Courcelle's Conjecture, part II: treewidth and cliquewidth

Michał Pilipczuk[†]

Based on a joint work with Mikołaj Bojańczyk and Martin Grohe

[†]Institute of Informatics, University of Warsaw, Poland

Shonan Meeting on Logic and Complexity Theory,
September 18^{th}, 2017

Outline

- **First half**:
  - Continuation of Mikołaj's talk:
    Lifting the pathwidth case to the treewidth case.
- **Second half**:
  - Statement of the conjecture for cliquewidth.
  - Highlight of the proof for linear cliquewidth (with MB and MG).

Part I

from pathwidth to treewidth

First idea

- **Pathwidth case**:

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!
- **Idea**: Use variants of Simon's factorization for trees.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!
- **Idea**: Use variants of Simon's factorization for trees.
  - Generalization to trees due to Colcombet.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!
- **Idea**: Use variants of Simon's factorization for trees.
  - Generalization to trees due to Colcombet.
  - **Outcome**: Completely does not work.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!
- **Idea**: Use variants of Simon's factorization for trees.
  - Generalization to trees due to Colcombet.
  - **Outcome**: Completely does not work.
  - **Reason**: Focus on paths in trees, not on (multi-)contexts.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!
- **Idea**: Use variants of Simon's factorization for trees.
  - Generalization to trees due to Colcombet.
  - **Outcome**: Completely does not work.
  - **Reason**: Focus on paths in trees, not on (multi-)contexts.
- **Final approach**: Reduce the treewidth case to the pathwidth case.

First idea

- **Pathwidth case**:
  - Decomposition as a word over a finite alphabet of operations.
  - Design **abstraction** of a subword as a finite info about it.
  - Use **Simon's factorization theorem** to get a bounded depth factorization of the word that respects abstractions.
  - Combine transductions bottom-up on the factorization.
  - **Key**: Efficient composition of transductions in the idempotent nodes.
  - Turns out to be a really robust approach!
- **Idea**: Use variants of Simon's factorization for trees.
  - Generalization to trees due to Colcombet.
  - **Outcome**: Completely does not work.
  - **Reason**: Focus on paths in trees, not on (multi-)contexts.
- **Final approach**: Reduce the treewidth case to the pathwidth case.
  - **Caveat**: Not a robust approach.

Guidance systems: intuition

- **Problem**: We cannot quantify over sets of k-tuples of vertices.

Guidance systems: intuition

- **Problem**: We cannot quantify over sets of k-tuples of vertices.
- But we can quantify over sets of single vertices.

Guidance systems: intuition

- **Problem**: We cannot quantify over sets of k-tuples of vertices.
- But we can quantify over sets of single vertices.
- **Idea**: Encode interesting k-tuples in single vertices so that given a vertex u, the k-tuple associated with u can be recovered in MSO.

Guidance systems: intuition

- **Problem**: We cannot quantify over sets of k-tuples of vertices.
- But we can quantify over sets of single vertices.
- **Idea**: Encode interesting k-tuples in single vertices so that given a vertex u, the k-tuple associated with u can be recovered in MSO.
  - Quantification over k-tuples $\rightsquigarrow$ Quantification over single vertices

Guidance systems: intuition

- **Problem**: We cannot quantify over sets of k-tuples of vertices.
- But we can quantify over sets of single vertices.
- **Idea**: Encode interesting k-tuples in single vertices so that given a vertex u, the k-tuple associated with u can be recovered in MSO.
  - Quantification over k-tuples $\rightsquigarrow$ Quantification over single vertices
  - **Note**: Encoding can use some (nondeterministically guessed) coloring of the graph.

Guidance systems: intuition

- **Problem**: We cannot quantify over sets of k-tuples of vertices.
- But we can quantify over sets of single vertices.
- **Idea**: Encode interesting k-tuples in single vertices so that given a vertex u, the k-tuple associated with u can be recovered in MSO.
  - Quantification over k-tuples $\rightsquigarrow$ Quantification over single vertices
  - **Note**: Encoding can use some (nondeterministically guessed) coloring of the graph.
- **Guidance system**:
  Combinatorial object that provides this functionality.

Guidance systems

Guidance system

A **guidance system** Λ in a graph G is a tuple of rooted forests

$$(F_1, F_2, \ldots, F_k)$$

where $V(F_i) = V(G)$ and $E(F_i) \subseteq E(G)$ for each i.

Guidance systems

Guidance system

A **guidance system** Λ in a graph G is a tuple of rooted forests

$$(F_1, F_2, \ldots, F_k)$$

where $V(F_i) = V(G)$ and $E(F_i) \subseteq E(G)$ for each i.

- **Note**: Forests may overlap!

Guidance systems

Guidance system

A **guidance system** Λ in a graph G is a tuple of rooted forests

$$(F_1, F_2, \ldots, F_k)$$

where $V(F_i) = V(G)$ and $E(F_i) \subseteq E(G)$ for each i.

- **Note**: Forests may overlap!
- We think of each tree as oriented towards its root.

Guidance systems

Guidance system

A **guidance system** Λ in a graph G is a tuple of rooted forests

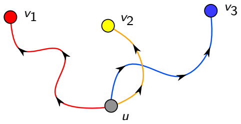
$$(F_1, F_2, \ldots, F_k)$$

where $V(F_i) = V(G)$ and $E(F_i) \subseteq E(G)$ for each i.

- **Note**: Forests may overlap!
- We think of each tree as oriented towards its root.
- For each $u \in V(G)$, define k-tuple $\Lambda(u)$ as

$$\Lambda(u) = (v_1, v_2, \ldots, v_k),$$

where v_i is the root of the tree of F_i that contains u.



Guidance systems

Guidance system

A **guidance system** Λ in a graph G is a tuple of rooted forests

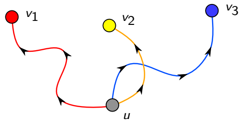
$$(F_1, F_2, \ldots, F_k)$$

where $V(F_i) = V(G)$ and $E(F_i) \subseteq E(G)$ for each i.

- **Note**: Forests may overlap!
- We think of each tree as oriented towards its root.
- For each $u \in V(G)$, define k-tuple $\Lambda(u)$ as

$$\Lambda(u) = (v_1, v_2, \ldots, v_k),$$

where v_i is the root of the tree of F_i that contains u.



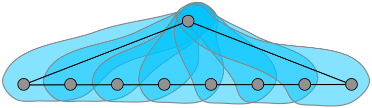
- A vertex subset X is **captured** by Λ if $X \subseteq \Lambda(u)$ for some vertex u.

Capturing tree decompositions

- Λ **captures** a tree decomposition iff Λ captures all its bags.

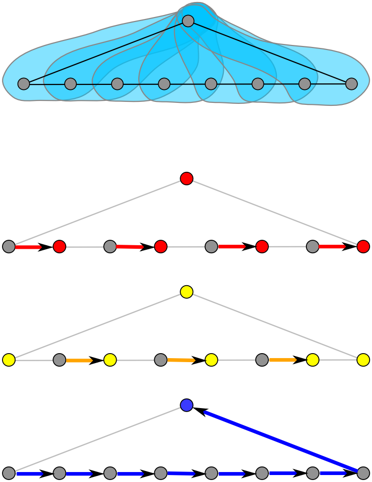
Capturing tree decompositions

- Λ **captures** a tree decomposition iff Λ captures all its bags.



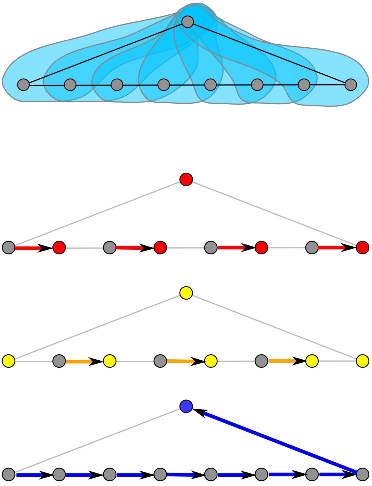
Capturing tree decompositions

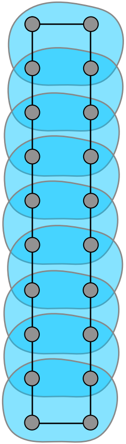
- Λ **captures** a tree decomposition iff Λ captures all its bags.



