powsybl-diagram

Release 5.4.0

unknown

Jun 02, 2026

CONTENTS OF THIS WEBSITE

CHAPTER

ONE

LAYOUTS

1.1 Layout parameters

The `LayoutParameters` class gathers parameters to customize the layout of the graph, i.e. the way the different elements are positioned on the graph.

All parameters have default values.

Name	Type	Default value
$maxSteps$	`int`	1000
$timeoutSeconds$	`double`	15
$textNodesForceLayout$	`boolean`	false
$textNodeFixedShift$	`Point`	Point(100, -40)
$textNodeEdgeConnectionYShift$	`double`	25
$injectionsAdded$	`boolean`	false

Users can customize one or several parameters according to their needs.

In the following example, the `maxSteps` parameter and the `textNodesForceLayout` parameter are customized, the other parameters are left to their default values:

```
LayoutParameters layoutParameters = new LayoutParameters().setMaxSteps(500).
↪setTextNodesForceLayout(true);
```

Layout parameters are further described below.

1.1.1 Layout algorithm parameters

The `maxSteps` parameter

The `maxSteps` parameter represents the maximum number of iterations that an automatic layout algorithm is permitted to perform. The value assigned to this parameter strikes a balance between speed and rendering quality.

With `maxSteps = 1000`(default value)

With `maxSteps = 100`:

```
LayoutParameters layoutParameters = new LayoutParameters().setMaxSteps(100);
```

NB1: for a very simple network like the one displayed above, the difference in speed is not significant.

NB2: the maximum number of iterations is not always reached as there are typically other stopping criteria in layout algorithms.

The `timeoutSeconds` parameter

The `timeoutSeconds` parameter represents the maximum amount of time an automatic layout algorithm may spend. As a consequence, this parameter will limit the number of iterations performed by the algorithm if the time limit is reached before the maximum number of iterations is reached. Again, as for `maxSteps` parameter, the value assigned strikes a balance between speed and rendering quality.

1.1.2 Text node parameters

The `textNodesForceLayout` parameter

If the `textNodesForceLayout` parameter is set to `true`, the text box nodes are positioned by the force layout algorithm. If the parameter is set to `false`, the text boxes are fixed in position relative to the voltage level node to which they are attached.

With `textNodesForceLayout = false` (default value)

With `textNodesForceLayout = true`

```
LayoutParameters layoutParameters = new LayoutParameters().setTextNodesForceLayout(true);
```

The `textNodeFixedShift` parameter

The `textNodeFixedShift` parameter represents the offset between the text box node and the voltage level node when the text node positions are fixed (i.e. when `textNodesForceLayout = false`).

With `textNodeFixedShift = Point(100, -40)` (default value)

With `textNodeFixedShift = Point(50, 40)`

```
LayoutParameters layoutParameters = new LayoutParameters().setTextNodeFixedShift(50, 40);
```

The `textNodeEdgeConnectionYShift` parameter

The `textNodeEdgeConnectionYShift` parameter is used to customize the position of the edge connection point on the text box.

With `textNodeEdgeConnectionYShift = 25` (default value)

With `textNodeEdgeConnectionYShift = 50`

```
LayoutParameters layoutParameters = new LayoutParameters().
↪setTextNodeEdgeConnectionYShift(50d);
```

1.1.3 Injection parameters

The `injectionsAdded` parameter

This parameter allows the user to display the injections that are present on the bus nodes of the voltage levels.

With `injectionsAdded = false`:

With `injectionsAdded = true`:

```
LayoutParameters layoutParameters = new LayoutParameters().setInjectionsAdded(true);
```

The represented injections are listed in the table below.

Icon in the DefaultLibrary	Injection type
	Generator
	Battery
	Load
	Shunt compensator (capacitor)
	Shunt compensator (inductor)
	Static VAR Compensator
	Unknown component

1.2 Atlas2 parameters

Apart from using layoutParameters, more parameters can be set by using the parameters from diagram-util. This is currently only available with Atlas2 (and not with basic force layout).

Atlas2 parameters can be built as follows:

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder()
        .withMaxSteps(500)
        .withRepulsion(3)
        .withSpeedFactor(1.2)
```

(continues on next page)

(continued from previous page)

```
        .build();
Atlas2ForceLayout atlas2ForceLayout = new Atlas2ForceLayout(new SquareRandomSetup<>(),␣
↪atlas2Parameters);
atlas2ForceLayout.run(graph, layoutParameters);
```

Here is an example on the ieee-118 nodes graph, with default parameters:

Ran in 106 steps, 55 ms

Note that once created, atlas2ForceLayout can be used on multiple different graph without having to create the object again (but the calculations will still have to be done again from the start).

1.2.1 Default values

Name	Type	Default value	Value range
$maxSteps$	`int`	6000	≥ 1
$repulsionIntensity$	`double`	4	0
$edgeAttractionIntensity$	`double`	1	0
$attractToCenterIntensity$	`double`	0.001	≥ 0
$speedFactor$	`double`	1	0
$maxSpeedFactor$	`double`	10	$speedFactor$
$swingTolerance$	`double`	1	0
$maxGlobalSpeedIncreaseRatio$	`double`	1.5	1
$attractToCenterForceEnabled$	`boolean`	true	true / false
$barnesHutTheta$	`double`	1.2	≥ 0
$quadtreeCalculationIncrement$	`int`	13	≥ 1

maxSteps

Change the maximum number of iteration the algorithm is allowed to run. Atlas2 has a stopping criterion, so for most networks, the run ends before the maximum number of steps is reached. Changing the maximum number of steps generally becomes relevant only when going past 8k nodes networks, with the default parameters.

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().withMaxSteps(50).
↪build();
```

repulsionIntensity

The coefficient of repulsion controls the intensity of the repulsion force between all nodes. Increasing this will make the network more sparse (ie nodes will be further apart).

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().withRepulsion(40).
↪build();
```

edgeAttractionIntensity

The coefficient of edge attraction controls the force between points that share an edge, increasing this might help with emphasizing clusters of points. It will also tend to make the graph smaller.

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().withEdgeAttraction(0.
↪5).build();
```

attractToCenterIntensity

The coefficient for the force that attracts all points to the center of the 2D space. Smaller values will lead to a less dense graph.

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().withAttractToCenter(0.
↪006).build();
```

speedFactor

Coefficient used to calculate individual point speed factor based on the global graph speed. The link between global and local speed is not a simple multiplication by this coefficient, but it is used in the calculation. If this is lower, points will be slower. A lower value might give worse results in terms of convergence speed, but it might help with stability. A value between 0.8 and 1.2 is generally good.

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().withSpeedFactor(2).
↪build();
```

maxSpeedFactor

The maximum factor for local speed compared to global speed. Lowering this might mean slower convergence, but it improves stability (ie points might swing less around their stability position). This value should always be bigger than the speedFactor (it still works if it's smaller, but it will work better if it's bigger). Increasing this too much opens the door to erratic behaviour.

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().
↪withMaxSpeedFactor(15).build();
```

swingTolerance

How much do we accept swing contributing to the global speed of the graph. A lower value means that we accept less swinging. If this value is too low, the global speed of the graph will be low and convergence will slow down. If it's too high, we can observe erratic behaviour in the way the points move. You probably shouldn't change this unless you know what you are doing.

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().withSwingTolerance(1.
↪3).build();
```

maxGlobalSpeedIncreaseRatio

How much can the global speed increase between each step. Higher means that the graph will reach a good global speed faster, but it might lead to more erratic behaviour between each step

```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().
↪withMaxGlobalSpeedIncreaseRatio(1.1).build();
```

activateAttractToCenterForce

Activate or deactivate the force that attracts points to the center of the graph. It is used to prevent non-connected points from drifting away. It is generally ill-advised to deactivate this, but if you are sure that everything is connected together then you can deactivate it

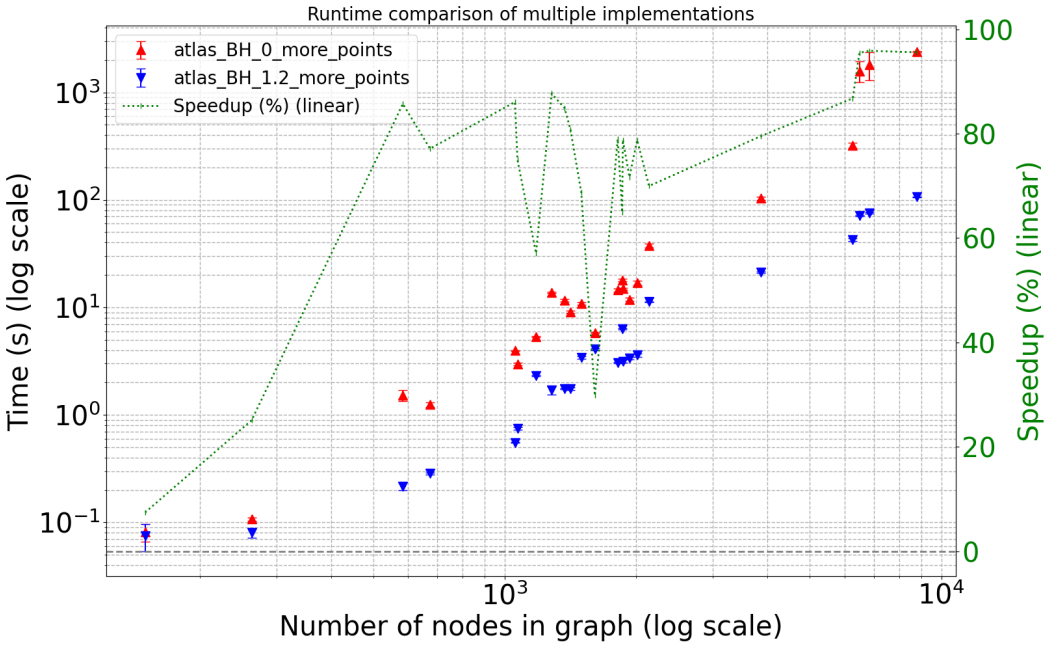
```
Atlas2Parameters atlas2Parameters = new Atlas2Parameters.Builder().
↪withActivateAttractToCenterForce(false).build();
```

barnesHutTheta

The θ parameter controlling the Barnes-Hut optimization. The higher the value, the more approximations we are doing. It is discouraged to use a value higher than 1.5~2

If your network is small (less than 500 points), you can set this to 0 using `withBarnesHutDisabled`, as it might be faster that way. This is especially relevant if you need to perform a force layout on many small networks (not so much if you are only doing it a few times).

Below is a graph showing the time taken to calculate a layout with Barnes-Hut (in blue) versus without it (in red).



The drop in relative performance for one of the networks in the middle is an exception due to the specificity of the shape of the network used.

Below are two images showing the visual result without (left) and with (right) Barnes-Hut. The value used in Barnes-Hut for the theta is the default of 1.2.

quadtreeCalculationIncrement

Barnes-Hut uses a quadtree to perform its optimization. A further optimization is to not re-calculate this quadtree at every step of the simulation, but at every $quadtreeCalculationIncrement$ step instead. For small values, it provides a noticeable increase in performance, but does not change the visual result too much. Generally, the larger your graph is, the bigger this value can be. A value bigger than 30 would probably be too much though.

Several layout factory implementations are available.

1.3 Force layout

1.3.1 Basic force layout

The layout factory `BasicForceLayoutFactory` is based on a force layout algorithm, with the following forces interacting on the voltage level graph:

- Repulsion Coulomb forces between all nodes;
- Attraction spring forces between nodes linked with an edge;

- Attraction force to the diagram center for all the nodes, to avoid the connected components to get further and further away at each step.

