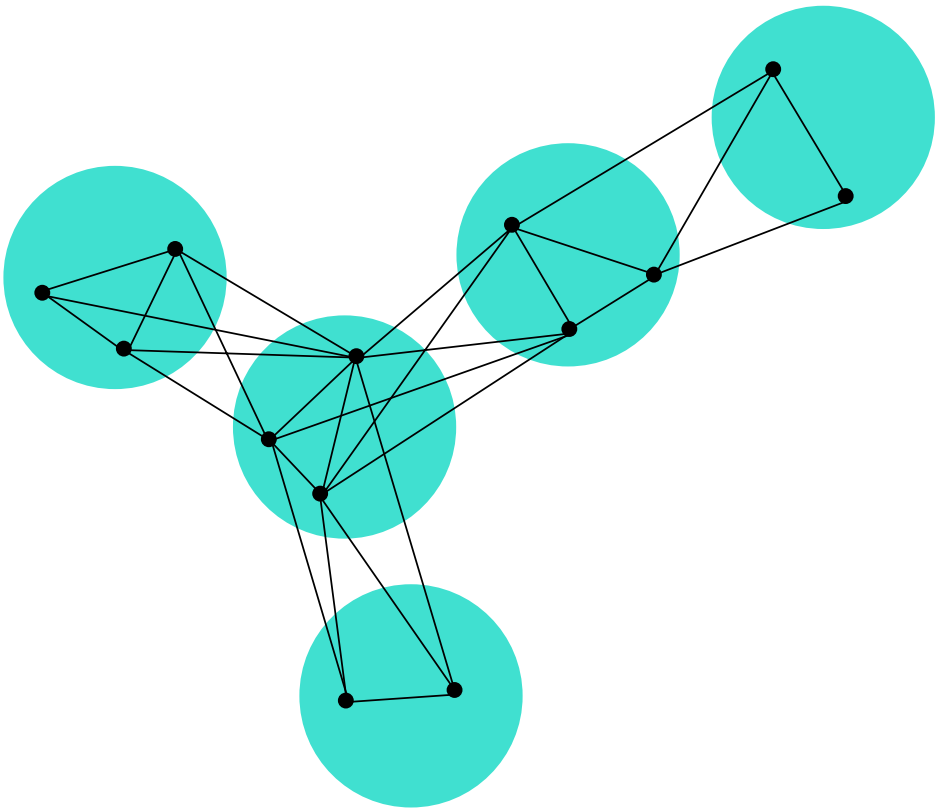
On the parameterized complexity of computing tree-partitions

Hans L. Bodlaender, Carla Groenland, Hugo Jacob

Tree-partition vs tree decomposition

What is a tree-partition?

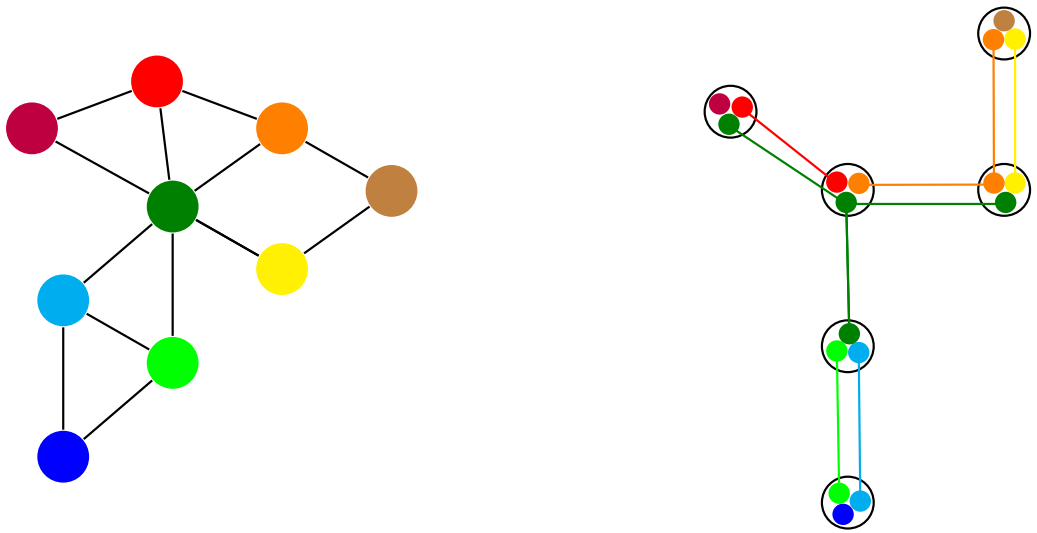


Tree-partition-width is largest bag size (3 here).

- Partition of vertex set indexed by tree nodes
- Each part of bounded size
- Edges are within parts or between adjacent parts

Tree-partition vs tree decomposition

What is treewidth?

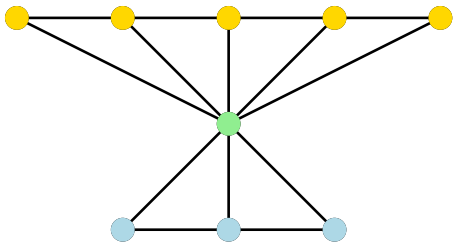


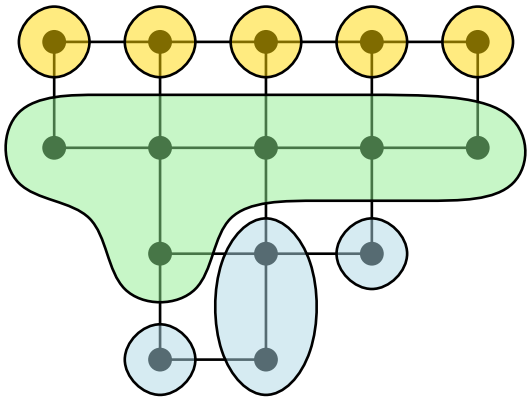
Bags are not disjoint but cover all vertices and all edges.
Each vertex is mapped to a (connected) subtree.

