

v0.0.0

2023-08-24

MIT

Typst-setting finite automata with CeTZ

Jonas NEUGEBAUER

<https://github.com/jneug/typst-finite>

FINITE is a Typst package to draw transition diagrams for finite automata (state machines) with the power of CeTZ.

The package provides new elements for manually drawing states and transitions on any CeTZ canvas, but also comes with commands to quickly create automata from a transition table.

Table of contents

I. Usage

I.1. Load from package repository (Typst 0.6.0 and later)	2
I.2. Dependencies	2
I.3. Quick start	2
I.4. Command reference	4
I.5. Drawing automata	6
I.6. Using layouts	7
I.6.1. Available layouts	7
I.6.2. Creating custom layouts	16
I.7. Utility functions	18
I.8. Doing other stuff with FINITE	20

II. Index

Part I.

Usage

I.1. Load from package repository (Typst 0.6.0 and later)

For Typst 0.6.0 and later, the package can be imported from the *preview* repository:

```
#import "@preview/finite:0.0.0:automaton"
```

Alternatively, the package can be downloaded and saved into the system dependent local package repository.

Either download the current release from GitHub¹ and unpack the archive into your system dependent local repository folder² or clone it directly:

```
git clone https://github.com/jneug/typst-finite finite/0.0.0
```

In either case, make sure the files are placed in a subfolder with the correct version number: `finite/0.0.0`

After installing the package, just import it inside your `typ` file:

```
#import "@local/finite:0.0.0:automaton"
```

I.2. Dependencies

FINITE loads **CETZ** and the utility package **T4T** from the preview package repository. The dependencies will be downloaded by Typst automatically on first compilation.

I.3. Quick start

FINITE helps you draw transition diagrams for finite automata in your Typst documents, using the power of **CETZ**.

To draw an automaton, import `#automaton()` from **FINITE** and use it like this:

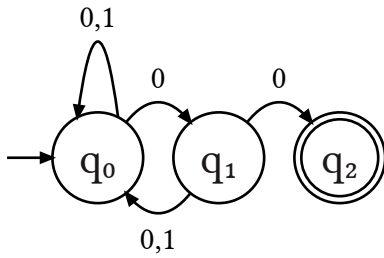
```
#automaton((
  q0: (q1:0, q0:"0,1"),
  q1: (q0:(0,1), q2:"0"),
  q2: (),
))
```

The output looks like this:

¹<https://github.com/jneug/typst-finite>

²<https://github.com/typst/packages#local-packages>

1.3 Quick start

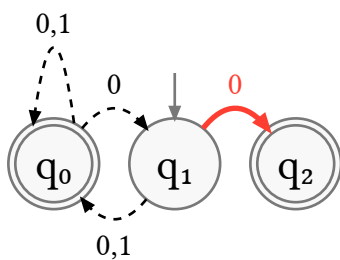


As you can see, an automaton is defined by a dictionary of dictionaries. The keys of the top-level dictionary are the names of the states to draw. The second-level dictionaries have the names of connected states as keys and transition labels as values.

In the example above, the states `q0`, `q1` and `q2` are defined. `q0` is connected to `q1` and has a loop to itself. `q1` transitions to `q2` and back to `q0`. `#automaton()` selected the first state in the dictionary (in this case `q0`) to be the initial state and the last (`q2`) to be a final state.

To modify the defaults, `#automaton()` accepts a set of options:

```
1 #automaton(  
2   (  
3     q0: (q1:0, q0:"0,1"),  
4     q1: (q0:(0,1), q2:"0"),  
5     q2: (),  
6   ),  
7   start: "q1",  
8   stop: ("q0", "q2"),  
9   style:(  
10    state: (fill: luma(248), stroke:luma(120)),  
11    transition: (stroke: (dash:"dashed")),  
12    q1: (start:top),  
13    q1-q2: (stroke: 2pt + red)  
14  )  
15 )
```