Capturing tree decompositions

- Λ **captures** a tree decomposition iff Λ captures all its bags.





Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.

Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.
- **Obs**: To guess a guidance system of size ℓ, one may quantify existentially ℓ subsets of edges and ℓ subsets of vertices.

Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.
- **Obs**: To guess a guidance system of size ℓ, one may quantify existentially ℓ subsets of edges and ℓ subsets of vertices.

Theorem

For every graph G of pathwidth $\leqslant k$, some tree decomposition of G is captured by a guidance system of size $f(k)$.

Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.
- **Obs**: To guess a guidance system of size ℓ, one may quantify existentially ℓ subsets of edges and ℓ subsets of vertices.

Theorem

For every graph G of pathwidth $\leqslant k$, some tree decomposition of G is captured by a guidance system of size $f(k)$.

- **Original proof**:

Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.
- **Obs**: To guess a guidance system of size ℓ, one may quantify existentially ℓ subsets of edges and ℓ subsets of vertices.

Theorem

For every graph G of pathwidth $\leqslant k$, some tree decomposition of G is captured by a guidance system of size $f(k)$.

- **Original proof**:
  - This statement is proved using Simon's factorization.

Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.
- **Obs**: To guess a guidance system of size ℓ, one may quantify existentially ℓ subsets of edges and ℓ subsets of vertices.

Theorem

For every graph G of pathwidth $\leqslant k$, some tree decomposition of G is captured by a guidance system of size $f(k)$.

- **Original proof**:
  - This statement is proved using Simon's factorization.
  - Then guess a guidance system and piece together a decomposition.

Guidance systems and MSO

- **Intuition**: Tree decompositions captured by small guidance systems are exactly those guessable in MSO.
- **Obs**: To guess a guidance system of size ℓ, one may quantify existentially ℓ subsets of edges and ℓ subsets of vertices.

Theorem

For every graph G of pathwidth $\leqslant k$, some tree decomposition of G is captured by a guidance system of size $f(k)$.

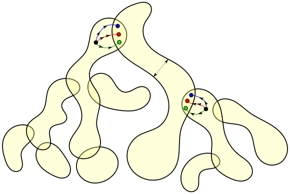
- **Original proof**:
  - This statement is proved using Simon's factorization.
  - Then guess a guidance system and piece together a decomposition.
- **Intuition**: Families of subsets captured by small guidance systems can be efficiently guessed in MSO.

Decomposition into low-pathwidth parts

Decomposition into low-pathwidth parts

Every graph G of treewidth k admits a tree decomposition s such that

- the torso of every bag of s has pathwidth bounded by $2k + 1$; and
- the adhesions of s are captured by a guid. system of size $4k^3 + 2k$.



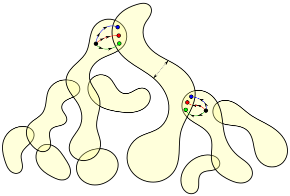
Decomposition into low-pathwidth parts

Decomposition into low-pathwidth parts

Every graph G of treewidth k admits a tree decomposition s such that

- the torso of every bag of s has pathwidth bounded by $2k + 1$; and
- the adhesions of s are captured by a guid. system of size $4k^3 + 2k$.

- **Torso** of S in G: take $G[S]$ and turn the neighbors of every conn. component of $G - S$ into a clique.



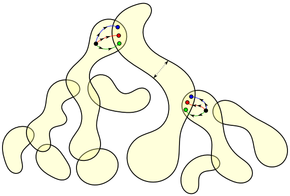
Decomposition into low-pathwidth parts

Decomposition into low-pathwidth parts

Every graph G of treewidth k admits a tree decomposition s such that

- the torso of every bag of s has pathwidth bounded by $2k+1$; and
- the adhesions of s are captured by a guid. system of size $4k^3+2k$.

- **Torso** of S in G: take $G[S]$ and turn the neighbors of every conn. component of $G-S$ into a clique.
- Having this, the proof follows easily.



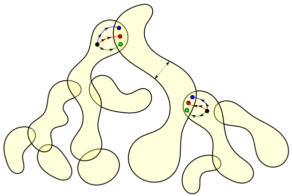
Decomposition into low-pathwidth parts

Decomposition into low-pathwidth parts

Every graph G of treewidth k admits a tree decomposition s such that

- the torso of every bag of s has pathwidth bounded by $2k + 1$; and
- the adhesions of s are captured by a guid. system of size $4k^3 + 2k$.

- **Torso** of S in G: take $G[S]$ and turn the neighbors of every conn. component of $G - S$ into a clique.
- Having this, the proof follows easily.
  - Construct the decomposition s by guessing a guidance system capturing its adhesions.



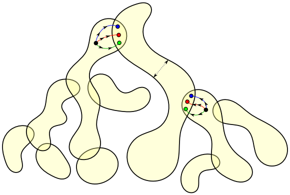
Decomposition into low-pathwidth parts

Decomposition into low-pathwidth parts

Every graph G of treewidth k admits a tree decomposition s such that

- the torso of every bag of s has pathwidth bounded by $2k+1$; and
- the adhesions of s are captured by a guid. system of size $4k^3+2k$.

- **Torso** of S in G: take $G[S]$ and turn the neighbors of every conn. component of $G-S$ into a clique.
- Having this, the proof follows easily.
  - Construct the decomposition s by guessing a guidance system capturing its adhesions.
  - Apply the transduction for pathwidth $\leqslant 2k+1$ on each bag.



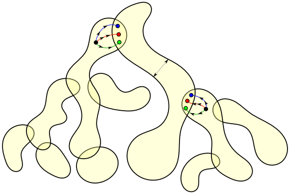
Decomposition into low-pathwidth parts

Decomposition into low-pathwidth parts

Every graph G of treewidth k admits a tree decomposition s such that

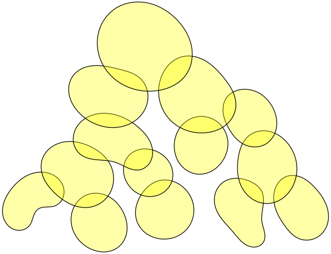
- the torso of every bag of s has pathwidth bounded by $2k+1$; and
- the adhesions of s are captured by a guid. system of size $4k^3+2k$.

- **Torso** of S in G: take $G[S]$ and turn the neighbors of every conn. component of $G-S$ into a clique.
- Having this, the proof follows easily.
  - Construct the decomposition s by guessing a guidance system capturing its adhesions.
  - Apply the transduction for pathwidth $\leqslant 2k+1$ on each bag.
  - Combine all the obtained decompositions along s.



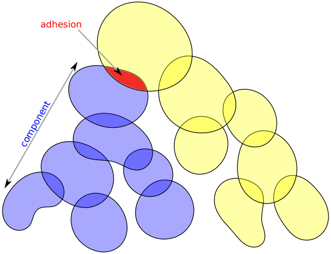
Requests

- Fix some tree decomposition t_0 of width k.



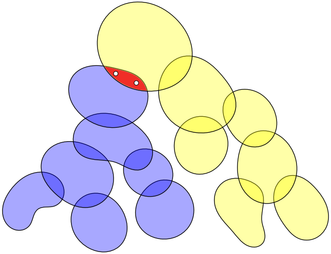
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.



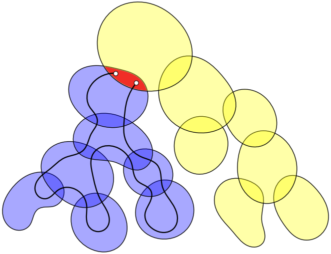
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.
- **Request**: Pair of vertices (u, v) from the adhesion.



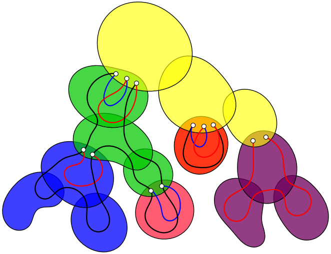
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.
- **Request**: Pair of vertices (u, v) from the adhesion.
- **Realization**: u-v path through vertices in the component below.