Treewidth is largest bag size minus one.

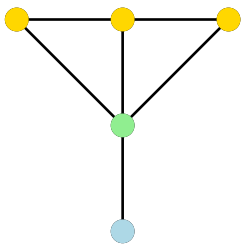
Tree-partition vs tree decomposition

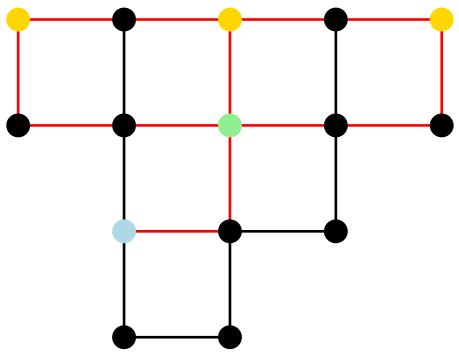
Graph minors





Minor



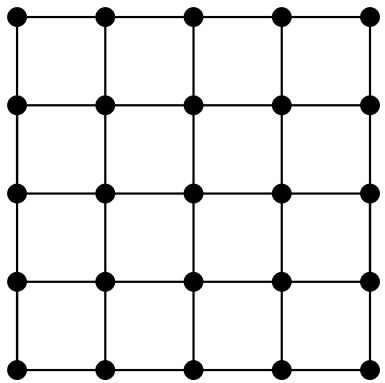


Topological minor

Tree-partition vs tree decomposition

Grid minor theorem

A $k \times k$-grid has treewidth k.



Theorem (Robertson and Seymour)

Every graph with large treewidth contains a large grid minor.

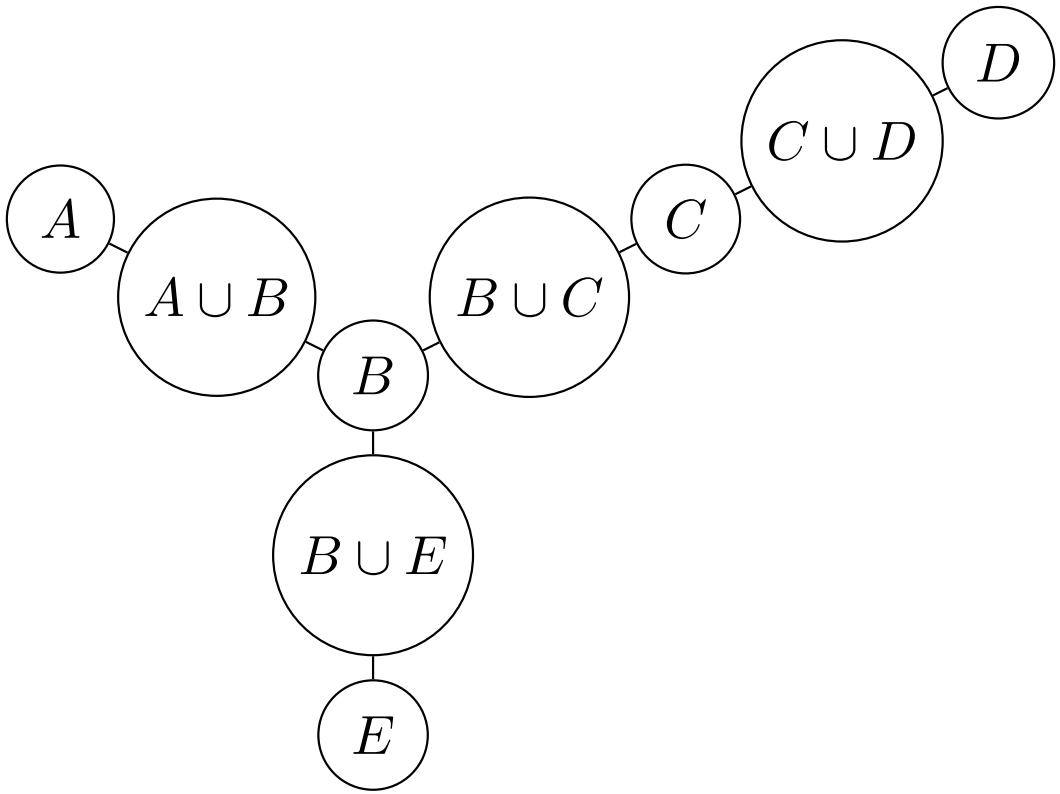
Tree-partition vs tree decomposition

Some context on tree-partitions

- Introduced independently by Seese ('85) and Halin ('91).
- Work of Ding and Oporowski ('95,'96) on tree-partition-width: its relationship with treewidth and a characterisation by topological minor obstructions.
- Mostly studied for 'structural' purposes (coloring, layout, ...)
- Recently Bodlaender, Cornelissen, and van der Wegen gave parameterized problems that are FPT parameterized by tree-partition-width but hard for treewidth.

