Efficient Low Diameter Clustering

with strong diameter in the CONGEST model

Christian Micheletti
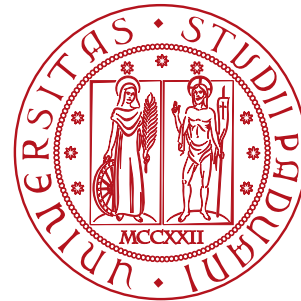
- We want to solve graph problems on **networks**
  - ‣ Computers are like nodes in a graph

Distribution ⇒ Multiple processors

- We want to solve graph problems on **networks**
  - ‣ Computers are like nodes in a graph

Distribution $\Rightarrow$ Multiple processors

💡 Each node gives a partial solution

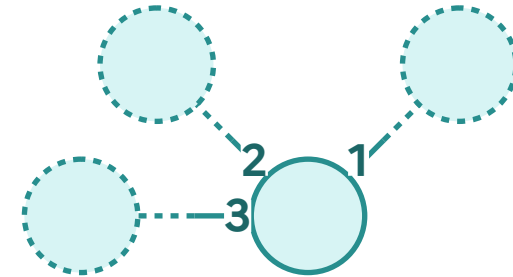- At the end they are all combined altogether

…how can we combine those partial solutions?

> 🚨 **Distribution ⇒ Collaboration**

- Arcs are **direct links** between computers
  - ‣ Computers have to exchange **messages**

- In the **PN-Network** a node only knows some *Numbered Ports*
  - ‣ Connected with a **different** nodes
    - – Such nodes are called *"neighbours"*
- There are no **self loops**

🚨   **A node can't see the whole topology**

🚨 **All nodes appear identical**

- The only difference *could be* in the total number of ports
    - ‣ Not enough!

- **We must break this symmetry**

💡 We add **unique identifiers** to the nodes

$$id : V \rightarrow \mathbb{N}$$

where $\forall v \in V : id(v) \leq n^c$ for some $c \geq 1$

We choose n^c so that we need $O(\log n)$ bits to represent an identifier, i.e. identifiers are reasonably *"small"*

- Collaborating requires **exchanging messages**
  - ‣ …on a medium that is **slow** and **unreliable**

🚨   ⇒ **Communication is the main pitfall**

- We are intersted in **quantifying** the number of messages that algorithms need
  - ‣ An ***"efficient"*** algorithm will need few messages

W.l.o.g.[1] we adopt a model of *"synchronous communication"*

Each round, a node $v \in V$ performs this actions:
1. v *"sends"* a message $msg \in \mathbb{N}$ to its neighbours;
2. v *"receives"* messages from its neighbours;
3. ...

- (1.) and (2.) establish a *"communication round"*
  - Main measure of complexity

---

[1]Without loss of generality.

3. *v **"executes locally"*** some algorithm (same for each node).
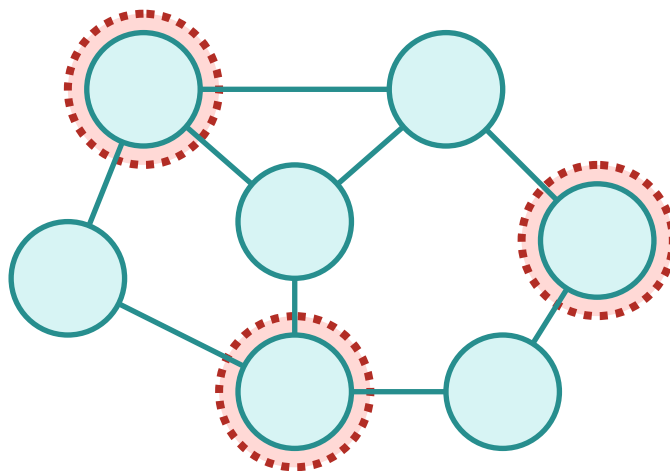
- A node can ***"stop"*** in this phase

🎶🎵   (3.) doesn't affect the algorithm's complexity

- When all nodes **stopped** the algorithm terminates

- The node with id(v) = 1 *"waves hello"* to neighbours
  - ‣ …sending them a message

- When a node receives the message, **forwards** it to its neighbours
  - ‣ And then **stops**

- The running time of this algorithm on a graph G is $O(diam(G))$

Example: **Maximal Independent Set** (MIS)

- Solving it **_"centralized"_** is easy
- How can we solve it **_"distributed"_**?

