`PlasmoData.jl` – A Julia Framework for Modeling and Analyzing Complex Data as Graphs

David L. Cole[†] and Victor M. Zavala[†‡*]

[†]Department of Chemical and Biological Engineering,
University of Wisconsin-Madison, Madison, WI 53706 United States of America

[‡]Mathematics and Computer Science Division,
Argonne National Laboratory, Lemont, IL 60439 United States of America

Abstract

Datasets encountered in scientific and engineering applications appear in complex formats (e.g., images, multivariate time series, molecules, video, text strings, networks). Graph theory provides a unifying framework to model such datasets and enables the use of powerful tools that can help analyze, visualize, and extract value from data. In this work, we present `PlasmoData.jl`, an open-source, `Julia` framework that uses concepts of graph theory to facilitate the modeling and analysis of complex datasets. The core of our framework is a general data modeling abstraction, which we call a `DataGraph`. We show how the abstraction and software implementation can be used to represent diverse data objects as graphs and to enable the use of tools from topology, graph theory, and machine learning (e.g., graph neural networks) to conduct a variety of tasks. We illustrate the versatility of the framework by using real datasets: i) an image classification problem using topological data analysis to extract features from the graph model to train machine learning models; ii) a disease outbreak problem where we model multivariate time series as graphs to detect abnormal events; and iii) a technology pathway analysis problem where we highlight how we can use graphs to navigate connectivity. Our discussion also highlights how `PlasmoData.jl` leverages native `Julia` capabilities to enable compact syntax, scalable computations, and interfaces with diverse packages. Overall, we show that the `DataGraph` abstraction and `PlasmoData.jl Julia` package are able to *model* data within graphs and enable useful analysis.

Keywords: graph theory, network theory, modeling, data, open-source, scalability.

1   Introduction

Data appears in complex formats that require the use of advanced tools for its representation and processing. In this context, it is important to highlight that any data object (e.g., image, video, audio signal, a text string, time series) needs to be represented as a mathematical model (a mathematical abstraction) to enable processing and analysis. Examples of mathematical models that are available

---

[*]Corresponding Author: victor.zavala@wisc.edu

to do this include those from linear algebra (e.g., data is modeled as vectors, matrices, higher-order tensors), graph/network theory (e.g., data is modeled as graphs or hypergraphs), statistics (e.g., data is modeled as random variables), and topology/geometry (e.g., data is modeled as graphs, manifolds, and simplicial complexes). Understanding how to model data and to recognize alternative representations of data is key in facilitating its analysis and in extracting information/value [1, 2, 3]. For example, an image can be represented as a matrix (each entry is a pixel), as a graph (each node is a pixel and edges represent connectivity of neighboring pixels), or as a function/manifold (the additional dimension represents the light intensity). These various representations reveal different (and often complementary) aspects/features of the data object; for instance, a matrix representation can help reveal correlation structures, while a graph representation can reveal connected structures. Importantly, how to represent a dataset is a *modeling decision* that can impact the information extracted from the data (can influence the value of the dataset); as such, it is necessary to have proper tools that can help experiment with different data models and associated analysis tools.

A modeling abstraction that has gained increased popularity in data science is graphs (networks). Graphs provide a unifying framework to capture diverse data objects because they have intuitive interpretation and they enable the use of powerful processing techniques. Simply stated, a graph is a mathematical model that comprises a set of nodes that are connected via edges; the nodes and edges are abstract objects that can encode data that may appear in different forms. In a typical graph, data attached to nodes and edges are scalar values or vectors, but more advanced graph abstractions can attach abstract objects such as algebraic models, text, or other graphs [4, 5]. Graph representations have been used in a wide variety of applications such as chemical processes [6], biological systems [7], brain networks [8, 9], hydrology [10, 11, 12], disease transmission [13, 14], and social networks [15]. An application that has recently gained substantial attention is the representation of molecules as graphs [16, 17, 18]; here, atoms are represented by nodes, bonds are represented by edges, and nodes/edges can embed diverse attributes such as atom type and bond length (represented as vectors). These representations have been used to predict diverse properties of molecules and to predict potential reactions that can occur between them.

Recently, the field of topological data analysis (TDA) has opened interesting and new perspectives on data modeling and processing For example, a matrix (e.g., a grayscale image) can be represented as a node-weighted graph. Here, nodes are matrix entries, node weights are numerical values (data) attached to the matrix entries, and edges capture adjacency of the matrix entries [19, 20]. This representation can be applied to high-dimensional data (tensors), where entries of the higher dimensional object are connected to adjacent entries. Symmetric matrices (e.g., correlation or covariance matrices) can also be represented as edge-weighted graphs, where the nodes represent the variable associated with a column/row and edge weights capture the degree of correlation/covariance. The graph representation of correlation matrices is widely used in neuroscience to analyze brain connectivity [19, 8, 9] and in disease transmission to identify extreme events [13]. This is done via the use of tools of graph theory and topology, which enable the quantification of the graph structure or shape. For example, the Euler Characteristic (EC) is a topological descriptor that quantifies the structure of an unweighted

2

graph by computing the number of connected components and the number of cycles in the graph [19]. To quantify the structure of weighted graphs (graphs with attached values to nodes and edges) one can use filtration/percolation operations; as the name suggests, these operations filter out nodes/edges with weight values below a certain threshold values. Filtration operations keep track of how topological descriptors appear or disappear at different weight threshold values, and this information is summarized in the form of topological summaries, such as the EC curve. Topological descriptors can reveal aspects of the data that might not be accessed by tools from linear algebra or statistics. For instance, eigenvalues and eigenvectors are linear algebra descriptors that extract information from matrices in the form of energy/variance, not shape.

Graph representations of data enables access to a wide variety of software tools for analysis and visualization. These tools range from broad to specific application scope. Some of the most general graph modeling tools include `NetworkX` [21] (in Python), `Graphs.jl` [22] (in Julia), `igraph` [23] (in Python and R), `graph-tool` [24] (in Python), and the `graph` and `digraph` functions of Matlab [25]. These packages are general and allow the user to define their own graph data structures. They also include functions for analyzing graph connectivity and other metrics/descriptors. These tools also allow the user to define abstract properties or attributes on nodes and edges (in the case of `Graphs.jl`, this ability is implemented in the package extension `MetaGraphs.jl`). These packages use different methods for storing the data; depending on the amount or nature of the data, this can have significant implications on scalability and application scope. These packages are limited in the methods that they use for processing and analyzing data; for instance, these tools do not implement topological analysis operations (e.g., filtration operations and computation of topological descriptors). Furthermore, these packages do not provide capabilities to transform data models to graph models (e.g., obtain a graph from a tensor).

In addition to general graph modeling packages, several other packages exist that have more specific application scope. For instance, there are several graph tools specifically designed for studying brain function, including `Graphvar` [26], `brain-connectivity-toolbox` [9], `GRETNA` [27], and `BRAPH` [28]. Because of this broad spectrum, a complete review of available tools is not practical here, but we will discuss the most pertinent and prominent options. Some application-specific packages are targeted towards creating models for machine learning, such as graph neural networks (GNNs). For instance, `MoleculeNet` [18] is designed for representing molecules as graphs, which can then be used for GNNs to enable predictions. The deep graph library [29], `scikit-network` [30], and `GraphNeuralNetworks.jl` [31] all provide an interface for defining graphs with data that can then be used for training and testing GNNs. These packages are effective at conducting machine learning tasks, but they are not designed for performing transformations of data objects and topological analysis of the graph structure (e.g., finding the EC or connected components). In summary, existing graph modeling tools are often limited in their ability to represent and process wide spans of data.

The goal of this work is to provide a general framework for modeling data as graphs. Recently, we have developed a framework for modeling optimization problems as graphs[4]; in this abstraction,

nodes and edges can embed algebraic optimization problems. This abstraction has been shown to unify a wide range of structures found in optimization (e.g., optimal control, stochastic optimization, and network optimization) and this unification has enabled a number of advances in theory and algorithms. Moreover, the abstraction has facilitated access to a broad range of graph analysis and software tools that facilitate visualization and processing (e.g., graph partitioning and aggregation). Our work aims to expand this graph abstraction to model general data and with this unify data objects found across domains and leverage the use of tools from such domains. Our abstraction is implemented as an open-source `Julia` package that we call `PlasmoData.jl`. This package has been designed to readily represent diverse data objects as graphs (e.g., images, matrices, and tensors as node-weighted graphs and symmetric matrices as edge-weighted graphs) and to store data within a user-defined graph structure. Our modeling framework interfaces to diverse software packages that enables processing of graph data objects (e.g., via filtration, partition, and aggregation), facilitates the computation of descriptors using tools from graph theory and topology, and enables the use of machine learning tools (e.g., graph neural networks or GNNs). Our software design principle is analogous to those of algebraic modeling languages (e.g., `JuMP` or `Plasmo.jl`), which provide provide interfaces to packages that process/solve such models.

The goal behind the implementation of `PlasmoData.jl` *is thus to change the way the user thinks about data (as a modeling task rather than just an analysis task).* We believe that the focus on modeling can bring significant benefits in the way we explore alternative data representations that are suitable for applications, in selecting suitable tools to enable data processing, and in interpreting analysis results. We demonstrate the versatility of `PlasmoData.jl` and of its unifying modeling abstraction by using applications that appear in quite distinct application domains. A graphical representation of this work is shown in Figure 1 and highlights how a variety of data can be modeled within `PlasmoData.jl`, which in turn provides access to diverse analysis tools and techniques.

The paper is structured as follows. Section 2 provides an overview of the `DataGraph` abstraction for modeling data as graphs, discusses how it is implemented in `PlasmoData.jl`, and illustrates how some common data structures can be represented under the proposed abstraction. Section 3 provides an overview of data analysis that can be performed for data modeled as graphs. Section 4 provides applications that illustrate the versatility of the modeling framework. Section 5 provides concluding remarks and future directions.

2 Graph Abstraction and Software Implementation

This section introduces the `DataGraph` modeling abstraction along with its implementation in `PlasmoData.jl`, available at https://github.com/zavalab/PlasmoData.jl. We introduce a mathematical formulation for the `DataGraph` and discuss how it can be applied to a variety of common data structures. We also provide brief code snippets showing how these representations are implemented in `PlasmoData.jl`. Versions of these code snippets are also available in the repository https://github.com/zavalab/JuliaBox/tree/master/PlasmoData_examples, along with the package and
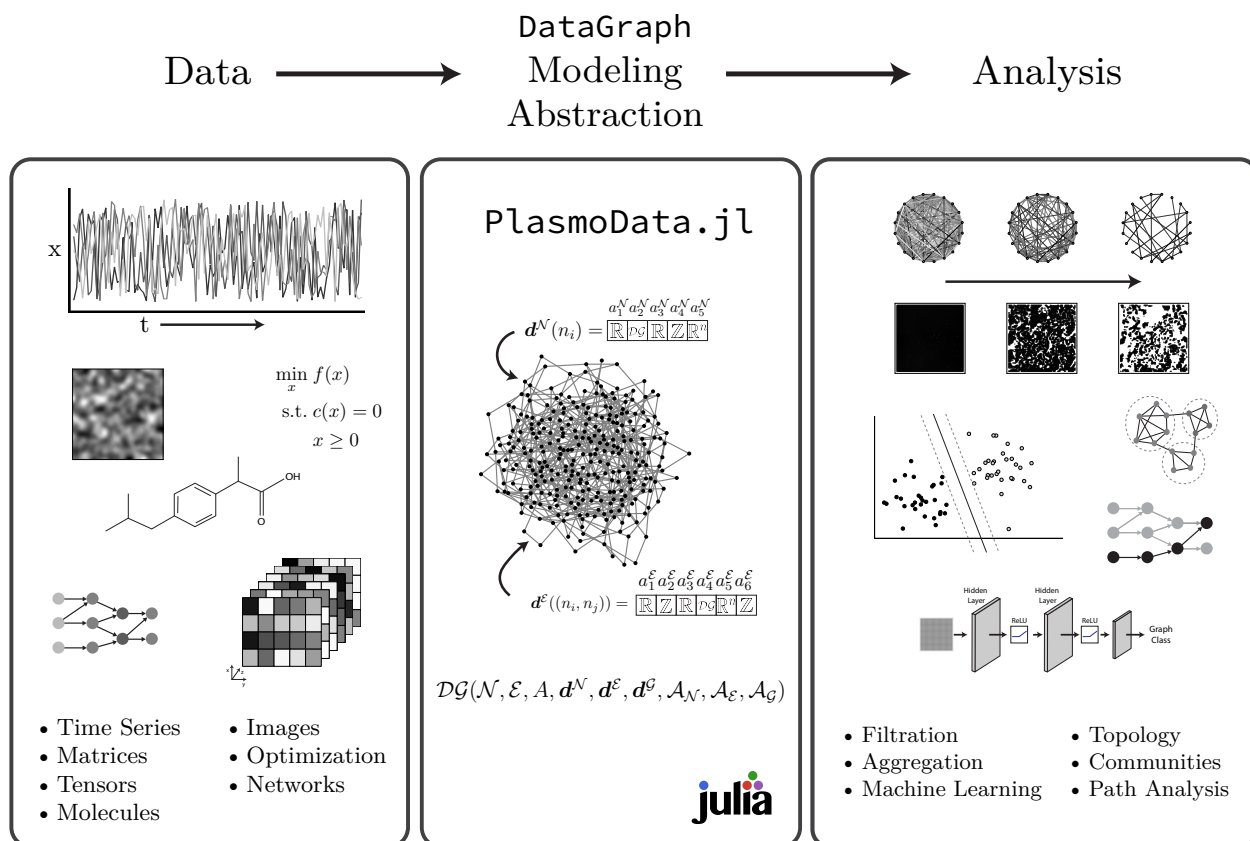
Figure 1: Overview of the `PlasmoData.jl` modeling framework. Diverse data objects can be modeled under the `DataGraph` abstraction; this provides access to a diverse analysis tools and techniques for analyzing the data

Julia versions used.

2.1 Mathematical Formulation

A graph is a mathematical model that contains a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$ (which connect nodes). A node $n \in \mathcal{N}$ and an edge $e \in \mathcal{E}$ can encode diverse abstract objects (e.g., values, text, equations). To indicate data embedded in the graph, we define the set of node (data) attributes as $\mathcal{A}_\mathcal{N}$, the set of edge (data) attributes as $\mathcal{A}_\mathcal{E}$, and a set of (global) graph attributes as $\mathcal{A}_\mathcal{G}$. To associate data to specific nodes, edges, and to the graph we make the following definitions:

$$d_a^\mathcal{N}(n) \in \mathcal{D}_a, \quad n \in \mathcal{N}, a \in \mathcal{A}_\mathcal{N} \tag{1}$$

$$d_a^\mathcal{E}(e) \in \mathcal{D}_a, \quad e \in \mathcal{E}, a \in \mathcal{A}_\mathcal{E} \tag{2}$$

$$d_a^\mathcal{G} \in \mathcal{D}_a, \quad a \in \mathcal{A}_\mathcal{G} \tag{3}$$

Here, $\mathcal{D}_a$ is the set of possible values for a data attribute a, $d_a^\mathcal{N}(n)$ is the data stored on the node n for attribute a, $d_a^\mathcal{E}(e)$ is the data stored on the edge e for attribute a, and $d_a^\mathcal{G}$ is the data stored on the graph for attribute a. Importantly, our definitions are abstract and we make no distinction on the form that any of that data must take. The data could be scalars, vectors, matrices, graphs, an optimization model, text strings, other graphs, or any other kind of information (see [32, 33, 4] for application examples for data that can be embedded in graphs). We use the notation $\boldsymbol{d}^\mathcal{N}$, $\boldsymbol{d}^\mathcal{E}$, and $\boldsymbol{d}^\mathcal{G}$ to define the set of all node, edge, and graph data corresponding to $\mathcal{A}_\mathcal{N}$, $\mathcal{A}_\mathcal{E}$, and $\mathcal{A}_\mathcal{G}$, respectively. We will use the notation $\boldsymbol{d}^\mathcal{N}(n)$ for the set of node data on node n for the attribute set $\mathcal{A}_\mathcal{N}$ and $\boldsymbol{d}^\mathcal{E}(e)$ for the set of node data on edge e for the attribute set $\mathcal{A}_\mathcal{E}$. Note that, under these definitions, the *attribute* sets are used for indicating subsets of data (and do not represent the data itself).

We use the previous definitions to define our modeling object, which we call a `DataGraph`. This modeling abstraction has a graph structure and it stores data on the nodes, edges, and the graph itself; the object has the following form:

$$\mathcal{DG}\left(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^\mathcal{N}, \boldsymbol{d}^\mathcal{E}, \boldsymbol{d}^\mathcal{G}, \mathcal{A}_\mathcal{N}, \mathcal{A}_\mathcal{E}, \mathcal{A}_\mathcal{G}\right). \tag{4}$$

A `DataGraph` object is fully defined by its nodes, edges, adjacency matrix, (A which encodes the graph connectivity), node data, edge data, graph data, node attributes, edge attributes, and graph attributes. Importantly, the adjacency matrix also encodes the type of edges in the graph (directed or undirected).

2.2 Software Implementation

The `DataGraph` abstraction is implemented in `Julia` in the package `PlasmoData.jl` (see Figure 2). `PlasmoData.jl` allows for any data structure to be encoded on the nodes, edges, or on the graph itself. To differentiate references to the mathematical formulation from the computer implementation, we

will refer to the software counterpart as `PlasmoData.DataGraph`.

The `PlasmoData.DataGraph` object implemented contains the following fundamental attributes (defined in the software data structure):

- `g`: A `Graphs.SimpleGraph` (or a `Graphs.SimpleDiGraph` for directed graphs) object. The `Graphs.SimpleGraph` object is defined within the `Graphs.jl` package, and it stores graph structure efficiently by storing only the number of edges and a vector of neighbors for each node. `PlasmoData.jl` has `Graphs.jl` as a dependency and stores the `SimpleGraph` object to enable access to a variety of graph algorithms and to provide a simpler interface with other tools.

- `nodes`: A vector of node names for each node in the graph. Within `PlasmoData.jl`, nodes can be "named" or defined using any data type. This is different from some other graph modeling tools. For example, `Graphs.jl` does not directly name any nodes, but instead only records an adjacency list for each node, and thus implicitly names nodes only by an integer index. `nodes`, corresponds to $\mathcal{N}$ in (4).

- `edges`: A vector of pairs (tuples) of integers corresponding to each edge of the graph. The integer pairs correspond to the index of nodes in `nodes` (e.g., pair $(1, 2)$ corresponds to an edge between the first and second nodes defined in `nodes`). Restricting `edges` to only contain integer pairs (rather than the node pairs themselves) is done to reduce the amount of memory required to store the graph. `edges`, corresponds to $\mathcal{E}$ in (4).

- `node_map`: A dictionary mapping the node names to their index in the `nodes` vector. This index also corresponds to the adjacency list saved in the `Graphs.jl` `g` object.

- `edge_map`: A dictionary mapping the edges (integer pairs) to their index in the `edges` vector.

- `node_data`: A mutable structure that contains a matrix of all node data where each row of the matrix corresponds to a node in the graph and each column corresponds to a node attribute. The order of the rows is identical to the order in `nodes`. This data structure also contains a list of all node attributes and their mapping to the data matrix. Node attributes are restricted to be strings. `node_data` corresponds to $\boldsymbol{d}^{\mathcal{N}}$ in (4).

- `edge_data`: A mutable structure that contains a matrix of all edge data where each row of the matrix corresponds to an edge in the graph and each column corresponds to an edge attribute. The order of the rows is identical to the order in `edges`. This data structure also contains a list of all edge attributes and their mapping to the data matrix. Edge attributes are restricted to be strings. `edge_data` corresponds to $\boldsymbol{d}^{\mathcal{E}}$ in (4).

- `graph_data`: A mutable structure that contains a vector of graph data. This data structure also contains a list of all graph attributes and their mapping to the data vector. Graph attributes are restricted to be strings. `graph_data` corresponds to $\boldsymbol{d}^{\mathcal{G}}$ in (4).