Tree-partition vs tree decomposition

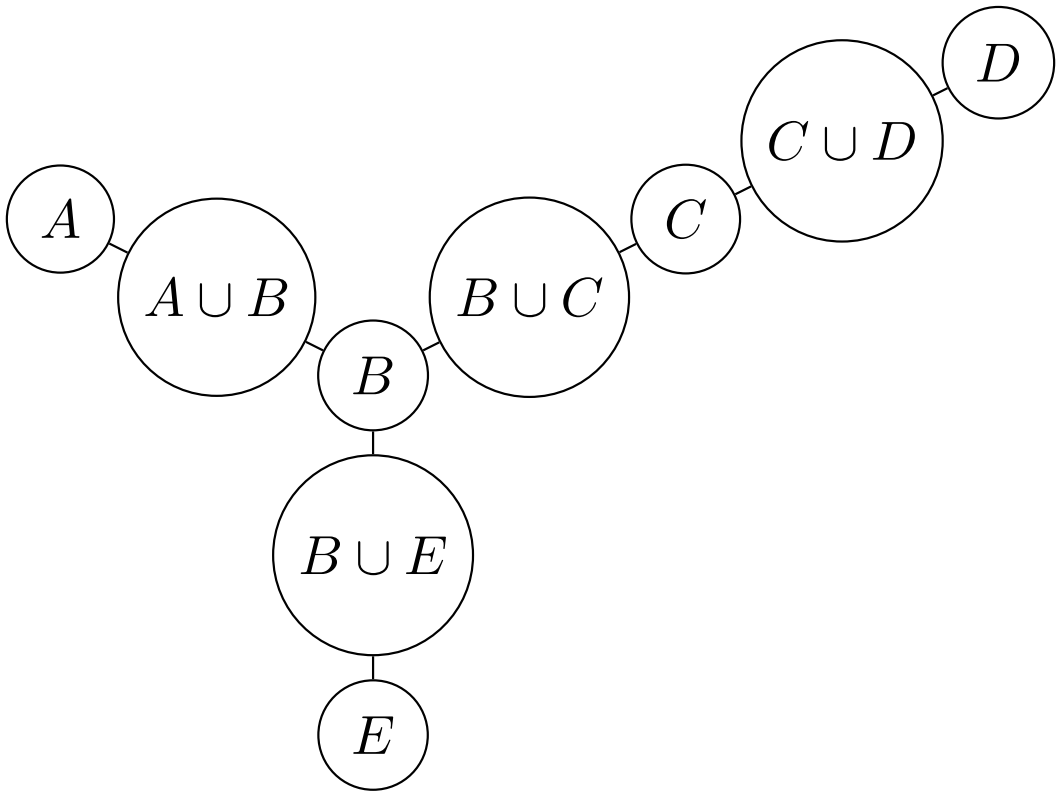
Relation to treewidth



$$\mathbf{tw} \leq 2\mathbf{tpw} - 1$$

Tree-partition vs tree decomposition

Relation to treewidth

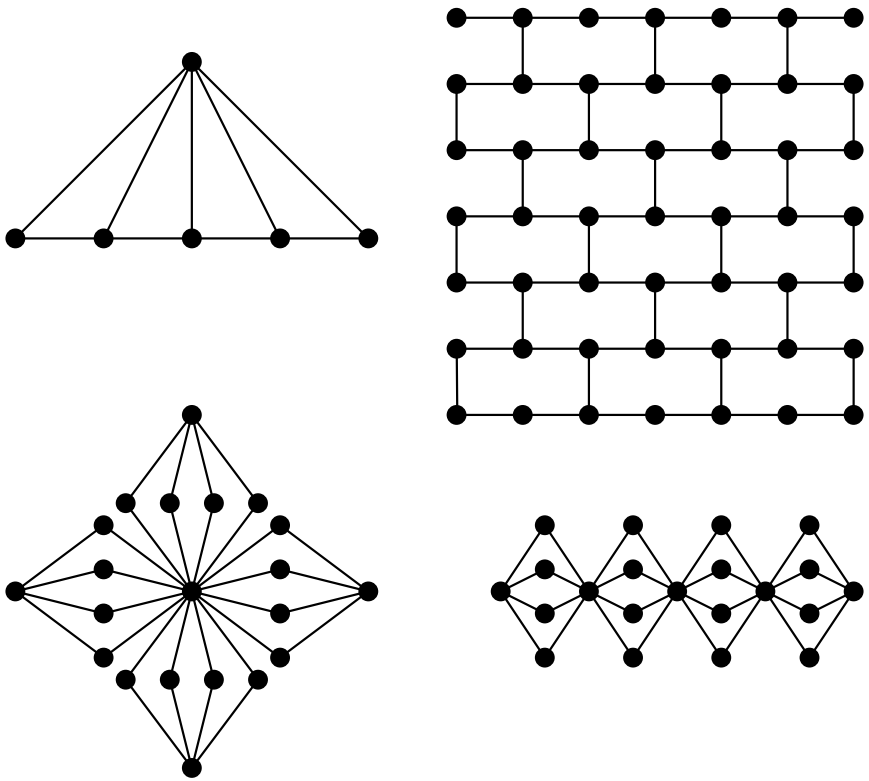


$$\mathbf{tw} \leq 2\mathbf{tpw} - 1$$

Ding and Oporowski ('95) and Wood ('09) show $\mathbf{tpw} = O(\Delta\mathbf{tw})$.

Tree-partition vs tree decomposition

Topological minor obstructions for tree-partition-width



Tree-partition vs tree decomposition

Our goal

The efficient algorithms parameterized by tree-partition-width require a given decomposition.

No known algorithms to efficiently compute a tree-partition of good width prior to our work.

Basic complexity classes

Definition (FPT)

The class of parameterized problems that can be solved (deterministically) in time $f(k) \cdot n^{O(1)}$.

Definition (XP)

The class of parameterized problems that can be solved (deterministically) in time $f(k) \cdot n^{g(k)}$.

Conjecture

FPT $\subsetneq$ W[1]

Quick introduction to XNLP

XNLP : what can be computed by a nondeterministic Turing machine using $f(k)\log n$ space, running in time $f(k)n^c$.