💡 Let's leverage id(v) to select the next MIS node

- At round #i, node v : id(v) = i executes
  - ‣ If no neighbour is in the MIS, add the node
    - – And inform the neighbours
  - ‣ Otherwise, the node is outside the MIS

- It is correct since no node has the same id

- This algorithm runs in $O(n^c)$ (the maximum id)
  - ‣ **Very bad**

💡 Running a centralized algorithm on a single node would take O(1) rounds

- We'd like to run a MIS algorithm on each node
  - ▸ Each must have a **local copy** of the **entire** graph
  - ▸ The algorithm must be deterministic
    - – When a node stops it checks if it is included in MIS

- The algorithm Gather-All makes all nodes build a local copy of the whole graph
  - ‣ At round i, each node v knows $ball(i, v)$

$$ball(0, v)$$

$$ball(1, v)$$
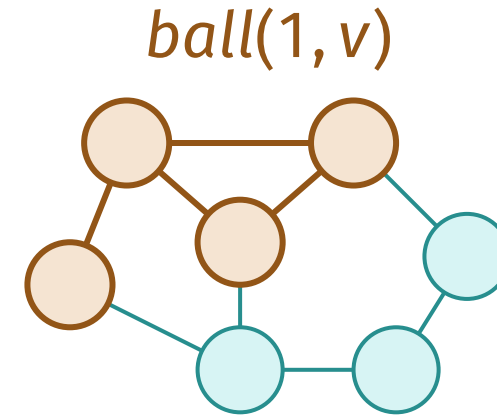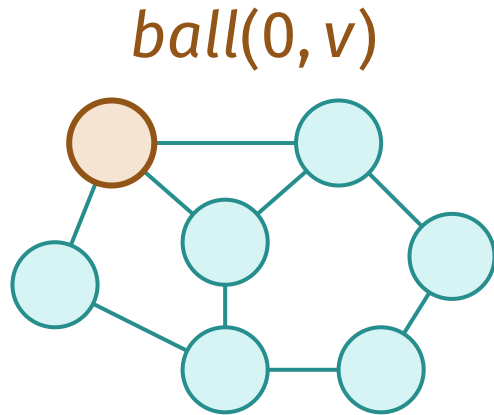
- All nodes will know the whole graph after $O(diam(G))$ rounds

- GATHER-ALL assumes that messages size is **unbounded**

> 🚨 **Requires to send the whole graph in one message**

- It is not always possible to send arbitrary large messages
  - ‣ Heavy ones may be "sharded"
- We provide an upper bound for message size
  - ‣ Messages need to be reasonably *"small"*
  - ‣ Large messages will require more rounds to be sent

In the CONGEST model, messages size has to be $O(\log n)$

- Sending k identifiers takes $O(1)$ rounds
- Sending a *set* of identifiers can take up to $O(n)$
- Sending the whole graph requires $O(n^2)$ rounds:
  - ‣ The adjacency matrix suffices...

🚨 ⇒ We can't use GATHER-ALL in the CONGEST model

We've established our computation model

Now we'll see how to solve problems in CONGEST

- Censor-Hillel et al. [1] provided an algorithm that solves MIS in $O\big(diam(G)\log^2 n\big)$ in CONGEST

🚨 **The diameter can be very large**

- Worst case: $diam(G) = n$
- How can we improve it?

- A *"Network Decomposition"* groups nodes in **colored clusters**
  - ‣ Clusters with the same color are not adjacent
  - ‣ We say it to *"have diameter"* d if each cluster has diameter at most d
  - ‣ It has c colors
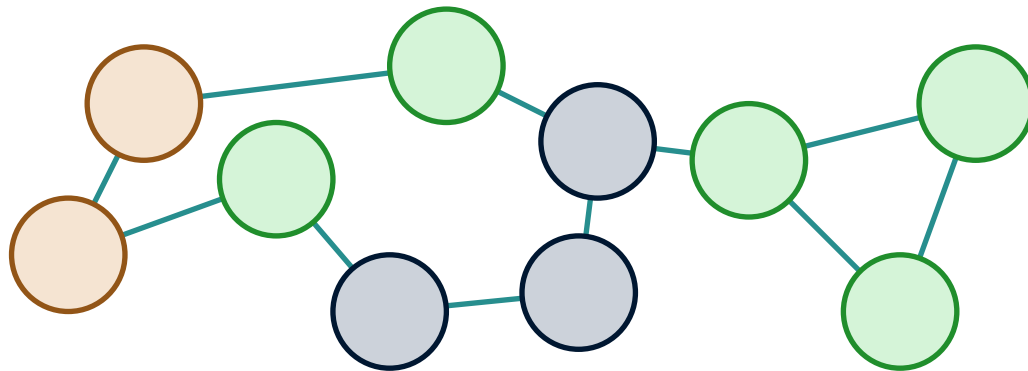
💡 Solving MIS in a color gives a partial solution