While the mathematical abstraction encodes directionality (or lack thereof) via the adjacency matrix, A, `PlasmoData.jl` includes `PlasmoData.DataGraph` and a `PlasmoData.DataDiGraph` modeling objects for undirected and directed graphs, respectively. This terminology is consistent with other graph modeling tools (e.g., `Graphs.jl` and `NetworkX`). For our software design, how we chose to store node, edge, and graph data was an important consideration that deserves some explanation as `PlasmoData.jl` stores this data differently than some alternative tools. A discussion of this decision, as well as a comparison of memory allocation for storing data in `PlasmoData.jl` and other graph modeling tools, is provided in the supporting information.

```
mutable struct DataGraph{T, T1, T2, T3, M1, M2}
                    <: AbstractDataGraph{T}

    g::Graphs.SimpleGraph{T}

    nodes::Vector{Any}
    edges::Vector{Tuple{T, T}}
    node_map::Dict{Any, T}
    edge_map::Dict{Tuple{T, T}, T}

    node_data::NodeData{T, T1, M1}
    edge_data::EdgeData{T, T2, M2}
    graph_data::GraphData{T, T3}
end
```

$$\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}})$$

$$\boldsymbol{d}^{\mathcal{N}}(n_i) = \begin{array}{ccccc} a_1^{\mathcal{N}} & a_2^{\mathcal{N}} & a_3^{\mathcal{N}} & a_4^{\mathcal{N}} & a_5^{\mathcal{N}} \\ \boxed{\mathbb{R}} & \boxed{\mathcal{DG}} & \boxed{\mathbb{R}} & \boxed{\mathbb{Z}} & \boxed{\mathbb{R}^n} \end{array}$$

$$\boldsymbol{d}^{\mathcal{N}} =$$

$$\boldsymbol{d}^{\mathcal{E}} =$$

$$\boldsymbol{d}^{\mathcal{E}}((n_i, n_j)) = \begin{array}{cccccc} a_1^{\mathcal{E}} & a_2^{\mathcal{E}} & a_3^{\mathcal{E}} & a_4^{\mathcal{E}} & a_5^{\mathcal{E}} & a_6^{\mathcal{E}} \\ \boxed{\mathbb{R}} & \boxed{\mathbb{Z}} & \boxed{\mathbb{R}} & \boxed{\mathcal{DG}} & \boxed{\mathbb{R}^n} & \boxed{\mathbb{Z}} \end{array}$$

$$\boldsymbol{d}^{\mathcal{G}} = \begin{array}{cccc} a_1^{\mathcal{G}} & a_2^{\mathcal{G}} & a_3^{\mathcal{G}} & a_4^{\mathcal{G}} \\ \boxed{\mathbb{R}} & \boxed{\mathcal{DG}} & \boxed{\mathbb{Z}} & \boxed{\mathbb{R}^n} \end{array}$$
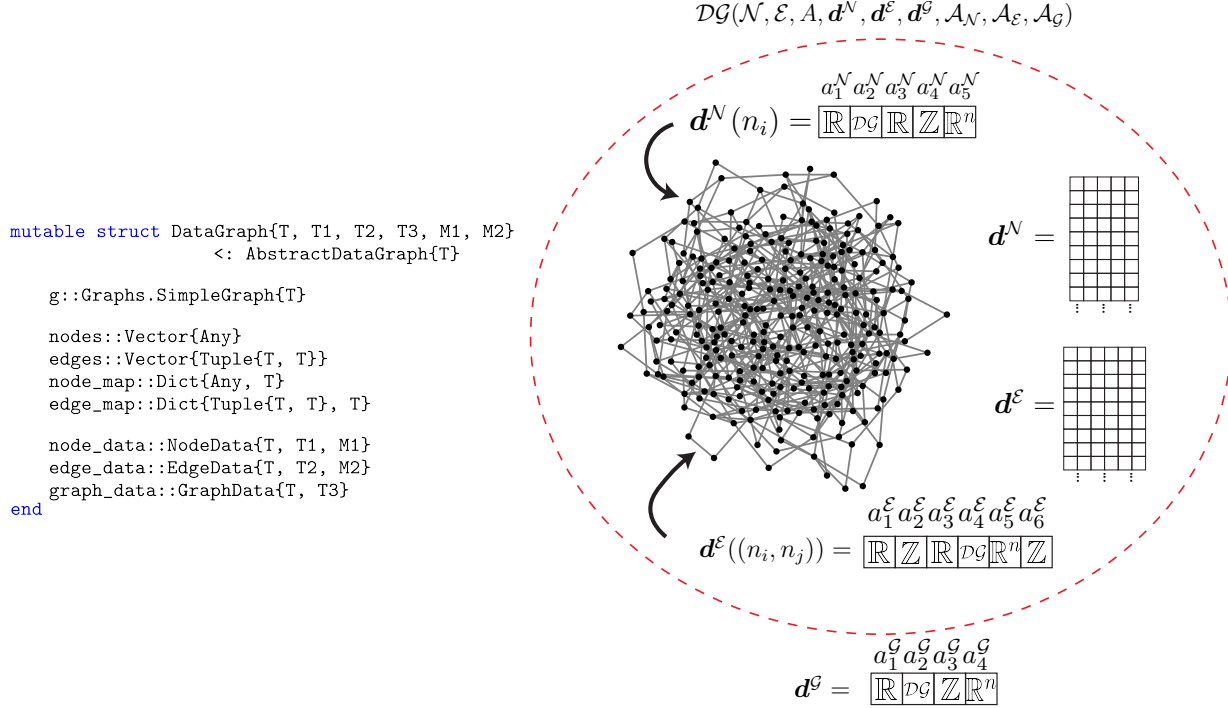
Figure 2: Visualization of the `DataGraph` object. The `PlasmoData.DataGraph` (left) contains fundamental attributes (`g`, `nodes`, `edges`, `node_map`, `edge_map`, `node_data`, `edge_data`, `graph_data`) which correspond to attributes of the mathematical `DataGraph` object (right) containing nodes ($\mathcal{N}$), edges ($\mathcal{E}$), directionality (A), node data ($\boldsymbol{d}^{\mathcal{N}}$), edge data ($\boldsymbol{d}^{\mathcal{E}}$), graph data ($\boldsymbol{d}^{\mathcal{G}}$), and node, edge, and graph attributes ($\mathcal{A}_{\mathcal{N}}$, $\mathcal{A}_{\mathcal{E}}$, and $\mathcal{A}_{\mathcal{G}}$, respectively). Each node and edge can embed data of a variety of forms, such as scalar values, vectors, or other graphs

2.2.1 `PlasmoData.jl` Tutorial Example

`PlasmoData.jl` implements the `DataGraph` abstraction as shown above, and it provides a user-friendly interface and suite of functions; basic funcationality for constructing the graph is illustrated in Code Snippet 1. An empty `PlasmoData.DataGraph` is instantiated on Line 3. The data types within the {} define the types of data that can be defined in the node, edge, and graph data (see Figure 2). In this example, we use `Any` for the data type in the node, edge, and graph data which will be important

when adding data to this graph. Alternatively, a user can call `DataGraph()` without the information in {}, and the default data types in the node, edge, and graph data will be `Float64`. Once the `PlasmoData.DataGraph` object is instantiated, nodes, edges, and data can be added to the graph. Nodes can be added to the `PlasmoData.DataGraph` by calling the `add_node!` function (Lines 6 - 10). `PlasmoData.jl` allows for any `Julia` object to be defined as a node. For example, as shown in the code snippet, the node names can be integers, strings, or symbols. This can be important, as will be shown later when representing matrices as `PlasmoData.DataGraph`s. Edges can be added to the `PlasmoData.DataGraph` by calling the `add_edge!` function (Lines 14 - 18). This function takes pairs of node names as arguments which represent the edge between those two nodes. These pairs of nodes can be passed as separate arguments (Lines 14 - 16) or as a tuple of two node names (Lines 17 - 18). This progressively builds the adjacency matrix of the graph object.

Code Snippet 1: Example showing the basic functionality for building a `PlasmoData.DataGraph` in `PlasmoData.jl`, including adding nodes, edges, and data and basic plotting functionality

```julia
using PlasmoData, PlasmoDataPlots

dg = DataGraph{Int, Any, Any, Any, Matrix{Any}, Matrix{Any}}()

# add_node!(datagraph, node)
add_node!(dg, 1)
add_node!(dg, 2)
add_node!(dg, 3)
add_node!(dg, "node4")
add_node!(dg, :node5)

# add_edge!(datagraph, node1, node2)
# add_edge!(datagraph, (node1, node2))
add_edge!(dg, 1, 2)
add_edge!(dg, 2, 3)
add_edge!(dg, "node4", 1)
add_edge!(dg, (:node5, 2))
add_edge!(dg, (3, "node4"))

# add_node_data!(datagraph, node, node_data, data_attribute)
add_node_data!(dg, 1, [6, 3, 4], "node_data1")
add_node_data!(dg, 2, 3.4, "node_data1")
add_node_data!(dg, 3, "this is on node 3", "node_data1")
add_node_data!(dg, "node4", [1 2; 3 4], "node_data1")
add_node_data!(dg, :node5, DataGraph(), "node_data1")

# add_edge_data!(datagraph, node1, node2, edge_data, edge_attribute)
# add_edge_data!(datagraph, (node1, node2), edge_data, edge_attribute)
add_edge_data!(dg, 1, 2, DataGraph(), "edge_data1")
add_edge_data!(dg, 2, 3, [1 2 ; 5 7], "edge_data1")
add_edge_data!(dg, "node4", 1, 1.0, "edge_data1")
add_edge_data!(dg, (:node5, 2), -0.00001, "edge_data1")
add_edge_data!(dg, (3, "node4"), Dict(), "edge_data1")

# add_graph_data!(dg, graph_data, graph_attribute)
add_graph_data!(dg, 1.0, "graph_data1")

PlasmoDataPlots.plot_graph(dg; xdim = 400, ydim = 400)
```

Data can also be embedded within the defined graph structure through API functions. Data is added to nodes through the `add_node_data!` function (Lines 21 - 25). This function takes the following arguments: i) the `PlasmoData.DataGraph`, ii) the node name, iii) the node data, and iv) the attribute name for the data (must be a string). The node data is restricted to be the type defined
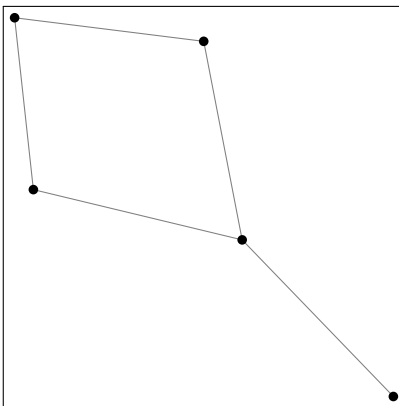
9

Figure 3: Visualization of the `PlasmoData.DataGraph` associated with Code Snippet 1

on the `PlasmoData.DataGraph` (in this case, the `PlasmoData.DataGraph` was instantiated with type `Any`, so any data can be passed). As seen in the code snippet, data have different formats, including floats, strings, vectors, matrices, or another `PlasmoData.DataGraph`. Data is added to edges through the `add_edge_data!` function (Lines 29 - 33). This function behaves similarly to the `add_node_data!` function, but in place of the node name argument, a pair of nodes (two separate arguments) or a tuple of two node names (one argument) is supplied. Data is added to the graph by the `add_graph_data!` function (Line 36). This function only takes the graph data and the graph attribute, as shown in the snippet. Additional node, edge, or graph data can be defined on the `DataGraph` by passing the data with a different attribute name. There is no limit to the number of attributes that can be added to the nodes, edges, or graph.

One of the benefits of graph representations is that they facilitate visualization of structure. For basic plotting capabilities, we have developed an accompanying package called `PlasmoDataPlots.jl` (available at https://github.com/zavalab/PlasmoDataPlots.jl) that interfaces to `PlasmoData.jl` and that enables visualizations of `PlasmoData.DataGraph` objects. This package provides basic functionalities for plotting both `PlasmoData.DataGraph`s and `PlasmoData.DataDiGraph`s primarily through the Julia packages `Plots.jl` and `GraphMakie.jl`. The visualization in Figure 3 is created by calling the `PlasmoDataPlots.jl` function `plot_graph` (Line 38).

The methods presented above for building `PlasmoData.DataGraph`s, adding nodes and edges, and adding data also work for `DataDiGraph`s. The only difference is that the instantiation function is called `DataDiGraph` and the edge order is maintained in the internal structure of the `DataDiGraph`.

2.3 Modeling Common Data Objects as `DataGraphs`

With the general mathematical formulation for the `DataGraph` defined, we next outline how specific data objects (matrices, 3D tensors, symmetric matrices) can be represented using the graph abstraction and how these can be implemented in `PlasmoData.jl`.Importantly, a benefit of representing matrices and tensors as graphs is that graphs are modeling objects that do not live in a Euclidean space. As

such, all that matters from a graph perspective is connectivity (not location of nodes and edges in space). As such, a graph object is not affected by rotations (matrices are). This property is quite useful when processing images/video and also when processing data objects from molecular simulations (in which nodes move randomly in space) [34]. Thus the approaches in this section could be applied to a variety of problems in different fields.

2.3.1 Matrices as a Node-Weighted Graphs

We can represent a matrix $M \in \mathbb{R}^{p \times q}$ (where p and q are positive integers) as a node-weighted graph with a mesh structure [19, 35], and we can express the resulting graph using the `DataGraph` abstraction. We use the notation $m_{i,j} \in \mathbb{R}$ for the matrix entry at row i and column j. For convenience, we define the integer set $\mathbb{N}_k := \{1, 2, ..., k\}$. To represent the matrix as a graph, we define a node, n_*, for every matrix entry and place edges between all adjacent entries of the matrix. On each node, $m_{i,j}$ is stored as the node data. As there is a single weight in each node, we define the attribute set as $\mathcal{A}_\mathcal{N} = \{a\}$ (a singleton). Mathematically, this is represented with the `DataGraph` abstraction as:

$$\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^\mathcal{N}, \emptyset, \emptyset, \mathcal{A}_\mathcal{N}, \emptyset, \emptyset)$$

$$\text{where } \mathcal{N} = \{n_{i,j} : i \in \mathbb{N}_p, j \in \mathbb{N}_q\}$$

$$\mathcal{E} = \{(n_{i,j}, n_{i+1,j}) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_q\} \cup \{(n_{i,j}, n_{i,j+1}) : i \in \mathbb{N}_p, j \in \mathbb{N}_{q-1}\} \quad (5)$$

$$\mathcal{A}_\mathcal{N} = \{a\}$$

$$d_a^\mathcal{N}(n_{i,j}) = m_{i,j}, i \in \mathbb{N}_p, j \in \mathbb{N}_q$$

where A is a symmetric matrix ($\mathcal{DG}$ is undirected) matching the connectivity defined by $\mathcal{E}$. An alternative structure can also be constructed by also adding edges between diagonal elements of the matrix [36]. In this case, the set of edges becomes $\mathcal{E} = \{(n_{i,j}, n_{i+1,j}) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_q\} \cup \{(n_{i,j}, n_{i,j+1}) : i \in \mathbb{N}_p, j \in \mathbb{N}_{q-1}\} \cup \{(n_{i,j}, n_{i+1,j+1}) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_{q-1}\} \cup \{(n_{i,j+1}, n_{i+1,j}) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_{q-1}\}$.

`PlasmoData.jl` facilitates the representation of matrices as graphs. Code Snippet 2 shows how a `PlasmoData.DataGraph` object can be constructed automatically by calling the `matrix_to_graph` function. Any matrix containing real numbers can be passed to this function. The boolean keyword argument `diagonal` can be passed to identify whether the diagonal edges should be included in the graph. The resulting graphs can also be visualized using `PlasmoDataPlots.jl`. In Line 11, nodes are given a fixed position (stored in the node data of the `PlasmoData.DataGraph`) for visualization. In Lines 14 - 21, the graph containing diagonals is plotted, where the weight data from the original matrix is used for coloring the nodes (see Line 19). Figure 4 visualizes the alternative representations. Because matrices themselves are a general abstraction for representing data, `PlasmoData.DataGraph` can be applied to many application. For instance, multivariate time series and grayscale images can be represented as matrices (and thus as graphs) [37, 38, 39].

2.3.2 3D Tensors as Node-Weighted Graphs

Below we outline how a 3D tensor can be represented as a graph. While tensors are not restricted to be 3D, when we refer to tensors in this work, we will be referring to 3D tensors unless otherwise

Code Snippet 2: Example of representing a matrix as a `PlasmoData.DataGraph`

```
using PlasmoData, Random, PlasmoDataPlots

Random.seed!(15)
random_matrix = rand(12, 12)

# Convert matrix to node−weighted graph
matrix_graph_diags = matrix_to_graph(random_matrix; diagonal = true)
matrix_graph_no_diags = matrix_to_graph(random_matrix; diagonal = false)

# Fix the node positions
set_matrix_node_positions!(matrix_graph_diags, random_matrix)

# Plot the graph
plot_graph(
    matrix_graph_diags,
    nodesize = 12,
    linewidth = 5,
    nodecolor = :grays,
    node_z = get_node_data(matrix_graph_diags, "weight"),
    rev = true,
)
```


Figure 4: Visualization of graph representations of a matrix. The top graph corresponds to the matrix_graph_no_diags PlasmoData.DataGraph object from Code Snippet 2, and the lower graph corresponds to the matrix_graph_diags PlasmoData.DataGraph object from Code Snippet 2 and is the output of the code snippet. Matrix entries are represented by nodes with the matrix entry values stored as data on each node

noted. Any tensor $T \in \mathbb{R}^{p \times q \times r}$ (where p, q, and r are positive integers) can be represented as a node-weighted graph. There are at least a couple of methods for how to represent T. The first approach is to construct a node-weighted mesh following the description outlined in Section 2.3.1 based on the first couple of dimensions of T (i.e., the mesh structure will be $p \times q$). The third dimension is then represented as a vector of weights attached to every element of the mesh (with $r = |\mathcal{A}_\mathcal{N}|$). This can be used, for instance, to represent color images (which have different color channels). In addition, we use the notation $t_{i,j,k}$ for the tensor entry, where i, j, and k are the entries in the first, second, and third dimensions, respectively. Under the `DataGraph` abstraction, this has the form:

$$\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^\mathcal{N}, \emptyset, \emptyset, \mathcal{A}_\mathcal{N}, \emptyset, \emptyset)$$

$$\text{where } \mathcal{N} = \{n_{i,j} : i \in \mathbb{N}_p, j \in \mathbb{N}_q\}$$

$$\mathcal{E} = \{(n_{i,j}, n_{i+1,j}) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_q\} \cup \{(n_{i,j}, n_{i,j+1}) : i \in \mathbb{N}_p, j \in \mathbb{N}_{q-1}\} \quad (6)$$

$$\mathcal{A}_\mathcal{N} = \{a_1, a_2, ..., a_r\}$$

$$d_{a_k}^\mathcal{N}(n_{i,j}) = t_{i,j,k}, i \in \mathbb{N}_p, j \in \mathbb{N}_q, k \in \mathbb{N}_r$$

where A is a symmetric matrix ($\mathcal{DG}$ is undirected) matching the connectivity defined by $\mathcal{E}$. As with the matrix form in Section 2.3.1, edges can also be placed between the entries that are diagonal to one another in the mesh structure.

Alternatively, each tensor entry can be represented by a node with each node/entry connected to the adjacent nodes/entries (similar to the mesh structure, but now across a third dimension). The value of each tensor entry is then embedded in the corresponding node, so each node only contains one node weight. Under the `DataGraph` abstraction, this has the form:

$$\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^\mathcal{N}, \emptyset, \emptyset, \mathcal{A}_\mathcal{N}, \emptyset, \emptyset)$$

$$\text{where } \mathcal{N} = \{n_{i,j,k} : i \in \mathbb{N}_p, j \in \mathbb{N}_q, k \in \mathbb{N}_r\}$$

$$\mathcal{E} = \{(n_{i,j,k}, n_{i+1,j,k}) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_q, k \in \mathbb{N}_r\} \cup \{(n_{i,j,k}, n_{i,j+1,k}) :$$

$$i \in \mathbb{N}_p, j \in \mathbb{N}_{q-1}, k \in \mathbb{N}_r\} \cup \{(n_{i,j,k}, n_{i,j,k+1}) : i \in \mathbb{N}_p, j \in \mathbb{N}_q, k \in \mathbb{N}_{r-1}\} \quad (7)$$

$$\mathcal{A}_\mathcal{N} = \{a\}$$

$$d_a^\mathcal{N}(n_{i,j,k}) = t_{i,j,k}, i \in \mathbb{N}_p, j \in \mathbb{N}_q, k \in \mathbb{N}_r$$

where A is a symmetric matrix ($\mathcal{DG}$ is undirected) matching the connectivity defined by $\mathcal{E}$. While (7) is defined for 3D tensors, it could easily be extended to higher-order tensors.

`PlasmoData.jl` facilitates both of the representations discussed. In Code Snippet 3, the random tensor can be formed into the mesh structure discussed in Section 2.3.1 by calling the `matrix_to_graph` function (Line 7). This function recognizes that this is a 3D array and creates the mesh structure based on the first two dimensions. The third dimension are the weights on each node. The default name for these weights is the string `"weight"` with a number (e.g., `"weight1"`, `"weight2"`), but a user can define their own names for the attributes in $\mathcal{A}_\mathcal{N}$. Alternatively, the user can call the function

13

tensor_to_graph (Line 10) which performs the second method discussed above. These graphs can be visualized using PlasmoDataPlots.jl as before (Lines 13 - 20). In this case, the node locations are determined automatically within PlasmoDataPlots.jl through the NetworkLayout.jl package. A visualization of the methods in the code snippet is presented in Figure 5. Note that at this time, PlasmoData.jl only automates representing 3D tensors as graphs and not higher-order tensors.