1.3.2 Atlas2 force layout

The layout `Atlas2ForceLayout` is based on a research paper for an algorithm (Atlas2). The forces used are:

- Linear repulsion force between all nodes
- Linear attraction force between nodes with an edge
- Attraction for to the diagram center for all the nodes, to avoid the connected components to get further and further away at each step.

Compared to the Basic force layout, it tends to give better results visually, and it also does it faster (for multiple reasons, like adaptative local speed for each point and forces that are easier to calculate). This is the preferred way to make graphs that are visually pleasing.

1.4 Fixed layout

The layout factory `FixedLayoutFactory` is based on a set of provided fixed positions, and on an additional layout. The provided additional layout is run only on voltage levels with missing positions.

1.5 Geographical layout

The layout factory `GeographicalLayoutFactory` is based on the geographical positions provided by the `SubstationPosition` extension, and on an additional layout.

First, a Mercator projection (scale factor may be specified) is used to put each latitude/longitude coordinate on an x/y plane. Note that to avoid overlapping, voltage levels within the same substation are placed on a circle (radius may be specified).

Then, the provided additional layout is run, with the previously mentioned projected coordinates fixed. Note that, by default, the additional layout is the basic force layout mentioned above, with some forces disabled:

- The repulsion Coulomb forces are only between non-fixed nodes;
- The attraction force to the diagram center is disabled.

CHAPTER

TWO

LABEL PROVIDER

The `LabelProvider` interface provides a way to customize labels.

2.1 Edge information

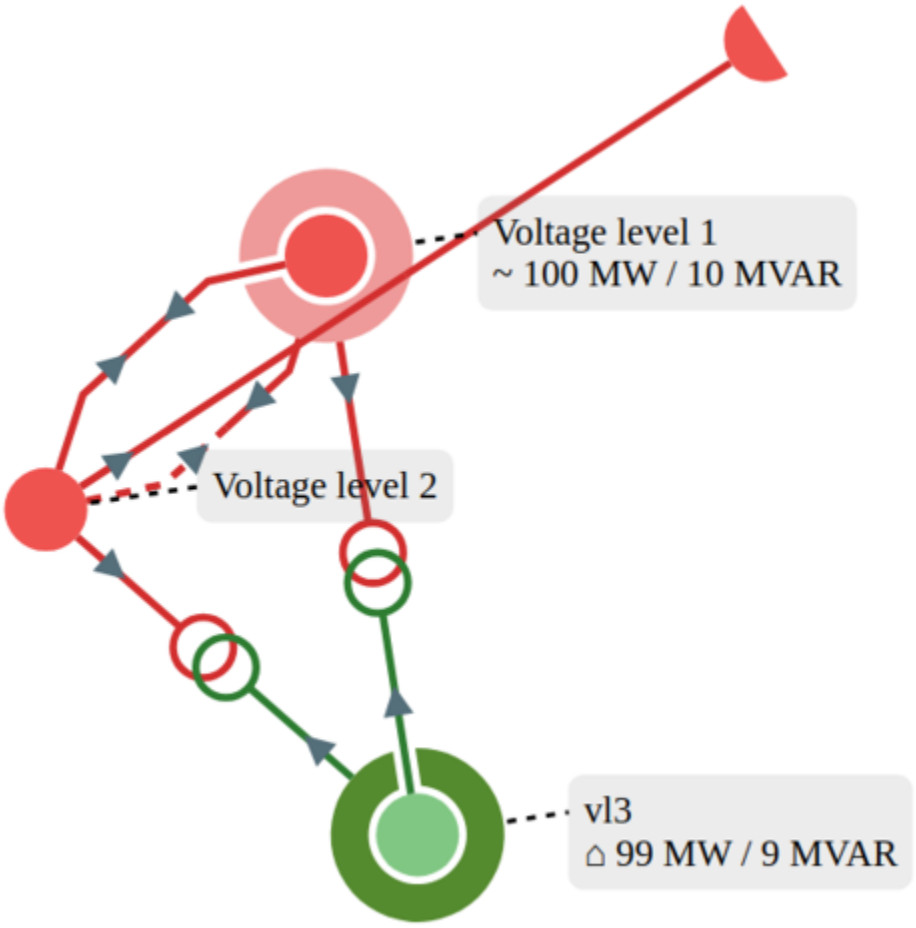
2.2 Voltage level description

2.3 Bus description

2.4 Voltage level details

2.4.1 Default implementation

If `svgParameters.isVoltageLevelDetails()` returns `true`, then the production (~) and consumption () of each voltage level is displayed as text. If there is no value (for production or consumption), the corresponding line in the text box is not displayed.


Voltage level 1
~ 100 MW / 10 MVAR
Voltage level 2
vl3
⌂ 99 MW / 9 MVAR

CHAPTER

THREE

STYLE PROVIDER

The `StyleProvider` interface provides a way to customize the appearance of the network-area diagram.

Regarding nodes and bus nodes, the color and the blinking is fully customizable. Regarding edges, the color, the width, the stroke and the blinking is also fully customizable

Currently, there are 2 implementations of the `StyleProvider`: the `NominalStyleProvider` and the `TopologicalStyleProvider`

3.1 Common features

The common features are factorized in the abstract classes `AbstractStyleProvider` and `AbstractVoltageStyleProvider`.

3.1.1 Overvoltage and undervoltage voltage levels

- Bus nodes which are over-voltage (meaning their voltage is above the voltage level high voltage limit) have their corresponding ring marked with class `nad-overvoltage`. If default CSS is used, this leads to the corresponding ring stroke blinking in orange.
- Similarly, for bus nodes which are under-voltage (meaning their voltage is below the voltage level low voltage limit); they have their corresponding ring marked with class `nad-undervoltage`. If default CSS is used, this leads to the corresponding ring stroke blinking in blue.

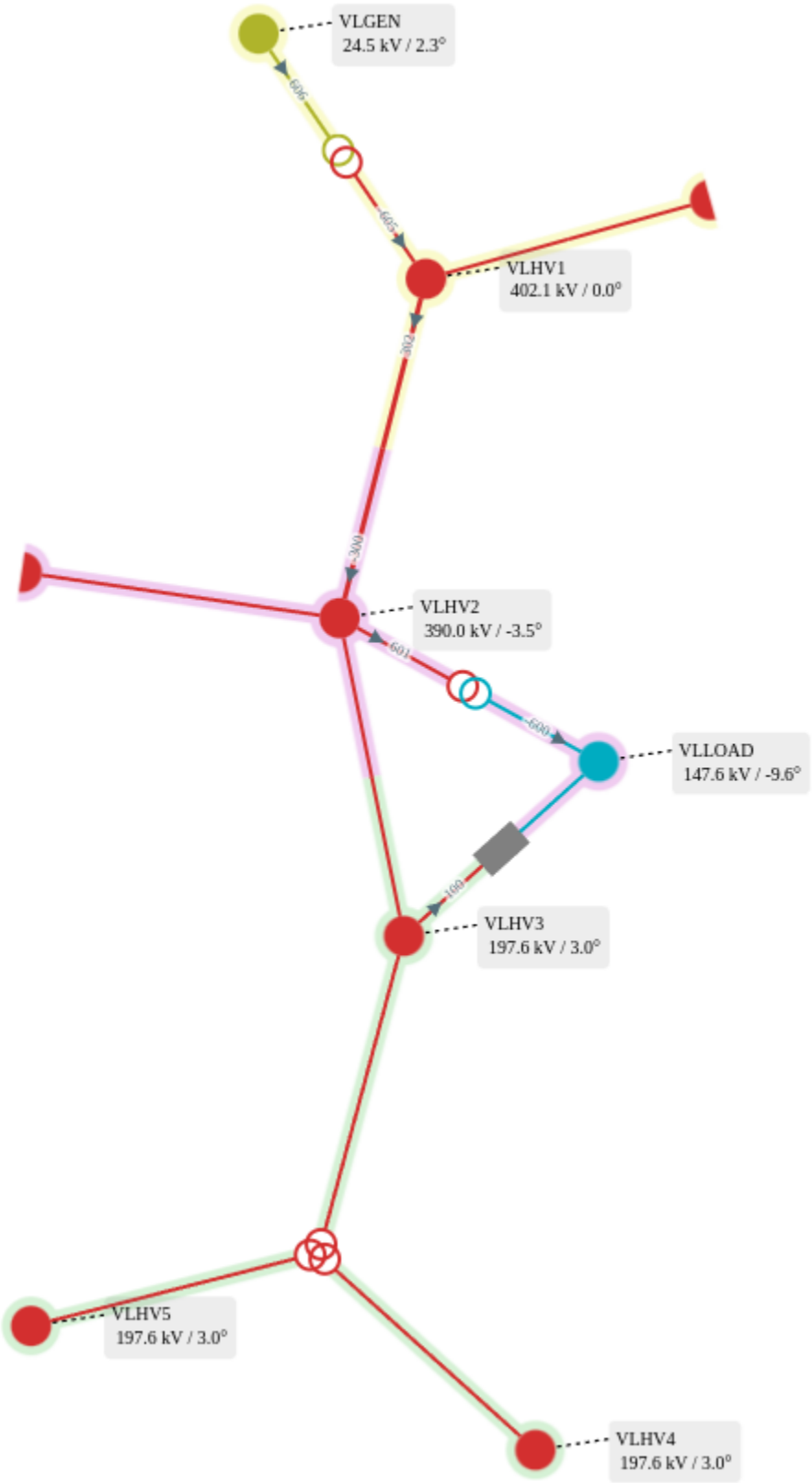
3.1.2 Overloaded lines

Branches which are overloaded are marked with class `nad-overload`. If default CSS is used, this leads to a branch yellow-blinking.

3.1.3 Subnetworks

The classes added by the style provider for the "highlight" graph duplicate are based on the subnetwork corresponding to each element. With default CSS used, this leads to subnetworks being highlighted.

> Note: the `SvgParameters` attribute highlightGraph`needs to be set to`true`, so that the SvgWriter adds that simplified "highlight" graph duplicate, which contains only branches and voltage level nodes.


VLGEN
24.5 kV / 2.3°
VLHV1
402.1 kV / 0.0°
VLHV2
390.0 kV / -3.5°
VLLOAD
147.6 kV / -9.6°
VLHV3
197.6 kV / 3.0°
VLHV5
197.6 kV / 3.0°
VLHV4
197.6 kV / 3.0°

3.2 NominalStyleProvider feature

The voltage level nodes are marked with a class depending on their nominal voltage, leading to one colour for each range of nominal voltages defined by the BaseVoltagesConfig.

3.3 TopologicalStyleProvider feature

The bus nodes of a voltage level are each marked with a class depending on the nominal voltage and on their bus index. This leads to a colour shading for each range of nominal voltages defined by the BaseVoltagesConfig. The text node of corresponding voltage level gives a legend for the colour shading, based on the selected given `LabelProvider` - the default being the voltage and angle.

CHAPTER

FOUR

EDGE ROUTING

The `EdgeRouting` interface provides a way to customize the computation of paths of the network-area diagram. The chosen edge routing can be specified through `NadParameters::setEdgeRouting`. Note that, internally, the edge routing is passed to the `SvgWriter` which launches the computation.

