# The graph2tab library, an introduction

This package is a generic implementation of a method for producing spreadsheets out of pipeline graph. The core of this processing is based on the node splitting approach: all the nodes in the input pipeline are reduced to ‘chain nodes’, i.e.: nodes that have at most one input and at most one output. This is done by taking those nodes having splittings or pooling, creating copies of them and distributing the excessive inputs or outputs over the copies. See the class `ChainsBuilder` for details about such procedure.

The package is designed in a way that allows to adapt a particular object model (eg.: ISATAB or MAGETAB) and to reuse classes like `TableBuilder` to produce spreadsheets describing a particular experimental pipeline (ie.:CSV/TSV files).

# Summary



## To-do list

In a nutshell, what you've to do to use graph2tab, in most cases is:

- define your node wrappers and the attributes the yield
- define your node factory
- define your table builder
- if needed, define some criteria to sort nodes, e.g.: tells the library that ‘Source Name’ comes before ‘Sample Name’
- invoke your table builder with the right set of nodes coming from your to-be-converted model (such as the sources or the end nodes). Collect the results as a matrix, the SDRF-like table.

Let's see the details below.

## Set up

The graph2tab library is available as a Maven Java project. The quickest way to use it is to include it as a dependency in your Maven project:

```
<dependency>
  <groupId>org.isatools</groupId>
  <artifactId>graph2tab</artifactId>
  <version>3.0</version>
</dependency>
```

```
<repository>
  <id>oerc</id>
  <name>Oxford e-Research Center</name>
  <url>http://frog.oerc.ox.ac.uk:8080/artifactory/repo/</url>
</repository>
```

You may want to download the sources, especially to look at the examples we describe in this guide. The sources are available via GIT at the address (TODO: will change):

```
https://github.com/marco-brandizi/ISAvalidator-ISAconverter-BIImanager/tree/master/graph2tab
```

We have never tried it without using Maven. If that is your case, we suggest that you use the Assembly plug-in from the sources to build a Jar, that can be included in your project:

TODO: this is not ready yet, we need to add the Assembly descriptor for this.

## The basics

graph2tab can be used to convert a graph about an experimental pipeline to a tabular view, similar to the SDRF structure in the MAGETAB format, or the Sample/Assay files in the ISATAB format.

In order to do that, you must start from a Java object model that is used to represent the experimental pipeline (e.g.: the BII model, or the model defined in the Limpopo MAGETAB parser). For example, you'll have instances of classes like `BioSource` or `Hybridization` and these will have properties like factorValues, characteristics, inputs, outputs.

graph2tab is designed to be generic: apart from few basic elements, the graph/table conversion does not depend neither on the particular graph that is being converted, nor on the particular set of headers that the output consists of. The library ensures this generality by working with a few basic interfaces.

*Node* is one of the most important of them, it just defines inputs, outputs and attributes. The former point at other instances of the `Node` interface itself. The attributes are defined in in a tabular-oriented way, that is: every attribute is a pair of strings, the header and the value. These will correspond to headers/values in the spawn table format. The way this is implemented consists of two lists of strings, defined in `TabValueGroup`, one for the headers and one for the values.

### uk/ac/ebi/tablib/export/graph_algorithm/Node.java

```
package uk.ac.ebi.tablib.export.graph_algorithm;

public interface Node extends Comparable<Node>
{
   public SortedSet<Node> getInputs ();
   public boolean addInput ( Node input );
   public boolean removeInput ( Node input );

   public SortedSet<Node> getOutputs ();
   public boolean addOutput ( Node output );
   public boolean removeOutput ( Node output );

   public List<TabValueGroup> getTabValues ();

   public String getType ();
   public int getOrder ();

   public Node createIsolatedClone ();
}
```

### uk/ac/ebi/tablib/export/graph_algorithm/TabValueGroup.java

```
public interface TabValueGroup
{
	public String getHeader();
	public String getValue();
	public List<TabValueGroup> getTail ();
}
```

An example of how these interface can be used is provided in the package `uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests` (test/ folder). Here, you've two examples of how a to-be-converted model can be adapted to graph2tab. One is very simple and it's in the package `dummy_graphs_tests`. Here nodes from the input model are directly implementing out `Node` interface. While this is so simple, it's not very realistic, because you will typically have your model and you will not want to derive it from classes in the `grap2tab` package. So, in such cases, what you need is something similar to the example in the package `simple_biomodel_tests`. Let's describe this in detail. Suppose that the experiment model you've to convert has a node like `BioMaterial`, which has this shape:

### uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/model/BioMaterial.java

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.model;

public class BioMaterial extends ExperimentNode
{
	public BioMaterial ( String name )
	{
		super ( name );
	}

	/**
	 * Biomaterial characteristics are managed by means of {@link Annotation}.
	 */
	public void addCharacteristic ( String type, String value, String termAcc, String termSrc )
	{
		super.addAnnotation ( type, value, termAcc, termSrc );
	}

	/**
	 * Biomaterial characteristics are managed by means of {@link Annotation}.
	 */
	protected List<Annotation> getCharacteristics () {
		return super.getAnnotations ();
	}

}
```

### uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/model/Annotation.java

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.model;

public class Annotation
{
	...

	public Annotation ( String type, String value, String termAcc, String termSrc )
	{
	...
	}

	public Annotation ( String type, String value, OntoTerm ontoTerm ) {
	...
	}
```

```
        public Annotation ( String type, String value ) {
        ...
        }

        public String getType () {
        ...
        }

        public String getValue () {
        ...
        }

        public OntoTerm getOntoTerm () {
        ...
        }

}
```

As you can see, `Annotation` defines the possible attributes for the input model. This is an example quite similar to the kind of attributes the nodes have in MAGE-OM, MAGETAB, ISATAB. In order to be adaptable to concrete specific situations, the graph2tab library sees the model nodes in an abstract generic form, where the node is able to tell what are its inputs and outputs, plus the list of header/value pairs that needs to be used to represent it in the final result table, that is a list like `Source Name = 'source 1', Characteristics[Organism] = 'Mus-musculus', …` Most of the work that you have to do to use the library for your particular case consists of writing code to convert the attributes from your model, whatever they are, to such string pairs, i.e.: you've to convert them to instances of `TabValueGroup`. This is an interface defined this way:

```
public interface TabValueGroup
{
        public String getHeader();
        public String getValue();
        public List<TabValueGroup> getTail ();
}
```

As you can see it is a recursive data structure, the reason for that is that certain groups of attribute pairs must be kept always together, in order to produce correct results. For example, `Characteristics[Organism] = 'Mus-musculus', Term Source REF = 'NCBI-Tax', Term Accession = '123'` must be the a successive groups of columns in the resulting table. Hence, a good encoding of this example would be[1]:

```
TabValueGroup (
     header = "Characteristics [ Organism ]"
     value = "Mus-musculus"
     tail = (
       TabValueGroup (
         header = "Term Source REF"
         value = "NCBI-Tax"
         tail = (
           TabValueGroup (
             header = "Term Accession"
             value = "123"
           )
         )
       )
     )
   )
```

With this kind of nesting you are telling grap2tab that the 'NCBI-Tax' term source goes always next to the organism annotation and, that, in turn, the 123 accession goes always next to the NCBI-Tax thing (ie,

1 Note this is pseudo-code, not something defined in a particular real language.

the three of them are kept together in the final result).

You may have noted the existence of the methods `getType()` and `getOrder()`. This is explained in the section ‘Uneven graphs and their layering’.

## Model wrapping

The typical approach to implement such an adaptation from your model to the graph2tab interfaces is defining wrappers. Let's go straight to an example about the aforementioned `Biomaterial`. This derives from another class, written for this particular model, which is the abstract `ExpNodeWrapper`.

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/ExpNodeWrapper.java**

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.node_wrappers;

...
public abstract class ExpNodeWrapper extends DefaultAbstractNode
{
	private ExperimentNode base;

	ExpNodeWrapper ( ExperimentNode base )
	{
		this.base = base;
	}

	/**
	 * This is used by the implementation of {@link Node#createIsolatedClone()}. Essentially, copies
	 * the wrapped node and makes empty input/output sets.
	 *
	 */
	protected ExpNodeWrapper ( ExpNodeWrapper original )
	{
		this.base = original.base;
		this.inputs = new TreeSet<Node> ();
		this.outputs = new TreeSet<Node> ();
	}

