints. Note here that Figure 20c shows the RTS with 48 time points and the down-time constraint can best be seen in this figure: two generators in this system have down times of 48-hours, and the central "cloud" in the middle of this graph are the down-time constraints, where each of the nodes containing these generator variables are linked to each other since the down-time limit is equal to the time span the entire graph. Importantly, none of these expansions results in the required structures for gBD. Figures 19c and 20d show an OptiGraph with a longer horizon (24-hours and 15-days for the 5-bus system and RTS, respectively), which can then be partitioned across time to form the subgraph structure shown in the bottom of the subfigure. This resulting subgraph structure can be solved using gBD.

To highlight how gBD can be applied to graph-based problems, we solved an instance of the 5-bus system and the RTS using `PlasmoBenders`' implementation of gBD (see Algorithm 2) with the backwards pass parallelized (i.e., when solving stage i of iteration k in the backwards pass, we do not wait for the cut from stage $i+1$ and add cuts to all problems at once when all problems have finished solving). Each system included 1-hour time steps. The 5-bus system used a 90-day total horizon (2160 time points) and the RTS used a 15-day total horizon (360 time points). The 5-bus system included 127,450 variables (17,280 binary) and 157,680 constraints while the RTS included 406,655 variables (59,040 binary) and 360,360 constraints (constraints do not include variable bounds). Each system was structured as a graph, and the convergence tolerance was set to be 0.5%. To create the structure required for gBD, we must partition the nodes into subgraphs; the partitions used are arbitrary, so long as they result in a tree structure in the subgraphs (see for instance the bottom graph in Figures 19c and 20d). We chose to test different partition sizes and report the results for the gBD solution algorithm in Figure 21. For the 5-bus system, we used 3, 5, and 10 partitions (subgraphs), where each subgraph of the partition represented consecutive time points (i.e., partition sizes in the three cases were 30 days, 15 days, and 9 days, respectively). For the RTS system, we used 3, 5, and 8 partitions (i.e., partition sizes were 5, 3, and 2 days, respectively; note that the final partition in the 8-partition case only contained 1 day). Each of these partitioning decisions resulted in a linear graph, and we set the subgraph containing the first time point as the root subgraph in each case (e.g., for the subgraph structure in Figure 20d, we set $\mathcal{G}_1$ as the root subgraph). To highlight how the decision of root subgraph can impact the algorithm, we also tested two different root subgraphs for the 3-partition case of the

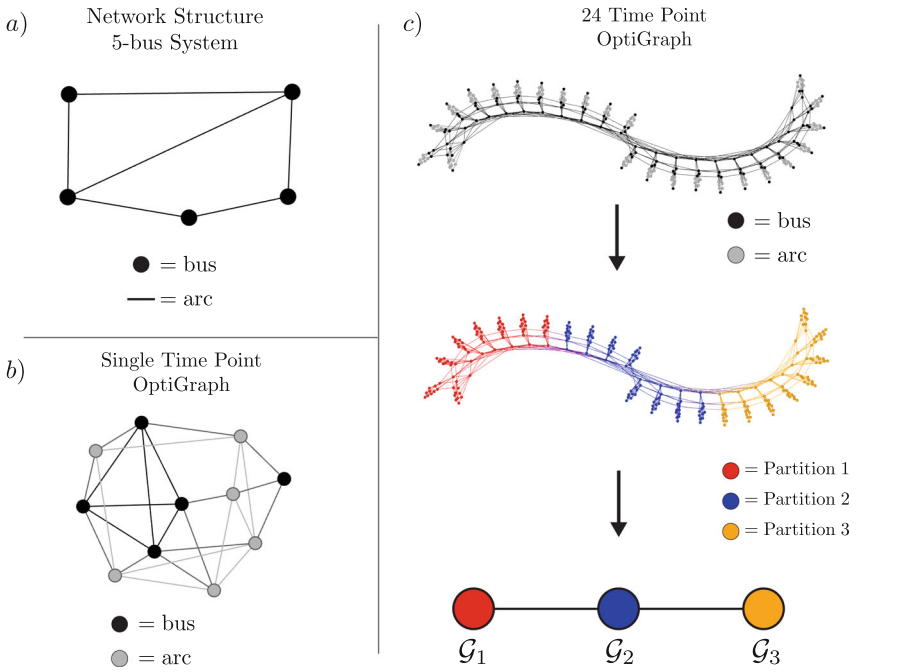


Fig. 19 Example of the 5-bus system used in Case Study 3, represented as a graph, including a) the basic network structure with buses represented by a node and system arcs (transmission lines) represented by edges; b) a single time point OptiGraph with buses and arcs represented by nodes (each node can contain variables and constraints); c) a 24 time point OptiGraph where each time point is also linked across time (e.g., via up/down time constraints, storage device state of charge constraints). This results in a graph with many cycles, but the graph can be partitioned to form the subgraph structure shown on the bottom, to which gBD can be applied. Adapted from [26]