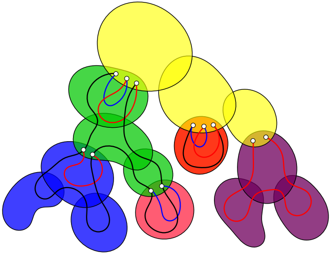
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.
- **Request**: Pair of vertices (u, v) from the adhesion.
- **Realization**: u-v path through vertices in the component below.
- **Goal**: Partition of t_0 into subtrees so that:



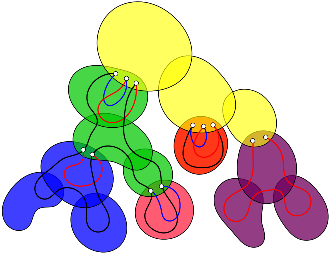
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.
- **Request**: Pair of vertices (u, v) from the adhesion.
- **Realization**: u-v path through vertices in the component below.
- **Goal**: Partition of t_0 into subtrees so that:
  - The torso of the union of bags in each subtree has bnd pathwidth.



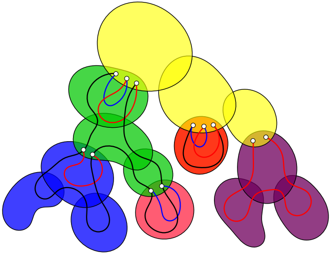
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.
- **Request**: Pair of vertices (u, v) from the adhesion.
- **Realization**: u-v path through vertices in the component below.
- **Goal**: Partition of t_0 into subtrees so that:
  - The torso of the union of bags in each subtree has bnd pathwidth.
  - We can realize all request in adhesions between pieces using a path system that can be colored with a bounded number of colors.



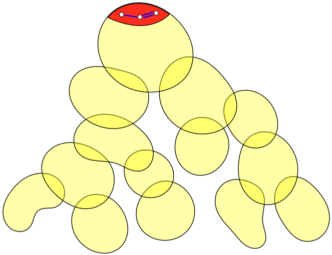
Requests

- Fix some tree decomposition t_0 of width k.
  - **Wlog**: the component at each node is connected its neighborhood is exactly the whole adhesion.
- **Request**: Pair of vertices (u, v) from the adhesion.
- **Realization**: u-v path through vertices in the component below.
- **Goal**: Partition of t_0 into subtrees so that:
  - The torso of the union of bags in each subtree has bnd pathwidth.
  - We can realize all request in adhesions between pieces using a path system that can be colored with a bounded number of colors.
- **Idea**: Extract pieces by a top-down induction.



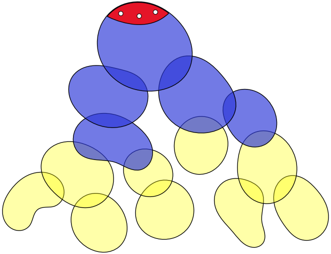
Induction

- **Assumptions**:
  - A subtree t of t_0, with top adhesion S.
  - A multiset $\mathcal{R}$ of $\leqslant p(k)$ requests on pairs in S.



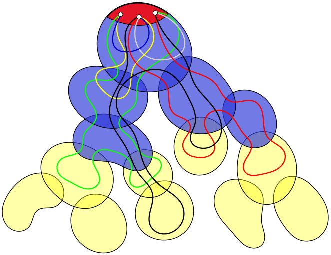
Induction

- **Assumptions**:
  - A subtree t of t_0, with top adhesion S.
  - A multiset $\mathcal{R}$ of $\leqslant p(k)$ requests on pairs in S.
- **Goal**: A prefix X of t such that
  - the torso of the union of bags in X has bnd pathwidth; and



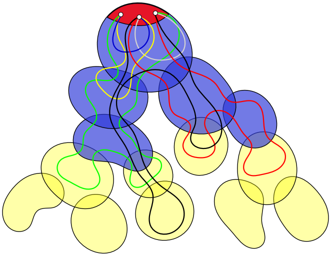
Induction

- **Assumptions**:
  - A subtree t of t_0, with top adhesion S.
  - A multiset $\mathcal{R}$ of $\leqslant p(k)$ requests on pairs in S.
- **Goal**: A prefix X of t such that
  - the torso of the union of bags in X has bnd pathwidth; and
  - requests from $\mathcal{R} \cup \binom{S}{2}$ can be realized with $\leqslant p(k)$ requests imposed on every component below X.

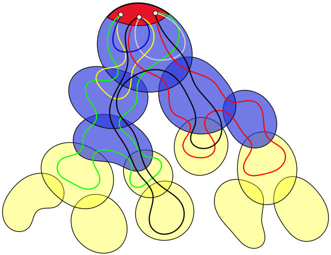


Induction

- **Assumptions**:
  - A subtree t of t_0, with top adhesion S.
  - A multiset $\mathcal{R}$ of $\leqslant p(k)$ requests on pairs in S.
- **Goal**: A prefix X of t such that
  - the torso of the union of bags in X has bnd pathwidth; and
  - requests from $\mathcal{R} \cup \binom{S}{2}$ can be realized with $\leqslant p(k)$ requests imposed on every component below X.
- **Goal achieved** $\Rightarrow$
  Paths can be colored greedily top-down with $p(k) + \binom{k}{2}$ colors.

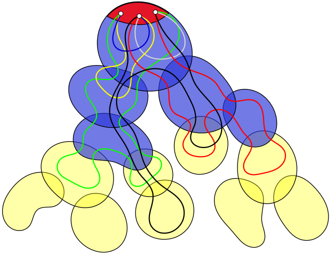


- **Assumptions**:
  - A subtree t of t_0, with top adhesion S.
  - A multiset $\mathcal{R}$ of $\leqslant p(k)$ requests on pairs in S.
- **Goal**: A prefix X of t such that
  - the torso of the union of bags in X has bnd pathwidth; and
  - requests from $\mathcal{R} \cup \binom{S}{2}$ can be realized with $\leqslant p(k)$ requests imposed on every component below X.
- **Goal achieved** $\Rightarrow$
  Paths can be colored greedily top-down with $p(k) + \binom{k}{2}$ colors.
  - Every path in conflict with $\leqslant p(k) + \binom{k}{2} - 1$ other paths.



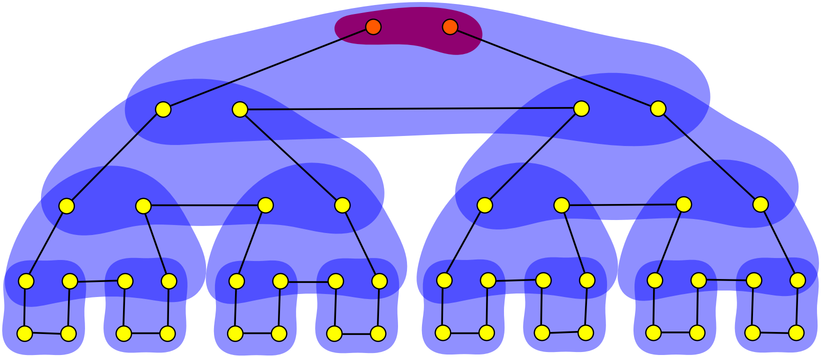
Induction

- **Assumptions**:
  - A subtree t of t_0, with top adhesion S.
  - A multiset $\mathcal{R}$ of $\leqslant p(k)$ requests on pairs in S.
- **Goal**: A prefix X of t such that
  - the torso of the union of bags in X has bnd pathwidth; and
  - requests from $\mathcal{R} \cup \binom{S}{2}$ can be realized with $\leqslant p(k)$ requests imposed on every component below X.
- **Goal achieved** $\Rightarrow$
  Paths can be colored greedily top-down with $p(k) + \binom{k}{2}$ colors.
  - Every path in conflict with $\leqslant p(k) + \binom{k}{2} - 1$ other paths.
  - **Caveat**: Not quite true, needs a slightly different choice of kings.



