Hypergraph Acyclicity and Propositional Model Counting

Florent Capelli, Arnaud Durand, Stefan Mengel

IMJ-PRG, Université Paris Diderot

17/07/2014

Propositional model counting

#SAT

input: CNF-formula F
task: compute number of satisfying assignments of F

- Canonical #P-complete problem
- #2-**SAT** is also #P-complete
- Even #P-hard to approximate

Goal: structural restrictions on the input to find tractable instances

Structural Restrictions

Associate a graph to a CNF-formula:

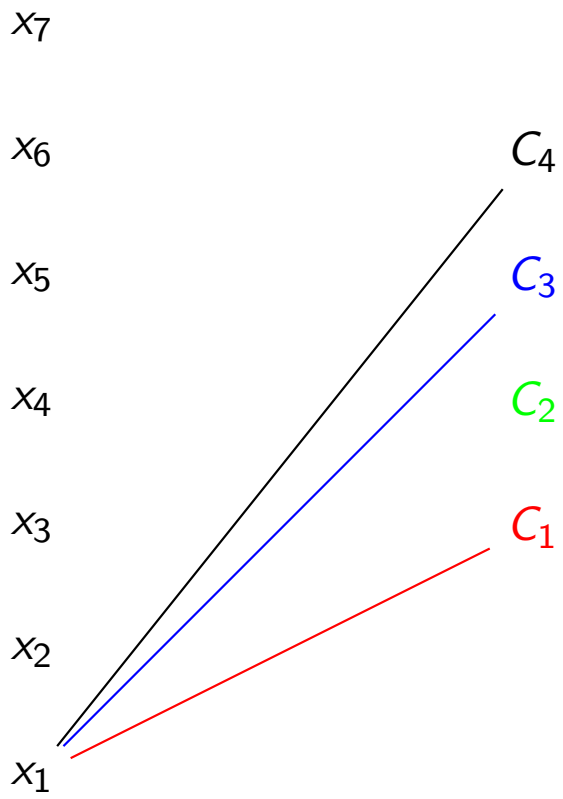


Figure: $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee x_5 \vee x_6) \wedge (x_1 \vee \neg x_3 \vee x_5 \vee \neg x_7)$

Structural Restrictions

Associate a graph to a CNF-formula:

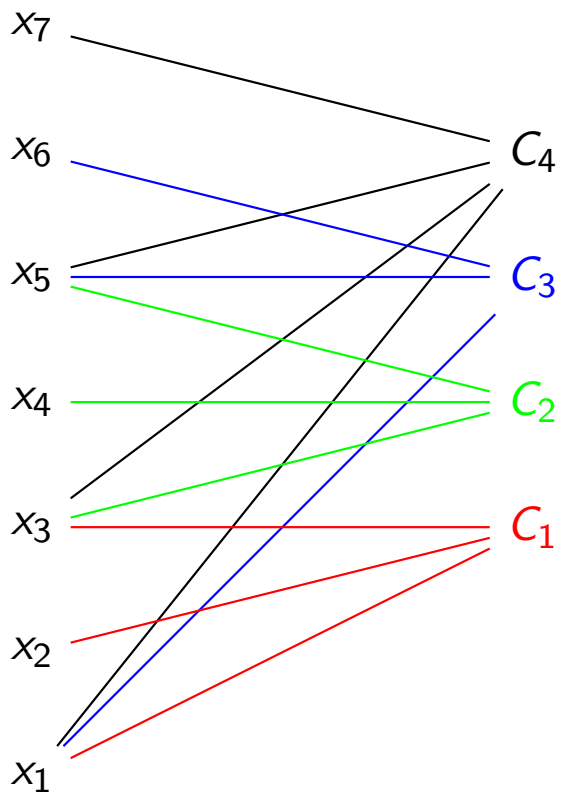


Figure: $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee x_5 \vee x_6) \wedge (x_1 \vee \neg x_3 \vee x_5 \vee \neg x_7)$