5-bus system by solving with a root subgraph containing the initial time point of the PCM ($\mathscr{G}_r = \mathscr{G}_1$) and the "middle" subgraph (e.g., $\mathscr{G}_2$ in Figure 19c; additional slack variables had to be added to this problem to ensure recourse, but the final solution was equal to that of the problems without these slack variables).

For both the 5-bus system and RTS, the gBD algorithm in `PlasmoBenders` was able to get to a high quality solution, though partitioning decisions impacted convergence (Figure 21; see Tables 1 and 2 in the Supporting Information for CPU time and memory usage). For the 5-bus system, the 3 subgraph problem converged in only 3 iterations, while the problem with 5 subgraphs required 7 iterations. With 10 subgraphs, the problem was not able to close the gap beyond a gap of 1.7% (which it reached after 17 iterations) even if it ran for more than 50 iterations. Interestingly, using the middle subgraph as the root subgraph in the 3 subgraph case was able to converge in only 2 iterations. For the RTS, the problem did not converge to the 0.5% tolerance for any of the partitionings within the set K_{max} iterations (for each partitioning, the gap was still closing very slowly at termination). The 3, 5, and 8 subgraph problems only reached a gap of 0.7%, 2.1%, and 4.7% (for K_{max} of 75), respectively. This may be due in part to the presence of a duality gap in the mixed integer problem.

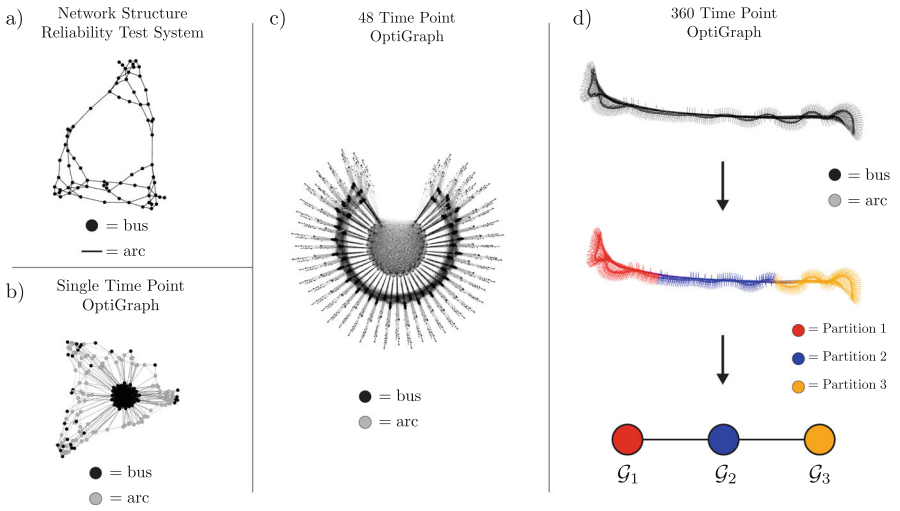


Fig. 20 Example of the RTS used in Case Study 3, represented as a graph, including a) the basic network structure with buses represented by a node and system arcs (transmission lines) represented by edges; b) a single time point OptiGraph with buses and arcs represented by nodes (each node can contain variables and constraints); c) a 48 time point OptiGraph where each time point is also linked across time (e.g., via up/down time constraints, storage device state of charge constraints); d) a 360 time point OptiGraph with more than 65,000 nodes, highlighting the complexity that can be present in these types of problems. This graph contains many cycles, but can be partitioned to form the subgraph structure shown on the bottom, to which gBD can be applied

