The "diagraph" package

v0.3.3 MIT

Draw graphs with Graphviz. Use mathematical formulas as labels.

Robotechnic @Robotechnic
Malo @MDLC01

Table of Contents

Part I

Drawing graphs

To draw a graph you have two options: `#raw-render` and `#render`. `#raw-render` is a wrapper around the `#render` and allow to use raw block as input.

```
#render(
  ⟨dot⟩,
  ⟨labels⟩: (:),
  ⟨xlabels⟩: (:),
  ⟨edges⟩: (:),
  ⟨clusters⟩: (:),
  ⟨engine⟩: "dot",
  ⟨width⟩: auto,
  ⟨height⟩: auto,
  ⟨clip⟩: true,
  ⟨debug⟩: false,
  ⟨background⟩: none,
  ⟨stretch⟩: true,
  ⟨math-mode⟩: auto
)
```

Argument

`⟨dot⟩` `str`

The dot code to render.

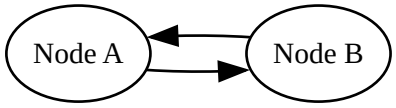
Argument

`⟨labels⟩` `dictionary` | `function`

Nodes labels to overwrite.

If the provided argument is a dictionary, the keys are the node names and the values are the labels. If the provided argument is a function, the function is called with the node name as argument and must return the label. A none value means that the label is not overwritten.

````
#raw-render(```
  digraph G {
    rankdir=LR
    A -> B
    B -> A
  }
```,
labels: ("A": "Node A", "B": "Node B")
)
````


````

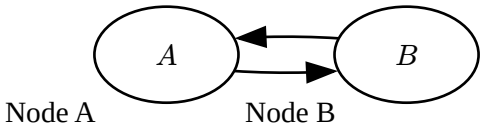
Argument

⟨xlabels⟩ `dictionary` | `function`

Nodes xlabels to overwrite.

If the provided argument is a dictionary, the keys are the node names and the values are the xlabels. If the provided argument is a function, the function is called with the node name as argument and must return the xlabel. A none value means that the xlabel is not overwritten.

```
#raw-render(```
  digraph G {
    rankdir=LR
    A -> B
    B -> A
  }
```,
xlabels: ("A": "Node A", "B": "Node B")
)
```




Argument

⟨edges⟩ `dictionary` | `function`

Edges labels to overwrite.

If the parameter is a dictionary, the dictionary keys are source nodes. The values are dictionaries indexed by the target node. Each edge is one dictionary of valid keys or a list designating multiple edges between the same nodes.

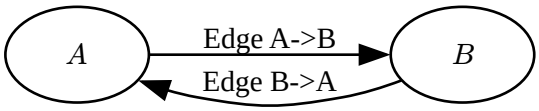
If the parameter is a function, the function is called with two parameters:
- The source node
- A list of the nodes with an edge from the source node to them

It must return a dictionary with the target nodes as keys and a dictionary or list of dictionary as values.

Valid keys for the edges dictionary are:
- `label`: The label of the edge.
- `xlabel`: The xlabel of the edge.
- `taillabel`: The taillabel of the edge.
- `headlabel`: The headlabel of the edge.

Instead of a dictionary, you can only specify a content value. In this case, the content is used as the label of the edge.
```

```
#raw-render(
  ```
    digraph G {
      rankdir=LR
      A -> B
      B -> A
    }
  ```,
  edges: (
    "A": ("B": "Edge A->B"),
    "B": ("A": "Edge B->A"),
  ),
)
```



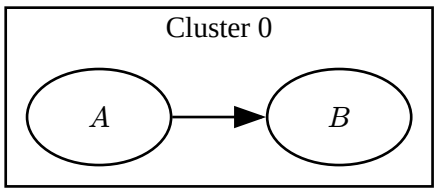
Argument

⟨clusters⟩ `dictionary`

Clusters labels to overwrite.

If the provided argument is a dictionary, the keys are the cluster names and the values are the labels. If the provided argument is a function, the function is called with the cluster name as argument and must return the label. A none value means that the label is not overwritten.

```
  #raw-render(```
    digraph G {
      rankdir=LR
      subgraph cluster_0 {
        A -> B
      }
    }
  ```,
  clusters: ("cluster_0": "Cluster 0")
)
```


```

Argument

⟨engine⟩ `str`

The engine to use to render the graph. The currently supported engines are:

- circo
- dot
- fdp
- neato
- nop
- nop1
- nop2
- osage
- patchwork
- sfdp
- twopi

Argument

⟨width⟩ `length` | `ratio`

The width of the rendered image.