Quick introduction to XNLP

XNLP : what can be computed by a nondeterministic Turing machine using $f(k)\log n$ space, running in time $f(k)n^c$.

What do XNLP-hard problems look like ?
Linear structure + locally hard parameterized problem.

Quick introduction to XNLP

XNLP : what can be computed by a nondeterministic Turing machine using $f(k)\log n$ space, running in time $f(k)n^c$.

What do XNLP-hard problems look like ?
Linear structure + locally hard parameterized problem.

Conjecture

There exist no deterministic algorithms running in XP time and FPT space for XNLP-hard problems.

Quick introduction to XALP

XALP : what can be computed by an alternating Turing machine using $f(k)\log n$ space, with total computation in time $f(k)n^c$.

Or equivalently, what can be computed by a nondeterministic Turing machine using $f(k)\log n$ space and a stack, running in time $f(k)n^c$.

Quick introduction to XALP

XALP : what can be computed by an alternating Turing machine using $f(k)\log n$ space, with total computation in time $f(k)n^c$.

Or equivalently, what can be computed by a nondeterministic Turing machine using $f(k)\log n$ space and a stack, running in time $f(k)n^c$.

What do XALP-hard problems look like ?
Tree-like structure + locally hard parameterized problem.

Quick introduction to XALP

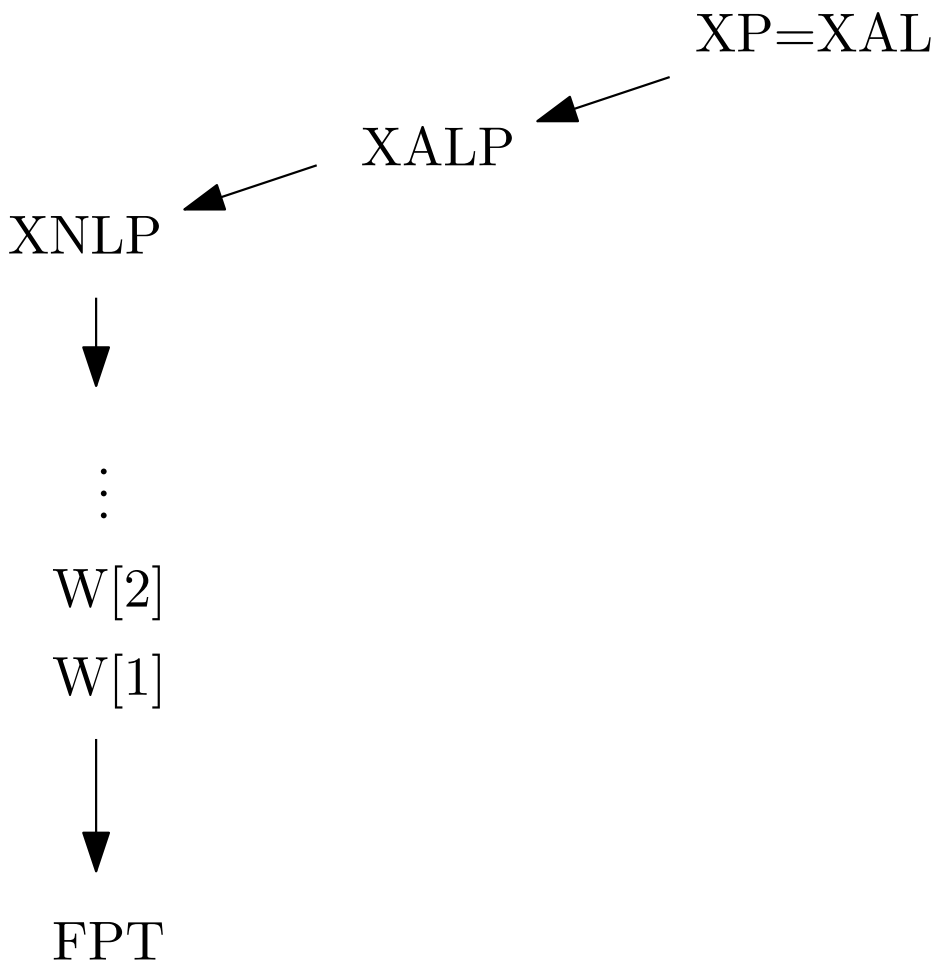
XALP : what can be computed by an alternating Turing machine using $f(k)\log n$ space, with total computation in time $f(k)n^c$.

Or equivalently, what can be computed by a nondeterministic Turing machine using $f(k)\log n$ space and a stack, running in time $f(k)n^c$.

What do XALP-hard problems look like ?
Tree-like structure + locally hard parameterized problem.

XALP-hardness $\Rightarrow$ XNLP-hardness $\Rightarrow$ W[t]-hardness, for all t

Complexity classes



Courcelle's theorem

Theorem

Given a graph G, any property expressible by a formula ϕ in MSO logic can be tested in time $f(\mathbf{tw}(G), |\phi|) \cdot n$.

This extends to optimisation variants and modulo counting predicates.

This applies to graphs of bounded tree-partition-width.