	/**
	 * In this case we're able to write a generic method that is customised by the descendants.
	 * The methods shows how to add an ontology term to a free text value. This is done by keeping
	 * all into the same {@link TabValueGroup}.
	 *
	 * @param nameHeader eg, "BioMaterial Name", "Protocol REF"
	 * @param annHeaderPrefix eg: "Characteristic", "Parameter Value", here headers will be built
	 * with the schema annHeaderPrefix [ type ], eg: Characteristic [ Organism ]
	 *
	 */
	protected List<TabValueGroup> getTabValues ( String nameHeader, String annHeaderPrefix )
	{
		List<TabValueGroup> result = new ArrayList<TabValueGroup> ();
		result.add ( new DefaultTabValueGroup ( nameHeader, base.getName () ) );

		for ( Annotation annotation: base.getAnnotations () )
		{
			DefaultTabValueGroup tbg = new DefaultTabValueGroup (
				annHeaderPrefix + " [ " + annotation.getType () + " ]",
				annotation.getValue () );
			OntoTerm ot = annotation.getOntoTerm ();
			if ( ot != null )
			{
				tbg.append (
					new DefaultTabValueGroup ( "Term Accession Number", ot.getAcc (),
						new DefaultTabValueGroup (
							"Term Source REF", ot.getSource () )
					));
```

```
                        }
                        result.add ( tbg );
                }
                return result;
        }

        /**
         * If it's not a clone produced by {@link Node#createIsolatedClone()},
         * it uses {@link NodeFactory} to build wrappers
         * for the input nodes of the base and to return them.
         *
         * This is the typical way this method is implemented by.
         */
        @Override
        public SortedSet<Node> getInputs ()
        {
                if ( inputs != null ) return super.getInputs ();
                inputs = new TreeSet<Node> ();
                NodeFactory nodeFact = NodeFactory.getInstance ();
                for ( ExperimentNode in: base.getInputs () ) inputs.add ( nodeFact.getNode ( in ) );
                return super.getInputs ();
        }

        ...
        @Override
        public SortedSet<Node> getOutputs ()
        {
                ...
        }
        ...
}
```

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/BioMaterialWrapper.java**

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.node_wrappers;
...

public class BioMaterialWrapper extends ExpNodeWrapper
{
        BioMaterialWrapper ( BioMaterial base ) {
                super ( base );
        }

        private BioMaterialWrapper ( ExpNodeWrapper original ) {
                super ( original );
        }

        public List<TabValueGroup> getTabValues () {
                return getTabValues ( "Biomaterial Name", "Characteristic" );
        }

        public Node createIsolatedClone () {
                return new BioMaterialWrapper ( this );
        }

}
```

## Details

You're free to directly implement the `Node` interface for defining your wrappers. However, for common cases, extending `DefaultAbstractNode` is probably quite easier. The latter defines common functionality, such as the collection class members for maintaining input/output nodes.

Differently than `DefaultAbstractNode`, which is a general class and part of the main package, `ExpNodeWrapper` makes sense in this example only (it's in the test/ folder too) and define a few common elements that apply to the simple model used for this particular case.

Every node type in your original model has a corresponding wrapper and the two are linked by the base object received by the constructor.

`getInputs(), getOutputs()`. Your custom implementation will typically behave like (for `getInput()`): if the inputs collection != null returns it, else initialise it by getting wrappers for each node that is input of the base node for the current wrapper. Wrappers are usually created via a custom wrapper factory (more later on that). In short, what we're doing is creating a graph of wrappers that is isomorphic to the graph of nodes from your to-be converted model.

`getTabValues()`. In this particular model (yours may be more complicated), there is only one type of attribute that can be attached to nodes: the `Annotation` class. So the `getTabValues()` method converts the attribute into the required list of headers/values. In this foo example, each header is typed, ie.: it is like `'Characteristics[ Organism ]'`, every attribute has a string value and can have an optional ontology term, which, in turn is made of an accession and a source. Note that the latter two are nested, as explained in the previous section.

So, the wrapper defines a node identifier first `TableValueGroup` (eg.: `'BioMaterial'/'source1'`), followed by one `TableGroup` per `Annotation`, which contains the annotation type, its value and, optionally, additional pairs like `'Term Accession Number'/'123', 'Term Source REF'/'MO'`. The ontology bits are nested.