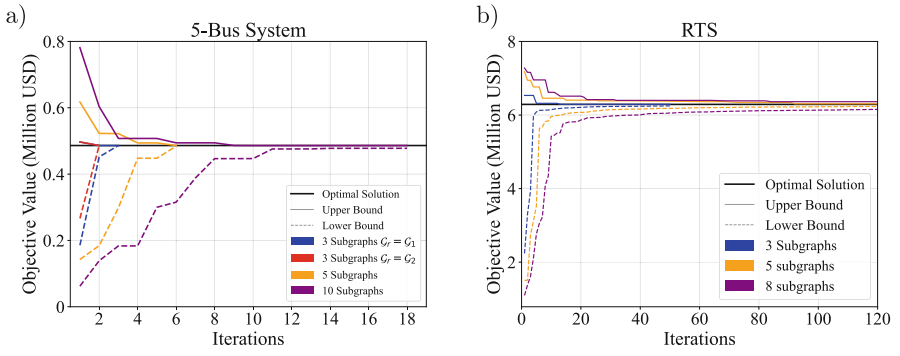


Fig. 21 Results for the gBD approach for a) the 5-bus system with a 90-day time horizon using 3, 5, and 10 partitions, and b) the RTS with a 15-day time horizon using 3, 5, and 8 partitions. gBD was applied by setting the first subgraph (containing the first time point of the model) to be the root subgraph. For the 5-bus system, we also tested gBD using both $\mathcal{G}_1$ and $\mathcal{G}_2$ as the root subgraphs

Overall, the graph-based approach was effective at constructing problems that can be decomposed by gBD. Representing these problems under the OptiGraph abstraction can give flexibility in applying different graph-based decomposition schemes (such as enabling different numbers of partitions or selection of alternative root subgraphs as shown above) and could lead to non-intuitive applications of algorithms. For instance, changing the root subgraph of the 3 subgraph case of the 5-bus system was able

to converge effectively. Temporal decompositions like those above are not typically solved in this way, but by representing these problems as graphs, we can identify alternative solution approaches.

5 Conclusions and future work

We have presented a graph abstraction for modeling and solving optimization problems that contain hierarchical optimization problems. The OptiGraph abstraction, implemented in `Plasmo.jl`, allows for constructing hierarchical subgraphs to capture hierarchical relationships in optimization problems. The OptiGraph abstraction also gives significant flexibility by enabling partitioning and aggregation of graph structures. We have shown how the OptiGraph can can be used as a foundational abstraction for developing generalized solution algorithms and for implementing approximation schemes. We have given an example of generalizing algorithms to a graph with graph-based Benders decomposition, gBD, which is implemented in the package `PlasmoBenders`. By using this graphical approach, it is possible to identify alternative non-intuitive applications of algorithms or flexibly reconfigure problem structure to improve convergence properties or better leverage available computing resources (e.g., parallel computing hardware). We have provided different case studies showing how the OptiGraph abstraction can be used for modeling and solving hierarchical problems.

There are several areas we would like to explore in future work. There are possible extensions of gBD that can be explored. For instance, there have been different works presenting parallel implementations of nested BD [28, 78, 81, 82], such as by solving subproblems asynchronously. Similar extensions could be applied to graphs, or new algorithms could be developed. For example, if the linear graph in Figure 7 could be partitioned such that all odd numbered subgraphs are in one partition and all even numbered subgraphs in another partition (this would form a bipartite graph); this would be the structure required for the traditional BD application, but within each subgraph, there would be two or three additional subgraphs with no original connections to other subgraphs in their partition. It is possible that, since there are no connections of subgraphs within the two partitions, a parallel algorithm could be more readily developed. We would also like to improve the gBD implementation in `PlasmoBenders.jl` to parallelize the solution within stages (currently, this is only supported for problems with two stages), and we would like to implement feasibility cuts for multi-stage problems. We are also interested in considering how the structuring decisions impact the strength of cutting planes. As an example, for the linear graph in Figure 7, by changing the root subgraph from $\mathcal{G}_1$ to $\mathcal{G}_3$, we also could change the number of cutting planes added to $\mathcal{G}_3$. When $\mathcal{G}_r = \mathcal{G}_1, \mathcal{G}_3$ would only have cutting planes from the solution of $\mathcal{G}_4$, but when $\mathcal{G}_r = \mathcal{G}_3$, then $\mathcal{G}_3$ would also receive cutting planes from the solution of $\mathcal{G}_2$. Understanding how to choose the root subgraph and how these cutting planes impact the solution could be informative. Fourth, we would like to extend gBD to nonlinear problems, such as by following generalized BD frameworks [36]. We would also like to explore how regularization schemes could be applied within gBD for problems with many stages as `PlasmoBenders.jl` currently only applies regularization to two

stage problems. Finally, we are interested in adding functionality for different types of cutting planes or other advancements (such as the adaptive oracles of [101]) within `PlasmoBenders.jl`.