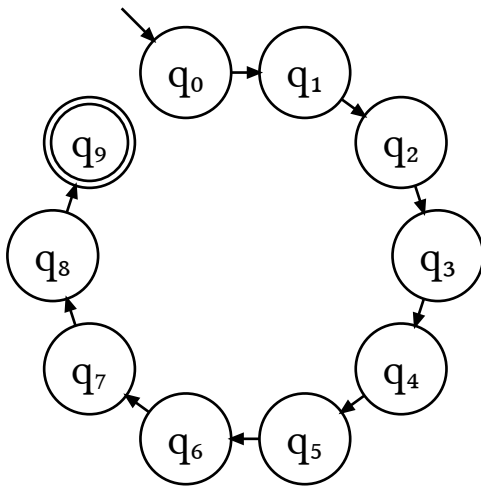


For larger automata, the states can be arranged in different ways:

```

1 #let aut = (:)
2 #for i in range(10) {
3   let name = "q"+str(i)
4   aut.insert(name, (:))
5   if i < 9 {
6     aut.at(name).insert("q" + str(i + 1), none)
7   }
8 }
9 #automaton(
10   aut,
11   layout: finite.layout.circular.with(offset: 2),
12   style: (
13     transition: (curve: 0),
14     q0: (start: top+left)
15   )
16 )

```



See Section I.6 for more details about layouts.

I.4. Command reference

```

#automaton(
  states,
  start: auto,
  stop: auto,
  style: "(:)",
  label-format: (...) => ...,
  layout: "layout.linear",
  ..canvas-styles
)

```

Draw an automaton from a transition table.

Argument

states

dictionary

A dictionary of dictionaries, defining the transition table of an automaton.

Argument

1.4 Command reference

start: `auto` `string` | `auto` | `none`

The name of the initial state. For `auto`, the first state in `states` is used.

—Argument—

stop: `auto` `array` | `auto` | `none`

A list of final state names. For `auto`, the last state in `states` is used.

—Argument—

style: `(:)` `dictionary`

A dictionary with styles for states and transitions.

—Argument—

label-format: `(..) => ..` `function`

A function (`string`, `boolean`) => `string` to format labels.

- layout (function: A layout function.

—Argument—

..canvas-styles `any`

Arguments for `#cetz.canvas()`

#transition-table(`states`, `start: auto`, `stop: auto`, `format: (...) => ...`)

Displays a transition table for an automaton.

The format for states is the same as for `automaton()`.

```
1 #finite.transition-table((
2   q0: (q1: 0, q0: 1),
3   q1: (q0: 1, q2: 0),
4   q2: (q0: 1, q2: 0),
5 ))
```

	0	1
q0	q1	q0
q1	q2	q0
q2	q2	q0

—Argument—

states `dictionary`

A dictionary of dictionaries, defining the transition table of an automaton.

—Argument—

start: `auto` `string` | `auto` | `none`

The name of the initial state. For **auto**, the first state in `states` is used.

Argument

stop: **auto**

array | **auto** | **none**

A list of final state names. For **auto**, the last state in `states` is used.

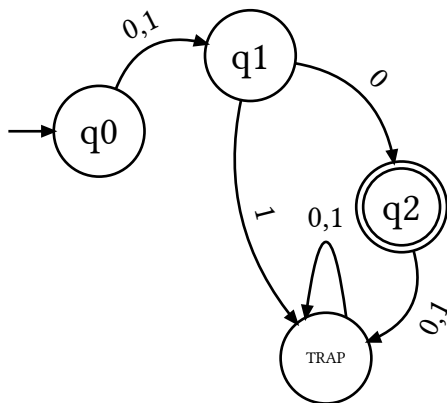
1.5. Drawing automata

The above commands use custom **CETZ** elements, to draw states and transitions. For complex automata, the functions in the **draw** module can be used directly.

```

1  #cetz.canvas({
2      import cetz.draw: set-style
3      import finite.draw: state, transition
4
5      state((0,0), "q0", start:true)
6      state((2,1), "q1")
7      state((4,-1), "q2", stop:true)
8      state((3,-3), "trap", label:"TRAP")
9
10     transition("q0", "q1", label:(0,1))
11     transition("q1", "q2", label:(0))
12     transition("q1", "trap", label:(1), curve:-1)
13     transition("q2", "trap", label:(0,1))
14     transition("trap", "trap", label:(0,1))
15 })

```



```

#state(
    position,
    name,
    label: auto,
    start: "false",
    stop: "false",
    anchor: "center",
    ..style
)

```

Draw a state at the given position.

`#transition(from, to, label: none, ..style)`

Draw a transition between two states.

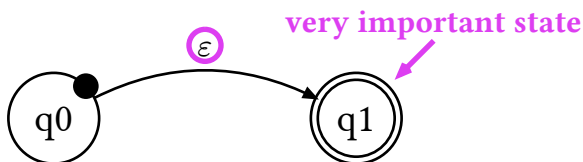
The two states `from` and `to` have to be drawn first.