`createIsolatedClone()` and `ExpNodeWrapper( ExpNodeWrapper original )`. The conversion algorithm needs to duplicate nodes. That is, given a node, it needs to create another wrapper node that has the same base (points to the same node in your original model), but different edges (ie: different inputs/outputs), taken from the original node. For instance, a 2-input/2-output node is split into two nodes, the second node will get one input and one output from the original, which will be left with the other two edges. The algorithm decides which edges to remove/add, so the duplicated node is initially isolated (no inputs/outputs). Now, `createIsolatedClone()` provides such a duplicate of the current node (ie: of `this`). The typical implementation of such method is something like `return new ExpNodeWrapper( this )`. So the constructor of this type (the ones that receive an instance of the same type the class is), as you can see, actually creates the isolated duplicate. Indeed, it's quite simple: the new wrapper gets the base from the original one and initialises inputs/outputs with empty sets (see `ExpNodeWrapper` above).

**More notes**

- The order the attributes are defined with in the wrappers is reflected in the final output table (this also depends on how nodes of the same types are merged, see below).

- As explained above, those header/value pairs that go together have to be nested, starting from one `TableGroup` object and adding other `TableGroup` to its tail (and tail of tail, in a tree-like recursive fashion). This way, the headers (and hence columns) that must be reported together in the final table are actually never split apart during the graph-to-table conversion.

- By nesting table value groups this way, two nodes of the same type (eg: two sample nodes) may have different lists of attributes, eg: one has a `'Characteristics[cell line]'` attribute and the other doesn't, you just don't define any cell line or value for the second one. The converter takes care automatically of 'merging' the two (or more) samples, creating the right set of columns and populating the lines with proper values or empty cells.

- `getInputs()` and `getOutputs()` return `SortedSet`(s). This has a few implications. First, it's a bonus that allows you to establish some order on the output lines. Typically, nodes are compared on the basis of attributes like sample name or hybridization name. Second, you need to implement proper versions of `compareTo()`, `equals()` and `hashCode()`. `DefaultAbstractNode` has sensible default implementations for these methods that take the value of the value returned by the first `TableGroup` found in `getTabValues()`, i.e.,

things like the ‘Source Name’ or the ‘Sample Name’. This is usually what you're happy with. The implementation in `DefaultAbstractNode` also ensures a fundamental property: that two physically different node wrappers are considered different (e.g., by collection-related methods), even when they are duplicated and point at the same base (the same node from your to-be-converted model, which also means the same source or alike). This is really important, because unwanted behaviours would be observed otherwise. For example, without such an implementation, duplicated nodes would be considered the same when you attempted to store them in the same `Set` structure (so one of them would be discarded). So, the message is: extend `DefaultAbstractNode` or, should you have particular needs why this implementation doesn't suit you well, at least give a look to the source of this class. Plus, if you really decide to go for your own implementation, drop us an email in case of problems.

## Node Factory

If you adapt mage2tab to your model with the wrappers approach, you typically will use an extension of `AbstractNodeFactory` either. This just keeps a map of base nodes (nodes from your model) => wrappers. It uses it in the method `NodeWrapper getNode( Base node )`, which creates a new wrapper and links it to the the given base, but only if this is not created yet and already inside the node/wrapper map. Obviously, it stores the new wrapper in the map, to return it in subsequent calls that have the same base as parameter (instead of creating a new wrapper). This ensures that you create a 1-1 graph of wrappers from the original graph is based on your to-be-converted model. Indeed, you don't need to care about these details, you only need to implement `createNewNode()`, which will be like this:

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/NodeFactory.java**

```
public class NodeFactory extends AbstractNodeFactory<ExpNodeWrapper, ExperimentNode>
{
	private NodeFactory() {}

	private static final NodeFactory instance = new NodeFactory ();

	public static NodeFactory getInstance () {
		return instance;
	}

	@Override
	protected ExpNodeWrapper createNewNode ( ExperimentNode base )
	{
		if ( base instanceof BioMaterial) return new BioMaterialWrapper ( (BioMaterial) base );
		if ( base instanceof Data ) return new DataWrapper ( (Data) base );
		if ( base instanceof ProtocolRef) return new ProtocolRefWrapper ( (ProtocolRef) base );
		throw new IllegalArgumentException (
			"Node of type " + base.getClass ().getSimpleName () + " not supported"
		);
	}
}
```

This method is what the `getNode()` above will call, when it decides it needs a new wrapper.