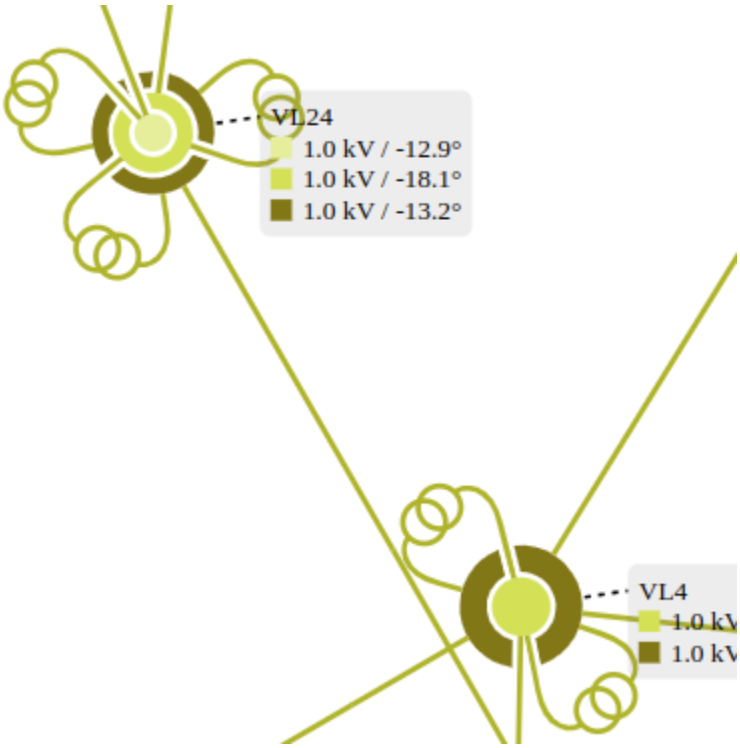
Currently, there are 2 implementations of the `EdgeRouting`: the `StraightEdgeRouting` and the `CustomPathRouting`. The default implementation if none specified is the `StraightEdgeRouting`.

4.1 Common features

The common features are factorized in the abstract class `AbstractEdgeRouting`.

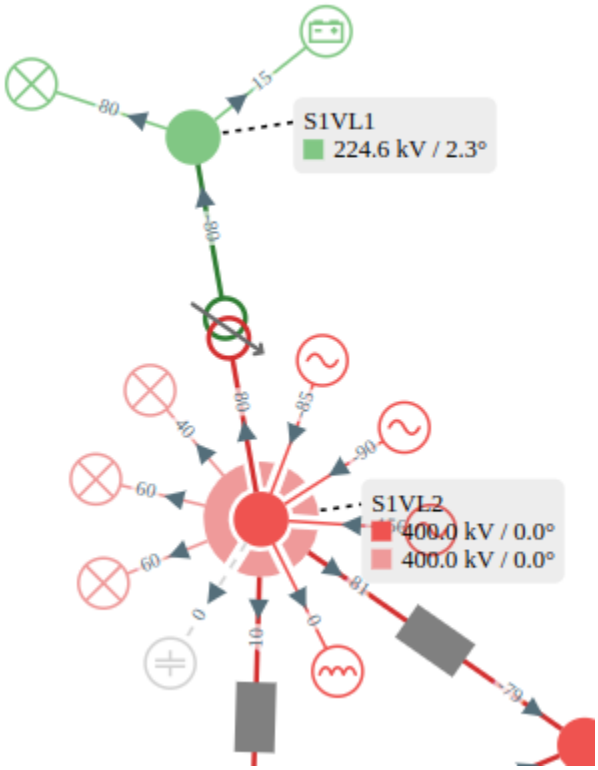
4.1.1 Loops

The loop edges points are computed so that they are distributed between branch edges when the space available is big enough.



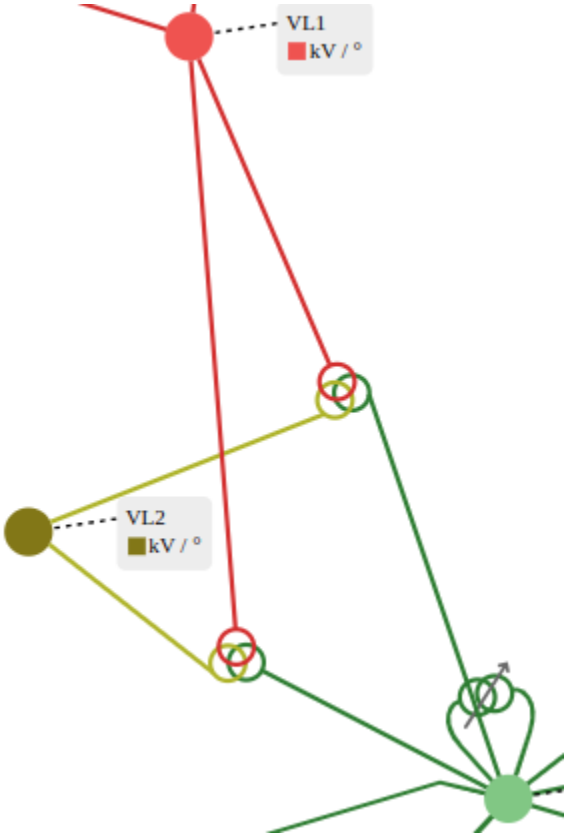
4.1.2 Injections

If injections are displayed, the injection edge points are computed similarly to loops: they are distributed between branch edges when the space available is big enough.



4.1.3 Three-winding transformers

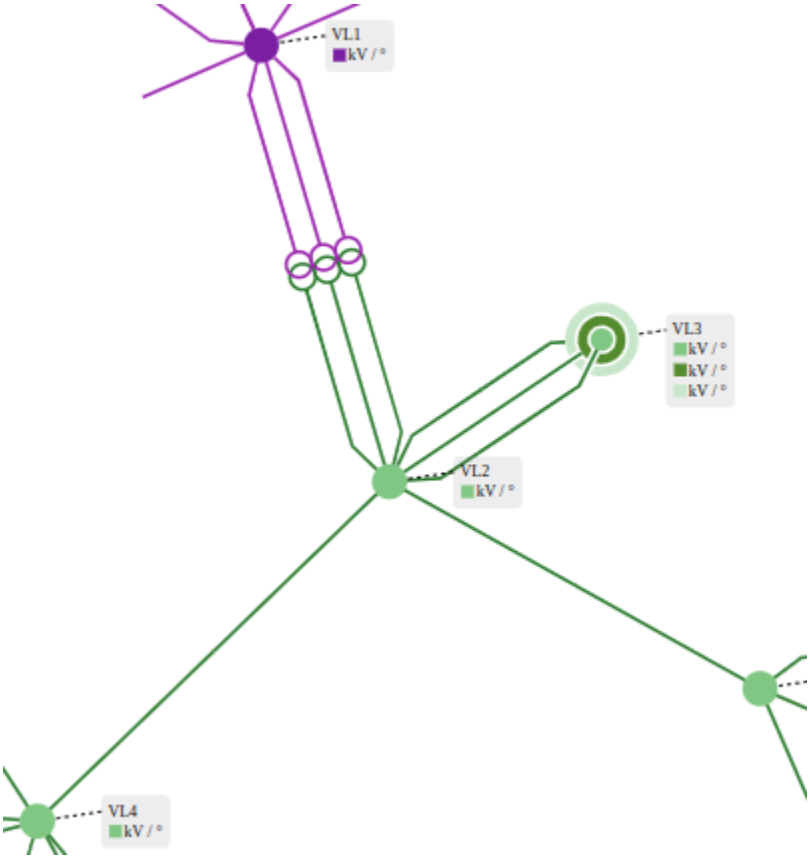
The three-winding edge points are computed by finding a "leading" transformer edge and then placing the other edges at 120°. The leading edge is defined as the opposite edge of the smallest aperture.



4.2 StraightEdgeRouting feature

In the `StraightEdgeRouting` implementation,

- the parallel edges points are computed so that they form a fork,
- the other edges points are computed as straight lines.

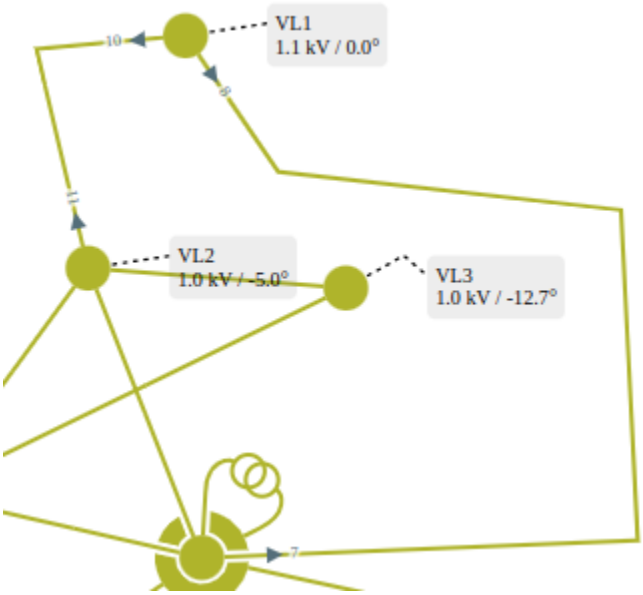


4.3 CustomPathRouting feature

In the `CustomPathRouting` implementation, custom paths can be provided through two maps in the constructor

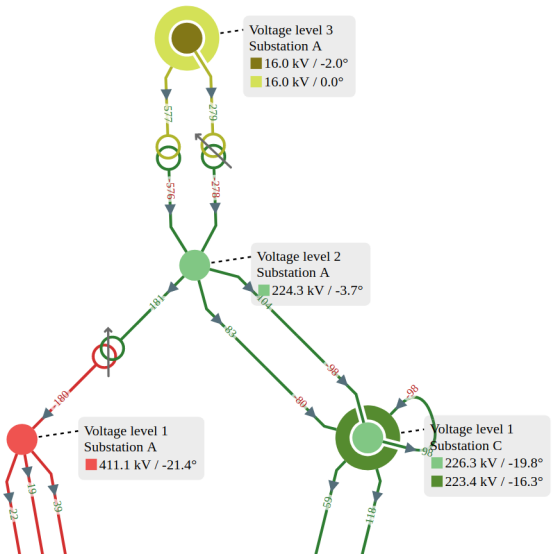
- a map whose keys are branch ids and whose values are the list of "bending" points to add to the corresponding branch edge,
- a map whose keys are voltage level ids and whose values are the list of "bending" points to add to the corresponding text edge.

If a branch id is missing in the given map, we fall back to `StraightEdgeRouting` implementation. Similarly for text edges, if a voltage level id is missing in the corresponding map, we fall back to `StraightEdgeRouting` implementation.



The powsybl-network-area-diagram artifact provides features to generate concise and customized diagrams of the network:

- Display of the graph whose nodes are the network voltage levels, and whose edges are the lines and transformers between those voltage levels;
- Generation of diagrams of the whole network or of part of the network, given a voltage level and a depth, or a list of voltage levels and a (unique) depth;
- Graph layout default implementation using a basic force layout algorithm, inspired by springy
- Diagram customization:
  - Possible use of your own graph layout implementation;
  - Possible use of your own label provider to display custom directed values on the graph edges (default label provider displays the active power);
  - Possible use of your own style provider to have a custom style for nodes and edges (default style provider gives the nodes and edges a class corresponding to their voltage level and gives disconnected lines a specific class);
  - Possible use of your own edge routing to have custom edge paths;
  - Possible use of your custom layout parameters and svg rendering parameters.



The powsybl-network-area-diagram artifact belongs to the powsybl-diagram repository. Before the 3.0.0 version, it was stored in an independent repository (now archived).

CHAPTER

FIVE

EXAMPLES

These examples show how to write a network-area diagram into an SVG file.

- Generate a network-area diagram for the network `network`

```
NetworkAreaDiagram.draw(network, Path.of("/tmp/diagram.svg"));
```

- Generate a network-area diagram for the network `network`, with customized `SvgParameters` and `LayoutParameters`

```
SvgParameters svgParameters = new SvgParameters().setFixedHeight(1000);
LayoutParameters layoutParameters = new LayoutParameters().setMaxSteps(300);
NadParameters nadParameters = new NadParameters().setSvgParameters(svgParameters).
↪setLayoutParameters(layoutParameters);
NetworkAreaDiagram.draw(network, Path.of("/tmp/diagram2.svg"), nadParameters,␣
↪VoltageLevelFilter.NO_FILTER);
```

CHAPTER

SIX

SINGLE-LINE DIAGRAMS

6.1 SVG Parameters

The SLD `SvgParameters` is the configuration class to determine how to customize the SVG render of a single line diagram. The way to integrate it is to set a `SvgParameters` value object to a `SldParameters` before drawing the SVG.