At present, `Plasmo.jl` does not readily facilitate distributing memory with the OptiGraphs, but this can be important for large-scale decomposition problems on high-performance compute clusters or for cloud-based computing. We would like to develop a distributed OptiGraph abstraction for distributing memory and solution of large-scale problems. Second, we would like to consider different applications of the hierarchical graphs; one such application would be stochastic programming, and in particular, multi-horizon stochastic programming (MHSP) [51, 100, 101]. In MHSP, these problems are frequently visualized as graphs and decomposition approaches like BD can be required for solving them. Third, we are interested in implementing additional decomposition schemes for OptiGraphs, such as developing a graph-based progressive hedging, ADMM. or other Lagrangian decompositions. Finally, these hierarchical graphs could facilitate including surrogate models into an optimization problem (such as replacing a single subgraph or a hierarchical layer with a neural network). Surrogate models have been embedded in optimization problems in other works [24, 43, 93], but the hierarchical subgraphs could provide a powerful tool for including these kinds of models.

Acknowledgements The authors thank Dr. Jordan Jalving for his technical support and continued maintenance of `Plasmo.jl`.

Availability of data and materials All data and code used in this work are publicly available. URLs are included in this published article.

Declarations

Conflicts of Interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abaecherli, M.L., Gonzáles, D.S., Perren, S., Capón-García, E., Szijjarto, A., Hungerbühler, K.: Hierarchical waste incineration planning and scheduling system for industrial operation support. In: Computer Aided Chemical Engineering, vol. 40, pp. 1327–1332. Elsevier (2017)
2. Abujarad, S.Y., Mustafa, M.W., Jamian, J.J.: Recent approaches of unit commitment in the presence of intermittent renewable energy resources: A review. Renew. Sustain. Energy Rev. **70**, 215–223 (2017)
3. Albornoz, V.M., Cid-García, N.M., Ortega, R., Ríos-Solís, Y.A.: A Hierarchical Planning Scheme Based on Precision Agriculture, pp. 129–162. Springer New York, New York, NY (2015)
4. Allman, A., Tang, W., Daoutidis, P.: Decode: a community-based algorithm for generating high-quality decompositions of optimization problems. Optim. Eng. **20**, 1067–1084 (2019)