Known results on structural restriction

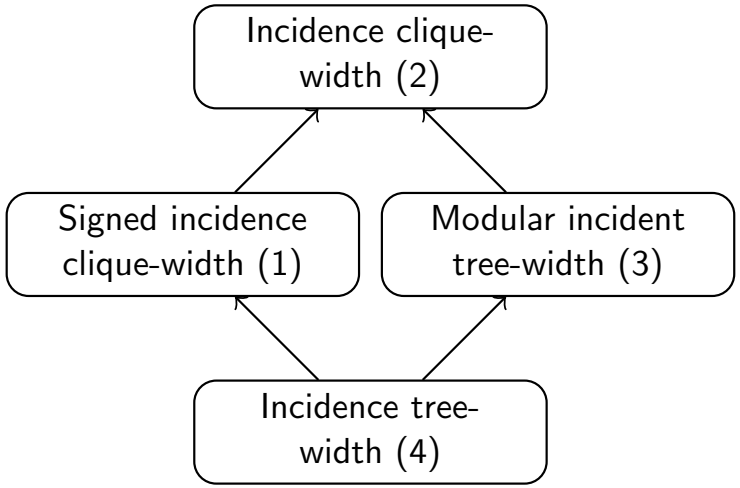


Figure: Tractable restrictions of #**SAT**

1. Fisher, Makowski, Ravve, 2008
2. Slivovsky, Szeider, 2013
3. Paulusma, Slivovsky, Szeider, 2013
4. Samer, Szeider, 2010

Tractable: complexity of the form $f(k)n^{O(1)}$ (FPT) or $n^{O(k)}$ (XP) where k is the parameter

Structural Restrictions

Or associate a hypergraph to a CNF formula F

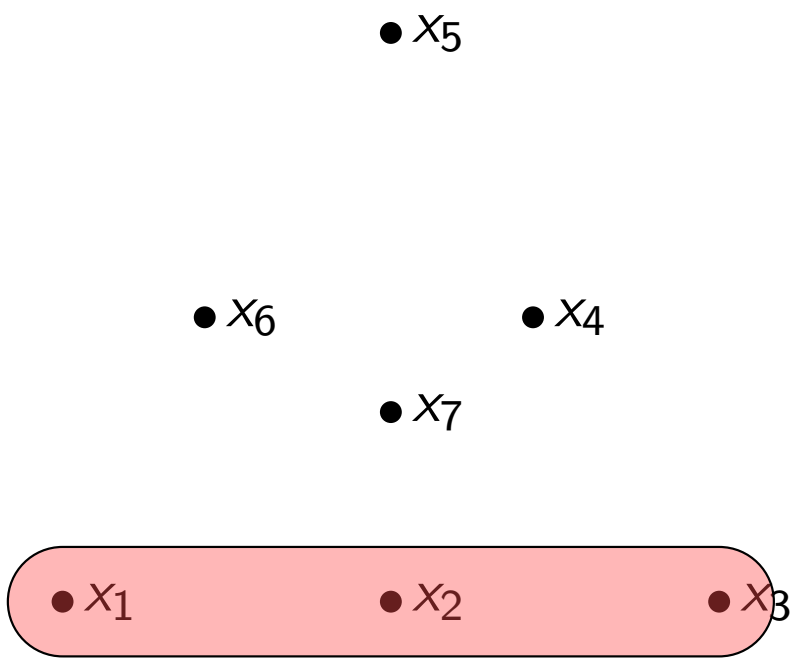


Figure: $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee x_5 \vee x_6) \wedge (x_1 \vee \neg x_3 \vee x_5 \vee \neg x_7)$