- We can apply [1] for all colors
  - ‣ (dropping MIS neighbours after each iter)
  - ‣ This has complexity $O(c \cdot d \log^2 n)$
    - – If $c = O(\log n) = d$ it would be **efficient**
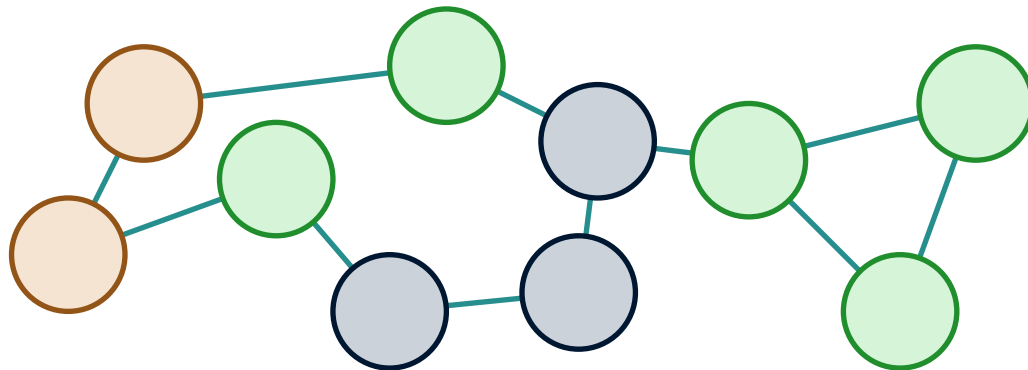
A *"low diameter clustering"* $\mathscr{C} \subseteq 2^V$ for a graph G with diameter d is such:

1. $\forall C_1 \neq C_2 \in \mathscr{C} : dist_G(C_1, C_2) \geq 2$
   - *"There are no adjacent clusters"*
2. $\forall C \in \mathscr{C} : diam(G[C]) \leq d$
   - *"Any cluster has diameter at most"* d

A clustering **can not be a partitioning**: some nodes have to be left out

Main iteration:
1. Find a low diameter clustering
2. Assign a free color to its nodes
3. Repeat to discarded nodes until there are no more left
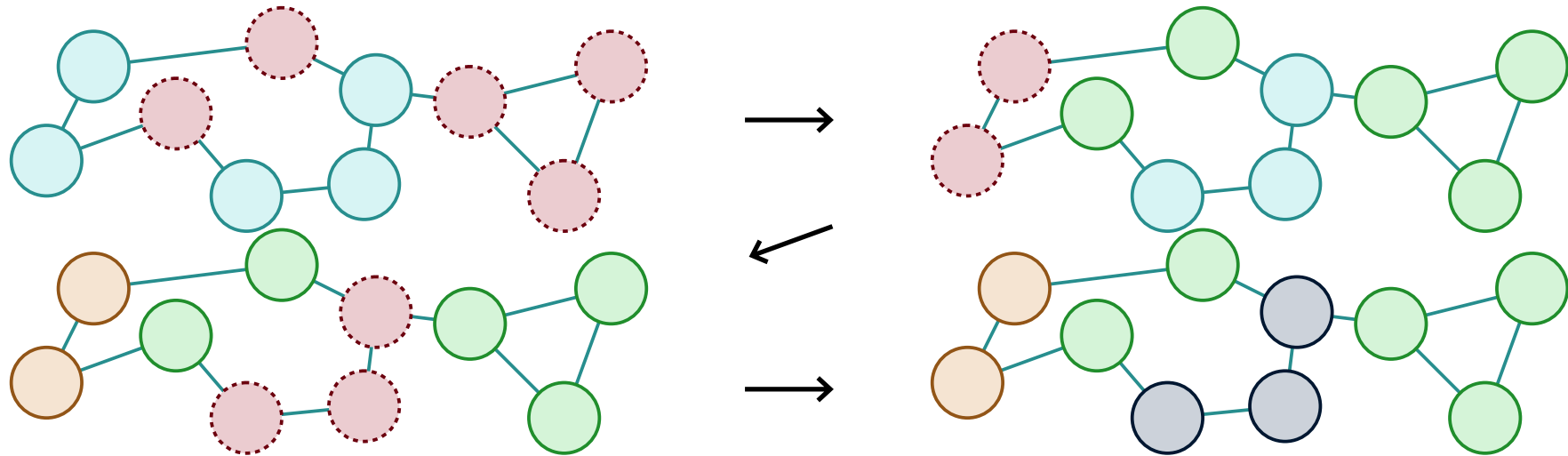
To get a $O(\log n)$-colors decomposition, each color has to cluster **at least half** of the uncolored nodes