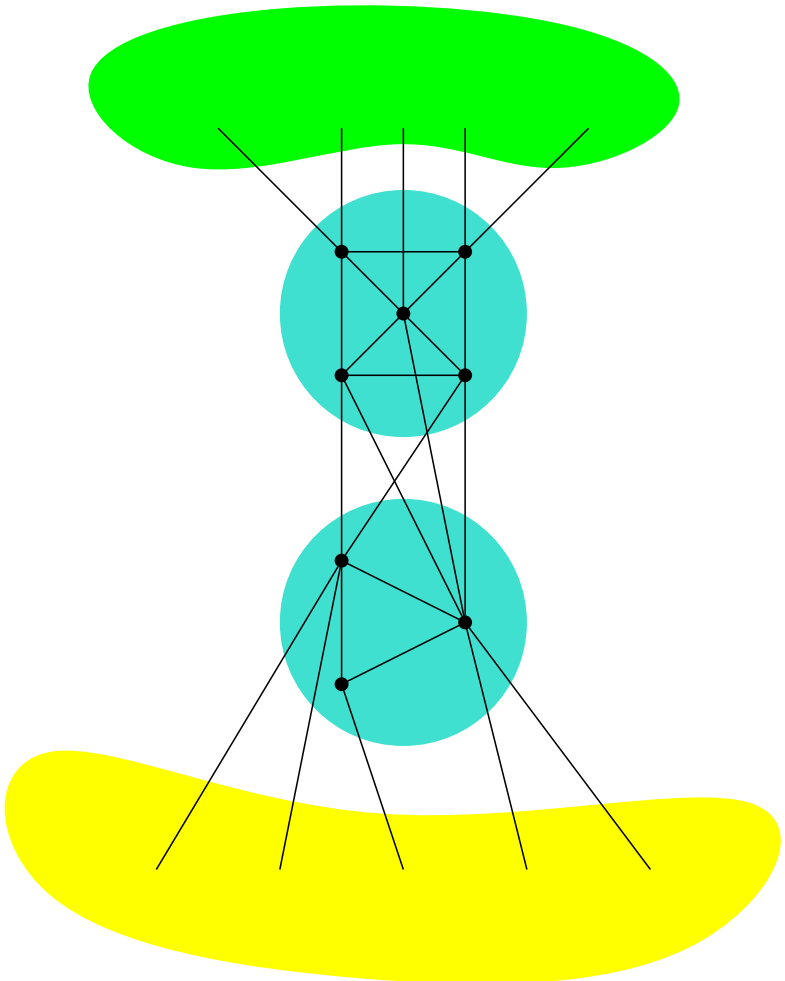
Other problems

Some problems are XALP-hard for treewidth and FPT for tree-partition-width:

- Capacitated Vertex Cover, Capacitated Dominating Set, Bounded-Degree Vertex Deletion
- Target Outdegree Orientation, Chosen Maximum Outdegree, Minimum Maximum Outdegree, Circulating Orientation, Undirected Flow with Lower Bounds, All-or-Nothing Flow

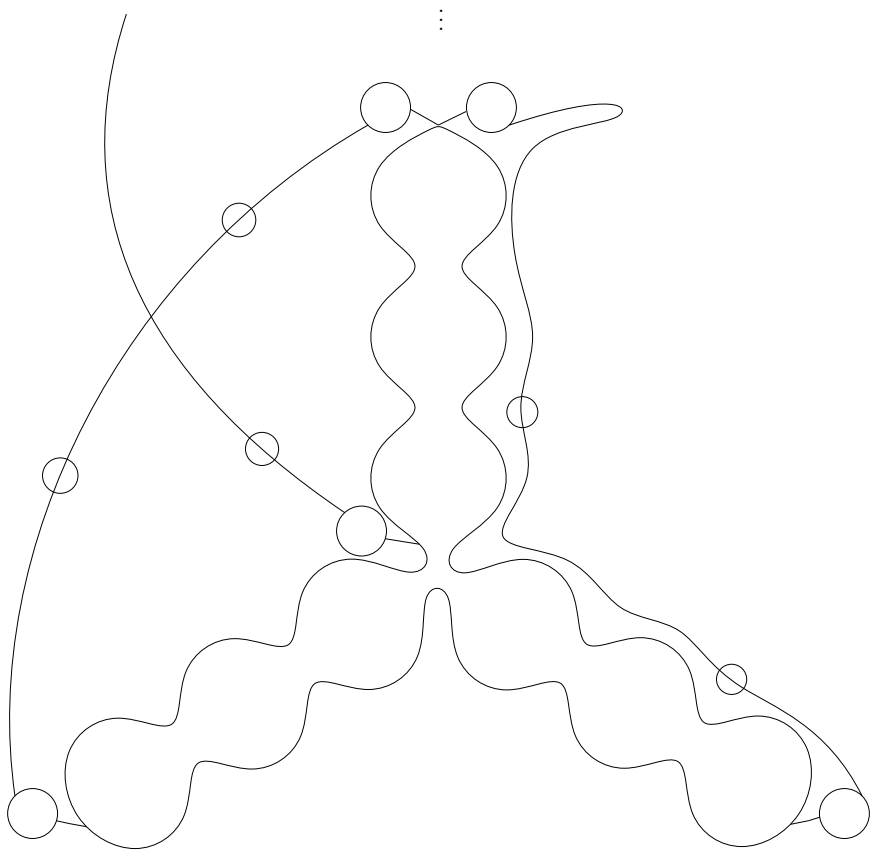
Computing an optimal tree-partition

An XP algorithm for tree-partitions



Computing an optimal tree-partition

Hardness



Reduction from a tree-like variant of Independent Set.

Computing an optimal tree-partition

Hardness

Theorem

Tree-Partition-Width *is XALP-complete.*

This means we do not expect a significantly better algorithm to find an optimal tree-partition to exist.