Code Snippet 3: Example of representing a tensor as a `PlasmoData.DataGraph` using two different approaches

```
1   using PlasmoData, Random, PlasmoDataPlots
2
3   Random.seed!(15)
4   random_tensor = rand(4, 5, 6)
5
6   # Convert the tensor to a node weighted graph with 4 × 5 nodes
7   tensor_graph_2d = matrix_to_graph(random_tensor; diagonal = true)
8
9   # Convert the tensor to a node weighted graph 4 × 5 × 6 nodes
10  tensor_graph_3d = tensor_to_graph(random_tensor)
11
12  # Plot the graph
13  plot_graph(
14      tensor_graph_3d,
15      nodesize = 8,
16      linewidth = 4,
17      nodecolor = :grays,
18      node_z = get_node_data(tensor_graph_3d, "weight"),
19      rev = true,
20  )
```

The above graph representations are flexible and can be applied to many types of datasets. For example, videos or space-time data can be represented as a tensor (i.e., a 2D field that changes over a third dimension of time) and data embedded in a 3D space (e.g., temperature in a room) can be stored as a tensor. Color images (e.g., RGB or hyperspectral images) can also be represented as a tensor, where the third dimension corresponds to the light intensity of different color channels.

2.3.3 Symmetric Matrices as Edge-Weighted Graphs

While any matrix can be represented as a node-weighted graph, a symmetric matrix $S \in \mathbb{R}^{p \times p}$ (e.g., a correlation matrix) can also be represented as an edge-weighted graph. We will use the notation $s_{i,j}$ for the entry of the ith row and jth column of S. In this case, $s_{i,j}$ is the edge weight between node i and node j. Under the DataGraph abstraction, this has the form:

$$
\begin{aligned}
&\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \emptyset, \boldsymbol{d}^{\mathcal{E}}, \emptyset, \emptyset, \mathcal{A}_{\mathcal{E}}, \emptyset) \\
&\text{where } \{n_i : i \in \mathbb{N}_p\} \\
&\quad \mathcal{E} = \{(n_i, n_j) : i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_p \setminus \mathbb{N}_i\} \\
&\quad \mathcal{A}_{\mathcal{E}} = \{a\} \\
&\quad d_a^{\mathcal{E}}((n_i, n_j)) = s_{i,j}, i \in \mathbb{N}_{p-1}, j \in \mathbb{N}_p \setminus \mathbb{N}_i
\end{aligned}
\tag{8}
$$

where A is a symmetric matrix ($\mathcal{DG}$ is undirected) matching the connectivity defined by $\mathcal{E}$.

PlasmoData.jl also facilitates this symmetric matrix graph representation. The implementation for this representation is shown in Code Snippet 4. The symmetric matrix is passed to the function
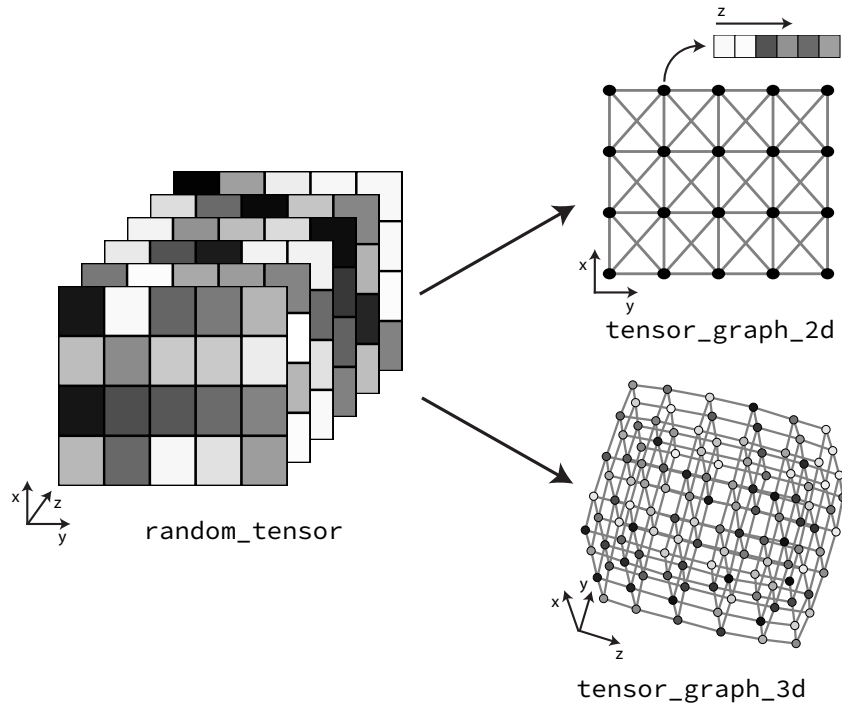
Figure 5: Visualization of graph representations of a tensor. The upper graph corresponds to the tensor_graph_2d PlasmoData.DataGraph object from Code Snippet 3, and the lower graph corresponds to the tensor_graph_3d PlasmoData.DataGraph object from Code Snippet 3

symmetric_matrix_to_graph (Line 10) which forms the edge weighted PlasmoData.DataGraph. The visualization of this is shown in Figure 6. The PlasmoData.DataGraph can also be visualized using PlasmoDataPlots.jl, where we first set the node positions using set_circle_node_positions!, which provides the circular structure shown in Figure 6. The graph is visualized with the plot_graph function (Lines 16 - 23), where we color edges based on the edge weight through the line_z argument and the get_edge_data API from PlasmoData.jl (Line 22).

Code Snippet 4: Example of representing a symmetric matrix as a PlasmoData.DataGraph

```
1   using PlasmoData, Random, PlasmoDataPlots, LinearAlgebra
2
3   # Create symmetric matrix
4   Random.seed!(5)
5   random_matrix = rand(6, 6)
6   symmetric_matrix = (random_matrix .+ random_matrix') / 2
7   symmetric_matrix[diagind(symmetric_matrix)] .= 1
8
9   # Convert symmetric matrix to edge weighted graph
10  symmetric_matrix_graph = symmetric_matrix_to_graph(symmetric_matrix)
11
12  # Set node positions
13  set_circle_node_positions!(symmetric_matrix_graph)
14
15  # Plot the graph
16  plot_graph(
17      symmetric_matrix_graph,
18      nodesize = 12,
19      nodecolor = "gray",
20      linewidth = 5,
21      linecolor = :binary,
22      line_z = get_edge_data(symmetric_matrix_graph, "weight"),
23  )
```
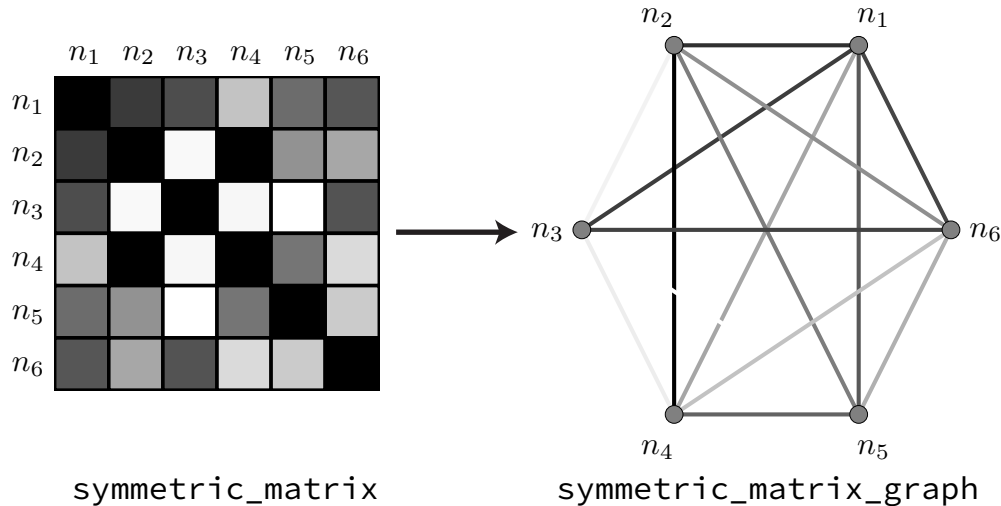

symmetric_matrix → symmetric_matrix_graph

Figure 6: Visualization of a graph representation of a symmetric matrix where the graph corresponds to the symmetric_matrix_graph PlasmoData.DataGraph object from Code Snippet 4. Values of matrix entries are stored as data on the edges of the graph

Data is also commonly found directly as symmetric matrices and the above approach has been

used for different analyses in literature [8, 9, 13, 19]. For instance, symmetric matrices include correlation/covariance matrices (e.g., from a time series), Hessian matrices, or a matrix of distances. The representation of symmetric matrices as graphs can be useful in extracting connectivity information from covariance/correlation matrices, which cannot be extracted using traditional tools such as eigenvalue decompositions(e.g., principal component analysis-PCA).

3 Data Analysis using the `DataGraph` Abstraction

Modeling data as graphs enable the use of tools for processing, manipulating, and analyzing the data. In this section, we highlight how the graph structure can be manipulated to reveal hidden aspects and reduce dimensionality. In addition, we provide an introduction to tools of topological data analysis (TDA) that can be used to extract information from the data model. Finally, we discuss some of the limitations of graph representations.

3.1 Structure Manipulation

Graph structures can be manipulated by filtering out (eliminating) nodes/edges based on data encoded on them or by aggregating sets of nodes. These processes can help reduce the data and can provide valuable insight into the data.

3.1.1 Node and Edge Removal

Perhaps the simplest form of structure manipulation is removal of a single node or edge. To formalize this process for the `DataGraph` model, we define the following operations:

Definition 3.1 (Node-Removal Function $f_{rem}^{\mathcal{N}}$). For a `DataGraph`, $\mathcal{DG}\left(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right)$ where $\mathbb{G}$ is the set of all possible `DataGraph`s and $\mathbb{V}$ is the set of all possible nodes, the node-removal function $f_{\mathcal{N}} : (\mathbb{G}, \mathbb{V}) \to \mathbb{G}$ is defined

$$
\begin{aligned}
f_{rem}^{\mathcal{N}}\left(\mathcal{DG}, n_{rem}\right) &= \mathcal{DG}_r\left(\mathcal{N}_r, \mathcal{E}_r, A_r, \boldsymbol{d}_r^{\mathcal{N}}, \boldsymbol{d}_r^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right) \\
\text{where } \mathcal{N}_r &= \mathcal{N} \setminus \{n_{rem}\} \\
\mathcal{E}_r &= \{(n_i, n_j) : n_i \in \mathcal{N}_r, n_j \in \mathcal{N}_r, (n_i, n_j) \in \mathcal{E}\} \\
\boldsymbol{d}_r^{\mathcal{N}}(n) &= \boldsymbol{d}^{\mathcal{N}}(n), \quad n \in \mathcal{N}_r \\
\boldsymbol{d}_r^{\mathcal{E}}(e) &= \boldsymbol{d}^{\mathcal{E}}(e), \quad e \in \mathcal{E}_r
\end{aligned}
\tag{9}
$$

where $n_{rem} \in \mathcal{N}$ is the node removed and A_r is the adjacency matrix for the new set of edges $\mathcal{E}_r$.

Definition 3.2 (Edge-Removal Function $f_{rem}^{\mathcal{E}}$). For a `DataGraph`, $\mathcal{DG}\left(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right)$ where $\mathbb{E}$ is the set of possible edges, the edge-removal function $f_{rem}^{\mathcal{E}} : (\mathbb{G}, \mathbb{E}) \to \mathbb{G}$ is defined such that

$$
\begin{aligned}
f_{rem}^{\mathcal{E}}\left(\mathcal{DG}, e_{rem}\right) &= \mathcal{DG}_r\left(\mathcal{N}, \mathcal{E}_r, A_r, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}_r^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right) \\
\text{where } \mathcal{E}_r &= \mathcal{E}_r \setminus \{e_{rem}\} \\
\boldsymbol{d}_r^{\mathcal{E}}(e) &= \boldsymbol{d}^{\mathcal{E}}(e), \quad e \in \mathcal{E}_r
\end{aligned}
\tag{10}
$$

where $e_{rem} \in \mathcal{E}$ is the edge to be removed, and A_r is the adjacency matrix for the new set of edges $\mathcal{E}_r$.

The node- and edge-removal functions are simple operations but form the basis of more complex operations that manipulate the graph structure. They also provide a basis for different analysis, such as analyzing the topology of a graph after the removal of a node or edge. The node- and edge-removal functions are implemented in `PlasmoData` as functions `remove_node!` and `remove_edge!`.

3.1.2 Graph Filtration

Filtering out nodes or edges of a graph involves removing nodes or edges whose data does not meet specified (logical) criteria. We formalize the filtration process for `DataGraph`s by defining the following:

Definition 3.3 (Node-Filtration Function $f_{\mathcal{N}}$). For a `DataGraph`, $\mathcal{DG}\left(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right)$, containing a set of node attributes, $\mathcal{A}_{\mathcal{N}}$, such that $\boldsymbol{d}^{\mathcal{N}} \neq \emptyset$, and where $\mathbb{G}$ is the set of all possible `DataGraph`s and $\mathbb{L}$ is the set of all logic sets, the node-filtration function $f_{\mathcal{N}} : (\mathbb{G}, \mathbb{L}) \to \mathbb{G}$ is defined

$$
\begin{aligned}
f_{\mathcal{N}}\big(\mathcal{DG}, \mathcal{L}\big) &= \mathcal{DG}_f\big(\mathcal{N}_f, \mathcal{E}_f, A_f, \boldsymbol{d}_f^{\mathcal{N}}, \boldsymbol{d}_f^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\big) \\
\text{where } \mathcal{N}_f &= \{n : \boldsymbol{d}^{\mathcal{N}}(n) \in \mathcal{L}, n \in \mathcal{N}\} \\
\mathcal{E}_f &= \{(n_i, n_j) : (n_i, n_j) \in \mathcal{E}, n_i \in \mathcal{N}_f, n_j \in \mathcal{N}_f\} \\
\boldsymbol{d}_f^{\mathcal{N}}(n) &= \boldsymbol{d}^{\mathcal{N}}(n), \quad n \in \mathcal{N}_f \\
\boldsymbol{d}_f^{\mathcal{E}}(e) &= \boldsymbol{d}^{\mathcal{E}}(e), \quad e \in \mathcal{E}_f
\end{aligned}
\tag{11}
$$

for the logic set $\mathcal{L}$, where A_f is the adjacency matrix for the edge set $\mathcal{E}_f$.

Definition 3.4 (Edge-Filtration Function $f_{\mathcal{E}}$). For a `DataGraph`, $\mathcal{DG}\left(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right)$, containing a set of edge attributes, $\mathcal{A}_{\mathcal{E}}$, such that $\boldsymbol{d}^{\mathcal{E}} \neq \emptyset$, the edge-filtration function $f_{\mathcal{E}} : (\mathbb{G}, \mathbb{L}) \to \mathbb{G}$ is defined such that

$$
\begin{aligned}
f_{\mathcal{E}}(\mathcal{DG}, \mathcal{L}) &= \mathcal{DG}_f\big(\mathcal{N}, \mathcal{E}_f, A_f \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}_f^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\big) \\
\text{where } \mathcal{E}_f &= \{e : \boldsymbol{d}^{\mathcal{E}}(e) \in \mathcal{L}, e \in \mathcal{E}\} \\
\boldsymbol{d}_f^{\mathcal{E}}(e) &= \boldsymbol{d}^{\mathcal{E}}(e), \quad e \in \mathcal{E}
\end{aligned}
\tag{12}
$$

for the logic set $\mathcal{L}$, where A_f is the adjacency matrix for the edge set $\mathcal{E}_f$.

Here, we refer to a "logic set", $\mathcal{L}$, as a set of data corresponding to the data and attributes on nodes or edges of the graph. For a set of attributes $\{a_1, a_2, ..., a_r\}$, we use the notation

$$
\mathcal{L} = \begin{cases}
a_1 & : \mathcal{Z}_1 \\
a_2 & : \mathcal{Z}_2 \\
\vdots & \vdots \\
a_r & : \mathcal{Z}_r
\end{cases}
$$

where $a_1 : A_1$ indicates that the set $\mathcal{Z}_1 \subseteq \mathcal{D}_{a_1}$ is used for comparing the data of attribute a_1. As the data on the nodes, edges, or graph can be of many forms, $\mathcal{L}$ can span multiple attributes and take many different shapes. Filtration is often applied to scalar weights (e.g., filtering out all nodes whose weight is less than some threshold value); however, filtration can be much more general than the scalar

weight case. For example, the data attached to nodes could be text, matrices, or even an optimization model; as such, the filtration can be performed by using logic defined to attributes of the data (e.g., filtering out any optimization problems that are "nonlinear" or "unconstrained"). In addition, we note that edge-filtrations do not alter the nodes or the node data while the node-filtrations can result in edges being removed when nodes are filtered out.

The above node- and edge-filtrations are implemented in `PlasmoData.jl` as shown in Code Snippet 5 and visualized in Figure 7. The graph defined on Line 6 can be filtered by calling the function `filter_nodes` (the software implementation of $f_{\mathcal{N}}$). This function takes arguments of the `PlasmoData.DataGraph`, a threshold value (often used in the filter function), the attribute name, and an optional keyword argument `fn`, which is the filter function used for the filtration. Note that all nodes where the filter function, `fn`, does not return true are filtered out. On Line 9, we filter out all nodes whose "weight" attribute is less than 0.7 through the `Base.isless` function. On Lines 11 - 13, we define our own function for filtration called `extreme_vals`, which we pass to `filter_nodes` on Line 16 and which filters out all values that are not less than 0.2 or greater than 0.8. This highlights how user-defined functions can be used to filter out a graph. On Line 19, we add random weights to all the edges in the graph (note that `add_edge_dataset!` has a similar function as `add_edge_data`, but instead adds data of a single attribute to multiple edges, whereas the latter function only adds data to a single edge). Once data is defined on the edges, we can likewise perform filtrations based on the edge data, as shown on lines 22 and 25 where we now call `filter_edges` (the software implementation of $f_{\mathcal{E}}$). This function behaves similarly to `filter_nodes`, but now operates on the edge data rather than the node data. All arguments are the same for these functions, except that `filter_edges` requires an attribute defined on the edges rather than on the nodes. While these functions are shown for `PlasmoData.DataGraph`s, they also apply for `PlasmoData.DataDiGraph`s.