- Otherwise we can end up with too many colors

- Our previous definition of diamter is also called ***"strong"*** diameter

We say a clustering has ***"weak"*** diameter when:
1. (unchanged) **"There are no adjacent clusters"**
2. "Any cluster has diameter **in** G at most" d

How to compute a low diameter clustering

- Main accomplishments of [2]:
  - ▸ Terminates in $O(\log^6 n)$ rounds in the CONGEST model
  - ▸ Outputs a clustering with $O(\log^3 n)$ colors
  - ▸ Directly strong diameter

- Main accomplishments of [2]:
  - ‣ Terminates in $O(\log^6 n)$ rounds in the CONGEST model
  - ‣ Outputs a clustering with $O(\log^3 n)$ colors
  - ‣ Directly strong diameter

- Previously [3] provided a l.d.c with **"weak"** diameter
  - ‣ $O(\log^7 n)$ rounds with $O(\log^3 n)$ colors
  - ‣ It's possible to turn it into strong diameter

- Main accomplishments of [2]:
  - ‣ Terminates in $O(\log^6 n)$ rounds in the CONGEST model
  - ‣ Outputs a clustering with $O(\log^3 n)$ colors
  - ‣ Directly strong diameter

- Previously [3] provided a l.d.c with **"weak"** diameter
  - ‣ $O(\log^7 n)$ rounds with $O(\log^3 n)$ colors
  - ‣ It's possible to turn it into strong diameter

- [4] did it in $O(\log^4 n)$ rounds with $O(\log^3 n)$ colors
  - ‣ Has to pass by a weak d. intermediate solution

"Phases"

- There are $b = \log(\max i) = O(\log n)$ *"phases"*
- "One phase for each bit in index"
  - ‣ Phase $i \in [0, b - 1]$ computes *""terminals" set"* Q_i

"Notation"

- Q_i is the terminals set built *before* phase i
- Q_b is the terminals set built *after* phase $b - 1$

"Objective 1:" Creating **connected components** with low diameters

- Eventually, each connected component will contain exactly one terminal
  - ‣ Keep terminals <u>**close**</u> to active nodes
    - – This ensures polylogarithmic diameter
    - – Removing nodes is allowed

- Q_{i+1} is "*closer*" than Q_i to any node
  - ‣ Q_b is <u>**close**</u> to any node

"Objective 2:"

- The algorithm must cluster at least $\frac{n}{2}$ nodes
  - ‣ Each phase **removes** at most $\frac{n}{2b}$ nodes
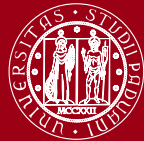
"Further notation:"

- V_i is the set of **living nodes** at the beginning of phase i ($V_0 = V$)

- V' is the set of **living nodes** after the last phase;

  - ▸ $V' = V_b$

1. Q_i is R_i-ruling, i.e. $dist_G(Q_i, v) \leq R_i$ for all $v \in V$
   - We set $R_i = i * O(\log^2 n)$
   - Q_0 is 0-ruling, trivially true with $Q_0 = V$
   - Q_b is $O(\log^3 n)$-ruling

> Each node has polylogarithmic distance from $Q_b \Rightarrow$ each connected component has *"at least one"* terminal

2. let $q_1, q_2 \in Q_i$ s.t. they are in the same connected component in $G[V_i]$. Then $\text{id}(q_1)[0..i] = \text{id}(q_2)[0..i]$

- for $i = 0$ it's trivially true
- for $i = b$ there is ≤ 1 terminal in each c.c.

Along with invariant (1.), it means that each c.c. has polylogarithmic diameter!