Structural Restrictions

Or associate a hypergraph to a CNF formula F

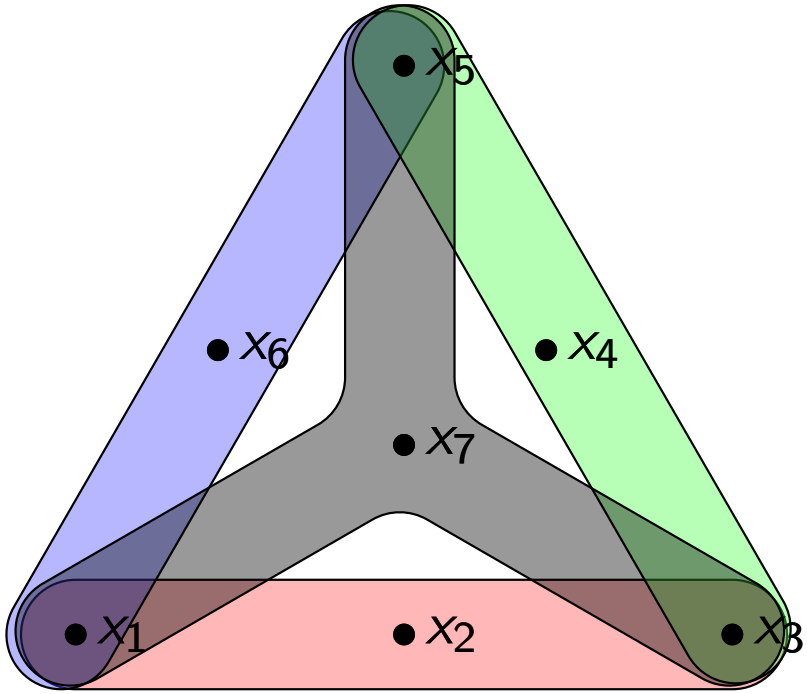


Figure: $(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee x_5 \vee x_6) \wedge (x_1 \vee \neg x_3 \vee x_5 \vee \neg x_7)$

What kind of structural restrictions for hypergraphs can yield an efficient algorithm for #**SAT**?

Acyclicity for hypergraphs

How to generalize acyclicity to hypergraphs?

- should agree with acyclicity when restricted to graphs
- should allow useful datastructures for algorithmic purposes
- should be decidable in polynomial time

Studied extensively in database theory.

α-acyclicity

Idea : organizing the edges in a tree, as for the treewidth for graphs.

α-acyclicity

Idea : organizing the edges in a tree, as for the treewidth for graphs.

A hypergraph $\mathcal{H} = (V, E)$ is α-acyclic if there exists $\mathcal{T} = (T, A)$ a tree and a labelling function $\lambda : T \to E$ such that:

- for all $e \in E$, there exists a vertex of $\mathcal{T}$ whose label is e

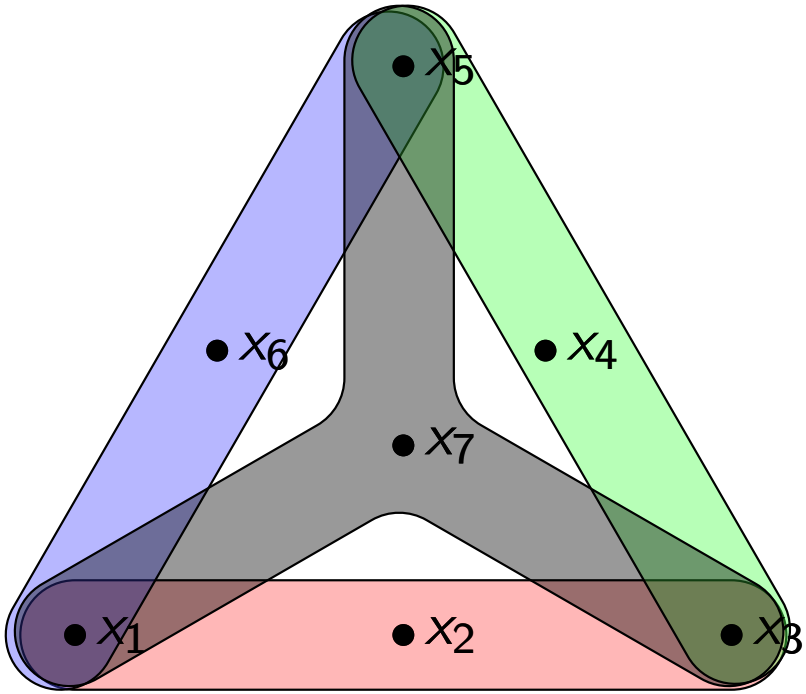
α-acyclicity

Idea : organizing the edges in a tree, as for the treewidth for graphs.

A hypergraph $\mathcal{H} = (V, E)$ is α-acyclic if there exists $\mathcal{T} = (T, A)$ a tree and a labelling function $\lambda : T \to E$ such that:

- for all $e \in E$, there exists a vertex of $\mathcal{T}$ whose label is e
- for all vertex $v \in V$, $\{t \in T \mid v \in \lambda(t)\}$ is a connected subtree of $\mathcal{T}$