Argument

⟨height⟩ `length` | `ratio`

The height of the rendered image.

Argument

⟨clip⟩ `bool`

Whether to hide part of the graphs that goes outside the bounding box given by graphviz.

Argument

⟨debug⟩ `bool`

Show debug boxes around the labels.

Argument

⟨background⟩ `none` | `color`

The background color of the rendered image. `none` means transparent.

Argument

⟨stretch⟩ `bool`

By default, if both with and height are set, the graph will be stretched to fit in the bounding box. If you want to keep the aspect ratio, you can set this parameter to false. It then behaves like the css property `object-fit: contain`.

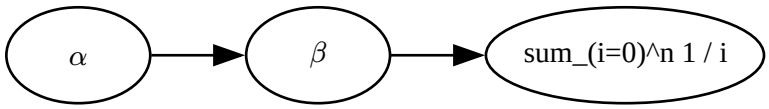
Argument

⟨math-mode⟩ `auto` | `math` | `text`

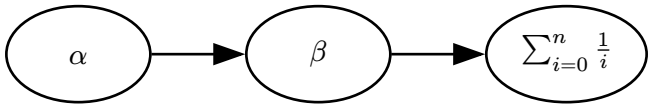
The math mode to use for the labels. Can be `auto`, `"math"` or `"text"`. If set to `auto`, the mode will be determined by label content. In `"text"` mode, the label content will be parsed as a string. In `"math"` mode, the label content will be parsed as a math expression.

This parameter acts globally on the graph. You can also set the math mode more precisely for each node, edge and cluster using the appropriate dot attribute (See Section II)

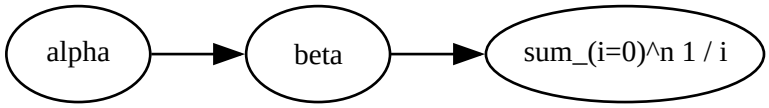
```
#raw-render(```
  digraph G {
    rankdir=LR
    alpha -> beta
    beta -> "sum_(i=0)^n 1 / i"
  }
```,
math-mode: auto
)
```




```
#raw-render(```
  digraph G {
    rankdir=LR
    alpha -> beta
    beta -> "sum_(i=0)^n 1 / i"
  }
```,
math-mode: "math"
)
```




```
#raw-render(```
  digraph G {
    rankdir=LR
    alpha -> beta
    beta -> "sum_(i=0)^n 1 / i"
  }
```,
math-mode: "text"
)
```


```

Part II

Math mode

This module uses a custom graphviz parameter to control how the math mode is managed by the renderer. By default, a simple algorithm convert automatically all text to math mode. You can control this behavior for each node, edge and cluster by using a parameter to enable full math mode or disable it completely.

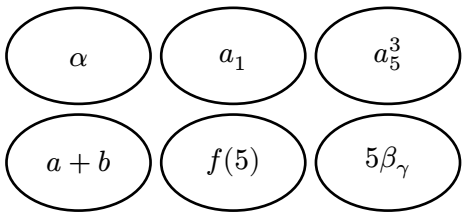
Object	Label	Math mode
Node	label	math
	xlabel	xmath
Edge	label	lmath
	xlabel	lxmath
	headlabel	hmath
	taillabel	tmath
Cluster	label	math

II.1 Default behavior

By default, the algorithm detect the patterns that are likely to be math expressions. The following patterns are detected:

- Single letters
- Numbers
- Greek letters
- All of the above separated by a non letter or number character: an underscore, a space, a parenthesis, etc.

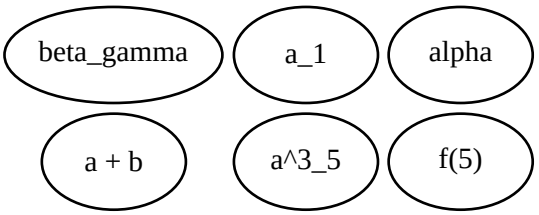
```
#raw-render(```
  graph {
    alpha
    a_1
    "a^3_5"
    "a + b"
    "f(5)"
    "5 beta_gamma'"
  }
```, engine: "osage")
```


```

II.2 No math mode

This mode diable the math mode for the node, edge or cluster. The text is rendered as is.