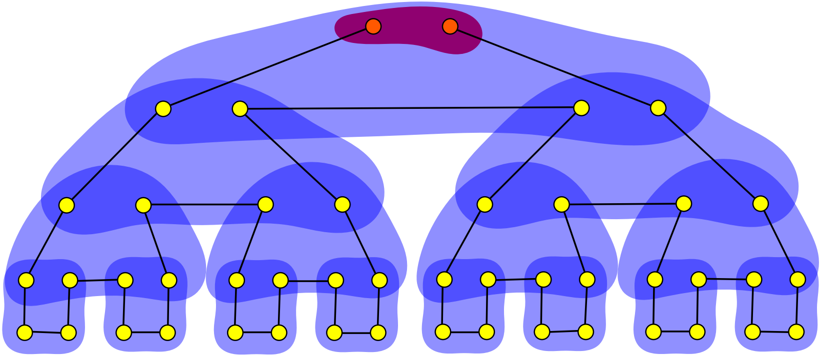
Motivating example

- Consider the following tree decomposition of a path, with $p(k)$ requests on the top vertices.



Motivating example

- Consider the following tree decomposition of a path, with $p(k)$ requests on the top vertices.



- In this case, X should be the whole t.

Strategy for constructing X

- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.

Strategy for constructing X

- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.
  - **Anyhow**: Take any path realizing the request (exists by connectivity) and replace visits of components at children by requests.

Strategy for constructing X

- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.
    - **Anyhow**: Take any path realizing the request (exists by connectivity) and replace visits of components at children by requests.
- **From now on**: $\mathcal{R}' = \mathcal{R} \cup \binom{S}{2}$ has more than $p(k)$ requests.

Strategy for constructing X

- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.
  - **Anyhow**: Take any path realizing the request (exists by connectivity) and replace visits of components at children by requests.
- **From now on**: $\mathcal{R}' = \mathcal{R} \cup \binom{S}{2}$ has more than $p(k)$ requests.
- **Key idea**: Let (u, v) be the request with highest multiplicity.

Strategy for constructing X

- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.
  - **Anyhow**: Take any path realizing the request (exists by connectivity) and replace visits of components at children by requests.
- **From now on**: $\mathcal{R}' = \mathcal{R} \cup \binom{S}{2}$ has more than $p(k)$ requests.
- **Key idea**: Let (u, v) be the request with highest multiplicity.
- Say (u, v) is requested $\ell > p(k)/\binom{k}{2}$ times.

Strategy for constructing X

- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.
  - **Anyhow**: Take any path realizing the request (exists by connectivity) and replace visits of components at children by requests.
- **From now on**: $\mathcal{R}' = \mathcal{R} \cup \binom{S}{2}$ has more than $p(k)$ requests.
- **Key idea**: Let (u, v) be the request with highest multiplicity.
- Say (u, v) is requested $\ell > p(k)/\binom{k}{2}$ times.
- **Goal**: Find X so that (u, v)-requests can be routed in such a manner that each component below X gets load $\leqslant \ell/2$ from them.

Strategy for constructing X

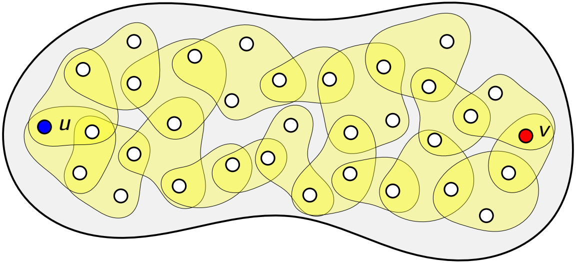
- If $|\mathcal{R}| \leqslant p(k) - |\binom{S}{2}|$, then we can fix $X = \{\mathrm{root}\}$, route requests anyhow, and apply the induction assumption in children.
  - **Anyhow**: Take any path realizing the request (exists by connectivity) and replace visits of components at children by requests.
- **From now on**: $\mathcal{R}' = \mathcal{R} \cup \binom{S}{2}$ has more than $p(k)$ requests.
- **Key idea**: Let (u, v) be the request with highest multiplicity.
- Say (u, v) is requested $\ell > p(k)/\binom{k}{2}$ times.
- **Goal**: Find X so that (u, v)-requests can be routed in such a manner that each component below X gets load $\leqslant \ell/2$ from them.
- If achieved, then remaining requests are routed arbitrarily, and

$$p(k) + \binom{k}{2} - \frac{p(k)}{2\binom{k}{2}} < p(k)$$

for a quartic polynomial $p(k)$.

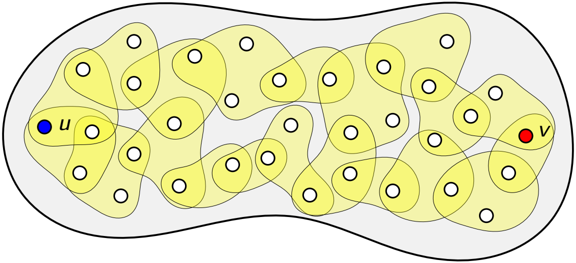
Constructing X

- Construct the following hypergraph H:
  - The vertex set is the root bag.
  - Each child node gives rise to a hyperedge equal to the adhesion.



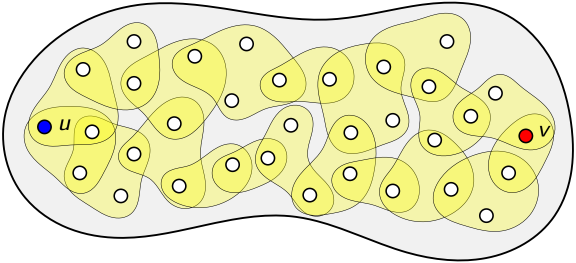
Constructing X

- Construct the following hypergraph H:
  - The vertex set is the root bag.
  - Each child node gives rise to a hyperedge equal to the adhesion.
- **Note**: Edges of the graph are in leaves.



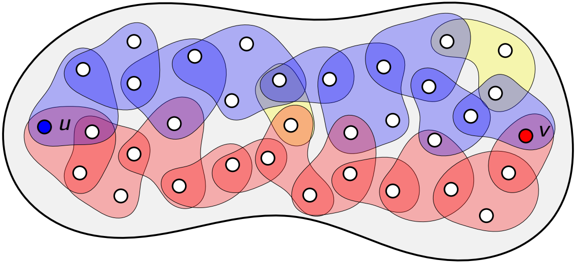
Constructing X

- Construct the following hypergraph H:
  - The vertex set is the root bag.
  - Each child node gives rise to a hyperedge equal to the adhesion.
- **Note**: Edges of the graph are in leaves.
- **Paths in** H: Alternating sequences of vertices and hyperedges.



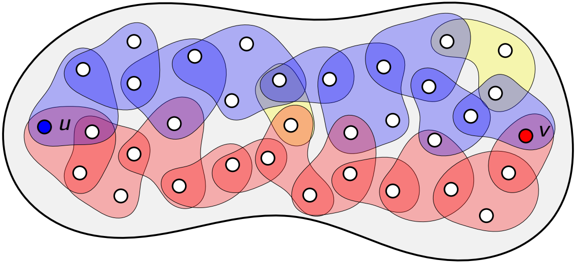
Constructing X

- Construct the following hypergraph H:
  - The vertex set is the root bag.
  - Each child node gives rise to a hyperedge equal to the adhesion.
- **Note**: Edges of the graph are in leaves.
- **Paths in** H: Alternating sequences of vertices and hyperedges.
- **Flow-cut duality**: If there is no hyperedge cutting u from v, then there are two hyperedge-disjoint paths from u to v.



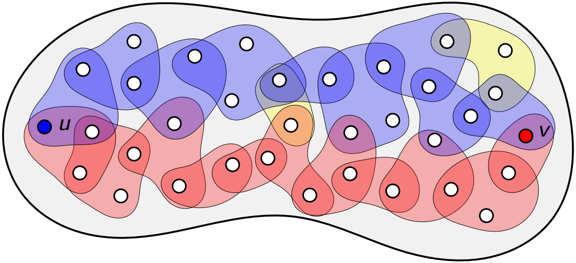
Constructing X

- Construct the following hypergraph H:
  - The vertex set is the root bag.
  - Each child node gives rise to a hyperedge equal to the adhesion.
- **Note**: Edges of the graph are in leaves.
- **Paths in** H: Alternating sequences of vertices and hyperedges.
- **Flow-cut duality**: If there is no hyperedge cutting u from v, then there are two hyperedge-disjoint paths from u to v.
- Then we can split the (u, v)-requests equally between them.