Code Snippet 5: Example of filtering a `PlasmoData.DataGraph` by node or edge data

```
1   using PlasmoData, Random
2
3   Random.seed!(15)
4   random_matrix = rand(12, 12)
5
6   matrix_graph = matrix_to_graph(random_matrix; diagonal = true)
7
8   # Filter out/remove nodes with weight >= 0.7
9   filter_nodes_graph1 = filter_nodes(matrix_graph, 0.7, "weight", fn = Base.isless)
10
11  function extreme_vals(a, b)
12      return ((a <= .2) || a >= .8)
13  end
14
15  # Filter out/remove nodes with weight >= 0.2 and <= 0.8
16  filter_nodes_graph2 = filter_nodes(matrix_graph, nothing, "weight", fn = extreme_vals)
17
18  n_edges = length(matrix_graph.edges)
19  add_edge_dataset!(matrix_graph, rand(n_edges), "weight")
20
21  # Filter out/remove edges with weight <= 0.8
22  filter_edges_graph1 = filter_edges(matrix_graph, 0.8, "weight", fn = Base.isgreater)
23
24  # Filter out/remove edges with weight >= 0.5
25  filter_edges_graph2 = filter_edges(matrix_graph, 0.5, "weight", fn = Base.isless)
```
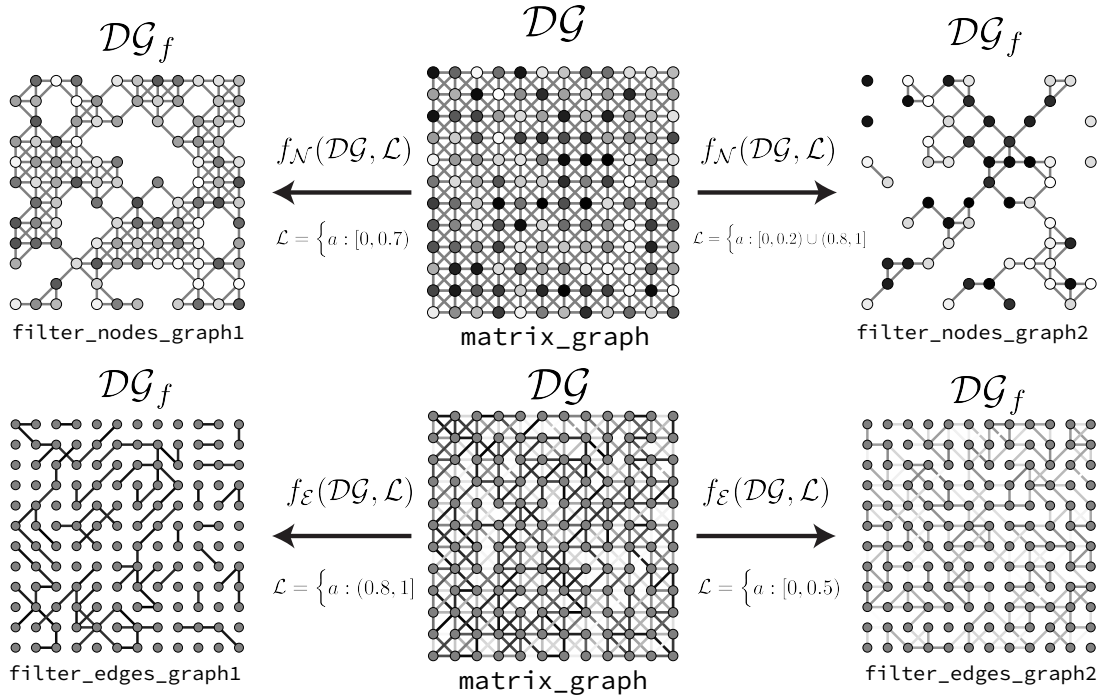
$\mathcal{DG}_f$ $\mathcal{DG}$ $\mathcal{DG}_f$

$f_{\mathcal{N}}(\mathcal{DG}, \mathcal{L})$ $f_{\mathcal{N}}(\mathcal{DG}, \mathcal{L})$

$\mathcal{L} = \left\{ a : [0, 0.7) \right\}$ $\mathcal{L} = \left\{ a : [0, 0.2) \cup (0.8, 1] \right\}$

filter_nodes_graph1   matrix_graph   filter_nodes_graph2

$\mathcal{DG}_f$ $\mathcal{DG}$ $\mathcal{DG}_f$

$f_{\mathcal{E}}(\mathcal{DG}, \mathcal{L})$ $f_{\mathcal{E}}(\mathcal{DG}, \mathcal{L})$

$\mathcal{L} = \left\{ a : (0.8, 1] \right\}$ $\mathcal{L} = \left\{ a : [0, 0.5) \right\}$

filter_edges_graph1   matrix_graph   filter_edges_graph2

Figure 7: Visualization of different examples of node or edge filtration associated with Code Snippet 5, where different logic sets are used for removing nodes or edges of the graph

20

3.1.3 Graph Aggregation

Aggregation is a useful structural manipulation that can help reduce the dimensionality of the graph and enable scalable analysis. Graph aggregation is also often referred to as coarsening. For a `DataGraph` $\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}})$, aggregation combines a subset of nodes $\bar{\mathcal{N}} \subseteq \mathcal{N}$ into a new node, n_{agg}. This aggregation can also require aggregation of node and/or edge data. We formalize the aggregation process for `DataGraph`s by defining the following:

Definition 3.5 (Aggregation Function f_+). For an undirected `DataGraph`
$\mathcal{DG}\left(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right)$ and a subset of nodes to be aggregated, $\bar{\mathcal{N}} \in \mathcal{N}$, the aggregation function collapses the node set into a single new node and aggregates node and edge data where applicable. The aggregation function $f_+ : (\mathbb{G}, \mathbb{V}, \mathbb{F}, \mathbb{F}) \to \mathbb{G}$ (where $\mathbb{V}$ is the set of all possible nodes and $\mathbb{F}$ is the set of all possible functions for aggregating sets of node or edge data) is defined as

$$f_+(\mathcal{DG}, \mathcal{N}', f_+^{\mathcal{N}}, f_+^{\mathcal{E}}) = \mathcal{DG}_{agg}\left(\mathcal{N}_{agg}, \mathcal{E}_{agg}, A_{agg}, \boldsymbol{d}_{agg}^{\mathcal{N}}, \boldsymbol{d}_{agg}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}}\right)$$

$$\text{where } \mathcal{N}_{agg} = \{n_{agg}\} \cup \mathcal{N} \setminus \bar{\mathcal{N}}$$

$$\bar{\mathcal{E}}_{agg} = \{(n_i, n_{agg}) : n_i \in \mathcal{N} \setminus \bar{\mathcal{N}}, n_j \in \bar{\mathcal{N}}, (n_i, n_j) \in \mathcal{E} \vee (n_j, n_i) \in \mathcal{E}\}$$

$$\mathcal{E}_{agg} = \{(n_i, n_j) : n_i \notin \bar{\mathcal{N}}, n_j \notin \bar{\mathcal{N}}, (n_i, n_j) \in \mathcal{E}\} \cup \bar{\mathcal{E}}_{agg}$$

$$\boldsymbol{d}_{agg}^{\mathcal{N}}(n) = \boldsymbol{d}^{\mathcal{N}}(n), \quad n \in \mathcal{N} \setminus \bar{\mathcal{N}} \tag{13}$$

$$\boldsymbol{d}_{agg}^{\mathcal{N}}(n_{agg}) = f_+^{\mathcal{N}}(\boldsymbol{d}^{\mathcal{N}}, \bar{\mathcal{N}})$$

$$\boldsymbol{d}_{agg}^{\mathcal{E}}(e) = \boldsymbol{d}^{\mathcal{E}}(e), \quad e \in \{(n_i, n_j) : (n_i, n_j) \in \mathcal{E}, n_i \notin \bar{\mathcal{N}}, n_j \notin \bar{\mathcal{N}}\}$$

$$\boldsymbol{d}_{agg}^{\mathcal{E}}((n_i, n_{agg})) = f_+^{\mathcal{E}}\big(\boldsymbol{d}^{\mathcal{E}}, \{(n_i, n_j) : n_j \in \bar{\mathcal{N}}, (n_i, n_j) \in \mathcal{E}\} \cup$$

$$\{(n_j, n_i) : n_j \in \bar{\mathcal{N}}, (n_j, n_i) \in \mathcal{E}\}\big) \quad (n_i, n_{agg}), \in \bar{\mathcal{E}}_{agg}$$

where $f_+^{\mathcal{N}}$ is a function that aggregates node data in $\boldsymbol{d}^{\mathcal{N}}$ for nodes $\bar{\mathcal{N}}$ into node data for a single node (n_{agg}) and $f_+^{\mathcal{E}}$ is a function that aggregates edge data in $\boldsymbol{d}^{\mathcal{E}}$ for the set of edges being passed to the function, and A_{agg} is the adjacency matrix for edge set $\mathcal{E}_{agg}$.

In the above definition, we introduce two new functions, $f_+^{\mathcal{N}}$ and $f_+^{\mathcal{E}}$. These functions can take many forms because of the variety of data that can be stored on the nodes and edges. In addition, the function for aggregating edge data, $f_+^{\mathcal{E}}$, is only needed when there are multiple nodes in $\bar{\mathcal{N}}$ connected to some node in $\mathcal{N} \setminus \bar{\mathcal{N}}$; in this case, there are multiple edges that will be replaced by a single edge for undirected graphs. We also note that the aggregation function for directed graphs is different because the order of the edges will matter; consequently, rather than the function $f_+^{\mathcal{E}}$ acting on the combined set of edges $\{(n_i, n_j) : n_j \in \bar{\mathcal{N}}, (n_i, n_j) \in \mathcal{E}\}$ and $\{(n_j, n_i) : n_j \in \bar{\mathcal{N}}, (n_j, n_i) \in \mathcal{E}\}$, the function could need to be applied to multiple, independent sets. In addition, the set $\bar{\mathcal{E}}$ would need to be expanded to consider the edges where n_{agg} could be the source node (not just the destination) for an aggregated edge. For both directed and undirected graphs, $f_+^{\mathcal{N}}$ and $f_+^{\mathcal{E}}$ must be permutation invariant.

`PlasmoData.jl` implements aggregation for graph objects, as shown in Code Snippet 6 and Figure 8. Here, a node set is defined in Line 9, and the node set is aggregated in line 12 using the function `aggregate`. The default for aggregating node data or edge data is to average the values of the data for

each attribute, but one can also use, for instance, a max operator (as done in graph neural networks). The user can use these different functions by passing the function to the key word arguments `node_fn` and `edge_fn`.

Code Snippet 6: Example of aggregating a set of nodes in a `PlasmoData.DataGraph`

```
1  using PlasmoData, Random
2
3  Random.seed!(15)
4  random_matrix = rand(12, 12)
5
6  matrix_graph = matrix_to_graph(random_matrix; diagonal = true)
7
8  # Define nodes to be aggregated
9  nodes_for_aggregation = [(3, 7), (3, 8), (3, 9), (4, 7), (4, 8)]
10
11 # Aggregate nodes in the graph
12 aggregated_graph = aggregate(matrix_graph, nodes_for_aggregation, "agg_node")
```


matrix_graph   aggregated_graph

Figure 8: Visualization associated with Code Snippet 6 where a subset of nodes is aggregated into a single new node with the node weights on the aggregated node averaged to get a new node weight for the new node

There is an extension of (13) that could be considered. Under (13), any data on the edges between nodes in $\bar{\mathcal{N}}$ is lost, but such data could be important in different applications. We could extend (13) by defining a function $f_+^{\mathcal{E} \to \mathcal{N}}$ that reduces $\boldsymbol{d}^{\mathcal{E}}(e), e \in \{(n_i, n_j) : (n_i, n_j) \in \mathcal{E}, n_i \in \bar{\mathcal{N}}, n_j \in \bar{\mathcal{N}}\}$ into new data on n_{agg} for an additional set of attributes, $\mathcal{A}_{\mathcal{E} \to \mathcal{N}}$. Such attributes would not be restricted to be in either $\mathcal{A}_{\mathcal{N}}$ or $\mathcal{A}_{\mathcal{E}}$. This would result in a slightly different `DataGraph` such that $\mathcal{DG}_{agg}(\mathcal{N}_{agg}, \mathcal{E}_{agg}, A_{agg}, \boldsymbol{d}_{agg}^{\mathcal{N}}, \boldsymbol{d}_{agg}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}} \cup \mathcal{A}_{\mathcal{E} \to \mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}})$, where $\boldsymbol{d}_{agg}^{\mathcal{N}}$ now contains additional data for $\mathcal{A}_{\mathcal{E} \to \mathcal{N}}$. `PlasmoData.jl` does not (yet) support this extension, so we have restricted (13) to follow the implementation of `PlasmoData.jl`, but this could be an area of future work.

note that aggregation here refers to reducing the number of nodes in the graph and requires a defined set of nodes to be aggregated. The definition for fN+ and fE+ must be permutation invariant; in addition,

3.2 Graph Structural Analysis

We now discuss different tools for analyzing and quantifying the structure of the `DataGraph` model, such as Topological Data Analysis (TDA) and community detection. The implementation of these procedures (or interfacing to software that conducts these procedures) is greatly facilitated by the proposed graph abstraction. These procedures can also be combined with machine learning tools to enable supervised and unsupervised learning tasks (e.g., for classifying and clustering graphs or for predicting emerging properties from graphs). In addition, these graph analysis tools are often enhanced by the ability to manipulate the graph structure as discussed above.

Community detection and clustering in graphs focuses on how to identify organization within a graph, typically by identifying communities/clusters of nodes that have several connections within that grouping but relatively few connections to nodes outside of the grouping [40, 41]. Community detection has been applied to a variety of problems including optimization programs represented as graphs [42, 43], biological networks [44], and fraud detection [45]. Identifying communities can give insight into the data. In addition, there are numerous algorithms for identifying communities [41], and different algorithms will identify different communities. Some of these algorithms (such as clique percolation [46], label propagation [47], or Newman's modularity [48]) are already implemented within the `Graphs.jl` package and are thus are easily accessible by `PlasmoData.jl`.

TDA is a growing field in data science that develops tools for analyzing and quantifying the shape/structure of data objects [49, 50, 51, 52]. Graphs are topologically invariant objects that can be approached through the lens of TDA. Many data objects may not have an inherent topology in their mathematical definition (e.g., matrices); modeling these types of data as graphs enables TDA applications that otherwise might not be possible. Many TDA tools can also be combined with the structural manipulation based on data (e.g., graph filtration) to elucidate further insights into the data. We now outline several TDA concepts and tools that can be used to analyze and quantify the shape/structure of graphs.

3.2.1 Graph Connectivity

There are several general metrics/descriptors describing graph connectivity which could be considered within TDA. These descriptors could be used to help describe a graph, or they could be used in combination with the structure manipulation (e.g., analyzing these metrics after performing filtration). A few of these descriptors include:

- **Paths** - In graphs, a walk is a way for moving from one node to another node (via edges), and a path is a walk in which no node appears more than once [53]. There are existing algorithms for finding the shortest path between a couple of nodes or for finding whether nodes are connected. The paths in a graph could yield information about the graph and its data. For example, paths are frequently used for analyzing biological systems [54, 55, 56]. As a further example, in a supply chain problem, we may be interested in finding paths (or the number of paths) between

raw materials and products (see for example [57, 58]), as well as how those paths change if nodes or edges are filtered out. In addition, we may generally be interested in how the length of the shortest path between two nodes changes as the graph topology changes (e.g., via filtration). `PlasmoData.jl` provides functions for analyzing paths by extending functions within `Graphs.jl`, such as the `has_path` or `get_path` functions.

- **Cycles** - A cycle is a walk which starts and ends at the same node using entirely distinct edges. Cycles are used in many graph analyses, including construction [59] and electrical networks [60, 61, 62]. The number of cycles or length of cycles could be used in analyzing data represented as graphs. The function `cycle_basis` is extended from `Graphs.jl` for `PlasmoData.DataGraph`s.

- **Connected Components** - Connected components are sets of nodes that are all connected via paths. The number of connected components can describe how "connected" a graph may be, and this metric could give insight into how connected or separated the data may be. The function `connected_components` is extended from `Graphs.jl` for `PlasmoData.DataGraph`s.

- **Node Degree** - The node degree is the number of nodes a node is connected to (i.e., the number of neighbors of a given node). The average node degree can also be computed for an entire graph. This metric has been used in node ranking [63] and sensor analysis [64, 65].

- **Diameter** - The diameter of a graph is the length of the shortest path connecting the most distanced nodes (i.e., the maximum shortest path length for any two nodes in a graph). This metric also provides an idea of the "connectedness" of the data, and it has been used in analyzing the degree of separation for different entities [66]. The function `diameter` is extended from `Graphs.jl` for `PlasmoData.DataGraph`s.

The above metrics can yield important insights into data. These metrics may also change after filtration/aggregation of nodes and edges, which could likewise elucidate information about the dataset (e.g., aggregating a graph can help uncover features that are not apparent when dealing with the full-resolution graph). The `DataGraph` abstraction therefore provides a framework for obtaining to topological descriptors.

3.2.2 Euler Characteristic

The EC is a valuable descriptor for topological objects, including graphs. The EC is a scalar integer value that summarizes the shape of a topological space [19, 67]. For a graph, this can be defined as [19]:

$$\mathcal{X} = \# \text{ Connected Components} - \# \text{ Holes} = |\mathcal{N}| - |\mathcal{E}| = \beta_0 - \beta_1 \tag{14}$$

where β_i are the ith betti numbers of the graph, the "holes" are equivalent to cycles in the graph, and $|\cdot|$ is the cardinality of the set. We will use the notation $EC(\mathcal{DG})$ for the EC of a `DataGraph` $\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}})$.

The EC has also been combined with filtration of node- or edge-weighted graphs to create an EC curve [19, 68]. The EC curve is formed by filtering out nodes and/or edges at different threshold values

24

(e.g., removing all nodes with a weight value less than some threshold value) and computing the EC for the filtered graph. The shape of the EC curve can give insight into how the shape of the data changes with the filtration (e.g., where connected components dominate or where holes dominate). For instance, the EC curve has been used for analysis of liquid crystal sensors and MD data [19, 68, 69]. In Section 4.1, we will show how the EC curve can be used in a classification model with real data.

For a `DataGraph`, we can generalize the EC curve for a filtration process for node filtration as follows (edge filtration has a similar definition but for $f_{\mathcal{E}}$ instead of $f_{\mathcal{N}}$). For a `DataGraph`, $\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \boldsymbol{d}^{\mathcal{G}}, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \mathcal{A}_{\mathcal{G}})$, with $\mathcal{A}_{\mathcal{N}} \neq \emptyset$ and $\boldsymbol{d}^{\mathcal{N}} \neq \emptyset$, there exist a sequence of logic sets $\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_r$ such that the filtration forms a sequence of nested subgraphs given in (15) associated with the logic sets, where $\mathcal{DG}_{\mathcal{L}_i} = f_{\mathcal{N}}(\mathcal{DG}, \mathcal{L}_i)$. For a graph to be considered a "nested subgraph" of another graph, the nodes and edges of the "nested subgraph" must be subsets of the nodes and edges, respectively, of the other graph.

$$\mathcal{DG}_{\mathcal{L}_1} \subseteq \mathcal{DG}_{\mathcal{L}_2} \subseteq ... \subseteq \mathcal{DG}_{\mathcal{L}_r} \tag{15}$$

The EC curve can then be given by the vector $(EC(\mathcal{DG}_{\mathcal{L}_1}), EC(\mathcal{DG}_{\mathcal{L}_2}), ..., EC(\mathcal{DG}_{\mathcal{L}_r}))$. This approach for generating an EC curve is more general than the idea of using threshold values because the `DataGraph` could include a variety of data types. In the case of single scalar node or edge weights where all weights are in $[0, 1]$, the logic sets could, for example, be a series of ranges such as $[0, 0.1], [0, 0.2], ..., [0, 1]$ (thus effectively filtering out all nodes with weights greater than a single value). However, an EC curve of sorts could be generated for more complex data, such as considering data with multiple weights. For `PlasmoData.DataGraph`s with scalar data for each attribute, computing the EC curve can be done with the functions `run_EC_on_nodes` and `run_EC_on_edges`.

4   Case Studies

We now provide case studies that highlight different applications and benefits of the `DataGraph` abstraction. All scripts and data necessary to replicate the results in this section are available at https://github.com/zavalab/JuliaBox/tree/master/PlasmoData_examples. These examples include image analysis (representing matrices and tensors as graphs), multivariate time series analysis (representing symmetric matrices as graphs), and connectivity analysis (analyzing pathways within graphs).

4.1   Image Analysis

We consider a case study arising in image classification and show how the `DataGraph` abstraction and `PlasmoData.jl` help integrate with machine learning tools and facilitate feature extraction. We will show how these images of surfaces can be analyzed by representing these images as 3D tensors the representing these tensors using the `DataGraph` abstraction. We will show how we use `PlasmoData.jl` for extracting features from this data and use those features to train a machine learning model. We will also highlight how `PlasmoData`'s data structure can be employed for GNNs.

The images we consider are of chemoresponsive liquid crystals (LCs) which have garnered increased interest because of their potential as sensors for gas contaminants [70, 71, 72]. LCs have been shown

to elicit different, visible chemical responses to varying chemical environments as the LCs change their orientation ordering [73, 74]. One of the challenges that naturally arises from these sensors is identifying the chemical environment surrounding a LC based on the appearance of the LC's surface. We are therefore interested in being able to classify images of LCs based on the chemical environment into which the LC has been placed. The dataset we consider are images of LCs exposed to four different concentrations of sulfur dioxide (SO_2): 0.5, 1.0, 2.0, and 5.0 ppm (i.e., four classes of images corresponding to each concentration). Each image is 134×134 pixels, and there are 72 images in each class (288 total). Examples of images from each class can be seen in Figure 9 [69, 73].

Figure 9: Images of LCs exposed to gaseous environments with different concentrations of SO_2 [69, 73]

4.1.1 Classification using TDA and SVMs

To classify these images, we will represent these images as tensors and then each tensor as a graph. Each of the 288 images can be treated as a tensor, $T \in [0,1]^{134 \times 134 \times 3}$. Under the `DataGraph` abstraction, we can represent the ith tensor as a graph with (16) using the first method shown in Section 2.3.2, where the third dimension of the tensor represents the red, green, and blue channels of the pixel intensity and where $t_{j,k,l}$ is the j, k, l entry of T.

26

$$\mathcal{DG}_i(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \emptyset, \emptyset, \mathcal{A}_{\mathcal{N}}, \emptyset, \emptyset)$$

$$\text{where } \mathcal{N} = \{n_{j,k} : j \in \mathbb{N}_{134}, k \in \mathbb{N}_{134}\}$$

$$\mathcal{E} = \{(n_{j,k}, n_{j+1,k}) : j \in \mathbb{N}_{133}, k \in \mathbb{N}_{134}\} \cup \{(n_{j,k}, n_{j,k+1}) : j \in \mathbb{N}_{134}, k \in \mathbb{N}_{133}\} \cup$$

$$\{(n_{j,k}, n_{j+1,k+1}) : j \in \mathbb{N}_{133}, k \in \mathbb{N}_{133}\} \cup \{(n_{j,k+1}, n_{j+1,k}) : j \in \mathbb{N}_{133}, k \in \mathbb{N}_{133}\} \tag{16}$$

$$\mathcal{A}_{\mathcal{N}} = \{a_{red}, a_{green}, a_{blue}\}$$

$$d^{\mathcal{N}}_{a_{red}}(n_{j,k}) = t_{j,k,1}, j \in \mathbb{N}_{134}, k \in \mathbb{N}_{134}$$

$$d^{\mathcal{N}}_{a_{green}}(n_{j,k}) = t_{j,k,2}, j \in \mathbb{N}_{134}, k \in \mathbb{N}_{134}$$

$$d^{\mathcal{N}}_{a_{blue}}(n_{j,k}) = t_{j,k,3}, j \in \mathbb{N}_{134}, k \in \mathbb{N}_{134}$$

where A is a symmetric matrix ($\mathcal{DG}$ is undirected) matching the connectivity defined by $\mathcal{E}$.