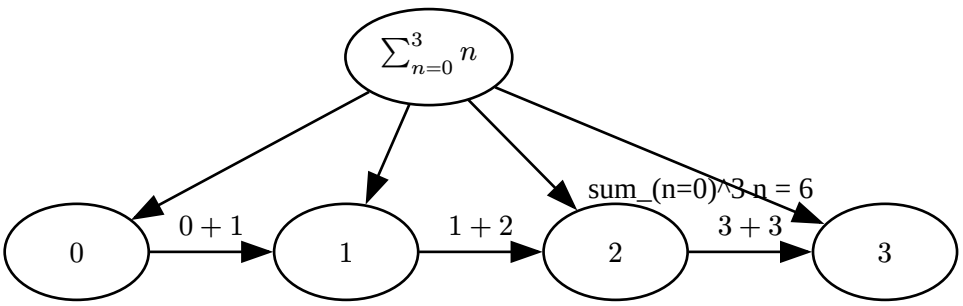
```
#raw-render(```
  digraph G {
    rankdir=LR
    node[math=false]
    alpha
    a_1
    "a^3_5"
    "a + b"
    "f(5)"
    "beta_gamma"
  }
```, engine: "osage")
```




## II.3 Full math mode

In this mode, all the text is considered as a typst math expression. So, any invalid math expression will cause an error.

```
#raw-render(```
  digraph {
    node[math=true]
    edge[lmath=true]
    s[label="sum_(n=0)^3 n"]
    s -> s1
    s -> s2
    s -> s3
    s -> s4
    {rank=same
    s1[label="0"]
    s2[label="1"]
    s3[label="2"]
    s4[label="3"]
    s1 -> s2[label="0 + 1"]
    s2 -> s3[label="1 + 2"]
    s3 -> s4[label="3 + 3"]
    }
    s4[xlabel="sum_(n=0)^3 n = 6"]
  }
```)
```



Part III

Adjacency list

If you don’t want to write the graph in dot format, you can use the adjacency list format. This format allow you to pass a matrix of edges values that represent the graph.

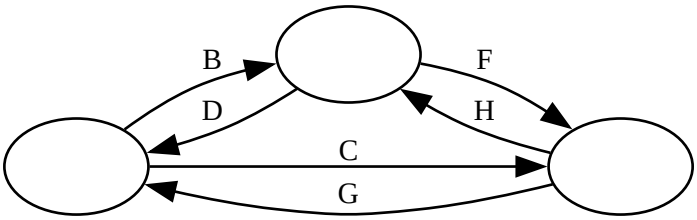
```
#adjacency(
  ⟨matrix⟩,
  ⟨directed⟩: true,
  ⟨vertex-labels⟩: (),
  ⟨clusters⟩: (),
  ⟨debug⟩: false,
  ⟨clip⟩: true,
  ..⟨dot arguments⟩
)
```

Argument

⟨matrix⟩ `array`

The matrix of edges. The matrix must be a list of lists. Each list represents a row of the matrix. The values of the matrix are the labels of the edges. If the value is none, there is no edge between the nodes.

```
#adjacency(
  (
    (none, "B", "C"),
    ("D", none, "F"),
    ("G", "H", none),
  ),
  rankdir: "LR"
)
```

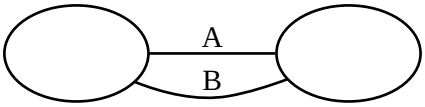


Argument

⟨directed⟩ `bool`

Whether the graph is directed or not.

```
#adjacency(
  (
    (none, "A"),
    ("B", none),
  ),
  rankdir: "LR",
  directed: false
)
```

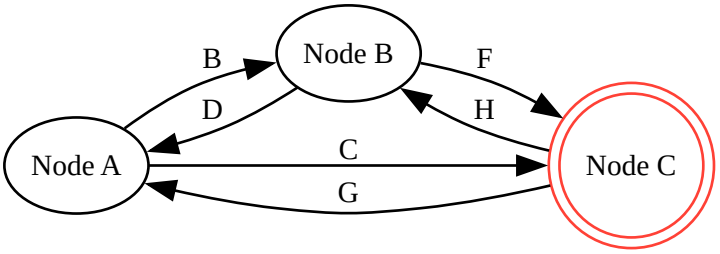


Argument

⟨vertex-labels⟩ `dictionary`

List of labels. If the value is a content, the content is used as the label of the vertex. If the value is a dictionary then the keys are the nodes attributes.

```
#adjacency(
  (
    (none, "B", "C"),
    ("D", none, "F"),
    ("G", "H", none),
  ),
  rankdir: "LR",
  vertex-labels: (
    "Node A", "Node B",
    (label: "Node C", color: red, shape: "doublecircle"),
  )
)
```