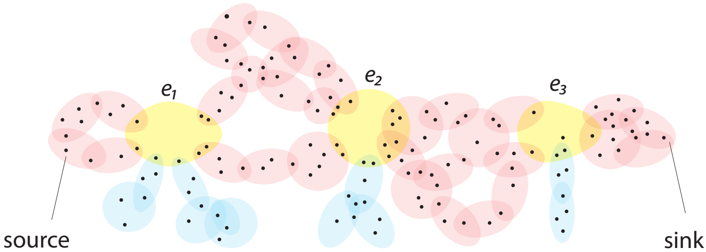
Constructing X

- Construct the following hypergraph H:
  - The vertex set is the root bag.
  - Each child node gives rise to a hyperedge equal to the adhesion.
- **Note**: Edges of the graph are in leaves.
- **Paths in** H: Alternating sequences of vertices and hyperedges.
- **Flow-cut duality**: If there is no hyperedge cutting u from v, then there are two hyperedge-disjoint paths from u to v.
- Then we can split the (u, v)-requests equally between them.
- **Ergo**: If no cutedge, then again $X = \{\mathrm{root}\}$ does the job.



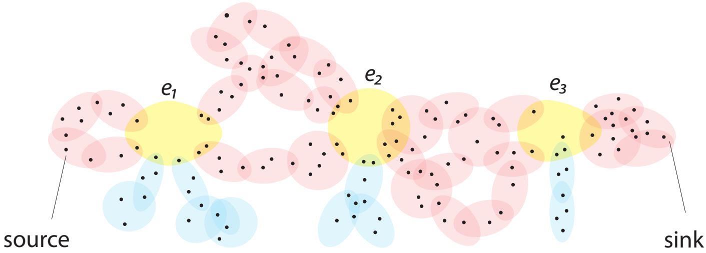
Constructing X

- Otherwise, there is a **sequence** of cutedges.



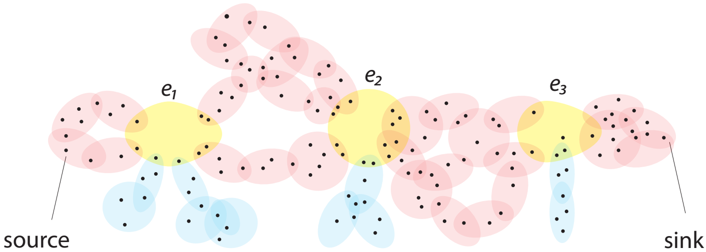
Constructing X

- Otherwise, there is a **sequence** of cutedges.
  - **Observation**: Between every two consecutive cutedges, there are two hyperedge-disjoint paths.



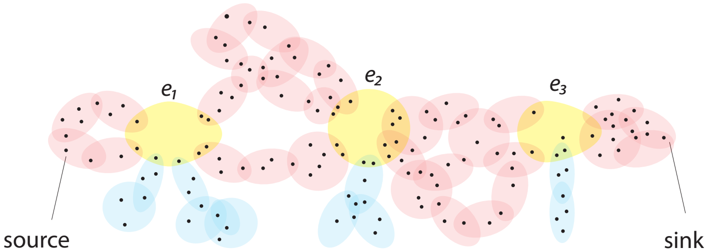
Constructing X

- Otherwise, there is a **sequence** of cutedges.
  - **Observation**: Between every two consecutive cutedges, there are two hyperedge-disjoint paths.
  - **Ergo**: We can have load $\ell/2$ on all hyperedges apart from cutedges.



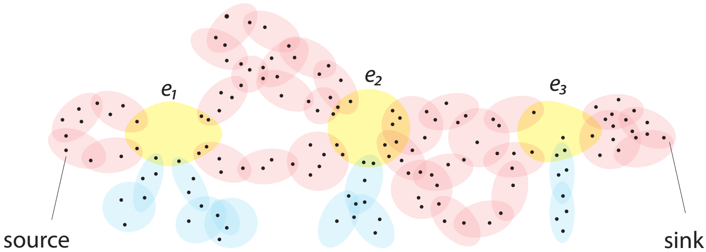
Constructing X

- Otherwise, there is a **sequence** of cutedges.
  - **Observation**: Between every two consecutive cutedges, there are two hyperedge-disjoint paths.
  - **Ergo**: We can have load $\ell/2$ on all hyperedges apart from cutedges.
- **Construction**: Extend X to the roots of those subtrees that correspond to cutedges, and recurse.



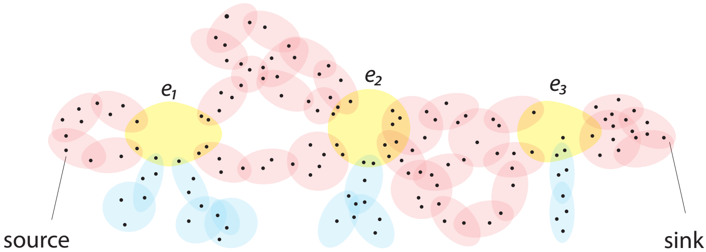
Constructing X

- Otherwise, there is a **sequence** of cutedges.
  - **Observation**: Between every two consecutive cutedges, there are two hyperedge-disjoint paths.
  - **Ergo**: We can have load $\ell/2$ on all hyperedges apart from cutedges.
- **Construction**: Extend X to the roots of those subtrees that correspond to cutedges, and recurse.
- **Observation**: After unraveling all the recursive calls and examining the torso of $\bigcup X$, we see one long **path** decomposition.



Constructing X

- Otherwise, there is a **sequence** of cutedges.
  - **Observation**: Between every two consecutive cutedges, there are two hyperedge-disjoint paths.
  - **Ergo**: We can have load $\ell/2$ on all hyperedges apart from cutedges.
- **Construction**: Extend X to the roots of those subtrees that correspond to cutedges, and recurse.
- **Observation**: After unraveling all the recursive calls and examining the torso of $\bigcup X$, we see one long **path** decomposition.
  - This is exactly what happens in the motivating example.



Part II

(linear) cliquewidth

Cliquewidth

- **Treewidth algebra**:

Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.

Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.

Cliquewidth

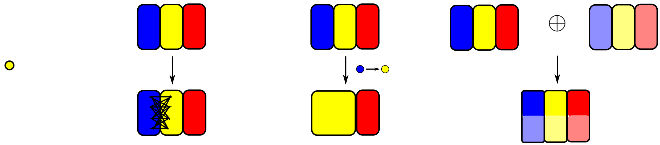
- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.
- **Cliquewidth algebra**:

Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.
- **Cliquewidth algebra**:
  - **Support**: k-colored graphs, colors from 1 to k.

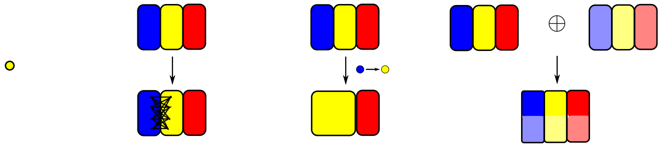
Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.
- **Cliquewidth algebra**:
  - **Support**: k-colored graphs, colors from 1 to k.
  - **Operations**:
    - A single **Vertex** of color i.
    - **Connect** all vertices of colors i and j by making them adjacent.
    - **Recolor** all vertices of color i to color j.
    - **Disjoint Union** of two k-colored graphs.



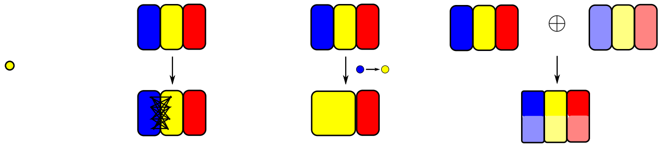
Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.
- **Cliquewidth algebra**:
  - **Support**: k-colored graphs, colors from 1 to k.
  - **Operations**:
    - A single **Vertex** of color i.
    - **Connect** all vertices of colors i and j by making them adjacent.
    - **Recolor** all vertices of color i to color j.
    - **Disjoint Union** of two k-colored graphs.
  - **Cliquewidth**: Min. number of colors needed to construct a graph.