3. $|V_i| \geq \left(1 - \frac{i}{2b}\right)|V|$

- $V_0 \geq V$
- $V' \geq \frac{1}{2}|V|$

"The algorithm doesn't discard too much vertexes from the graph"

"Objective:" Keeping only terminals from which is possible to build a **forest** whose **trees** have polylogarithmic diameter

- Leaves of the trees may be connected in G

"Outline:"

- $2b^2$ *"steps"*, each computing a forest
- resulting into a sequence of forests $F_0..F_{2b^2}$

"Inductive definition:"

- F_0 is a BFS forest with roots in Q_i
- let T be any tree in F_j and r its root
  - ‣ if id(r)[i] = 0 the whole tree is `red`, if not `blue`
    - – `red` vertexes stay `red`
    - – some `blue` nodes stay `blue`
    - – some others *"**propose**"* to join `red` trees

"Proposal:"

$v \in V_j^{propose} \Leftrightarrow v$ is `blue`

$\land v$ is the only one in $path(v, root(v))$

that neighbours a `red` node

Define T_v the (blue) subtree rooted at v

♪♪ v is the only node in T_v that is also in $V_j^{propose}$

"Proposal:"

- Each node in $V_j^{propose}$ proposes to an arbitrary `red` neighbour
- Each `red` tree decides to grow or not
  - ‣ If it grows, it accepts all proposing trees
  - ‣ If not, all proposing subtrees are frozen
- *"Criteria:"* it decides to grow if it would gain at least $\frac{|V(T)|}{2b}$ nodes

♪♪ **If the `red` tree doesn't decide to grow, it will neighbour `red` nodes only**

- This means it will be able to delete nodes only once in the whole phase

  $\Rightarrow$ At most $\frac{|V|}{2b}$ nodes are lost in each phase

  $\Rightarrow$ After the b phases at most $\frac{|V|}{2}$ nodes are removed

1: $V_0 \leftarrow V$

2: $Q_0 \leftarrow V$

3: **for** $i \in 0..b - 1$ **do**

4:      INIT F_0

5:      **for** $j \in 0..2b^2 - 1$ **do**

6:          BUILD $V_j^{propose}$

7:          $F_{j+1} \leftarrow$ STEP

8:      $V_{i+1} \leftarrow V\left(F_{2b^2}\right)$

9:      $Q_{i+1} \leftarrow roots\left(F_{2b^2}\right)$

$\left. \begin{array}{c} \\ \\ \end{array} \right\} O(diam(T_v))$

$\left. \begin{array}{c} \\ \\ \end{array} \right\} 2b^2 = O(\log^2 n)$

$\left. \begin{array}{c} \\ \\ \end{array} \right\} b = O(\log n)$

- Recall invariant (1.)
  - $\forall v \in V : dist_G(Q_i, v) = O(\log^3 n)$, for all $i \in 0..b$
  - Hence, $diam(T_v) = O(\log^3 n)$, for all $v \in V$

- Complexity is $\#steps \times \#phases \times O(diam(T_v))$
  - $= O(\log n) \times O(\log^2 n) \times O(\log^3 n)$

The algorithm runs in $O(\log^6 n)$ communication steps

- We've seen how to build a **low diameter clustering** $(O(\log^6 n))$
  - ‣ It clusters at least $\frac{n}{2}$ nodes
- We can apply that until all nodes have a color
  - ‣ $O(\log n)$ steps and therefore $O(\log n)$ colors
- For each color, we solve MIS [1]
  - ‣ In parallel in the clusters $(O(\log^3 n \times \log^2 n))$

$\Rightarrow$ We end up solving MIS in $O(\log^6 n)$ rounds

Bibliography

[1] K. Censor-Hillel, M. Parter, and G. Schwartzman, "Derandomizing Local Distributed Algorithms under Bandwidth Restrictions." [Online]. Available: https://arxiv.org/abs/1608.01689

[2] V. Rozhoň, B. Haeupler, and C. Grunau, "A Simple Deterministic Distributed Low-Diameter Clustering." [Online]. Available: https://arxiv.org/abs/2210.11784

[3] V. Rozhoň and M. Ghaffari, "Polylogarithmic-Time Deterministic Network Decomposition and Distributed Derandomization." [Online]. Available: https://arxiv.org/abs/1907.10937

[4] V. Rozhoň, M. Elkin, C. Grunau, and B. Haeupler, "Deterministic Low-Diameter Decompositions for Weighted Graphs and Distributed and Parallel Applications." [Online]. Available: https://arxiv.org/abs/2204.08254

This presentation is supposed to briefly showcase what you can do with this package.