An example



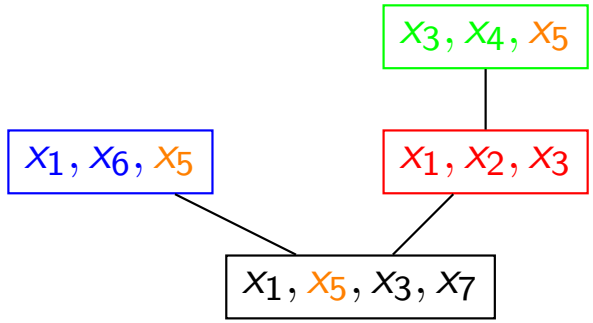
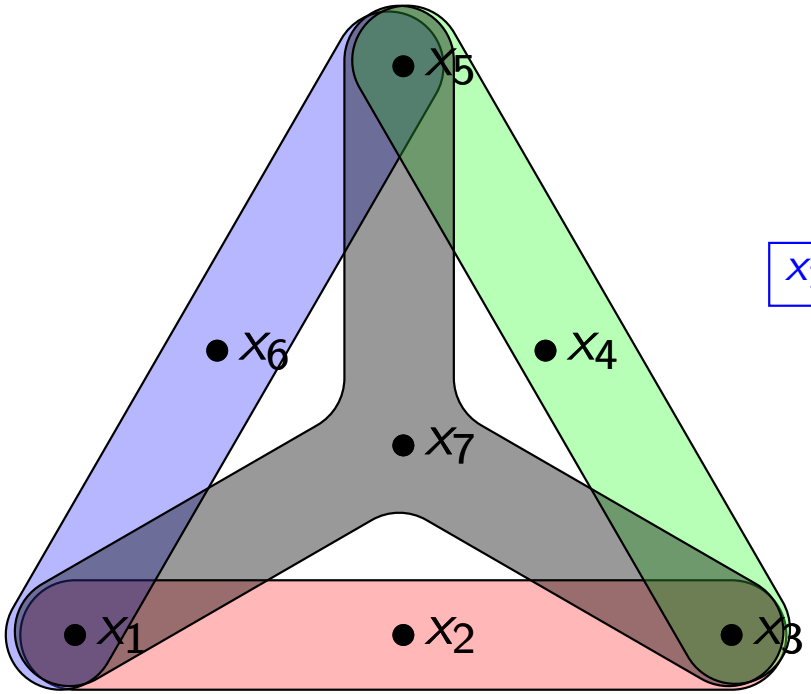


Figure: Bad tree

An example



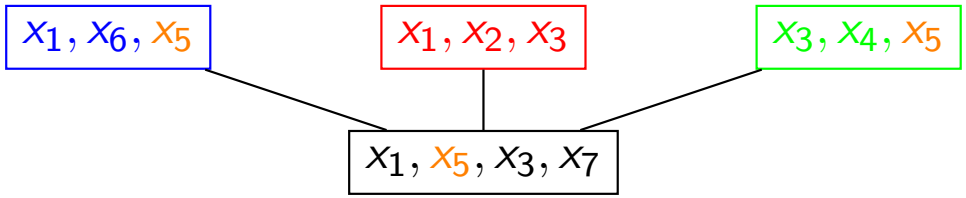


Figure: Good tree

α-acyclicity and #**SAT**

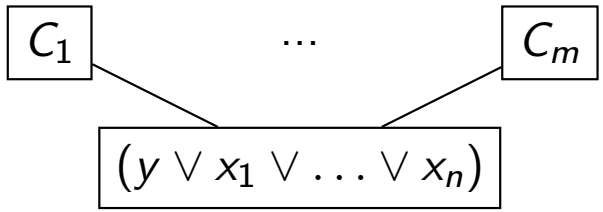
α-acyclicity is not a helpful notion for **SAT** and #**SAT**:

- if $\mathcal{H}$ has an edge containing all vertices, it is α-acyclic

α-acyclicity and #**SAT**

α-acyclicity is not a helpful notion for **SAT** and #**SAT**:

- if $\mathcal{H}$ has an edge containing all vertices, it is α-acyclic
- $\phi \wedge (y \vee x_1 \vee \ldots \vee x_n)$ has a solution if and only if $\phi = \bigwedge C_i$ has one (where $y \notin \mathrm{var}(\phi) = \{x_1, \ldots, x_n\}$) and is α-acyclic:



Disjoint branches acyclicity (Duris (2009))

Idea: additional condition on the join tree

Definition

A hypergraph $\mathcal{H}$ admits a disjoint branches decomposition if there exists a (rooted) join tree $\mathcal{T}$ for $\mathcal{H}$ such that no variable appears in more than one branch.