5. Anandalingam, G., Friesz, T.L.: Hierarchical optimization: An introduction. Ann. Oper. Res. **34**, 1–11 (1992)
6. Atakan, S., Gangammanavar, H., Sen, S.: Towards a sustainable power grid: Stochastic hierarchical planning for high renewable integration. Eur. J. Oper. Res. **302**(1), 381–391 (2022)
7. Ávila, D., Papavasiliou, A., Löhndorf, N.: Parallel and distributed computing for stochastic dual dynamic programming. CMS **19**(2), 199–226 (2022)
8. Behrunani, V.N., Cai, H., Heer, P., Smith, R.S., Lygeros, J.: Distributed multi-horizon model predictive control for network of energy hubs. Control. Eng. Pract. **147**, 105922 (2024)
9. Bennett, K.P., Ferris, M.C., Pang, J.S., Solodov, M.V., Wright, S.J.: Hierarchical optimization. Math. Program. **198**(2), 1121–1123 (2023)
10. van de Berg, D., Shah, N., del Rio-Chanona, E.A.: Hierarchical planning-scheduling-control–optimality surrogates and derivative-free optimization. Computers & Chemical Engineering p. 108726 (2024)
11. Bertels, S., Fahle, T.: A hybrid setup for a hybrid scenario: combining heuristics for the home health care problem. Computers & Operations Research **33**(10), 2866–2890 (2006)
12. Bichot, C.E., Siarry, P.: Graph partitioning. John Wiley & Sons (2013)
13. Multi-scale optimization for process systems engineering: Biegler, L.T., Lang, Y.d., Lin, W. Computers & Chemical Engineering **60**, 17–30 (2014)
14. Birge, J.R.: Decomposition and partitioning methods for multistage stochastic linear programs. Oper. Res. **33**(5), 989–1007 (1985)
15. Birge, J.R., Louveaux, F.: Introduction to stochastic programming. Springer Science & Business Media (1997)
16. Birge, J.R., Louveaux, F.V.: A multicut algorithm for two-stage stochastic linear programs. Eur. J. Oper. Res. **34**(3), 384–392 (1988)
17. Blair, N., Augustine, C., Cole, W., Denholm, P., Frazier, W., Geocaris, M., Jorgenson, J., McCabe, K., Podkaminer, K., Prasanna, A., et al.: Storage futures study: Key learnings for the coming decades. Tech. rep., National Renewable Energy Lab.(NREL), Golden, CO (United States) (2022). NREL/TP-7A40-81779
18. BnnoBRs, J.: Partitioning procedures for solving mixed-variables programming problems. Numer. Math. **4**(1), 238–252 (1962)
19. Bosák, J.: Induced subgraphs. In: Finite and Infinite Sets, pp. 109–118. Elsevier (1984)
20. Bracken, J., McGill, J.T.: Mathematical programs with optimization problems in the constraints. Oper. Res. **21**(1), 37–44 (1973)
21. Braun, P., Faulwasser, T., Grüne, L., Kellett, C.M., Weller, S.R., Worthmann, K.: Hierarchical distributed admm for predictive control with applications in power networks. IFAC Journal of Systems and Control **3**, 10–22 (2018)
22. Brunaud, B.: Models and algorithms for multilevel supply chain optimization. Ph.D. thesis, Carnegie Mellon University (2019)
23. Bundy, A., Wallen, L.: Breadth-first search. Catalogue of artificial intelligence tools pp. 13–13 (1984)
24. Ceccon, F., Jalving, J., Haddad, J., Thebelt, A., Tsay, C., Laird, C.D., Misener, R.: Omlt: Optimization & machine learning toolkit. J. Mach. Learn. Res. **23**(349), 1–8 (2022)
25. Chaieb, M., Jemai, J., Mellouli, K.: A hierarchical decomposition framework for modeling combinatorial optimization problems. Procedia Computer Science **60**, 478–487 (2015)
26. Cole, D.L., Dalvi, S., Zavala, V.M., Guerra, O.J.: Toward understanding the complexity of long-duration energy storage siting in high renewable power grids. iScience **28**(6) (2025)
27. Cole, D.L., Shin, S., Zavala, V.M.: A julia framework for graph-structured nonlinear optimization. Industrial & Engineering Chemistry Research **61**(26), 9366–9380 (2022)
28. Colonetti, B., Finardi, E., Brito, S., Zavala, V.: Parallel dual dynamic integer programming for large-scale hydrothermal unit-commitment. IEEE Trans. Power Syst. **38**(3), 2926–2938 (2022)
29. Conejo, A.J., Baringo, L.: Power system operations, vol. 14. Springer (2018)
30. Conejo, A.J., Castillo, E., Minguez, R., Garcia-Bertrand, R.: Decomposition techniques in mathematical programming: engineering and science applications. Springer Science & Business Media (2006)
31. Constante-Flores, G.E., Conejo, A.J.: Relaxation and decomposition. In: Optimization via Relaxation and Decomposition: Applications to Large-Scale Engineering Problems, pp. 1–10. Springer (2025)
32. Daoutidis, P., Tang, W., Allman, A.: Decomposition of control and optimization problems by network structure: Concepts, methods, and inspirations from biology. AIChE Journal **65**(10) (2019)