For a full documentation, read the online book.

Let's explore what we have here.

On the top of this slide, you can see the slide title.

We used the `title` argument of the `#slide` function for that:

```
#slide(title: "First slide")[
    ...
]
```

(This works because we utilise the `clean` theme; more on that later.)

Titles are not mandatory, this slide doesn't have one.

But did you notice that the current section name is displayed above that top line?

We defined it using `#new-section-slide`(`"Introduction"`).

This helps our audience with not getting lost after a microsleep.

You can also spot a short title above that.

Now, look down!

There we have some general info for the audience about what talk they are actually attending right now.

You can also see the slide number there.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e.

Sometimes we don't want to display everything at once.

Sometimes we don't want to display everything at once.

That's what the #pause function is there for!

Sometimes we don't want to display everything at once.

That's what the #pause function is there for!

It makes everything after it appear at the next subslide.

(Also note that the slide number does not change while we are here.)

When `#pause` does not suffice, you can use more advanced commands to show or hide content.

These are some of your options: - `#uncover`

- `#only`
- `#alternatives`
- `#one-by-one`
- `#line-by-line`

Let's explore them in more detail!

With #uncover, content still occupies space, even when it is not displayed.

For example,   are only visible on the second"subslide".

In () behind #uncover, you specify *when* to show the content, and in [] you then say *what* to show:

```
#uncover(3)[Only visible on the third "subslide"]
```

With #uncover, content still occupies space, even when it is not displayed.

For example, these words are only visible on the second "subslide".

In () behind #uncover, you specify *when* to show the content, and in [] you then say *what* to show:

```
#uncover(3)[Only visible on the third "subslide"]
```

With `#uncover`, content still occupies space, even when it is not displayed.

For example,   are only visible on the second"subslide".

In () behind `#uncover`, you specify *when* to show the content, and in [] you then say *what* to show:

```
#uncover(3)[Only visible on the third "subslide"]
```

Only visible on the third"subslide"

So far, we only used single subslide indices to define when to show something.

We can also use arrays of numbers ...

```
#uncover((1, 3, 4))[Visible on subslides 1, 3, and 4]
```

Visible on subslides 1, 3, and 4

...or a dictionary with `beginning` and/or `until` keys:

```
#uncover((beginning: 2, until: 4))[Visible on subslides 2, 3,
and 4]
```

So far, we only used single subslide indices to define when to show something.

We can also use arrays of numbers ...

```
#uncover((1, 3, 4))[Visible on subslides 1, 3, and 4]
```

...or a dictionary with `beginning` and/or `until` keys:

```
#uncover((beginning: 2, until: 4))[Visible on subslides 2, 3, and 4]
```

Visible on subslides 2, 3, and 4

So far, we only used single subslide indices to define when to show something.

We can also use arrays of numbers ...

```
#uncover((1, 3, 4))[Visible on subslides 1, 3, and 4]
```

Visible on subslides 1, 3, and 4

...or a dictionary with `beginning` and/or `until` keys:

```
#uncover((beginning: 2, until: 4))[Visible on subslides 2, 3,
and 4]
```

Visible on subslides 2, 3, and 4

So far, we only used single subslide indices to define when to show something.

We can also use arrays of numbers ...

```
#uncover((1, 3, 4))[Visible on subslides 1, 3, and 4]
```

Visible on subslides 1, 3, and 4

...or a dictionary with `beginning` and/or `until` keys:

```
#uncover((beginning: 2, until: 4))[Visible on subslides 2, 3,
and 4]
```

Visible on subslides 2, 3, and 4

As as short hand option, you can also specify rules as strings in a special syntax.

Comma separated, you can use rules of the form

1-3	from subslide 1 to 3 (inclusive)
-4	all the time until subslide 4 (inclusive)
2-	from subslide 2 onwards
3	only on subslide 3

Everything that works with `#uncover` also works with `#only`.

However, content is completely gone when it is not displayed.

For example, the rest of this sentence moves.

Again, you can use complex string rules, if you want.

```
#only("2-4, 6")[Visible on subslides 2, 3, 4, and 6]
```

Everything that works with `#uncover` also works with `#only`.

However, content is completely gone when it is not displayed.

For example, see how the rest of this sentence moves.

