Efficient Low Diameter Clustering

with strong diameter in the CONGEST model

Christian Micheletti

01/01/1980

Università degli Studi di Padova

Many modern problems apply to **networks of computers**

💡 Distribution ⇒ Parallelism

Idea: we can leverage parallelism to speed up computations

🚨 Distribution ⇒ Remoteness

Bottleneck: remote communication

Our aim is to study algorithms that execute in a distributed environment

W.l.o.g.[1] we adopt a model of execution divided in **communication rounds**

Each round, a node $v \in V$ performs this actions:

1. v **sends** messages to its neighbours;
2. v **receives** messages from its neighbours;

[1] Without loss of generality.

3. v **executes locally** some algorithm (same for each node)

W.l.o.g. rounds are **synchronized**

In Wave a single node starts “waving hello” to its neighbours that, in turn, “wave” to their neighbours

Each communication round can take a significant amount of time to happen

🗝️ Complexity is measured in **rounds**

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

We say **efficient** meaning $O(\text{polylog } n)$, where $n = |V|$

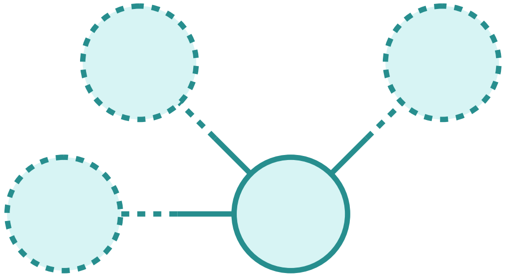
Example: **Maximal Independent Set** (MIS)

Solving it in a centralized model is easily done with a greedy algorithm

def **"Centralized" ≡ "knowing the graph topology"**

🚨 From the perspective of a single node, we don't see the whole topology

What can a node see?



In the **PN-Network** a node only knows that it has some **ports**, each connected with a **different** node

🚨 Each node appears identical to any other

We must break this symmetry

💡 We add unique identifiers to the model

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

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**

Naive MIS

Università degli Studi di Padova

$m \leftarrow m \parallel \perp$
if m = `selected` **then**
    stop (`result: 1`)
SEND m
RECEIVE *messages*
if `selected` $\in$ *messages* **then**
    stop (`result: 0`)
if `round` = $id(v)$ **then**

9: $m \leftarrow$ `selected`

This algorithm runs in $O(n^c)$

We can be way smarter than that

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

The algorithm GATHER-ALL makes all nodes build a local copy of the whole graph

It takes $O(diam(G))$

Then, we can run a deterministic centralized MIS algorithm on each node and output 1 if the node is in the computed MIS

We aim to bind the message sizes to a reasonably small limit

🗝️ In the CONGEST model, messages can only be in the size of $O(\log n)$

To send messages bigger than that, more rounds are needed

Examples:

- Sending a single (or a constant amount of) identifier takes $O(1)$ rounds;
- Sending a *set* of identifiers can take up to $O(n)$ rounds;

For this reason, we can't use Gather-All in the CONGEST model.

There is an algorithm that solves MIS in $O\left(diam(G)\log^2 n\right)$ in CONGEST [1]

🚨 The diameter can be very large: we can only say that $diam(G) \leq n$

A **Network Decomposition** divides a network in colored clusters, where clusters with the same color are not adjacent

- It has diameter d if all of its clusters have diameter at almost d;
- It has c colors.

💡 We can run MIS [1] for each color, in parallel in its clusters and remove the neighbours of the newly added nodes

This algorithm would have complexity $O\left(c \cdot d \log^2 n\right)$

If $c = O(\log n) = d$ then we would have a MIS algorithm in polylogarithmic time

By definition, each color induces a **low diameter clustering**

💡 We can find a low diameter clustering, color them with a color, and repeat on uncolored nodes

🎶 To get a $O(\log n)$ colored decomposition, each clustering has to cluster at least half of the nodes

def A **clustering** $\mathscr{C} \subseteq 2^V$ is any set of **disjoint subsets** of V

def We say it has (strong) **diameter** $d \in \mathbb{N}$ when:

1. No two clusters are adjacent, i.e. $\forall C_1, C_2 \in \mathscr{C}$: $dist_G(C_1, C_2) \geq 2$;

🎶 This means that a clustering *can not* be a partitioning: some node has to be left out

2. Each cluster has at most diameter d, i.e. $\forall C \in \mathscr{C} diam_C(C) \leq d$.

def We say it has **low** diameter instead when:

1. (unchanged);

2. Each cluster has at most diameter in G d, i.e. $\forall C \in \mathscr{C} diam_G(C) \leq d$].

The state of the art before [2] is [3] and [4]

- TODO

The main accomplishment of [2] is to provide an easier algorithm that still runs in polylogarithmic time

Objective: Creating connected components with low diameters

Outline:

- There are $b = O(\log n)$ **phases**, phase i computes a set of **terminals** Q_i;
- Q_i is R-ruling, i.e. $dist_G(Q_i, v) \leq R$ for all $v \in V$.