33. Denholm, P., Cole, W., Blair, N.: Moving beyond 4-hour li-ion batteries: challenges and opportunities for long(er)-duration energy storage. Tech. rep., National Renewable Energy Laboratory (NREL), Golden, CO (United States) (2023). NREL/TP-6A40-85878
34. Dowson, O., Kapelevich, L.: Sddp. jl: a julia package for stochastic dual dynamic programming. INFORMS Journal on Computing **33**(1), 27–33 (2021)
35. Fikar, C., Hirsch, P.: Home health care routing and scheduling: A review. Computers & Operations Research **77**, 86–95 (2017)
36. Geoffrion, A.M.: Generalized benders decomposition. J. Optim. Theory Appl. **10**, 237–260 (1972)
37. González-Fernández, R.A., da Silva, A.M.L., Resende, L.C., Schilling, M.T.: Composite systems reliability evaluation based on monte carlo simulation and cross-entropy methods. IEEE Trans. Power Syst. **28**(4), 4598–4606 (2013)
38. Guerra, O.J.: Beyond short-duration energy storage. Nat. Energy **6**(5), 460–461 (2021)
39. Guerra, O.J., Calderón, A.J., Papageorgiou, L.G., Reklaitis, G.V.: Integrated shale gas supply chain design and water management under uncertainty. AIChE J. **65**(3), 924–936 (2019)
40. Guerra, O.J., Dalvi, S., Thatte, A., Cowiestoll, B., Jorgenson, J., Hodge, B.M.: Towards robust and scalable dispatch modeling of long-duration energy storage. Renew. Sustain. Energy Rev. **207**, 114940 (2025)
41. Guo, F., Wen, C., Mao, J., Chen, J., Song, Y.D.: Hierarchical decentralized optimization architecture for economic dispatch: A new approach for large-scale power system. IEEE Trans. Industr. Inf. **14**(2), 523–534 (2017)
42. Hansknecht, C., Heinzelreiter, B., Pearson, J.W., Potschka, A.: A framework for the solution of tree-coupled saddle-point systems. arXiv preprint arXiv:2410.23385 (2024)
43. Henao, C.A., Maravelias, C.T.: Surrogate-based superstructure optimization framework. AIChE J. **57**(5), 1216–1232 (2011)
44. Ishikawa, S., Horio, K., Kubota, R.: Effective hierarchical optimization using integration of solution spaces and its application to multiple vehicle routing problem. In: 2015 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), pp. 406–411. IEEE (2015)
45. Jacobson, A., Pecci, F., Sepulveda, N., Xu, Q., Jenkins, J.: A computationally efficient benders decomposition for energy systems planning problems with detailed operations and time-coupling constraints. INFORMS Journal on Optimization **6**(1), 32–45 (2024)
46. Jalving, J., Shin, S., Zavala, V.M.: A graph-based modeling abstraction for optimization: Concepts and implementation in Plasmo.jl. Mathematical Programming Computation **14**(4), 699–747 (2022)
47. Jemai, J., Chaieb, M., Mellouli, K.: The home care scheduling problem: A modeling and solving issue. In: 2013 5th International Conference on Modeling, Simulation and Applied Optimization (ICMSAO), pp. 1–6. IEEE (2013)
48. Jenkins, J.D., Sepulveda, N.A.: Enhanced decision support for a changing electricity landscape: the genx configurable electricity resource capacity expansion model. MIT Energy Initiative (2017). Https://hdl.handle.net/1721.1/130589. Accessed 27 Sep 2024
49. Kahn, E.: Regulation by simulation: The role of production cost models in electricity planning and pricing. Oper. Res. **43**(3), 388–398 (1995)
50. Karypis, G., Kumar, V.: Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (1997). Retrieved from University Digital Conservancy, https://hdl.handle.net/11299/215346. Technical Report 97-061
51. Kaut, M., Midthun, K.T., Werner, A.S., Tomasgard, A., Hellemo, L., Fodstad, M.: Multi-horizon stochastic programming. CMS **11**(1), 179–193 (2014)
52. Khaki, B., Chu, C., Gadh, R.: A hierarchical admm based framework for ev charging scheduling. In: 2018 IEEE/PES Transmission and Distribution Conference and Exposition (T&D), pp. 1–9. IEEE (2018)
53. Kong, C., Jovanovic, R., Bayram, I.S., Devetsikiotis, M.: A hierarchical optimization model for a network of electric vehicle charging stations. Energies **10**(5), 675 (2017)
54. Kumar, R., Wenzel, M.J., ElBsat, M.N., Risbeck, M.J., Drees, K.H., Zavala, V.M.: Dual dynamic programming for multi-scale mixed-integer mpc. Computers & Chemical Engineering **148**, 107265 (2021)
55. Lara, C.L., Mallapragada, D.S., Papageorgiou, D.J., Venkatesh, A., Grossmann, I.E.: Deterministic electric power infrastructure planning: Mixed-integer programming model and nested decomposition algorithm. Eur. J. Oper. Res. **271**(3), 1037–1054 (2018)