Again, you can use complex string rules, if you want.

```
#only("2-4, 6")[Visible on subslides 2, 3, 4, and 6]
```
Visible on subslides 2, 3, 4, and 6

Everything that works with #uncover also works with #only.

However, content is completely gone when it is not displayed.

For example, the rest of this sentence moves.

Again, you can use complex string rules, if you want.

```
#only("2-4, 6")[Visible on subslides 2, 3, 4, and 6]
```
Visible on subslides 2, 3, 4, and 6

Everything that works with #uncover also works with #only.

However, content is completely gone when it is not displayed.

For example, the rest of this sentence moves.

Again, you can use complex string rules, if you want.

```
#only("2-4, 6")[Visible on subslides 2, 3, 4, and 6]
```
Visible on subslides 2, 3, 4, and 6

Everything that works with `#uncover` also works with `#only`.

However, content is completely gone when it is not displayed.

For example, the rest of this sentence moves.

Again, you can use complex string rules, if you want.

```
#only("2-4, 6")[Visible on subslides 2, 3, 4, and 6]
```

Everything that works with `#uncover` also works with `#only`.

However, content is completely gone when it is not displayed.

For example, the rest of this sentence moves.

Again, you can use complex string rules, if you want.

```
#only("2-4, 6")[Visible on subslides 2, 3, 4, and 6]
```
Visible on subslides 2, 3, 4, and 6

You might be tempted to try

```
#only(1)[Ann] #only(2)[Bob] #only(3)[Christopher] likes #only(1)
[chocolate] #only(2)[strawberry] #only(3)[vanilla] ice cream.

Ann

likes chocolate

ice cream.
```

But it is hard to see what piece of text actually changes because everything moves around. Better:

```
#alternatives[Ann][Bob][Christopher] likes
#alternatives[chocolate][strawberry][vanilla] ice cream.
```

Ann   likes chocolate ice cream.

You might be tempted to try

```
#only(1)[Ann] #only(2)[Bob] #only(3)[Christopher] likes #only(1)
[chocolate] #only(2)[strawberry] #only(3)[vanilla] ice cream.
```

Bob

likes strawberry

ice cream.

But it is hard to see what piece of text actually changes because everything moves around. Better:

#alternatives[Ann][Bob][Christopher] likes
#alternatives[chocolate][strawberry][vanilla] ice cream.

Bob   likes strawberry ice cream.

You might be tempted to try

```
#only(1)[Ann] #only(2)[Bob] #only(3)[Christopher] likes #only(1)
[chocolate] #only(2)[strawberry] #only(3)[vanilla] ice cream.
```

Christopher

likes vanilla

ice cream.

But it is hard to see what piece of text actually changes because everything moves around. Better:

```
#alternatives[Ann][Bob][Christopher] likes
#alternatives[chocolate][strawberry][vanilla] ice cream.
```

Christopher likes vanilla   ice cream.

#alternatives is to #only what #one-by-one is to #uncover.

#one-by-one behaves similar to using #pause but you can additionally state when uncovering should start.

```
#one-by-one(start: 2)[one ][by ][one]
```

start can also be omitted, then it starts with the first subside:

```
#one-by-one[one ][by ][one]
```

one

#alternatives is to #only what #one-by-one is to #uncover.

#one-by-one behaves similar to using #pause but you can additionally state when uncovering should start.

```
#one-by-one(start: 2)[one ][by ][one]

one
```

start can also be omitted, then it starts with the first subside:

```
#one-by-one[one ][by ][one]
```

oneby

#alternatives is to #only what #one-by-one is to #uncover.

#one-by-one behaves similar to using #pause but you can additionally state when uncovering should start.

```
#one-by-one(start: 2)[one ][by ][one]
```

oneby

start can also be omitted, then it starts with the first subside:

```
#one-by-one[one ][by ][one]
```

onebyone

#alternatives is to #only what #one-by-one is to #uncover.

#one-by-one behaves similar to using #pause but you can additionally state when uncovering should start.

```
#one-by-one(start: 2)[one ][by ][one]

onebyone
```

start can also be omitted, then it starts with the first subside:

```
#one-by-one[one ][by ][one]
```

onebyone

Sometimes it is convenient to write the different contents to uncover one at a time in subsequent lines.

This comes in especially handy for bullet lists, enumerations, and term lists.

```
#line-by-line(start: 2)[
    - first
    - second
    - third
]
```

-
-
-

`start` is again optional and defaults to 1.