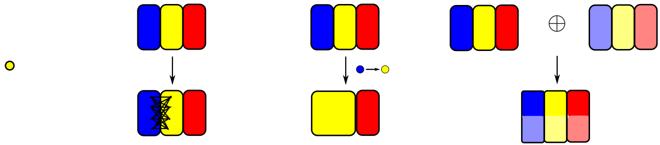
Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.
- **Cliquewidth algebra**:
  - **Support**: k-colored graphs, colors from 1 to k.
  - **Operations**:
    - A single **Vertex** of color i.
    - **Connect** all vertices of colors i and j by making them adjacent.
    - **Recolor** all vertices of color i to color j.
    - **Disjoint Union** of two k-colored graphs.
  - **Cliquewidth**: Min. number of colors needed to construct a graph.
  - **Linear cliquewidth**: Vertices have to be added one by one.



Cliquewidth

- **Treewidth algebra**:
  - **Support**: Graphs with $\leqslant k$ interfaces, numbered from 1 to k.
  - **Operations**: introduce, forget, join, leaf.
- **Cliquewidth algebra**:
  - **Support**: k-colored graphs, colors from 1 to k.
  - **Operations**:
    - A single **Vertex** of color i.
    - **Connect** all vertices of colors i and j by making them adjacent.
    - **Recolor** all vertices of color i to color j.
    - **Disjoint Union** of two k-colored graphs.
  - **Cliquewidth**: Min. number of colors needed to construct a graph.
  - **Linear cliquewidth**: Vertices have to be added one by one.
    - **Add Vertex** instead of Vertex, Connect, and Disjoint Union.



VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.

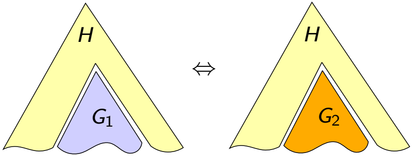
VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:

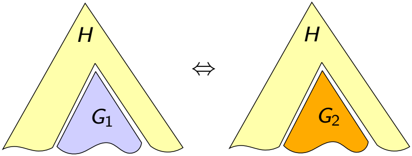
VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:
  - k-colored graphs G_1 and G_2 are L-equivalent if for every context H,

$$H \circ G_1 \in L \;\Leftrightarrow\; H \circ G_2 \in L.$$



- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:
  - k-colored graphs G_1 and G_2 are L-equivalent if for every context H,
$$H \circ G_1 \in L \;\Leftrightarrow\; H \circ G_2 \in L.$$
  - L is **recognizable** if for each k this relation has finite index.



VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:
  - k-colored graphs G_1 and G_2 are L-equivalent if for every context H,

$$H \circ G_1 \in L \iff H \circ G_2 \in L.$$

  - L is **recognizable** if for each k this relation has finite index.
- **Homomorphism** definition:

VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:
  - k-colored graphs G_1 and G_2 are L-equivalent if for every context H,

$$H \circ G_1 \in L \;\Leftrightarrow\; H \circ G_2 \in L.$$

  - L is **recognizable** if for each k this relation has finite index.
- **Homomorphism** definition:
  - Consider a homomorphism h from the algebra of k-colored graphs to some finite algebra $\mathbb{A}$ over the same operations.

$$\mathbb{G}_k \xrightarrow{\quad h \quad} \mathbb{A}$$

VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:
  - k-colored graphs G_1 and G_2 are L-equivalent if for every context H,

$$H \circ G_1 \in L \;\Leftrightarrow\; H \circ G_2 \in L.$$

  - L is **recognizable** if for each k this relation has finite index.
- **Homomorphism** definition:
  - Consider a homomorphism h from the algebra of k-colored graphs to some finite algebra $\mathbb{A}$ over the same operations.
  - Homomorphism h **recognizes** L if $L = h^{-1}(S)$ for some $S \subseteq \mathbb{A}$.

$$\begin{array}{ccc} \mathbb{G}_k & \xrightarrow{\quad h \quad} & \mathbb{A} \\ L & \xleftarrow{\quad h^{-1} \quad} & S \end{array}$$

VR-recognizability

- We define recognizability for the cliquewidth algebra similarly as for the treewidth algebra.
- **Myhill-Nerode** relation for a graph language L:
  - k-colored graphs G_1 and G_2 are L-equivalent if for every context H,

$$H \circ G_1 \in L \;\Leftrightarrow\; H \circ G_2 \in L.$$

  - L is **recognizable** if for each k this relation has finite index.
- **Homomorphism** definition:
  - Consider a homomorphism h from the algebra of k-colored graphs to some finite algebra $\mathbb{A}$ over the same operations.
  - Homomorphism h **recognizes** L if $L = h^{-1}(S)$ for some $S \subseteq \mathbb{A}$.
  - L is **recognizable** if for each k it is recognized as above.

$$\begin{array}{ccc} \mathbb{G}_k & \xrightarrow{\quad h \quad} & \mathbb{A} \\ L & \xleftarrow{\quad h^{-1} \quad} & S \end{array}$$

Conjecture for cliquewidth

Conjecture

Let L be a language of graphs of cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

Conjecture for cliquewidth

Conjecture

Let L be a language of graphs of cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

($\Rightarrow$) Follows as in the treewidth case.

Conjecture for cliquewidth

Conjecture

Let L be a language of graphs of cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

($\Rightarrow$) Follows as in the treewidth case.
($\Leftarrow$) The same issue as in the treewidth case.

Conjecture for cliquewidth

Conjecture

Let L be a language of graphs of cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

($\Rightarrow$) Follows as in the treewidth case.
($\Leftarrow$) The same issue as in the treewidth case.

Conjecture

For each $k \in \mathbb{N}$ there is an MSO_1 transduction that given a graph of cliquewidth at most k outputs some its clique decomposition.

Conjecture for cliquewidth

Conjecture

Let L be a language of graphs of cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

($\Rightarrow$) Follows as in the treewidth case.
($\Leftarrow$) The same issue as in the treewidth case.

Conjecture

For each $k \in \mathbb{N}$ there is an MSO_1 transduction that given a graph of cliquewidth at most k outputs some its clique decomposition.

Theorem [BGP,17+]

For each $k \in \mathbb{N}$ there is an MSO_1 transduction that given a graph of **linear** cliquewidth at most k outputs some its clique decomposition.

Conjecture for cliquewidth

Conjecture

Let L be a language of graphs of cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

($\Rightarrow$) Follows as in the treewidth case.
($\Leftarrow$) The same issue as in the treewidth case.

Conjecture

For each $k \in \mathbb{N}$ there is an MSO_1 transduction that given a graph of cliquewidth at most k outputs some its clique decomposition.

Theorem [BGP,17+]

For each $k \in \mathbb{N}$ there is an MSO_1 transduction that given a graph of **linear** cliquewidth at most k outputs some its clique decomposition.

Corollary [BGP,17+]

Let L be a language of graphs of **linear** cliquewidth $\leqslant k$, for some $k \in \mathbb{N}$. Then L is definable in CMSO_1 iff it is recognizable.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.
  - Combine transductions by a bottom-up induction.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.
  - Combine transductions by a bottom-up induction.
  - **Key**: Implement binary and idempotent nodes.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.
  - Combine transductions by a bottom-up induction.
  - **Key**: Implement binary and idempotent nodes.
- **Message**:

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.
  - Combine transductions by a bottom-up induction.
  - **Key**: Implement binary and idempotent nodes.
- **Message**:
  - The plan above can be implemented.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.
  - Combine transductions by a bottom-up induction.
  - **Key**: Implement binary and idempotent nodes.
- **Message**:
  - The plan above can be implemented.
  - **Far** more technical details than in the pathwidth case.

Proof strategy

- The **definable cliquewidth** of a graph is the minimum size of an MSO transduction that constructs some its clique decomposition.
- **Goal**: Def. cliquewidth is bounded by a function of lin. cliquewidth.
- **Strategy**:
  - View linear clique decomposition as a word over instructions.
  - Define bounded-size **abstraction** for subwords of instructions, endowed with structure of a semigroup.
  - Construct Simon's factorization of the linear clique decomposition.
  - Combine transductions by a bottom-up induction.
  - **Key**: Implement binary and idempotent nodes.
- **Message**:
  - The plan above can be implemented.
  - **Far** more technical details than in the pathwidth case.
  - Lack of combinatorial abstraction is a nuisance.