Argument

⟨clusters⟩ `content`

List of clusters. If the value is a list, it is used as the list of nodes in the cluster. If the value is a dictionary, the mandatory key is `nodes` which is the list of nodes in the cluster. The other keys are the cluster attributes.

Argument

⟨debug⟩ `bool`

Show debug boxes around the labels.

Argument

⟨clip⟩ `bool`

Whether to hide part of the graphs that goes outside the bounding box given by graphviz.

Argument

⟨dot arguments⟩

Graph arguments. There are two types of values:

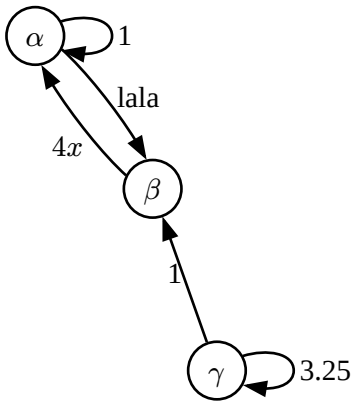
- A key – dot value pair: The key is the attribute name and the value is the attribute value.

  For example, `rankdir: "LR"` will be converted to `rankdir=LR` in dot.

- A key – dictionary pair: The key is the element name and the value is a dictionary of attributes. The attributes must be all key – dot value pairs.

  For example, `node: (shape: "circle")` will be converted to `node[shape=circle]` in dot.

```
#adjacency(
  vertex-labels: ($alpha$, $beta$, $gamma$),
  (
    ([1], move(dx: 25pt, dy: -5pt)[lala], none),
    ($4x$, none, none),
    (none, 1, 13 / 4),
  ),
  layout: "neato",
  rankdir: "LR",
  edge: (
    arrowsize: 0.7,
  ),
  node: (
    shape: "circle",
    width: 0.3,
    fixedsize: true,
  ),
)
```



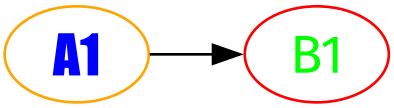
Part IV

Fonts, colors and sizes

If you don’t overwrite the labels, the specified graphviz font, color and size will be used. However, the fonts are not necessarily the same as the graphviz fonts. If you want to use a font, it must be accessible by Typst. For colors, you can us any valid graphviz color you want as they are converted to Typst colors automatically. Font size works as usual. If you overwrite a label, the font, color and size will be the ones used by Typst where the graph is rendered.

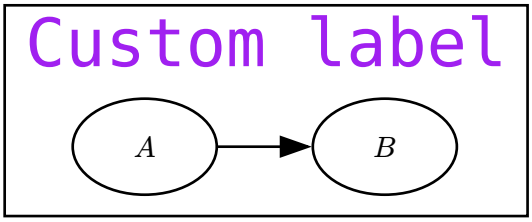
```
#raw-render(```
  digraph G {
    rankdir=LR
    A1 -> B1
    A1[color=orange, fontname="Impact", fontsize=20, fontcolor=blue]
    B1[color=red, fontname="Fira Sans", fontsize=20, fontcolor=green]
  }
```)
```

---




The same goes for edges and clusters.

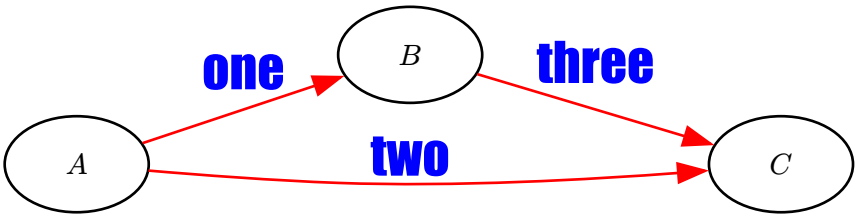
```
#raw-render(```
  digraph G {
    rankdir=LR
    subgraph cluster_0 {
      label="Custom label"
      fontname="DejaVu Sans Mono"
      fontcolor=purple
      fontsize=25
      A -> B
    }
  }
```)
```