Sometimes it is convenient to write the different contents to uncover one at a time in subsequent lines.

This comes in especially handy for bullet lists, enumerations, and term lists.

```
#line-by-line(start: 2)[
    - first
    - second
    - third
]
```

- first

-

-

`start` is again optional and defaults to 1.

Sometimes it is convenient to write the different contents to uncover one at a time in subsequent lines.

This comes in especially handy for bullet lists, enumerations, and term lists.

```
#line-by-line(start: 2)[
    - first
    - second
    - third
]
```

- first
- second
-

`start` is again optional and defaults to 1.

Sometimes it is convenient to write the different contents to uncover one at a time in subsequent lines.

This comes in especially handy for bullet lists, enumerations, and term lists.

```
#line-by-line(start: 2)[
    - first
    - second
    - third
]
```

- first
- second
- third

`start` is again optional and defaults to 1.

While `#line-by-line` is very convenient syntax-wise, it fails to produce more sophisticated bullet lists, enumerations or term lists. For example, non-tight lists are out of reach.

For that reason, there are `#list-one-by-one`, `#enum-one-by-one` , and `#terms-one-by-one`, respectively.

```
#enum-one-by-one(start: 2,                    i)
tight: false, numbering: "i)")
[first][second][third]                        ii)

                                              iii)
```

Note that, for technical reasons, the bullet points, numbers, or terms are never covered.

`start` is again optional and defaults to 1.

While `#line-by-line` is very convenient syntax-wise, it fails to produce more sophisticated bullet lists, enumerations or term lists. For example, non-tight lists are out of reach.

For that reason, there are `#list-one-by-one`, `#enum-one-by-one` , and `#terms-one-by-one`, respectively.

```
#enum-one-by-one(start: 2,
tight: false, numbering: "i)")
[first][second][third]
```

i) first

ii)

iii)

Note that, for technical reasons, the bullet points, numbers, or terms are never covered.

`start` is again optional and defaults to 1.

While `#line-by-line` is very convenient syntax-wise, it fails to produce more sophisticated bullet lists, enumerations or term lists. For example, non-tight lists are out of reach.

For that reason, there are `#list-one-by-one`, `#enum-one-by-one` , and `#terms-one-by-one`, respectively.

```
#enum-one-by-one(start: 2,              i) first
tight: false, numbering: "i)")
[first][second][third]                  ii) second

                                        iii)
```

Note that, for technical reasons, the bullet points, numbers, or terms are never covered.

`start` is again optional and defaults to 1.

While `#line-by-line` is very convenient syntax-wise, it fails to produce more sophisticated bullet lists, enumerations or term lists. For example, non-tight lists are out of reach.

For that reason, there are `#list-one-by-one`, `#enum-one-by-one` , and `#terms-one-by-one`, respectively.

```
#enum-one-by-one(start: 2,
tight: false, numbering: "i)")
[first][second][third]
```

i) first

ii) second

iii) third

Note that, for technical reasons, the bullet points, numbers, or terms are never covered.

`start` is again optional and defaults to 1.

... is defined by the *theme* of the presentation.

This demo uses the `unipd` theme.

Because of it, the title slide and the decoration on each slide (with section name, short title, slide number etc.) look the way they do.

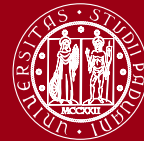Themes can also provide variants, for example ...

... this one!

It's very minimalist and helps the audience focus on an important point.

If you want to create your own design for slides, you can define custom themes!

The book explains how to do so.

Polylux ships a `utils` module with solutions for common tasks in slide building.

You can scale content such that it has a certain height using `#fit-to-height(height, content)`:

This function also allows you to fill the remaining space by using fractions as heights, i.e. `fit-to-height(1fr)[...]`:

Often you want to put different content next to each other. We have the function `#side-by-side` for that:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor.

Why not include an outline?

1. Distributed Algorithms

2. Distributed Algorithms

3. Network Decomposition

4. Computing a Clustering

5. Computing a Clustering

6. Computing a Clustering
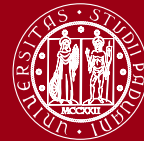
Typst gives us so many cool things[2] . Use them!

---

[2]For example footnotes!

Hopefully you now have some kind of idea what you can do with this template.

Consider giving it a GitHub star or open an issue if you run into bugs or have feature requests.