Use example:

```
SvgParameters svgParams = new SvgParameters();
SldParameters sldParams = new SldParameters().setSvgParameters(svgParams);
SingleLineDiagram.draw(network, "VL1", Path.of("/tmp/vl1.svg"), sldParams);
```

6.1.1 Identifier and naming parameters

prefixId

Default value: is empty

The prefix to set on SVG generated identifiers (attributes ids and xml elements (cells)). It is useful to distinct id collapsing when several SVG are displayed on the same HTML page.

useName

Default value: `false`

Boolean value to determine whether we identify an equipment node (`EquipmentNode`) by its id or its name. When set to `true` the identification is based on the name.

diagramName

Default value: `null`

The diagram name to display in the title or SVG header

6.1.2 Value formatter parameters

The `ValueFormatter` consumes some of the `SvgParameters`, related to the value precision and language to format electrical values to display.

powerValuePrecision

Default value: `0`

Number of decimals for active and reactive power display (in kV).

```
svgParams.setPowerValuePrecision(2);
```

voltageValuePrecision

Default value: `1`

Number of decimals for voltage display (in kV).

currentValuePrecision

Default value: `0`

Number of decimals for current display (in A or kA, depending on `currentUnit` value).

angleValuePrecision

Default value: `1`

Number of decimals for phase angles display (in degrees °).

percentageValuePrecision

Default value: `0`

Number of decimals for percentage display (power report rate, etc..).

languageTag

Default value: `en`

Language tag to dertermine the `Locale` used for number formatting (decimal separator, etc…)

undefinedValueSymbol

Default value: `\u2014` (-) (em dash unicode for undefined value)

Symbol to display in place of a numerical value when that value is not defined (e.g. for undefined current, power, angle, percentage, voltage values for use cases: power flow not calculated, sensor missing).

6.1.3 Units parameters

currentUnit

Default value: is empty

Unit suffix displayed after active power values. Examples: “A”, “kA”. init_current_arrow.svg

```
svgParams.setCurrentUnit("A");
```

activePowerUnit

Default value: is empty

Unit suffix displayed after current values. Examples: “MW”, “kW”.

```
svgParams.setActivePowerUnit("MW");
```

reactivePowerUnit

Default value: is empty

Unit suffix displayed after current values. Examples: “MVAR”, “kVAR”.

```
svgParams.setReactivePowerUnit("MVAR");
```

6.1.4 Margin and spacing parameters

busInfoMargin

Default value: `0.0`

Horizontal offset of the busInfo (info about the busbar state) indicator related to the busbar. Can be negative in order to offset to the left. Useful for avoiding overlaps with other graphical elements.

feederInfosIntraMargin

Default value: `10`

Vertical spacing between data of a same feeder.
For instance, is P and Q are displayed then the distance between both values is set by this parameter.

```
svgParams.setFeederInfosIntraMargin(50);
```

feederInfosOuterMargin

Default value: `20`

External spacing between the equipment symbol (breaker, transformer, generator...) and the beginning of the feeder data block (active and reactive power...).

```
svgParams.setFeederInfosOuterMargin(50);
```

6.1.5 Labeling and tagging parameters

angleLabelShift

Default value: `15.` (in degree (°))

Defines the rotation angle applied to the feeder labels (resp. bus labels) when the parameter *labelDiagonal* (resp. *busLabelDiagonal*) is set to true.

```
svgParams.setAngleLabelShift(45);
svgParams.setLabelDiagonal(true);
svgParams.setBusLabelDiagonal(true);
```

labelCentered

Default value: `false`

When `true` equipments tags are centered.
When `false` equipments tags are aligned on the left or following the renderer position rules.

```
svgParams.setLabelCentered(true);
```

labelDiagonal

Default value: `false`

When `true` feeder labels are displayed diagonally with an angle set by the *angleLabelShift*. Useful for substations with many feeders to avoid text overlaps.

```
svgParams.setLabelDiagonal(true);
```

busLabelDiagonal

Default value: `false`

When `true` bus labels are displayed diagonally with an angle set by the *angleLabelShift*.

```
svgParams.setBusLabelDiagonal(true);
```

6.1.6 SVG rendering parameters

cssLocation

Default value: `CssLocation.INSERTED_IN_SVG`

Controls where and how the CSS is integrated within the generated SVG when adding the style in the `SvgWriter`. 3 modes available:

- `INSERTED_IN_SVG`: directly in the SVG via a <style> tag (recommended for standalone exports)
- `EXTERNAL_IMPORTED`: external style sheets file referenced in a <style> tag from the SVG thanks to an `@import` url. Requires the CSS to be accessible via the referenced URL. Useful for web integrations with dynamic themes.
- `EXTERNAL_NO_IMPORT`: no CSS style integrated or imported. The style sheets file must be related to the integration environment (via <link> in the HTML parent).

```
svgParams.setCssLocation(SvgParameters.CssLocation.EXTERNAL_NO_IMPORT);
```

svgWidthAndHeightAdded

Default value: `false`

Used in the default SVG writer.
When `true`, `width` and `height` attributes are added to the root <svg> tag and set with the diagram width and height.
When `false` viewBox is redefined to let the SVG more flexible for web inegration (CSS responsive).

```
svgParams.setSvgWidthAndHeightAdded(true);
```

avoidSVGComponentsDuplication

Default value: `false`

Used in the default SVG writer to chose between direct writing of components or creation of reusable definitions via <defs> tag.
When `true` identical graphic components in SVG are defined only once with a <defs> tag and then reused with <use href="...">. It reduces the SVG file weight, usefull for big nework with multipl identical components.

```
svgParams.setAvoidSVGComponentsDuplication(true);
```

drawStraightWires

Default value: `false`

When `true` connections between equipments are drawn with straight lines instead of routing algorithm calculated polyline paths. The renderer is more simple but too simple for complexes schemes readability.

```
svgParams.setDrawStraightWires(true);
```

6.1.7 Displaying parameters

feederInfoSymmetry

Default value: `false`

When `true` feeder data are displayed symmetrically on both sides of the equipment symbol.

```
svgParams.setFeederInfoSymmetry(true);
```

busesLegendAdded

Default value: `false`

When `true` a busbar legend is added to the diagram, explaining the mapping between CSS colors (on buses), base voltages value and busbar indices.

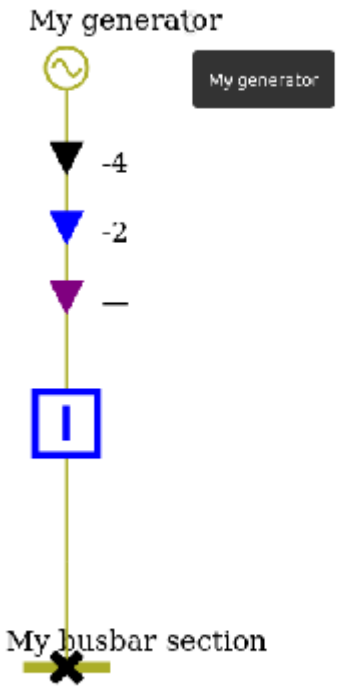
```
svgParams.setBusesLegendAdded(true);
```

tooltipEnabled

Default value: `false`

When `true`, a tooltip (native `<title>` tag) is added to graphical components when hovering it in JavaScript.

```
svgParams.setTootltipEnabled(true);
```



showGrid

Default value: `false`

When `true` a debugging grid is superimposed on the diagram, revealing the positioning grid used by the layout algorithm.

```
svgParams.setShowGrid(true);
```

showInternalNodes

Default value: `false`

When `true` internal nodes are displayed on the diagram (fictitious nodes not associated to physical equipments are rendered on the diagram).

```
svgParams.setShowInternalNodes(true);
```

displayEquipmentNodesLabel

Default value: `false`

When `true` text labels (id or name) are also displayed on the nodes representing equipments.

```
svgParams.setDisplayEquipmentNodesLabel(true);
```

displayConnectivityNodesId

Default value: `false`

When `true` connectivity nodes identifiers are displayed on the diagram. Should be used with the displaying of internal nodes parameter (see *ShowInternalNodes*)

```
svgParams.setDisplayConnectivityNodesId(true);
svgParams.setShowInternalNodes(true);
```

unifyVoltageLevelColors

Default value: `false`

When `true` all the base voltages of a sub station diagram are displayed with the same color instead of one color per base voltage. Useful for simplifying the readability or uniform graphical chart.

```
svgParams.setUnifyVoltageLevelColors(true);
```

6.2 GraphBuilder creation requirements

Implementing a GraphBuilder is the way to build the graph to be rendered by SingleLineDiagram. This shall implement builder for `VoltageLevelGraph`, `SubstationGraph` and `ZoneGraph`. Here are some hints that are to be considered.

6.2.1 VoltageLevelGraph

A `VoltageLevelGraph` is made of nodes that extend the `model.nodes.Node` class. The `Node` holds an `NodeType` enum that can take the following values:

- `BUS`: representing the BusBar, rendered as one straight horizontal line. The `BusNode` class extending `Node` is initialized with this value
- `FEEDER`: representing what connect the VoltageLevel to the outside, rendered on top or bottom of the voltageLevel. The `FeederNode` class extending `Node` is initialized with this value
- `INTERNAL` and `SWITCH`: representing nodes that constitute the graph connecting the Buses and the Feeders. Note that only the `SwitchNode` class should be initialized with `SWITCH`.

The graph shall be built:

- using `graph.NodeFactory` class for creating nodes. There are factories for each kind of nodes. If new nodes are needed, ensure to add the created node to the graph appropriately.
- connecting them with `VoltageLevelGraph.addEdge`

Components

Each node has a `componentTypeName` that is related to a `ComponentLibrary`.

- an easy way to add nodes with components that are not in the NodeFactory is to create a Node with `NodeFactory.createNode`. See `test.raw.TestAddExternalComponent`.
- Nodes that are to be drawn connected to a BusBar shall have this ability. No need to care about that rule as an adapted node (a `Node` with `BUS_CONNECTION` component) will be inserted between the bus and the node if the node is not directly connected to the bus. Note that:
  - This ability is given to some components in `layout.LayoutParameters.componentsOnBusbars`. By default, this is set with `DISCONNECTOR` (`BUS_CONNECTION` is implicit as it is necessary for the algorithm to work).
  - The `BUS_CONNECTION` component is the one that will be inserted if the component connected to the bus doesn't have this ability. Therefore, ensure the `ComponentLibrary` contains the `BUS_CONNECTION` component.