Objective:

- Each phase removes some nodes: at most $\frac{|V|}{2b}$;

Outline:

- 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;
  - $V' = V_b$

Objective (formalised):

- Each connected component of $G[V']$ contains exactly one terminal
- Moreover, it has polylogarithmic diameter

1. Q_i is R_i-ruling, i.e. $dist_G(Q_i, v) \leq R_i$ for all $v \in V$, with $R_i = i * O(\log n)$
   - Q_0 is 0-ruling, trivially true since $Q_0 = V$;
   - 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

Invariants $\forall i \in [0..b]$

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 forests whose trees have polylogarithmic diameter

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 $\mathrm{id}(r)[i] = 0$ 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`

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

that neighbours a `red` node

Define T_v the (`blue`) subtree rooted at v

Step Outline

🎶 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 (i.e. after $2b^2$ steps)
- Hence at most $\frac{|V|}{2b}$ nodes are lost in each phase
- After the b phases at most $\frac{|V|}{2}$ nodes are deleted

UNIVERSITÀ DEGLI STUDI DI PADOVA

High Level Pseudocode

```
V_0 ← V
Q_0 ← V
for i ∈ 0..b − 1 do
    INIT F_0
    for j ∈ 0..2b^2 − 1 do
        BUILD V_j^propose
        F_{j+1} ← STEP
    V_{i+1} ← V(F_{2b^2})
    Q_{i+1} ← roots(F_{2b^2})
```

$O(\log diam(T_v))$

$2b^2 = O(\log^2 n)$

$b = O(\log n)$

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 $v \in V$

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

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.

The bottom of the slide

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.

A dynamic slide with `pauses`

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:

Università degli Studi di Padova

#uncover: Reserving space

```
#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: Reserving space

```
#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: Reserving space

```
#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]
```

#only: Reserving no space

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.

#only: Reserving no space

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: Reserving no space

```
#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: Reserving no space

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

Visible on subslides 2, 3, 4, and 6

`#only`: Reserving no space

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: Reserving no space

```
#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: Reserving no space

```
#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.

#one-by-one: An alternative for #pause

#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-by-one: An alternative for #pause

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

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

one

#one-by-one: An alternative for #pause

Università degli Studi di Padova

#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

#one-by-one: An alternative for #pause

#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

#one-by-one: An alternative for #pause

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

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

onebyone

#one-by-one: An alternative for #pause

#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

#one-by-one: An alternative for #pause

`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`.

`#line-by-line`: syntactic sugar for `#one-by-one`

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
```

- first
-

```
    - second                    •
    - third
]
```

`start` is again optional and defaults to `1`.

#line-by-line: syntactic sugar for #one-by-one

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
```

- first
- second

```
    - second
    - third
]
```

•

`start` is again optional and defaults to 1.

#line-by-line: syntactic sugar for #one-by-one

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
```

- first
- second

```
    - second
    - third
]
```

- 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,
tight: false, numbering:
"i)")[first][second]
[third]
```

i)

ii)

iii)

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

`start` is again optional and defaults to 1.

`#list-one-by-one` and Co: when `#line-by-line` doesn't suffice

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.

`#list-one-by-one` and Co: when `#line-by-line` doesn't suffice

```
#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.

#list-one-by-one and Co: when #line-by-line doesn’t suffice

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

i) first

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.

`#list-one-by-one` and Co: when `#line-by-line` doesn't suffice

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.

#list-one-by-one and Co: when #line-by-line doesn't suffice

```
#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.

Your own theme?

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

The book explains how to do so.

The `utils` module

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

Fit to height

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)[...]`:

Side by side content

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

Side by side content

eiusmod tempor incididunt ut labore.

eiusmod tempor.

Why not include an outline?

1. Overview
2. Overview
3. Models
4. Models

5. LOCAL Algorithms
6. LOCAL Algorithms
7. CONGEST Algorithms
8. CONGEST Algorithms
9. Clusterings

Università
degli Studi
di Padova

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.