Hardness

Theorem

Tree-Partition-Width *is XALP-complete.*

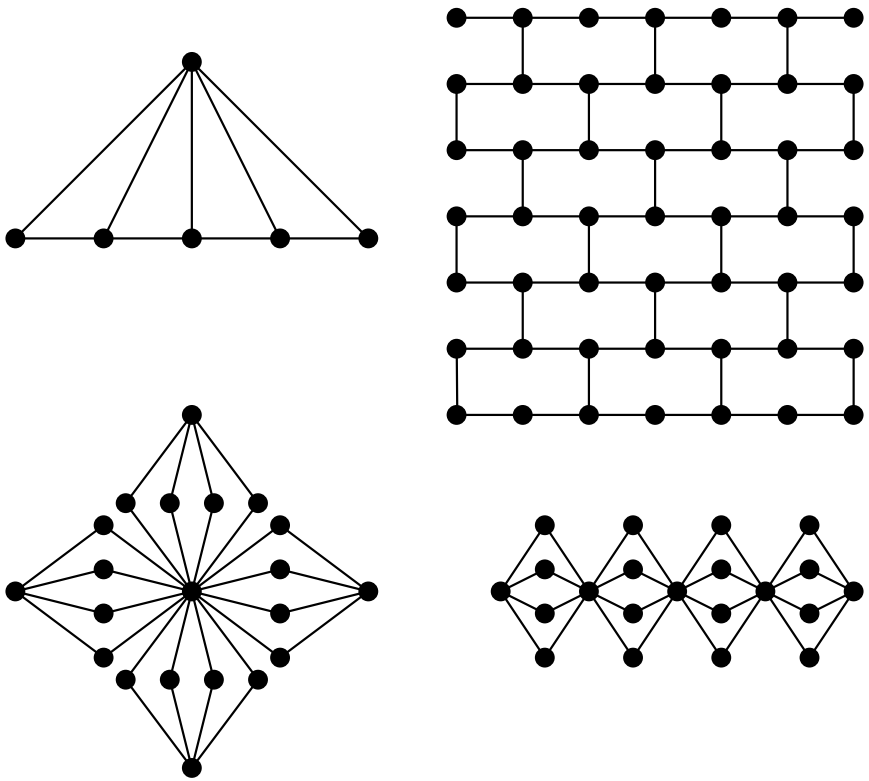
This means we do not expect a significantly better algorithm to find an optimal tree-partition to exist.

We can still hope for efficient approximation!
This is sufficient to be used as a routine by FPT algorithms.

Exploiting the topological minor obstructions

- We can find topological minor obstructions in FPT time, using the algorithm of Grohe, Kawarabayashi, Marx, and Wollan (STOC 2011).
- The result of Ding and Oporowski is constructive and can be turned into an algorithm. However, the algorithm is not very efficient in terms of running time and in terms of approximation guarantees.