Disjoint branches acyclicity (Duris (2009))

Idea: additional condition on the join tree

Definition

A hypergraph $\mathcal{H}$ admits a disjoint branches decomposition if there exists a (rooted) join tree $\mathcal{T}$ for $\mathcal{H}$ such that no variable appears in more than one branch.

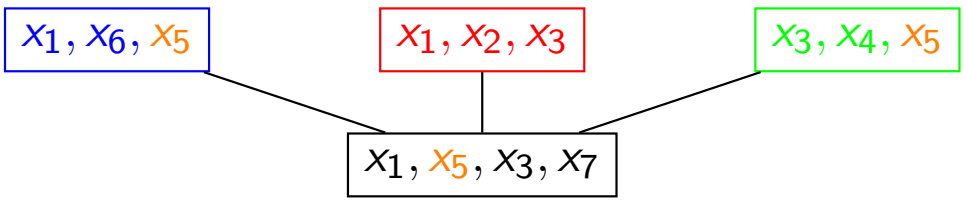


Figure: This is not disjoint branches

Disjoint branches acyclicity (Duris (2009))

Idea: additional condition on the join tree

Definition

A hypergraph $\mathcal{H}$ admits a disjoint branches decomposition if there exists a (rooted) join tree $\mathcal{T}$ for $\mathcal{H}$ such that no variable appears in more than one branch.

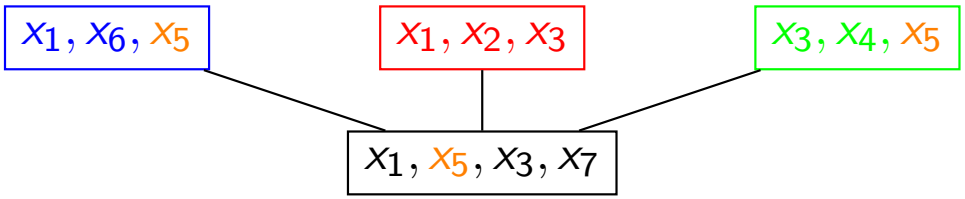


Figure: This is not disjoint branches

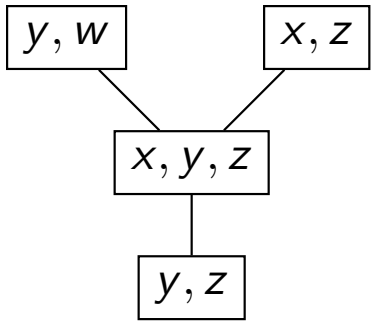


Figure: Disjoint branches tree

Disjoint branches $\subsetneq$ α-acyclicity

Contributions of the paper

Polynomial time algorithms for two problems:

1. **Counting problem**: given a disjoint branches decomposition of the hypergraph associated to a CNF-formula, compute the number of models of this formula

Contributions of the paper

Polynomial time algorithms for two problems:

1. **Counting problem**: given a disjoint branches decomposition of the hypergraph associated to a CNF-formula, compute the number of models of this formula
2. **Discovery problem**: given a hypergraph, decide if it has a disjoint branches decomposition and compute one if it exists

Counting models

- For a partial assignment a of the variables,

$$S(\phi, a) = \#\{b \simeq a \mid b \models \phi\}$$

- Counting models of $\phi := \bigwedge_{i=1}^{m} C_i$ or of $\neg\phi = \bigvee_{i=1}^{m} \neg C_i$ is equivalent:

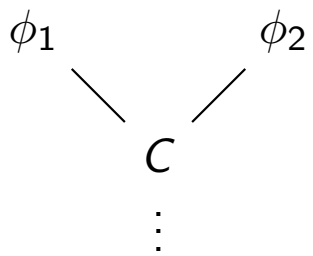
$$S(\phi) = 2^{|\mathrm{var}(\phi)|} - S(\neg\phi)$$

- If $\mathrm{var}(\phi_1) \cap \mathrm{var}(\phi_2) = \emptyset$, then

$$S(\phi_1 \vee \phi_2, a) = S(\phi_1, a|_{\mathrm{var}(\phi_1)}) \times S(\phi_2, a|_{\mathrm{var}(\phi_2)})$$

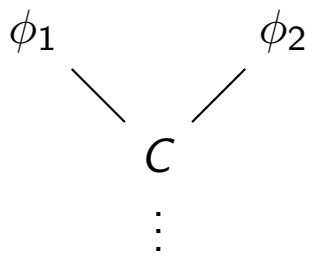
Counting models

Take a disjoint branches join tree:



Counting models

Take a disjoint branches join tree:



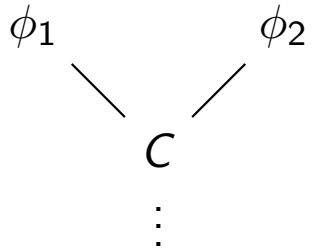
If a is the *only* assignment of $\mathrm{var}(C)$ such that $a \models \neg C$:

$$S(\neg C \vee \phi_1 \vee \phi_2) =$$

$$2^{|\mathrm{var}(C) \setminus \mathrm{var}(\phi_1 \vee \phi_2)|} (S(\phi_1) S(\phi_2) \text{ solutions of } \phi_1 \vee \phi_2$$

Counting models

Take a disjoint branches join tree:



If a is the *only* assignment of $\mathrm{var}(C)$ such that $a \models \neg C$:

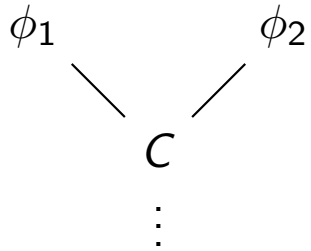
$$S(\neg C \vee \phi_1 \vee \phi_2) =$$

$2^{|\mathrm{var}(C) \setminus \mathrm{var}(\phi_1 \vee \phi_2)|}$ ($S(\phi_1)S(\phi_2)$ solutions of $\phi_1 \vee \phi_2$

$- S(\phi_1, a_1)S(\phi_2, a_2))$ solutions of $\phi_1 \vee \phi_2$ that agree on a

Counting models

Take a disjoint branches join tree:

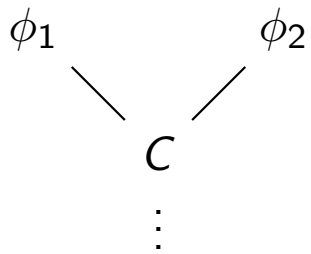


If a is the *only* assignment of $\mathrm{var}(C)$ such that $a \models \neg C$:

$$
\begin{aligned}
S(\neg C \vee \phi_1 \vee \phi_2) = & \\
2^{|\mathrm{var}(C) \setminus \mathrm{var}(\phi_1 \vee \phi_2)|} & (S(\phi_1)S(\phi_2) && \text{solutions of } \phi_1 \vee \phi_2 \\
& - S(\phi_1, a_1)S(\phi_2, a_2)) && \text{solutions of } \phi_1 \vee \phi_2 \text{ that agree on } a \\
& + 2^{|\mathrm{var}(\phi_1 \vee \phi_2) \setminus \mathrm{var}(C)|} && \text{solutions with variables of } C \text{ set to } a
\end{aligned}
$$

Counting models

Take a disjoint branches join tree:



If a is the *only* assignment of $\mathrm{var}(C)$ such that $a \models \neg C$:

$$
\begin{aligned}
S(\neg C \vee \phi_1 \vee \phi_2) = & \\
2^{|\mathrm{var}(C) \setminus \mathrm{var}(\phi_1 \vee \phi_2)|} & (S(\phi_1)S(\phi_2) && \text{solutions of } \phi_1 \vee \phi_2 \\
& - S(\phi_1, a_1)S(\phi_2, a_2)) && \text{solutions of } \phi_1 \vee \phi_2 \text{ that agree on } a \\
& + 2^{|\mathrm{var}(\phi_1 \vee \phi_2) \setminus \mathrm{var}(C)|} && \text{solutions with variables of } C \text{ set to } a
\end{aligned}
$$

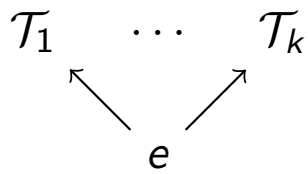
One can propagate dynamically similar equalities along the tree to compute $S(\phi)$ in polynomial time.

Computing the disjoint branches decomposition

- **Discovery algorithm**: given e, is there a disjoint branches decomposition rooted in e?