States have the common anchors (like `top`, `top-left` ...), transitions have a `start`, `end`, `center` and `label` anchors. These can be used to draw additional elements:

```

1  #cetz.canvas({
2    import cetz.draw: circle, line, place-marks, content
3    import finite.draw: state, transition
4
5    state((0,0), "q0")
6    state((4,0), "q1", stop:true)
7    transition("q0", "q1", label:$epsilon$)
8
9    circle("q0.top-right", radius:.4em, stroke:none, fill:black)
10
11    let magenta-stroke = 2pt+rgb("#dc41f1")
12    circle("q0-q1.label", radius:.5em, stroke:magenta-stroke)
13    place-marks(
14      line(
15        name: "q0-arrow",
16        (rel:(.6,.6), to:"q1.top-right"),
17        (rel:(.15,.15), to:"q1.top-right"),
18        stroke:magenta-stroke
19      ),
20      (mark:">", pos:1, stroke:magenta-stroke)
21    )
22    content(
23      (rel:(0,.25), to:"q0-arrow.start"),
24      text(fill:rgb("#dc41f1"), [*very important state*])
25    )
26  })

```



I.6. Using layouts

Layouts calculate coordinates for every state in a transition table and return a dictionary with all computed locations.

`FINITE` ships with a bunch of layouts, to accomodate different scenarios.

Layouts currently are very simple. They will most likely see improvements in the future. At the moment, layouts don't know about the context and can't resolve coordinates other than absolute coordinate pairs.



I.6.1. Available layouts

1.6.1 Using layouts

`#circular()`
`#fixed()`

`#grid()`
`#group()`

`#linear()`
`#snake()`

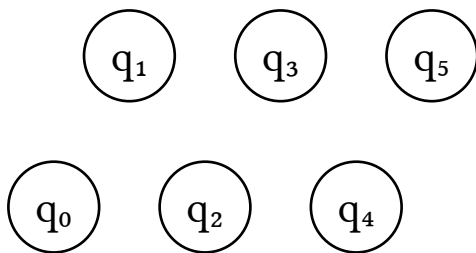
`#fixed(states, start, stop, pos: "(:)")`

A fixed list of coordinates.

Can be used to manually specify individual coordinates for each state. Note, that coordinates may be any specified in any coordinate system `CETZ` knows. If you want to use fixed coordinates as a sub-layout (e.g. in `group()`), all coordinates must be in the (x,y)-system.

Notes not in `pos` will be placed at `(0, 0)`.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #finite.automaton(
3   aut,
4   start: none, stop: none,
5   layout: finite.layout.fixed.with(pos:(
6     q0: (0, -1), q1: (1, 1), q2: (2, -1),
7     q3: (3, 1), q4: (4, -1), q5: (5, 1)
8   ))
9 )
```



Argument

`states`

dictionary

Transition table.

Argument

`start`

string | none

Initial state.

Argument

`stop`

array | none

List of final states.

Argument

`pos: (:)`

dictionary

Dictionary with coordinates for each state.

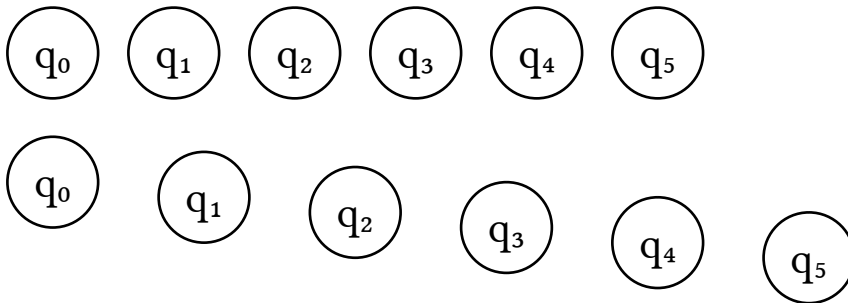
1.6.1 Using layouts

```
#linear(  
  states,  
  start,  
  stop,  
  x: 1.6,  
  y: 0  
)
```

Arrange states in a line.

The direction of the line can be set by x and y.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})  
2 #finite.automaton(  
3   aut,  
4   start: none, stop: none,  
5   layout:finite.layout.linear  
6 )  
7 #finite.automaton(  
8   aut,  
9   start: none, stop: none,  
10  layout:finite.layout.linear.with(x:2, y:-.2)  
11 )
```



Argument

states

dictionary

Transition table.