6.2.2 SubstationGraph

6.2.3 ZoneGraph

6.3 Layouts

6.3.1 Graph refiner

A preliminary step is done inside the `GraphRefiner` class. It performs some preprocessing on the input graph by calling several methods:

- An *optional* call to `Graph.substituteInternalMiddle2wtByEquipmentNodes()` to simplify the graph by substituting each internal two-winding transformers (i.e. both ends are in the same voltage level) `FeederNode` by a `EquipmentNode`, to avoid unnecessary snake lines;
- A *systematic* call to `GraphRefiner::handleConnectedComponents` which ensures that each connected component contains a `BusNode`: it is adding one if there is none;

- A *systematic* call to `Graph.substituteFictitiousNodesMirroringBusNodes()` to simplify the graph by removing `FICTITIOUS` nodes which are the only adjacent node of a `BusNode`;
- An *optional* call to `Graph.removeUnnecessaryConnectivityNodes()` to simplify the graph by removing redundant `FICTITIOUS` nodes;
- An *optional* call to `Graph.substituteSingularFictitiousByFeederNode()` to simplify the graph by replacing internal nodes with only one neighbor with a fictitious feeder node;
- An *optional* call to `Graph.removeFictitiousSwitchNode()` to simplify the graph by removing the fictitious switch nodes;
- A *systematic* call to `Graph.extendBusesConnectedToBuses()` to add 2 connectivity nodes between 2 buses that are connected to each other;
- A *systematic* call to `Graph.insertBusConnections` to create a connection between a bus node and its adjacent nodes;
- A *systematic* call to `Graph.insertHookNodesAtBuses()` to TODO
- A *systematic* call to `Graph.insertHookNodesAtFeeders()` to TODO
- A *systematic* call to `Graph.substituteNodesMirroringGroundDisconnectionComponent()` to deal with ground disconnector displaying.

6.3.2 Cell detection

A `Cell` represents a connected subgraph of nodes that participate in a common purpose.

The goal of the cell detection is to create these subgraphs that will then:

- Structure the organization of the busbar layout;
- Be displayed according to their types.

Cell types

Cells can be of one of the following enum `CellType`:

- `INTERN`
  The smallest subGraph delimited by `BUS` nodes (i.e., not including `FEEDER`) and that is not shunting `ExternCell` cells (see `SHUNT` definition below).
  Such cells instantiate the `InternCell` subclass.
- `EXTERN`
  The smallest subGraph delimited by `BUS` nodes and `FEEDER` nodes with at least one node having the following property: each branch extracted from this very node ends with nodes of a single `nodeType` among `BUS` or `FEEDER` or `SHUNT`.
  Such cells instantiate the `ExternCell` subclass.
- `SHUNT`
  A path between two `FictitiousNode` nodes of two `ExternCell` cells.
  Such cells instantiate the `ShuntCell` subclass.
- `ARCH`
  A path between an `ExternCell` and a `BusNode`.
  Such cells instantiate the `ArchCell` class.
  This type is obtained when an `ExternCell` is linked to two subsections, which cannot be displayed with the current algorithm. Such an `ExternCell` is then cut in an `ExternCell` only on one subsection, and in several `ArchCell`.

The figure shows examples of cells of several `CellType`.

Cell detection algorithm

The `ImplicitCellDetector` class implements an algorithm sticking to the above cell type definitions. The `detectCell` method is used to detect cells by exploring the graph.

It takes as parameters two lists of types that delimit the traversal algorithm :

- `stopTypes` list: for types that end a current branch
- `exclusionTypes` list: for types that invalidate the current subgraph.

The algorithm is explained based on the following graph that would result in the figure displayed to illustrate the cellTypes enum:

Step 1: identify `InternCell` cells

- stopTypes: BUS
- exclusionTypes: FEEDER

`InternCell` cells are easy to determine as being exclusively bordered by `BUS` nodes.

Step 2: identifies `ExternCell` cells

- stopTypes: BUS, FEEDER
- exclusionTypes:

If one node of the subgraph has each of its branches ending with one single kind of `NodeType` among `BUS` and `FEEDER`, ("*bottleneck*" node in the picture) this is an `ExternCell`.

Other `ExternCell` cells could be discovered in the next steps when adding the `SHUNT NodeType`.

Step 3: discriminates `EXTERN` and `SHUNT` cells

- stopTypes:
- exclusionTypes: BUS, FEEDER, SHUNT

To identify the first candidate `SHUNT` node, each `FICTITIOUS` node with more than 3 branches is visited. The expected property of the `SHUNT` node is that:

- At least one branch ends with only `BUS` nodes
- At least one branch ends with only `FEEDER` nodes
- One branch ends with `FEEDER` *and* `BUS` nodes.

The branches of the first two categories constitute the first `ExternCell` cell.

Then the `SHUNT` cell is constituted of:

- The first `SHUNT` node
- The string of nodes that have only 2 adjacent nodes
- The first node with more than 2 adjacent nodes that becomes the second `SHUNT` node

Last, the second `ExternCell` cell is build with the second `SHUNT` node and the remaining nodes.

The algorithm does not handle any other pattern.

6.3.3 Structure the cells into `Blocks`

Decompose each cell into `Block` and merge into the Cell Root Block.

Blocks are the building elements that are assembled hierarchically to organize the layout of the cells.

when a busbar on one vertical position is spanning over many busbars having another vertical position.

6.3.4 Position of `BusNodes` and `Cells` order

BSCluster

VerticalBusSet

We have the principle that all the `BusNodes` of an `ExternCell` shall be presented in parallel, as well as all the `BusNode` of one side of an `InternCell`.

Therefore, a `VerticalBusSet` gather a set of BusNode that shall be presented in parallel.

It is composed of:

- a set of `BusNodes` that are to be in parallel
- a set of `ExternCell` and a set `InternCellSide` (composition of an InternCell and the `Side` of the leg) that put the verticality constraint.

When the set of `BusNodes` of one `VerticalBusSet` contains all the `BusNodes` of the one of another, then it can absorb it (all the constraints given by the second one are covered by the one of the first one). The absorption consists in merging the sets of cells.

`VerticalBusSet` has a utility method `createVerticalBusSets` that creates a `List<VerticalBusSet>` going through the cells of a `VoltageLevelGraph` (on which the cells detection was already done!) and ensuring that all the possible absorptions are performed.

HorizontalBusList

Definition

`BSCluster` (`BusNode` Sets Cluster) is used by implementations of `PositionFinder`.

It is composed of two kinds of `BusNode` Sets that present a horizontal and a vertical view of the structure of a `VoltageLevel`:

- `List<VerticalBusSet> verticalBusSets` see *VerticalBusSet*
- `List<HorizontalBusList> horizontalBusLists` see *HorizontalBusList*

  Important: `VerticalBusSet` is a `Set` as it is important to have no duplicate `BusNodes`. On the opposite, it is possible to have duplicate `BusNodes` in the `HorizontalBusList`.

  The rules are as follows:

  - for `VerticalBusSet`, a `BusNode` may appear:
    - in several `VerticalBusSet`,
    - but only once in a `VerticalBusSet`;
  - for `HorizontalBusList`, a `BusNode` may appear:
    - several times in a `HorizontalBusList`, in that case the occurrences shall have contiguous indexes,
    - only in one single `HorizontalBusList`.

The goal is to merge all the `BSCluster` in the `VoltageLevel` into a single `BSCluster` having this kind of pattern:

Key methods

Build

A `BSCluster` is initiated with one `VerticalBusSet` that:

- is put as the first element of the `verticalBusSets` list,
- initiates one `HorizontalBusList` for each of its `BusNode`

Each `PositionFinder` using `BSCluster` implementation provides a strategy to merge them to get a single `BSCluster`.

Merge

The merge of two `BSCluster` is done by calling the `merge` method, thanks to the `HorizontalBusListsMerger` given. Indeed, if the merging of the `verticalBusSets` is just a concatenation, the `horizontalBusLists` merging differs from one `PositionFinder` to another.

PositionFromExtension: Position from an explicit configuration

Context

`PositionFromExtension` implements *PositionFinder* and takes the information from the `iidm` extension to organize `BSClusters`. When building the `VoltageLevelGraph`, `NetworkGraphBuilder` retrieves and sets:

- `BusNode.busbarIndex` and `BusNode.sectionIndex`,
- `FeederNode.order` which is used to define `ExternCells::getOrder`

Algorithm

Principle

The algorithm considers that the positional information given (by the **iidm** extension) is coherent. Sorting orders are defined based on this information and are used when building `VerticalBusSets` and `BSClusters`. These objects can then be arranged relying on the consistency inherited from the given coherent orders.

The algorithm may not sound straightforward, but that approach eases the elaboration of the next step (building of `List<Substation>`), and naturally addresses the constraints raised by non horizontally symmetrical arrangements of `BusNodes` (case developed in the example below).

Steps

- `PositionFromExtension::indexBusPosition` builds the `Map<BusNode, Integer> busToNb`
- `PositionFromExtension::organizeBusSets`:
  - builds `List<BSCluster>bsClusters` by providing the list of initial `VerticalBusSet` sorted by the comparator `VBSCOMPARATOR`
  - merges `bsClusters` sequentially by merging the first `BSCluster` of the list with the second until only one remain.

Sorting the `BusNodes` of the `VerticalBusSets`

This sorting order helps `busbarIndex` reflect the **vertical view** of the structure.

The algorithm relies on the fact that `PositionFromExtension::indexBusPosition` sorts the `BusNodes` by their `busbarIndex` (first, and then by their `sectionIndex`). In `PositionFromExtension`, this order will never be modified later on, ensuring the lower the `busbarIndex` is, the higher it will be vertically laid out.

For example,

- if `BusNode` **A** has a lower `busBarIndex` than the one of **B**,
- **A** will be before **B** in a `VerticalBusSet`,
- **A** will be laid out above **B**.

Sorting the bsClusters

This sorting order helps sectionIndex reflect the **horizontal view** of the structure.

VBSCOMPARATOR is designed to sort the given List<VerticalBusSet>verticalBusSet using the positional information given by the extension. It sorts:

- **First and most important sorting rule**: compare sectionIndex (which corresponds to horizontal order) of BusNodes having the same busbarIndex. i.e.:
  - Match a pair of BusNodes having the same busbarIndex, taken in:
    - the right side of the first BSCluster
    - the left side of the second BSCluster.
  - If they have different BusNode.sectionIndex, **return** the difference
  - if they are equal, then try again with another pair.
- **Else**, compare BusCell::getOrder of BusCells in both VerticalBusSets
- **Else**, some rules, to ensure uniqueness.

Merging

The merging process consists in repeating the merging of the first BSCluster with the second one until only one remains.

HorizontalBusListsMerger::mergeHbl merges the HorizontalBusLists from the 2 BSClusters by concatenating them when they have the same busbarIndex. As the BSClusters is sorted by busbarIndex, there is a coherent progression in each merged HorizontalBusList.

Example

The end in mind

The picture hereafter shows what we expect to display. The complete information from the extension is given.

(h,v) positions of BusNodes and ExternCell cells order

Input information

The raw graph looks like:

For which the information is represented as follows:

raw Bus id	(busbarIndex, sectionIndex)
B1	(2, 2)
B2	(2, 1)
B3	(1, 2)
B4	(1, 3)
B5	(1, 1)

And the `ExternalCell order` of **ECx** is **x**.

Steps

Step 1: Build of `VerticalBusSets`

First we define the `Map<BusNode, Integer> busToNb` according to the sorting order `(busbarIndex, sectionIndex)`: **vertical sorting**.

raw Bus id	Nb	(busbarIndex, sectionIndex)
B5	1	(1, 1)
B3	2	(1, 2)
B4	3	(1, 3)
B2	4	(2, 1)
B1	5	(2, 2)

Now `VerticalBusSet.createVerticalBusSets` will create the `VerticalBusSet` and they will be sort it according to `VBSCOMPARATOR`: **horizontal sorting**.

vbs	BusNodes(busBarIndex, sectionIndex)	ExternCells	InternCellSides
vbs-1	[B5(1, 1)]		[IC1.R, IC2.L]
vbs-2	[B2(2, 1)]		[IC1.L]
vbs-3	[B3(1, 2), B1(2, 2)]	[EC1]	[IC2.R, IC3.L]
vbs-4	[B4(1, 3), B1(2, 2)]	[EC2, EC3, EC4]	[IC3.R]

Note the position of **vbs-2**, as **B2(2,1)** shall be before **B1(2,2)**.