IV Fonts, colors and sizes

```
#raw-render(```
  digraph G {
    rankdir=LR
    edge [color=red, fontname="Impact", fontsize=20, fontcolor=blue]
    A -> B[label="one"]
    A -> C[label="two"]
    B -> C[label="three"]
  }
```)
```

---

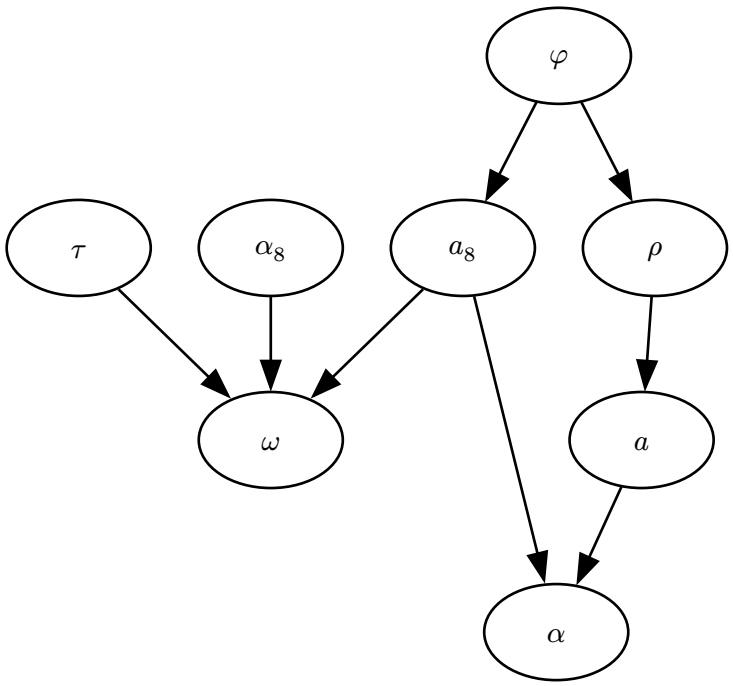

```

Part V

Automatic math mode detection

Diagraph tries to automatically detect simple math expressions. Single letters, numbers, and greek letters are automatically put in math mode.

```
#raw-render(```
  digraph {
    a -> alpha
    phi -> rho
    rho -> a
    tau -> omega
    phi -> a_8
    a_8 -> alpha
    a_8 -> omega
    alpha_8 -> omega
  }
```)
```

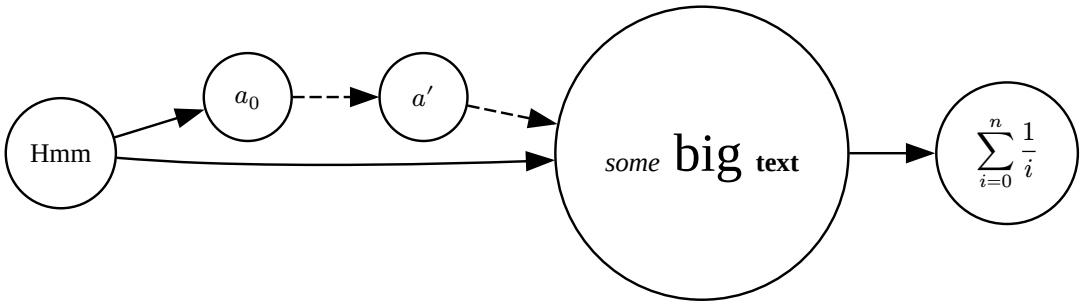

```

Part VI

Examples

Those examples are here to demonstrate the capabilities of diagraph. For more information about the Graphviz Dot language, you can check the official documentation[1]. Some of the following examples are taken from the graphviz gallery[2].

```
#raw-render(
  ```
  digraph {
    rankdir=LR
    node[shape=circle]
    Hmm -> a_0
    Hmm -> big
    a_0 -> "a'" -> big [style="dashed"]
    big -> sum
  }
  ```,
  labels: (
    big: [_some_ #text(2em)[ big ]*text*],
    sum: $ sum_(i=0)^n 1 / i $,
  ),
  width: 100%,
)
```