Argument

start

string | none

Initial state.

Argument

stop

array | none

List of final states.

Argument

x: 1.6

float

Step size along the x-axis.

Argument

1.6.1 Using layouts

y: 0

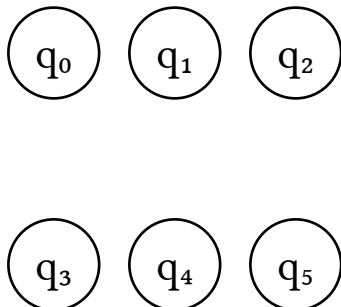
float

Step size along the y-axis.

```
#grid(  
  states,  
  start,  
  stop,  
  columns: 6,  
  x: 1.6,  
  y: 2.8  
)
```

Arrange states in rows and columns.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})  
2 #finite.automaton(  
3   aut,  
4   start: none, stop: none,  
5   layout: finite.layout.grid.with(columns: 3)  
6 )
```



Argument

states

dictionary

Transition table.

Argument

start

string | none

Initial state.

Argument

stop

array | none

List of final states.

Argument

columns: 6

integer

1.6.1 Using layouts

Number of columns per row.

Argument

x: 1.6

float

Distance between the center of each column.

Argument

y: 2.8

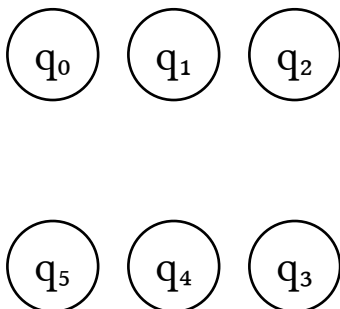
float

Distance between the center of each row.

```
#snake(  
  states,  
  start,  
  stop,  
  columns: 6,  
  x: 1.6,  
  y: 2.8  
)
```

Arrange states in a grid, but alternate the direction in every even and odd row.

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})  
2 #finite.automaton(  
3   aut,  
4   start: none, stop: none,  
5   layout: finite.layout.snake.with(columns: 3)  
6 )
```



Argument

states

dictionary

Transition table.

Argument

start

string | none

Initial state.

Argument

1.6.1 Using layouts

stop

array | none

List of final states.

—Argument—

columns: 6

integer

Number of columns per row.

—Argument—

x: 1.6

float

Distance between the center of each column.

—Argument—

y: 2.8

float

Distance between the center of each row.

```
#circular(  
  states,  
  start,  
  stop,  
  radius: auto,  
  spacing: 1.6,  
  dir: right,  
  offset: 0  
)
```

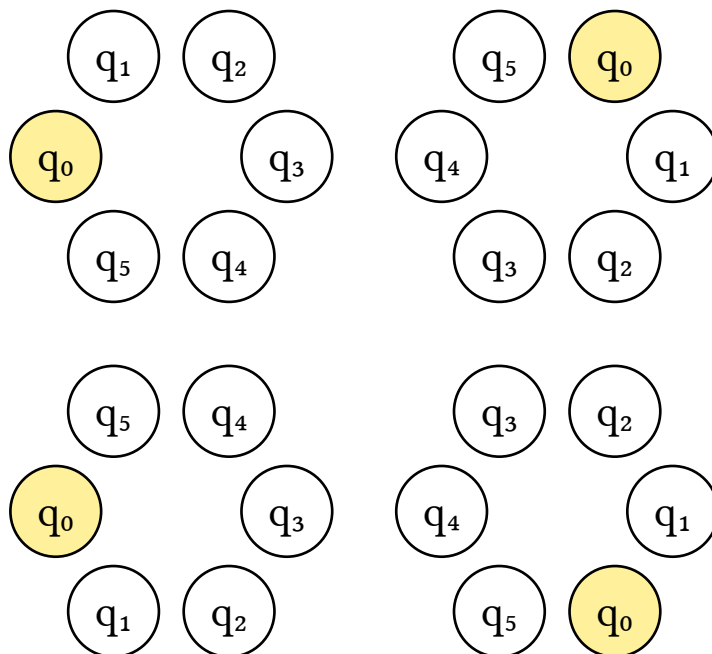
Arrange states in a circle.