Step 2: Build of unitary `BSClusters`

This consist in creating one `BSCluster` per `VerticalBusSet`. This results in:

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-1	[([B5] , , [IC1.R , IC2.L])]	[[B5(1, 1)]]
bsc-2	[([B2] , , [IC1.L])]	[[B2(2, 1)]]
bsc-3	[([B3, B1] , [EC1] , [IC2.R, IC3.L])]	[[B3(1, 2)] , [B1(2, 2)]]
bsc-4	[([B1, B4] , [EC2, EC3, EC4] , [IC3.R])]	[[B1(2, 2)], [B4(1, 3)]]

Step 3: Merge of `BSClusters` into a single one

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-12= bsc-1 + **bsc-2**	[([B5] , , [IC1.R , IC2.L]),(**[B2] , , [IC1.L])**]	[[B5(1,1)],**[B2(2,1)]**]
bsc-123= bsc-12 + **bsc-3**	[([B5] , , [IC1.R , IC2.L]),([B2] , , [IC1.L]),(**[B3, B1] , [EC1] , [IC2.R, IC3.L])**]	[[B5(1,1), **B5(1,1), B3(1,2)**],[B2(2,1), **B1(2,2)**]]
Resulting BSClusterbsc-1234= bsc-123 + **bsc-4**	[([B5] , , [IC1.R , IC2.L]),([B2] , , [IC1.L]),([B3, B1] , [EC1] , [IC2.R, IC3.L]),(**[B1, B4] , [EC2, EC3, EC4] , [IC3.R])**]	[[B5(1,1), B5(1,1), B3(1,2), **B4(1,3)**],[B2(2,1), B1(2,2), **B1(2,2)**]]

This results in:

Note – On the merge of bsc-1 + bsc-2:

- **bsc-1** and **bsc-2** have only one `NodeBus` in their `VerticalBusSet`. The parallelization of both will be handled by `Subsection::createSubsections` by an absorption mechanism.
- in its `HorizontalBusList`, **B5** is replicated until a change occurs, but this replication has no impact on the `VerticalBusSet` (ie **[B2]** is not extended to **[B5, B2]**).
- `HorizontalBusList` has a `startingIndex` which implies it does not necessarily align on the left side. That's the case of the second one for which the `startingIndex` is 2.

Step 4: Build of the `List<Subsection>subsections`

Done by calling `Subsection::createSubsections`. See *Subsection*.

PositionByClustering: Establishing positions driven by similarities

Context

The goal of the algorithm is to find an organization of the `VoltageLevel` with no other information than the graph itself.

Algorithm

Principle

The positioning is based on sequential merges of `BSCluster` considering:

- the fact that all external cells and any "leg" of an `InternCell` shall be **stackable** - meaning, that all the corresponding busbars are aligned in parallel to be able to connect them with a vertical string of isolators. This implies the `busNodes` of a leg shall be spread in different vertical structural positions. The initialization of the `VerticalBusSets` reflects this constraint.
- attractivity, which is expressed in terms of strength of a `Link` between sides (left/right) of `BSCluster`. The stronger the link, the more likely it is to be subject to a merge.

The name of the implementation comes from the fact that `BSClusters` are absorbing one another growing clusters to a single one.

Steps

- PositionByClustering::indexBusPosition builds the Map<BusNode, Integer> busToNb. (The sorting order (BusNode::getId) only goal is to avoid randomness.)
- PositionByClustering::organizeBusSets:
  - builds List<BSCluster>bsClusters
  - calls Links::create, which creates:
    - bsClusterSides with all the initial BSClusterSide (ie RIGHT and LEFT for each BSCluster)
    - linkSet with all the feasible Link between them, sorted by the "strength" of their similarities
  - successively:
    - merges the BSClusterSide of the first (and strongest) Link,
    - updates bsClusterSides list and linkSet. (Each merge reducing their size.)
    - until linkSet is empty (and bsClusterSides reduced to both sides of a single encompassing BSCluster)
  - calls tetrisHorizontalBusLists which arranges the HorizontalBusLists to reduce their number, and consequently the number of necessary busbarIndex merging them when they don't overlap.
  - transfers the resulting structure into BusNode.busbarIndex and BusNode.sectionIndex, and ExternCell.order.
  - the remaining BSCluster is ready to be processed by Subsection class.

Dedicated classes

The BSClusterSide class

The BSClusterSide is a composition of a BSCluster and a Side (LEFT/RIGHT).

The BSClusterSide class provides methods to evaluate its contribution to the strength of a Link (see below).

It is necessary to make the distinction between the sides of BSCluster when looking for similarities for merging. Indeed, once a BSCluster grows and has more than one VerticalBusSet, the attractivity characteristics of both sides are differing. For example, one BusNode could be right at the edge of the LEFT side in a HorizontalBusList that has other BusNodes (on its right). For linking with COMMONBUSES, the BusNode will be exposed only on the LEFT side.

The Link class

The key principle of the algorithm relies on the fact that the Link class implements Comparable, comparing the strength of each considered Link. This is established by a lexicographic comparison of grades sorted per Category.

The enum Category defines a lexicographic order of the considered similarities kinds between 2 BSClusterSides:

- COMMONBUSES
- FLATCELLS
- CROSSOVER
- SHUNT

The Link class holds:

- 2 BSClusterSide,
- a map that holds the similarity grade per Category.

Therefore, the comparison between 2 Link is as follows:

- the more buses there is in common between the 2 BSClusterSides the greater the similarity is.
- If equal, then compare flat cells (see note below) as follows:
  - 100 * number of common candidate flat cells.
  - minored by the sum of the distances between each flat cell and the edge of each BSClusterSide (more details in the note below)
- if equal, then the number of InternCell that will never become flat (and will have to cross over the layout) is compared
- if equal, then compare the attractivity of ShuntCells in common (i.e., having one ExternCell in each BSClusterSide and for which attractivity is assessed considering their distance to the considered side edge).

  Note – About flat cells:

  - At that stage, they are only considered potential flat cell (their type is CANDIDATEFLATCELL). They will have to be confirmed as FLATCELL later in Subsection,
  - The flat cells are identified when both legs are at the appropriate side of the 2 considered BSClusterSide, which means they are:
    - at the beginning of a HorizontalBusSet if the BSClusterSide is LEFT
    - at the end if it is RIGHT
  - For a leg, respecting this criteria does not imply the distance to the edge is 0. For example, in the case of the LEFT side of the BSClusterSide, the BusNode involved in the cell shall be at the beginning of its HorizontalBusSet, but this does not imply the starting index of the HorizontalBusSet to be 0. The distance to the edge is this starting index.
  - The value **100** that multiplies the number of flat cells is arbitrary. It must be big enough so that the number of flat cells is fostered over the penalty of the distances to the edges. By doing so, the distance minoration discriminates only when the numbers of flat cells are the same.

  Important – Link and side:

  - No Link shall be created between both BSClusterSide of the same BSCluster as it is not possible to merge them.
  - Two BSClusterSide having the same side can have a Link. If they are to be merged, one will be flipped.

The Links class

The Links class holds and manages:

- bsClusterSides: a list of all the BSClusterSide
- a TreeSet<Link>linkSet: which stores all the Link feasible between the bsClusterSides elements. This linkSet is sorted according to the Comparable implemented by Link.

When a Link is selected for a merge of its BSClusterSide, all the Link involving any of both BSClusterSide are destroyed. New Link are created between both (left/right) BSClusterSide created with the newly merged BSCluster

and each element of `bsClusterSides`. Therefore, `linkSet` always contains an updated sorted set of all the feasible `Link`.

Example

Input information

The raw graph looks:

Steps

Step 1: Build of `VerticalBusSets`

Contrary to `PositionFromExtension` no order is necessary, let's arbitrarily use the suffix in the name of the `BusNode`.

vbs	BusNodes(busBarIndex, sectionIndex)	ExternCells	InternCellSides
vbs-1	[B1(2, 2), B3(1, 2)]	[EC1]	[IC2.R, IC3.L]
vbs-2	[B1(2, 2), B4(1, 3)]	[EC2, EC3, EC4]	[IC3.R]
vbs-3	[B2(2, 1)]		[IC1.L]
vbs-4	[B5(1, 1)]		[IC1.R, IC2.L]

Step 2.1: Build of unitary `BSCluster`

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-1	[([B1, B3] , [EC1] , [IC2.R, IC3.L])]	[[B1(2, 2)] , [B3(1, 2)]]
bsc-2	[([B1, B4] , [EC2, EC3, EC4] , [IC3.R])]	[[B1(2, 2)], [B4(1, 3)]]
bsc-3	[([B2] , , [IC1.L])]	[[B2(2, 1)]]
bsc-4	[([B5] , , [IC1.R , IC2.L])]	[[B5(1, 1)]]

Step 2.2: Build the list of `BSClusterSide` and link them

Here is a list of all the `Link` that are created with the original list of `BSCluster`, and the grade assessed per `Category`.

BSClusterSide1	BSClusterSide2	Common Buses	Flat cells	Crossover	Shunt
bsc-1.L	bsc-2.L	[B1]-> **1**	[IC3]-> 100*1 = **100**	0	0
bsc-1.L	bsc-2.R	[B1]-> **1**	[IC3]-> 100*1 = **100**	0	0
bsc-1.R	bsc-2.L	[B1]-> **1**	[IC3]-> 100*1 = **100**	0	0
bsc-1.R	bsc-2.R	[B1]-> **1**	[IC3]-> 100*1 = **100**	0	0
bsc-1.L	bsc-3.L	**0**	**0**	0	0
bsc-1.L	bsc-3.R	**0**	**0**	0	0
bsc-1.R	bsc-3.L	**0**	**0**	0	0
bsc-1.R	bsc-3.R	**0**	**0**	0	0
bsc-1.L	bsc-4.L	**0**	[IC2]-> 100*1 = **100**	0	0
bsc-1.L	bsc-4.R	**0**	[IC2]-> 100*1 = **100**	0	0
bsc-1.R	bsc-4.L	**0**	[IC2]-> 100*1 = **100**	0	0
bsc-1.R	bsc-4.R	**0**	[IC2]-> 100*1 = **100**	0	0
bsc-2.L	bsc-3.L	**0**	**0**	0	0
bsc-2.L	bsc-3.R	**0**	**0**	0	0
bsc-2.R	bsc-3.L	**0**	**0**	0	0
bsc-2.R	bsc-3.R	**0**	**0**	0	0
bsc-2.L	bsc-4.L	**0**	**0**	0	0
bsc-2.L	bsc-4.R	**0**	**0**	0	0
bsc-2.R	bsc-4.L	**0**	**0**	0	0
bsc-2.R	bsc-4.R	**0**	**0**	0	0
bsc-3.L	bsc-4.L	**0**	[IC1]-> 100*1 = **100**	0	0
bsc-3.L	bsc-4.R	**0**	[IC1]-> 100*1 = **100**	0	0
bsc-3.R	bsc-4.L	**0**	[IC1]-> 100*1 = **100**	0	0
bsc-3.R	bsc-4.R	**0**	[IC1]-> 100*1 = **100**	0	0

Let's remove the unnecessary fields for the example (Crossover and Shunt) and sort the table according to the `Link` order.