With the `DataGraph`, $\mathcal{DG}_i$, defined for each image, we can generate an EC curve (or a set of EC curves) for the graph. We will generate an EC curve for the data for each attribute in $\mathcal{A}_{\mathcal{N}}$. Because we have scalar weights, we will filter the nodes by removing nodes that are greater than a threshold value, and we will do this for a series of threshold values. Under the formulation given in Section 3.2.2, we can define a sequence of logic sets for the data corresponding to each attribute in $\mathcal{A}_{\mathcal{N}}$ such that

$$\mathcal{L}_{j,a_{red}} = \begin{cases} a_{red} & : [0, (j-1) \times 0.005] \\ a_{green} & : [0, 1] \\ a_{blue} & : [0, 1] \end{cases}, \qquad j \in \mathbb{N}_{201}$$

$$\mathcal{L}_{j,a_{green}} = \begin{cases} a_{red} & : [0, 1] \\ a_{green} & : [0, (j-1) \times 0.005] , \\ a_{blue} & : [0, 1] \end{cases} \qquad j \in \mathbb{N}_{201}$$

$$\mathcal{L}_{j,a_{blue}} = \begin{cases} a_{red} & : [0, 1] \\ a_{green} & : [0, 1] \\ a_{blue} & : [0, (j-1) \times 0.005] \end{cases}, \qquad j \in \mathbb{N}_{201}$$

One approach to classify these images is to feed a topological descriptor of the weighted graph (e.g., the EC curve) to a machine learning model. We generate an EC curve for the ith image and for each color channel, and we denote these curves for attribute (color channel) a as

$$\boldsymbol{\mathcal{X}}_{i,a} = (EC(f_{\mathcal{N}}(\mathcal{DG}_i, \mathcal{L}_{1,a})), EC(f_{\mathcal{N}}(\mathcal{DG}_i, \mathcal{L}_{2,a})), ..., EC(f_{\mathcal{N}}(\mathcal{DG}_i, \mathcal{L}_{201,a}))).$$

To perform the classification for the ith image, we concatenate the EC curves $\boldsymbol{\mathcal{X}}_{i,a_{red}}, \boldsymbol{\mathcal{X}}_{i,a_{green}}$, and $\boldsymbol{\mathcal{X}}_{i,a_{blue}}$ into a vector $\boldsymbol{\mathcal{X}}_i$ and train a linear support vector machine (SVM) on sets of $\boldsymbol{\mathcal{X}}_i$ for $i \in \mathbb{N}_{288}$.

The code for performing the tasks of generating the graph representations, building the EC curves, and training an SVM can be seen in Code Snippet 7. The data is in the format of a 4D array of size $(288, 134, 134, 3)$. The graph is formed by treating each pixel of the image (dimensions 2 and 3 of the

27

array) as a node using `PlasmoData.jl`'s `matrix_to_graph` function (Line 17). The intensity values for each RGB channel (dimension 4 of the array) are saved as weights on each node of the graph. After each graph is formed, we then find the EC curve for each weight (channel) on the graph and store this in the array `so2_ECs` (Lines 19 - 23). The three EC curves (one for each channel) are then concatenated into a vector and used to train a linear SVM using 5-fold cross validation (CV) (Lines 29 - 49).

Code Snippet 7: Classifying images from the EC using SVMs

```
1   using PlasmoData, JLD, MLUtils, LIBSVM
2
3   ### Construct matrices as graphs and get EC curves ###
4
5   # Load in the SO2 data; so2_data is size (288, 134, 134, 3)
6   so2_data = load("so2_data.jld")["data"]
7   so2_classes = load("so2_classes.jld")["classes"]
8
9   # Define threshold range for each EC curve
10  thresh = 0:.005:1
11
12  # Define a matrix for the EC curves; EC curves will be concatenated
13  so2_ECs = Array{Float64, 2}(undef, (length(thresh)*3, 288))
14
15  for i in 1:288
16      # Build a graph from a 3—D array (134 × 134 × 3)
17      mat_graph = matrix_to_graph(so2_data[i, :, :, :])
18
19      for j in 1:3
20          # Iterate through each channel and concatenate the EC curve
21          range_bounds = (1 + (j - 1) * length(thresh)):(j * length(thresh))
22          so2_ECs[range_bounds, i] = run_EC_on_nodes(mat_graph, thresh, "weight$j", false)
23      end
24  end
25
26  ### Perform 5 fold CV with SVMs on EC data ###
27
28  # shuffle data
29  Xs, ys = shuffleobs((so2_ECs, so2_classes))
30
31  # define a function for calculating accuracy
32  function get_accuracy(yhat, ytest)
33      num_errors = 0
34      for i in 1:length(yhat)
35          if yhat[i] != ytest[i]
36              num_errors += 1
37          end
38      end
39      return 1 - num_errors/length(yhat)
40  end
41
42  # Perform 5—fold CV
43  accuracy_values = []
44  for (train_data, val_data) in kfolds((Xs, ys); k = 5)
45      model = svmtrain(train_data[1], train_data[2], kernel = Kernel.Linear)
46      yhat, decision_values = svmpredict(model, val_data[1]
47      accuracy = get_accuracy(yhat, val_data[2])
48      push!(accuracy_values, accuracy)
49  end
```

The method can effectively classify these images in a reasonable time. The classification accuracy, based on 5-fold CV, was 94.8% with a standard deviation of 2.1%. Running this script on an Intel(R) Core i9-10885H (2.40GHz) processor on a single thread with Julia 1.7.3 resulted in a data processing time (Lines 10 - 24) of 36.2 s and a 5-fold CV (Lines 29 - 49) time of 0.09 s. This is comparable to what

Jiang and co-workers [69] found, where they had 87.5% accuracy from other topological methods that took less time to process and train (~ 14 s). For comparison, Jiang and co-workers [69] also trained a convolutional neural network (CNN) for classifying these images, and the CNN was able to classify with 95.1% accuracy while taking 154 seconds to train. Here, we do not aim to show superiority of one method over another or provide a rigorous comparison in processing time but rather to provide a general validation as to whether our methods yield reasonable results. In particular, our results suggest that it is possible to obtain high accuracy while performing these operations in a competitive time.

We also note—as Jiang and co-workers [69] did in their study—that there are benefits to topological methods such as those we followed above. First, the linear SVM above had relatively few parameters—each EC curve contained 201 points before concatenation, so there were only 603 parameters in the SVM. In contrast, CNNs can use high numbers of parameters (thousands to millions). They also can be sensitive to rotation, whereas graphs are rotationally invariant (e.g., the EC curve does not depend on the orientation of the graph). In addition, Jiang and co-workers [69] note that the CNN in their study was trained on a GPU, whereas our above methods (and Jiang and co-workers [69] TDA methods) functioned efficiently on a CPU.

A further benefit of TDA is that the results can be interpretable and potentially provide insight into the data. For example, the EC curve can reveal how the topology of a graph is changing throughout a filtration. For example, we show in Figure 10 an example of what the filtration looks like for an example image from each class. In addition, the EC curves for the three different channels are shown in Figure 11. In Figure 10, different topologies are evident at different filtration thresholds. For example, the red channel of the 0.5 ppm SO_2 class has several small, scattered holes throughout, and its corresponding EC curve (Figure 11) is less steep than the 2.0 ppm SO_2 class which exhibits larger and fewer holes in the topology. The channels also exhibit slightly different topologies at different filtration levels and can thus provide different information in differentiating the images. For example, the average EC curves for the 0.5 and 5.0 ppm SO_2 classes (Figure 11) are almost on top of each other for most of the blue channel, whereas the two curves are more offset within the red channel.

The EC curve for each graph (and each channel) could provide insight into the image. Because the EC of a graph is only the number of nodes minus the number of edges, the EC curve begins at the origin but can end far from the origin. In the case of images, the full graph (i.e., when nothing is filtered out) has a large negative EC since the number of edges is much more than the number of nodes. Thus, the EC curve of a graph tends to show when and how quickly edges and cycles begin to appear. For example, in the case of images, if the filtration results mostly in nodes that are far apart, there are few if any edges between them and the EC is more positive. In contrast, if the filtration results in groups of clustered nodes, there are more edges connecting nodes and the EC is more negative. This helps explain why the EC curves for the 0.5 ppm SO_2 class tend to be more positive at a given threshold value than the other classes (Figure 11) since, as can be seen in Figure 10, the filtration results in more spread out, small clusters at the lower threshold values for the 0.5 ppm SO_2 class than the other classes.
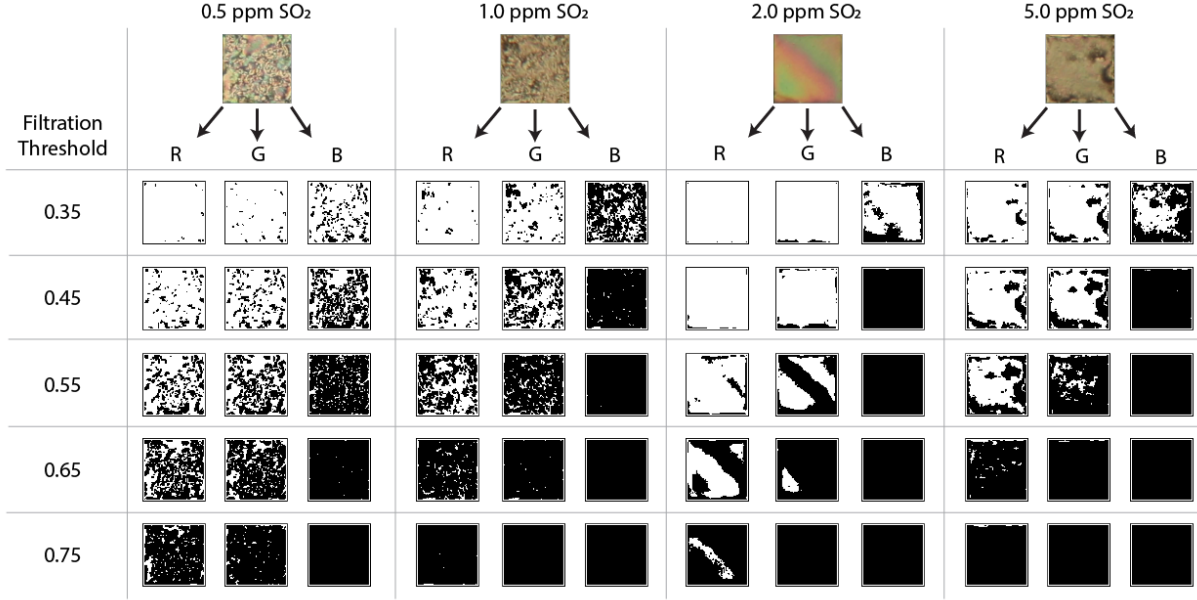
29

Figure 10: Examples by class of graph filtrations at different threshold values. Each image has a value corresponding to the RGB channels, and these channels result in different topologies at the filtration thresholds. The EC curve captures these different topologies across channels and across classes. Data comes from the work of Bao and co-workers [73] and Jiang and co-workers [69]

The EC curves also reveal some minor differences across color channels, and this helps highlight how the multiple node weights within a graph can be used for more effectively analyzing the data. If only gray-scale images are used, the classification accuracy of these images is at least 10% lower (see the Supporting Information). We can thus analyze our graph structure using different weights to gain different information. One area that could particularly benefit from this approach is hyperspectral imaging [75, 76]. In hyperspectral images, rather than recording only 3 channels of colors, each image can contain several (10s - 100s) weights at each pixel corresponding to different wavelengths of light. Thus, it is possible for each node of the graph to contain dozens or hundreds of weights that could be analyzed. `PlasmoData.jl` is able to readily handle this type of data.

In this example, we have described how `PlasmoData.jl` can be used to combine TDA with machine learning. There are more details that could be valuable on these topics which we include in the supporting information, including i) a comparison of memory allocation between `PlasmoData.jl`, `Networkx`, `MetaGraphs.jl`, and `Matlab`'s graph function for the image application; and ii) a brief discussion of how the graph structure and data impact classification results. In the above example, if gray-scale data is used instead of colored data (i.e., one weight instead of three weights are stored on each node), the classification accuracy decreases.
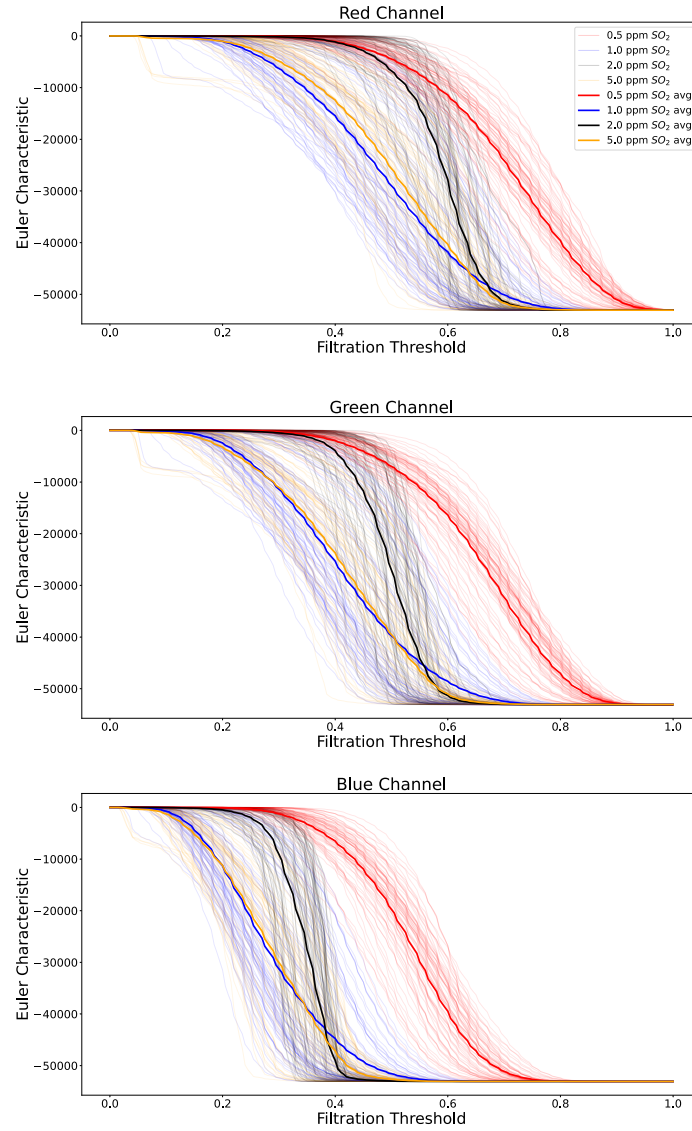
Figure 11: EC Curves for the RGB channels of four classes of images for LC data from Bao and co-workers [73] and Jiang and co-workers [69]

4.1.2   Classification using Graph Neural Networks

An alternative method to TDA+SVM for graph classification are convolutional GNNs. `PlasmoData.jl` facilitates the implementation of this method and enables systematic comparisons with other approaches. Since the data for nodes and edges is stored as a matrix within the `PlasmoData.DataGraph` (or `DataDiGraph`) structure, we can easily query this data to create the needed structure for GNNs. In Code Snippet 8, we show how our data for the image classification can be easily transformed into a format used by a GNN. In this example, we use the packages `GraphNeuralNetworks.jl` [31] and `Flux.jl` [77]. After the data is loaded, we build the `PlasmoData.DataGraph` as before from the image data (Line 12). The `PlasmoData.DataGraph` can then be passed to the `GNNGraph` function from `GraphNeuralNetworks.jl` (Lines 13 - 16). This function takes as an argument a `Graphs.SimpleGraph` (or `Graphs.SimpleDiGraph`) to provide the adjacency lists, and a matrix of node and/or edge data. Since the `PlasmoData.DataGraph` stores a `Graphs.SimpleGraph`, this is readily provided to the `GNNGraph` function. In addition, the node or edge data is provided by calling the API function `get_node_data` (Line 15). The classes are also passed to `Flux.onehotbatch` (Line 21) to provide a one hot encoded array that can be used by the GNN.

Code Snippet 8: Interfacing `PlasmoData.jl` with GNNs

```
 1  using PlasmoData, JLD, Flux, GraphNeuralNetworks
 2
 3  function getdataset()
 4      # Load in data
 5      so2_data = load("so2_data.jld")["data"]
 6      so2_classes = load("so2_classes.jld")["classes"]
 7
 8      GNN_graphs = []
 9
10      # Create GNNGraph from DataGraph
11      for i in 1:288
12          mat_graph = matrix_to_graph(data[i, :, :, :])
13          gnn_graph = GraphNeuralNetworks.GNNGraph(
14              mat_graph.g,
15              ndata = get_node_data(mat_graph)'
16          )
17          push!(GNN_graphs, gnn_graph)
18      end
19
20      # One hot encode classes
21      y = Flux.onehotbatch(so2_classes, 1:4)
22
23      return GNN_graphs, y
24  end
```

With the data formatted as `GNNGraph`s, we can build a full GNN through `GraphNeuralNetworks.jl`. We ran our script on an AMD EPYC 7302 16-core processor with access to a NVIDIA Quadro RTX 6000 GPU (used for training the GNN). Using the same 288 images as before, we were able to classify the images with 89.2% accuracy with a standard deviation of 5.1% based on 5-fold CV. While the time for data processing was short (3.8 s), the GNN did take a long time to train and perform CV (947 s).

The ability to use `PlasmoData.DataGraph`s for GNNs is a powerful capability. This framework

for building GNNs from `PlasmoData.DataGraph`s could be applied much more broadly than just to images (e.g., molecular property prediction). The user has significant flexibility in constructing their own graph data and using GNNs with that data, enabling application of `PlasmoData.jl` to a variety of fields or use cases. For further details on constructing a `GNNGraph` from data using `PlasmoData.jl`, see the Supporting Information.

4.2   Multivariate Time Series Analysis

In this example, we illustrate how multivariate time series can be represented as graphs, and we highlight how the topology of this graph representation can provide interesting insights. Recently, de Souza and co-workers [13] explored the use of TDA (and in particular the EC) as a potential tool for data-driven surveillance of epidemic outbreaks. Specifically, they were interested in finding indicators or fingerprints of when an outbreak is occurring (or likely to occur), and they based their efforts on locally reported disease data and on simulated data. In their work, they constructed correlation matrices as graphs, filtered these graphs, and looked at the resulting topology. They identified that there is a strong correlation between the EC curve and epidemic outbreaks, and they suggest that TDA could be used as a tool in disease surveillance. In this section, we will recreate some of their results using the `DataGraph` abstraction and using `PlasmoData.jl`, and we will suggest some other topological metrics that could be useful in analyzing datasets like these. In doing this, we emphasize that our purpose in this example is to highlight how data can be modeled with graphs and how `PlasmoData.jl` facilitates much of this analysis.

First, we highlight the general data and methodology of de Souza and co-workers [13], but point the reader to their original paper to find an in depth explanation [13]. For this analysis, we will use their data from Recife, Brazil of new daily dengue cases from 93 individual districts from 2014 - 2021 (Figure 12). For each seven-day moving window of data, they computed the Pearson correlation for the 93 time series (one for each district) and then constructed the edge-weighted graph as outlined earlier (see for example Figure 6). They then computed a characteristic threshold value (what they call a critical percolation value) for this edge-weighted graph. This characteristic threshold value is the maximum possible threshold value that does not change the number of connected components from the number in the original graph when all edges with weights less than that threshold value are removed. A visualization of a this process can be seen in Figure 13, where ℓ is the threshold value used for filtration.