1.6.1 Using layouts

```

1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #grid(columns: 2, gutter: 2em,
3   finite.automaton(
4     aut,
5     start: none, stop: none,
6     layout:finite.layout.circular,
7     style: (q0: (fill: yellow.lighten(60%)))
8   ),
9   finite.automaton(
10    aut,
11    start: none, stop: none,
12    layout:finite.layout.circular.with(offset:2),
13    style: (q0: (fill: yellow.lighten(60%)))
14  ),
15  finite.automaton(
16    aut,
17    start: none, stop: none,
18    layout:finite.layout.circular.with(dir:left),
19    style: (q0: (fill: yellow.lighten(60%)))
20  ),
21  finite.automaton(
22    aut,
23    start: none, stop: none,
24    layout:finite.layout.circular.with(dir:left, offset:2),
25    style: (q0: (fill: yellow.lighten(60%)))
26  )
27 )

```



Argument

states

dictionary

Transition table.

Argument

1.6.1 Using layouts

`start` string | none
Initial state.

—Argument—
`stop` array | none
List of final states.

—Argument—
`radius: auto` float | auto
Radius of the circle, the states are placed on. With `auto`, the radius is calculated based on spacing.

—Argument—
`spacing: 1.6` float
Distance between the center of each state on the circle.

—Argument—
`dir: right` alignment
`left` or `right` to indicate the direction, the states are arranged on the circle.

—Argument—
`offset: 0` integer
An offset, where the initial state should be placed on the circle. For `offset: 0`, the initial state is placed to the left or right (depending on `dir`) of the circle. For `offset: 1`, the initial state is placed one step in the direction indicated by `dir`.

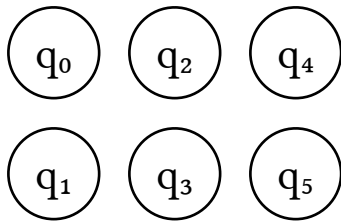
```
#group(  
  states,  
  start,  
  stop,  
  x: 1.6,  
  y: 0,  
  grouping: 5,  
  layout: "linear.with(x:0, y:-1.6)"  
)
```

Group states and layout each group with its own layout.

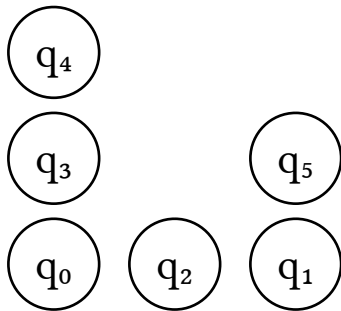
1.6.1 Using layouts

```
1 #let aut = range(6).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 Group every two states together:
3 #finite.automaton(
4   aut,
5   start: none, stop: none,
6   layout: finite.layout.group.with(
7     grouping: 2
8   )
9 )
10 Group specific states and arrange from bottom to top:
11 #finite.automaton(
12   aut,
13   start: none, stop: none,
14   layout: finite.layout.group.with(
15     grouping: (
16       ("q0", "q3", "q4"),
17       ("q2",),
18       ("q1", "q5"),
19     ),
20     layout: finite.layout.linear.with(x:0, y:1.4)
21   )
22 )
```

Group every two states together:



Group specific states and arrange from bottom to top:



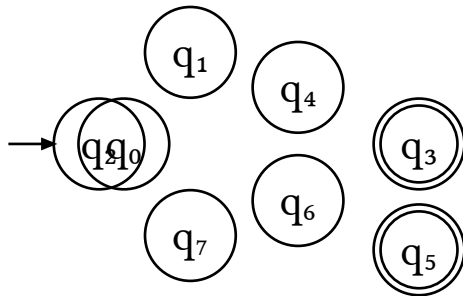
#start-stop(states, start, stop, layout: "snake")

Arrange initial state on the left, final states on the right and the rest in the center with a custom layout.

```

1 #let aut = range(8).fold((:), (d, s) => {d.insert("q"+str(s), none); d})
2 #finite.automaton(
3   aut,
4   start: "q2", stop: ("q3", "q5"),
5   layout: finite.layout.start-stop.with(
6     layout: finite.layout.circular
7   ),
8 )

```



1.6.2. Creating custom layouts

A layout is a function, that, provided with information about the automaton, returns a dictionary with the state names as keys and valid coordinates as values.

`#linear-layout()` is a simple example:

```

1 let linear-layout(states, start, stop, x:2.2, y:0) = {
2   let positions = (:)
3   for (i, name) in states.keys().enumerate() {
4     positions.insert(name, (i * x, i * y))
5   }
6   return positions
7 }

```

A layout function always gets passed the states dictionary (the same dictionary that is passed to `#automaton()`), the start state and the list of end states `stop`.