BSClusterSide1	BSClusterSide2	Common Buses	Flat cells
bsc-1.L	bsc-2.L	[B1]-> **1**	[IC3]-> 100*1 = **100**
bsc-1.L	bsc-2.R	[B1]-> **1**	[IC3]-> 100*1 = **100**
bsc-1.R	bsc-2.L	[B1]-> **1**	[IC3]-> 100*1 = **100**
bsc-1.R	bsc-2.R	[B1]-> **1**	[IC3]-> 100*1 = **100**
bsc-3.L	bsc-4.L	**0**	[IC1]-> 100*1 = **100**
bsc-3.L	bsc-4.R	**0**	[IC1]-> 100*1 = **100**
bsc-3.R	bsc-4.L	**0**	[IC1]-> 100*1 = **100**
bsc-3.R	bsc-4.R	**0**	[IC1]-> 100*1 = **100**
bsc-1.L	bsc-4.L	**0**	[IC2]-> 100*1 = **100**
bsc-1.L	bsc-4.R	**0**	[IC2]-> 100*1 = **100**
bsc-1.R	bsc-4.L	**0**	[IC2]-> 100*1 = **100**
bsc-1.R	bsc-4.R	**0**	[IC2]-> 100*1 = **100**
bsc-1.L	bsc-3.L	**0**	**0**
bsc-1.L	bsc-3.R	**0**	**0**
bsc-1.R	bsc-3.L	**0**	**0**
bsc-1.R	bsc-3.R	**0**	**0**
bsc-2.L	bsc-3.L	**0**	**0**
bsc-2.L	bsc-3.R	**0**	**0**
bsc-2.R	bsc-3.L	**0**	**0**
bsc-2.R	bsc-3.R	**0**	**0**
bsc-2.L	bsc-4.L	**0**	**0**
bsc-2.L	bsc-4.R	**0**	**0**
bsc-2.R	bsc-4.L	**0**	**0**
bsc-2.R	bsc-4.R	**0**	**0**

Step 3: Merge of `BSClusters` into a single one

At the first iteration, the `Link` between **bsc-1.L** and **bsc-2.L** is the strongest. Let's merge them into **bsc-12**:

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-12 = bsc1 + bsc2	[([B1, B3] , [EC1] , [IC2.R, IC3.L]),([B1, B4] , [EC2, EC3, EC4] , [IC3.R])]	[[B1(2, 2) , B1(2, 2)],[B3(1, 2), B4(1, 3)]]
bsc-3	[([B2] , , [IC1.L])]	[[B2(2, 1)]]
bsc-4	[([B5] , , [IC1.R , IC2.L])]	[[B5(1, 1)]]

Note - regarding the merge:

- **B1** is common to both, and therefore **B1** spans over 2 indexes in its resulting `HorizontalBusList`
- **IC3** links **B3** and **B4** so they are merged in the same `HorizontalBusList`

Let's update the list of `Link`.

BSClusterSide1	BSClusterSide2	Common Buses	Flat cells
bsc-3.L	bsc-4.L	**0**	[IC1]-> 100*1 = **100**
bsc-3.L	bsc-4.R	**0**	[IC1]-> 100*1 = **100**
bsc-3.R	bsc-4.L	**0**	[IC1]-> 100*1 = **100**
bsc-3.R	bsc-4.R	**0**	[IC1]-> 100*1 = **100**
bsc-12.L	bsc-4.L	**0**	[IC2]-> 100*1 = **100**
bsc-12.L	bsc-4.R	**0**	[IC2]-> 100*1 = **100**
bsc-12.R	bsc-4.L	**0**	[IC2]-> 100*1 = **100**
bsc-12.R	bsc-4.R	**0**	[IC2]-> 100*1 = **100**
bsc-12.L	bsc-3.L	**0**	**0**
bsc-12.L	bsc-3.R	**0**	**0**
bsc-12.R	bsc-3.L	**0**	**0**
bsc-12.R	bsc-3.R	**0**	**0**

Now the strongest link is between **bsc-3.L** and **bsc-4.L**. Let's merge them into **bsc-34**:

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-12 = bsc1 + bsc2	[([B1, B3] , [EC1] , [IC2.R, IC3.L]),([B1, B4] , [EC2, EC3, EC4] , [IC3.R])]	[[B1(2, 2) , B1(2, 2)],[B3(1, 2), B4(1, 3)]]
bsc-34 = bsc3 + bsc4	[([B2] , , [IC1.L]),([B5] , , [IC1.R , IC2.L])]	[[B2(2, 1), B5[1, 1]]]

BSClusterSide1	BSClusterSide2	Common Buses	Flat cells
bsc-12.L	bsc-34.R	**0**	[IC2]-> 100*1 = **100**
bsc-12.R	bsc-34.R	**0**	[IC2]-> 100*1 = **100**
bsc-12.L	bsc-34.L	**0**	**0**
bsc-12.R	bsc-34.L	**0**	**0**

IC2 is what makes the strongest link. It is on the `RIGHT` side of **bsc-34**, and in its `HorizontalBuslList` **B3** is on its left, therefore **IC2** is not visible from the `LEFT` side, which explains why the flat cell grade is 0 for the last 2 `Link`.

Final merge: **bsc-12.L** with **bsc-34.R** that will become **bsc-3412** as the right side of **bs-34** is connected to the left side of **bsc-12**. (Reminder, when merging 2 `BSClusterSide` having the same side, one is to be flipped – which is actually not necessary here.)

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-3412 = bsc34 + bsc12	[([B2] , , [IC1.L]),([B5] , , [IC1.R , IC2.L]),([B1, B3] , [EC1] , [IC2.R, IC3.L]),([B1, B4] , [EC2, EC3, EC4] , [IC3.R])]	[[B2(2, 1), B5[1, 1], B1(2, 2) , B1(2, 2)],[. , . , B3(1, 2), B4(1, 3)]]

Note that the second `HorizontalBusSet` starts with index 2.

No "tetrissing" will be required as the final arrangement is directly correct.

This results in:

Step 4: Build of the List<Subsection>subsections

Done by calling Subsection::createSubsections. See *Subsection*.

Final result

Subsection

Context

A Subsection is a part of the VoltageLevel defined by the BusNodes that must be displayed in parallel. The VoltageLevel is partitioned into Subsections each of them having a different set of BusNodes.

A List<Subsection> will then be given to BlockPositionner that will sequentially treat each Subsection assigning the Positions *(H,V)* of BusNodes and Cells.

Therefore, the List<Subsection> shall consistently respect the following rule: if a BusNode spans over many Subsections, it shall:

- be in contiguous Subsections in the list
- have the same index (ie same vertical position) in each Subsection

Definition

A Subsection has the following attributes:

- size which is the number of parallel BusNodes and is defined in the constructor and cannot be changed.
- busNodes a BusNodes array of size size
- externCells a list of ExternCell
- InternCellSides a Set InternCellSide which correspond to the leg of one InternCell combined with its Side.

  Note - When an InternCell overlaps 2 Subsections:

  - its LEFT leg will be on the right side of the first Subsection, i.e., at the end of internCellSides
  - its RIGHT leg will be on the left side of the second Subsection, i.e., at the beginning of internCellSides

Building the List<Subsection>

Subsection has createSubsections as a utility method to build the List<Subsection> subsections based on a consolidated BSCluster and is called by implementations of PositionFinder.

It calls many methods that aim at ensuring the coherence of the position of the Cells. Especially, that's where

- InternCell.Shape are defined,
- InternCell are flipped (RIGHT / LEFT) if needed to be consistent with the arrangement,
- if handleShunt is set, ExternCells involved in a ShuntCell are shifted to be as close to one another as possible.

Definitions and goal

Positioning the BusNodes and determination of the ExternalCells order are performed by implementing the PositionFinder interface.

The goal is to:

- set the horizontal and vertical **structural** ***(h,v)*** BusNode.structuralPosition
- set the horizontal **order** of the cells: Cell.order
- provide a List<Subsection>. See *Subsection*.

The picture hereafter shows the information that is to be established.

(h,v) positions of BusNodes and ExternCell cells order

Available implementations

Two implementations are available:

- PositionFromExtension which rely on explicitly given positions (for example, to reflect the on-site real structure and/or the way the SCADA organizes it). See *PositionFromExtension*
- PositionByClustering which finds an organization of the VoltageLevel with no other information than the graph itself. See *PositionByClustering*

Both rely on the BSCluster (see *BSCluster*) and have the same skeleton:

- Step 1: Build of VerticalBusSets
- Step 2: Build of unitary BSCluster in the bsClusters list
- Step 3: Merge of bsClsuters into a single BSCluster
- Step 4: Build of the List<Subsection>subsections

Illustration of algorithms based on `BSCluster`

The illustration will be based on the following graph and shall result in the above layout.

Step 1: Build `VerticalBusSets`

The result of `VerticalBusSet.createVerticalBusSets` is

VerticalBusSet	BusNodes	ExternCells	InternCellSides
vbs-1	B3, B1	EC1	IC2.R, IC3.L
vbs-2	B2		IC1.L
vbs-3	B1, B4	EC2, EC3, EC4	IC3.R
vbs-4	B5		IC1.R, IC2.L

Note: At that stage, the `LEFT` and `RIGHT` side of an `InternCell` is arbitrary. They will be flipped if necessary later on (handled in `Subsection.createSubsections`).

Step 2: Build unitary `BSClusters`

This consist in creating one `BSCluster` per `VerticalBusSet`. This results in:

BSCluster	VerticalBusSets	HorizontalBusLists
bsc-1	[([B3, B1] , [EC1] , [IC2.R, IC3.L])]	[[B3] , [B1]]
bsc-2	[([B2] , , [IC1.L])]	[[B2]]
bsc-3	[([B1, B4] , [EC2, EC3, EC4] , [IC3.R])]	[[B1], [B4]]
bsc-4	[([B5] , , [IC1.L , IC2.L])]	[[B5]]

Important - On this result:

- It is representative of the general case. But note that for `PositionFromExtension`, the `verticalBusSets` is sorted to end up to a ready-to-merge `bsClusters`. See *PositionFromExtension*.
- In this picture, the `NodeBus` are on different rows to show that they are not necessarily aligned. Only both `B1` will necessarily be on the same row.

Step 3: Merge `BSClusters` into a single one

That's where the magic happens. This is where the implementations mainly differ. The goal is to merge the `BSClusters` to one another.

The principle of the merging of a `BSCluster` are:

- simply concat the `VerticalBusSet` List
- merge the `HorizontalBusList` using a proper implementation of `HorizontalBusLane::mergeHbl`.

This expected result should be similar to the following `BSCluster`:

VerticalBusSets	HorizontalBusLists
[([B2, B5] , , [IC1.L, IC1.R, IC2.L]) , ([B4, B1] , [EC2, EC3, EC4] , [IC3.R]) , ([B3, B1] , [EC1] , [IC2.R, IC3.L])]	[[B2, B4, B3] , [B5, B1, B1]]

The way this example is handled is detailed in each implementation documentation: *PositionFromExtension*, *PositionByClustering*.

Step 4: Build the `List<Subsection>`

See *Subsection*.

6.3.5 Zone Matrix Layout

In this layout, the substations are displayed like elements of a matrix (rows and columns). The user can choose the location of each substation.