Instructions and derivations

- A linear cw decomposition of width k is a word over instructions:

Instructions and derivations

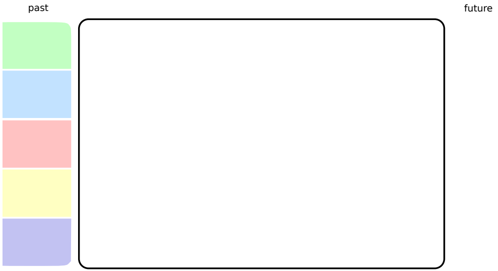
- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi\colon [k] \to [k]$.

Instructions and derivations

- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi\colon [k] \to [k]$.
  - **Add vertex** of color i and adjacent to colors $X \subseteq [k]$.

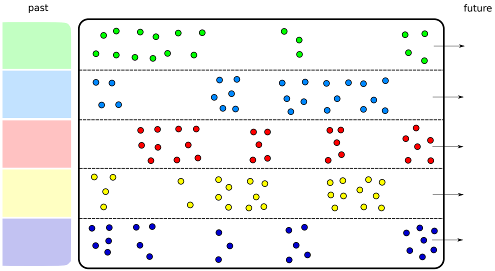
Instructions and derivations

- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi \colon [k] \to [k]$.
  - **Add vertex** of color i and adjacent to colors $X \subseteq [k]$.
- *k*-**derivation** corresponds to a word of instructions, and consists of:

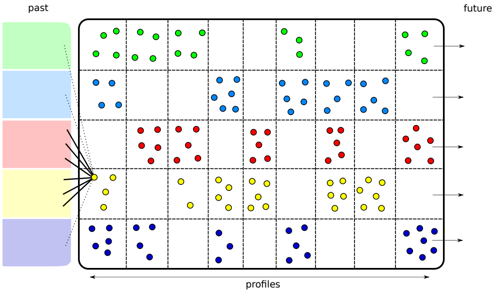


Instructions and derivations

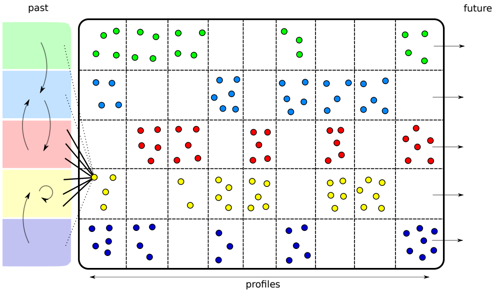
- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi\colon [k] \to [k]$.
  - **Add vertex** of color i and adjacent to colors $X \subseteq [k]$.
- k-**derivation** corresponds to a word of instructions, and consists of:
  - the underlying k-colored graph G;



- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi\colon [k] \to [k]$.
  - **Add vertex** of color i and adjacent to colors $X \subseteq [k]$.
- k-**derivation** corresponds to a word of instructions, and consists of:
  - the underlying k-colored graph G;
  - for each $u \in G$, its **profile** $\lambda(u) \subseteq [k]$;

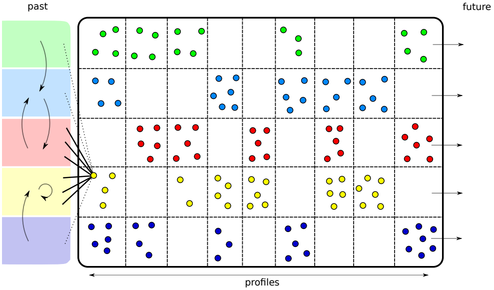


- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi\colon [k] \to [k]$.
  - **Add vertex** of color i and adjacent to colors $X \subseteq [k]$.
- k-**derivation** corresponds to a word of instructions, and consists of:
  - the underlying k-colored graph G;
  - for each $u \in G$, its **profile** $\lambda(u) \subseteq [k]$;
  - recoloring $\phi\colon [k] \to [k]$.



Instructions and derivations

- A linear cw decomposition of width k is a word over instructions:
  - **Recolor** according to a function $\phi\colon [k] \to [k]$.
  - **Add vertex** of color i and adjacent to colors $X \subseteq [k]$.
- k-**derivation** corresponds to a word of instructions, and consists of:
  - the underlying k-colored graph G;
  - for each $u \in G$, its **profile** $\lambda(u) \subseteq [k]$;
  - recoloring $\phi\colon [k] \to [k]$.
- Derivations have a natural semigroup structure.



Binary Lemma

Binary Lemma

For two k-derivations σ_1, σ_2, we have

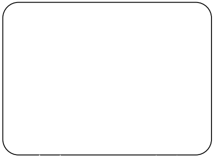
$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

Binary Lemma

For two k-derivations σ_1, σ_2, we have

$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

- **Proof**: We are given the underlying graph G of $\sigma_1 \cdot \sigma_2$.

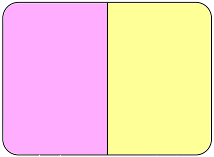


Binary Lemma

Binary Lemma

For two k-derivations σ_1, σ_2, we have
$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

- **Proof**: We are given the underlying graph G of $\sigma_1 \cdot \sigma_2$.
- Guess the partition of G into G_1 and G_2.



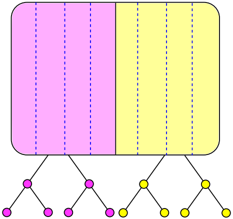
Binary Lemma

Binary Lemma

For two k-derivations σ_1, σ_2, we have

$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

- **Proof**: We are given the underlying graph G of $\sigma_1 \cdot \sigma_2$.
- Guess the partition of G into G_1 and G_2.
- Apply transductions to G_1 and G_2, obtaining clique decompositions.



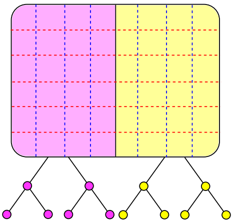
Binary Lemma

Binary Lemma

For two k-derivations σ_1, σ_2, we have

$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

- **Proof**: We are given the underlying graph G of $\sigma_1 \cdot \sigma_2$.
- Guess the partition of G into G_1 and G_2.
- Apply transductions to G_1 and G_2, obtaining clique decompositions.
- Cut between G_1 and G_2 has modular width at most 2^k.



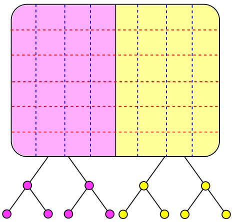
Binary Lemma

Binary Lemma

For two k-derivations σ_1, σ_2, we have

$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

- **Proof**: We are given the underlying graph G of $\sigma_1 \cdot \sigma_2$.
- Guess the partition of G into G_1 and G_2.
- Apply transductions to G_1 and G_2, obtaining clique decompositions.
- Cut between G_1 and G_2 has modular width at most 2^k.
- Enrich decompositions with neighborhoods on the other side.

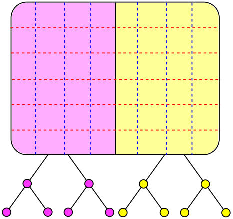


Binary Lemma

Binary Lemma

For two k-derivations σ_1, σ_2, we have
$$\mathrm{dcw}(\sigma_1 \cdot \sigma_2) \leqslant f(\mathrm{dcw}(\sigma_1), \mathrm{dcw}(\sigma_2)).$$

- **Proof**: We are given the underlying graph G of $\sigma_1 \cdot \sigma_2$.
- Guess the partition of G into G_1 and G_2.
- Apply transductions to G_1 and G_2, obtaining clique decompositions.
- Cut between G_1 and G_2 has modular width at most 2^k.
- Enrich decompositions with neighborhoods on the other side.
- Combine. $\square$



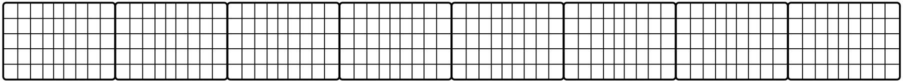
Abstraction and Idempotent Lemma

- **Abstraction**: Constant-size compositional information about a k-derivation that enables the following.