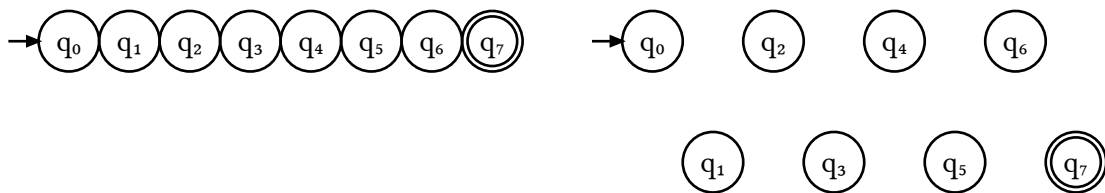
Additional named arguments may be used to configure the layout via `with`:

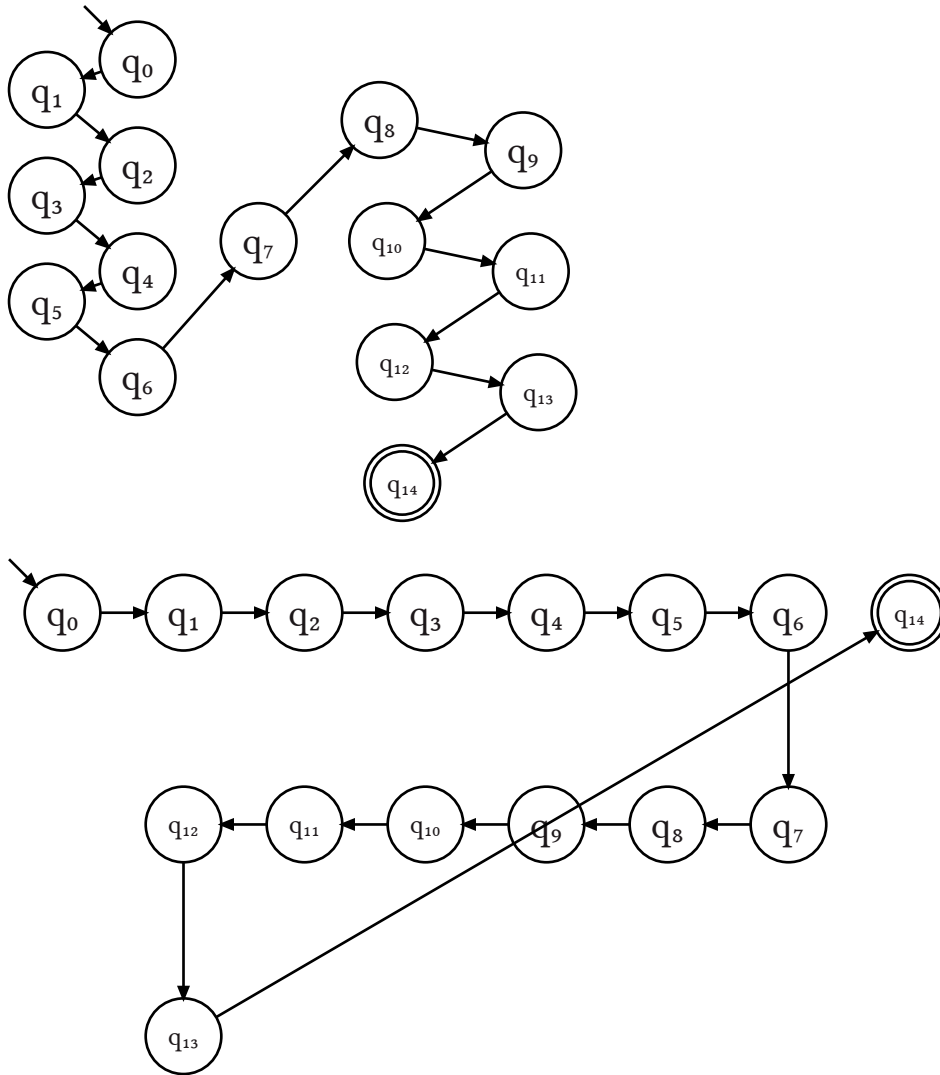
```
linear-layout.with(x:-1, y:2.2)
```

This example arranges the states in a wave:

1.6.2 Using layouts

```
1 #let wave-layout(  
2   states, start, stop,  
3   x: 1.6, amp: 1, peri: 1,  
4   generator: calc.sin  
5 ) = {  
6   let positions = (:)  
7  
8   for (i, state) in states.keys().enumerate() {  
9     positions.insert(state, (  
10      i * x, peri * generator(2 * calc.pi * i * (1/amp))  
11    ))  
12  }  
13  
14  return positions  
15 }  
16  
17 #let aut = (:)  
18 #for i in range(8) {  
19   aut.insert("q"+str(i), none)  
20 }  
21  
22 #grid(  
23   columns:(1fr,1fr),  
24   automaton(  
25     aut,  
26     layout: wave-layout.with(generator: calc.sin, x:.8),  
27     style: (state: (radius: .4))  
28   ),  
29   automaton(  
30     aut,  
31     layout: wave-layout.with(generator: calc.cos, x:.8, amp:2, peri:.8),  
32     style: (state: (radius: .4))  
33   )  
34 )
```





I.7. Utility functions

```
#ctrl-pt()  
#fit-content()
```

```
#label-pt()  
#mark-dir()
```

```
#prepare-ctx()  
#transition-pts()
```

#ctrl-pt(a, b, curve: 1)

Calculate the controlpoint for a transition.

#mark-dir(a, b, c, scale: 1)

Calculate the direction vector for a transition mark (arrowhead)

#label-pt(

```
  a,  
  b,  
  c,
```

1.7 Utility functions

```
style,  
loop: "false"  
)
```

Calculate the location for a transitions label, based on its bezier points.

```
#transition-pts(  
  start,  
  end,  
  start-radius,  
  end-radius,  
  curve: 1  
)
```

Calculate start, end and ctrl points for a transition.

Argument	
start	vector
Center of the start state.	
Argument	
end	vector
Center of the end state.	
Argument	
start-radius	length
Radius of the start state.	
Argument	
end-radius	length
Radius of the end state.	
Argument	
curve: 1	float
Curvature of the transition.	

```
#fit-content(  
  ctx,  
  width,  
  height,  
  content,  
  size: auto,  
  min-size: "6pt"  
)
```

Fits (text) content inside the available space.

Argument

1.7 Utility functions

`ctx`

dictionary

The canvas context.

Argument

`content`

string | content

The content to fit.

Argument

`size: auto`

length | auto

The initial text size.

Argument

`min-size: 6pt`

length

The minimal text size to set.

#prepare-ctx(ctx)

Prepares the CeTZ context for use with finite

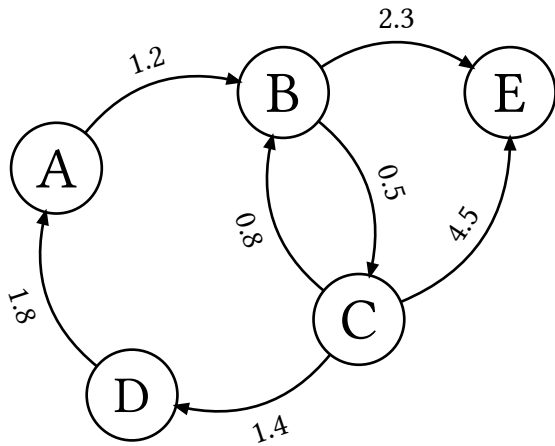
1.8. Doing other stuff with finite

Since transition diagrams are effectively graphs, **FINITE** could also be used to draw graph structures:

```

1  #cetz.canvas({
2      import cetz.draw: set-style
3      import finite.draw: state, transitions
4
5      state((0,0), "A")
6      state((3,1), "B")
7      state((4,-2), "C")
8      state((1,-3), "D")
9      state((6,1), "E")
10
11     transitions((
12         A: (B: 1.2),
13         B: (C: .5, E: 2.3),
14         C: (B: .8, D: 1.4, E: 4.5),
15         D: (A: 1.8),
16         E: (:)
17     )),
18     C-E: (curve: -1.2))
19 })

```



Part II.

Index

A

#automaton 2, 4

C

#circular 12

#ctrl-pt 18

F

#fit-content 19

#fixed 8

G

#grid 10

#group 14

L

#label-pt 18

#linear 9

#linear-layout 16

M

#mark-dir 18

P

#prepare-ctx 20

S

#snake 11

#start-stop 15

#state 6

T

#transition 7

#transition-pts 19

#transition-table 5