## Table Builder

The `TableBuilder` class coordinates the conversion job. We suggest you extend it and define a method that receives nodes from your input graph, as it is done in the constructor `SimpleModelTableBuilder( Set<ExperimentNode> nodes )`:

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/SimpleModelTableBuilder.java**

```
public class SimpleModelTableBuilder extends TableBuilder
{
	public SimpleModelTableBuilder ( Set<ExperimentNode> nodes )
	{
		this.nodes = new HashSet<Node> ();
		NodeFactory nodeFact = NodeFactory.getInstance ();
		for ( ExperimentNode node: nodes ) this.nodes.add ( nodeFact.getNode ( node ) );
	}

}
```

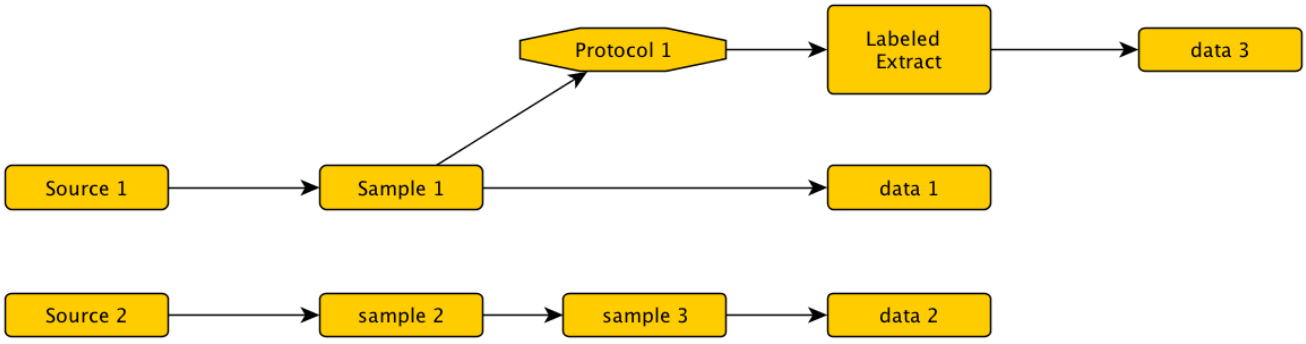
As you can see, this method just populates the nodes class member, containing node wrappers and does it by using your node factory. As for the nodes that should be passed to the table builder, any set of nodes that allow one to reach all the graph by walking from such nodes upstream (following inputs up to the sources) and downstream (following outputs up to the sinks/end nodes) is fine. For a discussion about optimal choices, see the next section about the layering algorithm.

## Go!

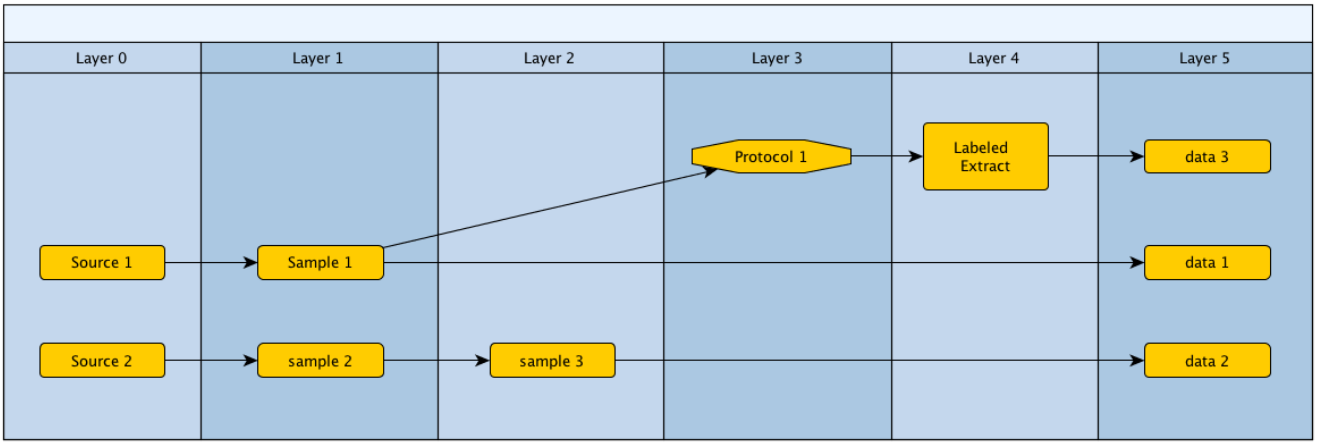
Once all is in place, you can go with the conversion, basically invoking `getTable()` from your table builder. An invocation example can be found in `SimpleBioModelTest`.