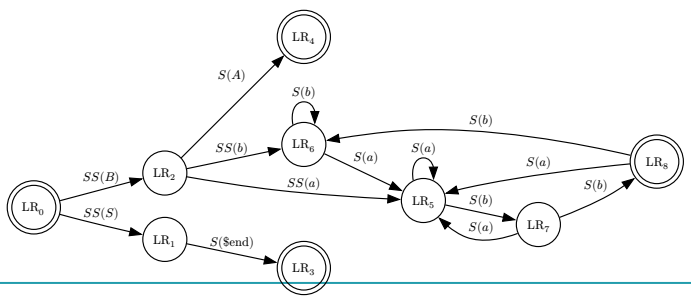
[1] https://graphviz.org/documentation/
[2] https://graphviz.org/gallery/

```
#raw-render(```
 digraph finite_state_machine {
   rankdir=LR
   size="8,5"

   node [shape=doublecircle]
   LR_0
   LR_3
   LR_4
   LR_8

   node [shape=circle]
   LR_0 -> LR_2 [label="SS(B)"]
   LR_0 -> LR_1 [label="SS(S)"]
   LR_1 -> LR_3 [label="S($end)"]
   LR_2 -> LR_6 [label="SS(b)"]
   LR_2 -> LR_5 [label="SS(a)"]
   LR_2 -> LR_4 [label="S(A)"]
   LR_5 -> LR_7 [label="S(b)"]
   LR_5 -> LR_5 [label="S(a)"]
   LR_6 -> LR_6 [label="S(b)"]
   LR_6 -> LR_5 [label="S(a)"]
   LR_7 -> LR_8 [label="S(b)"]
   LR_7 -> LR_5 [label="S(a)"]
   LR_8 -> LR_6 [label="S(b)"]
   LR_8 -> LR_5 [label="S(a)"]
 }
```,
 labels: (
   "LR_0": $"LR"_0$,
   "LR_1": $"LR"_1$,
   "LR_2": $"LR"_2$,
   "LR_3": $"LR"_3$,
   "LR_4": $"LR"_4$,
   "LR_5": $"LR"_5$,
   "LR_6": $"LR"_6$,
   "LR_7": $"LR"_7$,
   "LR_8": $"LR"_8$,
 ),
 edges:(
  "LR_0": ("LR_2": $S S(B)$, "LR_1": $S S(S)$),
  "LR_1": ("LR_3": $S(dollar"end")$),
  "LR_2": ("LR_6": $S S(b)$, "LR_5": $S S(a)$),
 ),
 height: 10em
)
```

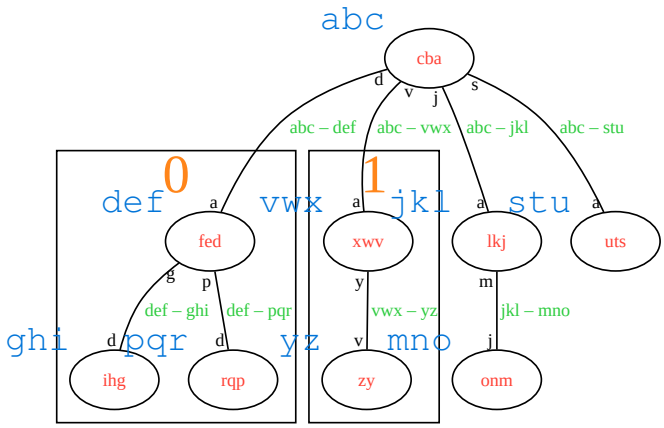
---


```

```
#import "@preview/diagraph:0.3.3": *

#raw-render(```
  graph {
    abc -- def
    def -- ghi
    jkl -- mno
    abc -- jkl
    subgraph cluster_0 {
      ghi
      def -- pqr
    }
    abc -- vwx
    abc -- stu
    subgraph cluster_1 {
      vwx -- yz
    }
  }
```,
height: 15em,
labels: (name) => {
  text(fill: red, name.rev())
},
xlabels: (name) => {
  text(fill: blue, size: 2em, font: "FreeMono", name)
},
clusters: (name) => {
    text(fill: orange, size: 3em, name.at(-1))
},
edges: (name, edges) => {
  let labels = (:)
  for edge in edges {
    labels.insert(edge, (
      label: text(fill: green, [#name -- #edge]),
      headlabel: name.at(0),
      taillabel: edge.at(0)
    ))
  }
  labels
},
)
```

---


```

Part VII

Index

A

R