Input parameters

- `VoltageLevelLayoutFactory`: builder of the layout used by voltage levels
- `SubstationLayoutFactory`: builder of the layout used by substations
- `2D String array`: substation matrix position (ex: `{{"A", "B", "C"}}` = 1 row, 3 columns)

Usage example: The following example displays three substations distributed on two columns and two lines, with an empty area at the middle of the second line.

```
// build zone graph
Network network = ...
List<String> zone = Arrays.asList("A", "B", "C");
ZoneGraph g = new NetworkGraphBuilder(network).buildZoneGraph(zone);

// Create substation 2D array representation
String[][] substationsIds = {{"A", "B"},
                             {"D", ""}};
// Create matrix zone layout using 2D array
ZoneLayoutPathFinderFactory pFinderFactory = DijkstraPathFinder::new;
SubstationLayoutFactory sFactory = new HorizontalSubstationLayoutFactory();
```

(continues on next page)

(continued from previous page)

```
VoltageLevelLayoutFactory vFactory = new PositionVoltageLevelLayoutFactory();
MatrixZoneLayoutFactory mFactory = new MatrixZoneLayoutFactory();
Layout matrixLayout = mFactory.create(g, substationsIds, pFinderFactory, sFactory,␣
↪vFactory);

// Apply matrix zone layout
matrixLayout.run(layoutParameters);
```

Path finding description

Premise:

- The column width is computed for each column as the maximum width of all the substations on the column
- The row height is computed for each row as the maximum height of all the substations on the row
- Snakeline lane dimensions (both horizontal and vertical) are set with `LayoutParameters.setZoneLayoutSnakeLinePadding`

Example:

		ZonePaddinç				ZonePaddinç		
		VL TOP Padding				**VL TOP Padding**		
ZonePac	**VL LEFT Padding**	**A**	**VL RIGHT Padding**	**ZonePac**	**VL LEFT Padding**	**B**	**VL RIGHT Padding**	**ZonePadding**
		VL BOT-TOM Padding				**VL BOT-TOM Padding**		
		ZonePadding				**ZonePadding**		
		VL TOP Padding				**VL TOP Padding**		
ZonePac	**VL LEFT Padding**	**D**	**VL RIGHT Padding**	**ZonePac**	**VL LEFT Padding**	**_**	**VL RIGHT Padding**	**ZonePadding**
		VL BOT-TOM Padding				**VL BOT-TOM Padding**		
		ZonePadding				**ZonePadding**		

The class `MatrixZoneLayout` represents the matrix layout. The class `MatrixZoneLayoutModel` represents the matrix and the path finder information. The class `Matrix` contains an array of `MatrixCell`.

The class `MatrixCell` stores information related to the matrix cell:

- Position (indices) in the matrix: row, column
- The id of the substation graph contained by the cell

Substation positioning

1) In the `MatrixZoneLayout` constructor, each `SubstationGraph` is added to the `MatrixZoneLayoutModel` (internal model of matrix layout) as following:

```
for (int row = 0; row < matrix.length; row++) {
    for (int col = 0; col < matrix[row].length; col++) {
        String id = matrix[row][col];
        SubstationGraph sGraph = graph.getSubstationGraph(id);
        if (sGraph == null && !id.isEmpty()) {
            throw new PowsyblException("Substation '" + id + "' was not found in zone␣
↪graph '" + getGraph().getId() + "'");
        }
        model.addSubstationGraph(sGraph, row, col);
    }
}
    ...
```

2) Each substation position is computed using this method:

```
protected void calculateCoordSubstations(LayoutParameters layoutParameters) {
```

The `SubstationLayout` is applied on each not empty substation specified as following:

```
    // Display substations on not empty Matrix cell
    matrix.stream().filter(c -> !c.isEmpty()).map(MatrixCell::graph).forEach(graph ->␣
↪layoutBySubstation.get(graph).run(layoutParameters));
```

Each substation is moved into its matrix position as specified

Snakeline route computation between substations

The `Grid` class contains a 2D-array of `Node`, each `Node` representing a pixel of the SLD output file. Each `Node` stores :

- A position (x and y)
- Availability (whether the `Node` can be used to draw the snakeline or not)
- A walk-through cost
- A parent node reference
- The distance to the end point of the snakeline

Exclusion area

An exclusion area is an area where all `Node` have availability equals to `false` This area cannot be used to draw a `snakeline`. Those areas are:

- diagram padding
- voltage levels with padding
- snakelines right angles

Shorter path computation

Dijkstra's computation steps:

- The starting point cost is set to 0
- The nearest neighbors (left, right, up and down) are computed (no diagonal moves allowed here)
  - These neighbors are used only if:
    - The neighbor is available
    - The neighbor was not yet visited
  - To avoid superfluous right angles, the cost is increased when the next point creates a right angle
- If no route can be computed by the algorithm, a straight line is drawn from the starting point to the end point of the snakeline (diagonal moves are allowed here)

A layout represents the way in which the elements of a graph are arranged.

It is possible to use your own graph layout implementation, but there are also existing layouts in powsybl-diagram, ready to use.

6.3.6 Layouts for voltage levels

Existing implementations

PositionVoltageLevelLayout

This layout positions the different elements inside a voltage level according to the following process:

- Clean the graph to have the expected patterns (see *Graph Refiner*)
- Detect the cells (intern / extern / shunt) (see *Cell detection*)
- Organize the cells into blocks (see *CellBlockDecomposer*)
- Compute real coordinates of busNode and blocks connected to busbars (see *PositionFinder*)

The `PositionVoltageLevelLayoutFactoryParameters` class

The `PositionFinder` class

RandomVoltageLevelLayout

With this layout, the graph node coordinates are randomly fixed:

- Between 0 and `width` for the x coordinate;
- Between 0 and `height` for the y coordinate. The `width` and `height` variables are provided by the user.

CgmesVoltageLevelLayout

With this layout, the elements of the graph are arranged according to the data included in the CGMES DL profile.

Choosing a `VoltageLevelLayout`

The `voltageLevelLayoutFactoryCreator` attribute in the `SldParameters` class is the customization parameter to use to choose a specific `VoltageLevelLayout`.

The `VoltageLevelLayoutFactoryCreator` creates a `VoltageLevelLayoutFactory` which in turn creates a `VoltageLevelLayout`.

Choose a specific PositionVoltageLevelLayout

Static methods are available in the `VoltageLevelLayoutFactoryCreator` interface to help users manipulate those objects.

Some examples are shown below.

Example 1

PositionVoltageLevelLayout using default parameters:

```
VoltageLevelLayoutFactoryCreator voltageLevelLayoutFactoryCreator =␣
↪VoltageLevelLayoutFactoryCreator.newPositionVoltageLevelLayoutFactoryCreator();
SldParameters sldParameters = new SldParameters().
↪setVoltageLevelLayoutFactoryCreator(voltageLevelLayoutFactoryCreator);
```

Example 2

PositionVoltageLevelLayout with a chosen *PositionFinder*:

```
VoltageLevelLayoutFactoryCreator voltageLevelLayoutFactoryCreator =␣
↪VoltageLevelLayoutFactoryCreator.
↪newPositionVoltageLevelLayoutFactoryCreator(positionFinder);
SldParameters sldParameters = new SldParameters().
↪setVoltageLevelLayoutFactoryCreator(voltageLevelLayoutFactoryCreator);
```

Example 3

PositionVoltageLevelLayout with chosen *PositionVoltageLevelLayoutFactoryParameters*:

```
VoltageLevelLayoutFactoryCreator voltageLevelLayoutFactoryCreator =␣
↪VoltageLevelLayoutFactoryCreator.
↪newPositionVoltageLevelLayoutFactoryCreator(positionVoltageLevelLayoutFactoryParameters);
↪
SldParameters sldParameters = new SldParameters().
↪setVoltageLevelLayoutFactoryCreator(voltageLevelLayoutFactoryCreator);
```

Example 4

PositionVoltageLevelLayout with a chosen *PositionFinder* and chosen *PositionVoltageLevelLayoutFactoryParameters*:

```
VoltageLevelLayoutFactoryCreator voltageLevelLayoutFactoryCreator =␣
↪VoltageLevelLayoutFactoryCreator.
```

(continues on next page)

(continued from previous page)

```
↪newPositionVoltageLevelLayoutFactoryCreator(positionFinder,␣
↪positionVoltageLevelLayoutFactoryParameters);
SldParameters sldParameters = new SldParameters().
↪setVoltageLevelLayoutFactoryCreator(voltageLevelLayoutFactoryCreator);
```

Use the `SmartVoltageLevelLayoutFactory`

The SmartVoltageLevelLayoutFactory picks the "best" `VoltageLevelLayout` according to the information available in the network.

There is also a static method in the `VoltageLevelLayoutFactoryCreator` interface to help users pick the `SmartVoltageLevelLayoutFactory`:

```
VoltageLevelLayoutFactoryCreator voltageLevelLayoutFactoryCreator =␣
↪VoltageLevelLayoutFactoryCreator.newSmartVoltageLevelLayoutFactoryCreator();
SldParameters sldParameters = new SldParameters().
↪setVoltageLevelLayoutFactoryCreator(voltageLevelLayoutFactoryCreator);
```

6.3.7 Layouts for substations

TODO

6.3.8 Layouts for multi-substation graphs

Multi-substation graphs can use a matrix-like layout for the different substations. See *Zone Matrix Layout* for more details.

6.4 Style provider

The `StyleProvider` interface provides a way to customize the appearance of the single-area diagram.

Currently, there are 2 implementations of the `StyleProvider`: the `BasicStyleProvider`, the `NominalVoltageStyleProvider` and the `TopologicalStyleProvider`

6.4.1 Common features

The common features are factorized in the abstract classes `AbstractStyleProvider` and `AbstractVoltageStyleProvider`.

6.4.2 TopologicalStyleProvider feature

The bus nodes of a voltage level are each marked with a class depending on the nominal voltage and on their bus index. This leads to a colour shading for each range of nominal voltages defined by the BaseVoltagesConfig.

The `TopologicalStyleProvider` also supports highlighting of electrical buses on hover. In order to enable this feature, you can customize your `SldParameters` by using the `BusHighlightStyleProviderFactory`:

```
SldParameters sldParameters = new SldParameters().setStyleProviderFactory(new␣
↪BusHighlightStyleProviderFactory());
```

If you don't want to use the provided `BusHighlightStyleProviderFactory` in your `SldParameters`, you could also create your own `StyleProviderFactory` and include the `TopologicalStyleProvider` with the `busHighlightOnHover` parameter set to `true` in the `StyleProvidersList`:

```
public class YourCustomStyleProviderFactory implements StyleProviderFactory {
    @Override
    public StyleProvider create(Network network, SvgParameters svgParameters) {
        return new StyleProvidersList(new TopologicalStyleProvider(network,␣
↪svgParameters, true), ...);
    }
}
```

The `powsybl-single-line-diagram-core` artifact provides features to generate customized single-line diagrams:

- Creation of single-line diagrams for given voltage levels, substations or zones in SVG format, for both node/breaker and bus/breaker topologies:
  - From an IIDM network: a graph is built from the input network and then written as a single-line diagram;
  - By directly providing the underlying graph to the writer.
- Diagram customization:
  - Several layout algorithms to generate the diagrams;
  - Many layout parameters to adjust the rendering;
  - Possible use of your own component library. Modification of the existing library is also an option;

Some extensions are also available. You may check the powsybl-single-line-diagram-cgmes-iidm-extensions artifact to force positions in the diagram, for instance.

6.5 Examples

These examples show how to write a single-line diagram into an SVG file.

- Generate a single-line diagram for the voltage level `N` of the network `network`

```
SingleLineDiagram.draw(network, "N", "/tmp/n.svg");
```

- Generate a single-line diagram for the substation `A` of the network `network`, with customized `SvgParameters` (See *SvgParameters documentation page*).

```
SldParameters sldParameters = new SldParameters().setSvgParameters(new SvgParameters().
↪setUseName(true));
SingleLineDiagram.draw(network, "A", Path.of("/tmp/a.svg"), sldParameters);
```

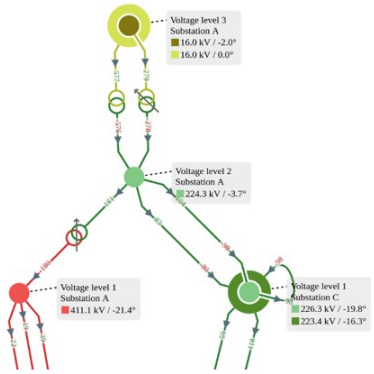
CHAPTER

SEVEN

WHAT IS POWSYBL-DIAGRAM?

PowSyBl diagram allows users to generate customizable network graph diagrams (denominated network-area diagrams in PowSyBl) and single-line diagrams in SVG format files.

Network-Area Diagram



Single-Line Diagram