## Uneven graphs and their layering

What we have explained so far works correctly only upon the condition that the graph to be converted is 'even', which means that all the paths from the sources (e.g., bio-sources) to the end nodes (e.g., data) have the same lengths. graph2tab supports 'uneven' graphs as well, e.g., a graph where you may have paths of different lengths, with some experimental workflow stages omitted. An example is this:




This kinds of graphs are converted into tables by means of the same approach described above, which transforms the initial graph into a set of chains. However, in order to cope with such uneven paths, the input graph is pre-processed by a 'layering' component (if you are curious, this is implemented by the `LayersBuilder` class, but you don't need to know anything about it in order to use the library to convert your experimental workflows). This component distributes the graph nodes into a set of layers, where every layer reflect walks from sources to end nodes and the layering algorithm arranges the nodes into the layers in a way such that all the nodes in a given layer have the same type. In the example above, this would produce:




Once the graph is re-arranged this way (i.e.: every node gets a layer index), it is easy to apply the chaining procedure above and to generate a table that has gaps:

| **Source Name** | **Sample Name** | **Sample Name** | **Protocol REF** | **Labeled Extract Name** | **Data File** |
|---|---|---|---|---|---|
| Source 1 | Sample 1 | | Protocol 1 | Labeled Extract | Data 3 |
| Source 1 | Sample 1 | | | | Data 1 |
| Source 2 | Sample 2 | Sample 3 | | | Data 2 |

The layering algorithm implemented in the library is essentially based on the method briefly described in the MAGE-TAB specification[2], with several necessary details (not explicitly considered in that description) added by us. Once a graph is adapted to the `Node` interface, there isn't much else that you have to care about in order to make graph2tab to compute the node layers. However, because of the generic nature of the library, there is some information that you are required to provide with. This is done via the `Node.getOrder()` method. It works like this: layer indexes are initially assigned to the graph based on the highest distance from their sources (e.g.: `'Protocol 1'` and `'Sample 3'` in the example above would get 2). After that, for each layer, all the nodes in that layer are compared each other. If two nodes in the same layer are of a different type, `getOrder()` is invoked to decide which one is supposed to shift on the right (the one having a bigger or equal order). If you expect to work with uneven graphs, you do have to provide an implementation of `getOrder()` in your custom implementation. Two typical ways to do this are:

- An array of header types is defined (ie, an array of of strings), such as `'Source Name', 'Sample Name', 'Extract Name' …` and the order of a node is computed by considering its first header (actually `Node.getType()`, see below) and taking the index of this header in such predefined array. For example, if the first header returned by a node is 'Sample', its order is 2, because in the predefined array headers it has that index. This approach is equivalent to providing a list of types which of order is known in advance because of the experimental stage they represent.
- An alternative is possible for the case the graph to be exported was initially generated from an existing tabular format (e.g.: loaded from a MAGE-TAB submission) and stored to a database. In such a case it is possible to track and store the column index the initial header of a node comes from and use such index as return value for `getOrder()`. This kind of solution is interesting for

2 Section 3.3.2 in http://annotare.googlecode.com/files/MAGE-TABv1.1.pdf

those who want to be able to re-export originally submitted data and reflect the initial submission as much as possible. For instance, in the example above the column for `'Protocol REF'` may have been determined according to the first criterion, assuming the input for the (labelling) protocol is omitted and the output is specified (as it is more often the case). The opposite may have been true and you could track this by using this second criterion for setting the order.

The use of the order approach is very flexible and can be used for many different custom cases. For example, you may decide to combine the two criteria above, using a predefined array for all the fields but the protocol references, for which you may want to apply the column position. Another more advanced example consists of considering the type of protocol a 'Protocol REF' node refers to and assigning the order on the basis of this. Then additional fields would be configured in the list of possible fields, e.g.: 'Sampling Protocol', 'Extraction Protocol' etc. This is the reason why the `Node` interface has the method `getType()`. Normally the type is the same as the first header (and this is the default implementation in `DefaultAbstractNode`). However, in a case like the one at issue, it may be that the first header is 'Protocol REF', while the node type is something like 'Extraction Protocol', achieved from the referred protocol. `getType()` could be useful when you have entities reported with the same header, but of slightly different subtype, e.g., you want to separate "treatment protocol" and "extraction protocol".