56. Lara, J.D., Barrows, C., Thom, D., Dalvi, S., Callaway, D.S., Krishnamurthy, D.: Powersimulations.jl–a power systems operations simulation library. arXiv preprint arXiv:2404.03074 (2024)
57. Lee, C.Y.: An algorithm for path connections and its applications. IRE Trans. Electron. Comput. **3**, 346–365 (1961)
58. Lemaréchal, C., Nemirovskii, A., Nesterov, Y.: New variants of bundle methods. Math. Program. **69**, 111–147 (1995)
59. Li, H., Sun, J., Tesfatsion, L.: Testing institutional arrangements via agent-based modeling: A us electricity market application. In: Computational methods in economic dynamics, pp. 135–158. Springer (2011)
60. Li, Z., Ierapetritou, M.G.: Production planning and scheduling integration through augmented lagrangian optimization. Computers & Chemical Engineering **34**(6), 996–1006 (2010)
61. Liu, M., Liu, Z., Chu, F., Dolgui, A., Chu, C., Zheng, F.: An optimization approach for multi-echelon supply chain viability with disruption risk minimization. Omega **112**, 102683 (2022)
62. Liu, T., Yuan, Q., Wang, Y.: Hierarchical optimization control based on crop growth model for greenhouse light environment. Comput. Electron. Agric. **180**, 105854 (2021)
63. Long, Y., Liu, S., Xie, L., Johansson, K.H.: A hierarchical distributed mpc for hvac systems. In: 2016 American Control Conference (ACC), pp. 2385–2390. IEEE (2016)
64. Lubin, M., Dowson, O., Dias Garcia, J., Huchette, J., Legat, B., Vielma, J.P.: JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. Mathematical Programming Computation (2023). https://doi.org/10.1007/s12532-023-00239-3
65. Marcucci, T.: Graphs of convex sets with applications to optimal control and motion planning. Ph.D. thesis, Massachusetts Institute of Technology (2024)
66. Marcucci, T., Umenberger, J., Parrilo, P., Tedrake, R.: Shortest paths in graphs of convex sets. SIAM J. Optim. **34**(1), 507–532 (2024)
67. Mitrai, I., Daoutidis, P.: Decomposition of integrated scheduling and dynamic optimization problems using community detection. J. Process Control **90**, 63–74 (2020)
68. Mitrai, I., Daoutidis, P.: Efficient solution of enterprise-wide optimization problems using nested stochastic blockmodeling. Industrial & Engineering Chemistry Research **60**(40), 14476–14494 (2021)
69. Mitrai, I., Daoutidis, P.: A graph classification approach to determine when to decompose optimization problems. In: Computer Aided Chemical Engineering, vol. 52, pp. 655–660. Elsevier (2023)
70. Mitrai, I., Tang, W., Daoutidis, P.: Stochastic blockmodeling for learning the structure of optimization problems. AIChE J. **68**(6), e17415 (2022)
71. Morozov, S., Marcucci, T., Graesdal, B.P., Amice, A., Parrilo, P.A., Tedrake, R.: Mixed discrete and continuous planning using shortest walks in graphs of convex sets. arXiv preprint arXiv:2507.10878 (2025)
72. Ocampo-Martinez, C., Fambrini, V., Barcelli, D., Puig, V.: Model predictive control of drinking water networks: A hierarchical and decentralized approach. In: Proceedings of the 2010 American Control Conference, pp. 3951–3956. IEEE (2010)
73. Pecci, F., Jenkins, J.D.: Regularized benders decomposition for high performance capacity expansion models. arXiv preprint arXiv:2403.02559 (2024)
74. Pena, I., Martinez-Anido, C.B., Hodge, B.M.: An extended ieee 118-bus test system with high renewable penetration. IEEE Trans. Power Syst. **33**(1), 281–289 (2017)
75. Pereira, M.V., Pinto, L.M.: Stochastic optimization of a multireservoir hydroelectric system: A decomposition approach. Water Resour. Res. **21**(6), 779–792 (1985)
76. Pereira, M.V., Pinto, L.M.: Multi-stage stochastic optimization applied to energy planning. Math. Program. **52**, 359–375 (1991)
77. Rahmaniani, R., Crainic, T.G., Gendreau, M., Rei, W.: The benders decomposition algorithm: A literature review. Eur. J. Oper. Res. **259**(3), 801–817 (2017)
78. Rahmaniani, R., Crainic, T.G., Gendreau, M., Rei, W.: An asynchronous parallel benders decomposition method for stochastic network design problems. Computers & Operations Research **162**, 106459 (2024)
79. Rais, A., Viana, A.: Operations research in healthcare: a survey. Int. Trans. Oper. Res. **18**(1), 1–31 (2011)
80. Ruszczyński, A.: A regularized decomposition method for minimizing a sum of polyhedral functions. Math. Program. **35**, 309–333 (1986)