Note that, in computing the EC of the filtered graph, de Souza and co-workers use methods different from that discussed in this present work. Where we have only discussed computing the EC for a graph (a 2D object), de Souza and co-workers [13] treat the filtered graph as a CW-complex. CW-complexes are different than graphs, and involve the union of all d-cells in a graph (equivalently, the union of all d-node or -vertex cliques; see [13, 78]). A d-cell or d-vertex clique is a subset of nodes in a graph such that every node is connected by an edge to every other node in the subset (i.e., every two nodes in the subset are adjacent). A detailed discussion of CW-complexes is outside the scope of this paper; however, we mention this to highlight that the EC computed by de Souza and co-workers
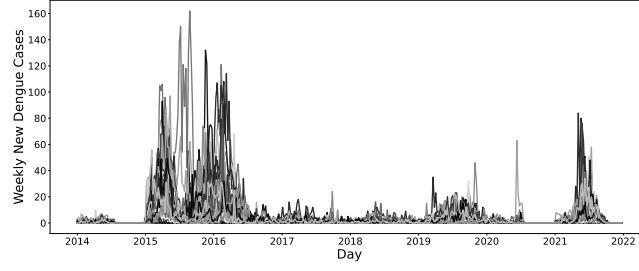
Figure 12: New dengue cases by week for 93 districts in Recife, Brazil based on data from de Souza and co-workers [13]
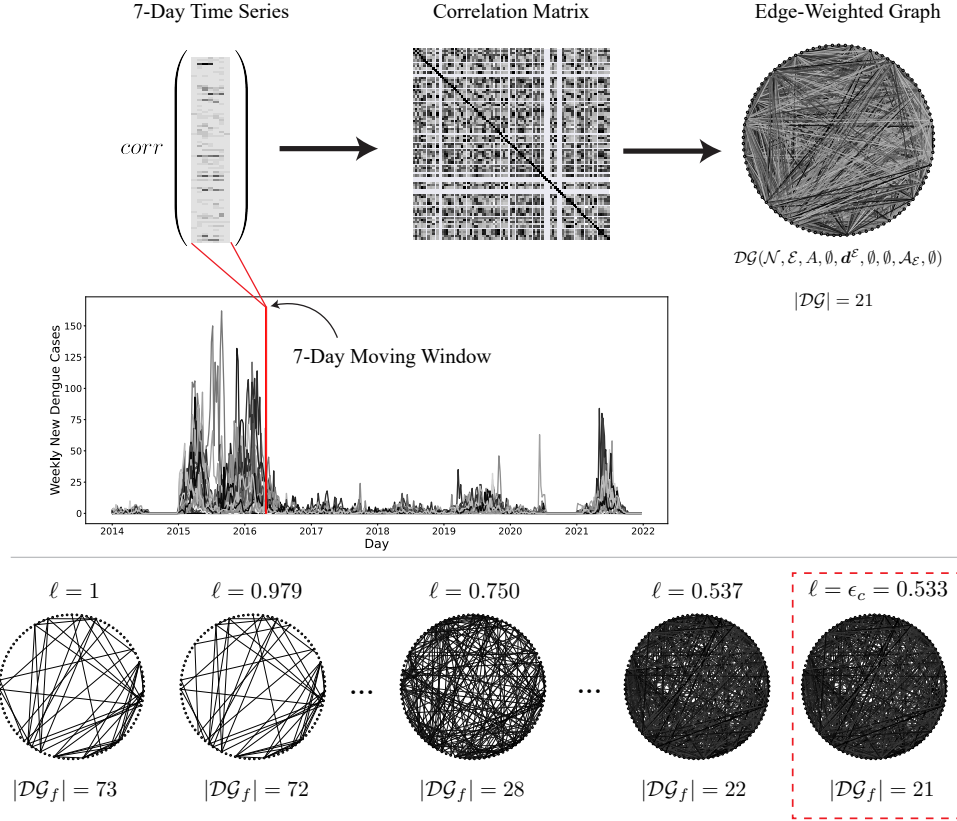

Figure 13: General procedure used by de Souza and co-workers [13]. For a 7-day moving time window, they formed a (Pearson) correlation matrix and then constructed an edge weighted graph. They then filtered out all edges and progressively added edges back into the graph by changing the filtration threshold, ℓ, until the number of connected components was equal to that of the original graph

34

for CW-complexes is different than the EC we will show computed for a graph. Instead, their methods for computing the EC involve finding all d-cells up to some d_{max} value. We will discuss later in this subsection some of the implications of this difference.

We can represent the Pearson correlation matrix at each time point within the `DataGraph` abstraction, and we then can perform a similar analysis as de Souza and co-workers [13] on the `DataGraph` within `PlasmoData.jl`. For the time series of data $\boldsymbol{X} \in \mathbb{Z}_+^{2922 \times 93}$ and $\boldsymbol{X} = [X_1, X_2, ..., X_{2922}]$, we first construct the Pearson correlation matrix $R_t = corr([X_{t-6}, X_{t-5}, ..., X_t])$ for each time t between 7 and 2922. Since R_t is a symmetric matrix, we can construct a `DataGraph` following the method outlined in Section 2.3.3. The `DataGraph` for R_t is given in (17), where $r_{i,j}$ is the i, j entry of R_t.

$$
\begin{aligned}
&\mathcal{DG}_t(\mathcal{N}, \mathcal{E}, A, \emptyset, \boldsymbol{d}^{\mathcal{E}}, \emptyset, \emptyset, \mathcal{A}_{\mathcal{E}}, \emptyset) \\
&\text{where } \{n_i : i \in \mathbb{N}_{93}\} \\
&\quad \mathcal{E} = \{(n_i, n_j) : i \in \mathbb{N}_{92}, j \in \mathbb{N}_{93} \setminus \mathbb{N}_i\} \\
&\quad \mathcal{A}_{\mathcal{E}} = \{a\} \\
&\quad d_a^{\mathcal{E}}((n_i, n_j)) = r_{i,j}, i \in \mathbb{N}_{92}, j \in \mathbb{N}_{93} \setminus \mathbb{N}_i
\end{aligned}
\tag{17}
$$

where A is a symmetric matrix ($\mathcal{DG}$ is undirected) matching the connectivity defined by $\mathcal{E}$.

With the `DataGraph` defined, we can perform the analysis shown in Figure 13. The graph edges are filtered using the logic set

$$
\mathcal{L}_{t,\ell} = \left\{a : [\ell, \max\left(\{r : r \in R_t\}\right)]\right.
$$

for attribute a. We define the characteristic threshold value for time t to be

$$
\epsilon_t = \sup\{\epsilon : |f_{\mathcal{N}}(\mathcal{DG}_t, \mathcal{L}_{t,\epsilon})| = |\mathcal{DG}_t|\}.
$$

The value of ϵ_t can be found by setting ℓ in $\mathcal{L}_{t,\ell}$ to be the largest edge weights and then progressively choosing the next largest edge weights until $|f_{\mathcal{N}}(\mathcal{DG}_t, \mathcal{L}_{t,\ell})| = |\mathcal{DG}_t|$. This process is repeated for each time value. For each graph filtered by the characteristic threshold value, we can consider various topological metrics like the EC or the number of communities.

We performed the above analysis in `PlasmoData.jl`, and we will show how the `DataGraph` abstraction can enable topological analysis of the dengue data. `PlasmoData.jl` readily facilitates the process of building the graph, filtering the edges, and performing TDA. Code Snippet 9 shows how these tasks can be implemented. First, the data for Recife, Brazil from de Souza and co-workers [13] is loaded on line 4. This data are daily reported new cases of dengue outbreaks in each of 93 city districts. We then define the function `find_smallest_filtered_graph` on Lines 7 - 25 that will add edges to the graph until the graph has only one connected componenet. The symmetric matrix for building the graph is formed by computing the Pearson correlation matrix on Line 34. Because we use a limited time window (7 days), some values in the matrix were not defined (i.e., `NaN` values). We removed all rows/columns of the symmetric matrix that were `NaN` values (Lines 37 - 38), and used

the resulting symmetric matrix to construct the edge weighted graph, $\mathcal{DG}$ (line 46) (this results in a symmetric matrix with fewer than 93 nodes). We next define a sequence of values $\ell_1, \ell_2, ..., \ell_k$ which are equal to the edge weights of the graph ($\boldsymbol{d}_a^{\mathcal{E}}$) in descending order (see Line 10; non-unique values are removed in the code). Once $\mathcal{DG}_t$ is defined, we then iterate through the sequnce of edges weights until $|f_{\mathcal{N}}(\mathcal{DG}_t, \mathcal{L}_{t,\ell_i})| = 1$, where ℓ_i is the ith value in the sequence, and where $\ell_i = \epsilon_t$. Note that we use $|f_{\mathcal{N}}(\mathcal{DG}_t, \mathcal{L}_{t,\ell_i})| = 1$ rather than $|f_{\mathcal{N}}(\mathcal{DG}_t, \mathcal{L}_{t,\ell_i})| = |\mathcal{DG}_t|$ because we have removed the rows and columns of the matrix which are not defined; this results in the edge-weighted `PlasmoData.DataGraph` formed in line 15 having only one connected component.

Code Snippet 9: Code for converting time series data to an edge-weighted graph. The code follows the general procedure of de Souza and co-workers [13] in filtering the graph

```
1   using PlasmoData, Graphs, DelimitedFiles
2
3   # Read in data
4   data = readdlm("Recife_data.csv", ',', Int)
5
6   # Define function for performing filtration
7   function find_smallest_filtered_graph(graph)
8
9       # Define values for iteration; this ensures that we only add one edge at a time
10      iter_values = sort(get_edge_data(graph)[:], rev = true)
11
12      for i in iter_values
13          # Filter out all edges below a given threshold
14          # Only keeps edges where Base.isgreater(edge_weight, i) is true
15          filtered_graph = filter_edges(graph, i; fn = Base.isgreater)
16
17          # If the number of connected components is 1, return the TDA metrics
18          if length(connected_components(filtered_graph)) == 1
19              EC = get_EC(filtered_graph)
20              num_max_cliques = length(maximal_cliques(filtered_graph))
21              num_communities = length(clique_percolation(filtered_graph, k = 25))
22              return EC, num_max_cliques, num_communities
23          end
24      end
25  end
26
27  # Create array for storing solutions
28  ECs = zeros(size(data, 1) - 6)
29  num_max_cliques = zeros(size(data, 1) - 6)
30  num_communities = zeros(size(data, 1) - 6)
31
32  for i in 1:length(sols)
33      # Form a correlation matrix based on 7 days of data
34      cor_mat = cor(data[i:(i + 6), :], dims = 1)
35
36      # Remove the matrix entries that are NaNs
37      bit_vec = (!).(isnan.(cor_mat[:, 1]))
38      sym_mat = cor_mat[bit_vec, bit_vec]
39
40      # If there are not more than 2 nodes in the graph, skip this iteration
41      node_count = size(sym_mat, 1)
42      if node_count <= 2
43          continue
44      end
45
46      sym_graph = symmetric_matrix_to_graph(sym_mat)
47
48      ECs[i], num_max_cliques[i], num_communities[i] = find_smallest_filtered_graph(sym_graph)
49  end
```

With the filtered `DataGraph` defined, we can consider several topological metrics for analyzing our data. The goal of de Souza and co-workers [13] was to identify ways of "fingerprinting" when disease

outbreaks were occurring. They chose the EC as their primary indicator, and they showed how the EC was able to help indicate when outbreaks were occurring, especially for larger d_{max} values. This may be because the higher d_{max} values indicate when there are several nodes (districts) are closely correlated. Consequently, we consider the EC of the graphs being formed, and then we also consider the size of cliques in the graph and the number of communities in the graph. In graph theory, a k-clique is a set of k nodes which are connected to all other nodes in the clique. We hypothesize that a possible fingerprint for outbreaks may be when several nodes (districts) are closely correlated. Thus, we compute the EC for comparison with de Souza and co-workers [13], we compute the total number of maximal cliques in the graph (i.e, the total number of k_{max}-cliques in the graph, where k_{max} is the largest integer value for which there is a nonempty set of nodes forming a k_{max}-clique in the graph) and we compute the number of communities in the graph (found by clique percolation [46], with $k =$ 25; see Lines 18 - 23). The results of these computations, as well as the average daily new cases during each 7 day moving window, are shown in Figure 14.
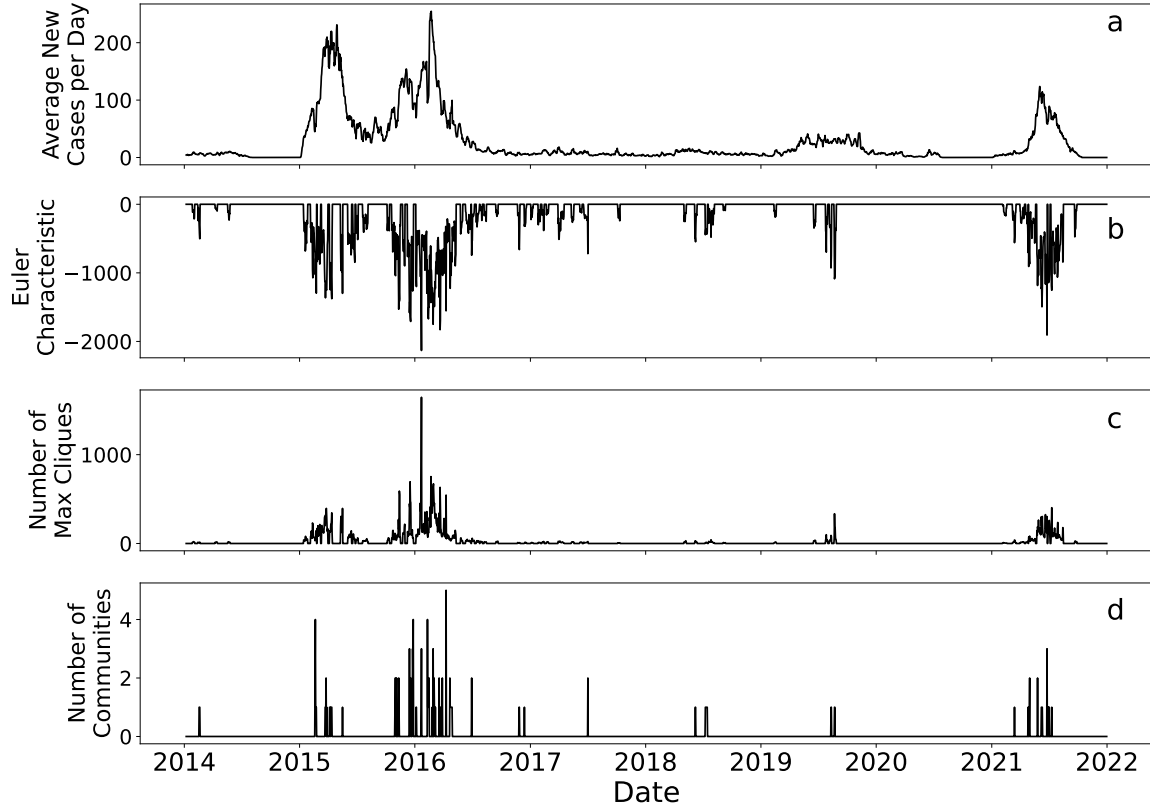

Figure 14: Results for TDA of dengue data for 93 districts in Recife, Brazil [13]. For each 7-day moving time window, an edge-weighted `DataGraph` was formed. Results presented include the average number of new cases per day for the moving time window (a), the EC for the `DataGraph` (b), the number of maximal cliques in the `DataGraph` (c), and the number of communities in the `DataGraph` using clique percolation ($k = 25$) [46]

The results shown in Figure 14 are consistent with the results of de Souza and co-workers [13].

As mentioned above, the EC we compute in the above code is for a graph, while de Souza and co-workers [13] computed the EC for CW-complexes, where their computed EC was a function of d_{max}. They used various d_{max} values up to 7, and found that the noise of the EC decreased with increasing d_{max}. In this case, the EC in Figure 14 does contain significant noise (visually similar to that seen for $d_{max} = 2$ and $d_{max} = 3$ of de Souza and co-workers' [13] work). However, the trends are similar, and we note that it is computationally much easier to compute the EC for a graph (i.e., number of nodes minus number of edges) than to compute the EC for a CW-complex with larger d_{max} (where each d_{max}-cell may have to be determined). In addition, the number of maximal cliques (Figure 14c) and the number of communities (Figure 14d) also correlate to the average number of new cases per day, and they appear slightly less noisy than the EC. A detailed analysis of these additional indicators as fingerprints for disease outbreaks is outside the scope of this work; instead, our purpose in showing these results is to highlight how the `DataGraph` abstraction and how `PlasmoData.jl` facilitates different TDA approaches.

4.3 Connectivity Analysis

In this subsection, we present a technology pathway analysis example to illustrate how the `DataGraph` abstraction and `PlasmoData.jl` can be used to navigate connectivity and provide insights into the technology pathway. The technology pathway we consider has five raw materials (petroleum naphtha, natural gas, corn stover, sugar beets, and sugarcane) that can be used in technologies to produce seven different polymers (low density polyethylene (LDPE), high density polyethylene (HDPE), polypropylene (PP), polyvinylchloride (PVC), polystyrene (PS), polyethylene terephthalate (PET), and Nylon66). There are 15 different intermediate chemicals produced and 27 different technologies that can be used to produce different chemicals. Each of these raw materials, products, intermediates, and technologies are represented by nodes. The interconnectivity of materials and technologies is complex and difficult to navigate [79]. This type of connectivity also arises in complex, multi-product supply chains [80].

Within the above technology pathway, different types of nodes have specific data attached to them. For example, nodes can contain supply limitations, demands, costs, and carbon emissions, and these will be saved as node data. This problem also contains connections between raw materials and technologies, technologies and intermediates, intermediates and technologies, and technologies and products which can all be represented by directed edges, forming a directed graph, with 57 total edges. In addition, we also have flows that are represented as edge data. These flows are dependent on the connections in the graph and on the data stored in the nodes (e.g., it depends on the cost of technologies or the maximum available supply of the products).

The `DataGraph` for this technology pathway is a heterogenous graph and contains multiple attributes for node and edge data. The `DataGraph` contains the set of 54 nodes and 57 directed edges discussed above, which we notate here as $\mathcal{N}$ and $\mathcal{E}$. There are four subsets of nodes, $\mathcal{N}_{raw}, \mathcal{N}_{prod}, \mathcal{N}_{int}$, and $\mathcal{N}_{tech}$ for the sets of raw materials, products, intermediates, and technologies, respectively, such

that $\mathcal{N} = \mathcal{N}_{raw} \cup \mathcal{N}_{prod} \cup \mathcal{N}_{int} \cup \mathcal{N}_{tech}$. Each node stores data under different attributes indicating whether the node is a raw material (a_{raw}), product (a_{prod}), intermediate (a_{int}), or technology (a_{tech}) and data under different attributes for the raw material cost (a_{cost}), the CO_2 emissions per weight of material (a_{CO_2}), maximum raw material supply (a_{sup}), and maximum production demand (a_{dem}). Each edge has data for the optimal flow of material along the edge under attribute a_{flow}.

The `DataGraph`, $\mathcal{DG}$, can be represented mathematically as:

$$
\begin{aligned}
&\mathcal{DG}(\mathcal{N}, \mathcal{E}, A, \boldsymbol{d}^{\mathcal{N}}, \boldsymbol{d}^{\mathcal{E}}, \emptyset, \mathcal{A}_{\mathcal{N}}, \mathcal{A}_{\mathcal{E}}, \emptyset) \\
&\text{where } \mathcal{N} = \mathcal{N}_{raw} \cup \mathcal{N}_{prod} \cup \mathcal{N}_{int} \cup \mathcal{N}_{tech} \\
&\qquad \mathcal{A}_{\mathcal{N}} = \{a_{raw}, a_{prod}, a_{tech}, a_{cost}, a_{CO_2}, a_{sup}, a_{dem}\} \\
&\qquad \mathcal{A}_{\mathcal{E}} = \{a_{flow}\} \\
&\qquad d^{\mathcal{N}}_{a_{raw}}(n) = 1, \quad n \in \mathcal{N}_{raw}, \qquad d^{\mathcal{N}}_{a_{raw}} = 0, \quad n \in \mathcal{N} \setminus \mathcal{N}_{raw} \\
&\qquad d^{\mathcal{N}}_{a_{prod}}(n) = 1, \quad n \in \mathcal{N}_{prod}, \qquad d^{\mathcal{N}}_{a_{prod}} = 0, \quad n \in \mathcal{N} \setminus \mathcal{N}_{prod} \\
&\qquad d^{\mathcal{N}}_{a_{int}}(n) = 1, \quad n \in \mathcal{N}_{int}, \qquad d^{\mathcal{N}}_{a_{int}} = 0, \quad n \in \mathcal{N} \setminus \mathcal{N}_{int} \\
&\qquad d^{\mathcal{N}}_{a_{tech}}(n) = 1, \quad n \in \mathcal{N}_{tech}, \qquad d^{\mathcal{N}}_{a_{tech}} = 0, \quad n \in \mathcal{N} \setminus \mathcal{N}_{tech} \\
&\qquad d^{\mathcal{N}}_{a_{cost}}(n) = \alpha_{cost}(n), \quad n \in \mathcal{N}_{raw} \cup \mathcal{N}_{tech}, \qquad d^{\mathcal{N}}_{a_{cost}} = 0, \quad n \in \mathcal{N}_{prod} \cup \mathcal{N}_{int} \\
&\qquad d^{\mathcal{N}}_{a_{CO_2}}(n) = \alpha_{CO_2}(n), \quad n \in \mathcal{N}_{raw} \cup \mathcal{N}_{tech}, \qquad d^{\mathcal{N}}_{a_{CO_2}}(n) = 0, \quad n \in \mathcal{N}_{prod} \cup \mathcal{N}_{int} \\
&\qquad d^{\mathcal{N}}_{a_{sup}}(n) = \alpha_{sup}(n), \quad n \in \mathcal{N}_{raw}, \qquad d^{\mathcal{N}}_{a_{sup}} = 0, \quad n \in \mathcal{N} \setminus \mathcal{N}_{raw} \\
&\qquad d^{\mathcal{N}}_{a_{dem}}(n) = \alpha_{dem}(n), \quad n \in \mathcal{N}_{prod}, \qquad d^{\mathcal{N}}_{a_{dem}} = 0, \quad n \in \mathcal{N} \setminus \mathcal{N}_{prod} \\
&\qquad d^{\mathcal{E}}_{a_{flow}}(e) = \alpha_{flow}(e), \quad e \in \mathcal{E}
\end{aligned}
\tag{18}
$$

where A is an asymmetric matrix ($\mathcal{DG}$ is directed) matching the connectivity defined by $\mathcal{E}$. Here, $\alpha_{cost}(n)$ is the raw material cost for node n, $\alpha_{CO_2}(n)$ is the CO_2 emissions per weight of material for node n, $\alpha_{sup}(n)$ is the maximum raw material supply for node n, $\alpha_{dem}(n)$ is the maximum production demand for node n, and $\alpha_{flow}(e)$ is the optimal flow on edge e (this value is computed separately by solving an optimization problem that uses the above data).