Reminder of obstructions

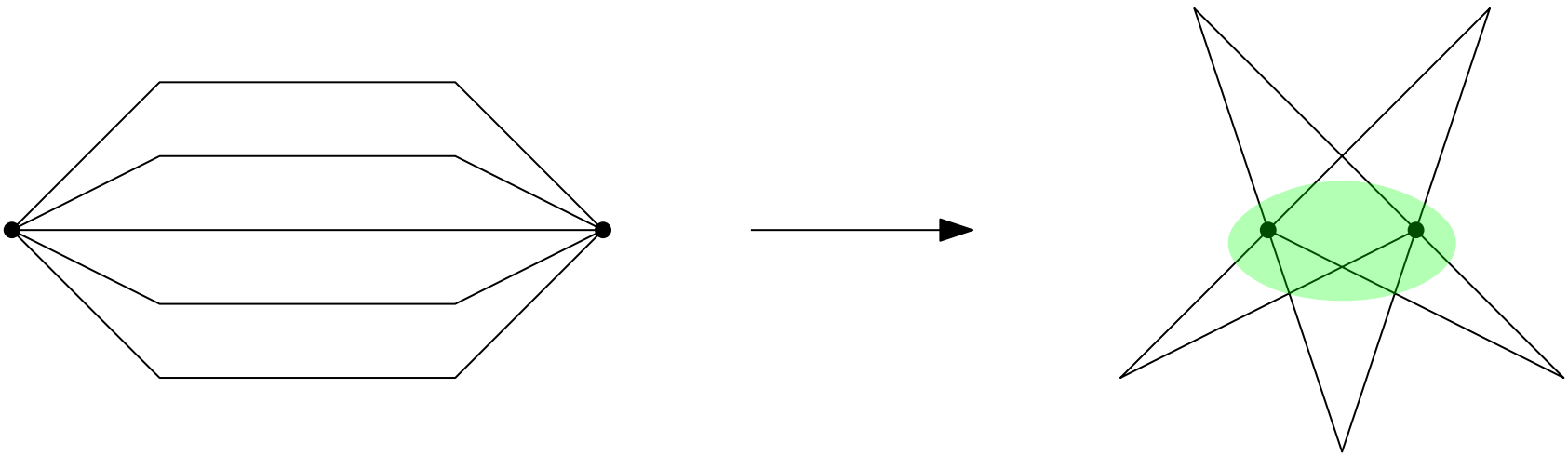


Scheme

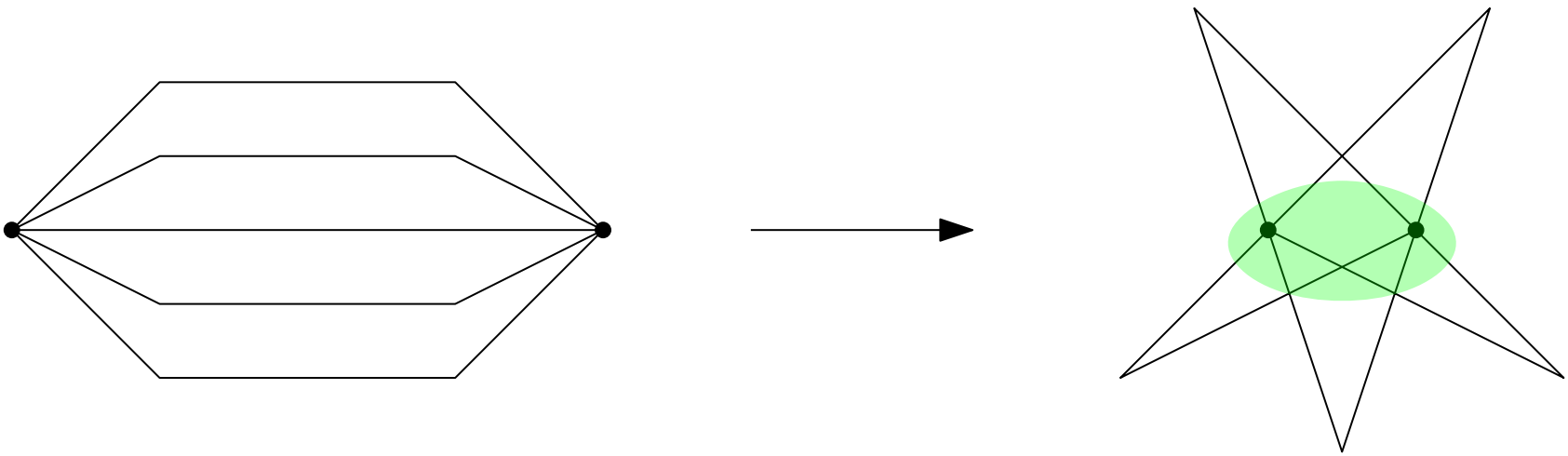
We start with a graph G, and parameter k, a believed upper bound to $\mathbf{tpw}(G)$.

- First check that the treewidth is small and get a tree decomposition of width w.
- Compute some clever auxiliary graph.
- Compute a tree-partition on blocks of the auxiliary graph.
- Combine them and deduce a tree-partition of the original graph.

Auxiliary graph

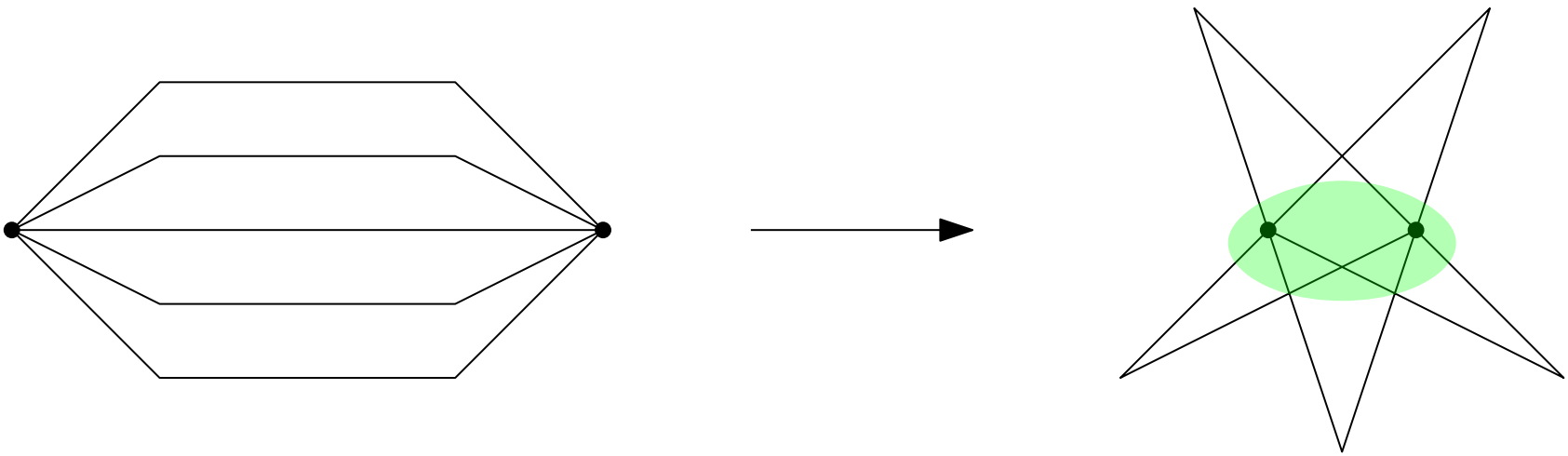


Auxiliary graph



High connectivity between a pair of vertices implies their proximity in a good tree-partition.

Auxiliary graph



High connectivity between a pair of vertices implies their proximity in a good tree-partition.

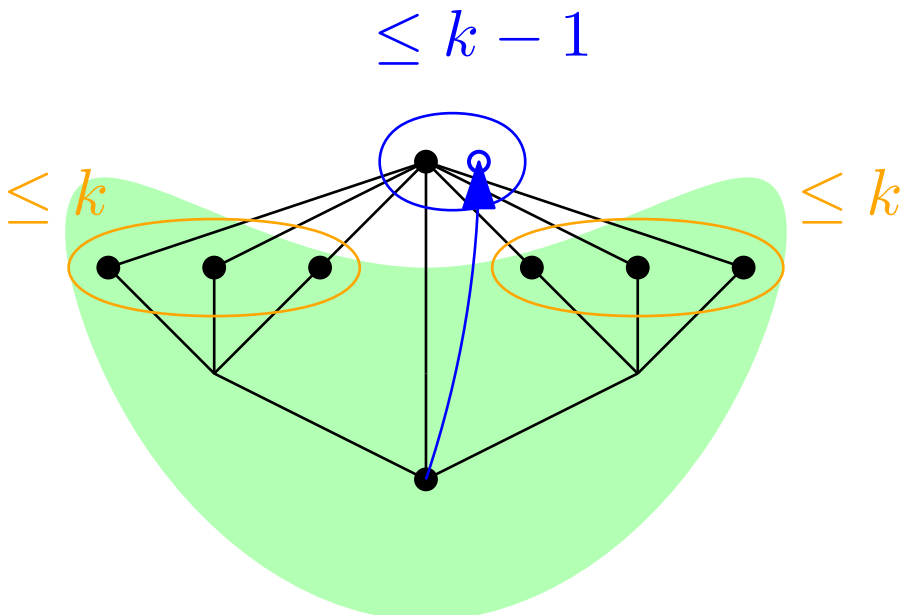
For bags of size at most k, connectivity of $2k$ between a pair of vertices enforces that they are in the same bag.