81. dos Santos, K.V., Colonetti, B., Finardi, E.C., Zavala, V.M.: Accelerated dual dynamic integer programming applied to short-term power generation scheduling. International Journal of Electrical Power & Energy Systems **145**, 108689 (2023)
82. Santos, T.N., Diniz, A.L., Borges, C.L.T.: A new nested benders decomposition strategy for parallel processing applied to the hydrothermal scheduling problem. IEEE Transactions on Smart Grid **8**(3), 1504–1512 (2016)
83. Schlag, S., Heuer, T., Gottesbüren, L., Akhremtsev, Y., Schulz, C., Sanders, P.: High-quality hypergraph partitioning. ACM J. Exp. Algorithmics **27**, 1–39 (2023)
84. Sharma, P., Sarker, B.R., Romagnoli, J.A.: A decision support tool for strategic planning of sustainable biorefineries. Computers & Chemical Engineering **35**(9), 1767–1781 (2011)
85. Shin, S.: Graph-structured nonlinear programming: Properties and algorithms. Ph.D. thesis, University of Wisconsin-Madison (2021). https://asset.library.wisc.edu/1711.dl/V2UHW7KSFIKBQ8Q/R/file-e04b2.pdf
86. Shin, S., Anitescu, M., Zavala, V.M.: Overlapping schwarz decomposition for constrained quadratic programs. In: 2020 59th IEEE Conference on Decision and Control (CDC), pp. 3004–3009 (2020). https://doi.org/10.1109/CDC42340.2020.9304139
87. Shin, S., Hart, P., Jahns, T., Zavala, V.M.: A hierarchical optimization architecture for large-scale power networks. IEEE Transactions on Control of Network Systems **6**(3), 1004–1014 (2019)
88. Shin, S., Zavala, V.M.: Diffusing-horizon model predictive control. IEEE Trans. Autom. Control **68**(1), 188–201 (2021)
89. Shin, S., Zavala, V.M., Anitescu, M.: Decentralized schemes with overlap for solving graph-structured optimization problems. IEEE Transactions on Control of Network Systems **7**(3), 1225–1236 (2020)
90. Tang, W., Allman, A., Mitrai, I., Daoutidis, P.: Resolving large-scale control and optimization through network structure analysis and decomposition: A tutorial review. In: 2023 American Control Conference (ACC), pp. 3113–3129. IEEE (2023)
91. Tang, W., Allman, A., Pourkargar, D.B., Daoutidis, P.: Optimal decomposition for distributed optimization in nonlinear model predictive control through community detection. Computers & Chemical Engineering **111**, 43–54 (2018)
92. Tian, P., Xiao, X., Wang, K., Ding, R.: A hierarchical energy management system based on hierarchical optimization for microgrid community economic operation. IEEE Transactions on Smart Grid **7**(5), 2230–2241 (2015)
93. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. arXiv preprint arXiv:1711.07356 (2017)
94. Vegetti, M., Henning, G.: Ontology network to support the integration of planning and scheduling activities in batch process industries. J. Ind. Inf. Integr. **25**, 100254 (2022)
95. Wang, P., Wu, Q., Huang, S., Li, C., Zhou, B.: Admm-based distributed active and reactive power control for regional ac power grid with wind farms. Journal of Modern Power Systems and Clean Energy **10**(3), 588–596 (2021)
96. Wood, A.J., Wollenberg, B.F., Sheblé, G.B.: Power generation, operation, and control. John Wiley & Sons (2013)
97. Wu, D., Ierapetritou, M.: Hierarchical approach for production planning and scheduling under uncertainty. Chem. Eng. Process. **46**(11), 1129–1140 (2007)
98. Yagi, K., Sioshansi, R.: Nested benders's decomposition of capacity-planning problems for electricity systems with hydroelectric and renewable generation. CMS **21**(1), 16 (2024)
99. Yue, D., You, F., Snyder, S.W.: Biomass-to-bioenergy and biofuel supply chain optimization: Overview, key issues and challenges. Computers & Chemical Engineering **66**, 36–56 (2014)
100. Zhang, H., Grossmann, I.E., Tomasgard, A.: Decomposition methods for multi-horizon stochastic programming. CMS **21**(1), 32 (2024)
101. Zhang, H., Mazzi, N., McKinnon, K., Nava, R.G., Tomasgard, A.: A stabilised benders decomposition with adaptive oracles for large-scale stochastic programming with short-term and long-term uncertainty. Computers & Operations Research **167**, 106665 (2024)
102. Zou, J., Ahmed, S., Sun, X.A.: Stochastic dual dynamic integer programming. Math. Program. **175**, 461–502 (2019)
103. Zverovich, V., Fábián, C.I., Ellison, E.F., Mitra, G.: A computational study of a solver system for processing two-stage stochastic lps with enhanced benders decomposition. Math. Program. Comput. **4**, 211–238 (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.