Idempotent Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with same idempotent abstraction. Then

$$\mathrm{dcw}(\sigma_1 \cdots \sigma_n) \leqslant f(\max_{i \in [n]} \mathrm{dcw}(\sigma_i)).$$



Abstraction and Idempotent Lemma

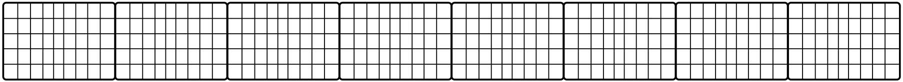
- **Abstraction**: Constant-size compositional information about a k-derivation that enables the following.

Idempotent Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with same idempotent abstraction. Then

$$\mathrm{dcw}(\sigma_1 \cdots \sigma_n) \leqslant f(\max_{i \in [n]} \mathrm{dcw}(\sigma_i)).$$

- **Intuition**: We can pack into abstraction all information that is relevant, provided it remains of size $\leqslant f(k)$.



Abstraction and Idempotent Lemma

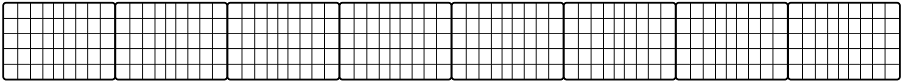
- **Abstraction**: Constant-size compositional information about a k-derivation that enables the following.

Idempotent Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with same idempotent abstraction. Then

$$\mathrm{dcw}(\sigma_1 \cdots \sigma_n) \leqslant f(\max_{i \in [n]} \mathrm{dcw}(\sigma_i)).$$

- **Intuition**: We can pack into abstraction all information that is relevant, provided it remains of size $\leqslant f(k)$.
- In our case:



Abstraction and Idempotent Lemma

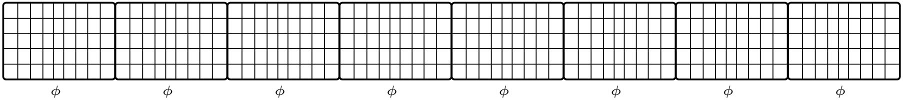
- **Abstraction**: Constant-size compositional information about a k-derivation that enables the following.

Idempotent Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with same idempotent abstraction. Then

$$\mathrm{dcw}(\sigma_1 \cdots \sigma_n) \leqslant f(\max_{i \in [n]} \mathrm{dcw}(\sigma_i)).$$

- **Intuition**: We can pack into abstraction all information that is relevant, provided it remains of size $\leqslant f(k)$.
- In our case:
  - All σ_i use the same idempotent recoloring ϕ.



Abstraction and Idempotent Lemma

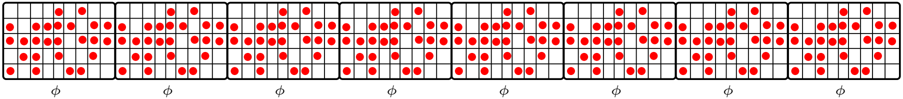
- **Abstraction**: Constant-size compositional information about a k-derivation that enables the following.

Idempotent Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with same idempotent abstraction. Then

$$\mathrm{dcw}(\sigma_1 \cdots \sigma_n) \leqslant f(\max_{i \in [n]} \mathrm{dcw}(\sigma_i)).$$

- **Intuition**: We can pack into abstraction all information that is relevant, provided it remains of size $\leqslant f(k)$.
- In our case:
  - All σ_i use the same idempotent recoloring ϕ.
  - In all σ_i the set of nonempty cells is the same.



Abstraction and Idempotent Lemma

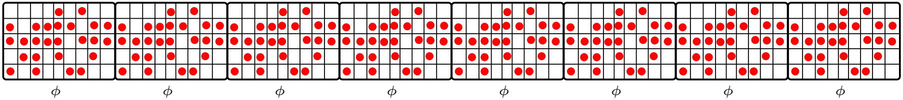
- **Abstraction**: Constant-size compositional information about a k-derivation that enables the following.

Idempotent Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with same idempotent abstraction. Then

$$\mathrm{dcw}(\sigma_1 \cdots \sigma_n) \leqslant f(\max_{i \in [n]} \mathrm{dcw}(\sigma_i)).$$

- **Intuition**: We can pack into abstraction all information that is relevant, provided it remains of size $\leqslant f(k)$.
- In our case:
  - All σ_i use the same idempotent recoloring ϕ.
  - In all σ_i the set of nonempty cells is the same.
  - We keep some information about paths between the cells.



Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.
  - Interpret the block order in each connected component.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.
  - Interpret the block order in each connected component.
  - Apply the assumed transductions to each block in parallel.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.
  - Interpret the block order in each connected component.
  - Apply the assumed transductions to each block in parallel.
  - Combine everything along the block order.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.
  - Interpret the block order in each connected component.
  - Apply the assumed transductions to each block in parallel.
  - Combine everything along the block order.
- Proving the Definable Order Lemma:

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.
  - Interpret the block order in each connected component.
  - Apply the assumed transductions to each block in parallel.
  - Combine everything along the block order.
- Proving the Definable Order Lemma:
  - Analyze interactions between cells.

Definable Order Lemma

- **Block order** on $\sigma_1 \cdots \sigma_n$: $u \preceq v$ iff $u \in \sigma_i$, $v \in \sigma_j$, and $i \leqslant j$.
- **Flip** of k-derivation τ:
  for some pairs of cells, revert the adjacency between them.

Definable Order Lemma

Let $\sigma_1, \ldots, \sigma_n$ be k-derivations with the same idempotent abstraction. Then there is some flip H of $\sigma_1 \cdots \sigma_n$ such that within every connected component of H, the block order can be expressed by an MSO formula of size bounded by a function of k.

- Using the Definable Order Lemma:
  - Guess partition into cells and the flip.
  - Interpret the block order in each connected component.
  - Apply the assumed transductions to each block in parallel.
  - Combine everything along the block order.
- Proving the Definable Order Lemma:
  - Analyze interactions between cells.
  - **Flip**: turn full adjacencies into full non-adjacencies to make connections local.

Conclusions

- **Treewidth** and **HR-recognizability**:

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.
  - First step robust, second treewidth-specific.

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.
  - First step robust, second treewidth-specific.
  - Direct attempts via Simon-like factorizations so far unsuccessful.

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.
  - First step robust, second treewidth-specific.
  - Direct attempts via Simon-like factorizations so far unsuccessful.
  - **Bonus**: One can compute even a decomposition of optimum width.

Conclusions

- **Treewidth** and **HR-recognizability**:
    - First prove the pathwidth case using Simon's factorization.
    - Then lift to the treewidth case via reduction to the pathwidth case.
    - First step robust, second treewidth-specific.
    - Direct attempts via Simon-like factorizations so far unsuccessful.
    - **Bonus**: One can compute even a decomposition of optimum width.
- **Cliquewidth** and **VR-recognizability**:

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.
  - First step robust, second treewidth-specific.
  - Direct attempts via Simon-like factorizations so far unsuccessful.
  - **Bonus**: One can compute even a decomposition of optimum width.
- **Cliquewidth** and **VR-recognizability**:
  - Linear cliquewidth case can be done using Simon's factorization.

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.
  - First step robust, second treewidth-specific.
  - Direct attempts via Simon-like factorizations so far unsuccessful.
  - **Bonus**: One can compute even a decomposition of optimum width.
- **Cliquewidth** and **VR-recognizability**:
  - Linear cliquewidth case can be done using Simon's factorization.
  - Full conjecture for cliquewidth remains wide open.

Conclusions

- **Treewidth** and **HR-recognizability**:
  - First prove the pathwidth case using Simon's factorization.
  - Then lift to the treewidth case via reduction to the pathwidth case.
  - First step robust, second treewidth-specific.
  - Direct attempts via Simon-like factorizations so far unsuccessful.
  - **Bonus**: One can compute even a decomposition of optimum width.
- **Cliquewidth** and **VR-recognizability**:
  - Linear cliquewidth case can be done using Simon's factorization.
  - Full conjecture for cliquewidth remains wide open.
- **Thank you for your attention!**