Auxiliary graph

By contracting all pairs of vertices with connectivity at least $2k$, we obtain the auxiliary graph H.

We will use the following key property: within blocks of H, maximum degree is bounded by a function of k.

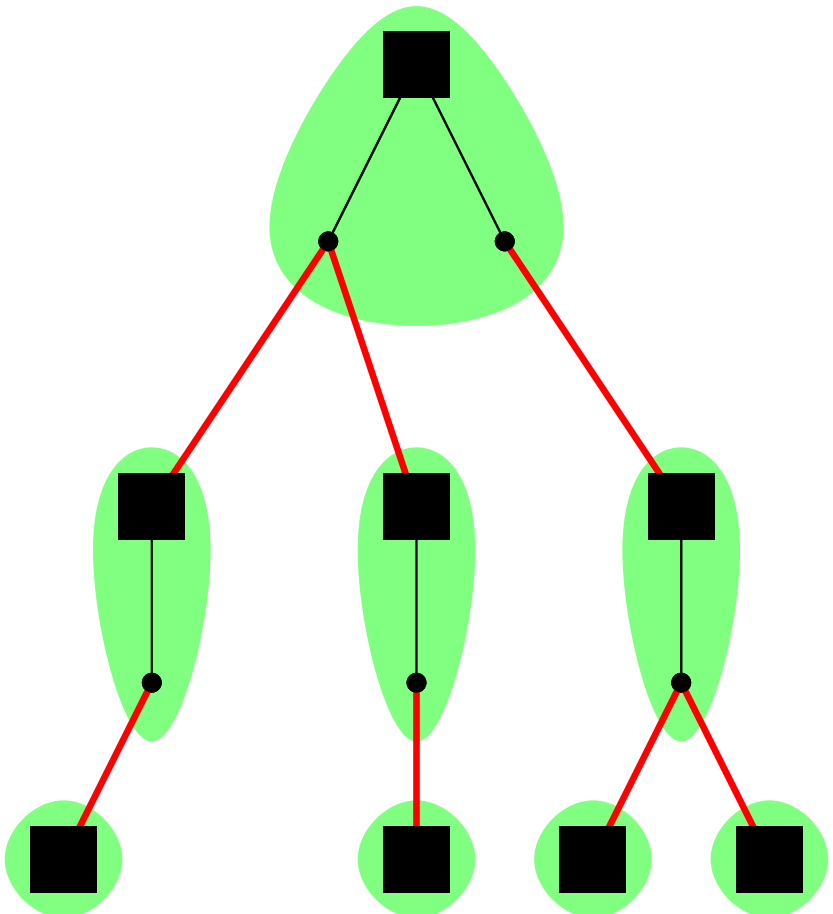
Degree in blocks of the reduced graph



Computing the tree-partition

- We can compute tree-partitions for blocks of H using constructions with width $O(\Delta \mathbf{tw})$.
- We can then combine them to obtain a tree-partition of H, by merging bags that share vertices.
- We should be careful about cutvertices since they may be contained in many blocks. To deal with this obstacle, we can make sure that each cutvertex has only one block adding vertices to its bag.

Dealing with cutvertices



Concluding the construction

- The constructions of width $O(\Delta \mathbf{tw})$ allow for the isolation of exactly one vertex for free.
- We can get a tree-partition of width $O(\Delta(B^*)w)$ for H where B^* is the block that maximizes its maximum degree. We should have $\Delta(B^*) = O(wk^2)$, otherwise we can contradict $\mathbf{tpw}(G) \leq k$.
- Expanding the vertices of H, we get a tree-partition for G of width $O(w^2k^3)$.

Possible implementations

Computing auxiliary graph H

We can restrict contractible pairs to vertices sharing a bag of the tree decomposition.

- We can compute the number of vertex disjoint paths using dynamic programming on the tree decomposition.
  $\rightsquigarrow 2^{O(k \log k)} n$
- We can use simple flow computations for each pair of vertices.
  $\rightsquigarrow k^{O(1)} n^2$

Possible implementations

Finding balanced separators

- We can reduce the diameter of the tree decomposition to $O(\log n)$ and find balanced separators by moving around the decomposition. This runs in $O(k^{O(1)} n \log n$ overall.
- We can use more elaborate techniques to obtain a running time of $O(2^{O(k)} n)$ overall.

Results

The scheme allows to find tree-partitions of width:

- $O(k^5 \log k)$ in polynomial time; or
- $O(k^5)$ in $2^{O(k \log k)} n$ time; or
- $O(k^7)$ in $k^{O(1)} n^2$.

or conclude that any tree-partition has width more than k.

Conclusion

Open questions

- Can we find better approximations ?
- Can we improve the running time of our approximation to $k^c n \log n$?
- What are the problems that can be solved in FPT time when parameterized by tree-partition-width ?