The code for creating $\mathcal{DG}$ in `PlasmoData.jl` is given in Code Snippet 10. In this code, each type of node (Raw Material, Product, Intermediate, and Technology) is loaded as a separate CSV file, where the CSV contains the name of the node and data corresponding to that node. Each node is added using the `add_node!` function, and data is added using the `add_node_data!` function. In addition, we add an attribute to each node (Lines 16, 24, 30, and 35) where a value of 1 indicates the type of node. The edges and edge data are then added to the graph using the `add_edge!` and `add_edge_data!` functions (Lines 41 - 45).

Code Snippet 10: Code for constructing a directed graph for a technology pathway problem

```
1   using PlasmoData, Graphs, DelimitedFiles, PlasmoDataPlots
2
3   # Read in data
4   raw_data = readdlm("rawmaterial_data_array.csv", ',')
5   prod_data = readdlm("product_data_array.csv", ',')
6   int_data = readdlm("intermediates_data_array.csv", ',')
7   tech_data = readdlm("technology_data_array.csv", ',')
8   edge_data = readdlm("edge_data.csv", ',')
9
10  # Define DataDiGraph
11  dg = DataDiGraph()
12
13  # Add nodes and node data
14  for i in 1:size(raw_data, 1)
15      add_node!(dg, raw_data[i, 1])
16      add_node_data!(dg, raw_data[i, 1], 1, "Raw Material")
17      add_node_data!(dg, raw_data[i, 1], raw_data[i, 2], "Cost")
18      add_node_data!(dg, raw_data[i, 1], raw_data[i, 3], "CO2 Cost")
19      add_node_data!(dg, raw_data[i, 1], raw_data[i, 4], "Max Supply")
20  end
21
22  for i in 1:size(prod_data, 1)
23      add_node!(dg, prod_data[i, 1])
24      add_node_data!(dg, prod_data[i, 1], 1, "Product")
25      add_node_data!(dg, prod_data[i, 1], prod_data[i, 2], "Demand Limit")
26  end
27
28  for i in 1:size(int_data, 1)
29      add_node!(dg, int_data[i, 1])
30      add_node_data!(dg, int_data[i, 1], 1, "Intermediate")
31  end
32
33  for i in 1:size(tech_data, 1)
34      add_node!(dg, tech_data[i, 1])
35      add_node_data!(dg, tech_data[i, 1], 1, "Technology")
36      add_node_data!(dg, tech_data[i, 1], tech_data[i, 2], "Cost")
37      add_node_data!(dg, tech_data[i, 1], tech_data[i, 3], "CO2 Cost")
38  end
39
40  # Add edges and edge data
41  for i in 1:size(edge_data, 1)
42      edge = (edge_data[i, 1], edge_data[i, 2])
43      PlasmoData.add_edge!(dg, edge)
44      PlasmoData.add_edge_data!(dg, edge, edge_data[i, 3], "Optimal Flow")
45  end
46
47  plot_graph(dg, dag_positions = true, nlabels = dg.nodes)
```

4.3.1 Visualization

One of the potential benefits of these graph representations is the ability to visualize the structure of the problem. In this example, $\mathcal{DG}$ is a directed acyclic graph (DAG), which is a common structure in pathway and supply chain problems. Because it is a DAG, the graph can be presented in different layers, and different algorithms have been proposed for identifying these layers and the positions of nodes within the visualization [81, 82]. The above technology problem is shown in Figure 15 using the algorithm of Zarate and co-workers [82] to determine node positions (as implemented within the `Julia` package `LayeredLayouts.jl` [83]). This visualization is automated through `PlasmoDataPlots.jl`. In this case, Figure 15 (without node coloring) can be easily constructed with just Line 47 of Code Snippet 10 (abbreviation meanings are available in the supporting information).
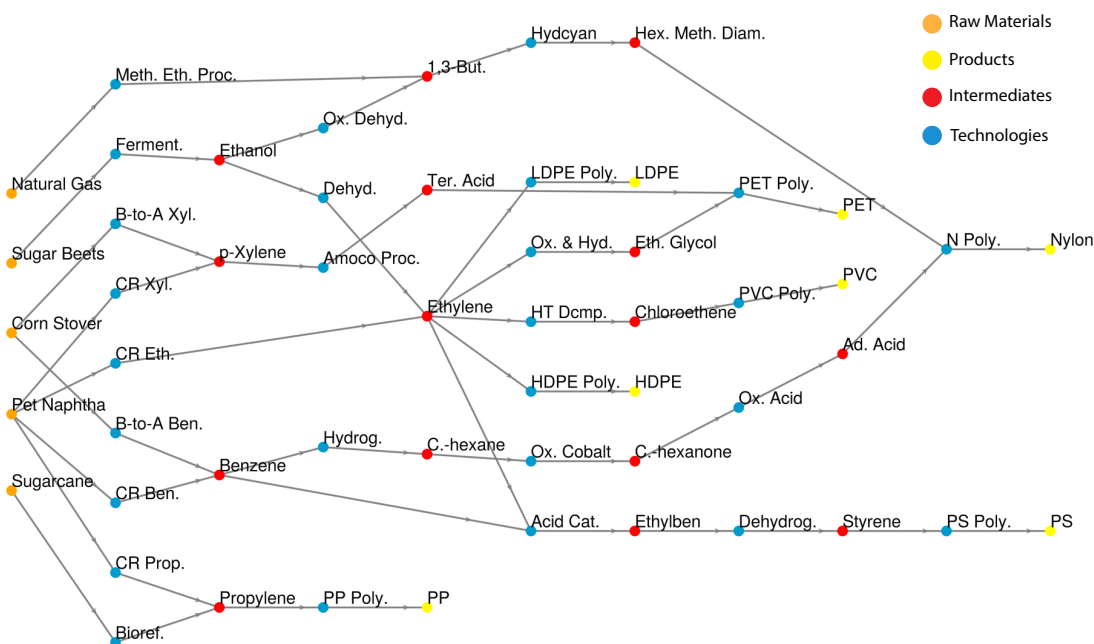

Figure 15: Directed graph of a layered technology pathway example; created using `PlasmoDataPlots.jl`

The ability to visualize these structures can be a significant resource in evaluating the data. For example, it can make clear what some of the paths are or what some of the most essential nodes are. For instance, from Figure 15, it is clear that the node "Ethylene" has more connections than any other node and is directly connected to several products. In addition, `PlasmoDataPlots.jl` provides the ability to highlight paths within a graph. For example, by calling the function `plot_graph_path` with the source ("Corn Stover") and destination ("Nylon") nodes of the path, Figure 16 can be generated. Visualizations for DAGs or for layered graphs are often not implemented directly in graph plotting tools, and other tools for visualization (e.g., yFiles [84]) can require subscriptions, so this ability for visualization is a powerful capability of `PlasmoData.jl`.
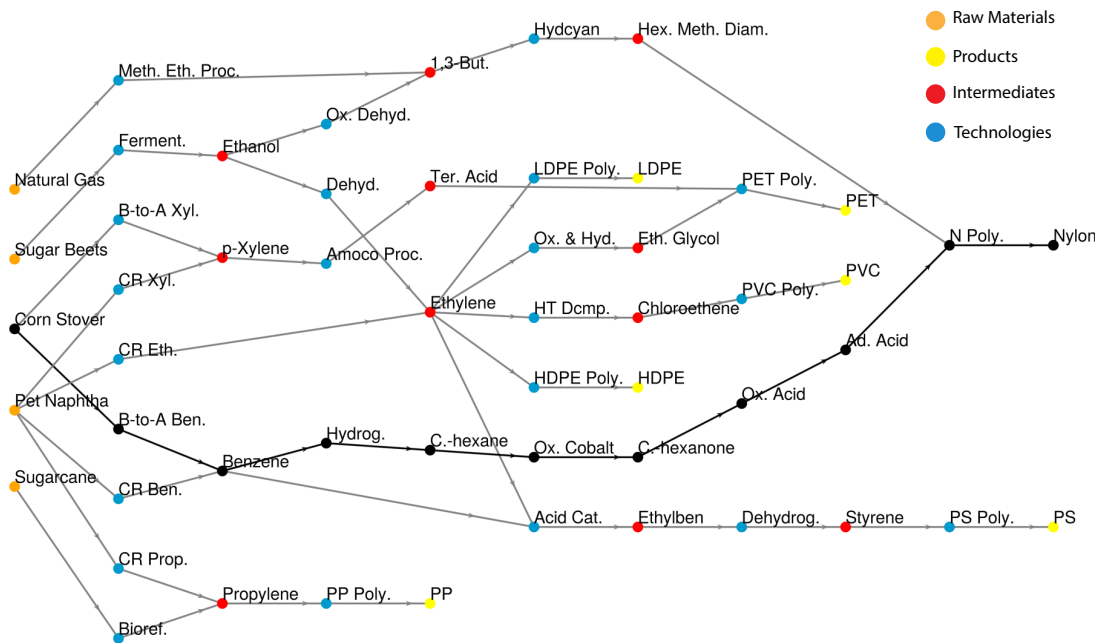
41

Figure 16: Directed graph with a pathway between "Corn Stover" and "Nylon" highlighted, created using `PlasmoDataPlots.jl`

In addition to plotting the layered structure, incorporating the data of the graph could also be useful. One way of doing this would be to create a Sankey diagram. Often, Sankey diagrams are not thought of as graphs, but they are effectively just directed graphs with edge weights. Thus, to go from the directed graph defined in Code Snippet 10 to a Sankey diagram, we essentially need only five lines of code (see Code Snippet 11). The resulting diagram can be seen in Figure 17. Thus, constructing data within a graph structure can provide unique and useful visualization capabilities.

Code Snippet 11: Code for constructing a Sankey diagram from a `PlasmoData.DataDiGraph`

```
using SankeyPlots

# Define src and dst for edges
src = [i for (i, j) in dg.edges]
dst = [j for (i, j) in dg.edges]
weights = get_edge_data(dg, "Optimal Flow")

# Plot sankey diagram
sankey(src, dst, weights;
    size = (1400, 800),
    node_labels = dg.nodes,
    edge_color = :black,
    label_size = 9
)
```
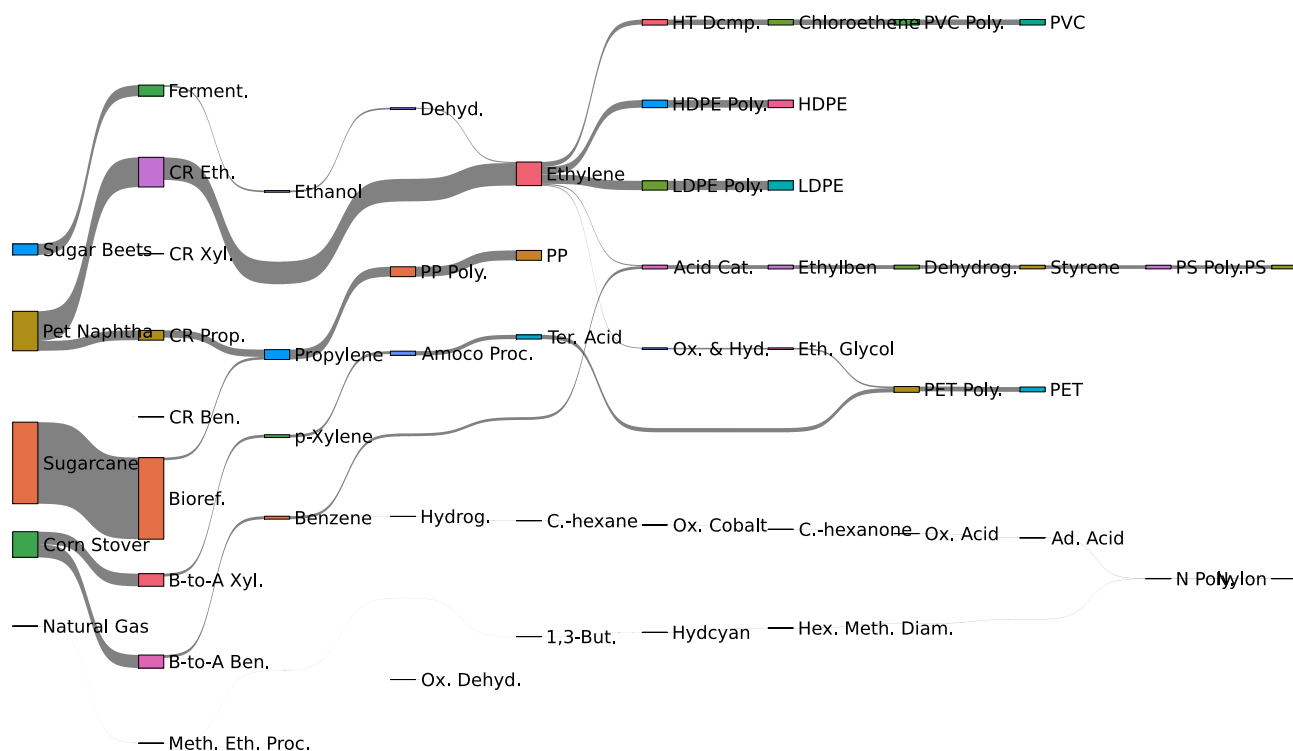
Figure 17: Sankey diagram of a technology pathway example

4.3.2 Connectivity Measures

An important challenge in working with graph models of data is how to navigate connections to get information out of the data. For example, in the above technology pathway, how nodes are connected can reveal information about each node. Different metrics can give insight into how "important" or connected a node may be. Thus, we can consider metrics such as the number of upstream or downstream nodes or the number of products to which a node is connected. `PlasmoData.jl` facilitates these kinds of metrics by providing functions for identifying paths between nodes and identifying sets of upstream or downstream nodes (in part through accessing functions in `Graphs.jl`). In addition, one might also be interested in how manipulating the graph structure can impact these paths (e.g., If a node is removed, what happens to the structure?). We will show here how some of these analyses can be performed.

We consider how connectivity metrics can assist in understanding the data. Algorithms have been proposed in literature for identifying paths and connections between nodes in a graph [85, 86, 87, 88]. For example, we can consider to how many products each raw material can contribute, or we can consider how many raw materials contribute to each product. Analyses like these can give insight into the raw materials or products; for example this could help identify which raw materials or products are most susceptible to disruption (i.e., if a product only has one raw material connected to it, it may be more easily disrupted if anything happens to the supply of that material). For each raw material and

product, we can also count the total number of upstream and downstream nodes to provide an idea of how "connected" each raw material and product is to the rest of the graph. All of the above metrics are easily performed in `PlasmoData` as shown in Code Snippet 12, using functions like `has_path` (Line 20), `downstream_nodes` (Line 4), and `upstream_nodes` (Line 17). The above metrics can also be added to the graph as new data by defining a new attribute for each metric (see Lines 5, 18, and 24).

Code Snippet 12: Code for determining the connectivity metrics of the technology graph given in Code Snippet 10

```
1   # Compute metrics for raw materials
2   for r in raw_data[:, 1]
3       count = 0
4       num_downstream = length(downstream_nodes(dg, r))
5       add_node_data!(dg, r, num_downstream, "Number Downstream")
6       for p in prod_data[:, 1]
7           if PlasmoData.has_path(dg, r, p)
8               count += 1
9           end
10      end
11      add_node_data!(dg, r, count, "Connected Products")
12  end
13
14  # Compute metrics for products
15  for p in prod_data[:, 1]
16      count = 0
17      num_upstream = length(upstream_nodes(dg, p))
18      add_node_data!(dg, p, num_upstream, "Number Upstream")
19      for r in raw_data[:, 1]
20          if PlasmoData.has_path(dg, r, p)
21              count += 1
22          end
23      end
24      add_node_data!(dg, p, count, "Connected Raw")
25  end
```

In addition to computing measures on the original graph, we can also consider how these metrics change as the graph structure is altered such as through node- or edge-removal, node- or edge-filtration, or aggregation. For example, we can look at what happens if we remove different intermediates or technologies from the graph. Doing so can be thought of as considering the impact if an intermediate material or technology were no longer available or taken off-line unexpectedly. Using `PlasmoData.jl`, we remove the intermediates Ethylene, Terephthalic Acid, and Cyclohexane separately from the graph and see what impact each has on the metrics (i.e., we call `remove_node!` and reevaluate Code Snippet 12). The original metrics for each graph and the resulting changes can be seen in Tables 1 and 2. Of those three intermediates, the only intermediate that is completely essential is Ethylene (when ethylene is removed, LDPE, HDPE, and PVC have no connection to any raw materials). It is also clear from the metrics that removing Ethylene from the graph could impact the production of PS and PET since these are no longer connected to as many raw materials. This type of analysis can be valuable in analyzing the resilience of a system in the face of failure of certain elements.

The above tools can also be coupled with the visualization tools to provide an easier interface to understand these changes. For example, Figure 18 shows the technology graph where nodes are sized by their number of downstream connections (i.e., sizing by node data). This shows visually which raw materials are connected to the most downstream nodes, but it also gives insight into which

44

Table 1: Metrics on a technology graph with the number of connected products (Prod) to each raw material and the number of downstream nodes (Down) for each raw material. The nodes Ethylene, Ter Acid, and C.-hexane were each individually removed from the original graph and metrics recomputed

Raw Material	Original Graph		w/o Ethylene		w/o Ter. Acid		w/o C.-hexane	
	Prod	Down	Prod	Down	Prod	Down	Prod	Down
Sugar Beets	6	29	1	10	6	29	6	29
Sugarcane	1	5	1	5	1	5	1	5
Corn Stover	3	23	3	23	2	20	2	16
Natural Gas	1	7	1	7	1	7	1	7
Pet Naphtha	7	39	4	28	7	38	6	32

Table 2: Metrics on the technology graph with the number of connected raw materials (Raw) to each product and the number of upstream nodes (Up) for each product. The nodes Ethylene, Ter Acid, and C.-hexane were each individually removed from the original graph and metrics recomputed

Product	Original Graph		w/o Ethylene		w/o Ter. Acid		w/o C.-hexane	
	Raw	Up	Raw	Up	Raw	Up	Raw	Up
LDPE	2	9	0	2	2	9	2	9
HDPE	2	9	0	2	2	9	2	9
PP	2	7	2	7	2	7	2	7
PVC	2	11	0	4	2	11	2	11
PS	3	17	2	11	3	17	3	17
PET	4	17	2	11	2	11	3	17
Nylon	4	22	4	22	4	22	2	15

intermediate and technology nodes contain the most downstream connections.