There will be certain fields that will have an undefined order (which corresponds to -1 in the graph2tab library), for instance `'Protocol REF'`. Where one of such fields have to be shifted, the layering procedure considers the order of the nodes on its left and on its right and the other node being compared, if this has an order different than -1. For instance, in the example above, the `protocol 1` is shifted on the right, since `sample 3`, the node that initially shares the layer with such protocol reference, has an order that is closer to the one of `sample 1` and `sample 2` than the lowest order found on its right, which is the one of `Labelled Extract` (order(sample 3) – order(sample 1/2) is 0, being the nodes of the same type). The rationale for this is to prefer a right shift of nodes like protocol references in such situations, because in real biological cases it is more frequent to report a protocol output together with a named protocol and omit the input, rather than the opposite. See the source of `LayersBuilder` for details about the implemented shifting criteria.

### *Layering example in the source code*

The graph2tab package contains a 'layered implementation' in the package `layering_tests`. Please see the Junit tests in the class `LayeringTests`. The class `ExpNodeWrapper`, which was reported above in reality contains an implementation of the `getOrder()` method that uses the first approach described before (the array of possible headers). The package `layering_tests.model` extends `simple_biomodel_tests.model` with a few nodes that makes it possible to build examples of uneven graphs.

This is a fragment from `ExpNodeWrapper`, showing how `getOrder()` is implemented there:

```
public abstract class ExpNodeWrapper extends DefaultAbstractNode
{
   ...
   public static final Map<String, Integer> TYPE_ORDER = new HashMap<String, Integer> ()
   {{
      put ( "Source Name", 0 );
      put ( "Sample Name", 1 );
      put ( "Extract Name", 2 );
      put ( "Labeled Extract Name", 3 );
      put ( "Assay Name", 4 );
      put ( "Data File Name", 5 );
   }};
   ...
   public int getOrder ()
   {
      String header = getType();
      Integer order = TYPE_ORDER.get ( header );
```

```
        return order == null ? -1 : order;
    }
    ...
}
```

### *Disabling the layering stage for even graphs*

By default the `TableBuilder` class creates and instance of the `LayersBuider` class and invokes the layering procedure, before doing the the rest of the conversion process. If you are absolutely sure that your custom implementation will deal with even graphs only, this layering step, while not affecting the results in your case, is not necessary. Therefore we have provided an option to disable it, so that you can save some processing time (and a little of RAM too) if you don't need this step. To disable the layering stage, see the `TableBuilder` constructor that accepts the parameter `isLayeringRequired`. This parameter is true by default.

### *Choosing the initial node set to be passed to the table builder*

We have already shown that `TableBuilder` needs a certain subset of nodes of your input graph, in order to reach all the nodes and work out the conversion to a table. A simple working choice for this consists of all the nodes the graph is composed of. However, you can speed things up a little by using these criteria:

- If the layering stage is enabled, as it is by default (see above), the best choice is to pass all the graph end-nodes, i.e.: all the nodes that have no output. This is because the layering stage needs to start working from the rightmost side of the graph and to compute the initial values for the layer indexes recursing from such nodes back to the sources. Hence, the `LayersBuilder` class will first of all walk the initial nodes you pass to the table builder toward right, seeking for such sink nodes. If you pass this type of nodes directly, this initial search will produce a minimal overhead.
- If the layering stage is disabled, the table builder is in a symmetric situation: the initial nodes are passed directly to `ChainsBuilder`, the class that transforms the graph into chains. Because in this case the graph has to be scanned in left-to-right direction, the chain builder will initially seek all the source nodes, doing a walk from initial nodes to the leftmost nodes that can be reached from them (i.e., all the nodes without any input). Consequently, in this case the fastest option is to pass your graph sources to the table builder.
- Note that when the layering is active and you pass sinks, as suggested above, `LayersBuilder` will compute the graph sources as part of its job and these, not the initial node set, will be those that will be passed to `ChainsBuilder` in this case. This confirms that the sinks is the best choice in such a case.