Computing the disjoint branches decomposition

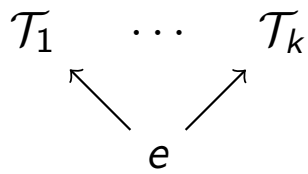
- **Discovery algorithm**: given e, is there a disjoint branches decomposition rooted in e?
- Recursive construction for each connected component of $\mathcal{H} \setminus e$



$\mathrm{var}(\mathcal{T}_1) \cap \mathrm{var}(\mathcal{T}_k) = \emptyset$: different connected components

Computing the disjoint branches decomposition

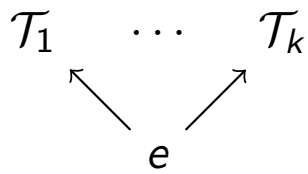
- **Discovery algorithm**: given e, is there a disjoint branches decomposition rooted in e?
- Recursive construction for each connected component of $\mathcal{H} \setminus e$



  $\mathrm{var}(\mathcal{T}_1) \cap \mathrm{var}(\mathcal{T}_k) = \emptyset$: different connected components
- Consequence of connectivity: there must exist an edge e_1 covering $e \cap \mathrm{var}(\mathcal{T}_1)$

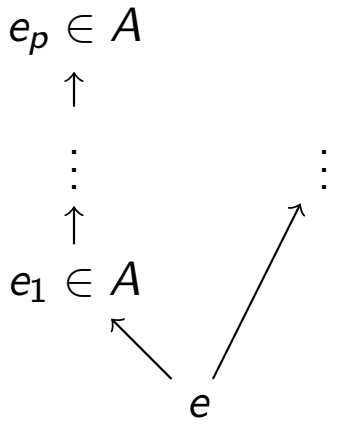
Computing the disjoint branches decomposition

- **Discovery algorithm**: given e, is there a disjoint branches decomposition rooted in e?
- Recursive construction for each connected component of $\mathcal{H} \setminus e$

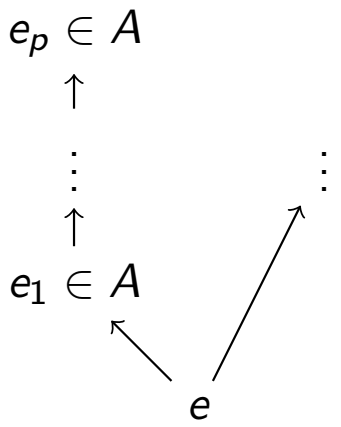


  $\mathrm{var}(\mathcal{T}_1) \cap \mathrm{var}(\mathcal{T}_k) = \emptyset$: different connected components
- Consequence of connectivity: there must exist an edge e_1 covering $e \cap \mathrm{var}(\mathcal{T}_1)$
- The set A of such edges is the set of all canditates to be the root of $\mathcal{T}_1$

- Candidates in A must lie along a path (by connectivity):

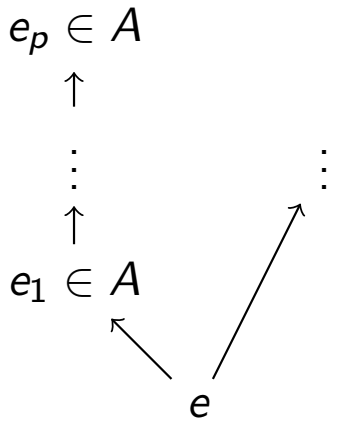


- Candidates in A must lie along a path (by connectivity):



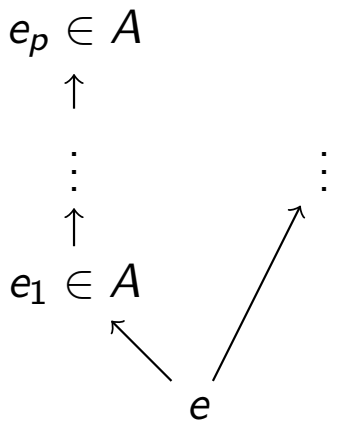
- Data structure, PQ-trees, introduced by Booth and Luecker, represents all such paths. Computable in linear time

- Candidates in A must lie along a path (by connectivity):



- Data structure, PQ-trees, introduced by Booth and Luecker, represents all such paths. Computable in linear time
- Additionnal conditions that our path must respect: introduction of PQF-trees

- Candidates in A must lie along a path (by connectivity):



- Data structure, PQ-trees, introduced by Booth and Luecker, represents all such paths. Computable in linear time
- Additionnal conditions that our path must respect: introduction of PQF-trees
- Recursive call on each connected component of $\mathcal{H} \setminus A$

Conclusion

We have presented:

- A classical and simple dynamic programming algorithm for the counting problem

Conclusion

We have presented:

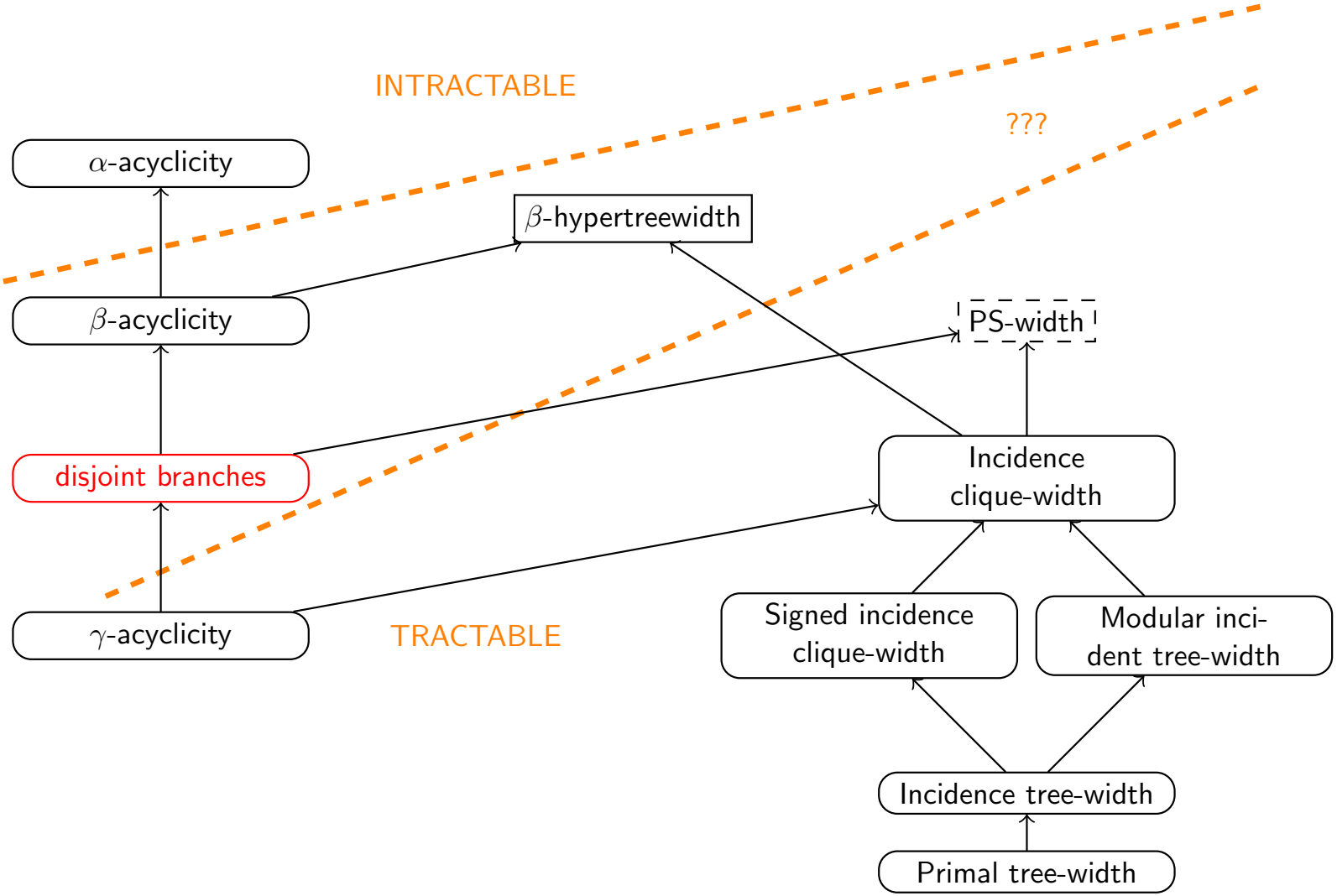
- A classical and simple dynamic programming algorithm for the counting problem
- But a rather more complicate algorithm for discovery problem. Can it be simplified?

Conclusion

We have presented:

- A classical and simple dynamic programming algorithm for the counting problem
- But a rather more complicate algorithm for discovery problem. Can it be simplified?
- Combining both algorithm leads to a polynomial time algorithm for disjoint branches formulas

The whole picture



The whole picture