The `DataGraph` abstraction and `PlasmoData.jl` provide a framework for examining the connections within data structures to gain insight into the data or system. In the above example, we have highlighted just a couple ways that this could be done, but several other methods (enabled within `PlasmoData.jl`) could likewise be used. For example, the structure of the graph could be filtered based on node weights (e.g., cost or carbon emissions) or edge weights (e.g., flows) which would alter the structure and connections of the graph. Further, other structure manipulations (such as adding edges) could be important in studying these graphs. For example, you could consider where edges could be placed to increase the number of connections between raw materials and products (hypothetically increasing the "resilience" of the structure). These are just a couple examples of other ways `PlasmoData.jl` could be used to elucidate more information about a graph.
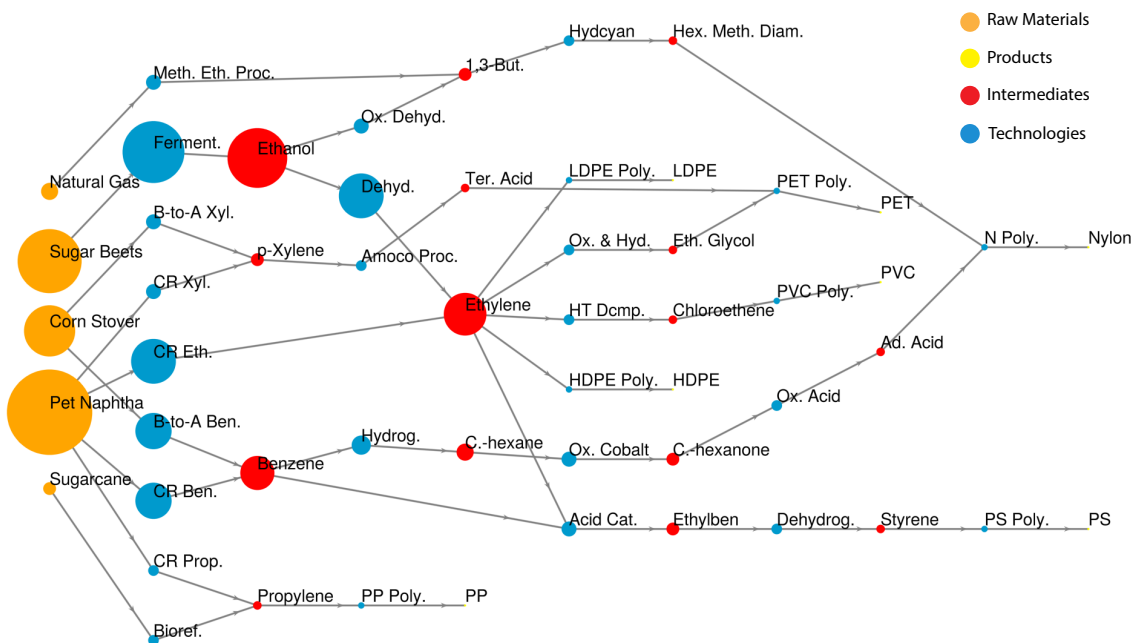
45

Figure 18: Directed graph of a layered technology pathway example with nodes sized by the number of downstream nodes, created using `PlasmoDataPlots.jl`

5   Conclusions and Future Work

We presented a graph-theoretic abstraction for modeling data; we call this abstraction `DataGraph` and provide a software implementation in the `Julia` package that we call `PlasmoData.jl`. We show that common data structures (e.g., matrices, tensors, images) can be represented under the `DataGraph` abstraction, and there are a variety of mathematical tools available for analyzing the data modeled as a graph (e.g., filtration, aggregation, topological analysis). We presented case studies to show this abstraction can be used for a variety of tasks including determining features for machine learning, analyzing time series, and performing path analysis. In doing this, we have highlighted how representing data can be a modeling' task and having the correct abstraction for the data can strongly influence the value of the data.

There are a couple of areas that we would like to focus on in future work. First, there are more applications we would like to explore. Using graphs to represent molecules has garnered significant interest and has been applied in many studies and we would like to streamline molecular representations within `PlasmoData.jl`, such as enabling construction of the graph from a SMILES (Simplified Molecular Input Line Entry System) string, and explore how `PlasmoData.jl` can enhance this analysis. We are also interested in other applications of the pathway analysis performed above. Additionally, we would like to expand the capabilities of `PlasmoData.jl`. There are other TDA metrics and path functions that could be implemented, and there are topological descriptors that could be approximated with graphs. For example, in the work by Jiang and co-workers [69], they use a marching-square

46

algorithm to approximate the Minkowski functionals of a filtered image. A marching-square algorithm could be implemented for an image represented by a graph by iterating through the nodes and analyzing the connections to the adjacent nodes. This could lead to more in-depth image analysis. Also, we would like to expand the integration of `PlasmoData.jl` with GNN packages; this could include improving the interface with GNN packages and enabling different graph manipulation functions with GNN algorithms, such as using aggregation for pooling operations or message-aggregator operators.

In addition, while graphs can be powerful tools for modeling data, they do have some limitations. Nodes and edges do not have any notion of placement in space. Oftentimes, the systems being represented by graphs have a fixed location (e.g., supply chains may have a fixed production location) which can significantly influence the system. Furthermore, graphs only capture connectivity between nodes; in many instances, it may be more accurate to include a link between more than two nodes. In these cases, hypergraphs (or other representations) may be more appropriate. Finally, graphs are restricted to edges between no more than two nodes; in future work, we would like to explore generalizations of hypergraphs and simplicial complexes (which may capture more complex and higher-order relations) and how to capture more complex attributes for these objects, such as spatial locations.

Acknowledgments

References

[1] Wesley Willett, Yvonne Jansen, and Pierre Dragicevic. Embedded data representations. IEEE transactions on visualization and computer graphics, 23(1):461–470, 2016.

[2] Philip K Robertson. A methodology for choosing data representations. IEEE Computer Graphics and Applications, 11(03):56–67, 1991.

[3] Leo Torres, Ann S Blevins, Danielle Bassett, and Tina Eliassi-Rad. The why, how, and when of representations for complex systems. SIAM Review, 63(3):435–485, 2021.

[4] Jordan Jalving, Sungho Shin, and Victor M Zavala. A graph-based modeling abstraction for optimization: Concepts and implementation in plasmo. jl. Mathematical Programming Computation, 14(4):699–747, 2022.

[5] Mustafa Hajij, Ghada Zamzmi, Theodore Papamarkou, Nina Miolane, Aldo Guzmán-Sáenz, Karthikeyan Natesan Ramamurthy, Tolga Birdal, Tamal K Dey, Soham Mukherjee, Shreyas N Samaga, et al. Topological deep learning: Going beyond graph data. arXiv preprint arXiv:2206.00606, 2023.

[6] Yue Shao and Victor M Zavala. Modularity measures: Concepts, computation, and applications to manufacturing systems. AIChE Journal, 66(6):e16965, 2020.

[7] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. Using graph theory to analyze biological networks. BioData mining, 4:1–27, 2011.

[8] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. Nature reviews neuroscience, 10(3):186–198, 2009.

[9] Mikail Rubinov and Olaf Sporns. Complex network measures of brain connectivity: uses and interpretations. Neuroimage, 52(3):1059–1069, 2010.

[10] David L. Cole, Gerardo J. Ruiz-Mercado, and Victor M. Zavala. A graph-based modeling framework for tracing hydrological pollutant transport in surface waters. arXiv preprint: https://arxiv.org/abs/2302.04991, 2023.

[11] Tobias Heckmann, Wolfgang Schwanghart, and Jonathan D Phillips. Graph theory—recent developments of its application in geomorphology. Geomorphology, 243:130–146, 2015.

[12] Katelyn BS King, Qi Wang, Lauren K Rodriguez, and Kendra S Cheruvelil. Lake networks and connectivity metrics for the conterminous us (lagos-us networks v1). Limnology and Oceanography Letters, 6(5):293–307, 2021.

[13] Danillo Barros de Souza, Everlon Figueirôa dos Santos, and Fernando AN Santos. The euler characteristic as a topological marker for outbreaks in vector-borne disease. Journal of Statistical Mechanics: Theory and Experiment, 2022(12):123501, 2022.

[14] Dane Taylor, Florian Klimm, Heather A Harrington, Miroslav Kramár, Konstantin Mischaikow, Mason A Porter, and Peter J Mucha. Topological data analysis of contagion maps for examining spreading processes on networks. Nature communications, 6(1):7723, 2015.

[15] Stanley Wasserman and Katherine Faust. Social Network Analysis: Methods and Applications. Structural Analysis in the Social Sciences. Cambridge University Press, 1994.

[16] Shiyi Qin, Tianyi Jin, Reid C Van Lehn, and Victor M Zavala. Predicting critical micelle concentrations for surfactants using graph convolutional neural networks. The Journal of Physical Chemistry B, 125(37):10610–10620, 2021.

[17] Shiyi Qin, Shengli Jiang, Jianping Li, Prasanna Balaprakash, Reid C Van Lehn, and Victor M Zavala. Capturing molecular interactions in graph neural networks: a case study in multi-component phase equilibrium. Digital Discovery, 2(1):138–151, 2023.

[18] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. Chemical science, 9(2):513–530, 2018.

[19] Alexander Smith and Victor M Zavala. The euler characteristic: A general topological descriptor for complex data. Computers & Chemical Engineering, 154:107463, 2021.

[20] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. Distill, 2021. https://distill.pub/2021/gnn-intro.

[21] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[22] James Fairbanks, Mathieu Besançon, Schölly Simon, Júlio Hoffiman, Nick Eubank, and Stefan Karpinski. JuliaGraphs/Graphs.jl: an optimized graphs package for the julia programming language, 2021. https://github.com/JuliaGraphs/Graphs.jl/. accessed 17 March 2023.

[23] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. InterJournal, Complex Systems:1695, 2006.

[24] Tiago P. Peixoto. The graph-tool python library. figshare, 2014.

[25] MathWorks. Graph and network algorithms. https://www.mathworks.com/help/matlab/graph-and-network-algorithms.html?s_tid=CRUX_lftnav. Accessed on February 25, 2023.

[26] JD Kruschwitz, D List, L Waller, M Rubinov, and H Walter. Graphvar: a user-friendly toolbox for comprehensive graph analyses of functional brain connectivity. Journal of neuroscience methods, 245:107–115, 2015.

[27] Jinhui Wang, Xindi Wang, Mingrui Xia, Xuhong Liao, Alan Evans, and Yong He. Gretna: a graph theoretical network analysis toolbox for imaging connectomics. Frontiers in human neuroscience, 9:386, 2015.

[28] Mite Mijalkov, Ehsan Kakaei, Joana B Pereira, Eric Westman, Giovanni Volpe, and Alzheimer's Disease Neuroimaging Initiative. Braph: a graph theory software for the analysis of brain connectivity. PloS one, 12(8):e0178798, 2017.

[29] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315, 2019.

[30] Thomas Bonald, Nathan de Lara, Quentin Lutz, and Bertrand Charpentier. Scikit-network: Graph analysis in python. Journal of Machine Learning Research, 21(185):1–6, 2020.

[31] Carlo Lucibello and other contributors. GraphNeuralNetworks.jl: a geometric deep learning library for the julia programming language, 2021. https://github.com/CarloLucibello/GraphNeuralNetworks.jl. Accessed 14 March 2023.

[32] Tara L Andrews and Caroline Macé. Beyond the tree of texts: Building an empirical model of scribal variation through graph analysis of texts and stemmata. Literary and Linguistic Computing, 28(4):504–521, 2013.

[33] Lianzhe Huang, Dehong Ma, Sujian Li, Xiaodong Zhang, and Houfeng Wang. Text level graph neural network for text classification. arXiv preprint arXiv:1910.02356, 2019.

[34] Victor M Zavala. Outlook: How i learned to love machine learning (a personal perspective on machine learning in process systems engineering). Industrial & Engineering Chemistry Research, 2023.

[35] Vikramsingh R Parihar. ICSES Trans. Image Process. Pattern Recognit., volume 4, chapter Chapter 5: Image segmentation based on graph theory and threshold, pages 61–82. ITIPPR, 2018.

[36] TN Janakiraman and PVSSR Chandra Mouli. Image segmentation using euler graphs. International Journal of Computers Communications & Control, 5(3):314–324, 2010.

[37] OJ Morris, M de J Lee, and AG Constantinides. Graph theory for image analysis: an approach based on the shortest spanning tree. In IEE Proceedings F (Communications, Radar and Signal Processing), volume 133, pages 146–152. IET, 1986.

[38] Benjamin Perret, Giovanni Chierchia, Jean Cousty, Silvio Jamil Ferzoli Guimaraes, Yukiko Kenmochi, and Laurent Najman. Higra: Hierarchical graph analysis. SoftwareX, 10:100335, 2019.

[39] Philippe Salembier and Luis Garrido. Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. IEEE transactions on Image Processing, 9(4):561–576, 2000.

[40] Santo Fortunato. Community detection in graphs. Physics reports, 486(3-5):75–174, 2010.

[41] Muhammad Aqib Javed, Muhammad Shahzad Younis, Siddique Latif, Junaid Qadir, and Adeel Baig. Community detection in networks: A multidisciplinary review. Journal of Network and Computer Applications, 108:87–111, 2018.

[42] Andrew Allman, Wentao Tang, and Prodromos Daoutidis. DeCODe: a community-based algorithm for generating high-quality decompositions of optimization problems. Optimization and Engineering, 20:1067–1084, 2019.

[43] Ilias Mitrai and Prodromos Daoutidis. Decomposition of integrated scheduling and dynamic optimization problems using community detection. Journal of Process Control, 90:63–74, 2020.

[44] Pall F Jonsson, Tamara Cavanna, Daniel Zicha, and Paul A Bates. Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis. BMC bioinformatics, 7:1–13, 2006.

[45] Aryya Gangopadhyay and Song Chen. Health care fraud detection with community detection algorithms. In 2016 IEEE International Conference on Smart Computing (SMARTCOMP), pages 1–5. IEEE, 2016.

[46] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. nature, 435(7043):814–818, 2005.

[47] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. Physical review E, 76(3):036106, 2007.

[48] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. Physical review E, 69(2):026113, 2004.

[49] Gunnar Carlsson. Topology and data. Bulletin of the American Mathematical Society, 46(2):255–308, 2009.

[50] Frédéric Chazal and Bertrand Michel. An introduction to topological data analysis: fundamental and practical aspects for data scientists. Frontiers in artificial intelligence, 4:108, 2021.

[51] Elizabeth Munch. A user's guide to topological data analysis. Journal of Learning Analytics, 4(2):47–61, 2017.

[52] Larry Wasserman. Topological data analysis. Annual Review of Statistics and Its Application, 5:501–532, 2018.

[53] Robin J Wilson. Introduction to graph theory. Pearson Education India, 1979.

[54] Tero Aittokallio and Benno Schwikowski. Graph-based methods for analysing networks in cell biology. Briefings in bioinformatics, 7(3):243–255, 2006.

[55] Banu Dost, Tomer Shlomi, Nitin Gupta, Eytan Ruppin, Vineet Bafna, and Roded Sharan. QNet: a tool for querying protein interaction networks. Journal of Computational Biology, 15(7):913–925, 2008.

[56] Pourya Naderi Yeganeh, Chrsitine Richardson, Erik Saule, Ann Loraine, and M Taghi Mostafavi. Revisiting the use of graph centrality models in biological pathway analysis. BioData mining, 13(1):1–23, 2020.

[57] Young Kim and Sunwon Park. Supply network modeling using process graph theory: a framework for analysis. In 2006 SICE-ICASE International Joint Conference, pages 1726–1729. IEEE, 2006.

[58] Tyler Wilson, Abheek Chatterjee, and Astrid Layton. Exploring the effects of partnership and inventory for supply chain resilience using an ecological network analysis. In International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, volume 86250, page V005T05A011. American Society of Mechanical Engineers, 2022.

[59] Gamage Sanka Nirodha Perera and Hans-Gerd Maas. Cycle graph analysis for 3d roof structure modelling: Concepts and performance. ISPRS Journal of Photogrammetry and Remote Sensing, 93:213–226, 2014.

[60] Franziska Berger, Peter Gritzmann, and Sven de Vries. Minimum cycle bases for network graphs. Algorithmica, 40:51–62, 2004.

[61] Florian Dörfler, John W Simpson-Porco, and Francesco Bullo. Electrical networks and algebraic graph theory: Models, properties, and applications. Proceedings of the IEEE, 106(5):977–1005, 2018.

[62] Telikepalli Kavitha, Christian Liebchen, Kurt Mehlhorn, Dimitrios Michail, Romeo Rizzi, Torsten Ueckerdt, and Katharina A Zweig. Cycle bases in graphs characterization, algorithms, complexity, and applications. Computer Science Review, 3(4):199–243, 2009.

[63] JiaQi Liu, XueRong Li, and JiChang Dong. A survey on network node ranking algorithms: Representative methods, extensions, and applications. Science China Technological Sciences, 64(3):451–461, 2021.

[64] Aarti Jain and BVR Reddy. Node centrality in wireless sensor networks: Importance, applications and advances. In 2013 3rd IEEE International Advance Computing Conference (IACC), pages 127–131. IEEE, 2013.

[65] Luiz Filipe M Vieira, Marcelo G Almiron, and Antonio AF Loureiro. Link probability, node degree and coverage in three-dimensional networks. Ad Hoc Networks, 37:153–159, 2016.

[66] Michele Borassi, Pierluigi Crescenzi, Michel Habib, Walter A Kosters, Andrea Marino, and Frank W Takes. Fast diameter and radius bfs-based computation in (weakly connected) real-world graphs: With an application to the six degrees of separation games. Theoretical Computer Science, 586:59–80, 2015.

[67] James R Munkres. Elements of algebraic topology. CRC press, 2018.

[68] Alexander Smith, Spencer Runde, Alex K Chew, Atharva S Kelkar, Utkarsh Maheshwari, Reid C Van Lehn, and Victor M Zavala. Topological analysis of molecular dynamics simulations using the euler characteristic. Journal of Chemical Theory and Computation, 2022.

[69] Shengli Jiang, Nanqi Bao, Alexander D. Smith, Shraddha Byndoor, Reid C. Van Lehn, Manos Mavrikakis, Nicholas L. Abbott, and Victor M. Zavala. Scalable extraction of information from spatiotemporal patterns of chemoresponsive liquid crystals using topological descriptors. The Journal of Physical Chemistry C, 127(32):16081–16098, 2023.

[70] Rahul R Shah and Nicholas L Abbott. Principles for measurement of chemical exposure based on recognition-driven anchoring transitions in liquid crystals. Science, 293(5533):1296–1299, 2001.

[71] Rebecca J Carlton, Jacob T Hunter, Daniel S Miller, Reza Abbasi, Peter C Mushenheim, Lie Na Tan, and Nicholas L Abbott. Chemical and biological sensing using liquid crystals. Liquid crystals reviews, 1(1):29–51, 2013.

[72] Carina Esteves, Efthymia Ramou, Ana Raquel Pina Porteira, Arménio Jorge Moura Barbosa, and Ana Cecília Afonso Roque. Seeing the unseen: the role of liquid crystals in gas-sensing technologies. Advanced optical materials, 8(11):1902117, 2020.

[73] Nanqi Bao, Jake I Gold, Jonathan K Sheavly, James J Schauer, Victor M Zavala, Reid C Van Lehn, Manos Mavrikakis, and Nicholas L Abbott. Ordering transitions of liquid crystals triggered by metal oxide-catalyzed reactions of sulfur oxide species. Journal of the American Chemical Society, 144(36):16378–16388, 2022.

[74] Tibor Szilvási, Nanqi Bao, Karthik Nayani, Huaizhe Yu, Prabin Rai, Robert J Twieg, Manos Mavrikakis, and Nicholas L Abbott. Redox-triggered orientational responses of liquid crystals to chlorine gas. Angewandte Chemie, 130(31):9813–9817, 2018.

[75] Shutao Li, Weiwei Song, Leyuan Fang, Yushi Chen, Pedram Ghamisi, and Jon Atli Benediktsson. Deep learning for hyperspectral image classification: An overview. IEEE Transactions on Geoscience and Remote Sensing, 57(9):6690–6709, 2019.

[76] Qu Shenming, Li Xiang, and Gan Zhihua. A new hyperspectral image classification method based on spatial-spectral features. Scientific Reports, 12(1):1541, 2022.

[77] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. CoRR, abs/1811.01457, 2018.

[78] John HC Whitehead. Combinatorial homotopy i. Bull. Amer. Math. Soc, 55(3):213–245, 1949.

[79] Blake Lopez, Jiaze Ma, and Victor M Zavala. Graph-based optimization for technology pathway analysis: A case study in decarbonization of university campuses. arXiv preprint arXiv:2311.00809, 2023.

[80] Philip A Tominac and Victor M Zavala. Economic properties of multi-product supply chains. Computers & Chemical Engineering, 145:107157, 2021.

[81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics, 11(2):109–125, 1981.

[82] David Cheng Zarate, Pierre Le Bodic, Tim Dwyer, Graeme Gange, and Peter Stuckey. Optimal sankey diagrams via integer programming. In 2018 IEEE pacific visualization symposium (PacificVis), pages 135–139. IEEE, 2018.

[83] Lyndon White and contributors. LayeredLayouts.jl, 6 2020. https://github.com/oxinabox/LayeredLayouts.jl. Accessed 14 March 2023.

[84] yworks. yFiles. https://www.yworks.com/yfiles-overview. Accessed 14 March 2023.

[85] Richard Bellman. On a routing problem. Quarterly of applied mathematics, 16(1):87–90, 1958.

[86] Edsger W Dijkstra. A note on two problems in connexion with graphs. In Edsger Wybe Dijkstra: His Life, Work, and Legacy, pages 287–290. 2022.

[87] Robert W Floyd. Algorithm 97: shortest path. Communications of the ACM, 5(6):345, 1962.

[88] Chin Yang Lee. An algorithm for path connections and its applications. IRE transactions on electronic computers, (3):346–365, 1961.